Analytics Programming and Data Visualisation: Complete Learning Book

Comprehensive Guide for Analytics Programming and Data Visualisation (H9APDV)
From Beginner to Advanced Level

Table of Contents

Module Overview

What You'll Learn

This complete book teaches you **Analytics Programming and Data Visualisation** using Python, pandas, Matplotlib, and Seaborn. By the end, you'll be able to:

Write clean, professional Python code for data analysis
Load, clean, transform data from multiple sources
Work with relational databases and APIs
Create compelling visualizations and dashboards
Build end-to-end analytics pipelines
Handle big data efficiently
Communicate insights to non-technical stakeholders

Who This Is For

- Aspiring data analysts and data engineers
- Business analysts learning Python
- Data science beginners
- Anyone working with data in Python

Learning Approach

Each section includes: - **Simple explanations** in plain English - **Real code examples** with comments - **Step-by-step exercises** with solutions - **Real-world scenarios** you'll encounter in jobs - **Common pitfalls** to avoid

---

Part 1: Foundations

Lecture 1: Introduction to Analytics Programming

1.1 Why Python?

Python has become the #1 language for data analytics because:

Reason	Why It Matters
Easy to Read	Code looks like English - fast to write and understand
Rich Ecosystem	Libraries for everything: pandas, NumPy, scikit-learn, Matplotlib
Flexible	Works for analytics, web, automation, AI, machine learning
Industry Standard	Used at Google, Netflix, Spotify, JPMorgan, NASA, etc.
Community	Massive open-source community with tutorials and support

1.2 Data Exploration: What Are We Working With?

Types of Data We'll Handle

```
Structured Data (main focus):
   Tabular (rows & columns like Excel)
   Relational (linked tables with keys)
   Time-series (data over time)
   Multidimensional (matrices, cubes)
```

```
Semi-structured:
  JSON (web APIs)
  XML (data feeds)

Unstructured:
  Text (can be turned into structured features)
  Web pages (scrape to extract structured data)
```

Real Example: Sales Data

```
Date        Product    Sales    Region
2024-01-01  Laptop     1500     North
2024-01-01  Phone      800      South
2024-01-02  Laptop     1200     North
```

This is tabular, structured data - exactly what we'll work with.

1.3 Data Analytics Methodologies

CRISP-DM: Industry Standard Process

CRISP-DM (Cross Industry Standard Process for Data Mining) is used by 80% of analytics teams.

```
1. BUSINESS UNDERSTANDING
     What's the business problem?
     What decisions will data help with?

2. DATA UNDERSTANDING
     Explore available data
     Check quality, identify issues

3. DATA PREPARATION
     Clean, transform, combine data
     Handle missing values

4. MODELING
     Build analytics models/summaries

5. EVALUATION
     Do results answer the business question?
     How confident are we?

6. DEPLOYMENT
     Implement solution
     Monitor performance
```

Real Example: Analyzing Customer Churn

Phase	Question	Action
Business Understanding	Why are customers leaving?	Meet with retention team
Data Understanding	What data do we have?	Explore customer database
Preparation	What patterns predict churn?	Clean billing, usage data
Modeling	Build a prediction model	Identify at-risk customers
Evaluation	Is the model accurate?	Test on historical data
Deployment	Tell the business	Alert retention team; take action

1.4 Good vs Bad Visualizations (Preview)

We'll go deep into this in Part 5, but here's the preview:

Bad Visualization:

```
3D pie chart (hard to compare)
Rainbow colors (no meaning)
No title or axis labels
Too many numbers on screen
```

Good Visualization:

```
Simple bar chart
Clear title: "Sales by Region"
Labeled axes with units
One key message
```

Lecture 2: Python Basics & Data Types

2.1 Setting Up Python

Installation

```bash
# Windows/Mac/Linux
# Download Python 3.11+ from python.org
# OR use conda (anaconda.com)

# Verify installation
python --version
pip --version
```

Essential Tools

```bash
# Install Jupyter Notebook (recommended for learning)
pip install jupyter pandas matplotlib seaborn

# Start Jupyter
jupyter notebook
```

2.2 Python Fundamentals: Variables and Data Types

Everything in Python has a Type

```python
# Let's create variables and check their types

# Integers (whole numbers)
age = 25
customer_id = 12345
print(type(age))  # <class 'int'>

# Floats (decimals)
price = 19.99
conversion_rate = 0.087  # 8.7%
print(type(price))  # <class 'float'>

# Strings (text)
name = "Alice Johnson"
email = "alice@company.com"
print(type(name))  # <class 'str'>

# Booleans (True/False)
is_active = True
is_premium = False
print(type(is_active))  # <class 'bool'>
```

Type Conversion (Coercion)

```python
# Convert between types when needed

# String to number
revenue_str = "1000000"
revenue_int = int(revenue_str)  # 1000000
revenue_float = float(revenue_str)  # 1000000.0

# Number to string
year = 2024
text = "The year is " + str(year)
print(text)  # Output: The year is 2024

# Better: use f-strings (modern Python)
```

```python
text = f"The year is {year}"
print(text)  # Output: The year is 2024

# String to boolean (careful - any non-empty string is True!)
bool("false")  # True (not False!)
bool("")  # False
```

2.3 Operators and Expressions

Arithmetic Operators

```python
# Basic math - works like a calculator
total_sales = 100000
num_months = 12
monthly_avg = total_sales / num_months  # 8333.33

# Common operators
addition = 5 + 3  # 8
subtraction = 10 - 4  # 6
multiplication = 7 * 8  # 56
division = 20 / 4  # 5.0
floor_division = 20 // 3  # 6 (rounds down)
modulus = 20 % 3  # 2 (remainder)
exponent = 2 ** 10  # 1024
```

Comparison Operators (return True or False)

```python
# Comparison - very useful for filtering data
price = 99.99
threshold = 100

price < threshold  # True
price <= threshold  # False
price == threshold  # False
price != threshold  # True
price > 50  # True

# Strings can be compared too
"Alice" < "Bob"  # True (alphabetically)
"apple" == "apple"  # True
```

Logical Operators (and, or, not)

```python
# Combine multiple conditions
age = 25
income = 50000
is_employed = True
```

```
# AND - all conditions must be True
eligible = (age >= 18) and (income >= 30000) and is_employed
print(eligible)  # True

# OR - at least one condition must be True
is_student = False
is_senior = False
eligible_for_discount = is_student or is_senior or (age >= 65)
print(eligible_for_discount)  # False

# NOT - reverses True/False
not is_employed  # False
```

2.4 Strings: Text Processing

Creating Strings

```
# Different ways to create strings
single_quotes = 'Hello'
double_quotes = "World"
multi_line = """This is
a multi-line
string"""

# Strings are immutable - you can't change them in place
name = "alice"
# name[0] = "A"  #   ERROR! Can't do this

# Instead, create a new string
name_fixed = "A" + name[1:]  # "alice"
print(name_fixed)  # Alice
```

String Operations

```
# Concatenation (joining)
first_name = "John"
last_name = "Smith"
full_name = first_name + " " + last_name
print(full_name)  # John Smith

# Repetition
dash_line = "-" * 50
print(dash_line)  # --------------------------------------------------

# Check if substring exists (membership)
email = "john@company.com"
"company" in email  # True
```

```python
"gmail" in email  # False
```

String Formatting (f-strings - Modern Way)

```python
# f-strings are fast, clean, and readable
name = "Alice"
age = 28
salary = 75000.50

# Old way (don't use)
message = "Name: " + name + ", Age: " + str(age)

# Better way (f-strings)
message = f"Name: {name}, Age: {age}, Salary: ${salary:,.2f}"
print(message)
# Output: Name: Alice, Age: 28, Salary: $75,000.50

# f-string formatting options
pi = 3.14159
print(f"Pi rounded: {pi:.2f}")  # 3.14
print(f"Percentage: {0.087*100:.1f}%")  # 8.7%
```

Common String Methods

```python
text = "  Hello World  "

# Remove whitespace
text.strip()  # "Hello World"
text.lstrip()  # "Hello World  "
text.rstrip()  # "  Hello World"

# Change case
text.lower()  # "  hello world  "
text.upper()  # "  HELLO WORLD  "
text.title()  # "  Hello World  "

# Find and replace
text.replace("World", "Python")  # "  Hello Python  "

# Split into list
"apple,banana,cherry".split(",")  # ['apple', 'banana', 'cherry']

# Join a list into string
["red", "green", "blue"].join("-")  # ERROR! Wrong order
"-".join(["red", "green", "blue"])  # "red-green-blue"

# Check content
"Hello".startswith("He")  # True
```

```python
"hello".endswith("lo")  # True
```

2.5 Collections: Lists, Tuples, Dictionaries

Lists: Ordered, Changeable Collections

```python
# Creating lists
empty_list = []
numbers = [1, 2, 3, 4, 5]
mixed = [1, "hello", 3.14, True]

# Accessing elements (indexing starts at 0)
numbers = [10, 20, 30, 40, 50]
print(numbers[0])  # 10 (first)
print(numbers[2])  # 30 (third)
print(numbers[-1])  # 50 (last)
print(numbers[-2])  # 40 (second-to-last)

# Slicing (get a subset)
print(numbers[1:4])  # [20, 30, 40] (from index 1 to 3, not including 4)
print(numbers[:3])  # [10, 20, 30] (first 3)
print(numbers[2:])  # [30, 40, 50] (from index 2 to end)
print(numbers[::2])  # [10, 30, 50] (every 2nd element)

# Modifying lists
sales = [100, 200, 150]
sales[1] = 250  # Change one element
sales.append(300)  # Add to end: [100, 250, 150, 300]
sales.insert(0, 50)  # Insert at position: [50, 100, 250, 150, 300]
sales.remove(250)  # Remove by value: [50, 100, 150, 300]

# Useful list methods
numbers = [3, 1, 4, 1, 5, 9, 2, 6]
len(numbers)  # 8 (length)
sum(numbers)  # 31 (total)
max(numbers)  # 9 (maximum)
min(numbers)  # 1 (minimum)
numbers.count(1)  # 2 (how many 1s)
numbers.index(4)  # 2 (position of first 4)

# Sorting
numbers.sort()  # [1, 1, 2, 3, 4, 5, 6, 9] - changes original
sorted_copy = sorted(numbers)  # doesn't change original

# Reversing
numbers.reverse()  # changes original
```

```
numbers[::-1]  # doesn't change original
```

Tuples: Ordered, Unchangeable Collections

```
# Creating tuples (use parentheses)
coordinates = (10, 20)
rgb_color = (255, 128, 0)
single_item = (42,)  # Note the comma - needed for single items!

# Accessing (same as lists)
print(coordinates[0])  # 10
print(rgb_color[-1])  # 0

# Can't modify
# coordinates[0] = 15  #   ERROR!

# Why use tuples?
# - Faster than lists
# - Can use as dictionary keys (lists can't)
# - Prevents accidental changes

# Unpacking
x, y = coordinates  # x=10, y=20
r, g, b = rgb_color  # r=255, g=128, b=0
```

Dictionaries: Key-Value Pairs (Like Real Dictionaries)

```
# Creating dictionaries
empty_dict = {}
person = {
    "name": "Alice",
    "age": 28,
    "city": "Dublin"
}

# Accessing values (by key)
print(person["name"])  # Alice
print(person["age"])  # 28

# Modifying
person["age"] = 29
person["email"] = "alice@company.com"  # Add new key-value

# Check if key exists
"name" in person  # True
"salary" in person  # False

# Get all keys, values, items
```

```python
person.keys()  # dict_keys(['name', 'age', 'city', 'email'])
person.values()  # dict_values(['Alice', 29, 'Dublin', 'alice@company.com'])
person.items()  # dict_items([('name', 'Alice'), ('age', 29), ...])

# Looping through dictionary
for key, value in person.items():
    print(f"{key}: {value}")

# Real-world example: Sales by region
sales_by_region = {
    "North": 150000,
    "South": 120000,
    "East": 180000,
    "West": 95000
}

total_sales = sum(sales_by_region.values())  # 545000
best_region = max(sales_by_region, key=sales_by_region.get)  # "East"
```

2.6 Control Flow: Making Decisions

If-Else Statements

```python
# Basic structure
age = 25

if age >= 65:
    print("Senior citizen")
elif age >= 18:
    print("Adult")
else:
    print("Minor")

# Output: Adult

# Example: Categorize sales
sales = 250000

if sales >= 200000:
    category = "Excellent"
    bonus_rate = 0.15  # 15%
elif sales >= 100000:
    category = "Good"
    bonus_rate = 0.10
elif sales >= 50000:
    category = "Average"
```

```python
    bonus_rate = 0.05
else:
    category = "Below Target"
    bonus_rate = 0.0

print(f"Sales category: {category}, Bonus: {bonus_rate*100}%")
# Output: Sales category: Excellent, Bonus: 15.0%
```

Loops: Repeating Actions

```python
# For loop (iterate through collection)
customers = ["Alice", "Bob", "Charlie"]

for customer in customers:
    print(f"Processing {customer}")

# Output:
# Processing Alice
# Processing Bob
# Processing Charlie

# For loop with range
for i in range(5):  # 0, 1, 2, 3, 4
    print(f"Count: {i}")

# While loop (repeat until condition is false)
count = 0
while count < 3:
    print(f"Count: {count}")
    count += 1  # count = count + 1

# Loop through dictionary
employee_salary = {"Alice": 75000, "Bob": 80000, "Charlie": 72000}

for name, salary in employee_salary.items():
    tax = salary * 0.20  # 20% tax
    net = salary - tax
    print(f"{name}: ${salary} - ${tax} tax = ${net} net")

# Breaking out of loop
for i in range(10):
    if i == 5:
        break  # Exit loop early
    print(i)
# Output: 0, 1, 2, 3, 4

# Skipping to next iteration
```

```python
for i in range(5):
    if i == 2:
        continue  # Skip this one
    print(i)
# Output: 0, 1, 3, 4
```

2.7 Functions: Reusable Code Blocks

Why Functions?

```
Without functions:
Calculate tax for employee 1
Calculate tax for employee 2
Calculate tax for employee 3
(repeat same code 100+ times)

With functions:
Define tax calculation once
Use it for all employees
(DRY principle: Don't Repeat Yourself)
```

Creating and Using Functions

```python
# Simple function (no parameters, no return)
def greet():
    print("Hello from Analytics!")

greet()  # Call the function
# Output: Hello from Analytics!

# Function with parameters
def calculate_tax(salary):
    """Calculate 20% tax on salary"""
    tax = salary * 0.20
    return tax

annual_tax = calculate_tax(75000)
print(f"Tax: ${annual_tax}")  # Tax: $15000.0

# Function with multiple parameters
def calculate_net_salary(salary, tax_rate=0.20):
    """Calculate net salary after tax"""
    tax = salary * tax_rate
    net = salary - tax
    return net

net_75k = calculate_net_salary(75000)  # Uses default 20%
```

```python
net_75k_custom = calculate_net_salary(75000, 0.25)  # Uses 25%

# Function with multiple returns
def analyze_sales(sales_list):
    """Analyze sales data"""
    total = sum(sales_list)
    average = total / len(sales_list)
    highest = max(sales_list)
    lowest = min(sales_list)
    return total, average, highest, lowest

sales = [100, 250, 150, 300, 200]
total, avg, high, low = analyze_sales(sales)
print(f"Total: {total}, Average: {avg}, High: {high}, Low: {low}")
# Output: Total: 1000, Average: 200.0, High: 300, Low: 100
```

Real-World Example: Data Validation Function

```python
def validate_email(email):
    """Check if email is valid"""
    if "@" not in email:
        return False, "Missing @"
    if email.count("@") > 1:
        return False, "Multiple @"
    if "." not in email.split("@")[1]:
        return False, "Domain missing ."
    return True, "Valid"

# Test it
result, message = validate_email("alice@company.com")
print(f"Email valid: {result}, Message: {message}")  # True, Valid

result, message = validate_email("alice@company")
print(f"Email valid: {result}, Message: {message}")  # False, Domain missing .
```

Lambda Functions (Quick One-Liners)

```python
# Regular function
def square(x):
    return x ** 2

# Same thing as lambda
square = lambda x: x ** 2

# Lambdas are useful for quick operations
numbers = [1, 2, 3, 4, 5]

# Using map to square all numbers
```

```python
squared = list(map(lambda x: x**2, numbers))
print(squared)  # [1, 4, 9, 16, 25]

# Using filter to keep only odd numbers
odd = list(filter(lambda x: x % 2 == 1, numbers))
print(odd)  # [1, 3, 5]
```

Exercises: Part 1

Exercise 1.1: Calculate Customer Metrics

```python
# Given customer data, calculate metrics
customers = {
    "C001": {"name": "Alice", "purchases": 5, "total_spent": 1500},
    "C002": {"name": "Bob", "purchases": 3, "total_spent": 800},
    "C003": {"name": "Charlie", "purchases": 8, "total_spent": 2200}
}

# TODO: Calculate for each customer:
# 1. Average spent per purchase
# 2. Customer worth (purchases * 100)
# 3. Categorize as "High", "Medium", or "Low" value

# Expected output:
# C001 (Alice): $300.00/purchase, Worth: 500, Category: High
# ... and so on
```

Solution 1.1

```python
customers = {
    "C001": {"name": "Alice", "purchases": 5, "total_spent": 1500},
    "C002": {"name": "Bob", "purchases": 3, "total_spent": 800},
    "C003": {"name": "Charlie", "purchases": 8, "total_spent": 2200}
}

for cust_id, data in customers.items():
    avg_spent = data["total_spent"] / data["purchases"]
    worth = data["purchases"] * 100

    if worth >= 500:
        category = "High"
    elif worth >= 300:
        category = "Medium"
    else:
        category = "Low"

    print(f"{cust_id} ({data['name']}): ${avg_spent:.2f}/purchase, Worth: {worth}, Category:
```

```python
# Output:
# C001 (Alice): $300.00/purchase, Worth: 500, Category: High
# C002 (Bob): $266.67/purchase, Worth: 300, Category: Medium
# C003 (Charlie): $275.00/purchase, Worth: 800, Category: High
```

Part 2: Working with Data

Lecture 3: Input/Output & File Handling

3.1 Reading and Writing Files

The Basic Pattern: with open()

```python
# IMPORTANT: Always use 'with' - it closes the file automatically

# Write to a file
with open("notes.txt", "w") as file:
    file.write("Hello World!\n")
    file.write("This is line 2")

# Read from a file
with open("notes.txt", "r") as file:
    content = file.read()
    print(content)

# Output:
# Hello World!
# This is line 2
```

Different Ways to Read

```python
# Method 1: Read entire file as one string
with open("data.txt", "r") as file:
    full_content = file.read()
    print(full_content[:100])  # First 100 characters

# Method 2: Read line by line
with open("data.txt", "r") as file:
    first_line = file.readline()
    second_line = file.readline()

# Method 3: Read all lines into a list
with open("data.txt", "r") as file:
    lines = file.readlines()
    for i, line in enumerate(lines):
```

```python
        print(f"Line {i+1}: {line.strip()}")

# Method 4: Loop directly (most Pythonic)
with open("data.txt", "r") as file:
    for line in file:
        print(line.strip())  # strip() removes newline character
```

3.2 Working with CSV Files

CSV Format Explanation

```
Name,Age,City,Salary
Alice,28,Dublin,75000
Bob,32,Cork,80000
Charlie,25,Galway,65000

^ Comma separates values
^ First row usually has headers
```

Reading CSV with pandas (Recommended)

```python
import pandas as pd

# Read CSV into DataFrame
df = pd.read_csv("employees.csv")

# See first few rows
print(df.head())

# Output:
#       Name  Age     City  Salary
# 0    Alice   28   Dublin   75000
# 1      Bob   32     Cork   80000
# 2 Charlie    25   Galway   65000

# Access columns
print(df["Name"])  # Get entire Name column
print(df["Salary"])  # Get entire Salary column

# Get statistics
print(df["Salary"].mean())  # Average salary
print(df["Salary"].min())  # Lowest salary
print(df["Salary"].max())  # Highest salary
```

Writing CSV with pandas

```python
import pandas as pd
```

```python
# Create data
data = {
    "Name": ["Alice", "Bob", "Charlie"],
    "Age": [28, 32, 25],
    "City": ["Dublin", "Cork", "Galway"],
    "Salary": [75000, 80000, 65000]
}

# Create DataFrame
df = pd.DataFrame(data)

# Write to CSV
df.to_csv("employees.csv", index=False)  # index=False removes row numbers

# Result: employees.csv file created
```

Reading CSV with Python's csv module (More Control)

```python
import csv

# Read and print
with open("employees.csv", "r") as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)

# Output (each row is a list):
# ['Name', 'Age', 'City', 'Salary']
# ['Alice', '28', 'Dublin', '75000']
# ['Bob', '32', 'Cork', '80000']

# Read as dictionaries (column names as keys)
with open("employees.csv", "r") as file:
    reader = csv.DictReader(file)
    for row in reader:
        print(f"{row['Name']}: ${row['Salary']}")

# Output:
# Alice: $75000
# Bob: $80000
```

3.3 Working with JSON Files

JSON Format Explanation

```json
{
  "employees": [
```

```
    {
      "name": "Alice",
      "age": 28,
      "skills": ["Python", "SQL", "Tableau"]
    },
    {
      "name": "Bob",
      "age": 32,
      "skills": ["Python", "R", "Power BI"]
    }
  ]
}

^ Structured like nested dictionaries and lists
^ Very common for web APIs
```

Reading JSON

```python
import json

# Read JSON file
with open("employees.json", "r") as file:
    data = json.load(file)  # Converts to Python dict

# Access data
for employee in data["employees"]:
    print(f"{employee['name']}: {employee['skills']}")

# Output:
# Alice: ['Python', 'SQL', 'Tableau']
# Bob: ['Python', 'R', 'Power BI']

# JSON from string
json_string = '{"name": "Alice", "age": 28}'
person = json.loads(json_string)
print(person["name"])  # Alice
```

Writing JSON

```python
import json

# Create data
employees = {
    "employees": [
        {"name": "Alice", "age": 28, "skills": ["Python", "SQL"]},
        {"name": "Bob", "age": 32, "skills": ["Python", "R"]}
    ]
}
```

```python
# Write to file
with open("employees.json", "w") as file:
    json.dump(employees, file, indent=2)  # indent=2 makes it readable

# Or convert to string
json_string = json.dumps(employees, indent=2)
print(json_string)
```

3.4 Fetching Data from APIs

What is an API?

An API (Application Programming Interface) lets you request data from a web service. Instead of downloading a file, you make a request and get data back.

Real Example: Weather API

```python
import requests
import json

# Request weather data
url = "https://api.openweathermap.org/data/2.5/weather"
params = {
    "q": "Dublin",  # City name
    "appid": "YOUR_API_KEY"  # You need to sign up for this
}

# Make the request
response = requests.get(url, params=params)

# Check if successful
if response.status_code == 200:
    data = response.json()  # Convert to Python dict
    print(f"Temperature: {data['main']['temp']}K")
    print(f"Weather: {data['weather'][0]['description']}")
else:
    print(f"Error: {response.status_code}")
```

API Response Status Codes

```
200 = OK (success!)
201 = Created (new resource made)
400 = Bad Request (you did something wrong)
401 = Unauthorized (need API key)
404 = Not Found
429 = Too Many Requests (slow down!)
500 = Server Error (their problem)
```

Real Data: Stock Price API

```python
import requests

# Get Apple stock data
url = "https://api.example.com/stock/AAPL"
headers = {
    "Authorization": "Bearer YOUR_TOKEN"
}

response = requests.get(url, headers=headers)

if response.status_code == 200:
    stock = response.json()
    print(f"Company: {stock['name']}")
    print(f"Current Price: ${stock['price']}")
    print(f"Previous Close: ${stock['previous_close']}")
    print(f"Change: {stock['change']}%")
else:
    print(f"Failed to fetch: {response.status_code}")
```

3.5 Functions for Code Reuse

Common Data Functions

```python
# Function to load data safely
def load_data(filename):
    """Load CSV file, handle errors gracefully"""
    try:
        import pandas as pd
        df = pd.read_csv(filename)
        print(f"Loaded {len(df)} rows from {filename}")
        return df
    except FileNotFoundError:
        print(f"Error: File {filename} not found")
        return None
    except Exception as e:
        print(f"Error loading file: {e}")
        return None

# Usage
df = load_data("sales.csv")
if df is not None:
    print(df.head())

# Function to save data
def save_data(df, filename):
```

```python
    """Save DataFrame to CSV"""
    try:
        df.to_csv(filename, index=False)
        print(f"Saved {len(df)} rows to {filename}")
    except Exception as e:
        print(f"Error saving file: {e}")

# Function to validate email
def validate_email(email):
    """Check if email looks valid"""
    if "@" not in email or "." not in email:
        return False
    return True

# Function to clean names (remove extra spaces, title case)
def clean_name(name):
    """Clean and standardize name"""
    return name.strip().title()

# Usage
names = ["  ALICE JOHNSON  ", "bob smith", " CHARLIE "]
cleaned = [clean_name(n) for n in names]
print(cleaned)  # ['Alice Johnson', 'Bob Smith', 'Charlie']
```

Exercises: Part 2

Exercise 2.1: Process Employee Data

Create a program that: 1. Reads employee data from CSV 2. Calculates total salary cost 3. Finds highest and lowest paid 4. Saves results to a new file

```python
# Sample CSV content:
# Name,Department,Salary
# Alice,Sales,75000
# Bob,IT,80000
# Charlie,Sales,65000
# Diana,IT,90000

# TODO: Write code to:
# - Load the CSV
# - Print total salary cost
# - Print highest paid employee
# - Print department with highest average salary
# - Save summary to results.txt
```

Solution 2.1

```python
import pandas as pd
```

```python
# Load data
df = pd.read_csv("employees.csv")

# Total salary
total_salary = df["Salary"].sum()
print(f"Total Salary Cost: ${total_salary:,}")

# Highest and lowest paid
highest_paid = df.loc[df["Salary"].idxmax()]
lowest_paid = df.loc[df["Salary"].idxmin()]
print(f"Highest Paid: {highest_paid['Name']} (${highest_paid['Salary']})")
print(f"Lowest Paid: {lowest_paid['Name']} (${lowest_paid['Salary']})")

# Department analysis
dept_avg = df.groupby("Department")["Salary"].mean()
best_dept = dept_avg.idxmax()
print(f"Highest Average Salary: {best_dept} (${dept_avg[best_dept]:,.2f})")

# Save summary
with open("summary.txt", "w") as file:
    file.write(f"Total Salary Cost: ${total_salary:,}\n")
    file.write(f"Highest Paid: {highest_paid['Name']} (${highest_paid['Salary']})\n")
    file.write(f"Department with Highest Avg: {best_dept}\n")

print("Summary saved to summary.txt")
```

Lecture 4: Web Scraping & Regex

4.1 Web Scraping: Getting Data from Websites

What is Web Scraping?

Web scraping = automatically extracting data from websites instead of manually copying it.

Workflow

```
1. Fetch: Get the HTML page
2. Parse: Understand its structure
3. Extract: Find the data we want
4. Clean: Remove extra stuff
5. Save: Store in CSV or database
```

Simple Example: Wikipedia Table

```python
import pandas as pd
```

```python
import requests
from bs4 import BeautifulSoup

# URL to scrape
url = "https://en.wikipedia.org/wiki/List_of_countries_by_population_(United_Nations)"

# Method 1: Use pandas (easiest for tables)
tables = pd.read_html(url)
df = tables[0]  # Get first table
print(df.head())

# Output: Top 10 countries by population
```

Using Beautiful Soup (More Control)

```python
import requests
from bs4 import BeautifulSoup

# Get page content
url = "https://example.com/products"
response = requests.get(url)

if response.status_code == 200:
    # Parse HTML
    soup = BeautifulSoup(response.content, "html.parser")

    # Find all product names (example HTML structure)
    products = soup.find_all("h2", class_="product-name")

    for product in products:
        name = product.text.strip()
        print(name)
else:
    print(f"Failed: {response.status_code}")
```

Real Example: Job Listings

```python
import requests
from bs4 import BeautifulSoup

def scrape_jobs(url):
    """Scrape job listings from a website"""
    response = requests.get(url)
    soup = BeautifulSoup(response.content, "html.parser")

    jobs = []

    # Find all job listings
```

```python
    for listing in soup.find_all("div", class_="job-listing"):
        job = {
            "title": listing.find("h3", class_="job-title").text.strip(),
            "company": listing.find("p", class_="company").text.strip(),
            "location": listing.find("p", class_="location").text.strip(),
            "salary": listing.find("p", class_="salary").text.strip()
        }
        jobs.append(job)

    return jobs

# Use it
jobs = scrape_jobs("https://jobs.example.com")
for job in jobs:
    print(f"{job['title']} at {job['company']}")
```

Important: Web Scraping Ethics

```python
import time
import requests

def scrape_responsibly(url, delay=2):
    """Scrape with respect for server resources"""

    # 1. Check robots.txt
    # (website tells you what's allowed to scrape)

    # 2. Use a custom User-Agent
    headers = {
        "User-Agent": "My Analytics Script/1.0 (Contact: your@email.com)"
    }

    # 3. Add delay between requests
    response = requests.get(url, headers=headers)
    time.sleep(delay)  # Wait 2 seconds

    return response

# Good practices:
#   Check robots.txt first
#   Add delays between requests
#   Respect server load
#   Use API if available (better than scraping)
#   Don't scrape behind login/paywall
#   Don't claim scraped data as your own
```

4.2 Regular Expressions (Regex)

What is Regex?

Regex = patterns to find, match, and extract text. Very powerful for data cleaning.

Simple Patterns

```python
import re

text = "Email: alice@company.com or bob@gmail.com"

# Find all emails (pattern explanation below)
pattern = r"[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}"
emails = re.findall(pattern, text)
print(emails)
# Output: ['alice@company.com', 'bob@gmail.com']

# Find all numbers
numbers = re.findall(r"\d+", "Order #12345 for product 789")
print(numbers)  # ['12345', '789']

# Replace pattern
text = "The price is $99.99"
clean = re.sub(r"\$", "€", text)
print(clean)  # The price is €99.99
```

Common Regex Patterns

```python
import re

# Email pattern
email = r"[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}"

# Phone number (US format)
phone = r"\(\d{3}\) \d{3}-\d{4}"

# URL
url_pattern = r"https?://[^\s]+"

# Digits only
digits = r"\d+"

# Letters only
letters = r"[a-zA-Z]+"

# Date (YYYY-MM-DD)
```

```python
date = r"\d{4}-\d{2}-\d{2}"

# Example: Extract phone numbers
text = "Call (555) 123-4567 or (555) 987-6543"
phones = re.findall(phone, text)
print(phones)  # ['(555) 123-4567', '(555) 987-6543']
```

Data Cleaning with Regex

```python
import re
import pandas as pd

# Sample messy data
sales_data = {
    "date": ["2024-01-15", "2024/01/16", "01-17-2024"],  # Different formats!
    "amount": ["$1,000.50", "€850.25", " 500"]  # Different currencies!
}

df = pd.DataFrame(sales_data)

# Clean amounts (remove currency symbols and commas)
df["amount_clean"] = df["amount"].apply(lambda x: re.sub(r"[^\d.]", "", x))
df["amount_float"] = df["amount_clean"].astype(float)

print(df)
# Output:
#         amount  amount_clean  amount_float
# 0  $1,000.50     1000.50         1000.50
# 1    €850.25     85025           850.25
# 2       500      500             500.00
```

Real-World Example: Extract Data from Text

```python
import re

# Sales report text
report = """
Sales Report - Q4 2024
North Region: $150,000
South Region: $120,500
East Region: $189,750
West Region: $95,250
Total Employees: 125
Average per Employee: $1,836
"""

# Extract all numbers
numbers = re.findall(r"\$([0-9,]+)|(\d+)", report)
```

```python
print(numbers)

# Extract region names and sales
pattern = r"(\w+)\s+Region:\s+\$([0-9,]+)"
matches = re.findall(pattern, report)
for region, sales in matches:
    sales_num = int(sales.replace(",", ""))
    print(f"{region}: {sales_num:,}")

# Output:
# North: 150,000
# South: 120,500
# East: 189,750
# West: 95,250
```

Exercises: Part 3

Exercise 3.1: Data Extraction and Cleaning

```python
# You receive this messy data from a website
raw_data = """
Customer: Alice Johnson, Email: ALICE@COMPANY.COM, Phone: (555) 123-4567, Purchase: $1,999.9
Customer: Bob Smith, Email: bob_smith@gmail.com, Phone: (555) 987-6543, Purchase: €1,500.50
Customer: Charlie Brown, Email: charlie.b@yahoo.com, Phone: (555) 456-7890, Purchase: $850
"""

# TODO: Use regex to extract:
# 1. All customer names
# 2. All email addresses (and convert to lowercase)
# 3. All phone numbers
# 4. All purchase amounts (remove currency symbol)
```

Solution 3.1

```python
import re

raw_data = """
Customer: Alice Johnson, Email: ALICE@COMPANY.COM, Phone: (555) 123-4567, Purchase: $1,999.9
Customer: Bob Smith, Email: bob_smith@gmail.com, Phone: (555) 987-6543, Purchase: €1,500.50
Customer: Charlie Brown, Email: charlie.b@yahoo.com, Phone: (555) 456-7890, Purchase: $850
"""

# Extract names
names = re.findall(r"Customer: ([A-Za-z\s]+),", raw_data)
names = [n.strip() for n in names]
print("Names:", names)
```

```python
# Extract emails
emails = re.findall(r"Email: ([a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,})", raw_data)
emails_lower = [e.lower() for e in emails]
print("Emails:", emails_lower)

# Extract phone numbers
phones = re.findall(r"\((\d{3})\) (\d{3})-(\d{4})", raw_data)
phones_formatted = [f"({p[0]}) {p[1]}-{p[2]}" for p in phones]
print("Phones:", phones_formatted)

# Extract amounts
amounts = re.findall(r"Purchase: [$€]([0-9,]+\.?\d*)", raw_data)
print("Amounts:", amounts)
```

Part 3: Data Management

Lectures 5-6: Relational Databases (SQL)

5.1 What is a Database?

Before Databases

```
Problem: Data stored in separate files
  accounting_2024.txt
  sales_2024.txt
  customers_2024.txt

Issues:
 Duplication (same customer in multiple files)
 Inconsistency (customer address different in each file)
 Hard to query (manual search through files)
 No relationships (can't connect customer to sales)
```

With Relational Databases

```
Solution: Structured tables with relationships
  Customers table (customer_id, name, email)
  Orders table (order_id, customer_id, amount)
  Relationships (orders linked to customers via customer_id)

Benefits:
 Single source of truth (one customer record)
 Consistency (database enforces rules)
 Easy queries (SQL)
 Relationships (connect tables)
```

5.2 Database Design: Tables and Keys

Tables (Relations)

```
CUSTOMERS table:
customer_id | name    | email               | city
------------|---------|---------------------|--------
1           | Alice   | alice@company.com   | Dublin
2           | Bob     | bob@company.com     | Cork
3           | Charlie | charlie@company.com | Galway

ORDERS table:
order_id | customer_id | date       | amount
---------|-------------|------------|--------
101      | 1           | 2024-01-15 | 1500.00
102      | 2           | 2024-01-16 | 800.00
103      | 1           | 2024-01-17 | 2200.00

The "customer_id" in ORDERS connects to "customer_id" in CUSTOMERS
```

Keys Explained

```
# PRIMARY KEY
# - Unique identifier for each row
# - Can't be NULL
# - Usually an ID number

# FOREIGN KEY
# - Links to another table's primary key
# - Creates relationships between tables

# Example: In ORDERS table:
#   - order_id is PRIMARY KEY (unique for each order)
#   - customer_id is FOREIGN KEY (references CUSTOMERS)
```

5.3 SQL Basics

SELECT: Reading Data

```sql
-- Basic select (get all data)
SELECT * FROM customers;

-- Select specific columns
SELECT name, email FROM customers;

-- Get first 5 rows
SELECT * FROM customers LIMIT 5;
```

```sql
-- With WHERE filter
SELECT * FROM customers WHERE city = 'Dublin';

-- Multiple conditions
SELECT * FROM orders
WHERE customer_id = 1 AND amount > 1000;

-- Sorting
SELECT * FROM customers ORDER BY name ASC;  -- A to Z
SELECT * FROM customers ORDER BY name DESC;  -- Z to A

-- With JOIN (combine two tables)
SELECT
    c.name,
    o.order_id,
    o.amount,
    o.date
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
WHERE c.city = 'Dublin';

-- GROUP BY (summarize data)
SELECT
    customer_id,
    COUNT(*) as order_count,
    SUM(amount) as total_spent
FROM orders
GROUP BY customer_id;
```

INSERT: Adding Data

```sql
-- Add one row
INSERT INTO customers (customer_id, name, email, city)
VALUES (4, 'Diana', 'diana@company.com', 'Limerick');

-- Add multiple rows
INSERT INTO customers (name, email, city) VALUES
('Eve', 'eve@company.com', 'Belfast'),
('Frank', 'frank@company.com', 'Dublin');
```

UPDATE: Modifying Data

```sql
-- Update one customer
UPDATE customers
SET email = 'alice.new@company.com'
WHERE name = 'Alice';

-- Update multiple fields
```

```sql
UPDATE customers
SET city = 'London', email = 'bob@uk.com'
WHERE customer_id = 2;

-- Be careful! This updates ALL rows
-- UPDATE customers SET email = 'test@test.com';  -- DON'T DO THIS!
```

DELETE: Removing Data

```sql
-- Delete one row
DELETE FROM customers WHERE customer_id = 4;

-- Delete multiple rows
DELETE FROM orders WHERE amount < 100;

-- Be careful! This deletes ALL rows
-- DELETE FROM customers;  -- DON'T DO THIS!
```

5.4 Data Types in Databases

```
"""
Common SQL Data Types

INTEGER / BIGINT
- Whole numbers: -100, 0, 1000, 999999999
- Use BIGINT for very large numbers

DECIMAL(10, 2) / NUMERIC
- For money: 1000.50, 99.99
- First number = total digits, second = decimal places

VARCHAR(50) / TEXT
- Text: "Alice", "hello@world.com"
- VARCHAR has max length, TEXT is unlimited

DATE
- Dates: 2024-01-15
- Format: YYYY-MM-DD

TIMESTAMP
- Date and time: 2024-01-15 14:30:45
- Includes timezone info

BOOLEAN
- True/False: is_active, is_premium

UUID
```

```
- Unique identifier: 550e8400-e29b-41d4-a716-446655440000
- Better for distributed systems
"""

# Example: Create table with data types
sql = """
CREATE TABLE products (
    product_id INTEGER PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    price DECIMAL(10, 2),
    in_stock BOOLEAN DEFAULT TRUE,
    created_date DATE,
    description TEXT
);
"""
```

5.5 Connecting Python to Databases

Using pandas (Simplest)

```
import pandas as pd
import sqlite3

# Connect to database (SQLite - file-based, no server needed)
conn = sqlite3.connect("analytics.db")

# Read data into DataFrame
df = pd.read_sql("SELECT * FROM customers", conn)
print(df.head())

# Write DataFrame to database
new_data = pd.DataFrame({
    "name": ["Eva", "Frank"],
    "email": ["eva@company.com", "frank@company.com"],
    "city": ["Belfast", "Dublin"]
})

new_data.to_sql("customers", conn, if_exists="append", index=False)
# if_exists options: "fail" (error if exists), "replace", "append"

conn.close()
```

Using psycopg2 (PostgreSQL)

```
import psycopg2

# Connect
```

```python
conn = psycopg2.connect(
    host="localhost",
    database="analytics_db",
    user="postgres",
    password="your_password"
)

# Create cursor
cursor = conn.cursor()

# Execute query
cursor.execute("SELECT * FROM customers LIMIT 5")
rows = cursor.fetchall()

for row in rows:
    print(row)

# Commit changes (important!)
conn.commit()

# Close
cursor.close()
conn.close()
```

Insert Data via Python

```python
import sqlite3

conn = sqlite3.connect("analytics.db")
cursor = conn.cursor()

# Insert single row
cursor.execute("""
    INSERT INTO customers (name, email, city)
    VALUES (?, ?, ?)
""", ("Grace", "grace@company.com", "Waterford"))

# Insert multiple rows
data = [
    ("Henry", "henry@company.com", "Kilkenny"),
    ("Iris", "iris@company.com", "Sligo")
]
cursor.executemany("""
    INSERT INTO customers (name, email, city)
    VALUES (?, ?, ?)
""", data)
```

```python
conn.commit()
cursor.close()
conn.close()

print("Data inserted successfully!")
```

5.6 Complex Queries

GROUP BY: Aggregate Data

```sql
-- Total sales by region
SELECT
    region,
    COUNT(*) as number_of_orders,
    SUM(amount) as total_sales,
    AVG(amount) as average_sale
FROM orders
GROUP BY region
ORDER BY total_sales DESC;

-- Output (example):
-- region | number_of_orders | total_sales | average_sale
-- North  | 45               | 125000.00   | 2777.78
-- South  | 38               | 110000.00   | 2894.74
```

HAVING: Filter Grouped Results

```sql
-- Find regions with more than 40 orders
SELECT
    region,
    COUNT(*) as order_count,
    SUM(amount) as total_sales
FROM orders
GROUP BY region
HAVING COUNT(*) > 40
ORDER BY order_count DESC;
```

JOIN: Combine Multiple Tables

```sql
-- INNER JOIN (show only matching records)
SELECT
    c.name,
    COUNT(o.order_id) as total_orders,
    SUM(o.amount) as total_spent
FROM customers c
INNER JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.customer_id, c.name;
```

```sql
-- LEFT JOIN (show all customers, even those with no orders)
SELECT
    c.name,
    COUNT(o.order_id) as total_orders,
    COALESCE(SUM(o.amount), 0) as total_spent
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id
GROUP BY c.customer_id, c.name
ORDER BY total_spent DESC;
```

Lecture 7: NoSQL & MongoDB

7.1 When to Use NoSQL

SQL vs NoSQL

```
SQL (Relational):
- Fixed schema (columns defined upfront)
- ACID transactions (reliable)
- Best for: structured data, relationships, strict consistency

NoSQL (Document-based):
- Flexible schema (add fields anytime)
- Eventually consistent
- Best for: unstructured data, rapid scaling, flexible structure

Example situation where NoSQL wins:
- E-commerce: Different products have different fields
  * A laptop product might have: brand, processor, RAM
  * A shirt product might have: size, color, material
  * Hard to fit in fixed SQL schema!

- Social media: Posts can have different content
  * Text posts, photo posts, video posts
  * Each with different fields
```

7.2 MongoDB Basics

Document Structure

```python
# MongoDB stores JSON-like documents
# Each document is like a dictionary in Python

{
    "_id": ObjectId("507f1f77bcf86cd799439011"),  # Unique ID
    "name": "Alice",
    "email": "alice@company.com",
```

```
    "age": 28,
    "purchases": [
        {"product": "Laptop", "price": 1200, "date": "2024-01-15"},
        {"product": "Mouse", "price": 25, "date": "2024-01-16"}
    ],
    "addresses": {
        "home": "123 Main St, Dublin",
        "work": "456 Business Ave, Dublin"
    }
}
```

Connecting to MongoDB in Python

```python
from pymongo import MongoClient

# Connect
client = MongoClient("mongodb://localhost:27017/")
database = client["analytics_db"]
collection = database["customers"]

# Insert document
customer = {
    "name": "Alice",
    "email": "alice@company.com",
    "age": 28,
    "city": "Dublin"
}
result = collection.insert_one(customer)
print(f"Inserted ID: {result.inserted_id}")

# Insert multiple documents
customers = [
    {"name": "Bob", "email": "bob@company.com", "city": "Cork"},
    {"name": "Charlie", "email": "charlie@company.com", "city": "Galway"}
]
collection.insert_many(customers)

# Find documents
# Find all
all_customers = collection.find()
for customer in all_customers:
    print(customer)

# Find one
alice = collection.find_one({"name": "Alice"})
print(alice)
```

```python
# Find with filter
dublin_customers = collection.find({"city": "Dublin"})

# Update
collection.update_one(
    {"name": "Alice"},
    {"$set": {"age": 29}}
)

# Delete
collection.delete_one({"name": "Charlie"})
```

Part 4: Data Processing & ETL

Lecture 9: ETL Pipelines & Data Processing

9.1 What is ETL?

```
ETL = Extract, Transform, Load

EXTRACT:
  Get data from sources (files, APIs, databases)

TRANSFORM:
  Clean, combine, reshape data
  Calculate new columns
  Filter and aggregate

LOAD:
  Save processed data to destination
  Database, CSV file, data warehouse
```

Real Example: Sales Data Pipeline

```
Raw sales from 3 sources:
  CSV file (sales.csv)
  API (internal system)
  Database (legacy system)

EXTRACT:
  Read CSV
  Call API
  Query database

TRANSFORM:
```

```
  Standardize date formats
  Remove duplicates
  Convert currency to EUR
  Combine into one dataset
  Calculate total per customer

LOAD:
  Save to analytics database
  Create reports
```

9.2 Building a Data Pipeline with Pandas

Pipeline Example: Customer Sales Analysis

```python
import pandas as pd
import numpy as np
from datetime import datetime

class SalesPipeline:
    """ETL Pipeline for sales data"""

    def __init__(self, raw_file, output_file):
        self.raw_file = raw_file
        self.output_file = output_file
        self.df = None

    def extract(self):
        """Step 1: Extract data"""
        print("Extracting data...")
        self.df = pd.read_csv(self.raw_file)
        print(f"  Loaded {len(self.df)} rows")
        return self

    def transform(self):
        """Step 2: Transform data"""
        print("Transforming data...")

        # Clean column names (remove spaces, lowercase)
        self.df.columns = self.df.columns.str.lower().str.replace(" ", "_")

        # Handle missing values
        self.df["email"].fillna("unknown@company.com", inplace=True)
        self.df["phone"] = self.df["phone"].fillna("N/A")

        # Remove duplicates
        self.df.drop_duplicates(subset=["customer_id"], inplace=True)
```

```python
        # Convert date column
        self.df["date"] = pd.to_datetime(self.df["date"])

        # Extract year and month
        self.df["year"] = self.df["date"].dt.year
        self.df["month"] = self.df["date"].dt.month

        # Calculate if high-value customer (>$5000 total)
        self.df["is_high_value"] = self.df["amount"] > 5000

        # Remove rows with missing critical data
        self.df.dropna(subset=["customer_id", "amount"], inplace=True)

        print(f"  After transformation: {len(self.df)} rows")
        return self

    def load(self):
        """Step 3: Load data"""
        print("Loading data...")
        self.df.to_csv(self.output_file, index=False)
        print(f"  Saved to {self.output_file}")
        return self

    def run(self):
        """Execute full pipeline"""
        self.extract()
        self.transform()
        self.load()
        return self.df

# Usage
pipeline = SalesPipeline("raw_sales.csv", "clean_sales.csv")
clean_df = pipeline.run()
print("\nSample of clean data:")
print(clean_df.head())
```

9.3 Data Cleaning Techniques

Handling Missing Values

```python
import pandas as pd

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

# Check missing data
```

```python
print(df.isnull().sum())
# Output: Shows count of missing values per column

# Method 1: Drop missing rows
df.dropna(inplace=True)  # Remove any row with missing data

# Method 2: Fill with specific value
df["age"].fillna(30, inplace=True)  # Age gets filled with 30

# Method 3: Fill with mean
df["salary"].fillna(df["salary"].mean(), inplace=True)

# Method 4: Forward fill (use previous value)
df.fillna(method="ffill", inplace=True)

# Method 5: Drop entire column if too many missing
threshold = len(df) * 0.3  # if > 30% missing
df.dropna(axis=1, thresh=threshold, inplace=True)
```

Data Type Conversion

```python
import pandas as pd

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

# Check data types
print(df.dtypes)

# Convert types
df["customer_id"] = df["customer_id"].astype(int)
df["amount"] = df["amount"].astype(float)
df["is_active"] = df["is_active"].astype(bool)

# Date conversion
df["date"] = pd.to_datetime(df["date"], format="%d/%m/%Y")

# Categorical (save memory for repeated values)
df["region"] = df["region"].astype("category")
```

Removing Duplicates

```python
import pandas as pd

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

# Check duplicates
print(df.duplicated().sum())  # How many duplicate rows
```

```python
# Remove all duplicate rows
df.drop_duplicates(inplace=True)

# Remove duplicates based on specific column
df.drop_duplicates(subset=["customer_id"], inplace=True)

# Keep first/last occurrence
df.drop_duplicates(subset=["email"], keep="first", inplace=True)
```

Standardizing Data

```python
import pandas as pd

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

# Standardize text (lowercase, strip spaces)
df["email"] = df["email"].str.lower().str.strip()
df["name"] = df["name"].str.title().str.strip()

# Standardize dates
df["date"] = pd.to_datetime(df["date"]).dt.strftime("%Y-%m-%d")

# Standardize numbers (currency)
df["price"] = df["price"].str.replace("$", "").str.replace(",", "").astype(float)

# Standardize categories
df["region"] = df["region"].map({
    "N": "North",
    "S": "South",
    "E": "East",
    "W": "West"
})
```

9.4 Data Aggregation

GROUP BY Operations

```python
import pandas as pd

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

# Sales by region
region_sales = df.groupby("region").agg({
    "amount": ["sum", "mean", "count"],
    "customer_id": "count"
}).round(2)
```

```python
print(region_sales)
# Output:
#           amount                  customer_id
#             sum    mean count      count
# region
# North  150000.00 3333.33 45       45
# South  120000.00 3157.89 38       38

# Multiple aggregations
summary = df.groupby(["region", "month"]).agg({
    "amount": "sum",
    "customer_id": "nunique",
    "date": "count"
}).rename(columns={"date": "transaction_count"})

# Custom aggregations
def custom_stats(amounts):
    """Calculate custom statistics"""
    return pd.Series({
        "total": amounts.sum(),
        "avg": amounts.mean(),
        "std": amounts.std(),
        "range": amounts.max() - amounts.min()
    })

df.groupby("region")["amount"].apply(custom_stats)
```

Pivot Tables

```python
import pandas as pd

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

# Create pivot table
pivot = df.pivot_table(
    values="amount",  # What to aggregate
    index="region",  # Rows
    columns="month",  # Columns
    aggfunc="sum"  # How to aggregate
)

print(pivot)
# Output:
# month    1        2        3
# region
# North    50000    55000    45000
# South    40000    38000    42000
```

```python
# Multiple aggregations
pivot = df.pivot_table(
    values="amount",
    index="region",
    columns="product",
    aggfunc=["sum", "count", "mean"]
)
```

Exercises: Part 4

Exercise 4.1: Build Complete ETL Pipeline

```python
# Raw data (messy):
raw_data = """
Date,CustomerID,Name,Email,Amount,City
2024-01-15,1,alice johnson,ALICE@COMPANY.COM,1500,Dublin
2024-01-16,2, bob smith ,bob@company.com,800,Cork
2024-01-17,1,Alice Johnson,alice@company.com,2200,Dublin
2024-01-18,3,charlie,charlie@example.com,,Galway
2024-01-19,2,Bob Smith,bob@company.com,950,Cork
"""

# TODO: Create a pipeline that:
# 1. Loads the CSV
# 2. Cleans names (title case, strip spaces)
# 3. Standardizes email (lowercase)
# 4. Removes duplicates
# 5. Handles missing values
# 6. Calculates customer totals
# 7. Saves clean version
# 8. Prints summary statistics
```

Solution 4.1

```python
import pandas as pd
import io

raw_data = """
Date,CustomerID,Name,Email,Amount,City
2024-01-15,1,alice johnson,ALICE@COMPANY.COM,1500,Dublin
2024-01-16,2, bob smith ,bob@company.com,800,Cork
2024-01-17,1,Alice Johnson,alice@company.com,2200,Dublin
2024-01-18,3,charlie,charlie@example.com,,Galway
2024-01-19,2,Bob Smith,bob@company.com,950,Cork
"""
```

```python
# Load
df = pd.read_csv(io.StringIO(raw_data))

# Clean names
df["Name"] = df["Name"].str.title().str.strip()

# Standardize email
df["Email"] = df["Email"].str.lower().str.strip()

# Remove duplicates (keep first)
df = df.drop_duplicates(subset=["CustomerID"], keep="first")

# Handle missing amounts
df["Amount"].fillna(0, inplace=True)

# Calculate customer totals
customer_totals = df.groupby("CustomerID").agg({
    "Name": "first",
    "Amount": "sum",
    "City": "first"
}).rename(columns={"Amount": "TotalSpent"})

print("Customer Summary:")
print(customer_totals)

print("\nBasic Statistics:")
print(f"Total Customers: {len(customer_totals)}")
print(f"Total Revenue: ${customer_totals['TotalSpent'].sum():,.2f}")
print(f"Average per Customer: ${customer_totals['TotalSpent'].mean():,.2f}")
```

Part 5: Visualization & Communication

Lecture 10-11: Data Visualization

10.1 Why Visualization Matters

The Power of Visualization

Same data, different presentation:

```
BAD: "Q4 revenue is $1,200,000 in North, $980,000 in South,
      $1,500,000 in East, $750,000 in West"

GOOD: [Simple bar chart showing same data]
      → Instantly clear: East is strongest, West is weakest
```

```
→ Easy to compare regions
→ Professional appearance
```

Quick Stats: - 90% of information processed by brain is visual - People remember images 65% better than words - Visualizations are processed 60,000x faster than text

10.2 Chart Types & When to Use Them

Bar Chart: Comparing Categories

```python
import matplotlib.pyplot as plt

# When to use: Compare values across categories
regions = ["North", "South", "East", "West"]
sales = [150000, 120000, 180000, 95000]

plt.figure(figsize=(10, 6))
plt.bar(regions, sales, color=["#1f77b4", "#ff7f0e", "#2ca02c", "#d62728"])
plt.xlabel("Region", fontsize=12)
plt.ylabel("Sales ($)", fontsize=12)
plt.title("Sales by Region - Q4 2024", fontsize=14, fontweight="bold")
plt.grid(axis="y", alpha=0.3)

# Add values on bars
for i, v in enumerate(sales):
    plt.text(i, v + 3000, f"${v:,.0f}", ha="center", va="bottom")

plt.tight_layout()
plt.show()
```

Line Chart: Trends Over Time

```python
import matplotlib.pyplot as plt

# When to use: Show how values change over time
months = ["Jan", "Feb", "Mar", "Apr", "May", "Jun"]
revenue = [100000, 115000, 98000, 125000, 140000, 155000]

plt.figure(figsize=(10, 6))
plt.plot(months, revenue, marker="o", linewidth=2, markersize=8, color="#1f77b4")
plt.xlabel("Month", fontsize=12)
plt.ylabel("Revenue ($)", fontsize=12)
plt.title("Monthly Revenue Trend", fontsize=14, fontweight="bold")
plt.grid(alpha=0.3)
plt.tight_layout()
plt.show()
```

Scatter Plot: Relationship Between Two Variables

```
import matplotlib.pyplot as plt

# When to use: Show correlation between two variables
age = [25, 28, 32, 35, 40, 45, 50]
salary = [50000, 60000, 75000, 80000, 95000, 105000, 120000]

plt.figure(figsize=(10, 6))
plt.scatter(age, salary, s=100, alpha=0.6, color="#2ca02c")
plt.xlabel("Age", fontsize=12)
plt.ylabel("Salary ($)", fontsize=12)
plt.title("Age vs Salary", fontsize=14, fontweight="bold")
plt.grid(alpha=0.3)
plt.tight_layout()
plt.show()

# Notice: Clear positive correlation between age and salary
```

Histogram: Distribution of Values

```
import matplotlib.pyplot as plt
import numpy as np

# When to use: Show distribution of a single variable
# Fake data: customer ages
ages = np.random.normal(35, 10, 1000)

plt.figure(figsize=(10, 6))
plt.hist(ages, bins=30, color="#ff7f0e", edgecolor="black", alpha=0.7)
plt.xlabel("Age", fontsize=12)
plt.ylabel("Frequency", fontsize=12)
plt.title("Distribution of Customer Ages", fontsize=14, fontweight="bold")
plt.grid(axis="y", alpha=0.3)
plt.tight_layout()
plt.show()
```

Pie Chart: Parts of a Whole

```
import matplotlib.pyplot as plt

# When to use: Show percentage breakdown
regions = ["North", "South", "East", "West"]
sales = [150000, 120000, 180000, 95000]

plt.figure(figsize=(8, 8))
colors = ["#1f77b4", "#ff7f0e", "#2ca02c", "#d62728"]
wedges, texts, autotexts = plt.pie(
```

```python
    sales,
    labels=regions,
    autopct="%1.1f%%",
    colors=colors,
    startangle=90
)

# Make percentage text bold and white
for autotext in autotexts:
    autotext.set_color("white")
    autotext.set_fontweight("bold")

plt.title("Sales Distribution by Region", fontsize=14, fontweight="bold")
plt.tight_layout()
plt.show()

# WARNING: Only use pie charts for 2-5 categories
# Humans are bad at comparing pie slices!
```

10.3 Visualization Best Practices

Good Visualization Checklist

```python
import matplotlib.pyplot as plt

def create_professional_chart():
    """Example of best practices"""

    regions = ["North", "South", "East", "West"]
    sales = [150000, 120000, 180000, 95000]

    plt.figure(figsize=(12, 7))

    # 1. Clear, descriptive title
    plt.title("Regional Sales Performance - Q4 2024",
              fontsize=16, fontweight="bold", pad=20)

    # 2. Professional color scheme
    colors = ["#1f77b4", "#ff7f0e", "#2ca02c", "#d62728"]
    bars = plt.bar(regions, sales, color=colors, width=0.7, edgecolor="black")

    # 3. Labeled axes with units
    plt.xlabel("Region", fontsize=12, fontweight="bold")
    plt.ylabel("Sales (USD)", fontsize=12, fontweight="bold")

    # 4. Grid for readability
```

```
    plt.grid(axis="y", alpha=0.3, linestyle="--")

    # 5. Values on bars
    for i, (region, value) in enumerate(zip(regions, sales)):
        plt.text(i, value + 5000, f"${value/1000:.0f}K",
                ha="center", va="bottom", fontweight="bold")

    # 6. Clean formatting
    ax = plt.gca()
    ax.spines["top"].set_visible(False)
    ax.spines["right"].set_visible(False)

    # 7. Proper sizing
    plt.tight_layout()

    return plt

# Call it
create_professional_chart().show()
```

Common Mistakes to Avoid

```
 3D effects (makes comparison hard)
 Rainbow colors (no meaning)
 No title or labels
 Too many data series (confusing)
 Wrong chart type (pie for trends, line for categories)
 Truncated axes (distorts data)
 Too much information (one chart = one message)
 Poor color choices (red/green for color-blind people)

 Do: Keep it simple
 Do: One chart = one message
 Do: Use professional colors
 Do: Label everything
 Do: Include units
 Do: Add data values when helpful
```

10.4 Matplotlib & Seaborn Practical Guide

Matplotlib: Foundation Library

```
import matplotlib.pyplot as plt

# Basic structure
fig, ax = plt.subplots(figsize=(10, 6))
```

```python
# Plot data
x = [1, 2, 3, 4, 5]
y = [10, 24, 36, 18, 42]
ax.plot(x, y, marker="o", linewidth=2, markersize=8)

# Customize
ax.set_xlabel("X Axis")
ax.set_ylabel("Y Axis")
ax.set_title("My Chart")
ax.grid(alpha=0.3)

plt.show()
```

Seaborn: High-Level Styling

```python
import seaborn as sns
import pandas as pd

# Seaborn makes pretty charts with less code
df = pd.DataFrame({
    "month": ["Jan", "Feb", "Mar"],
    "sales": [100, 120, 105],
    "region": ["North", "North", "North"]
})

sns.set_style("whitegrid")  # Style
sns.set_palette("husl")  # Color palette

plt.figure(figsize=(10, 6))
sns.barplot(data=df, x="month", y="sales", hue="region")
plt.title("Sales by Month")
plt.show()
```

10.5 Multiple Subplots & Dashboards

Create Multi-Chart Dashboard

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

# Sample data
df = pd.read_csv("sales.csv")

# Create dashboard (2x2 grid)
fig, axes = plt.subplots(2, 2, figsize=(14, 10))
fig.suptitle("Sales Dashboard - Q4 2024", fontsize=16, fontweight="bold")
```

```
# Chart 1: Sales by region (top-left)
region_sales = df.groupby("region")["amount"].sum().sort_values(ascending=False)
axes[0, 0].bar(region_sales.index, region_sales.values, color="#1f77b4")
axes[0, 0].set_title("Total Sales by Region")
axes[0, 0].set_ylabel("Sales ($)")

# Chart 2: Sales trend (top-right)
monthly = df.groupby("month")["amount"].sum()
axes[0, 1].plot(monthly.index, monthly.values, marker="o", color="#2ca02c", linewidth=2)
axes[0, 1].set_title("Monthly Sales Trend")
axes[0, 1].set_ylabel("Sales ($)")

# Chart 3: Top customers (bottom-left)
top_customers = df.groupby("customer")["amount"].sum().nlargest(5)
axes[1, 0].barh(top_customers.index, top_customers.values, color="#ff7f0e")
axes[1, 0].set_title("Top 5 Customers")
axes[1, 0].set_xlabel("Sales ($)")

# Chart 4: Summary table (bottom-right)
summary = df.groupby("region").agg({
    "amount": ["sum", "count", "mean"]
}).round(2)
axes[1, 1].axis("off")  # Remove axes
summary_text = summary.to_string()
axes[1, 1].text(0.1, 0.9, summary_text, fontfamily="monospace", fontsize=9, va="top")
axes[1, 1].set_title("Regional Summary")

plt.tight_layout()
plt.show()
```

10.6 Interactive Visualizations with Plotly

Plotly: Interactive, Web-Ready Charts

```
import plotly.express as px
import plotly.graph_objects as go
import pandas as pd

# Sample data
df = pd.read_csv("sales.csv")

# Interactive bar chart
fig = px.bar(
    df.groupby("region").agg({"amount": "sum"}).reset_index(),
    x="region",
    y="amount",
```

```python
    title="Sales by Region",
    labels={"amount": "Sales ($)"},
    color="region"
)
fig.show()

# Interactive line chart
fig = px.line(
    df.groupby("date")["amount"].sum().reset_index(),
    x="date",
    y="amount",
    title="Sales Over Time",
    markers=True
)
fig.show()

# Interactive scatter with hover info
fig = px.scatter(
    df,
    x="age",
    y="salary",
    hover_name="name",
    title="Age vs Salary",
    trendline="ols"  # Add trend line
)
fig.show()
```

10.7 Gestalt Principles & Color Theory

Gestalt Principles: How Humans Perceive Visuals

```python
"""
Gestalt Principles make visualizations easier to understand:

1. PROXIMITY: Elements close together are seen as related
     Group related data together

2. SIMILARITY: Elements with same color/shape are grouped
     Use same color for same category

3. CONTINUITY: Human eye follows smooth paths
     Use lines to show trends

4. CLOSURE: Brain completes incomplete shapes
     Incomplete shapes take less space but are understood
```

```python
5. FIGURE-GROUND: Separate foreground from background
     Make important data stand out
"""

# Example: Apply Gestalt principles
import matplotlib.pyplot as plt

regions = ["North", "South", "East", "West"]
sales = [150000, 120000, 180000, 95000]

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))

# LEFT: Principle of Proximity (group by color)
colors = ["#1f77b4", "#1f77b4", "#2ca02c", "#2ca02c"]  # Same colors = same group
ax1.bar(regions, sales, color=colors, edgecolor="black", linewidth=1.5)
ax1.set_title("Proximity: Group by Color")
ax1.set_ylabel("Sales ($)")

# RIGHT: Using Similarity
all_same_color = ["#1f77b4" if s > 140000 else "#808080" for s in sales]
ax2.bar(regions, sales, color=all_same_color)
ax2.set_title("Similarity: Highlight High Performers")
ax2.set_ylabel("Sales ($)")

plt.tight_layout()
plt.show()
```

Color Theory

```python
"""
Color Guidelines for Charts:

DON'T:
 Rainbow (no meaning, hard for colorblind)
 Red/Green together (colorblind people can't see)
 Too many colors (confusing)

DO:
 Use 2-5 colors max
 Use colorblind-friendly palettes
 Blue = positive, Red = negative
 Darker = more important

Good Color Palettes:
- Blue, Orange, Green, Red (colorblind-friendly)
- Viridis (scientific, works for colorblind)
- Sequential: light to dark (for continuous data)
```

```
"""

import matplotlib.pyplot as plt
import seaborn as sns

# Set colorblind-friendly palette
sns.set_palette("colorblind")

# Now all plots use friendly colors
```

Exercises: Part 5

Exercise 5.1: Create Professional Dashboard

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

# Sales data
data = {
    "date": ["2024-01-15", "2024-01-16", "2024-01-17", "2024-01-18", "2024-01-19"],
    "region": ["North", "South", "North", "East", "West"],
    "sales": [15000, 12000, 18000, 22000, 9500],
    "product": ["Laptop", "Phone", "Tablet", "Laptop", "Phone"]
}
df = pd.DataFrame(data)

# TODO: Create a 2x2 dashboard showing:
# 1. Sales by region (bar chart)
# 2. Sales by date (line chart)
# 3. Sales by product (horizontal bar)
# 4. Summary statistics table
```

Solution 5.1

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

data = {
    "date": ["2024-01-15", "2024-01-16", "2024-01-17", "2024-01-18", "2024-01-19"],
    "region": ["North", "South", "North", "East", "West"],
    "sales": [15000, 12000, 18000, 22000, 9500],
    "product": ["Laptop", "Phone", "Tablet", "Laptop", "Phone"]
}
df = pd.DataFrame(data)

fig, axes = plt.subplots(2, 2, figsize=(14, 10))
fig.suptitle("Sales Dashboard", fontsize=16, fontweight="bold")
```

```python
# Chart 1: Sales by region
region_sales = df.groupby("region")["sales"].sum()
axes[0, 0].bar(region_sales.index, region_sales.values, color="#1f77b4")
axes[0, 0].set_title("Sales by Region")
axes[0, 0].set_ylabel("Sales ($)")

# Chart 2: Sales by date
df["date"] = pd.to_datetime(df["date"])
axes[0, 1].plot(df["date"], df["sales"], marker="o", color="#2ca02c", linewidth=2)
axes[0, 1].set_title("Sales by Date")
axes[0, 1].set_ylabel("Sales ($)")
axes[0, 1].tick_params(axis="x", rotation=45)

# Chart 3: Sales by product
product_sales = df.groupby("product")["sales"].sum()
axes[1, 0].barh(product_sales.index, product_sales.values, color="#ff7f0e")
axes[1, 0].set_title("Sales by Product")
axes[1, 0].set_xlabel("Sales ($)")

# Chart 4: Summary
axes[1, 1].axis("off")
summary_text = f"""
Total Sales: ${df['sales'].sum():,}
Avg Sale: ${df['sales'].mean():,.0f}
Max Sale: ${df['sales'].max():,}
Min Sale: ${df['sales'].min():,}
Total Transactions: {len(df)}
"""
axes[1, 1].text(0.1, 0.9, summary_text, fontfamily="monospace", fontsize=11, va="top")

plt.tight_layout()
plt.show()
```

Part 6: Big Data & Advanced Topics

Lecture 12: Big Data & PySpark

12.1 When You Need Big Data Tools

Problem: Size

```
DataFrame limitations:
- Fits in memory (RAM)
- Single machine
```

```
- Typical size: 100MB to 10GB

Real-world data sizes:
- Netflix: Petabytes (1000s of TB) ← Need Spark
- Facebook: Exabytes ← Need Spark + Hadoop
- Your startup: Maybe 1TB → Use pandas

When to use PySpark:
 Data > 10GB
 Need distributed processing
 Real-time streaming
 Machine learning on massive datasets
```

12.2 Introduction to Spark

What is Spark?

```
Spark = Distributed computing framework
- Runs on cluster of machines
- Divides work across multiple nodes
- Much faster than pandas for huge datasets

Architecture:
  Driver (your laptop)
  Cluster (many machines)
     Worker 1 (processes part of data)
     Worker 2 (processes part of data)
     Worker 3 (processes part of data)

Results combined and returned to driver.
```

12.3 PySpark Basics

Setting Up Spark

```python
from pyspark.sql import SparkSession

# Create Spark session
spark = SparkSession.builder \
    .appName("Analytics") \
    .master("local[*]") \
    .getOrCreate()

print(f"Spark Version: {spark.version}")
```

Read and Write Data

```python
from pyspark.sql import SparkSession
```

```python
spark = SparkSession.builder.appName("MyApp").getOrCreate()

# Read CSV
df_spark = spark.read.csv("sales.csv", header=True, inferSchema=True)

# Read JSON
df_json = spark.read.json("data.json")

# Read database
df_db = spark.read \
    .format("jdbc") \
    .option("url", "jdbc:postgresql://localhost:5432/mydb") \
    .option("dbtable", "customers") \
    .load()

# Show data
df_spark.show()
df_spark.printSchema()

# Write CSV
df_spark.write.csv("output.csv", header=True, mode="overwrite")

# Write database
df_spark.write \
    .format("jdbc") \
    .option("url", "jdbc:postgresql://localhost:5432/mydb") \
    .option("dbtable", "results") \
    .save()
```

Basic Operations

```python
# Select columns
df_spark.select("name", "email", "salary").show()

# Filter rows
df_spark.filter(df_spark["salary"] > 50000).show()

# Add computed column
df_spark = df_spark.withColumn("tax", df_spark["salary"] * 0.2)

# Group by and aggregate
df_spark.groupby("region").agg({"salary": "sum"}).show()

# Sort
df_spark.sort("salary", ascending=False).show()
```

```python
# Distinct
df_spark.select("region").distinct().show()

# Count
df_spark.count()

# Join tables
df_customers = spark.read.csv("customers.csv", header=True)
df_orders = spark.read.csv("orders.csv", header=True)

joined = df_customers.join(df_orders, on="customer_id")
joined.show()
```

Complete Project Solutions

Project 1: E-Commerce Sales Analytics

Business Problem: An e-commerce company wants to understand sales patterns, identify top customers, and predict trends.

Data Sources: 1. Sales CSV file 2. Customer database 3. Product catalog JSON

Complete Solution:

```python
# ===== COMPLETE E-COMMERCE ANALYTICS PROJECT =====

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from datetime import datetime, timedelta
import json

class EcommerceDashboard:
    """Complete e-commerce analytics pipeline"""

    def __init__(self):
        self.sales_df = None
        self.customers_df = None
        self.products_df = None
        self.merged_df = None

    # ===== EXTRACT PHASE =====
    def load_sales_data(self, csv_path):
```

```python
        """Load sales from CSV"""
        print("Loading sales data...")
        self.sales_df = pd.read_csv(csv_path)
        self.sales_df["date"] = pd.to_datetime(self.sales_df["date"])
        print(f"  Loaded {len(self.sales_df)} sales records")
        return self

    def load_customers_data(self, json_path):
        """Load customer data from JSON"""
        print("Loading customer data...")
        with open(json_path, "r") as file:
            data = json.load(file)
        self.customers_df = pd.DataFrame(data["customers"])
        print(f"  Loaded {len(self.customers_df)} customer records")
        return self

    def load_products_data(self, json_path):
        """Load product data from JSON"""
        print("Loading product data...")
        with open(json_path, "r") as file:
            data = json.load(file)
        self.products_df = pd.DataFrame(data["products"])
        print(f"  Loaded {len(self.products_df)} products")
        return self

    # ===== TRANSFORM PHASE =====
    def clean_data(self):
        """Clean and standardize data"""
        print("Cleaning data...")

        # Remove duplicates
        self.sales_df.drop_duplicates(inplace=True)

        # Handle missing values
        self.sales_df["discount"].fillna(0, inplace=True)

        # Remove invalid records (negative amounts)
        self.sales_df = self.sales_df[self.sales_df["amount"] > 0]

        # Standardize customer names
        self.customers_df["name"] = self.customers_df["name"].str.title().str.strip()
        self.customers_df["email"] = self.customers_df["email"].str.lower()

        print("  Data cleaned")
        return self
```

```python
def merge_data(self):
    """Merge all datasets"""
    print("Merging datasets...")

    # Merge sales with customers
    self.merged_df = self.sales_df.merge(
        self.customers_df,
        on="customer_id",
        how="left"
    )

    # Merge with products
    self.merged_df = self.merged_df.merge(
        self.products_df,
        on="product_id",
        how="left"
    )

    print(f"  Merged to {len(self.merged_df)} records")
    return self

def calculate_metrics(self):
    """Add calculated columns"""
    print("Calculating metrics...")

    # Revenue after discount
    self.merged_df["final_amount"] = self.merged_df["amount"] * (1 - self.merged_df["dis

    # Extract date components
    self.merged_df["year"] = self.merged_df["date"].dt.year
    self.merged_df["month"] = self.merged_df["date"].dt.month
    self.merged_df["week"] = self.merged_df["date"].dt.isocalendar().week
    self.merged_df["day_of_week"] = self.merged_df["date"].dt.day_name()

    # Customer lifetime value
    self.merged_df["is_high_value"] = self.merged_df["final_amount"] > self.merged_df["f

    print("  Metrics calculated")
    return self

# ===== ANALYSIS PHASE =====
def get_summary_stats(self):
    """Generate summary statistics"""
    print("\n" + "="*50)
    print("SUMMARY STATISTICS")
    print("="*50)
```

```
        print(f"\nTotal Sales: ${self.merged_df['final_amount'].sum():,.2f}")
        print(f"Average Order: ${self.merged_df['final_amount'].mean():,.2f}")
        print(f"Median Order: ${self.merged_df['final_amount'].median():,.2f}")
        print(f"Total Transactions: {len(self.merged_df):,}")
        print(f"Unique Customers: {self.merged_df['customer_id'].nunique():,}")
        print(f"Unique Products: {self.merged_df['product_id'].nunique():,}")

        return self

    def get_top_metrics(self):
        """Get top entities"""
        print("\n" + "="*50)
        print("TOP PERFORMERS")
        print("="*50)

        # Top customers
        top_customers = self.merged_df.groupby("name")["final_amount"].sum().nlargest(5)
        print("\nTop 5 Customers by Revenue:")
        for name, amount in top_customers.items():
            print(f"  {name}: ${amount:,.2f}")

        # Top products
        top_products = self.merged_df.groupby("product_name")["final_amount"].sum().nlargest
        print("\nTop 5 Products:")
        for product, amount in top_products.items():
            print(f"  {product}: ${amount:,.2f}")

        # Best day
        daily_sales = self.merged_df.groupby("date")["final_amount"].sum()
        best_day = daily_sales.idxmax()
        print(f"\nBest Sales Day: {best_day.strftime('%Y-%m-%d')} (${daily_sales[best_day]:

        return self

    # ===== VISUALIZATION PHASE =====
    def create_dashboard(self):
        """Create comprehensive dashboard"""
        print("\nCreating dashboard...")

        fig = plt.figure(figsize=(16, 12))
        gs = fig.add_gridspec(3, 3, hspace=0.3, wspace=0.3)

        # 1. Daily sales trend
        ax1 = fig.add_subplot(gs[0, :2])
        daily_sales = self.merged_df.groupby("date")["final_amount"].sum()
```

```python
ax1.plot(daily_sales.index, daily_sales.values, color="#1f77b4", linewidth=2)
ax1.fill_between(daily_sales.index, daily_sales.values, alpha=0.3, color="#1f77b4")
ax1.set_title("Daily Sales Trend", fontsize=12, fontweight="bold")
ax1.set_ylabel("Sales ($)")
ax1.grid(alpha=0.3)

# 2. Sales by day of week
ax2 = fig.add_subplot(gs[0, 2])
day_sales = self.merged_df.groupby("day_of_week")["final_amount"].mean()
day_order = ["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Su
day_sales = day_sales.reindex(day_order)
ax2.bar(range(7), day_sales.values, color="#2ca02c")
ax2.set_xticks(range(7))
ax2.set_xticklabels([d[:3] for d in day_order], rotation=45)
ax2.set_title("Avg Sales by Day", fontsize=12, fontweight="bold")

# 3. Top 10 customers
ax3 = fig.add_subplot(gs[1, :2])
top_cust = self.merged_df.groupby("name")["final_amount"].sum().nlargest(10)
ax3.barh(top_cust.index, top_cust.values, color="#ff7f0e")
ax3.set_xlabel("Revenue ($)")
ax3.set_title("Top 10 Customers", fontsize=12, fontweight="bold")

# 4. Top 5 products
ax4 = fig.add_subplot(gs[1, 2])
top_prod = self.merged_df.groupby("product_name")["final_amount"].sum().nlargest(5)
ax4.bar(range(len(top_prod)), top_prod.values, color="#d62728")
ax4.set_xticks(range(len(top_prod)))
ax4.set_xticklabels([p[:10] for p in top_prod.index], rotation=45, ha="right")
ax4.set_ylabel("Revenue ($)")
ax4.set_title("Top 5 Products", fontsize=12, fontweight="bold")

# 5. Revenue distribution
ax5 = fig.add_subplot(gs[2, 0])
ax5.hist(self.merged_df["final_amount"], bins=30, color="#9467bd", edgecolor="black"
ax5.set_xlabel("Order Amount ($)")
ax5.set_ylabel("Frequency")
ax5.set_title("Order Amount Distribution", fontsize=12, fontweight="bold")

# 6. Monthly comparison
ax6 = fig.add_subplot(gs[2, 1])
monthly = self.merged_df.groupby("month")["final_amount"].sum()
ax6.plot(monthly.index, monthly.values, marker="o", color="#17becf", linewidth=2, ma
ax6.set_xlabel("Month")
ax6.set_ylabel("Revenue ($)")
ax6.set_title("Monthly Revenue", fontsize=12, fontweight="bold")
```

```
        ax6.grid(alpha=0.3)

        # 7. Summary box
        ax7 = fig.add_subplot(gs[2, 2])
        ax7.axis("off")
        summary_text = f"""
METRICS SUMMARY

Total Revenue: ${self.merged_df['final_amount'].sum():,.0f}
Avg Order: ${self.merged_df['final_amount'].mean():,.0f}
Total Orders: {len(self.merged_df):,}
Unique Customers: {self.merged_df['customer_id'].nunique():,}
Repeat Rate: {(1 - self.merged_df['customer_id'].nunique()/len(self.merged_df))*100:.1f}%
        """
        ax7.text(0.1, 0.9, summary_text, fontfamily="monospace", fontsize=9, va="top")

        fig.suptitle("E-Commerce Sales Dashboard", fontsize=16, fontweight="bold", y=0.995)
        plt.show()

        return self

    # ===== EXPORT PHASE =====
    def save_results(self, output_dir="results"):
        """Save analysis results"""
        print(f"\nSaving results to {output_dir}/...")
        import os
        os.makedirs(output_dir, exist_ok=True)

        # Save processed data
        self.merged_df.to_csv(f"{output_dir}/processed_sales.csv", index=False)
        print(f"   Saved processed data")

        # Save summary report
        with open(f"{output_dir}/summary_report.txt", "w") as f:
            f.write("="*50 + "\n")
            f.write("E-COMMERCE SALES REPORT\n")
            f.write("="*50 + "\n\n")

            f.write(f"Total Revenue: ${self.merged_df['final_amount'].sum():,.2f}\n")
            f.write(f"Total Transactions: {len(self.merged_df):,}\n")
            f.write(f"Unique Customers: {self.merged_df['customer_id'].nunique():,}\n")
            f.write(f"Average Order Value: ${self.merged_df['final_amount'].mean():,.2f}\n\n

            f.write("TOP 5 CUSTOMERS:\n")
            top_cust = self.merged_df.groupby("name")["final_amount"].sum().nlargest(5)
            for name, amount in top_cust.items():
```

```
                f.write(f"  {name}: ${amount:,.2f}\n")

        print(f"   Saved summary report")

        return self

    def run_complete_pipeline(self, sales_csv, customers_json, products_json):
        """Execute entire pipeline"""
        print("\n STARTING COMPLETE ANALYTICS PIPELINE\n")

        return (self
            .load_sales_data(sales_csv)
            .load_customers_data(customers_json)
            .load_products_data(products_json)
            .clean_data()
            .merge_data()
            .calculate_metrics()
            .get_summary_stats()
            .get_top_metrics()
            .create_dashboard()
            .save_results()
        )

# ===== USAGE EXAMPLE =====
if __name__ == "__main__":
    # Create dashboard object
    dashboard = EcommerceDashboard()

    # Run complete pipeline
    # (You would provide actual file paths)
    # dashboard.run_complete_pipeline(
    #     sales_csv="sales_2024.csv",
    #     customers_json="customers.json",
    #     products_json="products.json"
    # )
```

Conclusion

You've now learned **comprehensive analytics programming and data visualization** covering:

Foundations - Python basics, data types, control flow, functions
Data I/O - Reading/writing files, CSV, JSON, APIs, web scraping
Data Management - Relational databases, SQL, NoSQL
Processing - ETL pipelines, data cleaning, aggregations

Visualization - Chart types, dashboards, best practices
Big Data - Spark, distributed computing concepts

Next Steps

1. **Practice**: Build your own projects with real data
2. **Specialize**: Choose your path:
   - Data Analysis → Deep dive into pandas & statistics
   - Data Engineering → Master databases & pipelines
   - Data Science → Learn machine learning (scikit-learn)
3. **Build Portfolio**: Create projects to showcase on GitHub
4. **Stay Updated**: Follow analytics blogs, take courses on Coursera/DataCamp

Resources

- Python: https://python.org
- Pandas: https://pandas.pydata.org
- Matplotlib: https://matplotlib.org
- SQL: https://w3schools.com/sql
- Spark: https://spark.apache.org

Good luck on your analytics journey!