main

September 30, 2025

0.1 # Python for Data Analytics

Skill Level: Absolute beginners to intermediate learners

Career Paths: Data Analyst, Business Analyst, Junior Data Scientist.

Goal: Equip learners with job-ready skills in Python for data analytics, from programming fundamentals to applied analytics, data wrangling, visualization, and storytelling.

Tools: Jupyter Notebook, pandas, NumPy, matplotlib, seaborn, and scikit-learn.

Project Format: Mini-projects, real-world datasets, code reviews

---

1 Phase 1: Python Programming Foundations

(Weeks 1–4)-Build foundational Python skills for data work.

2 Week 1: Python Basics & Environment Setup

Topics: * Installing Python and Jupyter Notebooks * Data types, variables, inputs, outputs * Basic arithmetic and string formatting

Exercises: * BMI calculator * Name Formatter * Currency Convert

Project: * "Data Entry Sanitizer" Clean and format text inputs like phone numbers or names.

Learning Objectives

By the end of this week, you will be able to: - Set up Python and Jupyter notebooks on your system - Understand and use basic data types and variables - Perform arithmetic and format strings - Write small scripts that take user input and produce output

2.1 Setting Up Your Environment

- Python: The programming language we'll use to write all our data analysis code.
- Install Python: https://www.python.org/downloads/
- **Python virtual environment**
  - an isolated environment on your computer, where you can run and test your Python projects.

- It allows you to manage project-specific dependencies without interfering with other projects or the original Python installation.

Using virtual environments is important because:

- It prevents package version conflicts between projects
- Makes projects more portable and reproducible
- Keeps your system Python installation clean
- Allows testing with different Python versions

Creating Virtual Environment

- Python has the built-in venv module for creating virtual environments.
- `python -m venv /path/to/new/virtual/environment`

JupyterLab

- Install Jupyter using pip: https://jupyter.org/install

```
#Install JupyterLab with pip:
pip install jupyterlab

#launch jupyterlab
jupyter lab
```

Jupyter Notebook

- Jupyter Notebook: An interactive workspace to write, run, and document Python code easily.

```
#Install the classic Jupyter Notebook with:
pip install notebook

#To run the notebook:
jupyter notebook
```

3 Syntax

3.1 Indentation

- Indentation refers to the spaces at the beginning of a code line.
- Python uses indentation to indicate a block of code.
- Python will give you an error if you skip the indentation
- The recommended number of spaces is four and have to be the same in a block of code, otherwise Python will give you an error.

3.2 Comments

Used to:

- Explain code.
- Make the code more readable.
- Prevent execution when testing code.

3.3 Markdown Basics

Markdown is a lightweight markup language that uses characters like # for headings and * for emphasis to format text simply and intuitively. | Element | Markdown Syntax | |————————-|————————————————-| | Heading | `# H1## H2### H3` | | Bold | `**bold text**` | | Italic | `*italicized text*` | | Blockquote | `> blockquote` | | Ordered List | `1. First item2. Second item3. Third item` | | Unordered List | `- First item- Second item- Third item` | | Code | `` `code` `` | | Horizontal Rule| `---` | | Link | `[title](https://www.example.com)` | | Image | `![alt text](image.jpg)` |

Here is a more info on Markdown

4 Data types, variables, inputs, outputs

4.1 Key Concepts:

- **Variable:** A named container to store data.
- **Data types:** Types of data like numbers, text (strings), and True/False (booleans).
- **Input:** Getting data from the user.
- **Output:** Displaying information back to the user.

4.1.1 Data Types

- define what kind of data a variable holds: |Common data type || |————-|———-| |Text Type | str| |Numeric Types| int, float, complex |Sequence Types| list, tuple, range |Mapping Type| dict |Set Types| set, frozenset |Boolean Type| bool |Binary Types| bytes, bytearray, memoryview

[1]:
```
# Integer
age = 25

# Float
height = 5.9

# String
name = "Maale"

# Boolean
is_student = True
```

4.1.2 Variables

- Variables are temporary storage locations that hold data during a program's execution.
- Think of them as labeled boxes where we can store information like a name, number, or result.

[2]:
```
name:str = "Maale"
age:int = 25
pi:float = 3.1415
```

4.1.3 Getting User Input & Displaying Output

- **input()** lets users type data into the program.
- **print()** displays output or messages on the screen.

[7]:
```
# Input
user_name = str(input("Enter your name: "))

# Output
print("Hello,", user_name)
```

```
Enter your name:  ee

Hello, ee
```

5 Basic arithmetic and string formatting

5.1 Key Concepts:

- Arithmetic operators: +, -, *, / for math operations.
- String formatting: Inserting variables inside text cleanly for messages.

5.1.1 Arithmetic Operators

- Used to compute values or process numeric data.

[6]:
```
a = 10
b = 3

print("Addition:", a + b)
print("Subtraction:", a - b)
print("Multiplication:", a * b)
print("Division:", a / b)
print("Modulus:", a % b)
print("Exponent:", a ** b)
print("Floor division:", a // b)
```

```
Addition: 13
Subtraction: 7
Multiplication: 30
Division: 3.3333333333333335
Modulus: 1
Exponent: 1000
Floor division: 3
```

5.1.2 String Formatting

- Used to build readable messages or combine text and variables

[ ]:
```
name = "Ollie"
age = 12

# Using f-strings
print(f"My name is {name} and I am {age} years old.")
```

6 Exercises:

6.0.1 BMI calculator

[ ]:
```
# BMI = weight (kg) / height (m)^2

weight = float(input("Enter your weight in kg: "))
height = float(input("Enter your height in meters: "))

bmi = weight / (height ** 2)

print(f"Your BMI is {bmi:.2f}")
```

6.0.2 Name Formatter

[ ]:
```
print("Name Formatter")

first = input("Enter first name: ").strip()
last = input("Enter last name: ").strip()

formatted_name = f"{first.capitalize()} {last.upper()}"
print(f"Formatted Name: {formatted_name}")
```

6.0.3 Currency Convert

[34]:
```
usd = float(input("Enter amount in USD: "))
exchange_rate = 3600  # Example: 1 USD = 3600 UGX

inr = usd * exchange_rate

print(f"{usd} USD = {inr:.2f} UGX")
```

```
Enter amount in USD:  100
```

```
100.0 USD = 360000.00 UGX
```

7 Project:

7.0.1 Data Entry Sanitizer

Goal

Write a Python script that takes in raw text input (like names or phone numbers) and cleans/formats them for consistent data entry.

Requirements * Accept a full name (e.g., " maale AUSTIN ") → Output:"Maale Austin" * Accept a phone number (e.g., " (077) 395-3489 " or "0773953489") → Output: +256-077-395-3489 * Accept an email address and lowercase it

[ ]:
```python
def clean_name(name):
    parts = name.strip().split()
    return " ".join([part.capitalize() for part in parts])

def format_phone(phone):
    digits = ''.join(filter(str.isdigit, phone))
    if len(digits) == 10:
        return f"+256-{digits[:3]}-{digits[3:6]}-{digits[6:]}"
    return "Invalid number"

def clean_email(email):
    return email.strip().lower()

# User Input
raw_name = input("Enter full name: ")
raw_phone = input("Enter phone number: ")
raw_email = input("Enter email address: ")

# Cleaned Output
print("\n--- Cleaned Entry ---")
print(f"Name: {clean_name(raw_name)}")
print(f"Phone: {format_phone(raw_phone)}")
print(f"Email: {clean_email(raw_email)}")
```

[ ]:
```python
# Names (remove extra spaces, capitalize properly)
def sanitize_name(name):
    return name.strip().title()

# Example
raw_name = input("Enter a name: ")
cleaned_name = sanitize_name(raw_name)
print(f"Sanitized Name: {cleaned_name}")
```

[ ]:
```python
# Phone numbers (remove dashes/spaces, ensure valid format)
def sanitize_phone_number(phone):
    # Remove spaces, dashes, parentheses
    phone = phone.replace(" ", "").replace("-", "").replace("(", "").
 ↪replace(")", "")

    # Check if starts with country code; if not, add one
    if len(phone) == 10:
```

```
        phone = "+256" + phone  # Example for India
    elif not phone.startswith("+"):
        phone = "+" + phone

    return phone

# Example
raw_phone = input("Enter phone number: ")
clean_phone = sanitize_phone_number(raw_phone)
print(f"Sanitized Phone Number: {clean_phone}")
```

[ ]:
```
name = input("Enter your name: ")
phone = input("Enter your phone number: ")

print("\nSanitized Outputs:")
print("Name:", sanitize_name(name))
print("Phone:", sanitize_phone_number(phone))
```

8 Week 2: Control Flow & Logic in Data Contexts

Topics: * Conditional logic: if, elif, else logic * Loops: for, while, range() * Boolean expressions and comparison operators * Simple data validation (email, age, salary)

Project: * "Survey Quality Checker" Write a script to flag bad entries in a mock survey dataset.

Learning Objectives

By the end of this week, you will be able to: - Use `if`, `elif`, and `else` for conditional logic - Apply `for` and `while` loops to iterate over data - Use comparison and boolean operators - Validate simple inputs like email, age, and salary using logic

8.1 Control Flow Basics

How to make decisions in code.

8.2 Key Concepts:

- if checks a condition to see if it's true.
- elif (else if) checks another condition if the first was false.
- else runs if none of the above were true.

8.2.1 `if`, `elif`, `else`

- Used to make decisions in code.
- The program runs certain code blocks only if conditions are met.

[ ]:
```
age = 18

if age >= 18:
```

```
    print("You can vote.")
elif age == 17:
    print("Almost there!")
else:
    print("Too young to vote.")
```

9 Loops

Repeating actions multiple times.

9.1 Key Concepts:

for loops run once for each item in a list or range.

while loops run as long as a condition stays true.

range() creates a sequence of numbers to loop over.

9.1.1 for Loop

- iterates over a sequence (e.g., list of names).

[ ]:
```
for i in range(5):
    print(i)
```

9.1.2 while Loop

- runs as long as a condition is true.

[ ]:
```
count = 0
while count < 5:
    print(count)
    count += 1
```

9.1.3 Boolean Operators

Checking conditions and making decisions.

9.2 Key Concepts:

- Boolean values are True or False.
- Comparison operators: ==, !=, >, <, >=, <= to compare values.

Operator	Description	Example
`==`	Equal to	`a == b`
`!=`	Not equal to	`a != b`
`>`	Greater than	`a > b`
`<`	Less than	`a < b`
`>=`	Greater or equal	`a >= b`

Operator	Description	Example
`<=`	Less or equal	`a <= b`
`and`	Logical AND	`a > 5 and b < 10`
`or`	Logical OR	`a > 5 or b < 10`
`not`	Logical NOT	`not True`

10 Simple Data Validation

10.0.1 Making sure data makes sense before using it.

- Validating if inputs meet rules (e.g., email contains "@", age is positive).
- Using conditions and logic to check data quality.

10.0.2 Email Checker

[ ]:
```
email = input("Enter email: ")

if "@" in email and "." in email:
    print("Looks like a valid email.")
else:
    print("Invalid email.")
```

10.0.3 Age Validator

[ ]:
```
age = int(input("Enter your age: "))

if age <= 0:
    print("Invalid age.")
elif age < 18:
    print("Underage")
else:
    print("Eligible")
```

10.0.4 Real-World Relevance

Control flow is essential when: - filtering, validating, and processing data, critical for cleaning, building logic-based rules, and flagging inconsistencies in datasets.

11 Practice Exercises

11.0.1 1. Check if number is even or odd

[ ]:
```
num = int(input("Enter a number: "))
if num % 2 == 0:
    print("Even")
else:
    print("Odd")
```

11.0.2 2. Grade Checker

[ ]:
```
score = int(input("Enter your score: "))
if score >= 90:
    print("Grade: A")
elif score >= 75:
    print("Grade: B")
elif score >= 60:
    print("Grade: C")
else:
    print("Grade: F")
```

11.0.3 3. Password Length Validator

[ ]:
```
password = input("Enter password: ")
if len(password) < 8:
    print("Password too short")
else:
    print("Password accepted")
```

11.0.4 4. Loop through a list of numbers and print only positives

[ ]:
```
nums = [10, -5, 0, 22, -3]
for n in nums:
    if n > 0:
        print(n)
```

11.0.5 5. Count how many even numbers in a list

[ ]:
```
nums = [1, 2, 3, 4, 5, 6]
count = 0
for n in nums:
    if n % 2 == 0:
        count += 1
print(f"{count} even numbers")
```

11.0.6 6. Ask for input until user types “exit”

[ ]:
```
while True:
    cmd = input("Type something (or 'exit'): ")
    if cmd == "exit":
        break
```

11.0.7 7. Salary filter

[ ]:
```python
salary = float(input("Enter salary: "))
if salary < 30000:
    print("Below threshold")
elif salary < 70000:
    print("Mid-range")
else:
    print("High salary")
```

11.0.8 8. Email domain extractor

[ ]:
```python
email = input("Enter email: ")
if "@" in email:
    domain = email.split("@")[1]
    print(f"Domain: {domain}")
else:
    print("Invalid email")
```

12 Project:

12.0.1 Survey Quality Checker

12.0.2 Goal

Write a Python script that scans mock survey responses and **flags bad entries** based on: - Invalid emails - Age outside range (0–100) - Salary too low or negative

12.0.3 Sample Input (Mock Data)

```
survey_responses = [
    {"email": "john@example.com", "age": 28, "salary": 45000},
    {"email": "noatsymbol.com", "age": 45, "salary": 65000},
    {"email": "sara@domain.com", "age": -2, "salary": 50000},
    {"email": "mike@web.com", "age": 33, "salary": -15000},
]
```

12.0.4 Your Task

1. Loop through each dictionary in the list
2. Check for:
   - Valid email (@ and . present)
   - Age between 0 and 100
   - Salary >= 0
3. Print invalid entries with the reason

12.0.5 Starter Code

[ ]:
```python
def is_valid_email(email):
    return "@" in email and "." in email

def is_valid_age(age):
    return 0 <= age <= 100

def is_valid_salary(salary):
    return salary >= 0

# Data
survey_responses = [
    {"email": "john@example.com", "age": 28, "salary": 45000},
    {"email": "noatsymbol.com", "age": 45, "salary": 65000},
    {"email": "sara@domain.com", "age": -2, "salary": 50000},
    {"email": "mike@web.com", "age": 33, "salary": -15000},
]

# Check each entry
for response in survey_responses:
    errors = []
    if not is_valid_email(response["email"]):
        errors.append("Invalid email")
    if not is_valid_age(response["age"]):
        errors.append("Invalid age")
    if not is_valid_salary(response["salary"]):
        errors.append("Invalid salary")

    if errors:
        print(f"Bad entry: {response}")
        for err in errors:
            print(f"  {err}")
```

13 Stretch Goals

- Collect valid entries into a clean list
- Save invalid entries to a text file
- Build a function clean_survey_data() to reuse this logic

14 Week 3: Functions, Reusability & Error Handling

Topics:

- Defining functions (def)
- Parameters, return, default values
- Built-in functions vs user-defined
- Default arguments, scope

- Lambda expressions for quick filtering
- Error handling: Try/except, assert, basic logging

Exercise: * Reusable data cleaning and transformation functions

Project: * "Data Cleaning Toolbox" – Build reusable functions for tasks like removing whitespace, fixing dates, capitalizing names.

Learning Objectives

By the end of this week, you will be able to: - Define and use custom functions - Use parameters, return values, and default arguments - Understand scope and lambda functions - Handle runtime errors using try/except - Use assertions and basic logging

14.0.1 Define a Function

- Functions are blocks of reusable code.
- Defined using def keyword.
- Can take inputs (parameters) and return results.
- Built-in functions come with Python (e.g., print(), len()).
- User-defined functions are created by you.

[ ]:
```
def greet(name):
    return f"Hello, {name}!"
```

14.0.2 Parameters and Return

- Parameters are inputs to functions.
- return sends output back.
- Default values provide fallback inputs.

[ ]:
```
def add(a, b):
    return a + b

result = add(5, 3)  # 8
```

14.0.3 Default Arguments

- Default arguments let you call functions without always specifying all inputs.

[ ]:
```
def greet(name="Guest"):
    return f"Hello, {name}"
```

14.0.4 Reusability Example

[ ]:
```
def format_name(name):
    return name.strip().title()

def validate_age(age):
    return age >= 0 and age <= 120
```

14.0.5 Scope Basics

- Scope means where variables exist and can be accessed. ## Simple Analogy: Scope is like rooms in a house you can only use what's inside the room you're in. Variables inside a function are local; they don't affect code outside unless returned.

[ ]:
```
x = 5

def print_number():
    x = 10
    print(x)  # local x

print_number()
print(x)  # global x
```

14.0.6 Lambda Expressions

- Lambda expressions for quick filtering
- Small, one-line functions for quick jobs.
- Lambda creates anonymous functions.
- Useful for short, simple operations.

[ ]:
```
double = lambda x: x * 2
print(double(4))  # 8
```

14.0.7 Common use with filter() or map():

[ ]:
```
nums = [1, 2, 3, 4]
squared = list(map(lambda x: x**2, nums))
```

14.0.8 Error Handling

Making programs robust and handling errors gracefully.

- try/except to catch and handle errors without crashing.
- assert to check assumptions.
- Logging to record events/errors.

[ ]:
```
try:
    value = int(input("Enter a number: "))
except ValueError:
    print("That's not a number!")
```

14.0.9 Assert Statements

[ ]:
```
def divide(a, b):
    assert b != 0, "Division by zero!"
    return a / b
```

14.0.10 Basic Logging

```
[ ]: import logging
logging.basicConfig(level=logging.INFO)
logging.info("Process started")
```

14.0.11 Real-World Relevance

Functions are the backbone of clean, reusable code. Error handling is vital when working with unpredictable data (e.g., user input, CSV files, APIs).

15 Practice Exercises

15.0.1 1. Function to square a number

```
[ ]: def square(n):
    return n ** 2
```

15.0.2 2. Greet with default

```
[ ]: def greet(name="friend"):
    return f"Hello, {name}"
```

15.0.3 3. Convert Celsius to Fahrenheit

```
[ ]: def to_fahrenheit(celsius):
    return (celsius * 9/5) + 32
```

15.0.4 4. Safe Division with Try/Except

```
[ ]: def safe_divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        return "Cannot divide by zero"
```

15.0.5 5. Capitalize all names in a list

```
[ ]: names = ["maale", "DRANI", "Ollie"]

def format_names(name_list):
    return [name.title() for name in name_list]
```

15.0.6 6. Validate email with assert

[ ]:
```python
def check_email(email):
    assert "@" in email and "." in email, "Invalid email"
```

15.0.7 7. Use lambda to triple a number

[ ]:
```python
triple = lambda x: x * 3
```

15.0.8 8. Create logger that prints success message

[ ]:
```python
import logging
logging.basicConfig(level=logging.INFO)

def run_task():
    logging.info("Task completed!")
```

16 Exercises:

16.0.1 Reusable Data Cleaning Functions

16.0.2 Goal

Create small, reusable functions that clean common types of messy data.

16.0.3 Tasks

Write and test these functions:

- `clean_whitespace(text)`: removes leading/trailing spaces
- `fix_case(name)`: capitalizes names
- `remove_special_chars(text)`: removes symbols like `@#!`
- `validate_email(email)`: returns True/False

16.0.4 Sample Code

[ ]:
```python
import re

def clean_whitespace(text):
    return text.strip()

def fix_case(name):
    return name.title()

def remove_special_chars(text):
    return re.sub(r'[^a-zA-Z0-9\s]', '', text)

def validate_email(email):
    return "@" in email and "." in email
```

```
# Demo
dirty_name = "  mAaLe aUsTin@@ "
clean_name = fix_case(remove_special_chars(clean_whitespace(dirty_name)))

print("Cleaned Name:", clean_name)
```

17 Project:

17.0.1 Data Cleaning Toolbox

Goal

Build a mini Python library of functions for cleaning user input and textual data.

17.0.2 Requirements

1. Create at least 4 functions, e.g.:
   - clean_name()
   - validate_email()
   - clean_date_string()
   - fix_phone_format()
2. Accept raw data from user or dictionary
3. Return a cleaned version or report errors
4. Use try/except where needed
5. Use assert in at least one function

17.0.3 Sample Code

[ ]:
```
def clean_name(name):
    assert isinstance(name, str), "Name must be a string"
    return name.strip().title()

def validate_email(email):
    return "@" in email and "." in email

def clean_date_string(date_str):
    try:
        parts = date_str.split("/")
        if len(parts) == 3:
            return f"{parts[2]}-{parts[1].zfill(2)}-{parts[0].zfill(2)}"
        return "Invalid format"
    except Exception:
        return "Error parsing date"

def fix_phone(phone):
    digits = ''.join(filter(str.isdigit, phone))
    if len(digits) == 10:
```

```
        return f"+1-{digits[:3]}-{digits[3:6]}-{digits[6:]}"
    return "Invalid phone"

# Test
raw = {
    "name": "  mAaLe AuSTin ",
    "email": "MAALE@Example.COM",
    "dob": "12/5/1990",
    "phone": " (123) 456-7890 "
}

cleaned = {
    "name": clean_name(raw["name"]),
    "email": validate_email(raw["email"]),
    "dob": clean_date_string(raw["dob"]),
    "phone": fix_phone(raw["phone"])
}

print(cleaned)
```

17.0.4 Stretch Goals

- Wrap your functions into a Python module
- Handle missing keys with .get()
- Add logging to track processing

17.0.5 Week 4: Lists, Dictionaries & Working with Nested Data

Topics: * Data structures: list, dict, tuple, set * Indexing, slicing, updating * Nested structures and Iteration * List comprehensions (intro)

Exercises: * Frequency counters * Filtering and grouping

Project: * "Mini CRM System" – Store, search, and update mock customer records using nested dictionaries/lists.

Learning Objectives

By the end of this week, you will be able to: - Use Python's core data structures: `list`, `dict`, `tuple`, and `set` - Index, slice, update, and loop over these structures - Work with nested dictionaries/lists - Use list comprehensions for quick transformations

18 Data structures: list, dict, tuple, set

18.0.1 Storing multiple items efficiently.

Key Concepts:

- **List:** Ordered, changeable collection.
- **Dictionary:** Key-value pairs.
- **Tuple:** Ordered, unchangeable collection.

- **Set:** Unordered collection with no duplicates.
- **Indexing, slicing, updating**
  - Accessing and modifying data inside structures.
  - Indexing means picking one item by position.
  - Slicing picks a range.
  - Updating changes values.

19 Lists

- Ordered and changeable.
- Allows duplicate members.

19.0.1 Create and Access

[61]:
```
fruits = ["apple", "banana", "cherry"]
print(fruits[1])  # "banana"
print(fruits[-1])  # "cherry"
```

```
banana
cherry
```

19.0.2 Add

[62]:
```
fruits.append("orange")
fruits.insert(1, "orange")
tropical = ["mango", "pineapple", "papaya"]
fruits.extend(tropical)

print(fruits)
```

```
['apple', 'orange', 'banana', 'cherry', 'orange', 'mango', 'pineapple',
'papaya']
```

19.0.3 Remove, Slice

[64]:
```
fruits.remove("banana")
del fruits[0] # removes the specified index
del fruits # delete list completely
print(fruits[1:3])  # slice
print(fruits)
print('Last item in the list', fruits.pop()) # removes and return last element␣
 ↪"papaya"
fruits.clear() # empties the list
print(fruits)
```

```
['orange', 'banana']
['apple', 'orange', 'banana', 'cherry', 'orange', 'mango', 'pineapple']
Last item in the list pineapple
['apple', 'orange', 'banana', 'cherry', 'orange', 'mango']
```

19.0.4 Change a Range of Item Values

[43]:
```
# Change the values "banana" and "cherry" with the values "ovacado" and␣
  ↪"watermelon":
fruits[1:3] = ["ovacado", "watermelon"]
print(fruits)
```

```
['apple', 'ovacado', 'watermelon']
```

20 Dictionaries

- Ordered and changeable.
- No duplicate members ### Create and Access

[ ]:
```
person = {"name": "Drani", "age": 30}
print(person["name"])  # Drani
```

20.0.1 Add, Update, Delete

[ ]:
```
person["email"] = "drani@example.com"
person["age"] = 31
del person["email"]
```

20.0.2 Nested Structures

- Storing data inside data and looping through it.
- Lists/dicts can contain other lists/dicts.
- Use loops to go through nested data.

[ ]:
```
customers = [
    {"name": "Ollie", "email": "ollie@email.com"},
    {"name": "Austin", "email": "austin@email.com"}
]

for c in customers:
    print(c["name"], c["email"])
```

21 Tuple: Immutable

- Ordered and unchangeable.
- Allows duplicate members.

[ ]:
```
t = (1, 2, 3)
```

22 Set: Unique values

- Unordered, unchangeable, and unindexed.

- No duplicate members.

[ ]:
```
s = {1, 2, 2, 3}
print(s)  # {1, 2, 3}
```

23 List Comprehensions

- Shortcuts to create lists quickly.
- Compact syntax to build lists using loops and conditions.

[ ]:
```
numbers = [1, 2, 3, 4]
squared = [x**2 for x in numbers]
```

23.0.1 Real-World Relevance

Nested data is common in real-world datasets (e.g., JSON files, APIs, CRM systems). You'll use these patterns constantly in data cleaning, parsing, and storage tasks.

23.1 Practice Exercises

23.1.1 1. Create a list of 5 cities and print each

[ ]:
```
cities = ["Kampala", "Nairobi", "Tokyo", "New York", "lagos"]
for city in cities:
    print(city)
```

23.1.2 2. Get the last 3 elements from a list

[ ]:
```
nums = [10, 20, 30, 40, 50]
print(nums[-3:])
```

23.1.3 3. Create a dictionary of a product

[ ]:
```
product = {"name": "Laptop", "price": 999.99, "stock": 20}
print(product["price"])
```

23.1.4 4. Loop through dictionary keys and values

[ ]:
```
for key, value in product.items():
    print(f"{key}: {value}")
```

23.1.5 5. Count how many times a word appears in a list

[ ]:
```
words = ["yes", "no", "yes", "maybe", "yes"]
print(words.count("yes"))
```

23.1.6 6. Use list comprehension to filter values > 10

[ ]:
```
nums = [3, 12, 7, 25, 5]
filtered = [x for x in nums if x > 10]
```

23.1.7 7. Combine two dictionaries

[1]:
```
a = {"x": 1}
b = {"y": 2}
merged = {**a, **b}
```

23.1.8 8. Create and access nested dictionary

[ ]:
```
users = {
    "user1": {"name": "Maale", "email": "maale@email.com"},
    "user2": {"name": "Ollie", "email": "ollie@email.com"}
}
print(users["user2"]["email"])
```

23.1.9 9. Add a new customer to a list of dicts

[ ]:
```
customers = []
new_customer = {"name": "Drani", "email": "drani@email.com"}
customers.append(new_customer)
```

23.1.10 10. Create a frequency counter (manual)

[ ]:
```
nums = [1, 2, 2, 3, 3, 3]
counter = {}
for n in nums:
    counter[n] = counter.get(n, 0) + 1
print(counter)
```

24 Exercises

24.0.1 Frequency Counter

[ ]:
```
text = "banana"
counter = {}

for char in text:
    counter[char] = counter.get(char, 0) + 1

print(counter)
# Output: {'b': 1, 'a': 3, 'n': 2}
```

24.0.2 Filter & Group Names by First Letter

[32]:
```python
names = ["Maale", "Mai", "Ollie", "Asanti", "Aaron"]
grouped = {}

for name in names:
    key = name[0].upper()
    grouped.setdefault(key, []).append(name)

print(grouped)
# Output: {'M': ['Maale', 'Mai'], 'O': ['Ollie'], 'A': ['Asanti', 'Aaron']}
```

```
{'M': ['Maale', 'Mai'], 'O': ['Ollie'], 'A': ['Asanti', 'Aaron']}
```

25 Project:

25.0.1 Mini CRM System

Goal

Build a simple command-line customer tracking system using nested dictionaries/lists.

25.0.2 Features to Implement

- Add a new customer (name, email, phone)
- Search customer by name
- Update customer email or phone
- Delete a customer by name
- List all customers

25.0.3 Starter Code

[ ]:
```python
crm = []

def add_customer(name, email, phone):
    crm.append({"name": name, "email": email, "phone": phone})

def find_customer(name):
    for c in crm:
        if c["name"].lower() == name.lower():
            return c
    return None

def update_customer(name, email=None, phone=None):
    c = find_customer(name)
    if c:
        if email:
            c["email"] = email
        if phone:
```

```
            c["phone"] = phone
        return True
    return False

def delete_customer(name):
    global crm
    crm = [c for c in crm if c["name"].lower() != name.lower()]

def list_customers():
    for c in crm:
        print(f"{c['name']} - {c['email']} - {c['phone']}")

# Example Usage
add_customer("Maale", "maale@email.com", "1234567890")
add_customer("Ollie", "ollie@email.com", "9876543210")
list_customers()
```

25.0.4 Stretch Goals

Add command-line menu to interact with system

Save/load data from a .json file

Validate email and phone formats

---

26 Phase 2: Data Analytics With Python

(Weeks 5–10) - Hands-on with real-World Data Workflows

26.0.1 Week 5: File I/O and Data Ingestion

Topics: * Reading/writing: .txt, .csv using open(), csv module * Working with file paths * JSON parsing with json module * Basic exception handling for file errors

Project: "Sales Data Summary" - Load CSV and calculate totals, averages, handle missing/error rows.

Learning Objectives By the end of this week, you will be able to: - Read from and write to `.txt`, `.csv`, and `.json` files - Use the built-in `open()`, `csv`, and `json` modules - Handle file paths properly - Apply basic exception handling for file errors

26.0.2 Reading & Writing Text Files

Computers store data in files. To work with data in Python, we need to load it from these files into our programs.

You can read from or write to files using Python's built-in functions like open() or using modules like csv.

There are four different methods (modes) for opening a file:

- **“r”** - Read - Default value. Opens a file for reading, error if the file does not exist
- **“a”** - Append - Opens a file for appending, creates the file if it does not exist
- **“w”** - Write - Opens a file for writing, creates the file if it does not exist
- **“x”** - Create - Creates the specified file, returns an error if the file exists
- **“t”** - Text - Default value. Text mode
- **“b”** - Binary - Binary mode (e.g. images)

[ ]:
```python
# Write
with open("notes.txt", "w") as f:
    f.write("This is a line of text.")

# Read
with open("notes.txt", "r") as f:
    content = f.read()
    print(content)
```

26.0.3 Working with CSV Files

- CSV (Comma-Separated Values) files are common for storing spreadsheet-style data.
- Think of a CSV like a digital version of an Excel sheet, where each row is a record and each column is a variable.

[ ]:
```python
import csv

# Writing CSV
with open("data.csv", "w", newline="") as file:
    writer = csv.writer(file)
    writer.writerow(["Name", "Age"])
    writer.writerow(["Maale", 30])

# Reading CSV
with open("data.csv", "r") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

26.0.4 Reading JSON Data

- JSON is a data format used for structured data.
- Python’s json module converts JSON data to Python dictionaries and back.

[ ]:
```python
import json

# Save JSON
data = {"name": "Maale", "age": 30}
with open("data.json", "w") as f:
    json.dump(data, f)
```

```python
# Load JSON
with open("data.json", "r") as f:
    loaded = json.load(f)
    print(loaded["name"])
```

26.0.5 Handling File Paths

[ ]:
```python
import os

folder = "data"
filename = "example.txt"
path = os.path.join(folder, filename)
print(path)  # 'data/example.txt'
```

26.0.6 File Error Handling

- When working with files, things can go wrong the file might not exist, or be unreadable.
- Using try/except lets your program fail gracefully:

[ ]:
```python
try:
    with open("missing.txt", "r") as f:
        content = f.read()
except FileNotFoundError:
    print("File not found!")
```

26.0.7 Real-World Relevance

Reading data from files is the first step in nearly every data project. You'll handle messy CSV exports, logs, and JSON from APIs all the time.

27 Practice Exercises

27.1 1. Write and read a short note to a text file

[ ]:
```python
with open("note.txt", "w") as f:
    f.write("Don't forget to check the data.")

with open("note.txt", "r") as f:
    print(f.read())
```

27.1.1 2. Append to a file

[ ]:
```python
with open("log.txt", "a") as f:
    f.write("New log entry\n")
```

27.1.2 3. Count number of lines in a text file

[ ]:
```
with open("note.txt") as f:
    lines = f.readlines()
    print(len(lines))
```

27.1.3 4. Write a list of names to a CSV file

[ ]:
```
names = [["Name"], ["Maale"], ["Austin"]]
with open("names.csv", "w", newline="") as f:
    writer = csv.writer(f)
    writer.writerows(names)
```

27.1.4 5. Read and print names from CSV

[ ]:
```
with open("names.csv", "r") as f:
    reader = csv.reader(f)
    for row in reader:
        print(row[0])
```

27.1.5 6. Save a dictionary as JSON

[ ]:
```
profile = {"user": "Maale", "age": 29}
with open("profile.json", "w") as f:
    json.dump(profile, f)
```

27.1.6 7. Load and print data from JSON

[ ]:
```
with open("profile.json") as f:
    data = json.load(f)
    print(data["user"])
```

27.1.7 8. Gracefully handle missing file

[ ]:
```
try:
    with open("ghost.txt") as f:
        print(f.read())
except FileNotFoundError:
    print("File doesn't exist.")
```

27.1.8 9. Create a file path from folder + file

[ ]:
```
path = os.path.join("data", "sales.csv")
```

28 Project:

28.0.1 Sales Data Summary

28.0.2 Goal

Load a CSV file of sales transactions and generate a basic report including: - Total transactions - Total revenue - Average sale - Flag rows with missing or invalid data

28.0.3 Example CSV: `sales.csv`

```
Date,Product,Quantity,Price
2023-01-01,Phone,2,699.99
2023-01-02,Laptop,1,1099.99
2023-01-03,Tablet,,399.99
2023-01-04,Monitor,1,abc
```

28.0.4 Your Tasks

- Read the CSV file
- Skip header
- Validate that `Quantity` and `Price` are numeric
- Calculate:
  - `total sales` = sum of `Quantity * Price`
  - `average sale`
- Print invalid rows with reasons

28.0.5 Starter Code

[ ]:
```python
import csv

total_sales = 0
valid_rows = 0
invalid_rows = []

with open("sales.csv", "r") as f:
    reader = csv.DictReader(f)
    for row in reader:
        try:
            qty = int(row["Quantity"])
            price = float(row["Price"])
            total_sales += qty * price
            valid_rows += 1
        except (ValueError, TypeError):
            invalid_rows.append(row)

print(f"Total Valid Transactions: {valid_rows}")
print(f"Total Revenue: ${total_sales:.2f}")
print(f"Average Sale: ${total_sales / valid_rows:.2f}")
```

```
print("\nInvalid Rows:")
for row in invalid_rows:
    print(row)
```

[ ]:
```
Total Valid Transactions: 2
Total Revenue: $2499.97
Average Sale: $1249.99

Invalid Rows:
{'Date': '2023-01-03', 'Product': 'Tablet', 'Quantity': '', 'Price': '399.99'}
{'Date': '2023-01-04', 'Product': 'Monitor', 'Quantity': '1', 'Price': 'abc'}
```

28.0.6 Stretch Goals

Add date range filter

Write summary to a text or JSON report

Save invalid rows to a separate CSV

29 Week 6: NumPy for Efficient Numerical Computing

Topics: * Why Numpy: performance vs Lists * Arrays, slicing, reshaping * Element-wise operations and broadcasting * Aggregation: sum(), mean(), std() * Boolean indexing for filtering

Project:

- Analyze a small dataset using Numpy arrays for transformation and filtering

Learning Objectives By the end of this week, you will be able to: - Understand why NumPy is used in data analysis - Create and manipulate NumPy arrays - Use slicing, reshaping, and broadcasting - Perform efficient numerical operations and filtering - Apply aggregation functions for insights

29.0.1 Why NumPy?

- Lists are flexible but **slow** for numerical tasks
- NumPy uses fixed-type arrays in **C-level memory** = blazing fast
- Base for **Pandas**, **Scikit-learn**, **TensorFlow**, etc.

29.0.2 Creating Arrays

- Arrays are like grid-shaped containers that store numbers.
- You can cut arrays into pieces (slicing) or change their shape (e.g., from rows to columns).

arr[1:3] gives part of the array.

reshape(3, 2) changes a flat array into 3 rows, 2 columns.

[ ]:
```
import numpy as np

a = np.array([1, 2, 3])
```

```
b = np.array([[1, 2], [3, 4]])
```

29.0.3 Array Shapes & Types

[ ]:
```
print(a.shape)       # (3,)
print(b.shape)       # (2, 2)
print(a.dtype)       # int64
```

29.0.4 Reshaping Arrays

[ ]:
```
x = np.array([1, 2, 3, 4, 5, 6])
x = x.reshape((2, 3))  # 2 rows, 3 columns
```

29.0.5 Slicing & Indexing

[ ]:
```
a = np.array([[10, 20], [30, 40]])
print(a[0, 1])     # 20
print(a[:, 1])     # column 1
```

29.0.6 Vectorized Operations

[ ]:
```
x = np.array([1, 2, 3])
y = np.array([10, 20, 30])
print(x + y)       # [11, 22, 33]
print(x * 2)       # [2, 4, 6]
```

29.0.7 Aggregation

[ ]:
```
a = np.array([[1, 2], [3, 4]])
print(np.sum(a))         # 10
print(np.mean(a))        # 2.5
print(np.std(a))         # 1.118
```

29.0.8 Boolean Indexing

[ ]:
```
x = np.array([5, 10, 15, 20])
mask = x > 10
print(x[mask])           # [15 20]
```

29.0.9 Real-World Relevance

NumPy makes bulk computation fast and memory-efficient — especially important in large datasets, feature engineering, matrix math (ML), and statistical analysis.

30 Practice Exercises

30.0.1 1. Import numpy and create 1D array

[ ]:
```
import numpy as np
a = np.array([1, 2, 3, 4])
```

30.0.2 2. Create a 3x2 array of zeros

[ ]:
```
z = np.zeros((3, 2))
```

30.0.3 3. Multiply all elements by 10

[ ]:
```
a = np.array([1, 2, 3])
print(a * 10)
```

30.0.4 4. Reshape a flat array into 2x3

[ ]:
```
x = np.array([1, 2, 3, 4, 5, 6])
x = x.reshape((2, 3))
```

30.0.5 5. Get mean and std of array

[ ]:
```
x = np.array([10, 20, 30])
print(np.mean(x), np.std(x))
```

30.0.6 6. Slice a 2D array to get column 2

[ ]:
```
a = np.array([[1, 2], [3, 4], [5, 6]])
print(a[:, 1])
```

30.0.7 7. Filter values greater than 50

[ ]:
```
data = np.array([45, 55, 60, 40])
print(data[data > 50])
```

30.0.8 8. Create a range of even numbers

[ ]:
```
evens = np.arange(0, 20, 2)
```

30.0.9 9. Check shape and type

[ ]:
```
x = np.array([[1, 2], [3, 4]])
print(x.shape, x.dtype)
```

30.0.10 10. Add two arrays

[ ]:

30.1 Project:

30.1.1 Analyze a Small Dataset with NumPy

30.1.2 Goal

Use NumPy to: - Load numeric dataset (e.g., grades, sales, scores) - Perform analysis: total, mean, std, filtering - Print summary insights

30.1.3 Dataset: `scores.csv`

```
Name,Math,Science,English
Alice,85,90,88
Bob,70,75,72
Charlie,95,98,94
```

30.1.4 Tasks

- Load CSV (skip headers, use `np.loadtxt()` or `csv`)
- Extract numerical values
- Calculate:
  - Mean score per student
  - Mean score per subject
  - Std deviation overall
- Filter students with any score < 75

30.1.5 Example Code

[ ]:
```python
import numpy as np
import csv

# Load manually to skip names
scores = []

with open("scores.csv", "r") as f:
    reader = csv.reader(f)
    next(reader)  # skip header
    for row in reader:
        scores.append([int(row[1]), int(row[2]), int(row[3])])

arr = np.array(scores)

# Per-student average
student_means = np.mean(arr, axis=1)
print("Student Averages:", student_means)
```

```python
# Per-subject average
subject_means = np.mean(arr, axis=0)
print("Subject Averages:", subject_means)

# Overall stats
print("Overall Std Dev:", np.std(arr))

# Filter students with low scores
print("Any score < 75:")
print(arr[np.any(arr < 75, axis=1)])
```

[ ]:
```
Student Averages: [87.666 72.333 95.666]
Subject Averages: [83.33 87.66 84.66]
Overall Std Dev: 9.73
Any score < 75:
[[70 75 72]]
```

30.1.6 Stretch Goals

Normalize all scores (e.g., to 0–1 scale) Add names and link stats back to each student Plot histograms using matplotlib (next week)

30.1.7 Week 7: Pandas Fundamentals

Topics: * DataFrame vs Series * Loading CSV/Excel data * Exploring datasets: .head( ), .info( ), .describe( ) * Selecting/filtering with .loc[ ], .iloc[ ]

Min project: * Filter high-value transactions or recent records

Main Project: * "HR Analytics Explorer" – Analyze headcount, attrition rate, salary bands

Learning Objectives

By the end of this week, you will be able to: - Load CSV/Excel files into Pandas DataFrames - Understand the difference between Series and DataFrames - View and explore datasets (.head(), .info(), .describe()) - Select rows/columns with .loc[], .iloc[], and filters - Apply basic filters to isolate key records

30.1.8 What is Pandas?

- High-level data analysis library
- Built on top of NumPy
- Supports table-like data (like Excel or SQL)
- Core object: `DataFrame`

30.1.9 DataFrame vs Series

- DataFrame: Think of it as an Excel sheet — rows and columns.
- Series: A single column of data (like a list with labels).

[ ]:
```python
import pandas as pd

# Series: 1D
s = pd.Series([1, 2, 3])

# DataFrame: 2D table
df = pd.DataFrame({
    "Name": ["Alice", "Bob"],
    "Age": [25, 30]
})
```

30.1.10 Loading Data

You can load data from CSV, Excel, SQL, etc.

[ ]:
```python
df = pd.read_csv("sales.csv")
df = pd.read_excel("data.xlsx")  # needs openpyxl
```

30.1.11 Exploring Data

First step in data analysis is understanding what you're working with:

.head() shows the first few rows

.info() shows column names, types, and missing data

.describe() gives stats like mean, min, max, etc.

[ ]:
```python
df.head()       # first 5 rows
df.info()       # column types & nulls
df.describe()   # basic stats
```

30.1.12 Selecting Data

[ ]:
```python
# Columns
df["Age"]
df[["Name", "Age"]]

# Rows by index
df.loc[0]           # by label
df.iloc[0]          # by position

# Filter rows
df[df["Age"] > 30]
```

30.1.13 Common Operations

```
[ ]: df["Salary"].mean()
     df["City"].value_counts()
     df.sort_values("Salary", ascending=False)
```

30.1.14 Real-World Relevance

Pandas is your go-to tool for 80% of data wrangling tasks — Excel replacement, data cleaning, quick analysis, and prep for ML.

31 Practice Exercises

31.0.1 1. Import pandas and read a CSV

```
[ ]: import pandas as pd
     df = pd.read_csv("employees.csv")
```

31.0.2 2. View top and bottom rows

```
[ ]: print(df.head())
     print(df.tail())
```

31.0.3 3. Check number of rows and columns

```
[ ]: print(df.shape)
```

31.0.4 4. Get column names

```
[ ]: print(df.columns)
```

31.0.5 5. Summary statistics

```
[ ]: print(df.describe())
```

31.0.6 6. Get all rows where salary > 50,000

```
[ ]: df[df["Salary"] > 50000]
```

31.0.7 7. Select only Name and Department columns

```
[ ]: df[["Name", "Department"]]
```

31.0.8 8. Sort by salary descending

```
[ ]: df.sort_values("Salary", ascending=False)
```

31.0.9 9. Count unique departments

[ ]:
```
df["Department"].value_counts()
```

31.0.10 10. Get row by index

[ ]:
```
df.iloc[2]  # 3rd row
```

32 Project:

32.0.1 HR Analytics Explorer

32.0.2 Goal

Analyze a company's employee dataset to answer key HR questions: - What's the average salary by department? - Which departments have the most headcount? - What's the attrition rate?

32.0.3 Sample Dataset: `hr_data.csv`

```
EmployeeID,Name,Department,Salary,Status
1,Alice,Sales,70000,Active
2,Bob,Engineering,90000,Active
3,Charlie,Sales,65000,Left
4,Dana,Engineering,85000,Active
5,Eli,HR,60000,Left
6,Faith,Sales,72000,Active
```

32.0.4 Tasks

1. Load CSV into DataFrame
2. Use `.groupby()` to get:
   - Avg salary by department
   - Headcount by department
   - Count of "Left" vs "Active"
3. Use `.loc[]` to filter all employees who left
4. Sort by salary
5. Plot value counts (optional, next week)

32.0.5 Starter Code

[ ]:
```
import pandas as pd

df = pd.read_csv("hr_data.csv")

# Average salary per department
avg_salary = df.groupby("Department")["Salary"].mean()
print("Average Salary:\n", avg_salary)

# Headcount by department
```

```
headcount = df["Department"].value_counts()
print("Headcount:\n", headcount)

# Attrition counts
status_counts = df["Status"].value_counts()
print("Status Counts:\n", status_counts)

# List employees who left
left = df[df["Status"] == "Left"]
print("Employees who left:\n", left[["Name", "Department"]])
```

[ ]:
```
Average Salary:
Department
Engineering    87500.0
HR             60000.0
Sales          69000.0

Headcount:
Sales          3
Engineering    2
HR             1

Status Counts:
Active    4
Left      2
```

32.0.6 Stretch Goals

Create a new column: "Seniority Level" based on salary

Save summaries to a new Excel or CSV file

Combine this with data from another department next week

33 Week 8: Data Cleaning with Pandas

Topics: * Handling nulls: isna( ), fillna( ) * String operations: .str.lower( ), .strip( ), .replace( ) * Date parsing and type conversion * Renaming columns, dropping duplicates

Project: * "CRM Export Cleaner" – Clean a messy dataset for sales or marketing team

Learning Objectives

By the end of this week, you will be able to: - Identify and handle missing or duplicate data - Clean and standardize string values - Convert columns to proper data types (e.g., dates, numbers) - Rename columns, drop irrelevant data - Prepare raw exports for analysis or business use

33.0.1 Why Cleaning Matters

- Raw data is messy: typos, empty cells, bad formats

- Cleaning is 60–80% of real-world data work
- Clean data = accurate insights

33.1 Handling Missing Values

```
[ ]: df.isna().sum()                # Count nulls
     df.dropna()                    # Remove rows with nulls
     df.fillna(0)                   # Fill nulls with 0
     df["col"].fillna("Unknown")    # Fill specific column
```

33.1.1 String Cleaning (with .str)

```
[ ]: df["Name"] = df["Name"].str.strip()         # Remove spaces
     df["Name"] = df["Name"].str.title()         # Capitalize
     df["Email"] = df["Email"].str.lower()       # Lowercase
     df["Phone"] = df["Phone"].str.replace("-", "")
```

33.1.2 Data Type Conversion

```
[ ]: df["Salary"] = pd.to_numeric(df["Salary"], errors="coerce")
     df["JoinDate"] = pd.to_datetime(df["JoinDate"], errors="coerce")
```

33.1.3 Renaming & Dropping

```
[ ]: df = df.rename(columns={"emp_id": "EmployeeID"})
     df = df.drop(columns=["TempNotes"])
```

33.1.4 Dealing with Duplicates

```
[ ]: df.duplicated().sum()
     df = df.drop_duplicates()
```

33.1.5 Tip: Chain Methods

```
[ ]: df["Name"] = df["Name"].str.strip().str.title()
```

33.1.6 Real-World Relevance

You'll clean CRM exports, HR spreadsheets, survey data, and scraped data all the time. It's not glamorous — but it's mission critical.

33.1.7 Practice Exercises

33.2 1. Fill null salaries with 0

```
[ ]: df["Salary"] = df["Salary"].fillna(0)
```

33.2.1 2. Drop all rows with any nulls

```
[ ]: df = df.dropna()
```

33.2.2 3. Strip spaces from name field

```
[ ]: df["Name"] = df["Name"].str.strip()
```

33.2.3 4. Convert JoinDate to datetime

```
[ ]: df["JoinDate"] = pd.to_datetime(df["JoinDate"])
```

33.2.4 5. Capitalize names

```
[ ]: df["Name"] = df["Name"].str.title()
```

33.2.5 6. Lowercase all emails

```
[ ]: df["Email"] = df["Email"].str.lower()
```

33.2.6 7. Remove duplicates

```
[ ]: df = df.drop_duplicates()
```

33.2.7 8. Rename column “emp_id” to “EmployeeID”

```
[ ]: df = df.rename(columns={"emp_id": "EmployeeID"})
```

33.2.8 9. Remove column “Notes”

```
[ ]: df = df.drop(columns=["Notes"])
```

33.2.9 10. Fill missing values in “Department” with “Unknown”

```
[ ]: df["Department"] = df["Department"].fillna("Unknown")
```

34 Project:

34.0.1 CRM Export Cleaner

34.0.2 Goal

Take a messy customer export and clean it up for the marketing/sales team: - Standardize names, emails, and phone numbers - Remove duplicates - Convert types - Handle missing values

34.0.3 Sample Dataset: `crm_data.csv`

```
Name,Email,Phone,SignupDate,Status
alice ,ALICE@EXAMPLE.COM, 123-456-7890,2023/01/10,Active
BOB,BOB@EMAIL.COM,1234567890,2023-02-12,Active
alice,Alice@example.com,1234567890,2023-01-10,Active
Charlie,,123.456.7890,,Inactive
```

34.0.4 Tasks

1. Remove duplicates (same Name + Phone)
2. Strip/standardize Name and Email
3. Convert phone numbers to consistent format (remove punctuation)
4. Convert `SignupDate` to datetime
5. Fill missing SignupDate with default
6. Drop rows with missing emails
7. Final cleaned dataset saved to `clean_crm.csv`

34.0.5 Starter Code

[ ]:
```python
import pandas as pd

df = pd.read_csv("crm_data.csv")

# Strip and standardize strings
df["Name"] = df["Name"].str.strip().str.title()
df["Email"] = df["Email"].str.strip().str.lower()
df["Phone"] = df["Phone"].str.replace(r"\D", "", regex=True)

# Remove duplicates (same Name + Phone)
df = df.drop_duplicates(subset=["Name", "Phone"])

# Convert date
df["SignupDate"] = pd.to_datetime(df["SignupDate"], errors="coerce")

# Fill missing dates
df["SignupDate"] = df["SignupDate"].fillna(pd.Timestamp("2023-01-01"))

# Drop rows with missing email
df = df.dropna(subset=["Email"])

# Save cleaned version
df.to_csv("clean_crm.csv", index=False)

print(df)
```

34.0.6 Sample Output

[ ]:
```
      Name              Email        Phone SignupDate   Status
0    Alice  alice@example.com   1234567890  2023-01-10   Active
1      Bob      bob@email.com   1234567890  2023-02-12   Active
3  Charlie  charlie@email.com  1234567890  2023-01-01  Inactive
```

34.0.7 Stretch Goals

Validate emails using regex

Add a "SignupMonth" column

Save final summary to Excel

35 Week 9: Data Transformation & Feature Engineering

Topics: * Creating new columns with apply() & lambda * Binning & categorization (pd.cut(), qcut()) * Dummy variables (get_dummies) * Mapping values (e.g., scoring systems)

Project: * Customer segmentation or performance tiering (e.g., "Gold", "Silver", "Bronze")

Learning Objectives

By the end of this week, you will be able to: - Create new columns using `apply()` and `lambda` - Perform binning and categorization (e.g., age groups, spend tiers) - Convert categorical variables using dummy/one-hot encoding - Map and transform column values with dictionaries - Prepare datasets for downstream modeling or analysis

35.1 Creating New Columns

[ ]:
```
# Using arithmetic
df["Revenue"] = df["Price"] * df["Quantity"]

# Using apply + lambda
df["FullName"] = df["First"] + " " + df["Last"]
df["Tax"] = df["Price"].apply(lambda x: x * 0.07)
```

35.1.1 Feature Binning with pd.cut() or pd.qcut()

[ ]:
```
# Fixed bins
df["AgeGroup"] = pd.cut(df["Age"], bins=[0, 18, 35, 60, 100], labels=["Teen",␣
↪"Young Adult", "Adult", "Senior"])

# Quantile-based bins
df["SpendingTier"] = pd.qcut(df["TotalSpent"], q=3, labels=["Low", "Medium",␣
↪"High"])
```

35.1.2 Mapping Categorical Values

```
[ ]: rating_map = {"Poor": 1, "Average": 2, "Good": 3, "Excellent": 4}
df["RatingScore"] = df["Rating"].map(rating_map)
```

35.1.3 One-Hot Encoding (Dummy Variables)

```
[ ]: pd.get_dummies(df["Department"])

# Or include in main DataFrame
df = pd.get_dummies(df, columns=["Department"])
```

35.1.4 Why Feature Engineering?

Key step before modeling or deeper analysis

Allows you to capture patterns (segments, tiers, scores)

Helps convert messy real-world data into structured features

35.1.5 Tip: Avoid modifying original column unless intended

```
[ ]: # Don't overwrite your raw data unless you're sure
df["Normalized"] = df["Score"] / df["Score"].max()
```

36 Practice Exercises

36.0.1 1. Create "Revenue" = Price × Quantity

```
[ ]: df["Revenue"] = df["Price"] * df["Quantity"]
```

36.0.2 2. Combine First and Last Name into FullName

```
[ ]: df["FullName"] = df["First"] + " " + df["Last"]
```

36.0.3 3. Calculate DiscountedPrice = Price * 0.9

```
[ ]: df["DiscountedPrice"] = df["Price"].apply(lambda x: x * 0.9)
```

36.0.4 4. Create Age Groups with cut()

```
[ ]: df["AgeGroup"] = pd.cut(df["Age"], [0, 18, 35, 60, 100], labels=["Teen", "Young␣
  ↪Adult", "Adult", "Senior"])
```

36.0.5 5. Score performance using map()

```
[ ]: map_dict = {"Low": 1, "Medium": 2, "High": 3}
     df["Score"] = df["Performance"].map(map_dict)
```

36.0.6 6. Convert "Category" to dummy variables

```
[ ]: pd.get_dummies(df["Category"])
```

36.0.7 7. Normalize a column between 0 and 1

```
[ ]: df["Normalized"] = df["Score"] / df["Score"].max()
```

36.0.8 8. Add "Spending Tier" using qcut()

```
[ ]: df["Tier"] = pd.qcut(df["Spending"], 3, labels=["Low", "Medium", "High"])
```

36.0.9 9. Extract domain from email

```
[ ]: df["Domain"] = df["Email"].apply(lambda x: x.split("@")[1])
```

36.0.10 10. Create "IsHighValue" if Revenue > 1000

```
[ ]: df["IsHighValue"] = df["Revenue"] > 1000
```

36.1 Project: Customer Segmentation & Tiering

36.1.1 Goal

Transform raw customer purchase data into **meaningful segments**: - Calculate total and average spend - Categorize customers into tiers - Add flags for high-value behavior

36.1.2 Sample Dataset: `customers.csv`

```
CustomerID,Name,Age,TotalSpent,Transactions
101,Alice,34,1200.50,12
102,Bob,22,450.00,5
103,Charlie,45,3000.00,18
104,Dana,61,980.00,4
105,Eli,28,1750.00,8
```

36.1.3 Tasks

1. Create `AvgSpend` = TotalSpent / Transactions
2. Categorize customers into:
   - Spending Tiers using `pd.qcut()`
   - Age Groups using `pd.cut()`
3. Create binary flag: `IsHighValue` if TotalSpent > 1000 and Transactions > 10
4. One-hot encode age groups (optional)

36.1.4 Starter Code

[ ]:
```
import pandas as pd

df = pd.read_csv("customers.csv")

# Average Spend
df["AvgSpend"] = df["TotalSpent"] / df["Transactions"]

# Spending Tier (3 equal-sized groups)
df["SpendingTier"] = pd.qcut(df["TotalSpent"], 3, labels=["Low", "Medium",␣
↪"High"])

# Age Group
df["AgeGroup"] = pd.cut(df["Age"], bins=[0, 25, 45, 100], labels=["Young",␣
↪"Middle", "Senior"])

# High Value Flag
df["IsHighValue"] = (df["TotalSpent"] > 1000) & (df["Transactions"] > 10)

print(df)
```

36.1.5 Sample Output

[ ]:
```
   Name   Age  TotalSpent  Transactions  AvgSpend SpendingTier AgeGroup ␣
↪IsHighValue
0 Alice    34      1200.5            12      100.0       Medium   Middle       ␣
↪True
1 Bob      22       450.0             5       90.0          Low    Young       ␣
↪False
2 Charlie 45      3000.0            18      166.6         High   Middle       ␣
↪True
3 Dana     61       980.0             4      245.0          Low   Senior       ␣
↪False
4 Eli      28      1750.0             8      218.7       Medium   Middle       ␣
↪False
```

36.1.6 Stretch Goals

Add normalized “AvgSpend”

Create cohort labels (e.g. Young-HighSpender)

Export tier summary by group to Excel or PDF

37 Week 10: Aggregation, Grouping, and Pivoting

Topics: * Grouping with .groupby() and .agg() * Pivot tables with .pivot_table() * Sorting and filtering summaries * Multi-level aggregation

Project: * "Revenue by Product Category" – Analyze revenue trends and generate pivot tables by region/month

Learning Objectives

By the end of this week, you will be able to: - Use `.groupby()` and `.agg()` to summarize data - Perform multi-level groupings - Create pivot tables using `.pivot_table()` - Sort and filter summary tables - Apply business thinking to summarize KPIs

37.1 Grouping Data

[ ]:
```
df.groupby("Department")["Salary"].mean()
df.groupby("Region")[["Sales", "Profit"]].sum()
```

37.1.1 Aggregation with .agg()

[ ]:
```
df.groupby("Region").agg({
    "Sales": "sum",
    "Profit": "mean",
    "Discount": "max"
})
```

Use a dict to assign different aggregation methods

More flexible than just .mean() or .sum()

37.1.2 Multi-Level Grouping

[ ]:
```
df.groupby(["Region", "Category"])["Sales"].sum()
```

Group by multiple features for a detailed view Returns a MultiIndex DataFrame

37.1.3 Pivot Tables with .pivot_table()

[ ]:
```
df.pivot_table(index="Region", columns="Category", values="Sales",␣
↪aggfunc="sum")
```

Like Excel pivots — reshape and summarize data aggfunc can be sum, mean, count, etc.

37.1.4 Sorting Summaries

[ ]:
```
summary = df.groupby("Category")["Sales"].sum()
summary.sort_values(ascending=False)
```

37.1.5 Use .sort_values() to highlight top performers

37.1.6 When to Use What?

Task	Method
Grouping + Summary	`.groupby()`
Summary Across 2 Dimensions	`.pivot_table()`
Aggregation w/ multiple funcs	`.agg()`
Ranking results	`.sort_values()`

37.1.7 Real-World Relevance

Used in:

Sales analysis by region/month

HR analytics by department/gender

Marketing conversion by channel/campaign

38 Practice Exercises

38.0.1 1. Group by Department and get average salary

[ ]:
```
df.groupby("Department")["Salary"].mean()
```

38.0.2 2. Group by Gender and get employee count

[ ]:
```
df.groupby("Gender")["EmployeeID"].count()
```

38.0.3 3. Sum sales by Region and Category

[ ]:
```
df.groupby(["Region", "Category"])["Sales"].sum()
```

38.0.4 4. Aggregate sales (sum) and profit (mean) by Category

[ ]:
```
df.groupby("Category").agg({
    "Sales": "sum",
    "Profit": "mean"
})
```

38.0.5 5. Create a pivot table for total sales by Region and Month

[ ]:
```
df.pivot_table(index="Region", columns="Month", values="Sales", aggfunc="sum")
```

38.0.6 6. Sort departments by total salary spent

```
[ ]: df.groupby("Department")["Salary"].sum().sort_values(ascending=False)
```

38.0.7 7. Count employees per region and gender

```
[ ]: df.groupby(["Region", "Gender"])["EmployeeID"].count()
```

38.0.8 8. Create pivot table with average salary per Department and Gender

```
[ ]: df.pivot_table(index="Department", columns="Gender", values="Salary",␣
     ↪aggfunc="mean")
```

38.0.9 9. Show top 3 categories by total revenue

```
[ ]: df.groupby("Category")["Revenue"].sum().sort_values(ascending=False).head(3)
```

38.0.10 10. Add new column "Revenue" = Price × Quantity, then group by Product

```
[ ]: df["Revenue"] = df["Price"] * df["Quantity"]
     df.groupby("Product")["Revenue"].sum()
```

39 Project:

39.0.1 Revenue by Product Category

39.0.2 Goal

Use grouping and pivoting to analyze **sales performance** across products, categories, and regions.

39.0.3 Sample Dataset: `sales_data.csv`

```
OrderID,Region,Category,Product,Price,Quantity,OrderDate
1,West,Office Supplies,Pens,1.5,20,2023-01-12
2,East,Furniture,Chair,85,2,2023-01-14
3,South,Technology,Laptop,950,1,2023-02-02
4,West,Furniture,Table,200,1,2023-02-11
5,East,Office Supplies,Stapler,7.5,10,2023-03-05
```

39.0.4 Tasks

1. Create `Revenue` = Price × Quantity
2. Group by `Category` and `Region` to get:
   - Total Revenue
   - Average Quantity per sale
3. Create pivot table: Revenue by Category vs Region
4. Extract `Month` from `OrderDate` and group by `Month` and `Category`
5. Sort to find top-grossing categories by month

39.0.5 Starter Code

[6]:
```
import pandas as pd

df = pd.read_csv("sales_data.csv")

# Revenue Column
df["Revenue"] = df["Price"] * df["Quantity"]

# Group by Category + Region
summary = df.groupby(["Category", "Region"]).agg({
    "Revenue": "sum",
    "Quantity": "mean"
})
print(summary)

# Pivot Table
pivot = df.pivot_table(index="Category", columns="Region", values="Revenue",␣
 ↪aggfunc="sum")
print(pivot)

# Extract Month
df["Month"] = pd.to_datetime(df["OrderDate"]).dt.month

# Monthly Revenue by Category
monthly = df.groupby(["Month", "Category"])["Revenue"].sum().unstack()
print(monthly)
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
Cell In[6], line 3
      1 # import pandas as pd
----> 3 df = pd.read_csv(                )
      5 # Revenue Column
      6 df["Revenue"] = df["Price"] * df["Quantity"]

File ~/dev-projects/python-for-data-analytics/venv/lib/python3.12/site-packages
 ↪pandas/io/parsers/readers.py:1026, in read_csv(filepath_or_buffer, sep,␣
 ↪delimiter, header, names, index_col, usecols, dtype, engine, converters,␣
 ↪true_values, false_values, skipinitialspace, skiprows, skipfooter, nrows,␣
 ↪na_values, keep_default_na, na_filter, verbose, skip_blank_lines, parse_dates␣
 ↪infer_datetime_format, keep_date_col, date_parser, date_format, dayfirst,␣
 ↪cache_dates, iterator, chunksize, compression, thousands, decimal,␣
 ↪lineterminator, quotechar, quoting, doublequote, escapechar, comment,␣
 ↪encoding, encoding_errors, dialect, on_bad_lines, delim_whitespace,␣
 ↪low_memory, memory_map, float_precision, storage_options, dtype_backend)
   1013 kwds_defaults = _refine_defaults_read(
   1014     dialect,
   1015     delimiter,
   (…)   1022     dtype_backend=dtype_backend,
```

```
   1023 )
   1024 kwds.update(kwds_defaults)
-> 1026 return _read(filepath_or_buffer, kwds)

File ~/dev-projects/python-for-data-analytics/venv/lib/python3.12/site-packages
 ↪pandas/io/parsers/readers.py:620, in _read(filepath_or_buffer, kwds)
    617 _validate_names(kwds.get("names", None))
    619 # Create the parser.
--> 620 parser = TextFileReader(filepath_or_buffer, **kwds)
    622 if chunksize or iterator:
    623     return parser

File ~/dev-projects/python-for-data-analytics/venv/lib/python3.12/site-packages
 ↪pandas/io/parsers/readers.py:1620, in TextFileReader.__init__(self, f, engine␣
 ↪**kwds)
   1617     self.options["has_index_names"] = kwds["has_index_names"]
   1619 self.handles: IOHandles | None = None
-> 1620 self._engine = self._make_engine(f, self.engine)

File ~/dev-projects/python-for-data-analytics/venv/lib/python3.12/site-packages
 ↪pandas/io/parsers/readers.py:1880, in TextFileReader._make_engine(self, f,␣
 ↪engine)
   1878     if "b" not in mode:
   1879         mode += "b"
-> 1880 self.handles = get_handle(
   1881     f,
   1882     mode,
   1883     encoding=self.options.get(          , None),
   1884     compression=self.options.get(             , None),
   1885     memory_map=self.options.get(            , False),
   1886     is_text=is_text,
   1887     errors=self.options.get(                 ,         ),
   1888     storage_options=self.options.get(                , None),
   1889 )
   1890 assert self.handles is not None
   1891 f = self.handles.handle

File ~/dev-projects/python-for-data-analytics/venv/lib/python3.12/site-packages
 ↪pandas/io/common.py:873, in get_handle(path_or_buf, mode, encoding,␣
 ↪compression, memory_map, is_text, errors, storage_options)
    868 elif isinstance(handle, str):
    869     # Check whether the filename is to be opened in binary mode.
    870     # Binary mode does not support 'encoding' and 'newline'.
    871     if ioargs.encoding and "b" not in ioargs.mode:
    872         # Encoding
--> 873         handle = open(
    874             handle,
    875             ioargs.mode,
```

```
    876             encoding=ioargs.encoding,
    877             errors=errors,
    878             newline=  ,
    879         )
    880     else:
    881         # Binary mode
    882         handle = open(handle, ioargs.mode)

FileNotFoundError: [Errno 2] No such file or directory: 'sales_data.csv'
```

39.0.6 Sample Output (Pivot)

[ ]:

```
Region           East   South   West
Category
Furniture         540     0      1200
Office Supplies   250     0       180
Technology        950    1100     0
```

39.0.7 Stretch Goals

Create a bar chart from the pivot table (using matplotlib)

Export summaries to Excel (multiple sheets)

Highlight top revenue category per region

39.0.8 Phase 3: Visualization & Analytical Storytelling

(Weeks 11–13)-Communicate insights through data visuals

40 Week 11: Matplotlib Basics for Plotting

Topics: * Line, bar, scatter, histogram * Titles, labels, ticks, legends * Saving charts as images

Project: * "Revenue Dashboard" – Visualize monthly revenue and product trends

Learning Objectives

By the end of this week, you will be able to: - Create common chart types: line, bar, scatter, histogram - Label your charts with titles, axes, legends - Adjust chart size, colors, and styles - Save plots as image files (e.g. PNG) - Understand basic chart use-cases in business contexts

40.1 What is Matplotlib?

- Core plotting library in Python
- Powers most data visuals in notebooks
- Great for **custom**, **publication-quality** plots

40.1.1 Line Plot

[ ]:
```
import matplotlib.pyplot as plt

months = ["Jan", "Feb", "Mar", "Apr"]
sales = [1000, 1200, 900, 1400]

plt.plot(months, sales)
plt.title("Monthly Sales")
plt.xlabel("Month")
plt.ylabel("Sales ($)")
plt.show()
```

40.1.2 Bar Chart

[ ]:
```
categories = ["A", "B", "C"]
values = [10, 30, 20]

plt.bar(categories, values, color="skyblue")
plt.title("Category Performance")
plt.show()
```

41 Scatter Plot

[ ]:
```
x = [1, 2, 3, 4]
y = [2, 4, 1, 3]

plt.scatter(x, y, color="green")
plt.title("Relationship Example")
plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```

41.0.1 Histogram

[ ]:
```
ages = [22, 25, 27, 30, 25, 35, 40, 42, 50, 23]
plt.hist(ages, bins=5, color="orange")
plt.title("Age Distribution")
plt.xlabel("Age")
plt.ylabel("Count")
plt.show()
```

41.0.2 Save to File

[ ]:
```
plt.savefig("sales_chart.png")
```

41.0.3 Chart Types & When to Use

Chart	Use Case
Line	Trend over time
Bar	Compare values across categories
Scatter	Relationship between two variables
Histogram	Distribution of values

41.0.4 Tips for Business Plots

Label clearly (titles, axes)

Avoid clutter (keep it simple)

Use consistent color schemes

Highlight what matters (e.g. outliers, trends)

42 Practice Exercises

42.0.1 1. Plot revenue trend over 6 months

[ ]:
```
months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun"]
revenue = [1000, 1100, 1200, 900, 1400, 1500]

plt.plot(months, revenue)
plt.title("Monthly Revenue")
plt.xlabel("Month")
plt.ylabel("Revenue ($)")
plt.show()
```

42.0.2 2. Bar chart for sales by product

[ ]:
```
products = ["Pen", "Notebook", "Stapler"]
sales = [120, 90, 60]

plt.bar(products, sales, color="purple")
plt.title("Product Sales")
plt.show()
```

42.0.3 3. Scatter plot of Hours Studied vs Score

[ ]:
```
hours = [1, 2, 3, 4, 5]
scores = [50, 60, 65, 75, 80]

plt.scatter(hours, scores)
plt.title("Study Hours vs Exam Score")
plt.xlabel("Hours Studied")
```

```
plt.ylabel("Score")
plt.show()
```

42.0.4 4. Histogram of customer ages

[ ]:
```
ages = [22, 25, 30, 32, 35, 40, 42, 44, 45, 50, 55, 60]

plt.hist(ages, bins=6, color="teal")
plt.title("Customer Age Distribution")
plt.xlabel("Age")
plt.ylabel("Count")
plt.show()
```

42.0.5 5. Save chart to file

[ ]:
```
plt.plot([1, 2, 3], [3, 1, 4])
plt.title("Demo Plot")
plt.savefig("demo_plot.png")
```

42.1 Project: Revenue Dashboard (Matplotlib Only)

42.1.1 Goal

Create a basic **monthly revenue dashboard** showing: - Revenue trend over time - Revenue by product category - Customer volume by region - Save charts as images

42.1.2 Sample Dataset: `revenue_data.csv`

```
Month,Category,Region,Revenue,Customers
Jan,Office Supplies,East,1200,35
Jan,Technology,West,2200,50
Feb,Office Supplies,East,900,30
Feb,Technology,West,2500,52
Mar,Office Supplies,East,1100,40
Mar,Technology,West,2800,60
```

42.1.3 Tasks

1. Load CSV into DataFrame
2. Create:
   - Line chart: Revenue over Months
   - Bar chart: Total Revenue by Category
   - Bar chart: Customers by Region
3. Add labels and titles
4. Save each chart as PNG

42.1.4 Starter Code

```
[ ]: import pandas as pd
     import matplotlib.pyplot as plt

     df = pd.read_csv("revenue_data.csv")

     # Revenue trend
     monthly = df.groupby("Month")["Revenue"].sum()
     monthly.plot(kind="line", title="Monthly Revenue", xlabel="Month",␣
      ↪ylabel="Revenue")
     plt.savefig("monthly_revenue.png")
     plt.clf()

     # Revenue by category
     category = df.groupby("Category")["Revenue"].sum()
     category.plot(kind="bar", title="Revenue by Category", color="skyblue")
     plt.ylabel("Revenue")
     plt.savefig("revenue_by_category.png")
     plt.clf()

     # Customers by region
     region = df.groupby("Region")["Customers"].sum()
     region.plot(kind="bar", title="Customers by Region", color="green")
     plt.ylabel("Customers")
     plt.savefig("customers_by_region.png")
```

42.1.5 Sample Output (Charts)

```
[ ]: monthly_revenue.png
     revenue_by_category.png
     customers_by_region.png
```

43 Week 12: Seaborn for Statistical Visualization

Topics: * Distribution plots: distplot, boxplot, violinplot * Relationship plots: scatterplot, pairplot, heatmap * Themes and color palettes

Project: * "Product Analytics Visuals" – Visualize sales vs pricing vs customer rating

Learning Objectives

By the end of this week, you will be able to: - Create statistical plots using Seaborn - Visualize distributions and relationships - Apply built-in themes and color palettes - Combine categorical and numeric insights in one chart - Make better visual decisions using Seaborn's expressiveness

43.1 What is Seaborn?

- Built on top of Matplotlib

- High-level, easy-to-use API
- Better defaults, cleaner charts

[ ]:
```
import seaborn as sns
import matplotlib.pyplot as plt
```

43.1.1 Distribution Plots

[ ]:
```
# Histogram + KDE
sns.histplot(data=df, x="Age", kde=True)

# Boxplot
sns.boxplot(data=df, x="Department", y="Salary")

# Violin plot
sns.violinplot(data=df, x="Region", y="Revenue")
```

43.1.2 Relationship Plots

[ ]:
```
# Scatter + trend line
sns.scatterplot(data=df, x="Price", y="Rating")

# Add regression line
sns.lmplot(data=df, x="Experience", y="Salary")
```

43.1.3 Heatmaps (Correlation or Pivot Tables)

[ ]:
```
# Correlation matrix
sns.heatmap(df.corr(), annot=True, cmap="coolwarm")
```

43.1.4 Categorical Visuals

[ ]:
```
# Count of categories
sns.countplot(data=df, x="Category")

# Swarm for distribution per group
sns.swarmplot(data=df, x="Category", y="Score")
```

43.1.5 Themes & Styles

[ ]:
```
sns.set_style("whitegrid")  # options: white, darkgrid, ticks
sns.set_palette("pastel")   # or "deep", "muted", "Set2"
```

43.1.6 Seaborn Use Cases

Visual	Use Case
Boxplot	Compare distribution across categories
Heatmap	Show correlation or pivot summary
Scatter/LM	Relationship between numeric variables
Countplot	Distribution of categories
Violinplot	Mix of box + KDE (good for skewed data)

43.1.7 Seaborn vs Matplotlib

Feature	Matplotlib	Seaborn
Flexibility	High	Medium (high-level)
Aesthetics	Manual styling	Great by default
Stats Awareness	No	Yes

44 Practice Exercises

44.0.1 1. Plot distribution of ages

[ ]:
```
sns.histplot(df["Age"], kde=True)
```

44.0.2 2. Compare Salary across Departments using Boxplot

[ ]:
```
sns.boxplot(data=df, x="Department", y="Salary")
```

44.0.3 3. Scatterplot of Price vs Rating

[ ]:
```
sns.scatterplot(data=df, x="Price", y="Rating")
```

44.0.4 4. Regression line of Experience vs Salary

[ ]:
```
sns.lmplot(data=df, x="Experience", y="Salary")
```

44.0.5 5. Countplot of Product Categories

[ ]:
```
sns.countplot(data=df, x="Category")
```

44.0.6 6. Heatmap of correlation matrix

[ ]:
```
sns.heatmap(df.corr(), annot=True)
```

44.0.7 7. Violinplot of Revenue by Region

[ ]:
```
sns.violinplot(data=df, x="Region", y="Revenue")
```

44.0.8 8. Set theme to whitegrid and palette to pastel

[ ]:
```
sns.set_style("whitegrid")
sns.set_palette("pastel")
```

44.0.9 9. Swarmplot of Scores by Group

[ ]:
```
sns.swarmplot(data=df, x="Group", y="Score")
```

44.0.10 10. Save Seaborn chart to file

[ ]:
```
plot = sns.boxplot(data=df, x="Category", y="Sales")
plot.figure.savefig("boxplot_output.png")
```

45 Project:

45.0.1 Product Analytics Visuals (Seaborn)

45.0.2 Goal

Visualize product performance and customer behavior using: - Price vs Rating - Category distributions - Revenue vs Discount relationships - Correlations between metrics

45.0.3 Sample Dataset: `product_data.csv`

```
Product,Category,Price,Discount,Rating,Revenue,Region
Notebook,Office Supplies,15,0.1,4.2,3000,East
Monitor,Technology,200,0.15,4.7,8000,West
Chair,Furniture,85,0.2,4.0,5000,South
Pen,Office Supplies,1.5,0.05,4.1,1200,East
Desk,Furniture,250,0.25,4.5,9500,North
```

45.0.4 Tasks

1. Plot scatter of `Price vs Rating`
2. Use `lmplot()` to show regression line (Price vs Rating)
3. Boxplot: `Revenue by Category`
4. Violinplot: `Discount by Region`
5. Countplot: Number of products by Category
6. Correlation heatmap for numeric columns

45.0.5 Starter Code

[ ]:
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

sns.set_style("whitegrid")
sns.set_palette("Set2")
```

```
df = pd.read_csv("product_data.csv")

# 1. Price vs Rating
sns.scatterplot(data=df, x="Price", y="Rating")
plt.title("Price vs Customer Rating")
plt.show()

# 2. Regression Line
sns.lmplot(data=df, x="Price", y="Rating")

# 3. Revenue by Category
sns.boxplot(data=df, x="Category", y="Revenue")
plt.title("Revenue Distribution by Category")
plt.show()

# 4. Violinplot: Discount by Region
sns.violinplot(data=df, x="Region", y="Discount")
plt.title("Discount Range by Region")
plt.show()

# 5. Count of Products
sns.countplot(data=df, x="Category")
plt.title("Product Count by Category")
plt.show()

# 6. Heatmap of numeric correlation
sns.heatmap(df.corr(numeric_only=True), annot=True, cmap="coolwarm")
plt.title("Correlation Matrix")
plt.show()
```

45.0.6 Sample Output (Charts)

- Price vs Rating (scatter + trend)
- Revenue by Category (boxplot)
- Discount by Region (violin)
- Count of Products (bar)
- Correlation Heatmap

45.0.7 Stretch Goals

- Create a Seaborn pairplot across Price, Revenue, Rating
- Annotate highest-rated product
- Save all charts to files

46 Week 13: Exploratory Data Analysis (EDA) Projects

Topics: * Combining pandas + matplotlib/seaborn * Detect outliers, correlations, missing data * Business storytelling: crafting a data narrative

Capstone Project: * "EDA Case Study" – Titanic dataset, HR dataset, or a Kaggle dataset Include: * Data cleaning * Visual exploration * Insight summary in Markdown or slides

Learning Objectives

By the end of this week, you will be able to: - Perform full EDA on a real dataset - Clean and prepare messy data - Visualize relationships, distributions, and patterns - Detect outliers, nulls, and trends - Craft an analytical narrative with visual support

46.0.1 What is EDA?

EDA is the process of: - **Understanding** data: structure, types, quality - **Cleaning** data: handling nulls, outliers, formats - **Visualizing** distributions, trends, relationships - **Summarizing** insights for decisions

46.1 Typical EDA Steps

1. **Understand the Dataset**
   - `.head()`, `.info()`, `.describe()`
   - Check shapes, types, ranges
2. **Clean the Data**
   - Handle missing values
   - Fix data types (e.g., dates)
   - Rename columns
3. **Explore Distributions**
   - Histograms, boxplots, violinplots
   - Summary stats (mean, median, outliers)
4. **Explore Relationships**
   - Scatterplots, heatmaps, grouped summaries
5. **Group & Compare**
   - `groupby()`, `pivot_table()`, aggregation
6. **Tell a Story**
   - Structure findings logically
   - Use visuals to support each point

46.2 Key Questions to Ask in EDA

- What variables are important? Why?
- What patterns or anomalies exist?
- Are there missing or extreme values?
- How are groups different (e.g., by gender, region)?
- What story can the data tell?

46.3 Tools Used

- Pandas for data loading, cleaning, analysis

- Seaborn/Matplotlib for visuals
- Markdown/Slides for insights

46.3.1 Practice: Mini-EDA

47 Mini-EDA: HR Attrition Dataset

“‘python import pandas as pd import seaborn as sns import matplotlib.pyplot as plt

df = pd.read_csv(“hr_data.csv”)

48 Basic checks

print(df.head()) print(df.info()) print(df.describe())

49 Null values

print(df.isna().sum())

50 Turn ‘HireDate’ into datetime

df[“HireDate”] = pd.to_datetime(df[“HireDate”])

51 Add tenure

df[“Tenure”] = (pd.to_datetime(“2025-01-01”) - df[“HireDate”]).dt.days // 365

52 Histogram of Age

sns.histplot(df[“Age”], bins=10) plt.title(“Age Distribution”) plt.show()

53 Boxplot Salary by Department

sns.boxplot(data=df, x=“Department”, y=“Salary”) plt.title(“Salary by Department”) plt.show()

54 Countplot of Attrition

sns.countplot(data=df, x=“Attrition”) plt.title(“Attrition Distribution”) plt.show()

55 Correlation

sns.heatmap(df.corr(numeric_only=True), annot=True) plt.title(“Correlation Matrix”) plt.show()

55.1 Capstone Project: EDA Case Study

Choose one dataset and conduct a **full EDA**, delivering your findings as a Jupyter Notebook or Markdown report.

55.1.1 Choose One Dataset:

Dataset	Description
Retail Sales	Product, region, time, revenue
HR Attrition	Employees, salary, department, attrition
Kaggle: Titanic	Survival based on demographics
Marketing Campaign Funnel	Clicks, conversions, revenue
Survey NPS Sentiment	Text, scores, feedback sentiment

55.1.2 Requirements

1. **Data Cleaning**
   - Drop/fix nulls
   - Convert data types
   - Rename columns
2. **Feature Engineering**
   - Create derived columns (e.g. tenure, revenue)
   - Bin or categorize (e.g., age groups)
3. **Visualizations**
   - 3–5 high-quality charts using Seaborn/Matplotlib
   - Include:
     - Distribution
     - Relationship
     - Grouped comparison
     - Trend over time (if applicable)
4. **Insight Summary**
   - What did you learn from the data?
   - What's surprising or useful?
   - Use Markdown or slides to communicate
5. **Final Deliverables**
   - `.ipynb` notebook with code + visuals
   - Markdown summary OR exported slides
   - Optional: GitHub repo or downloadable PDF

55.1.3 Starter Template

[1]:
```
# Load data
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

df = pd.read_csv("your_dataset.csv")
```

```python
# Initial check
df.info()
df.head()

# Cleaning steps...

# Feature Engineering...

# Visuals
sns.histplot(df["SomeColumn"])
# Add 3-5 relevant plots

# Markdown Summary:
# - Key insights
# - Issues or gaps
# - Business relevance
```

```
The history saving thread hit an unexpected error (OperationalError('attempt to
write a readonly database')).History will not be written to the database.
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
Cell In[1], line 3
      1 # Load data
      2 import pandas as pd
----> 3 import seaborn as sns
      4 import matplotlib.pyplot as plt
      6 df = pd.read_csv("your_dataset.csv")

ModuleNotFoundError: No module named 'seaborn'
```

55.1.4 Stretch Goals

Use pairplot or FacetGrid for multi-view plots

Add interactivity with Plotly (optional)

Compare insights across groups (e.g., attrition by gender + department)

56 Phase 4: Advanced Analytics & Projects

(Weeks 14–16)-Apply complete workflow and optionally explore ML

56.0.1 Week 14: Time Series & Date Analysis

- Using .dt accessor for date components
- Resampling: daily, weekly, monthly
- Rolling averages, time windowing

Project: * Analyze and visualize website traffic or sales trends over time

Learning Objectives

By the end of this week, you will be able to: - Parse dates and extract time components - Use `.dt` accessor for time operations - Perform resampling (daily, weekly, monthly) - Calculate rolling statistics (moving averages) - Visualize time-based trends

56.1 Why Time Series?

- Time is key in sales, finance, traffic, user activity
- Many business questions are **time-dependent**:
  - What's the monthly growth?
  - Are we improving year-over-year?
  - What's the 7-day average?

56.1.1 Working with Dates in Pandas

```
[ ]: df["OrderDate"] = pd.to_datetime(df["OrderDate"])
```

Ensures datetime format for plotting and resampling

56.1.2 Extracting Date Parts

```
[ ]: df["Year"] = df["OrderDate"].dt.year
     df["Month"] = df["OrderDate"].dt.month
     df["Weekday"] = df["OrderDate"].dt.day_name()
     df["Hour"] = df["OrderDate"].dt.hour
```

56.1.3 Resampling (Daily, Monthly, etc.)

```
[ ]: df.set_index("OrderDate", inplace=True)

     # Monthly sales
     monthly = df["Revenue"].resample("M").sum()

     # Weekly average
     weekly = df["Visits"].resample("W").mean()
```

56.1.4 Common frequency codes:

- "D" – daily
- "W" – weekly
- "M" – month-end
- "MS" – month start
- "Q" – quarterly

56.1.5 Rolling Averages (Moving Window)

```
[ ]: monthly["7-day"] = monthly.rolling(window=7).mean()
```

Helps smooth volatility Great for trend detection

56.1.6 Time Series Plots

```
[ ]: import matplotlib.pyplot as plt

monthly.plot(title="Monthly Revenue Trend")
plt.ylabel("Revenue")
plt.show()
```

56.1.7 Common Use Cases

- Website traffic over time
- Sales performance by week/month
- Employee churn trends
- Rolling KPIs for campaigns

57 Practice Exercises

57.0.1 1. Convert column to datetime

```
[ ]: df["Date"] = pd.to_datetime(df["Date"])
```

57.0.2 2. Extract year, month, weekday

```
[ ]: df["Year"] = df["Date"].dt.year
df["Month"] = df["Date"].dt.month
df["Weekday"] = df["Date"].dt.day_name()
```

57.0.3 3. Set date column as index

```
[ ]: df.set_index("Date", inplace=True)
```

57.0.4 4. Resample daily and plot

```
[ ]: df["Revenue"].resample("D").sum().plot()
```

57.0.5 5. Monthly average orders

```
[ ]: df["Orders"].resample("M").mean()
```

57.0.6 6. Add 7-day moving average

[ ]:
```
df["Revenue"].rolling(7).mean().plot()
```

57.0.7 7. Visualize trends with multiple series

[ ]:
```
df[["Revenue", "7-day"]].plot()
```

57.0.8 8. Create weekday heatmap

[ ]:
```
df["Weekday"] = df.index.day_name()
df["Hour"] = df.index.hour
pivot = df.pivot_table(index="Weekday", columns="Hour", values="Sessions",␣
↪aggfunc="sum")
sns.heatmap(pivot)
```

57.0.9 9. Plot total revenue by weekday

[ ]:
```
df.groupby("Weekday")["Revenue"].sum().plot(kind="bar")
```

57.0.10 10. Compare weekly revenue across categories

[ ]:
```
df.groupby([pd.Grouper(freq="W"), "Category"])["Revenue"].sum().unstack().plot()
```

58 Project:

58.0.1 Website or Sales Trends Over Time

58.0.2 Goal

Analyze and visualize trends over time using: - Resampling - Rolling averages - Date decomposition

58.0.3 Sample Dataset: `web_traffic.csv` or `sales_data.csv`

```
Date,PageViews,Users,Revenue
2023-01-01,1200,300,2500
2023-01-02,1300,320,2600
2023-01-03,1100,290,2400
```

58.0.4 Tasks

1. Convert `Date` column to datetime
2. Set `Date` as index
3. Plot:
   - Daily revenue trend
   - 7-day rolling average
4. Resample by month and compare growth
5. Add weekday column and visualize patterns
6. Bonus: Heatmap of traffic by weekday & hour (if hour available)

58.0.5 Starter Code

[ ]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

df = pd.read_csv("web_traffic.csv")
df["Date"] = pd.to_datetime(df["Date"])
df.set_index("Date", inplace=True)

# Daily Revenue
df["Revenue"].plot(title="Daily Revenue")
plt.show()

# 7-Day Rolling Average
df["Revenue"].rolling(7).mean().plot(title="7-Day Revenue Trend")
plt.show()

# Monthly Resample
monthly = df["Revenue"].resample("M").sum()
monthly.plot(kind="bar", title="Monthly Revenue")
plt.show()

# Weekday Analysis
df["Weekday"] = df.index.day_name()
sns.boxplot(data=df, x="Weekday", y="Revenue")
plt.title("Revenue by Weekday")
plt.show()
```

Stretch Goals Compare weekday vs weekend behavior

Add seasonality markers (holidays, promos)

Save visualizations as PNG

58.0.6 Week 15: Intro to Predictive Modeling (Optional)

- Linear Regression with scikit-learn
- Model training, evaluation basics
- Avoiding overfitting, test/train split

Project: * Predict future sales or employee attrition using linear regression

Learning Objectives

By the end of this week, you will be able to: - Understand what a predictive model is - Build a simple linear regression model using `scikit-learn` - Split data into training and testing sets - Evaluate model performance (R^2, MAE, RMSE) - Use the model to make predictions

58.1 What is Predictive Modeling?

Predictive modeling uses **historical data** to forecast **future outcomes**.

Examples: - Predict next month's sales - Predict employee attrition - Predict customer ratings or satisfaction

58.2 Linear Regression Overview

- Simplest form of prediction
- Assumes a **linear relationship** between input (X) and output (y)
- Predicts **continuous** values

58.3 Modeling Workflow

1. **Define X (features) and y (target)**
2. **Split** the data into training and test sets
3. **Train** the model using `.fit()`
4. **Predict** on new data using `.predict()`
5. **Evaluate** the model

58.4 Key Metrics

Metric	Use
R^2	How well model explains variance (closer to 1 = better)
MAE	Mean Absolute Error (lower = better)
RMSE	Root Mean Squared Error (lower = better)

58.4.1 Tips for Good Predictions

- Make sure your features (X) are numeric
- Remove or fill missing values
- Avoid using features that "leak" future information
- Trai/test split is critical to avoid overfitting

58.4.2 scikit-learn Model Summary

Step	Function
Split data	`train_test_split()`
Train model	`model.fit()`
Predict	`model.predict()`
Evaluate	`r2_score()`, `mean_absolute_error()`

59 Practice Exercises

59.0.1 1. Load dataset and check shape

[ ]:
```
df = pd.read_csv("sales_prediction.csv")
print(df.shape)
print(df.head())
```

59.0.2 2. Define feature matrix X and target y

[ ]:
```
X = df[["Ad_Spend", "Email_Spend", "Social_Spend"]]
y = df["Total_Revenue"]
```

59.0.3 3. Train/test split

[ ]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=1)
```

59.0.4 4. Train model

[ ]:
```
from sklearn.linear_model import LinearRegression
model = LinearRegression()
model.fit(X_train, y_train)
```

59.0.5 5. Make predictions

[ ]:
```
y_pred = model.predict(X_test)
```

59.0.6 6. Evaluate model

[ ]:
```
from sklearn.metrics import mean_absolute_error, r2_score

print("MAE:", mean_absolute_error(y_test, y_pred))
print("R²:", r2_score(y_test, y_pred))
```

59.0.7 7. Visualize actual vs predicted

[ ]:
```
import matplotlib.pyplot as plt

plt.scatter(y_test, y_pred)
plt.xlabel("Actual Revenue")
plt.ylabel("Predicted Revenue")
plt.title("Actual vs Predicted")
plt.plot([y_test.min(), y_test.max()], [y_test.min(), y_test.max()], "r--")
plt.show()
```

60 Project:

60.0.1 Predict Sales or Attrition

60.0.2 Goal

Use a real dataset to train and evaluate a simple linear regression model.

60.0.3 Dataset Ideas

Dataset	Predict…
Marketing	Revenue from ad spend
HR	Salary from years of experience
Retail	Sales from price/discount

60.0.4 Tasks

1. Load and clean your dataset
2. Choose numeric input features (X) and target (y)
3. Split into train and test
4. Train `LinearRegression()` model
5. Evaluate using:
   - R^2 Score
   - MAE / RMSE
6. Visualize predictions vs actual
7. Bonus: Interpret coefficients

60.0.5 Starter Code Snippet

[ ]:
```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score, mean_absolute_error

df = pd.read_csv("your_data.csv")

# Clean/prep data
X = df[["Feature1", "Feature2"]]
y = df["Target"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)

model = LinearRegression()
model.fit(X_train, y_train)

predictions = model.predict(X_test)

print("R²:", r2_score(y_test, predictions))
print("MAE:", mean_absolute_error(y_test, predictions))
```

60.0.6 Stretch Goals

Plot residuals (actual - predicted)

Standardize features using StandardScaler

Try polynomial regression for curve-fitting

60.1 ### Week 16: Final Capstone Project

Choose one real dataset and business problem: * Retail Sales Dashboard * HR Attrition Analysis * Marketing Funnel Performance * Financial KPIs Tracker * Survey Sentiment and NPS Analysis

Deliverables: * Cleaned dataset * Data transformation notebook * 3–5 meaningful visualizations * Insightful summary (Markdown / PowerPoint) * GitHub repo or downloadable PDF

60.1.1 Purpose

Build and deliver a real-world data project from start to finish, using everything you've learned:

- Python foundations
- Data cleaning
- Pandas, NumPy
- Visualization (Matplotlib, Seaborn)
- (Optional) Predictive modeling

60.1.2 Project Options

Choose one dataset and business question: | Project | Dataset Type | Goal | | ——————— | ——————- | ——————————— | | **Retail Sales Dashboard** | Product sales (CSV) | Analyze revenue, seasonality, products | | **HR Attrition Analysis** | Employee data | Who is leaving and why? | | **Marketing Funnel KPIs** | Campaign tracking | Track conversions, drop-off points | | **Financial KPI Tracker** | Revenue & cost trends | Visualize and forecast KPIs | | **Survey NPS + Sentiment** | Survey + open feedback | Clean text + analyze satisfaction metrics |

60.1.3 Final Deliverables

Component	Required?	Format
Cleaned Dataset		CSV or processing code
Data Notebook		`.ipynb` (Jupyter/Colab)
Visuals		Seaborn / Matplotlib charts
Insight Summary		Markdown or Slides
Predictive Model	Optional	Linear regression (scikit-learn)
Submission		GitHub repo or ZIP folder

60.1.4 Workflow Checklist

60.1.5 STEP 1: Load & Explore

☐ Read dataset with `pd.read_csv()`

☐ Run `.info()` and `.describe()`
☐ Check nulls, dtypes, duplicates

60.1.6 STEP 2: Clean Data

☐ Handle missing values (drop or fill)
☐ Rename columns
☐ Convert dates
☐ Create new columns if needed

60.1.7 STEP 3: Analyze & Transform

☐ Aggregations: `.groupby()`, `.agg()`
☐ Feature engineering: `.apply()`, `.map()`
☐ Sort, filter, slice

60.1.8 STEP 4: Visualize

☐ Distribution plots (hist, box, violin)
☐ Relationships (scatter, lmplot)
☐ Group comparisons (bar, line, heatmap)
☐ Time trends (resample, rolling)

60.1.9 STEP 5 (Optional): Predict

☐ Split into X, y
☐ Train/test split
☐ LinearRegression fit + predict
☐ Evaluate with R^2, MAE

60.1.10 STEP 6: Communicate

☐ Write insights in Markdown or PPT
☐ Add titles to all charts
☐ Answer business question(s)

60.1.11 Example Visuals to Include

Chart Type	Shows
Line Plot	Trends over time
Bar Plot	Group comparisons (e.g., region, segment)
Boxplot	Distribution across categories
Heatmap	Correlation matrix
Scatterplot	Relationship between 2 variables

60.1.12 Insight Summary Template

You can deliver insights as:

- Markdown cell summary (in your .ipynb)
- PowerPoint slides
- README in your GitHub repo
- Sample structure:

60.2 Capstone Project: Retail Sales Dashboard

60.2.1 Key Questions

- What regions bring the most revenue?
- Are there seasonal sales trends?
- What product categories perform best?

60.2.2 Insights

- Western region has 45% higher avg monthly revenue
- Sales peak during Q4 (holiday season)
- Office Supplies outperform Furniture in margin

60.2.3 Recommendations

- Increase Q4 stock for high-volume items
- Focus marketing on Western & Southern regions
- Optimize Furniture discount strategy

60.2.4 Bonus Insight

- Regression shows 72% of revenue variance explained by ad spend

60.2.5 Submission Options

- GitHub repo with:
- notebook.ipynb
- README.md
- charts/ folder (optional)
- OR ZIP folder with notebook, dataset, and slides

60.2.6 Grading Criteria (if applicable)

Area	Weight
Data Cleaning	25%
Transformations	20%
Visuals	25%
Insights/Story	20%
Bonus: Modeling	10%

61 Sample Project: Retail Sales Dashboard

61.0.1 Dataset: retail_sales.csv

Here's a sample structure:

[ ]:
```
OrderDate,Region,ProductCategory,ProductName,UnitsSold,UnitPrice,Discount,Revenue
2024-01-01,West,Office Supplies,Printer,10,200,0.1,1800
2024-01-02,South,Technology,Laptop,5,800,0.15,3400
2024-01-03,West,Furniture,Desk,3,400,0.2,960
...
```

[ ]:
```
You can create your own, or I can generate it for you — just say the word.
```

62 Business Questions

You'll answer:

- Which region is most profitable?
- Are sales seasonal or monthly trends evident?
- Which product categories perform best?
- How does discounting affect revenue?

63 Starter Notebook Template

[ ]:
```
# Retail Sales Dashboard - Starter

import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# 1. Load Data
df = pd.read_csv("retail_sales.csv", parse_dates=["OrderDate"])
df.info()
df.head()

# 2. Clean & Engineer Features
df["Month"] = df["OrderDate"].dt.to_period("M")
df["EffectivePrice"] = df["UnitPrice"] * (1 - df["Discount"])
df["TotalRevenue"] = df["UnitsSold"] * df["EffectivePrice"]

# 3. Monthly Sales Trend
monthly = df.groupby("Month")["TotalRevenue"].sum()
monthly.plot(title="Monthly Revenue Trend", figsize=(10, 4))
plt.ylabel("Revenue")
plt.show()

# 4. Revenue by Region
```

```
sns.barplot(data=df, x="Region", y="TotalRevenue", estimator=sum)
plt.title("Revenue by Region")
plt.show()

# 5. Category Analysis
sns.boxplot(data=df, x="ProductCategory", y="TotalRevenue")
plt.title("Revenue Distribution by Category")
plt.show()

# 6. Discount vs Revenue Scatter
sns.scatterplot(data=df, x="Discount", y="TotalRevenue", alpha=0.6)
plt.title("Discount Impact on Revenue")
plt.show()
```

64 Markdown Summary Example

64.1 Capstone Project Summary: Retail Sales Dashboard

64.1.1 Business Questions

- What region generates the most revenue?
- How do discounts affect performance?
- Are there product categories with high returns?

64.1.2 Key Findings

- The **West region** had 40% more revenue than others.
- **Technology** products outperform in average revenue per unit.
- High discounts (>20%) reduce total revenue below average.

64.1.3 Recommendations

- Run promotions in low-performing regions.
- Reduce discounts on high-performing tech items.
- Focus Q4 inventory on Office Supplies & Technology.

64.1.4 Summary: How Does It Stack Up?

Your course is surprisingly rigorous for a self-paced or part-time structure.

It focuses heavily on hands-on learning, not just theory or slides.

You're giving learners portfolio-ready projects that resemble what they'd do on the job.

You cover Python as the primary data language, while many popular certs (Google) emphasize spreadsheets, SQL, and BI tools more than programming.

[ ]: