An abstract background graphic consisting of several light blue, 3D rectangular blocks arranged in a staggered, ascending pattern. A thin, light blue line with sharp turns, resembling a stylized staircase or a data path, winds through the blocks. On the left and right sides of the slide, there are dark blue, rounded rectangular shapes that partially frame the central content.

DATA WRANGLING & VISUAL EXPLORATION

Week 2

TABLE OF CONTENTS

01 The Role of Data Wrangling in Analytics

- What is data wrangling?
 - Why it's critical before modelling or visualisation
 - Real-world examples
-

02 Core Transformation Techniques

- with Pandas
-

03 Basics of Feature Engineering

- Encoding Features
 - Scaling & Normalization
 - Creating New Features
 - Feature Selection
-

04 Advance Pandas Techniques

TABLE OF CONTENTS

05 Introduction to SQL & Relational Thinking

- Why SQL Matters for Analysts?
 - SQL Basics – SELECT, WHERE, GROUP BY, JOIN
 - Relational Concepts
 - Visual example
-

06 Visual Exploration & Storytelling

- Principles of Effective Visualisation
 - Bad vs. Good Visuals
 - Choosing the Right Chart
-

07 Matplotlib Essentials

08 Seaborn Deep Dive



THE ROLE OF DATA WRANGLING IN ANALYTICS

~WHAT? WHY? REAL-WORLD EXAMPLES



WHAT IS DATA WRANGLING?

- The process of cleaning, transforming, and organising raw data into a usable format
- Also known as **data munging**
- Prepares data for analysis, visualization, or machine learning
- Involves handling **missing values, fixing errors, renaming columns, and changing data types**
- Merging and reshaping data from multiple sources into a consistent structure
- Ensures data is **accurate, complete, and analysis-ready**

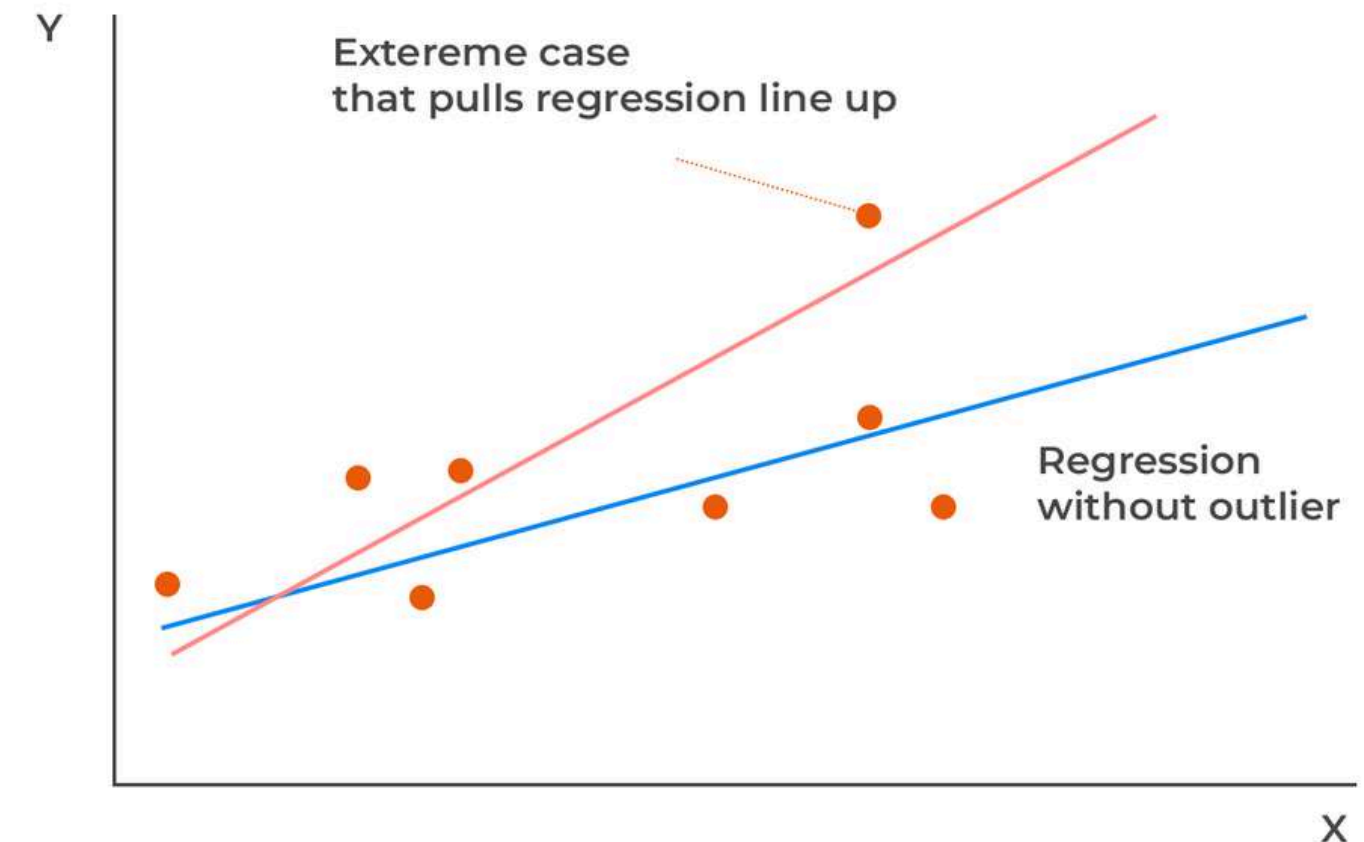


WHY IT'S CRITICAL BEFORE MODELLING OR VISUALISATION?

01. Models Rely on Clean Inputs

Missing values, outliers, or inconsistent categories can skew predictions

- Example 1 : A linear model on non-normalized revenue gives biased weights
- Example 2: A sudden market crash not accounted for by a model can result in erroneous future price forecasts, affecting investment strategies.
- Example 3: A model predicting housing prices fails to learn because 20% of sqft_living values are missing.



WHY IT'S CRITICAL BEFORE MODELLING OR VISUALISATION?

02. Visuals Can Be Misleading Without Preprocessing

- Dirty data → false trends, incorrect group sizes, distorted charts
- Charts lie when the data isn't properly cleaned or grouped.
- Example 1: Region = "West" vs "west" shows up as two bars
- Example 2: Months with no sales data appear as drops → misleading trend lines

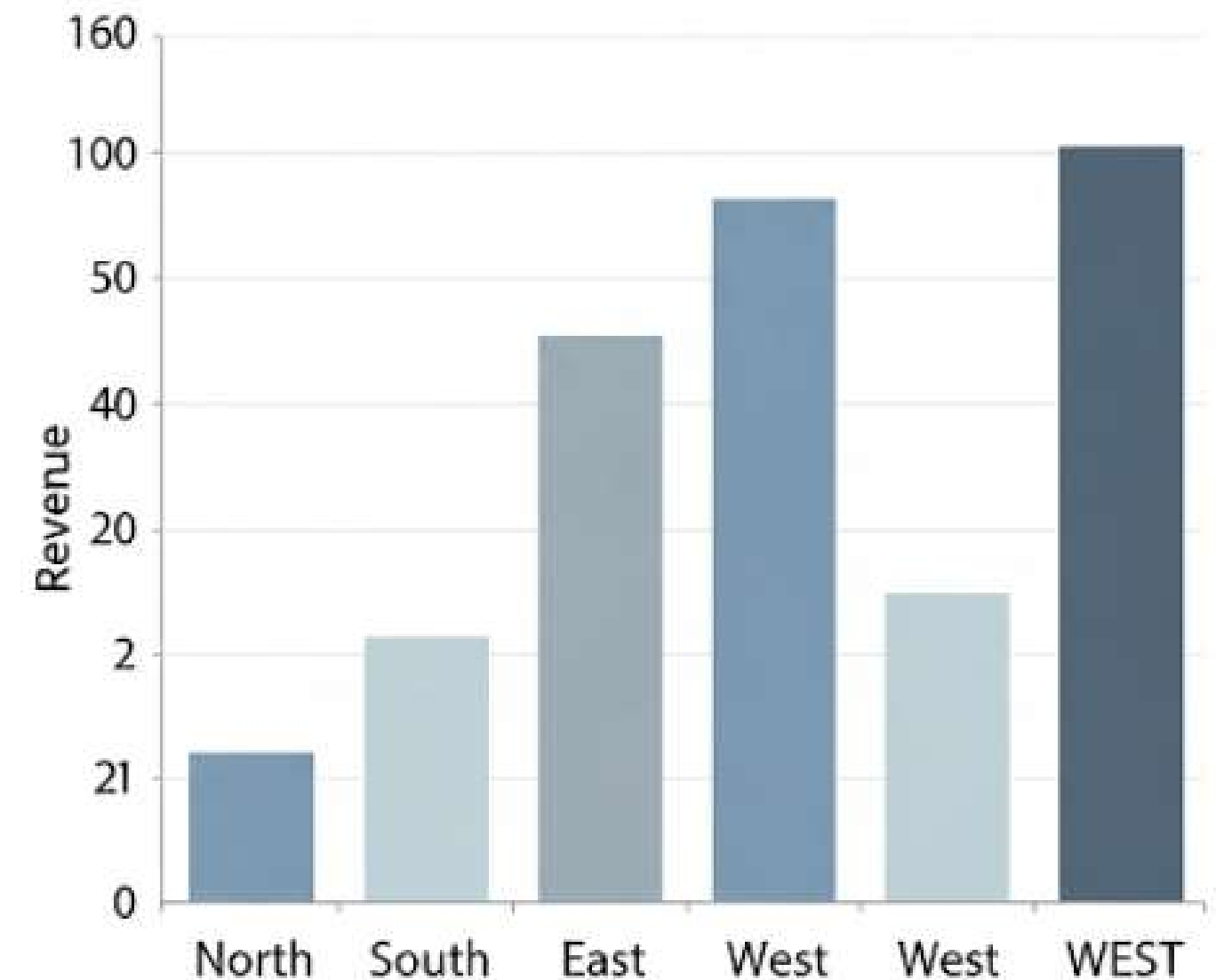
WHY IT'S CRITICAL BEFORE MODELLING OR VISUALISATION?

03. Feature Engineering Depends on Wrangling

- You can't extract insightful features without preparing the data first.
- Example 1: `signup_date` in string format prevents weekday, month, tenure extraction
- Example 2: Email field needs parsing to extract domain (e.g., gmail.com, outlook.com)
- Example 3: `duration = end_time - start_time` is only possible if times are parsed to datetime

Feature engineering, the most crucial subject in Data Science

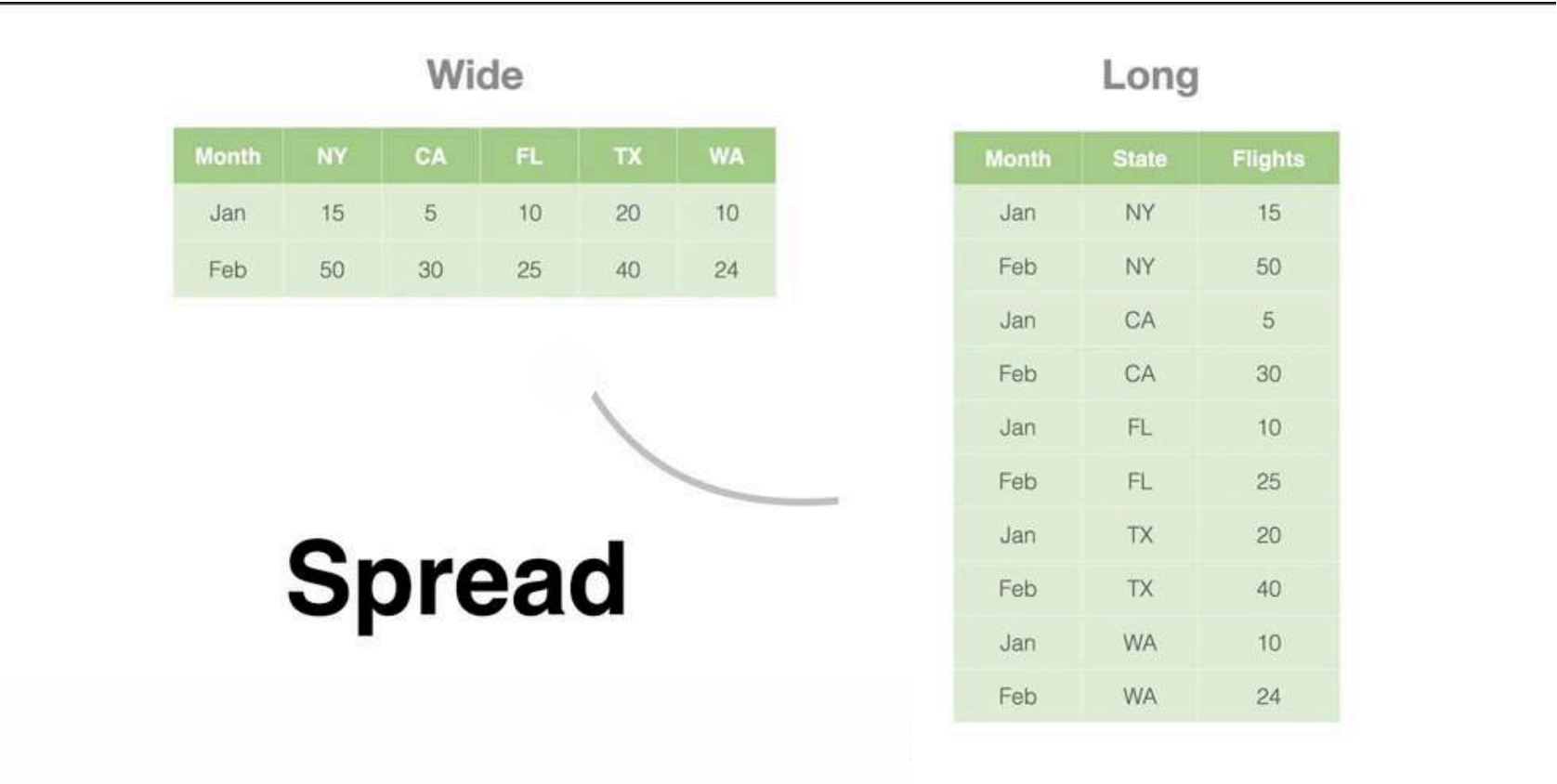
~ <https://www.linkedin.com/pulse/feature-engineering-most-crucial-subject-data-science-%C3%B3scar-azeem/>



WHY IT'S CRITICAL BEFORE MODELLING OR VISUALISATION?

04. Merging & Reshaping Enables Holistic Views

- Analytics requires combining multiple sources and reshaping them into usable structures.
- Example 1: Transform wide-format survey into long/normalized layout for analysis
- Example 2: Pivot product sales by region
- Example 3: Merge customer info + transaction logs → Show churn vs. spending across regions



WHY IT'S CRITICAL BEFORE MODELLING OR VISUALISATION?

05. Prevents Time-Wasting Later

- Clean early or debug endlessly later.
- Example 1: Nulls or unexpected labels break filters/charts in Power BI/Tableau
- Example 2: Months with no sales data appear as drops → misleading trend lines
- Example 3: Stakeholders misinterpret insights because data wasn't standardized
- Example 4: Left joins produce more rows than expected due to hidden whitespace or data type mismatches



REAL WORLD EXAMPLE

Wrangling Steps Applied:

- Standardized date formats
- Filled missing values
- Removed duplicates
- Normalized region codes
- Parsed revenue to float
- Created sentiment score from comments

❌

date	category	price	image_url	rating
04/01/2023 12:00 PM	Electronics	199.99	http://imageexample.com/pic.jpg	4
2023-04-02	Books	twenty	https://example.com/book.jpg	5
2023/04/03	Clothing	49.99	https://example.com/shirt.png	4 stars
03-04-2023	Fod	9.99	https://example.com/food.png	10
2023-04-05T08:00:00	Furniture	20€	https://example.com/chair.jpg	5

Non-uniform Date Format

currency symbol

extra whitespace ("5 ")

➡

✅

date	category	price	image_url	rating
2023-04-01 12:00:00+00:00	Electronics	199.99	https://imageexample.com/pic.jpg	4
2023-04-02 00:00:00+00:00	Books	20.0	https://example.com/book.jpg	5
2023-04-03 00:00:00+00:00	Clothing	49.99	https://example.com/shirt.png	4
2023-04-04 00:00:00+00:00	Food	9.99	https://example.com/food.png	5
2023-04-05 08:00:00+00:00	Furniture	20.0	https://example.com/chair.jpg	5

Uniform Date Format

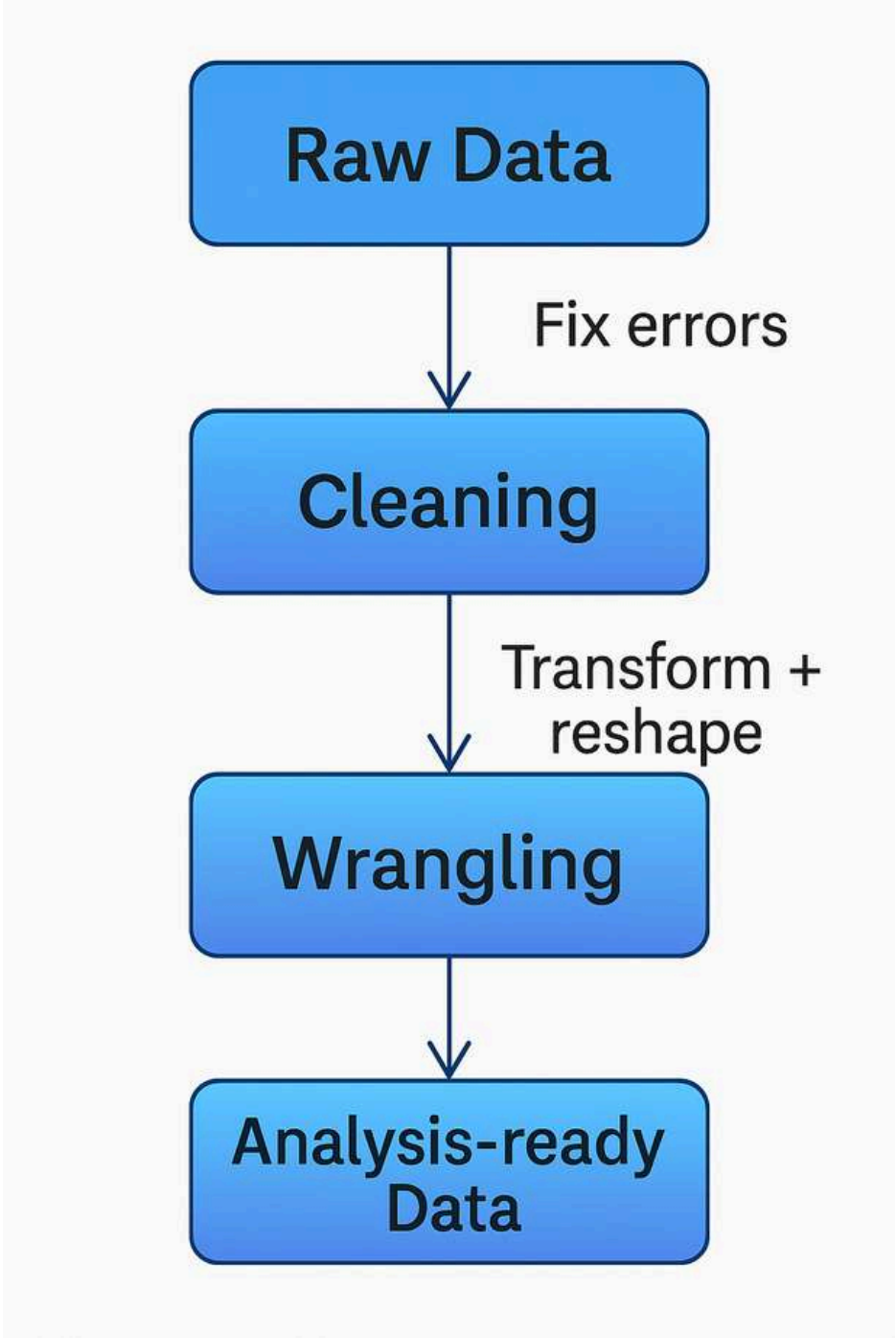
No extra whitespace Number

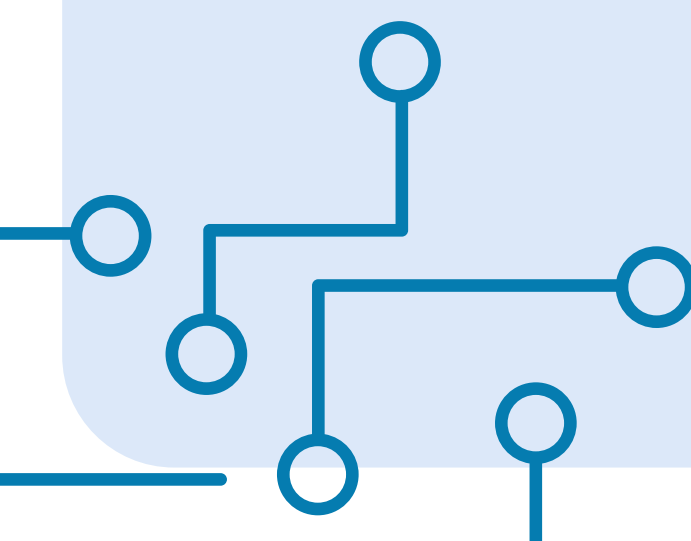
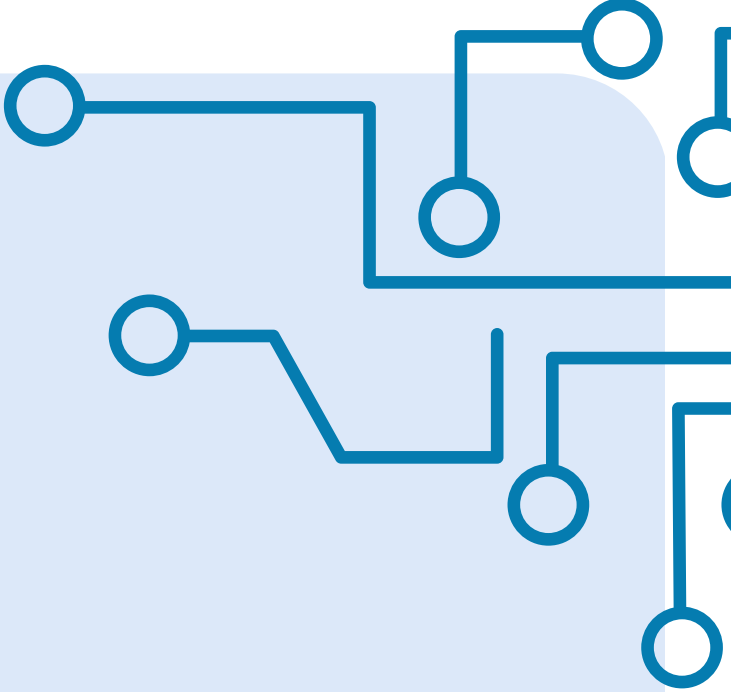
Handwritten annotations in the original image:

- Top Table (❌):**
 - Row 2: "twenty" is circled in red with "text" written next to it.
 - Row 2: "https://" is circled in red with "URL scheme" written next to it.
 - Row 3: "4 stars" is circled in red with "mix data types" written next to it.
 - Row 4: "Fod" is circled in red with "Typo" written next to it.
 - Row 4: "10" is circled in red with "out-of-range value" written next to it.
 - Row 5: "20€" is circled in red with "currency symbol" written below it.
 - Row 5: "5" is circled in red with "extra whitespace ('5 ')" written next to it.
- Bottom Table (✅):**
 - Row 1: "https://" is circled in green.
 - Row 2: "20.0" is circled in green.
 - Row 3: "4" is circled in green.
 - Row 4: "Food" is circled in green.
 - Row 4: "5" is circled in green with "Set to max" written next to it.
 - Row 5: "20.0" is circled in green.
 - Row 5: "5" is circled in green.

DIFFERENCE BETWEEN DATA CLEANING AND WRANGLING

DATA CLEANING	DATA WRANGLING
Fixes errors and inconsistencies in data	Prepares and reshapes data for analysis
Removes duplicates, handles missing values	Merges, pivots, filters, and engineers fields
Deals with formatting, typos, noise	Transforms structure to match analysis goals
Focus: <i>Data quality</i>	Focus: <i>Data usability</i>
Usually a subset of data wrangling	Broader: includes cleaning + reshaping





CORE TRANSFORMATION TECHNIQUES

~WITH PANDAS

TRANSFORMATION TECHNIQUES



Filtering Rows

Filtering allows you to extract only the rows that match certain conditions — the most common first step in exploratory analysis.

(e.g., “only show high-value customers in the West”)



Reshaping

Reshaping transforms the structure of your dataset — wide ↔ long, pivoted ↔ flat — for analysis or plotting.

You often need to reshape data for time series analysis, grouped aggregation, or creating input tables for dashboards.



Combining Data

Combining allows you to join, append, or stack datasets together, often from different sources (e.g., customers and orders).

Key Methods:
`merge()`, `join()`, `concat()`



Enrichment (Lookup Mapping)

Enrichment adds extra meaning to raw IDs or codes — typically by using a lookup table or mapping dictionary.

Enriched data is easier to analyze and present. “W” → “West Region” gives context that raw codes can’t.



Normalizing

Normalization rescales numeric values to a common range — typically 0–1 — so they’re easier to compare or feed into models.

Unnormalized features can distort models or charts. Scaling aligns features like sales, quantity, and rating for fair comparison.

Filtering Rows (Boolean Masks & Multiple Conditions)

Original DataFrame:

	Customer_ID	Region	Product	Month	Sales
0	1	West	Smartphone	Jan	1200
1	2	East	Headphones	Jan	850
2	3	West	Laptop	Feb	400
3	4	North	Smartphone	Feb	950

Boolean Masks: A method of filtering data using True/False values to select specific rows or elements.

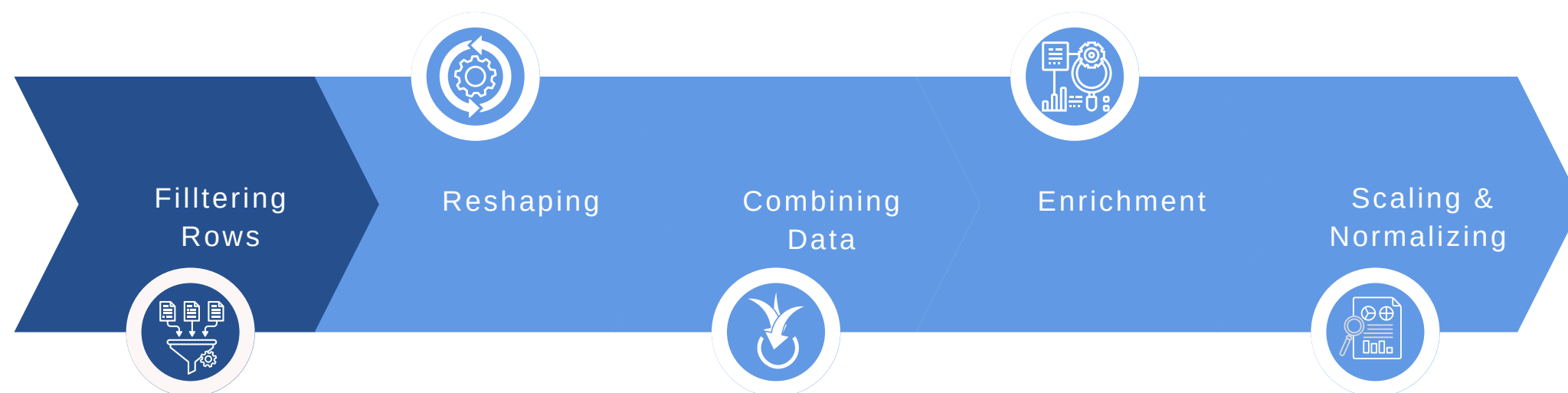
```
1 # Filter where sales > 900
2 df[df["Sales"] > 900]
```

	Customer_ID	Region	Product	Month	Sales
0	1	West	Smartphone	Jan	1200
3	4	North	Smartphone	Feb	950

Multiple Conditions: Combining two or more logical expressions to filter data based on complex criteria.

```
1 # Filter where region is West and sales > 900
2 df[(df["Region"] == "West") & (df["Sales"] > 900)]
```

	Customer_ID	Region	Product	Month	Sales
0	1	West	Smartphone	Jan	1200



Reshaping: pivot, melt, stack, unstack

Original DataFrame:

	Customer_ID	Region	Product	Month	Sales
0	1	West	Smartphone	Jan	1200
1	2	East	Headphones	Jan	850
2	3	West	Laptop	Feb	400
3	4	North	Smartphone	Feb	950

pivot: Reshapes data by turning unique column values into new columns, typically used for creating summary tables.

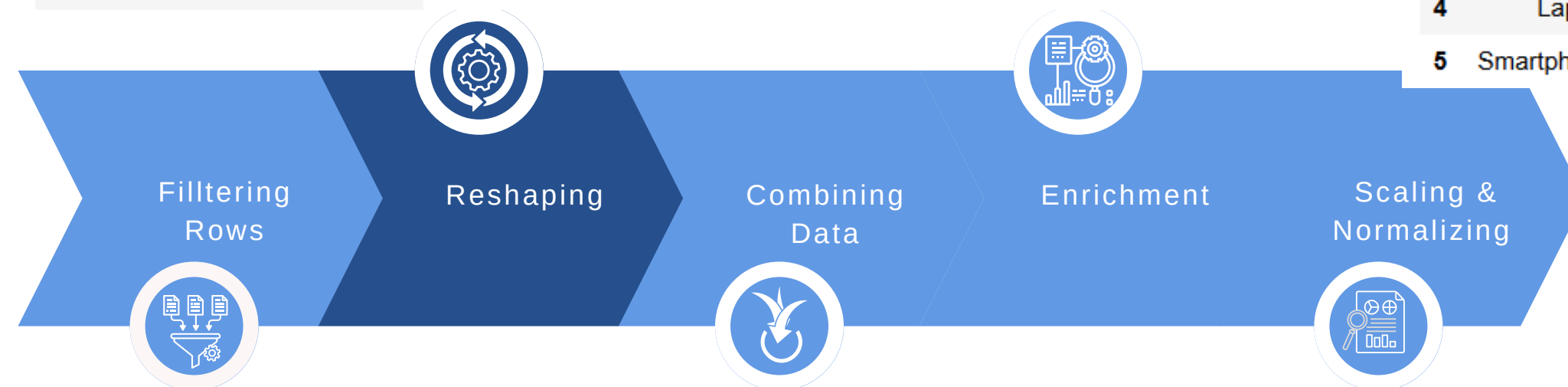
```
1 # Pivot: Show sales by product across months
2 df_pivot = df.pivot(index="Product", columns="Month", values="Sales")
3 display(df_pivot)
```

Month	Feb	Jan
Product		
Headphones	NaN	850.0
Laptop	400.0	NaN
Smartphone	950.0	1200.0

melt: Unpivots a wide DataFrame into a long format by turning columns into row values.

```
1 # Melt: Unpivot the pivoted table back
2 df_melted = pd.melt(df_pivot.reset_index(), id_vars="Product", value_name="Sales")
3 df_melted
```

	Product	Month	Sales
0	Headphones	Feb	NaN
1	Laptop	Feb	400.0
2	Smartphone	Feb	950.0
3	Headphones	Jan	850.0
4	Laptop	Jan	NaN
5	Smartphone	Jan	1200.0



Reshaping: pivot, melt, stack, unstack

Original DataFrame:

	Customer_ID	Region	Product	Month	Sales
0	1	West	Smartphone	Jan	1200
1	2	East	Headphones	Jan	850
2	3	West	Laptop	Feb	400
3	4	North	Smartphone	Feb	950

stack: Moves column labels into the row index, resulting in a tall (longer) format.

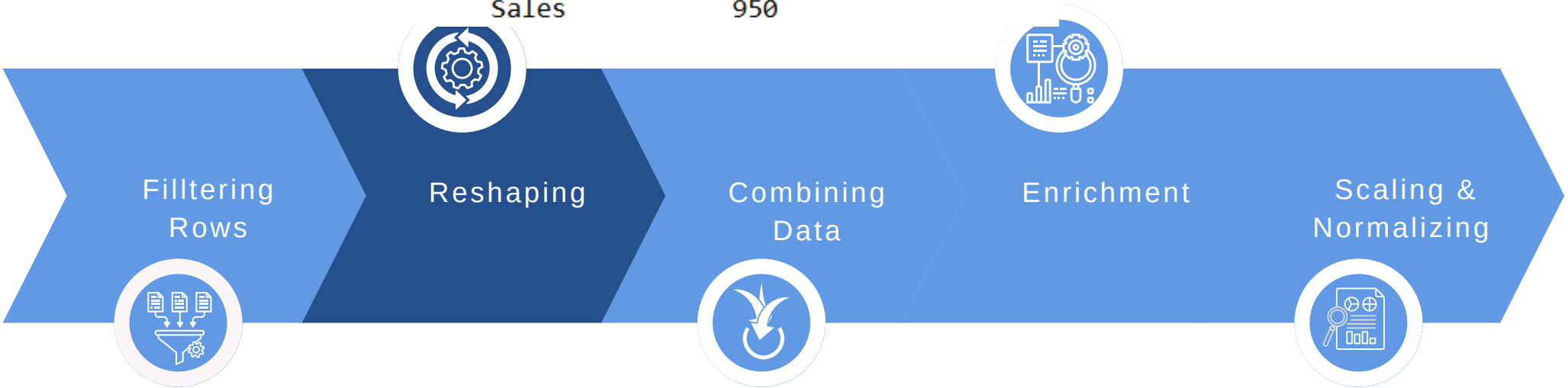
```
1 # Stack
2 stacked = df.set_index(["Region", "Product"]).stack()
3 stacked
```

Region	Product	Customer_ID	Month	Sales
West	Smartphone	1	Jan	1200
East	Headphones	2	Jan	850
West	Laptop	3	Feb	400
North	Smartphone	4	Feb	950

unstack: Moves index levels into columns, creating a wider format from a hierarchical index.

```
1 # Unstack
2 unstacked = df.set_index(["Region", "Product"]).unstack()
3 unstacked
```

	Customer_ID			Month			Sales		
Product	Headphones	Laptop	Smartphone	Headphones	Laptop	Smartphone	Headphones	Laptop	Smartphone
Region									
East	2.0	NaN	NaN	Jan	NaN	NaN	850.0	NaN	NaN
North	NaN	NaN	4.0	NaN	NaN	Feb	NaN	NaN	950.0
West	NaN	3.0	1.0	NaN	Feb	Jan	NaN	400.0	1200.0



Combining Data: merge, concat

Original DataFrame:

	Customer_ID	Region	Product	Month	Sales
0	1	West	Smartphone	Jan	1200
1	2	East	Headphones	Jan	850
2	3	West	Laptop	Feb	400
3	4	North	Smartphone	Feb	950

Customers DataFrame:

	Customer_ID	Customer_Name
0	1	Alice
1	2	Bob
2	3	Cathy
3	4	Dan
4	5	Charlie

`merge()`: Combines two DataFrames based on common columns or index keys, similar to SQL joins.

```
1 # Merge with customer names
2 df_merged = pd.merge(df, customers, on="Customer_ID", how="left")
3 df_merged
```

	Customer_ID	Region	Product	Month	Sales	Customer_Name
0	1	West	Smartphone	Jan	1200	Alice
1	2	East	Headphones	Jan	850	Bob
2	3	West	Laptop	Feb	400	Cathy
3	4	North	Smartphone	Feb	950	Dan



Combining Data: merge, concat

concat(): Joins multiple DataFrames along rows or columns, without matching keys unless specified.

```
1 # Creating quarters [using Boolean Mask]
2 df_q1 = df[df["Month"] == "Jan"]
3 df_q2 = df[df["Month"] == "Feb"]
```

Jan Quarter:

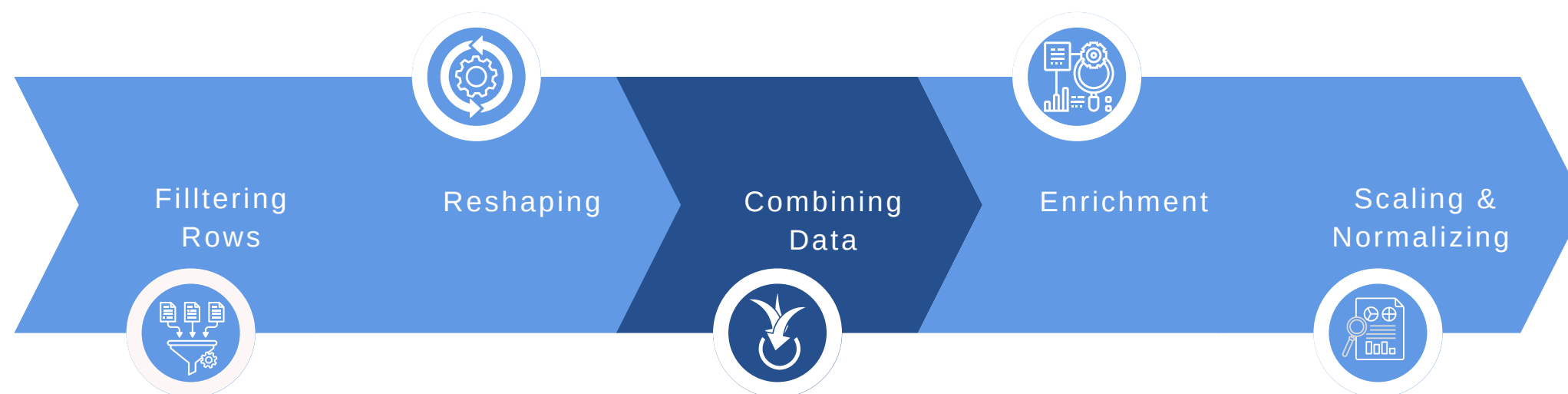
	Customer_ID	Region	Product	Month	Sales
0	1	West	Smartphone	Jan	1200
1	2	East	Headphones	Jan	850

Feb Quarter:

	Customer_ID	Region	Product	Month	Sales
2	3	West	Laptop	Feb	400
3	4	North	Smartphone	Feb	950

```
1 # Concatenate quarters
2 df_combined = pd.concat([df_q1, df_q2])
3 df_combined
```

	Customer_ID	Region	Product	Month	Sales
0	1	West	Smartphone	Jan	1200
1	2	East	Headphones	Jan	850
2	3	West	Laptop	Feb	400
3	4	North	Smartphone	Feb	950



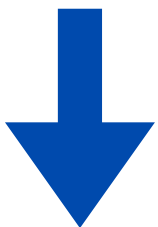
Enrichment: Lookup / Map

Add new information to a DataFrame by mapping existing values to a reference table or dictionary.

```
1 # Simulate region lookup dictionary
2 region_lookup = {
3     "West": "Western Region",
4     "East": "Eastern Region",
5     "North": "Northern Region"
6 }
7
8 # Enrich region description
9 df["region name"] = df["Region"].map(region_lookup)
10 # Rename columns
11 df.rename(columns={"Sales": "Sales (£)"}, inplace=True)
12 # lowercase all
13 df.columns = [col.lower() for col in df.columns]
14 # remove spaces
15 df.columns = [col.replace(" ", "_") for col in df.columns]
```

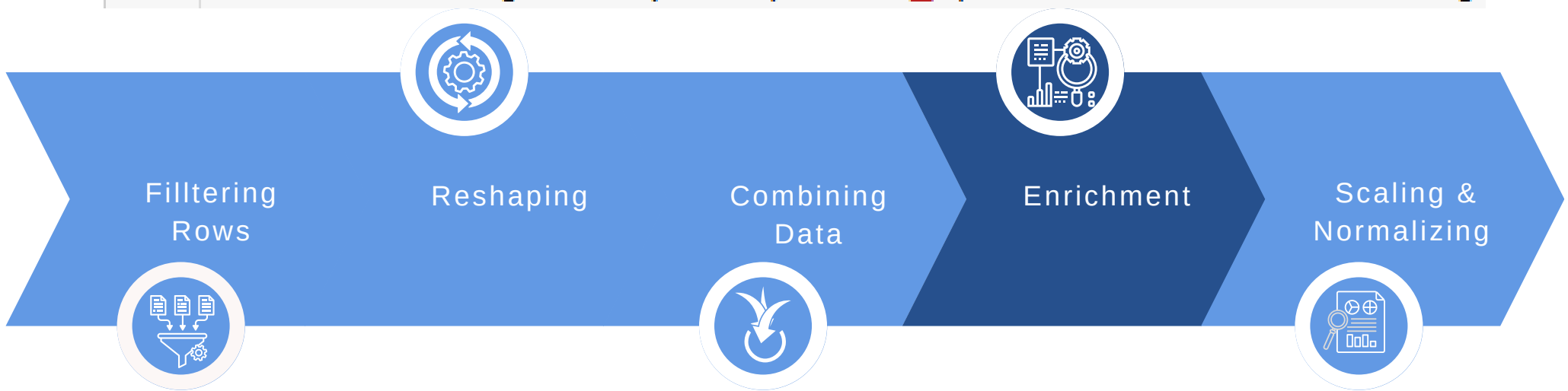
Original DataFrame:

	Customer_ID	Region	Product	Month	Sales
0	1	West	Smartphone	Jan	1200
1	2	East	Headphones	Jan	850
2	3	West	Laptop	Feb	400
3	4	North	Smartphone	Feb	950



Enriched DataFrame:

	customer_id	region	product	month	sales_(£)	region_name
0	1	West	Smartphone	Jan	1200	Western Region
1	2	East	Headphones	Jan	850	Eastern Region
2	3	West	Laptop	Feb	400	Western Region
3	4	North	Smartphone	Feb	950	Northern Region



Normalization: MinMaxScaler or Manual

MinMax Scaler: Automatically scales features to a fixed range (usually 0 to 1) based on their minimum and maximum values.

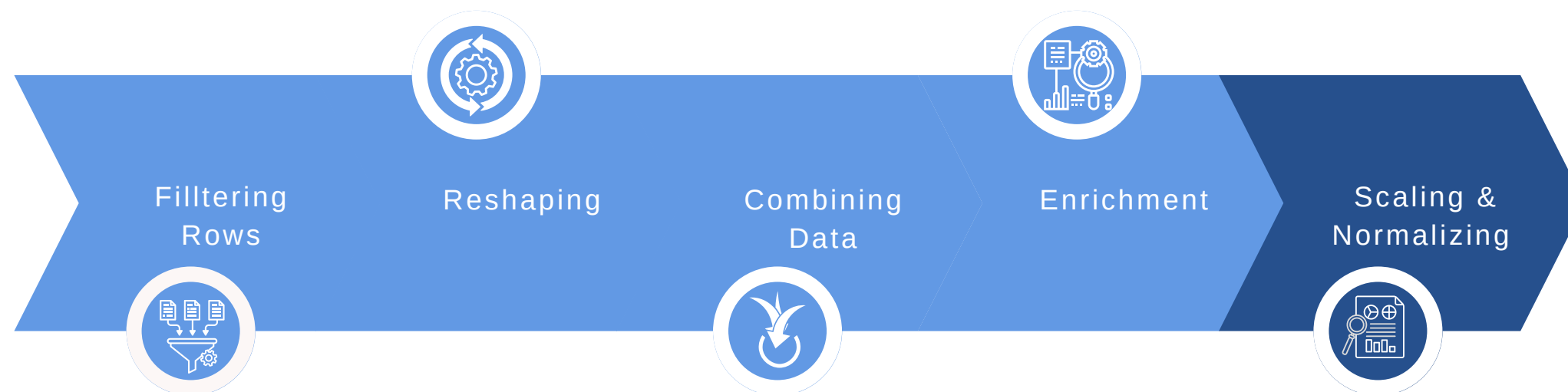
```
1 from sklearn.preprocessing import MinMaxScaler
2
3 # MinMax scaling
4 scaler = MinMaxScaler()
5 df["sales_scaled"] = scaler.fit_transform(df[["sales_(£)"]])
6 df
```

	customer_id	region	product	month	sales_(£)	region_name	sales_scaled
0	1	West	Smartphone	Jan	1200	Western Region	1.0000
1	2	East	Headphones	Jan	850	Eastern Region	0.5625
2	3	West	Laptop	Feb	400	Western Region	0.0000
3	4	North	Smartphone	Feb	950	Northern Region	0.6875

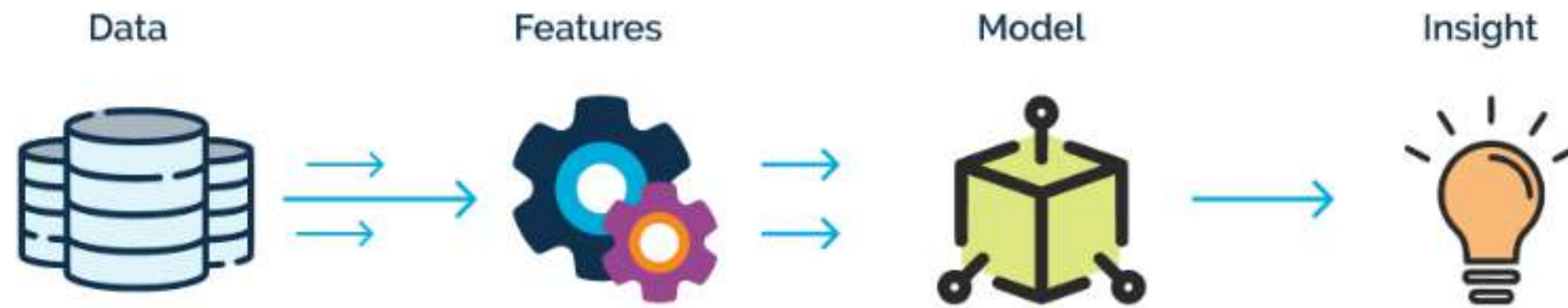
Manual Scaling: Custom scaling of values using your own formula, such as $(x - \min) / (\max - \min)$, done without a library.

```
1 # Manual scaling
2 df["sales_scaled_manual"] = (
3     (df["sales_(£)"] - df["sales_(£)"].min()) /
4     (df["sales_(£)"].max() - df["sales_(£)"].min())
5 )
6 df
```

	customer_id	region	product	month	sales_(£)	region_name	sales_scaled	sales_scaled_manual
0	1	West	Smartphone	Jan	1200	Western Region	1.0000	1.0000
1	2	East	Headphones	Jan	850	Eastern Region	0.5625	0.5625
2	3	West	Laptop	Feb	400	Western Region	0.0000	0.0000
3	4	North	Smartphone	Feb	950	Northern Region	0.6875	0.6875



Feature Engineering



FEATURE ENGINEERING

DEFINITION

Feature engineering is the process of transforming raw data into meaningful features that improve the performance of machine learning models.

Key Steps

- Handling Missing Values: Imputation or removal
- Encoding Categorical Variables: One-hot, label encoding
- Scaling and Normalisation: StandardScaler, MinMaxScaler
- Creating New Features: Ratios, time-based, polynomial features
- Feature Selection: Removing irrelevant or redundant features

Why It Matters

Good features = Better model accuracy, faster training, and improved generalisation



ENCODING CATEGORICAL VARIABLES

Most ML models can't process text — we need numeric codes.



One-hot encoding

- for nominal categories
- Converts each category into a new binary column
- Example: "region" → "region_West", "region_East", etc.



Label encoding

- for binary or ordinal categories
- Assigns each category a unique integer
- Example: "Low", "Medium", "High" → 0, 1, 2

Feature Engineering Essentials

1

Handling Missing Values

2

Encoding Categorical Variables

3

Scaling & Normalisation

4

Creating New Features

5

Feature Selection

ENCODING CATEGORICAL VARIABLES

Original DataFrame

	customer_id	gender	region	signup_date	age	income	items_purchased	total_spend
0	1001	Male	West	2023-01-10	25	45000	5	200.0
1	1002	Female	East	2022-11-05	34	52000	3	180.0
2	1003	Female	West	2023-03-15	39	61000	6	300.0
3	1004	Female	North	2023-02-01	45	49000	2	230.0
4	1005	Male	East	2023-01-10	52	51000	4	250.0

One-hot encoding

```
1 # Convert categorical column to one-hot
2 df_encoded = pd.get_dummies(df, columns=["region"], drop_first=True)
3
4 df_encoded
```

	customer_id	gender	signup_date	age	income	items_purchased	total_spend	region_North	region_West
0	1001	Male	2023-01-10	25	45000	5	200.0	False	True
1	1002	Female	2022-11-05	34	52000	3	180.0	False	False
2	1003	Female	2023-03-15	39	61000	6	300.0	False	True
3	1004	Female	2023-02-01	45	49000	2	230.0	True	False
4	1005	Male	2023-01-10	52	51000	4	250.0	False	False

Label encoding

```
1 from sklearn.preprocessing import LabelEncoder
2
3 le = LabelEncoder()
4 df["gender_encoded"] = le.fit_transform(df["gender"]) # e.g., Male → 1, Female → 0
5
6 df
```

	customer_id	gender	region	signup_date	age	income	items_purchased	total_spend	gender_encoded
0	1001	Male	West	2023-01-10	25	45000	5	200.0	1
1	1002	Female	East	2022-11-05	34	52000	3	180.0	0
2	1003	Female	West	2023-03-15	39	61000	6	300.0	0
3	1004	Female	North	2023-02-01	45	49000	2	230.0	0
4	1005	Male	East	2023-01-10	52	51000	4	250.0	1

Normalization: MinMaxScaler or Manual

MinMax Scaler: Automatically scales features to a fixed range (usually 0 to 1) based on their minimum and maximum values.

```
1 from sklearn.preprocessing import MinMaxScaler
2
3 # MinMax scaling
4 scaler = MinMaxScaler()
5 df["sales_scaled"] = scaler.fit_transform(df[["sales_(£)"]])
6 df
```

	customer_id	region	product	month	sales_(£)	region_name	sales_scaled
0	1	West	Smartphone	Jan	1200	Western Region	1.0000
1	2	East	Headphones	Jan	850	Eastern Region	0.5625
2	3	West	Laptop	Feb	400	Western Region	0.0000
3	4	North	Smartphone	Feb	950	Northern Region	0.6875

Manual Scaling: Custom scaling of values using your own formula, such as $(x - \min) / (\max - \min)$, done without a library.

```
1 # Manual scaling
2 df["sales_scaled_manual"] = (
3     (df["sales_(£)"] - df["sales_(£)"].min()) /
4     (df["sales_(£)"].max() - df["sales_(£)"].min())
5 )
6 df
```

	customer_id	region	product	month	sales_(£)	region_name	sales_scaled	sales_scaled_manual
0	1	West	Smartphone	Jan	1200	Western Region	1.0000	1.0000
1	2	East	Headphones	Jan	850	Eastern Region	0.5625	0.5625
2	3	West	Laptop	Feb	400	Western Region	0.0000	0.0000
3	4	North	Smartphone	Feb	950	Northern Region	0.6875	0.6875

* Discussed previously

CREATING NEW FEATURES

Well-designed features improve model accuracy and interpretability.

```
1 # Create ratio feature: spend per item
2 df["spend_per_item"] = df["total_spend"] / df["items_purchased"]
3
```

	customer_id	gender	region	signup_date	age	income	items_purchased	total_spend	gender_encoded	spend_per_item	signup_month	signup_weekday	tenure_days
0	1001	Male	West	2023-01-10	25	45000	5	200.0	1	40.0	1	Tuesday	905
1	1002	Female	East	2022-11-05	34	52000	3	180.0	0	60.0	11	Saturday	971
2	1003	Female	West	2023-03-15	39	61000	6	300.0	0	50.0	3	Wednesday	841
3	1004	Female	North	2023-02-01	45	49000	2	230.0	0	115.0	2	Wednesday	883
4	1005	Male	East	2023-01-10	52	51000	4	250.0	1	62.5	1	Tuesday	905

```
1 # Convert to datetime
2 df["signup_date"] = pd.to_datetime(df["signup_date"])
3
4 # Extract useful features
5 df["signup_month"] = df["signup_date"].dt.month
6 df["signup_weekday"] = df["signup_date"].dt.day_name()
7 df["tenure_days"] = (pd.to_datetime("today") - df["signup_date"]).dt.days
```

FEATURE SELECTION

1. Reduces Dimensionality

- Fewer features mean smaller datasets, which speed up training and prediction time.
- Especially important for high-dimensional data like text, images, or genomics.

2. Speeds Up Computation

- Less memory is used.
- Fewer calculations during model training = faster performance.

3. Improves Model Interpretability

- Simpler models are easier to explain and debug.
- Stakeholders can better understand what drives predictions.

4. Enhances Data Quality

- Focuses attention on the most informative variables.
- Easier to clean and validate fewer features.

- Remove metadata or irrelevant columns

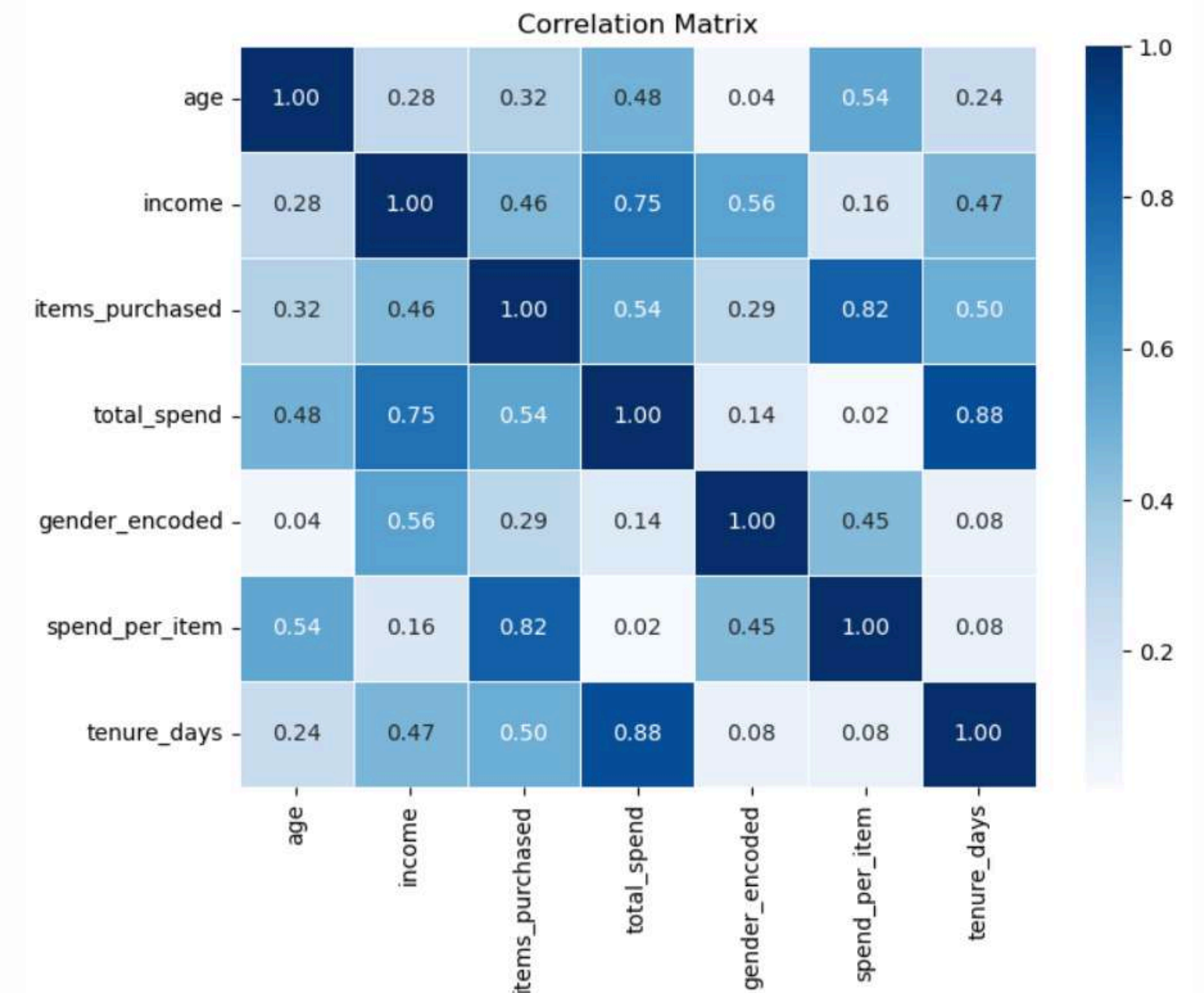
1	# Drop unused or ID columns
2	df.drop(columns=["customer_id", "signup_date"], inplace=True)
3	
4	df

	gender	region	age	income	items_purchased	total_spend	gender_encoded	spend_per_item	signup_month	signup_weekday	tenure_days
0	Male	West	25	45000	5	200.0	1	40.0	1	Tuesday	905
1	Female	East	34	52000	3	180.0	0	60.0	11	Saturday	971
2	Female	West	39	61000	6	300.0	0	50.0	3	Wednesday	841
3	Female	North	45	49000	2	230.0	0	115.0	2	Wednesday	883
4	Male	East	52	51000	4	250.0	1	62.5	1	Tuesday	905

FEATURE SELECTION

- Filter low-variance or highly correlated features

```
1 import seaborn as sns
2 import matplotlib.pyplot as plt
3 import numpy as np
4
5 # Step 1: Select only numeric columns
6 df_numeric = df.select_dtypes(include=[np.number])
7
8 # Step 2: Compute correlation matrix
9 corr_matrix = df_numeric.corr().abs()
10
11 # Step 3: Plot heatmap
12 plt.figure(figsize=(8, 6))
13 sns.heatmap(corr_matrix, annot=True, cmap='Blues', fmt=".2f", linewidths=0.5)
14 plt.title("Correlation Matrix")
15 plt.show()
16
17 # Step 4: Identify and drop highly correlated features
18 upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).astype(bool))
19 to_drop = [col for col in upper.columns if any(upper[col] > 0.9)]
20
21 print("Dropping highly correlated columns:", to_drop)
22
23 # Drop them from the original df
24 df.drop(columns=to_drop, inplace=True)
```



Advanced Pandas Techniques

Powerful tool for data summarisation, transformation, and cleaning using Pandas.

Intermediate to advanced methods for real-world data analysis.



Order Summary, this DataFrame will be used for codes:

	customer_id	customer_segment	region	product	sales	revenue	cost	total_spent	signup_month
0	1	Retail	North	A	1200	1500	1000	5000	1
1	2	Corporate	South	B	850	1000	700	3400	3
2	3	Home Office	East	C	560	700	400	2000	2
3	4	Retail	West	A	1430	1600	1100	6000	1
4	5	Corporate	North	C	760	900	600	2500	4
5	6	Retail	East	B	980	1150	800	4000	3
6	7	Home Office	South	A	3050	3200	2800	8000	2
7	8	Corporate	North	C	1300	1400	1100	5500	1
8	9	Retail	West	B	210	300	250	1300	4
9	10	Corporate	South	A	1670	1800	1300	6700	3

- customer_id: Unique ID per customer
- customer_segment: Customer grouping
- region: Geographical region
- product: Product involved
- sales, revenue, cost: Financial metrics of a single event
- total_spent: Appears cumulative or summarised per customer
- signup_month: When the customer signed up

groupby() and .agg() for Summary Tables

Group data by one or more columns and summarise with aggregation.

Why it is used?

- Quickly summarise large datasets.
- Combine multiple aggregations at once.

Original DataFrame

	customer_id	customer_segment	region	product	sales	revenue	cost	total_spent	signup_month
0	1	Retail	North	A	1200	1500	1000	5000	1
1	2	Corporate	South	B	850	1000	700	3400	3
2	3	Home Office	East	C	560	700	400	2000	2
3	4	Retail	West	A	1430	1600	1100	6000	1
4	5	Corporate	North	C	760	900	600	2500	4
5	6	Retail	East	B	980	1150	800	4000	3
6	7	Home Office	South	A	3050	3200	2800	8000	2
7	8	Corporate	North	C	1300	1400	1100	5500	1
8	9	Retail	West	B	210	300	250	1300	4
9	10	Corporate	South	A	1670	1800	1300	6700	3

Code

```
1 # Example: Average spend by customer segment
2 summary = df.groupby('customer_segment')['total_spent'].agg(['mean', 'max', 'min', 'count'])
3 print(summary)
```

Output

	mean	max	min	count
customer_segment				
Corporate	4525.0	6700	2500	4
Home Office	5000.0	8000	2000	2
Retail	4075.0	6000	1300	4

pivot_table() for Multi-Level Aggregation

Create multi-index tables like Excel pivot tables.

Features:

- Supports multiple indices and columns.
- Can specify aggregation method (e.g., mean, sum, count).

Original DataFrame

	customer_id	customer_segment	region	product	sales	revenue	cost	total_spent	signup_month
0	1	Retail	North	A	1200	1500	1000	5000	1
1	2	Corporate	South	B	850	1000	700	3400	3
2	3	Home Office	East	C	560	700	400	2000	2
3	4	Retail	West	A	1430	1600	1100	6000	1
4	5	Corporate	North	C	760	900	600	2500	4
5	6	Retail	East	B	980	1150	800	4000	3
6	7	Home Office	South	A	3050	3200	2800	8000	2
7	8	Corporate	North	C	1300	1400	1100	5500	1
8	9	Retail	West	B	210	300	250	1300	4
9	10	Corporate	South	A	1670	1800	1300	6700	3

Code

```
1 # Example: Average sales by region and product
2 pivot = df.pivot_table(values='sales', index='region', columns='product', aggfunc='mean', fill_value=0)
3 print(pivot)
```

Output

product	A	B	C
region			
East	0.0	980.0	560.0
North	1200.0	0.0	1030.0
South	2360.0	850.0	0.0
West	1430.0	210.0	0.0

query() for SQL-like Filtering

Clean, readable filtering syntax like SQL.

Why use it?

- Improves readability over chained conditions.
- Convenient for quick interactive filtering.

Original DataFrame

	customer_id	customer_segment	region	product	sales	revenue	cost	total_spent	signup_month
0	1	Retail	North	A	1200	1500	1000	5000	1
1	2	Corporate	South	B	850	1000	700	3400	3
2	3	Home Office	East	C	560	700	400	2000	2
3	4	Retail	West	A	1430	1600	1100	6000	1
4	5	Corporate	North	C	760	900	600	2500	4
5	6	Retail	East	B	980	1150	800	4000	3
6	7	Home Office	South	A	3050	3200	2800	8000	2
7	8	Corporate	North	C	1300	1400	1100	5500	1
8	9	Retail	West	B	210	300	250	1300	4
9	10	Corporate	South	A	1670	1800	1300	6700	3

Code

```
1 # Example: Filter rows where sales > 1000 and region is 'North'
2 filtered = df.query("sales > 1000 and region == 'North'")
3 display(filtered.head())
```

Output

	customer_id	customer_segment	region	product	sales	revenue	cost	total_spent	signup_month
0	1	Retail	North	A	1200	1500	1000	5000	1
7	8	Corporate	North	C	1300	1400	1100	5500	1

Lambda Functions and apply()

Apply custom logic across rows or columns.

Use cases:

- Create custom columns.
- Perform row-wise computations.

Original DataFrame

	customer_id	customer_segment	region	product	sales	revenue	cost	total_spent	signup_month
0	1	Retail	North	A	1200	1500	1000	5000	1
1	2	Corporate	South	B	850	1000	700	3400	3
2	3	Home Office	East	C	560	700	400	2000	2
3	4	Retail	West	A	1430	1600	1100	6000	1
4	5	Corporate	North	C	760	900	600	2500	4
5	6	Retail	East	B	980	1150	800	4000	3
6	7	Home Office	South	A	3050	3200	2800	8000	2
7	8	Corporate	North	C	1300	1400	1100	5500	1
8	9	Retail	West	B	210	300	250	1300	4
9	10	Corporate	South	A	1670	1800	1300	6700	3

apply() on columns:

```
1 # Example: Flag high-value customers
2 df['is_high_value'] = df['total_spent'].apply(lambda x: 'Yes' if x > 5000 else 'No')
3 df.head()
```

	customer_id	customer_segment	region	product	sales	revenue	cost	total_spent	signup_month	is_high_value
0	1	Retail	North	A	1200	1500	1000	5000	1	No
1	2	Corporate	South	B	850	1000	700	3400	3	No
2	3	Home Office	East	C	560	700	400	2000	2	No
3	4	Retail	West	A	1430	1600	1100	6000	1	Yes
4	5	Corporate	North	C	760	900	600	2500	4	No

apply() on rows:

```
1 # Row-wise logic (axis=1)
2 df['net_profit'] = df.apply(lambda row: row['revenue'] - row['cost'], axis=1)
3 df.head()
```

	customer_id	customer_segment	region	product	sales	revenue	cost	total_spent	signup_month	is_high_value	net_profit
0	1	Retail	North	A	1200	1500	1000	5000	1	No	500
1	2	Corporate	South	B	850	1000	700	3400	3	No	300
2	3	Home Office	East	C	560	700	400	2000	2	No	300
3	4	Retail	West	A	1430	1600	1100	6000	1	Yes	500
4	5	Corporate	North	C	760	900	600	2500	4	No	300

Cleaning Edge Cases

Pandas offers flexible tools to identify and correct common data issues efficiently.

Removing Duplicates:

```
1 df = df.drop_duplicates()
```

Inconsistent Labels:

```
1 df['customer_segment'] = df['customer_segment'].str.lower().str.strip()
```

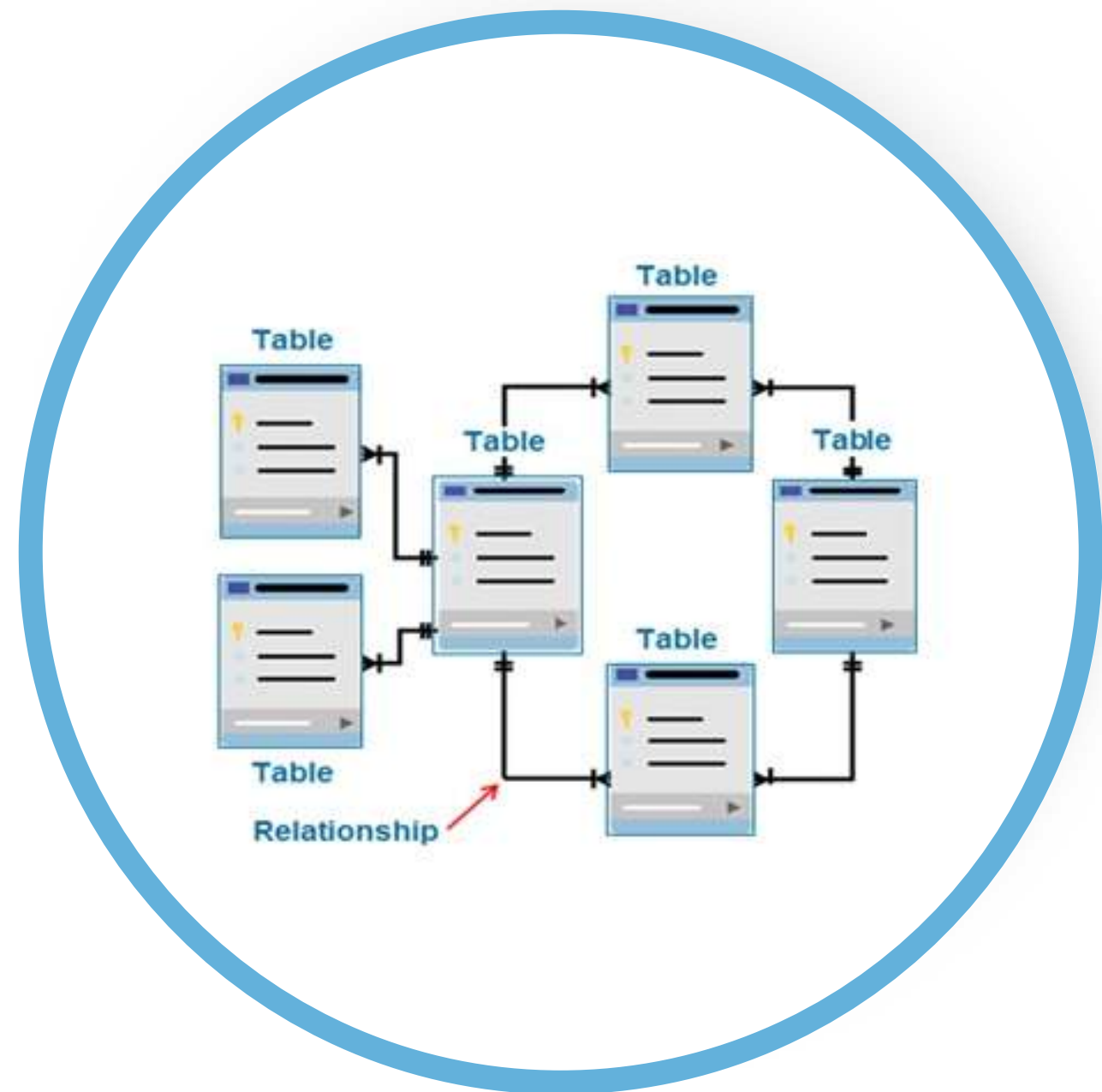


INTRODUCTION TO SQL & RELATIONAL THINKING

WHY SQL MATTERS FOR ANALYSTS?

- SQL is the **common language** used to communicate with **data systems**.
- Crucial for querying databases, especially in companies with centralised data warehouses.
- Works with **large-scale data efficiently** (PostgreSQL, MySQL, BigQuery, Snowflake, etc.).
- Used in most **data roles** (analytics, BI, product, data science).

RELATIONAL CONCEPTS



Primary Key

Uniquely identifies a record (e.g., customer_id)



Foreign Key

Links to another table's primary key (e.g., orders.customer_id)



Many-to-One Join

Multiple orders → one customer



One-to-Many Join

Think of joining "people" and their "orders" → many rows per person.

SQL BASICS – SELECT

SELECT - extracts data from a database

```
SELECT CustomerID, CustomerName, Country FROM customers;
```

CustomerID	CustomerName	Country
1	Alfreds Futterkiste	Germany
2	Ana Trujillo Emparedados y helados	Mexico
3	Antonio Moreno Taquería	Mexico
4	Around the Horn	UK
5	Berglunds snabbköp	Sweden

(Select specific columns)

```
SELECT * FROM customers;
```

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

(Select all columns)

SQL BASICS – SELECT, WHERE

WHERE- clause is used to filter records.

```
SELECT * FROM Customers WHERE Country='UK';
```

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
11	B's Beverages	Victoria Ashworth	Fauntleroy Circus	London	EC2 5NT	UK
16	Consolidated Holdings	Elizabeth Brown	Berkeley Gardens 12 Brewery	London	WX1 6LT	UK
19	Eastern Connection	Ann Devon	35 King George	London	WX3 6FW	UK
38	Island Trading	Helen Bennett	Garden House Crowther Way	Cowes	PO31 7PJ	UK
53	North/South	Simon Crowther	South House 300 Queensbridge	London	SW7 1RZ	UK
72	Seven Seas Imports	Hari Kumar	90 Wadhurst Rd.	London	OX15 4NB	UK

(Select all customers with Country equals to “UK”)

List of operators that can be used with WHERE clause:

=, >, <, >=, <=, <>, LIKE, BETWEEN, IN

SQL BASICS – SELECT, WHERE, ORDER BY

ORDER BY - keyword is used to sort the result in ascending or descending order.

```
SELECT * FROM Customers WHERE Country='UK' ORDER BY ContactName DESC;
```

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
11	B's Beverages	Victoria Ashworth	Fauntleroy Circus	London	EC2 5NT	UK
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
53	North/South	Simon Crowther	South House 300 Queensbridge	London	SW7 1RZ	UK
38	Island Trading	Helen Bennett	Garden House Crowther Way	Cowes	PO31 7PJ	UK
72	Seven Seas Imports	Hari Kumar	90 Wadhurst Rd.	London	OX15 4NB	UK
16	Consolidated Holdings	Elizabeth Brown	Berkeley Gardens 12 Brewery	London	WX1 6LT	UK
19	Eastern Connection	Ann Devon	35 King George	London	WX3 6FW	UK

ORDER BY keyword sorts the records in ascending order by default.

SQL BASICS – SELECT, WHERE, ORDER BY, GROUP BY

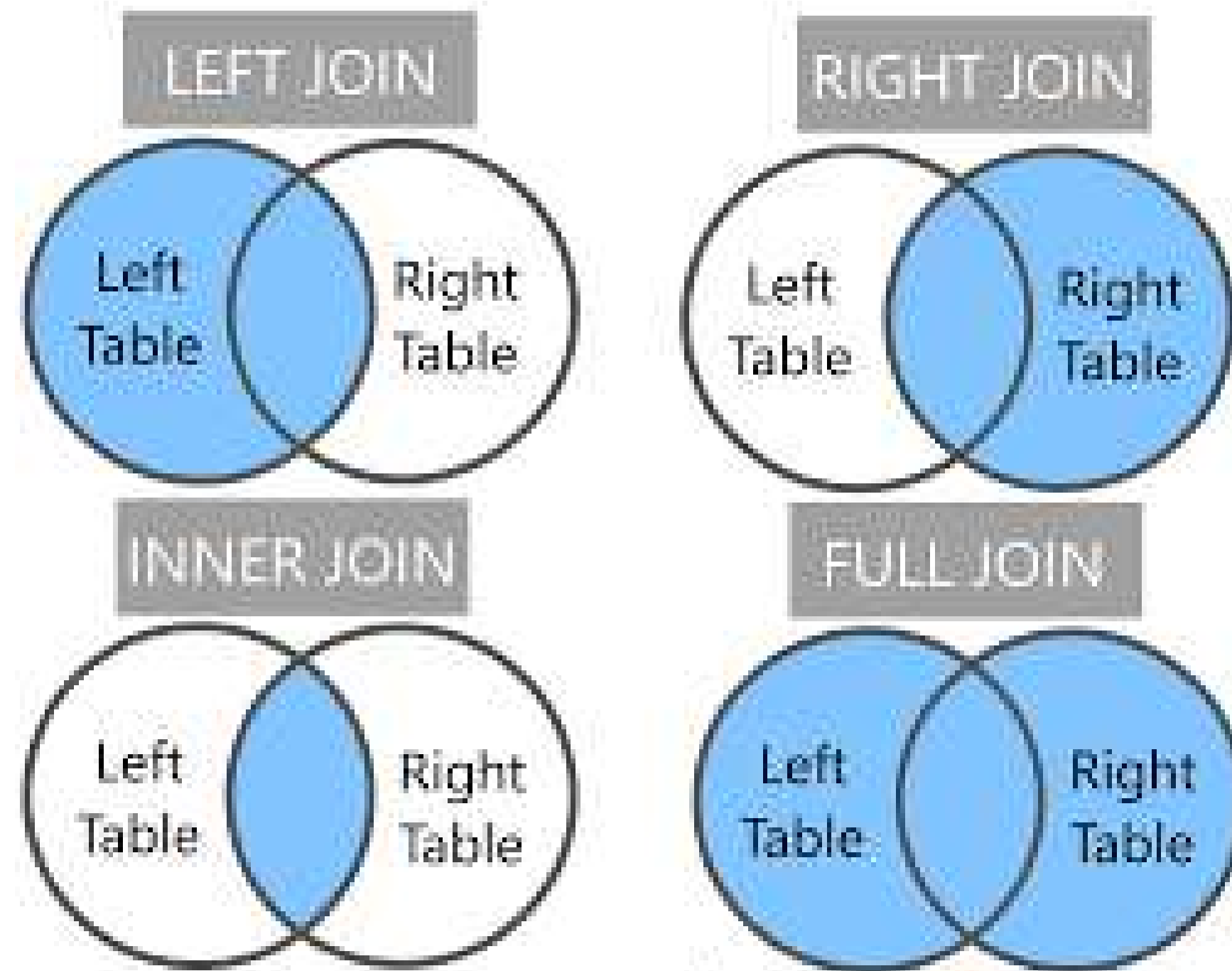
GROUP BY - statement groups rows that have the same values into summary rows.

often used with aggregate functions (COUNT(), MAX(), MIN(), SUM(), AVG()) to group the result-set by one or more columns.

```
SELECT Country, COUNT(CustomerID) FROM Customers GROUP BY Country;
```

Country	COUNT(CustomerID)
Venezuela	4
USA	13
UK	7
Switzerland	2
Sweden	2

TYPES OF JOINS



VISUAL EXAMPLE – JOINING DATA

```
1 # Create customer and order tables
2 customers = pd.DataFrame({
3     'customer_id': [1, 2],
4     'customer_segment': ['Retail', 'Corporate']
5 })
6
7 orders = pd.DataFrame({
8     'order_id': [101, 102],
9     'customer_id': [1, 1],
10    'sales': [250, 120]
11 })
12
13 # Join the two
14 merged_df = pd.merge(orders, customers, on='customer_id', how='left')
15 display(merged_df)
```

	order_id	customer_id	sales
0	101	1	250
1	102	1	120

+

	customer_id	customer_segment
0	1	Retail
1	2	Corporate

||

	order_id	customer_id	sales	customer_segment
0	101	1	250	Retail
1	102	1	120	Retail

VISUAL EXPLORATION & STORYTELLING

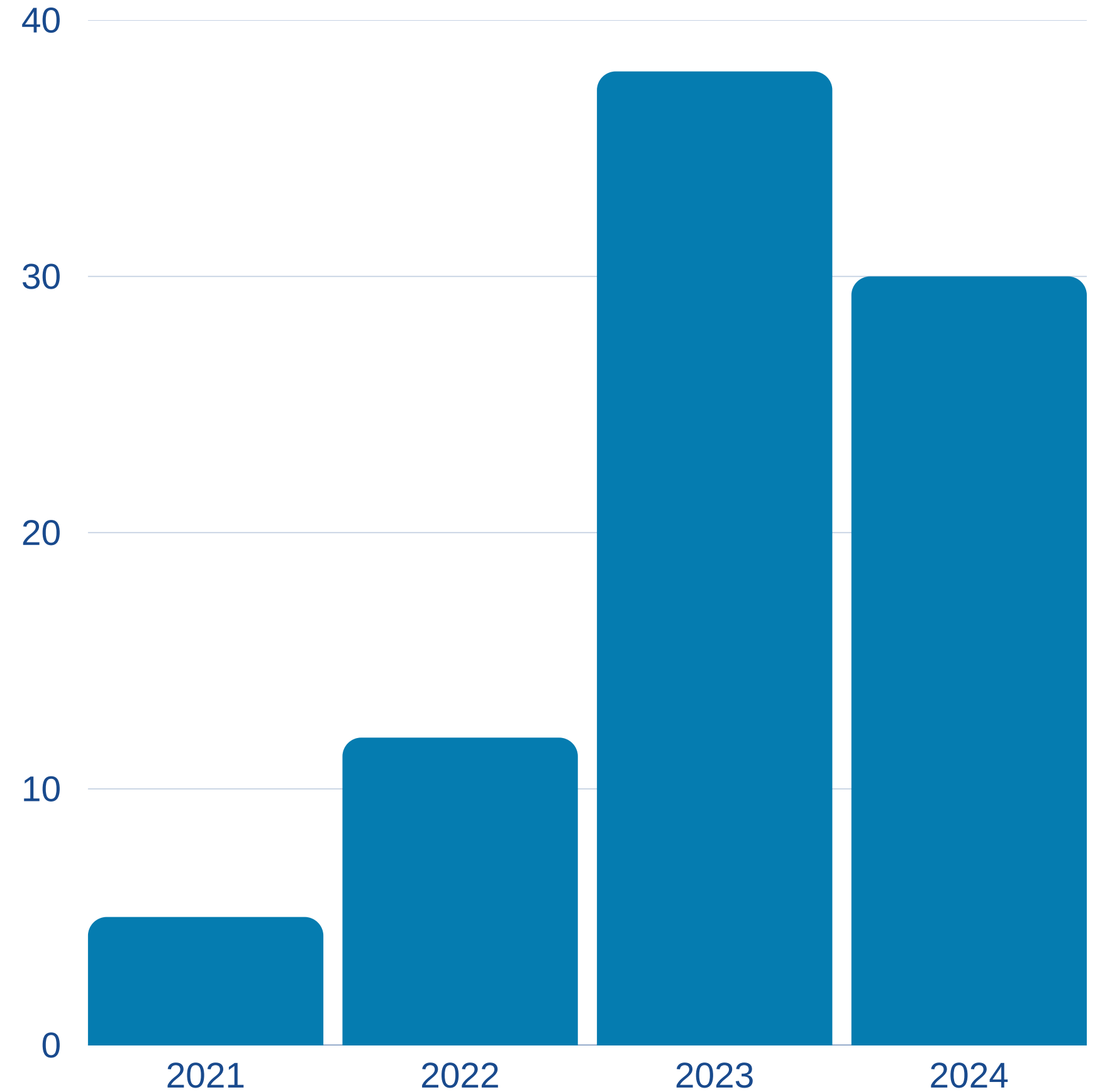
Principles of Effective Visualisation

● Good Design = Clarity + Honesty

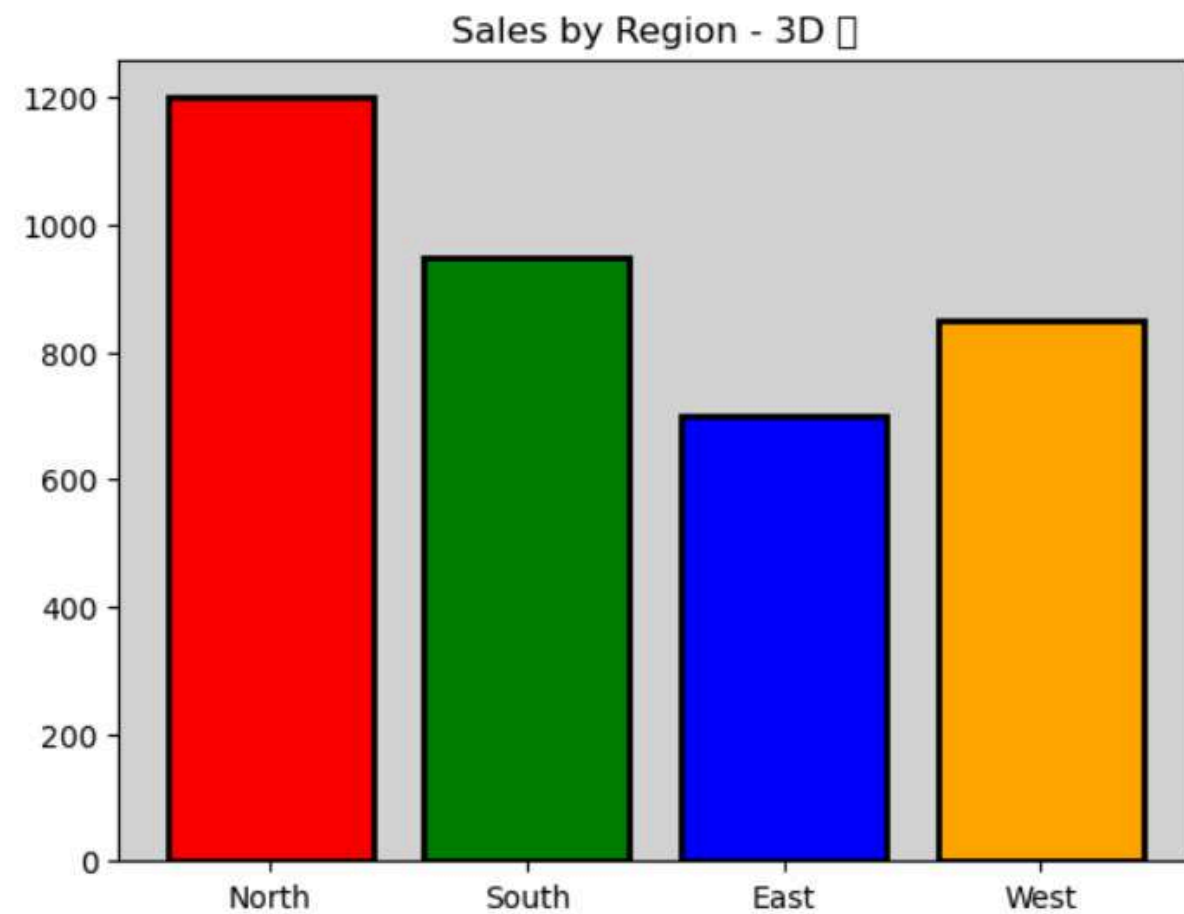
- Clear axes, labels, legends
- Remove clutter, focus on message

● Telling a Story

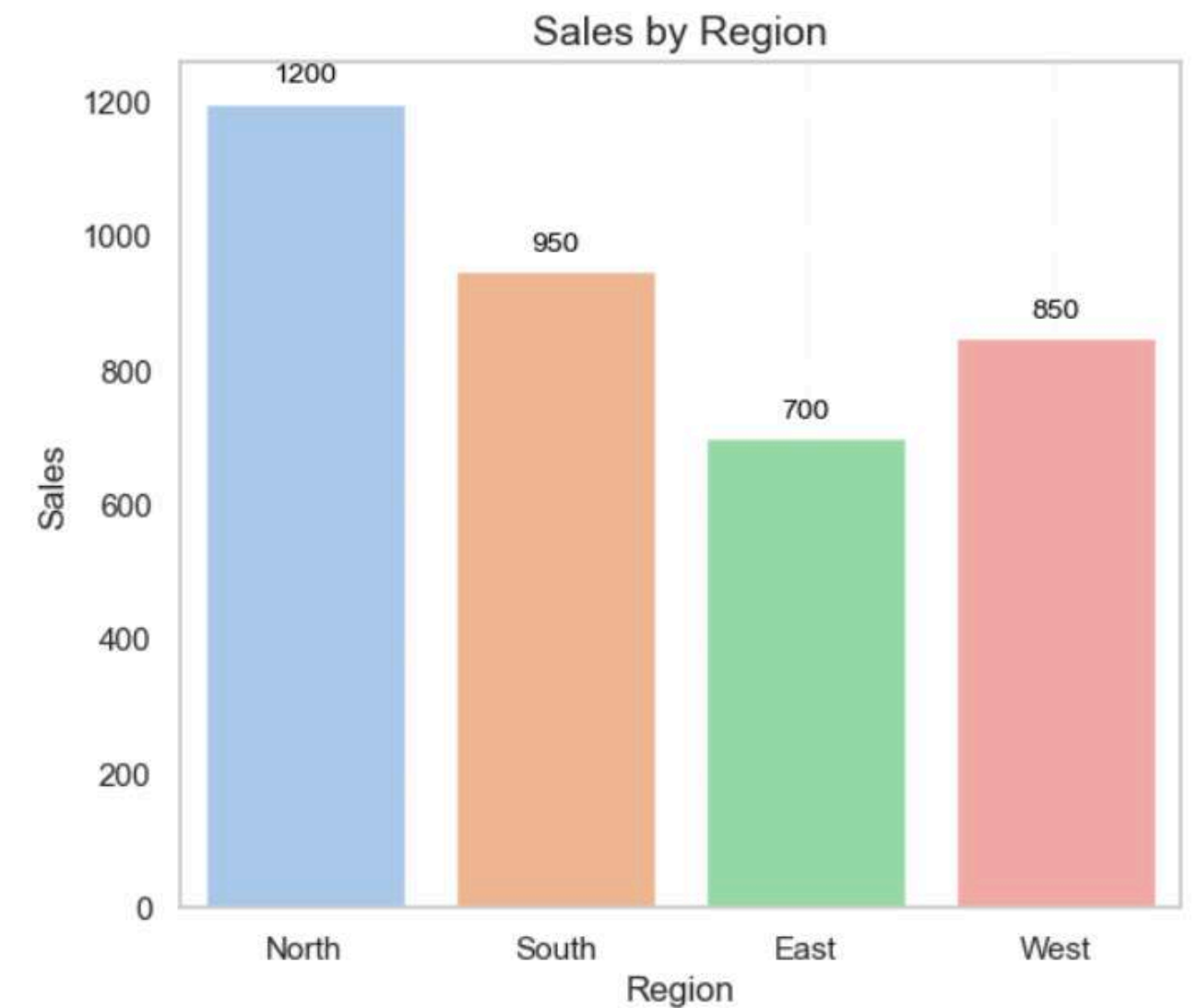
- Start with a question/use case
- Use chart to support insight
- Context matters more than decoration



AVOIDING CHARTJUNK – BAD VS. GOOD VISUALS

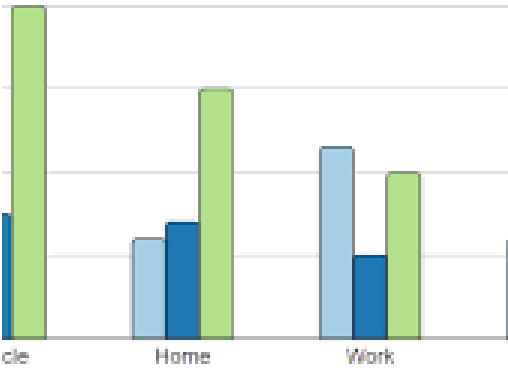


- ✓ Flat 2D bar chart
- ✓ Subtle colour scheme
- ✓ Clear labels and title
- ✓ Gridlines if needed
- ✓ Clean background

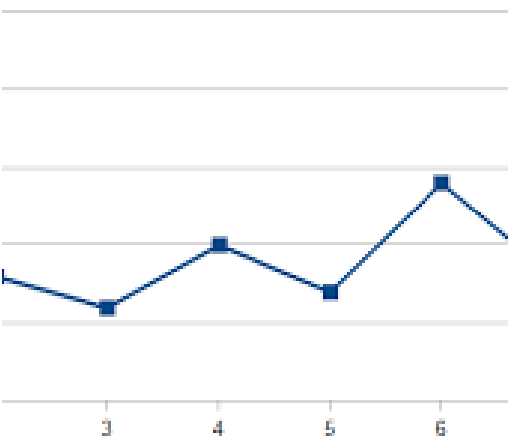


CHOOSING THE RIGHT CHART

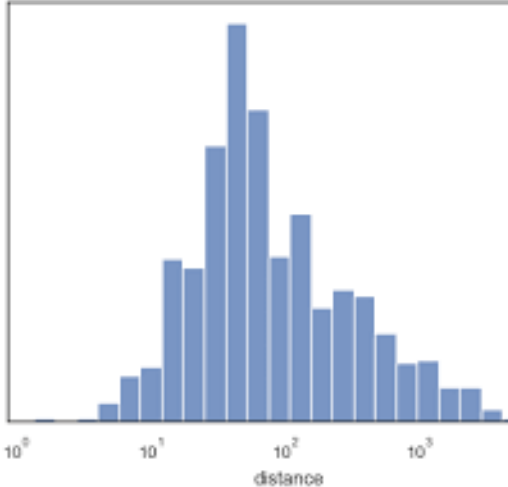
Bar Graph



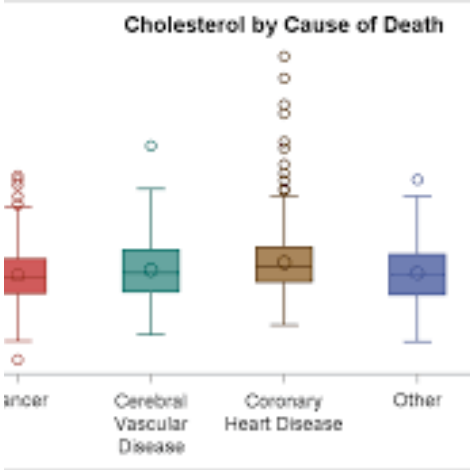
Line Graph



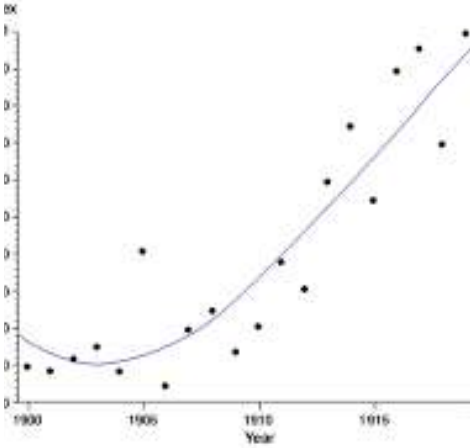
Histogram



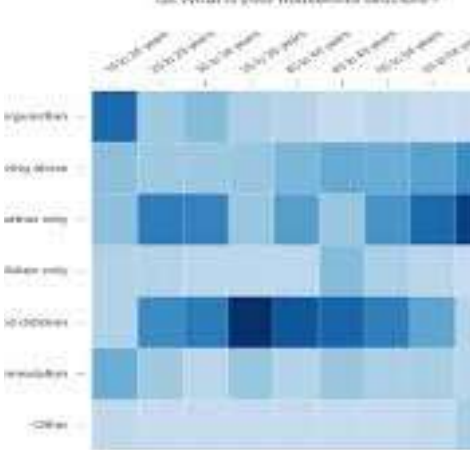
Boxplot



Scatter Plot



Heatmap



MATPLOTLIB ESSENTIALS

Matplotlib is the foundational plotting library in Python. It powers higher-level libraries like Seaborn and Plotly.

Good for low-level customisation and building static, publication-quality plots.

Diagnostic:
"Why did it happen?"

Analyzes data to uncover the causes behind past outcomes.



Predictive:
"What might happen?"

Uses historical data and models to forecast future outcomes.

Prescriptive:
"What should we do?"

Recommends actions based on predicted scenarios to optimise results.



MATPLOTLIB ESSENTIALS

01

Good for **low-level customisation** and building **static, publication-quality plots**.

02

Ideal when you need full control over labels, layout, colour, and figure size.

03

Often used with **NumPy** or **Pandas**.

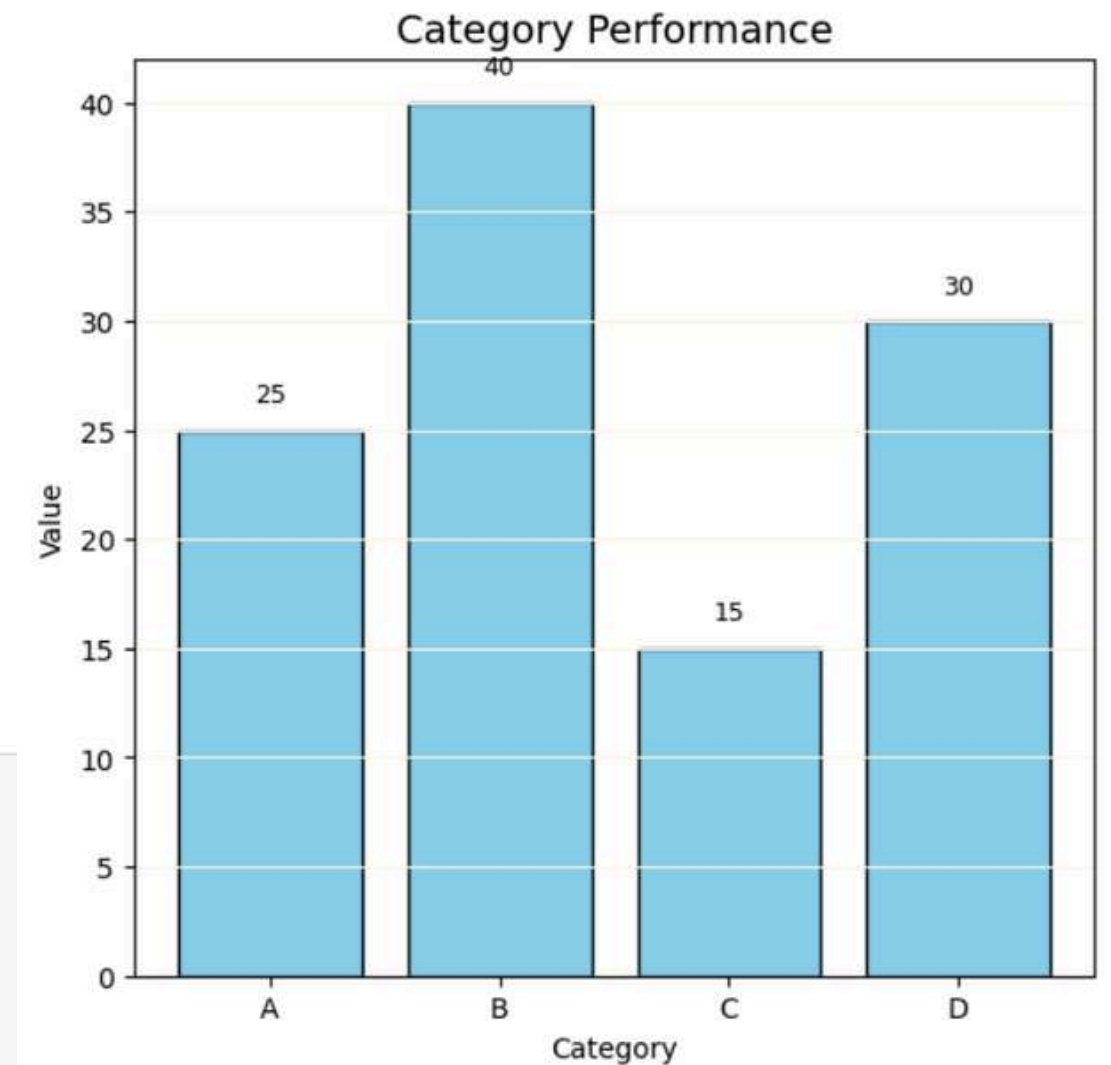
Matplotlib is the **foundational plotting library** in Python. It powers higher-level libraries like Seaborn and Plotly.

- **plt.plot()** is for line charts, great for trends.
- **plt.bar()**, **plt.hist()**, **plt.scatter()** are used for standard chart types.
- Use **plt.title()**, **plt.xlabel()**, **plt.ylabel()** for proper annotation.
- **plt.grid()** adds helpful axis gridlines.
- **plt.subplots_adjust()** controls padding around the plot.
- Best practice: always label axes and use **tight_layout()** or manual padding.

```

1 import matplotlib.pyplot as plt
2
3 # Data
4 categories = ['A', 'B', 'C', 'D']
5 values = [25, 40, 15, 30]
6
7 # Create figure and axis
8 fig, ax = plt.subplots(figsize=(6, 4))
9
10 # Bar plot
11 bars = ax.bar(categories, values, color='skyblue', edgecolor='black')
12
13 # Titles and labels
14 ax.set_title("Category Performance", fontsize=14)
15 ax.set_xlabel("Category")
16 ax.set_ylabel("Value")
17
18 # Light gridlines
19 ax.grid(True, axis='y', color='#fdf6e3')
20
21 # Add value labels above bars
22 for bar in bars:
23     height = bar.get_height()
24     ax.text(bar.get_x() + bar.get_width() / 2, height + 1, f'{height}',
25            ha='center', va='bottom', fontsize=9)
26
27 # ✅ Adjust inner padding
28 plt.subplots_adjust(left=0.15, right=0.95, top=1.58, bottom=0.15)
29
30 # Show plot
31 plt.show()

```



SEABORN DEEP DIVE

01

Designed for **statistical plots** and **data exploration**.

02

Comes with **sensible defaults**, beautiful styles, and built-in themes.

03

- Especially powerful for **grouped plots**, **distribution charts**, and **pairwise comparisons**.

Seaborn is a high-level data visualisation library built on top of Matplotlib.

- **Easier and faster** than Matplotlib for common tasks.
- **hue=, col=, and row=** enable automatic grouping and faceting.
- Built-in functions like **boxplot()**, **countplot()**, **heatmap()**, and **pairplot()** save time.
- Supports **themes and colour palettes** (e.g. Set2, coolwarm, flare).
- Great for **EDA, comparisons, and highlighting relationships** in data.

```
1 import seaborn as sns
2 import matplotlib.pyplot as plt
3
4 # Example dataset
5 tips = sns.load_dataset("tips")
6 display(tips.tail(10))
```

	total_bill	tip	sex	smoker	day	time	size
234	15.53	3.00	Male	Yes	Sat	Dinner	2
235	10.07	1.25	Male	No	Sat	Dinner	2
236	12.60	1.00	Male	Yes	Sat	Dinner	2
237	32.83	1.17	Male	Yes	Sat	Dinner	2
238	35.83	4.67	Female	No	Sat	Dinner	3
239	29.03	5.92	Male	No	Sat	Dinner	3
240	27.18	2.00	Female	Yes	Sat	Dinner	2
241	22.67	2.00	Male	Yes	Sat	Dinner	2
242	17.82	1.75	Male	No	Sat	Dinner	2
243	18.78	3.00	Female	No	Thur	Dinner	2

DataFrame Understanding:

total_bill: Numeric (£)

The total cost of the meal (including food and drinks).

tip: Numeric (£)

The tip amount given to the waiter/waitress.

sex: Categorical

The gender of the customer (Male or Female).

smoker: Categorical

Whether the customer was a smoker (Yes or No).

day: Categorical

Day of the week the bill was recorded (Thur, Fri, Sat, Sun).

time: Categorical

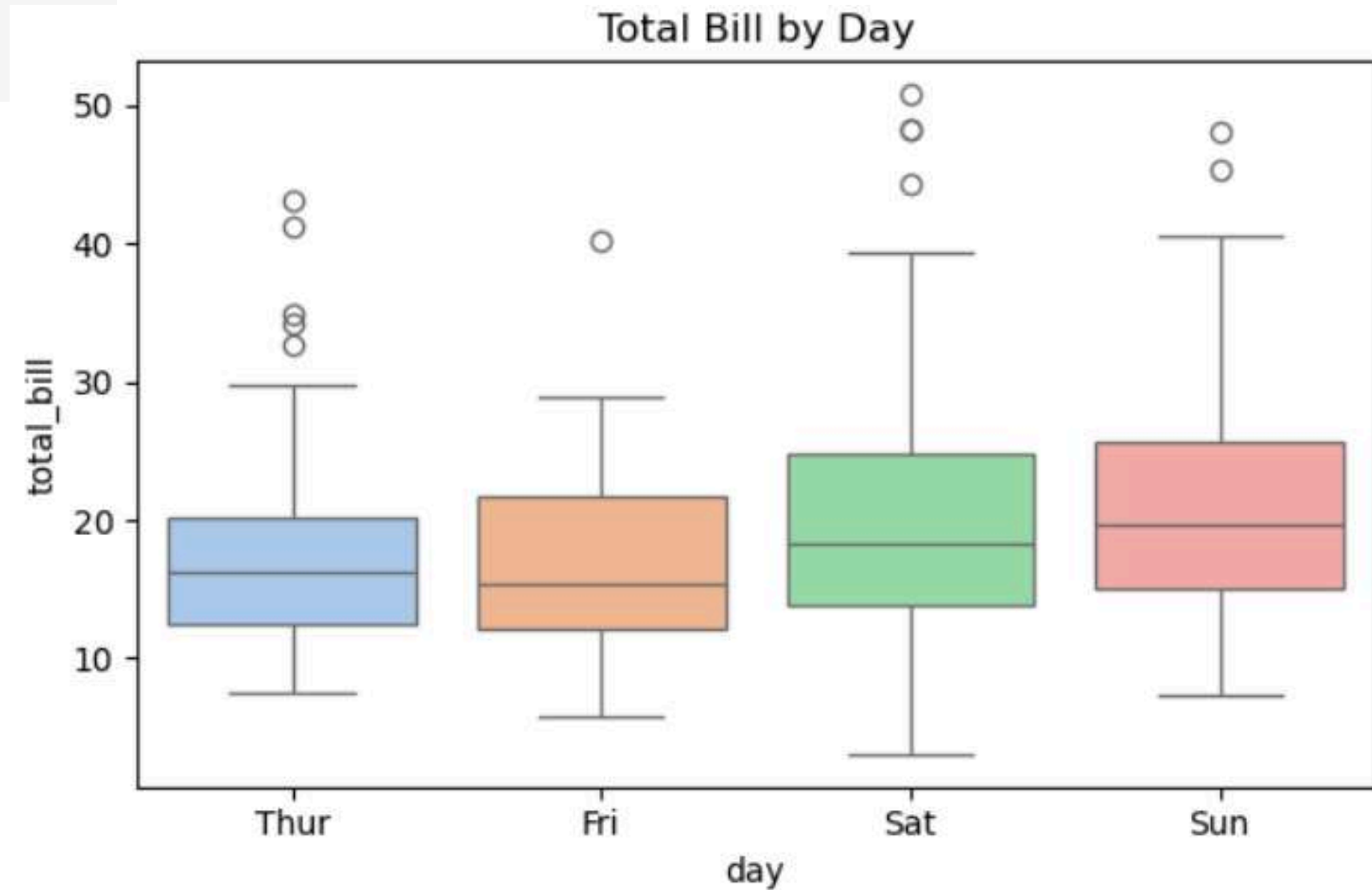
Time of day: Lunch or Dinner.

size: Numeric

Number of people in the dining party.

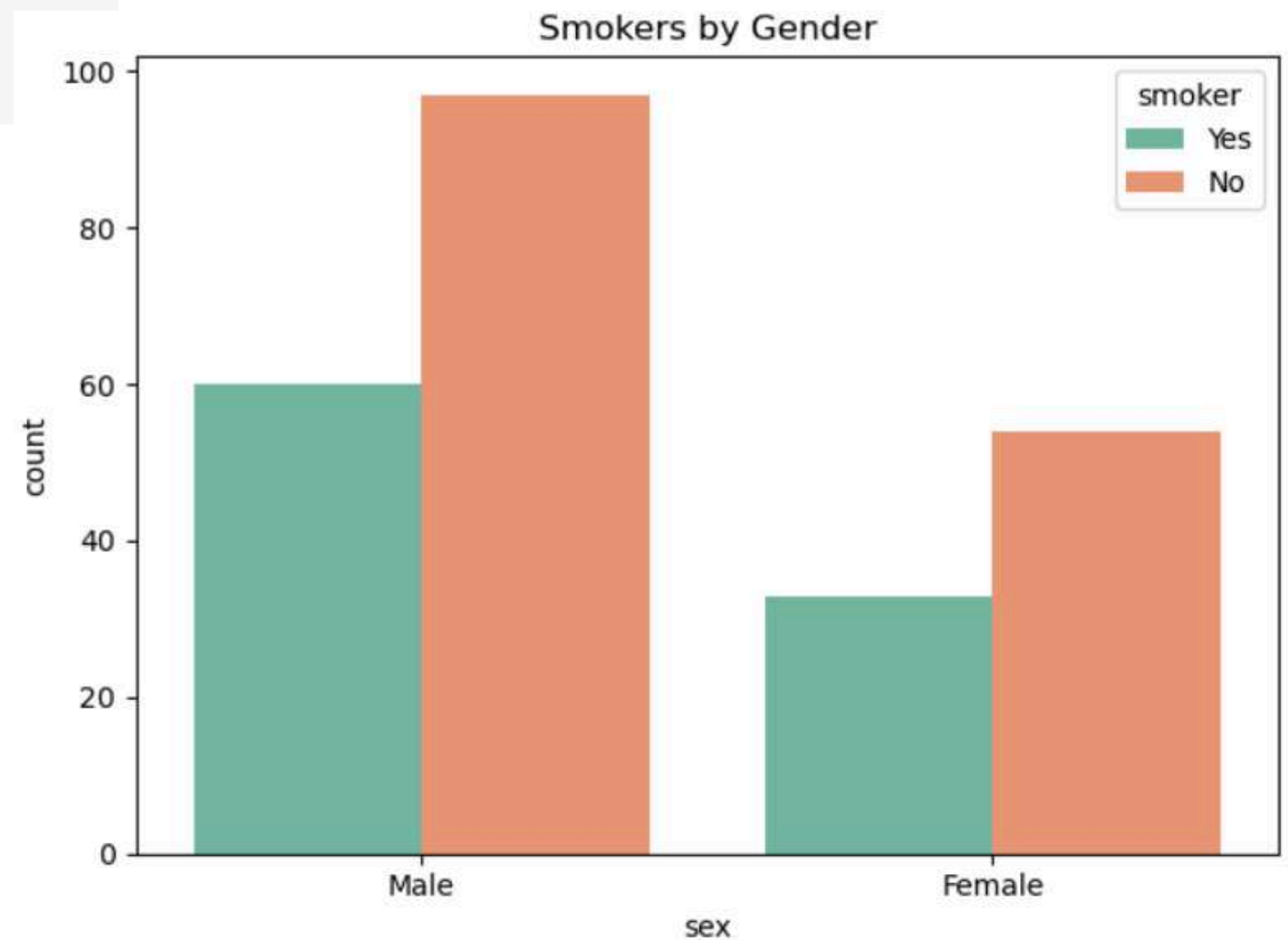
Boxplots show **spread and outliers** — perfect for comparing numeric distributions across categories.

```
7 # 1. Boxplot: distribution across categories
8 plt.figure(figsize=(6, 4))
9 sns.boxplot(x='day', y='total_bill', data=tips, palette='pastel')
10 plt.title("Total Bill by Day")
11 plt.tight_layout()
12 plt.show()
```



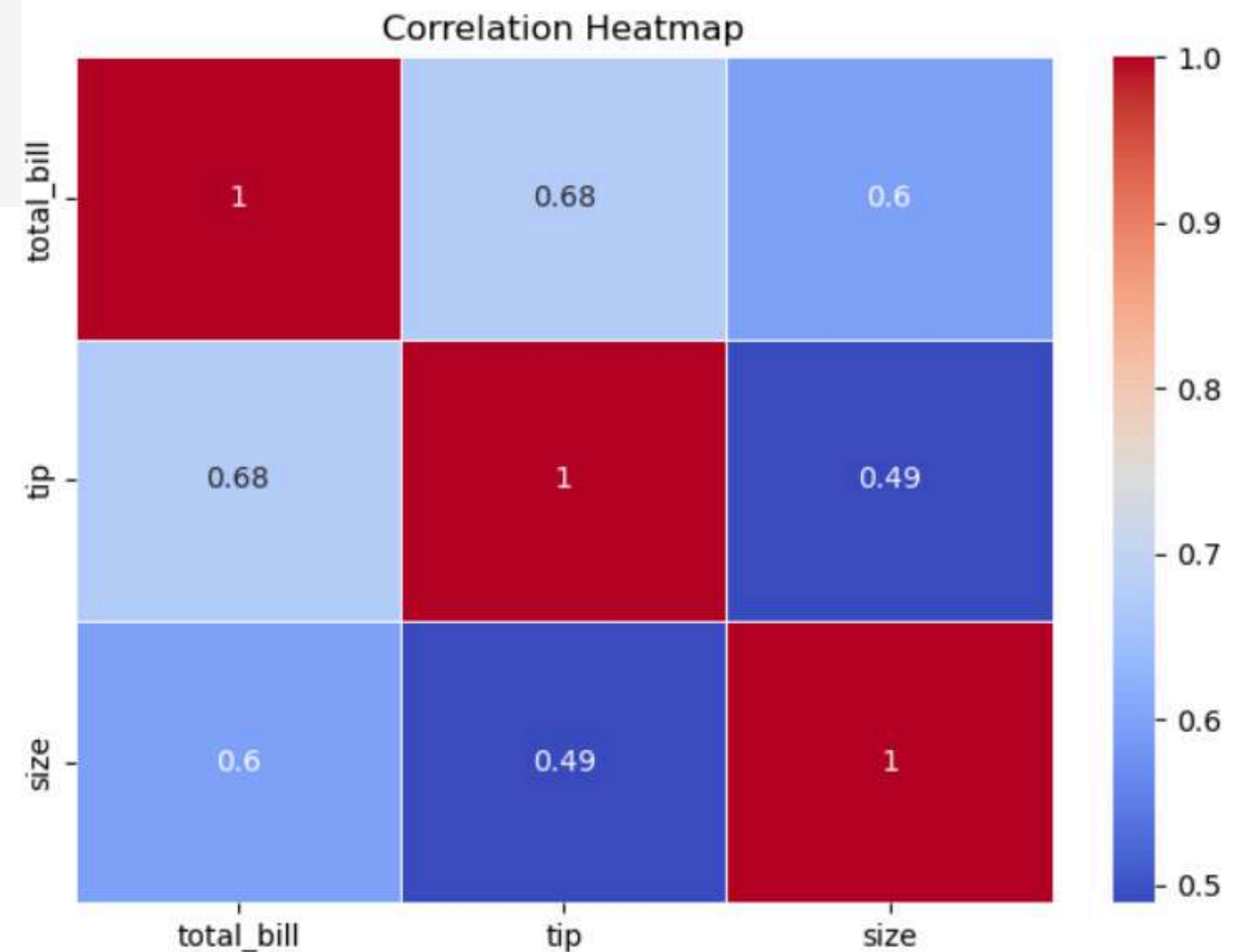
Countplots are excellent for **categorical frequency** (and quickly group with **hue=**).

```
14 # 2. Countplot: frequency of categories
15 sns.countplot(x='sex', hue='smoker', data=tips, palette='Set2')
16 plt.title("Smokers by Gender")
17 plt.tight_layout()
18 plt.show()
```



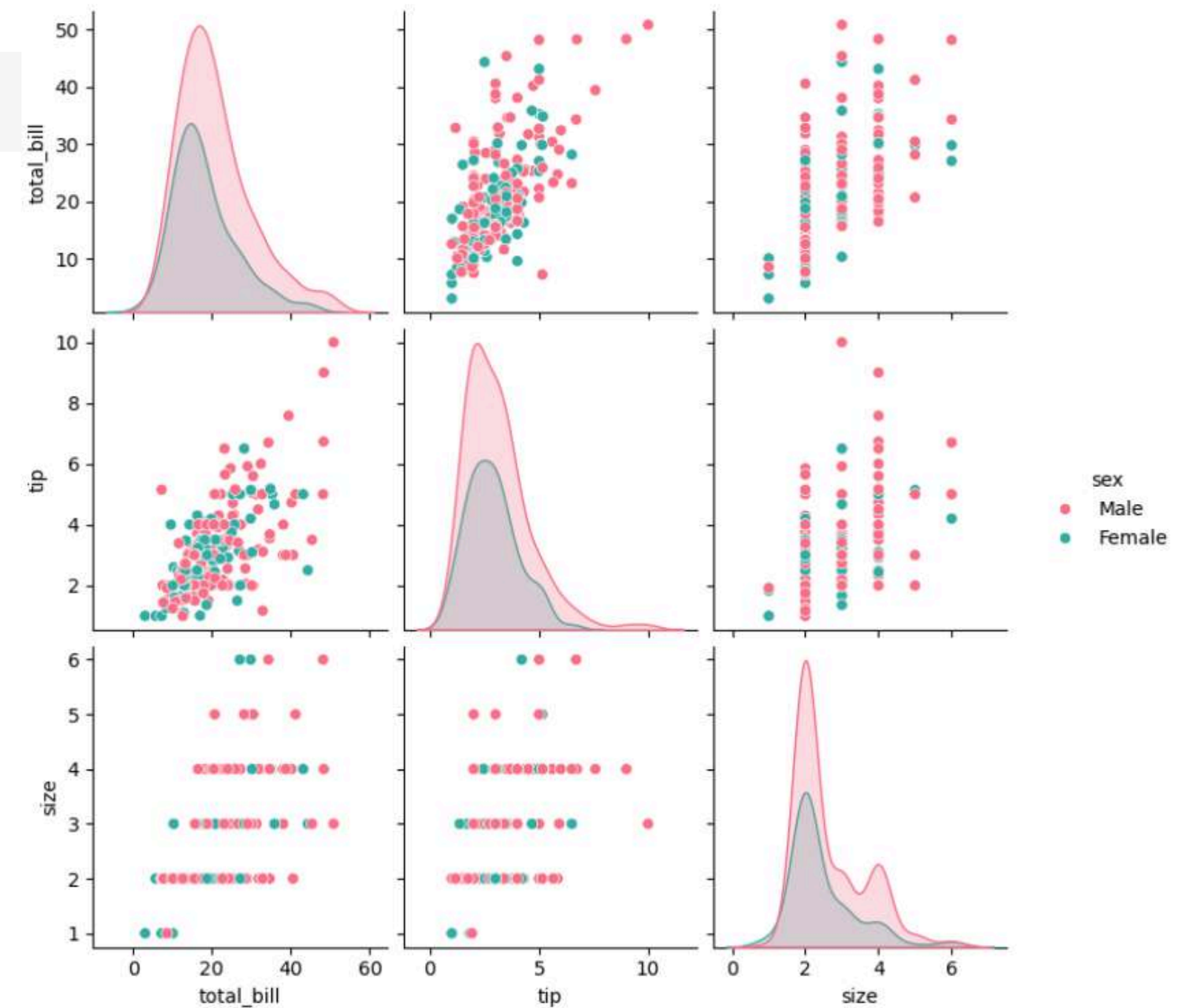
Heatmaps visually surface correlations — especially useful during feature selection.

```
20 # 3. Heatmap: correlation matrix
21 corr = tips.corr(numeric_only=True)
22 sns.heatmap(corr, annot=True, cmap='coolwarm', linewidths=0.5)
23 plt.title("Correlation Heatmap")
24 plt.tight_layout()
25 plt.show()
```



Pairplots give an instant **multivariate overview** with minimal effort.

```
27 # 4. Pairplot: scatter matrix
28 sns.pairplot(tips, hue='sex', palette='husl')
```



Mirrored bar chart enables clear **side-by-side comparison** between two groups across a shared variable, helping reveal **patterns, imbalances, or trends** at a glance.

```
import pandas as pd
import matplotlib.pyplot as plt

# Sample data: Age groups and male/female population
age_groups = ['80+', '75-79', '70-74', '65-69', '60-64', '55-59', '50-54',
              '45-49', '40-44', '35-39', '30-34', '25-29', '20-24', '15-19',
              '10-14', '5-9', '0-4']

# Note: Male values are negative for the mirrored effect
male = [-5000, -6000, -7000, -8000, -9000, -9500, -10000, -11000, -12000,
        -13000, -14000, -13500, -13000, -12000, -10000, -8000, -6000]

female = [5200, 6200, 7100, 8200, 9100, 9600, 10200, 11100, 12200,
          13300, 14300, 13700, 13200, 12200, 10100, 8100, 6200]

# Create DataFrame
df = pd.DataFrame({
    'Age Group': age_groups,
    'Male': male,
    'Female': female
})

# Plotting
fig, ax = plt.subplots(figsize=(10, 8))

# Horizontal bar chart
ax.barh(df['Age Group'], df['Male'], color='skyblue', label='Male')
ax.barh(df['Age Group'], df['Female'], color='lightpink', label='Female')

# Labels and formatting
ax.set_title('Age and Gender Distribution of Learners in London', fontsize=14)
ax.set_xlabel('Population')
ax.set_ylabel('Age Group')
ax.legend(loc='upper right')

# Make x-axis labels absolute values
xticks = ax.get_xticks()
ax.set_xticklabels([abs(int(x)) for x in xticks])

# Clean layout
plt.tight_layout()
plt.gca().invert_yaxis() # So the oldest age group is at the top
plt.show()
```



QUESTIONS?