Python Advanced Programming

PCPP1 Training

Training Manual

Contents

Day 1: Advanced Object-Oriented Programming

Objective:

Participants will learn advanced concepts in object-oriented programming (OOP) with Python, focusing on modeling real-world manufacturing scenarios. The exercises will use datasets such as production_line_data.csv to create realistic use cases involving machines, production runs, and products.

Morning Session: Introduction to Advanced OOP Concepts

---

1. Core Topics

- Classes, Instances, Attributes, and Methods
- Inheritance, Polymorphism, and Composition
- Real-world Modeling: Machines, Products, and Production Runs

---

2. Lecture Outline

- **Concepts of Classes and Instances**:
    - Explain class structure in Python.
    - Demonstrate how to create and initialize class attributes using __init__.
- **Inheritance and Polymorphism**:
    - Explain base and derived classes.
    - Illustrate how methods in derived classes can override base class methods.
- **Composition**:
    - Explain how one object can contain other objects (e.g., a machine contains runs).
    - Use production_line_data.csv to map machines and their production runs.

3. Hands-On Demo: Modeling Machines and Production Runs

Scenario: A manufacturing plant tracks machine production using production_line_data.csv. Each machine has production runs with specific start and end dates and units produced.

Code:

python

```
class Machine:
    def __init__(self, machine_id, equipment_type):
        self.machine_id = machine_id
        self.equipment_type = equipment_type
        self.runs = []

    def add_run(self, run_start, run_end, units_produced):
        self.runs.append({
            "run_start": run_start,
            "run_end": run_end,
            "units_produced": units_produced
        })

    def total_units_produced(self):
        return sum(run["units_produced"] for run in self.runs)

# Example Usage
machine = Machine("MC-1001", "Speaker Assembly Machine")
machine.add_run("2024-01-01", "2024-01-07", 210)
machine.add_run("2024-01-10", "2024-01-24", 420)

print(f"Total Units Produced:
{machine.total_units_produced()}")
```

---

4. Exercise: Extending the Model

Task: Extend the Machine class to include product categories and types. Load data from production_line_data.csv and create objects for each machine.

Steps:

1. Load data from production_line_data.csv.
2. Create Machine objects for each unique Machine_ID.
3. Add production runs for each machine based on the dataset.

Code Skeleton:

python

```
import csv


machines = {}


with open("Training_Datasets/Day1/production_line_data.csv",
mode="r") as file:
    reader = csv.DictReader(file)
    for row in reader:
        machine_id = row["Machine_ID"]
        if machine_id not in machines:
            machines[machine_id] = Machine(machine_id,
row["Equipment_Type"])
        machines[machine_id].add_run(row["Run_Start"],
row["Run_End"], int(row["Units_Produced"]))


for machine_id, machine in machines.items():
    print(f"Machine {machine_id} produced
{machine.total_units_produced()} units.")
```

Afternoon Session: Advanced Functionality in Classes

---

1. Core Topics

- Static and Class Methods
- Decorators and Encapsulation
- Practical Applications in Manufacturing

---

2. Lecture Outline

- **Static and Class Methods**:
    - Explain the difference between instance methods, class methods, and static methods.
    - Use static methods to calculate aggregate metrics (e.g., total production for all machines).
- **Encapsulation**:
    - Introduce private and protected attributes to restrict direct access to sensitive data.

---

3. Hands-On Demo: Aggregating Production Metrics

Scenario: A manufacturing manager wants to calculate the total production across all machines in the plant.

Code:

python

```
class Machine:
    all_machines = []

    def __init__(self, machine_id, equipment_type):
        self.machine_id = machine_id
        self.equipment_type = equipment_type
        self.runs = []
        Machine.all_machines.append(self)
```

```
    def add_run(self, run_start, run_end, units_produced):
        self.runs.append({"run_start": run_start, "run_end":
run_end, "units_produced": units_produced})


    @staticmethod
    def total_production():
        return sum(machine.total_units_produced() for machine
in Machine.all_machines)


    def total_units_produced(self):
        return sum(run["units_produced"] for run in self.runs)


# Example Usage
machine1 = Machine("MC-1001", "Speaker Assembly Machine")
machine1.add_run("2024-01-01", "2024-01-07", 210)
machine2 = Machine("MC-1002", "Camera Assembly Machine")
machine2.add_run("2024-02-01", "2024-02-14", 140)

print(f"Total Production: {Machine.total_production()}")
```

---

4. Exercise: Applying Encapsulation

Task: Update the Machine class to restrict direct access to sensitive attributes such as machine_id. Add validation logic for production runs.

Steps:

1. Make machine_id a private attribute.
2. Add a method to retrieve machine_id securely.
3. Validate that Run_End is after Run_Start.

Code Skeleton:

python

```
class Machine:
    def __init__(self, machine_id, equipment_type):
        self.__machine_id = machine_id  # Private attribute
        self.equipment_type = equipment_type
        self.runs = []

    def get_machine_id(self):
        return self.__machine_id

    def add_run(self, run_start, run_end, units_produced):
        if run_end <= run_start:
            raise ValueError("Run_End must be after
Run_Start.")
        self.runs.append({"run_start": run_start, "run_end":
run_end, "units_produced": units_produced})
```

Deliverables

- **Code**:
    - Class-based models for machines and production runs.
    - Total production calculations using static methods.
    - Encapsulated class design with validation.
- **Hands-On Outputs**:
    - Aggregated production metrics from production_line_data.csv.

Learning Outcomes

By the end of Day 1, participants will:

1. Understand and apply OOP principles in Python.
2. Model real-world manufacturing scenarios using Python classes.
3. Implement and use static methods and encapsulation for better code organization.

Day 2: Coding Conventions and GUI Programming

Objective:

Participants will learn Python coding standards (PEP8, PEP257) and best practices for writing clean, maintainable code. In addition, they will design and implement graphical user interfaces (GUIs) using tkinter to interact with production and inventory data.

Morning Session: Coding Standards and Best Practices

---

1. Core Topics

- PEP8 Guidelines:
    - Naming conventions, whitespace, and indentation.
    - Best practices for functions, classes, and comments.
- PEP257 Guidelines:
    - Writing and formatting docstrings.
    - Using docstrings for functions, methods, and modules.

---

2. Lecture Outline

- **PEP8 Overview**:
    - Explain the importance of adhering to coding standards for maintainability.
    - Discuss common PEP8 violations and how to fix them using tools like flake8.
- **PEP257 Overview**:
    - Introduce docstrings and their formatting rules.
    - Demonstrate how to use docstrings to document a function or class effectively.

---

3. Hands-On Demo: Refactoring a Script

Scenario: Refactor a script that processes supplier_schedule.xml to make it PEP8-compliant and document it using PEP257.

Original Code:

python

```
import xml.etree.ElementTree as ET

def parsexml(filepath):
 tree=ET.parse(filepath)
 root=tree.getroot()
 for s in root.findall("Schedule"):
  print(s.find("Supplier_ID").text)
```

Refactored Code:

python

```
import xml.etree.ElementTree as ET

def parse_xml(file_path):
    """
    Parses an XML file and prints supplier IDs.

    Args:
        file_path (str): Path to the XML file.

    Returns:
        None
    """
    tree = ET.parse(file_path)
    root = tree.getroot()
    for schedule in root.findall("Schedule"):
        supplier_id = schedule.find("Supplier_ID").text
        print(f"Supplier ID: {supplier_id}")
```

Running flake8:

1. Install flake8 if not already installed:

bash

```
pip install flake8
```

2. Run flake8 on the script:

bash

```
flake8 refactored_script.py
```

3. Fix any reported issues.

---

4. Exercise: Refactor and Document

Task: Refactor the code below to make it PEP8-compliant and document it with PEP257.

python

```
def
extractsuppliers(filepath):tree=ET.parse(filepath);root=tree.g
etroot();for s in root.findall("Supplier"):print(s)
```

Solution:

python

```
def extract_suppliers(file_path):
    """
    Extracts supplier data from an XML file and prints each
supplier.

    Args:
        file_path (str): Path to the XML file.

    Returns:
        None
    """
    tree = ET.parse(file_path)
    root = tree.getroot()
```

```
for supplier in root.findall("Supplier"):
    print(supplier.text)
```

Afternoon Session: Introduction to GUI Development

---

1. Core Topics

- Basics of tkinter:
    - Windows, labels, buttons, and input fields.
- Dynamic Data Interaction:
    - Using tkinter to input and display production data.
    - Simple validations for user inputs.

---

2. Lecture Outline

- **GUI Basics**:
    - Explain the structure of a tkinter application.
    - Demonstrate how to create a window, add widgets, and handle events.
- **Data Interaction**:
    - Show how to connect GUI inputs to data processing logic.
    - Introduce listboxes for displaying production data dynamically.

---

3. Hands-On Demo: Input and View Production Data

Scenario: Build a GUI to input machine details and view production runs dynamically.

Code:

python

```
import tkinter as tk

def add_entry():
    machine_id = entry_machine_id.get()
    equipment_type = entry_equipment_type.get()
    duration = int(entry_duration.get())
    units_per_day = int(entry_units_per_day.get())
    total_units = duration * units_per_day
    listbox.insert(tk.END, f"{machine_id} | {equipment_type} | {total_units} units")

# GUI Setup
root = tk.Tk()
root.title("Production Data Input")

# Input Fields
tk.Label(root, text="Machine ID").pack()
entry_machine_id = tk.Entry(root)
entry_machine_id.pack()

tk.Label(root, text="Equipment Type").pack()
entry_equipment_type = tk.Entry(root)
entry_equipment_type.pack()

tk.Label(root, text="Run Duration (days)").pack()
entry_duration = tk.Entry(root)
entry_duration.pack()
```

```
tk.Label(root, text="Units Produced Per Day").pack()
entry_units_per_day = tk.Entry(root)
entry_units_per_day.pack()

# Add Button
tk.Button(root, text="Add Entry", command=add_entry).pack()

# Listbox to Display Entries
listbox = tk.Listbox(root, width=50)
listbox.pack()

root.mainloop()
```

---

4. Exercise: Add Data Validation and Filtering

Task:

1. Add validation to ensure all fields are filled before adding an entry.
2. Add a "Filter" button to display only machines producing more than a threshold.

Steps:

1. Modify the add_entry function to validate inputs.
2. Add a threshold input field and a "Filter" button.

Code Skeleton:

python

```
def filter_entries():
    threshold = int(entry_threshold.get())
    listbox.delete(0, tk.END)
    for machine in machines:
        if machine["total_units"] > threshold:
            listbox.insert(tk.END, f"{machine['id']} |
{machine['type']} | {machine['total_units']} units")
```

Deliverables

- **Code**:
    - Refactored and documented XML parsing scripts.
    - GUI application for inputting and viewing production data.
- **Hands-On Outputs**:
    - Validated tkinter GUI that dynamically calculates and displays total units produced.

Learning Outcomes

By the end of Day 2, participants will:

1. Adhere to Python coding standards (PEP8, PEP257) for clean, maintainable code.
2. Build functional GUIs to interact with production datasets.
3. Implement dynamic data processing and validation in GUIs.

Day 3: Network Programming and File Processing

Objective:

Participants will gain an understanding of network programming concepts and learn how to interact with REST APIs. They will also work with file processing techniques to parse and manipulate data stored in CSV, JSON, and XML formats. The exercises will use datasets such as production_line_data.csv, inventory_levels.json, and supplier_schedule.xml.

Morning Session: Network Programming

---

1. Core Topics

- Socket Programming:
    - Basics of client-server communication using Python.
- REST APIs:
    - Concepts of HTTP methods (GET, POST, PUT, DELETE).
    - Interacting with APIs to fetch and update data.

---

2. Lecture Outline

- **Socket Programming Basics**:
    - Explain how sockets enable communication between systems.
    - Demonstrate a simple client-server interaction.
- **REST API Concepts**:
    - Define REST APIs and their role in modern applications.
    - Discuss HTTP methods and their use cases.

---

3. Hands-On Demo: REST API Interaction

Scenario: Use the requests library to interact with a mock REST API to fetch and update inventory data.

Code:

python

```
import requests


# Fetch data from the API
def fetch_inventory():
    response = 
requests.get("https://jsonplaceholder.typicode.com/posts/1")
    if response.status_code == 200:
        print("Data fetched successfully:")
        print(response.json())
    else:
        print(f"Failed to fetch data. Status code: 
{response.status_code}")


# Update data using the API
def update_inventory():
    payload = {"title": "Updated Inventory", "body": "This is 
the updated inventory data.", "userId": 1}
    response = 
requests.put("https://jsonplaceholder.typicode.com/posts/1", 
json=payload)
    if response.status_code == 200:
        print("Data updated successfully:")
        print(response.json())
    else:
        print(f"Failed to update data. Status code: 
{response.status_code}")


fetch_inventory()
update_inventory()
```

4. Exercise: Fetch and Analyze Inventory

Task:

1. Use a REST API to fetch a list of items.
2. Filter items with stock levels below a threshold.
3. Print the filtered results.

Code Skeleton:

python

```
def fetch_and_filter_inventory(threshold):
    response = requests.get("https://jsonplaceholder.typicode.com/posts")
    if response.status_code == 200:
        items = response.json()
        filtered_items = [item for item in items if item["userId"] < threshold]
        print(filtered_items)
    else:
        print(f"Failed to fetch data. Status code: {response.status_code}")
```

Afternoon Session: File Processing

---

1. Core Topics

- Working with CSV, JSON, and XML files.
- Parsing, filtering, and merging datasets from multiple sources.

---

2. Lecture Outline

- **CSV File Processing**:
  - Introduce the csv module for reading and writing CSV files.
- **JSON File Processing**:
  - Explain how to parse JSON files and filter data.
- **XML File Processing**:
  - Demonstrate parsing and manipulating XML data using ElementTree.

---

3. Hands-On Demo: Parsing and Filtering Data

Scenario: Parse inventory_levels.json to filter products below the reorder threshold and save the filtered results to a new JSON file.

Code:

python

```
import json

def filter_inventory(input_file, output_file, threshold):
    with open(input_file, mode="r") as infile:
        inventory = json.load(infile)

    filtered_inventory = [item for item in inventory if
item["Stock_Level"] < threshold]

    with open(output_file, mode="w") as outfile:
        json.dump(filtered_inventory, outfile, indent=4)
```

```
# Example Usage
filter_inventory("Training_Datasets/Day3/inventory_levels.json
", "filtered_inventory.json", 300)
```

---

4. Exercise: Parse and Merge Data

Task:

1. Parse supplier schedules from supplier_schedule.xml.
2. Merge the parsed data with filtered inventory levels from filtered_inventory.json.
3. Save the merged data to a new JSON file.

Code Skeleton:

python

```
import json
import xml.etree.ElementTree as ET

def merge_data(xml_file, json_file, output_file):
    # Parse XML
    tree = ET.parse(xml_file)
    root = tree.getroot()
    suppliers = [
        {"Supplier_ID": schedule.find("Supplier_ID").text,
         "Material_Type": schedule.find("Material_Type").text,
         "Delivery_Date": schedule.find("Delivery_Date").text}
        for schedule in root.findall("Schedule")
    ]

    # Parse JSON
    with open(json_file, mode="r") as infile:
        inventory = json.load(infile)

    # Combine Data
```

```
    combined_data = {
        "Suppliers": suppliers,
        "Inventory": inventory
    }

    # Save Combined Data
    with open(output_file, mode="w") as outfile:
        json.dump(combined_data, outfile, indent=4)

# Example Usage
merge_data(
    "Training_Datasets/Day3/supplier_schedule.xml",
    "filtered_inventory.json",
    "combined_data.json"
)
```

Deliverables

- **Code**:
    - REST API interaction scripts for fetching and updating inventory data.
    - File parsing scripts for CSV, JSON, and XML data.
    - Merged data from multiple sources.
- **Hands-On Outputs**:
    - Filtered inventory JSON file.
    - Combined JSON file of inventory and supplier schedules.

Learning Outcomes

By the end of Day 3, participants will:

1. Understand network programming basics and REST API interaction.
2. Parse and filter datasets from CSV, JSON, and XML files.

3. Integrate data from multiple sources to create a cohesive dataset.

Day 4: Advanced Topics and Final Project

Objective:

Participants will learn advanced database programming and logging techniques, applying them to production and maintenance scenarios. They will culminate the training by developing a capstone project integrating database, GUI, and REST API components.

Morning Session: Database Programming and Logging

---

1. Core Topics

- SQLite Integration:
    - Creating tables and managing data in a local SQLite database.
    - Querying production and maintenance data.
- Logging:
    - Configuring logs for database and system operations.
    - Logging production anomalies and error events.

---

2. Lecture Outline

- **Database Programming**:
    - Demonstrate SQLite database creation and data manipulation.
    - Explain how to query aggregated metrics (e.g., total production by machine).
- **Logging Basics**:
    - Introduce Python's logging module for monitoring processes.
    - Discuss best practices for structured and detailed logging.

---

3. Hands-On Demo: Database Operations

Scenario: Use SQLite to store and query data from production_line_data.csv and maintenance_logs.db.

Code:

python

```
import sqlite3

# Initialize the database
def initialize_database():
    conn =
sqlite3.connect("Training_Datasets/Day4/production_data.db")
    cursor = conn.cursor()

    # Create Production Table
    cursor.execute("""
    CREATE TABLE IF NOT EXISTS Production (
        Machine_ID TEXT,
        Equipment_Type TEXT,
        Run_Start TEXT,
        Run_End TEXT,
        Units_Produced INTEGER,
        Product_Category TEXT,
        Product_Type TEXT
    )
    """)

    conn.commit()
    conn.close()

# Insert data from CSV
def insert_production_data(csv_file):
```

```
    conn =
sqlite3.connect("Training_Datasets/Day4/production_data.db")
    cursor = conn.cursor()

    with open(csv_file, mode="r") as file:
        next(file)  # Skip header
        for line in file:
            cursor.execute("""
            INSERT INTO Production VALUES (?, ?, ?, ?, ?, ?,
?)
            """, line.strip().split(","))

    conn.commit()
    conn.close()

initialize_database()
insert_production_data("Training_Datasets/Day1/production_line
_data.csv")
```

---

4. Exercise: Query and Analyze Data

Task:

1. Query the total units produced by each machine.
2. Identify machines with production anomalies (e.g., units below expected levels).

Code Skeleton:

python

```
def query_total_production(database_file):
    conn = sqlite3.connect(database_file)
    cursor = conn.cursor()

    # Query total production by machine
    cursor.execute("""
    SELECT Machine_ID, SUM(Units_Produced)
    FROM Production
    GROUP BY Machine_ID
    """)
    results = cursor.fetchall()
    conn.close()
    return results

# Example Usage
results =
query_total_production("Training_Datasets/Day4/production_data
.db")
for row in results:
    print(f"Machine {row[0]} produced {row[1]} units.")
```

---

5. Hands-On Demo: Logging

Scenario: Log database insertions and anomalies for production data.

Code:

python

```
import logging


# Configure logging
logging.basicConfig(filename="Training_Datasets/Day4/production_logs.log", level=logging.INFO)


def log_production_data(machine_id, units_produced, expected_units):
    if units_produced < expected_units:
        logging.warning(f"Low production: Machine {machine_id} produced {units_produced} units (Expected: {expected_units}).")
    else:
        logging.info(f"Production recorded: Machine {machine_id} produced {units_produced} units.")


# Example Usage
log_production_data("MC-1001", 200, 300)
```

Afternoon Session: Capstone Project

---

1. Project Objective

Design a complete system that integrates:

- A GUI for inputting and viewing production data.
- REST API for inventory updates.
- SQLite database for persistent storage.

---

2. Project Requirements

- **GUI**:
    - Allow users to input production data.
    - Display aggregated production metrics dynamically.
- **Database**:
    - Store production and maintenance logs in SQLite.
- **REST API**:
    - Fetch updated inventory levels and incorporate them into the system.

---

3. Project Implementation

Complete Code:

python

```
import tkinter as tk
import sqlite3
import requests

# Database Setup
def initialize_database():
    conn = sqlite3.connect("capstone_project.db")
    cursor = conn.cursor()

    cursor.execute("""
    CREATE TABLE IF NOT EXISTS Production (
        Machine_ID TEXT,
        Equipment_Type TEXT,
        Run_Start TEXT,
        Run_End TEXT,
        Units_Produced INTEGER
    )
    """)
    conn.commit()
    conn.close()

def insert_production(machine_id, equipment_type, run_start,
run_end, units_produced):
    conn = sqlite3.connect("capstone_project.db")
    cursor = conn.cursor()
    cursor.execute("""
    INSERT INTO Production VALUES (?, ?, ?, ?, ?)
    """, (machine_id, equipment_type, run_start, run_end,
units_produced))
```

```
    conn.commit()
    conn.close()


# REST API Integration
def fetch_inventory():
    response =
requests.get("https://jsonplaceholder.typicode.com/posts")
    if response.status_code == 200:
        inventory_label.config(text="Inventory fetched
successfully!")
    else:
        inventory_label.config(text="Failed to fetch inventory
data.")


# GUI Setup
def add_production_entry():
    machine_id = entry_machine_id.get()
    equipment_type = entry_equipment_type.get()
    run_start = entry_run_start.get()
    run_end = entry_run_end.get()
    units_produced = int(entry_units_produced.get())
    insert_production(machine_id, equipment_type, run_start,
run_end, units_produced)
    production_listbox.insert(tk.END, f"{machine_id} |
{equipment_type} | {units_produced} units")


# Main GUI
root = tk.Tk()
root.title("Capstone Project: Production Management")


# Input Fields
tk.Label(root, text="Machine ID").pack()
entry_machine_id = tk.Entry(root)
entry_machine_id.pack()
```

```
tk.Label(root, text="Equipment Type").pack()
entry_equipment_type = tk.Entry(root)
entry_equipment_type.pack()

tk.Label(root, text="Run Start").pack()
entry_run_start = tk.Entry(root)
entry_run_start.pack()

tk.Label(root, text="Run End").pack()
entry_run_end = tk.Entry(root)
entry_run_end.pack()

tk.Label(root, text="Units Produced").pack()
entry_units_produced = tk.Entry(root)
entry_units_produced.pack()

tk.Button(root, text="Add Production Entry",
command=add_production_entry).pack()

# Listbox for Production Data
production_listbox = tk.Listbox(root, width=60)
production_listbox.pack()

# Inventory Button
tk.Button(root, text="Fetch Inventory Data",
command=fetch_inventory).pack()
inventory_label = tk.Label(root, text="")
inventory_label.pack()

# Initialize Database and Start GUI
initialize_database()
```

```
root.mainloop()
```

---

4. Exercise: Extend the System

Task:

1. Add a "Filter Production Data" button to display machines producing more than a threshold.
2. Log every operation performed by the user (e.g., adding an entry, fetching inventory).

Deliverables

- **Capstone System**:
  - GUI for production and inventory management.
  - Integrated SQLite database.
  - REST API data fetching.
- **Log Files**:
  - Operations and anomalies logged to a file.

Learning Outcomes

By the end of Day 4, participants will:

1. Build and manage SQLite databases for production data.
2. Develop a fully functional system integrating GUI, database, and REST API components.
3. Implement robust logging to monitor system operations.

Data Dictionary

Dataset: production_line_data.csv

Field Name	Data Type	Constraints	Description
Machine_ID	Text	Unique, Not Null	Unique identifier for the machine.
Equipment_Type	Text	Not Null	Type of machine used for production (e.g., "Speaker Assembly Machine").
Run_Start	Date	Not Null	Start date of the production run (YYYY-MM-DD).
Run_End	Date	Not Null	End date of the production run (YYYY-MM-DD).
Units_Produced	Integer	>= 0	Total units produced during the production run.
Product_Category	Text	Not Null	Category of the product being manufactured (e.g., "Audio Devices").
Product_Type	Text	Not Null	Specific type of product manufactured (e.g., "Speakers").

Dataset: supplier_schedule.xml

Field Name	Data Type	Constraints	Description
Supplier_ID	Text	Unique, Not Null	Unique identifier for the supplier (e.g., "SPL-101").
Material_Type	Text	Not Null	Type of material supplied (e.g., "Plastic", "Metal").
Delivery_Date	Date	Not Null	Scheduled delivery date for the material (YYYY-MM-DD).

Dataset: inventory_levels.json

Field Name	Data Type	Constraints	Description
Category	Text	Not Null	Product category (e.g., "Audio Devices", "Video Devices").
Product_Type	Text	Not Null	Specific type of product in inventory (e.g., "Speakers", "Cameras").
Stock_Level	Integer	>= 0	Current stock level for the product.
Reorder_Threshold	Integer	>= 0	Stock level at which reorder is triggered.

Dataset: maintenance_logs.db

Table: MaintenanceLogs

Field Name	Data Type	Constraints	Description
Machine_ID	Text	Foreign Key, Not Null	Identifier for the machine being maintained (e.g., "MC-1001").
Maintenance_Date	Date	Not Null	Date of the maintenance activity (YYYY-MM-DD).
Issue_Description	Text	Not Null	Description of the maintenance issue (e.g., "Routine Check", "Component Replacement").

Summary of Metadata

Dataset	File Type	Primary Use
production_line_data.csv	CSV	Tracking production details such as machine runs and product types.
supplier_schedule.xml	XML	Managing supplier delivery schedules and material types.
inventory_levels.json	JSON	Monitoring inventory levels and reorder thresholds for products.
maintenance_logs.db	SQLite DB	Logging maintenance activities and issues for machines.

Additional Notes

1. **Data Relationships**:
    - Machine_ID in production_line_data.csv is related to Machine_ID in maintenance_logs.db.
    - Product_Type in inventory_levels.json relates to Product_Type in production_line_data.csv.
2. **Validation**:
    - Each dataset is validated to ensure consistency, e.g., Run_End is always after Run_Start.

Code to Generate the Dataset

```
import os
import csv
import json
import sqlite3
import xml.etree.ElementTree as ET
from faker import Faker
from datetime import datetime, timedelta, date
import random

# Initialize Faker
fake = Faker()

# Base Directories
base_dir = "Training_Datasets"
days = ["Day1", "Day2", "Day3", "Day4"]
os.makedirs(base_dir, exist_ok=True)
for day in days:
    os.makedirs(os.path.join(base_dir, day), exist_ok=True)

# Fixed Parameters
RUN_DURATIONS = [7, 14, 21]  # Run durations in days
EQUIPMENT_TYPES = {
    "Speaker Assembly Machine": "Speakers",
    "Microphone Assembly Machine": "Microphones",
    "Headphone Assembly Machine": "Headphones",
    "Amplifier Assembly Machine": "Amplifiers",
    "Camera Assembly Machine": "Cameras"
}
PRODUCT_CATEGORIES = {
    "Audio Devices": ["Speakers", "Microphones", "Headphones",
"Amplifiers"],
```

```
    "Video Devices": ["Cameras", "Projectors", "Monitors",
"Streaming Devices"],
    "Accessories": ["Cables", "Remote Controls", "Mounts",
"Batteries"]
}
MATERIAL_TYPES = ["Plastic", "Metal", "Circuit Board",
"Wiring", "Packaging"]


# Units Produced Multiplier
UNITS_PER_DAY = {
    "Speaker Assembly Machine": 30,
    "Microphone Assembly Machine": 25,
    "Headphone Assembly Machine": 20,
    "Amplifier Assembly Machine": 15,
    "Camera Assembly Machine": 10
}


# 1. Generate Production Line Data (CSV)
def generate_production_line_data():
    file_path = os.path.join(base_dir, "Day1",
"production_line_data.csv")
    with open(file_path, mode="w", newline="") as file:
        writer = csv.writer(file)
        writer.writerow(["Machine_ID", "Equipment_Type",
"Run_Start", "Run_End", "Units_Produced", "Product_Category",
"Product_Type"])


        for year in range(2020, 2025):
            start_date = date(year, 1, 1)  # Use datetime.date
object
            end_date = date(year, 12, 31)  # Use datetime.date
object
            for _ in range(200):
                machine_id = f"MC-{random.randint(1000,
9999)}"
```

```
                equipment_type, product_type =
random.choice(list(EQUIPMENT_TYPES.items()))
                product_category = next((cat for cat, types in
PRODUCT_CATEGORIES.items() if product_type in types),
"Unknown")
                run_start =
fake.date_between(start_date=start_date, end_date=end_date)
                duration = random.choice(RUN_DURATIONS)
                run_end = run_start + timedelta(days=duration)
                units_produced = duration *
UNITS_PER_DAY[equipment_type]
                writer.writerow([machine_id, equipment_type,
run_start.strftime("%Y-%m-%d"), run_end.strftime("%Y-%m-%d"),
units_produced, product_category, product_type])


# 2. Generate Supplier Schedule Data (XML)
def generate_supplier_schedule():
    file_path = os.path.join(base_dir, "Day2",
"supplier_schedule.xml")
    root = ET.Element("SupplierSchedules")


    for year in range(2020, 2025):
        start_date = date(year, 1, 1)
        end_date = date(year, 12, 31)
        for _ in range(100):
            schedule = ET.SubElement(root, "Schedule")
            ET.SubElement(schedule, "Supplier_ID").text =
f"SPL-{random.randint(100, 999)}"
            ET.SubElement(schedule, "Material_Type").text =
random.choice(MATERIAL_TYPES)
            delivery_date =
fake.date_between(start_date=start_date, end_date=end_date)
            ET.SubElement(schedule, "Delivery_Date").text =
delivery_date.strftime("%Y-%m-%d")


    tree = ET.ElementTree(root)
```

```
    tree.write(file_path, encoding="utf-8",
xml_declaration=True)


# 3. Generate Inventory Levels (JSON)
def generate_inventory_levels():
    file_path = os.path.join(base_dir, "Day3",
"inventory_levels.json")
    inventory = []


    for product_category, product_types in
PRODUCT_CATEGORIES.items():
        for product_type in product_types:
            inventory.append({
                "Category": product_category,
                "Product_Type": product_type,
                "Stock_Level": random.randint(100, 1000),
                "Reorder_Threshold": random.randint(50, 300)
            })


    with open(file_path, mode="w") as file:
        json.dump(inventory, file, indent=4)


# 4. Generate Maintenance Logs (SQLite Database)
def generate_maintenance_logs():
    file_path = os.path.join(base_dir, "Day4",
"maintenance_logs.db")
    conn = sqlite3.connect(file_path)
    cursor = conn.cursor()


    cursor.execute("""
    CREATE TABLE IF NOT EXISTS MaintenanceLogs (
        Machine_ID TEXT,
        Maintenance_Date TEXT,
```

```
        Issue_Description TEXT
    )
    """)

    machines = [f"MC-{random.randint(1000, 9999)}" for _ in
range(10)]
    for _ in range(200):
        maintenance_date =
fake.date_between(start_date=date(2020, 1, 1),
end_date=date(2024, 12, 31))
        issue_description = random.choice(["Routine Check",
"Component Replacement", "Alignment Adjustment",
"Calibration"])
        machine_id = random.choice(machines)
        cursor.execute("INSERT INTO MaintenanceLogs VALUES (?,
?, ?)", (machine_id, maintenance_date.strftime("%Y-%m-%d"),
issue_description))

    conn.commit()
    conn.close()

# Main Script Execution
if __name__ == "__main__":
    print("Generating refined datasets with proportional units
produced...")
    generate_production_line_data()
    generate_supplier_schedule()
    generate_inventory_levels()
    generate_maintenance_logs()
    print(f"Datasets have been successfully generated in the
'{base_dir}' directory.")
```

Comprehensive Step-by-Step Setup for Local Installation

This guide provides instructions to set up the environment for running the Day 1–4 demos and exercises using **Python**, **IDLE**, and **PyCharm**. It includes the creation of a virtual environment named pcpp1, installation of required packages, and configuring both IDLE and PyCharm.

1. Prerequisites

1. **Install Python**:
    - Download and install Python 3.8 or above from Python.org.
    - During installation, check the box for **Add Python to PATH**.
2. **Verify Installation**:
    - Open a terminal or command prompt and run:

bash

```
python --version
```

    - Ensure the version is 3.8 or higher.

2. Create a Virtual Environment

1. **Open Terminal or Command Prompt**:
    - Navigate to the project directory (e.g., Training_Project).
2. **Create the Virtual Environment**:
    - Run the following command:

bash

```
python -m venv pcpp1
```

    - This creates a virtual environment named pcpp1 in the project folder.
3. **Activate the Virtual Environment**:
    - On Windows:

bash

```
pcpp1\Scripts\activate
```

- On macOS/Linux:

bash

```
source pcpp1/bin/activate
```

4. **Verify Activation**:
    - The terminal should now show (pcpp1) before the prompt, indicating the environment is active.

3. Install Required Packages

1. **Update pip**:
    - Run:

bash

```
python -m pip install --upgrade pip
```

2. **Install Required Packages**:
    - Install the necessary libraries:

bash

```
pip install requests faker flake8
```

3. **Verify Installation**:
    - List installed packages:

bash

```
pip list
```

- Ensure requests, faker, and flake8 are listed.

4. Prepare the Dataset

1. **Run the Dataset Generation Script**:
    - Place the dataset generation script (e.g., generate_datasets.py) in the project directory.
    - Run the script to generate datasets for all four days:

bash

```
python generate_datasets.py
```

- Verify the generated datasets in the Training_Datasets folder.

5. Configure IDLE

1. **Open IDLE**:
    - Launch IDLE from the Start menu or by typing idle in the terminal/command prompt.
2. **Set Python Environment**:
    - In IDLE, go to **Options** > **Configure IDLE** > **General**.
    - Ensure the path points to the Python interpreter in the pcpp1 virtual environment:
        - On Windows: <path-to-project>\pcpp1\Scripts\python.exe
        - On macOS/Linux: <path-to-project>/pcpp1/bin/python
3. **Run a Script**:
    - Open any script (e.g., Day1_Script1_Demo_1.py) and press F5 to run it in IDLE.

6. Configure PyCharm

1. **Install PyCharm**:
    - Download and install PyCharm Community Edition from JetBrains.
2. **Create a New Project**:
    - Open PyCharm and select **New Project**.
    - Choose the project directory (e.g., Training_Project).
3. **Set Python Interpreter**:
    - Go to **File** > **Settings** > **Project: Training_Project** > **Python Interpreter**.
    - Click the gear icon and select **Add** > **Existing Environment**.
    - Point to the Python interpreter in the pcpp1 environment:
        - On Windows: <path-to-project>\pcpp1\Scripts\python.exe
        - On macOS/Linux: <path-to-project>/pcpp1/bin/python

4. **Install Packages in PyCharm**:
    - PyCharm automatically detects installed packages in the environment. If needed, you can add packages via **File** > **Settings** > **Python Interpreter** > **+**.
5. **Run a Script**:
    - Open any script (e.g., Day1_Script1_Demo_1.py).
    - Click the green play button to execute the script.

7. Testing and Validation

1. **Run Demos and Exercises**:
    - Open each script for Day 1–4 in IDLE or PyCharm.
    - Execute the scripts to verify they work as intended.
2. **Check Dataset Outputs**:
    - Ensure datasets are generated in the Training_Datasets folder.
3. **Run flake8 for Code Quality**:
    - In the terminal, run:

bash

```
flake8 <script_name>.py
```

    - Fix any reported issues to maintain PEP8 compliance.

8. Directory Structure

Ensure the project directory is structured as follows:

bash

```
Training_Project/
├── pcpp1/              # Virtual environment
├── Training_Datasets/      # Generated datasets
│   ├── Day1/
│   ├── Day2/
│   ├── Day3/
│   ├── Day4/
├── generate_datasets.py    # Dataset generation script
├── Day1_Script1_Demo_1.py  # Day 1 scripts
├── Day2_Script1_Demo_1.py  # Day 2 scripts
├── Day3_Script1_Demo_1.py  # Day 3 scripts
├── Day4_Script1_Demo_1.py  # Day 4 scripts
```

9. Troubleshooting

1. **Virtual Environment Not Found**:
    - Ensure you activated the pcpp1 environment before running scripts.
2. **Missing Packages**:
    - Reinstall missing packages with:

bash

```
pip install <package-name>
```

3. **Dataset Not Generated**:
    - Check for errors in the dataset generation script and re-run it:

bash

```
python generate_datasets.py
```

This setup ensures a consistent environment for running all demos and exercises locally using IDLE or PyCharm.

Day 1: Consolidated Demo and Exercise Solutions

Below are the consolidated demo scripts and exercise solutions for Day 1.

Demo 1: Modeling Machines and Production Runs

File Name: Day1_Script1_Demo_1.py

Description: Demonstrates a Machine class that tracks production runs, calculates total units produced, and models relationships between machines and production data.

Code:

python

```
class Machine:
    def __init__(self, machine_id, equipment_type):
        self.machine_id = machine_id
        self.equipment_type = equipment_type
        self.runs = []

    def add_run(self, run_start, run_end, units_produced):
        self.runs.append({
            "run_start": run_start,
            "run_end": run_end,
            "units_produced": units_produced
        })

    def total_units_produced(self):
        return sum(run["units_produced"] for run in self.runs)

# Example Usage
if __name__ == "__main__":
    machine = Machine("MC-1001", "Speaker Assembly Machine")
    machine.add_run("2024-01-01", "2024-01-07", 210)
    machine.add_run("2024-01-10", "2024-01-24", 420)
```

```
    print(f"Total Units Produced:
{machine.total_units_produced()}")
```

Exercise 1: Extending the Machine Class

File Name: Day1_Script2_Exercise_1.py

Description: Participants will extend the Machine class to include product categories and types, load data from production_line_data.csv, and calculate total production.

Code:

python

```
import csv

class Machine:
    def __init__(self, machine_id, equipment_type):
        self.machine_id = machine_id
        self.equipment_type = equipment_type
        self.runs = []

    def add_run(self, run_start, run_end, units_produced):
        self.runs.append({
            "run_start": run_start,
            "run_end": run_end,
            "units_produced": units_produced
        })

    def total_units_produced(self):
        return sum(run["units_produced"] for run in self.runs)

if __name__ == "__main__":
    machines = {}
```

```
    # Load data from CSV
    with
open("Training_Datasets/Day1/production_line_data.csv",
mode="r") as file:
        reader = csv.DictReader(file)
        for row in reader:
            machine_id = row["Machine_ID"]
            if machine_id not in machines:
                machines[machine_id] = Machine(machine_id,
row["Equipment_Type"])
            machines[machine_id].add_run(row["Run_Start"],
row["Run_End"], int(row["Units_Produced"]))


    # Display total production
    for machine_id, machine in machines.items():
        print(f"Machine {machine_id} produced
{machine.total_units_produced()} units.")
```

Demo 2: Using Static Methods

File Name: Day1_Script3_Demo_2.py

Description: Introduces a static method to calculate total production for all machines.

Code:

python

```
class Machine:
    all_machines = []


    def __init__(self, machine_id, equipment_type):
        self.machine_id = machine_id
        self.equipment_type = equipment_type
        self.runs = []
        Machine.all_machines.append(self)


    def add_run(self, run_start, run_end, units_produced):
```

```
        self.runs.append({
            "run_start": run_start,
            "run_end": run_end,
            "units_produced": units_produced
        })

    def total_units_produced(self):
        return sum(run["units_produced"] for run in self.runs)

    @staticmethod
    def total_production():
        return sum(machine.total_units_produced() for machine
in Machine.all_machines)

# Example Usage
if __name__ == "__main__":
    machine1 = Machine("MC-1001", "Speaker Assembly Machine")
    machine1.add_run("2024-01-01", "2024-01-07", 210)

    machine2 = Machine("MC-1002", "Camera Assembly Machine")
    machine2.add_run("2024-02-01", "2024-02-14", 140)

    print(f"Total Production: {Machine.total_production()}")
```

Exercise 2: Adding Encapsulation

File Name: Day1_Script4_Exercise_2.py

Description: Participants will encapsulate sensitive attributes like machine_id and add validation logic for production runs.

Code:

python

```
class Machine:
    def __init__(self, machine_id, equipment_type):
        self.__machine_id = machine_id  # Private attribute
        self.equipment_type = equipment_type
        self.runs = []

    def get_machine_id(self):
        return self.__machine_id

    def add_run(self, run_start, run_end, units_produced):
        if run_end <= run_start:
            raise ValueError("Run_End must be after
Run_Start.")
        self.runs.append({"run_start": run_start, "run_end":
run_end, "units_produced": units_produced})

    def total_units_produced(self):
        return sum(run["units_produced"] for run in self.runs)

# Example Usage
if __name__ == "__main__":
    machine = Machine("MC-1001", "Speaker Assembly Machine")
    machine.add_run("2024-01-01", "2024-01-07", 210)
    print(f"Machine ID: {machine.get_machine_id()}")
    print(f"Total Units Produced:
{machine.total_units_produced()}")
```

Demo 3: Validating Input Data

File Name: Day1_Script5_Demo_3.py

Description: Demonstrates input validation for production runs, ensuring consistent data quality.

Code:

python

```
class Machine:
    def __init__(self, machine_id, equipment_type):
        self.__machine_id = machine_id  # Private attribute
        self.equipment_type = equipment_type
        self.runs = []

    def add_run(self, run_start, run_end, units_produced):
        if run_end <= run_start:
            raise ValueError("Run_End must be after
Run_Start.")
        if units_produced <= 0:
            raise ValueError("Units_Produced must be
positive.")
        self.runs.append({"run_start": run_start, "run_end":
run_end, "units_produced": units_produced})

    def total_units_produced(self):
        return sum(run["units_produced"] for run in self.runs)

# Example Usage
if __name__ == "__main__":
    machine = Machine("MC-1001", "Speaker Assembly Machine")
    try:
        machine.add_run("2024-01-01", "2024-01-07", 210)
        machine.add_run("2024-01-10", "2024-01-05", 300)  #
Invalid run
    except ValueError as e:
```

```
        print(f"Error: {e}")
    print(f"Total Units Produced:
{machine.total_units_produced()}")
```

Summary of Scripts for Day 1

File Name	Purpose
Day1_Script1_Demo_1.py	Models machines and production runs.
Day1_Script2_Exercise_1.py	Extends the Machine class to process CSV data.
Day1_Script3_Demo_2.py	Adds static methods for total production calculation.
Day1_Script4_Exercise_2.py	Implements encapsulation and validation.
Day1_Script5_Demo_3.py	Validates input data for consistent quality.

Day 2: Consolidated Demo and Exercise Solutions

Below are the consolidated demo scripts and exercise solutions for Day 2,

Demo 1: Refactoring Code with PEP8 Standards

File Name: Day2_Script1_Demo_1.py

Description: Refactors a script for parsing supplier_schedule.xml to adhere to PEP8 standards and adds PEP257-compliant docstrings.

Code:

python

```
import xml.etree.ElementTree as ET


def parse_supplier_schedule(file_path):
    """
    Parses the supplier schedule XML file and prints each
supplier ID.

    Args:
        file_path (str): Path to the XML file.

    Returns:
        None
    """
    tree = ET.parse(file_path)
    root = tree.getroot()
    for schedule in root.findall("Schedule"):
        supplier_id = schedule.find("Supplier_ID").text
        print(f"Supplier ID: {supplier_id}")

# Example Usage
if __name__ == "__main__":
```

```
parse_supplier_schedule("Training_Datasets/Day2/supplier_sched
ule.xml")
```

Exercise 1: Refactor Code for Compliance

File Name: Day2_Script2_Exercise_1.py

Description: Participants refactor the provided script to adhere to PEP8 standards and add PEP257 docstrings.

Original Code:

python

```
def
extract_suppliers(filepath):tree=ET.parse(filepath);root=tree.
getroot();for s in root.findall("Supplier"):print(s)
```

Refactored Solution:

python

```
import xml.etree.ElementTree as ET


def extract_suppliers(file_path):
    """
    Extracts supplier data from an XML file and prints each
supplier.


    Args:
        file_path (str): Path to the XML file.


    Returns:
        None
    """
    tree = ET.parse(file_path)
    root = tree.getroot()
    for supplier in root.findall("Supplier"):
        print(supplier.text)
```

```
# Example Usage
if __name__ == "__main__":

extract_suppliers("Training_Datasets/Day2/supplier_schedule.xml")
```

Demo 2: Basic tkinter GUI for Data Input

File Name: Day2_Script3_Demo_2.py

Description: Demonstrates how to create a simple tkinter GUI for inputting production data.

Code:

python

```
import tkinter as tk


def add_entry():
    machine_id = entry_machine_id.get()
    equipment_type = entry_equipment_type.get()
    run_duration = int(entry_duration.get())
    units_per_day = int(entry_units_per_day.get())
    total_units = run_duration * units_per_day
    listbox.insert(tk.END, f"{machine_id} | {equipment_type} | {total_units} units")


# GUI Setup
root = tk.Tk()
root.title("Production Data Input")


# Input Fields
tk.Label(root, text="Machine ID").pack()
entry_machine_id = tk.Entry(root)
entry_machine_id.pack()
```

```
tk.Label(root, text="Equipment Type").pack()
entry_equipment_type = tk.Entry(root)
entry_equipment_type.pack()

tk.Label(root, text="Run Duration (days)").pack()
entry_duration = tk.Entry(root)
entry_duration.pack()

tk.Label(root, text="Units Produced Per Day").pack()
entry_units_per_day = tk.Entry(root)
entry_units_per_day.pack()

# Add Button
tk.Button(root, text="Add Entry", command=add_entry).pack()

# Listbox to Display Entries
listbox = tk.Listbox(root, width=50)
listbox.pack()

root.mainloop()
```

Exercise 2: Adding Validation and Filters

File Name: Day2_Script4_Exercise_2.py

Description: Participants enhance the tkinter GUI by adding input validation and a filter for production entries.

Task:

1. Validate all fields to ensure no entry is empty.
2. Add a "Filter" button to display entries with production above a certain threshold.

Code:

```
python
import tkinter as tk


entries = []  # Store all entries


def add_entry():
    machine_id = entry_machine_id.get()
    equipment_type = entry_equipment_type.get()
    run_duration = int(entry_duration.get())
    units_per_day = int(entry_units_per_day.get())


    if not machine_id or not equipment_type or not
run_duration or not units_per_day:
        error_label.config(text="All fields are required!")
        return


    total_units = run_duration * units_per_day
    entry = {"machine_id": machine_id, "equipment_type":
equipment_type, "total_units": total_units}
    entries.append(entry)


    listbox.insert(tk.END, f"{machine_id} | {equipment_type} |
{total_units} units")
    error_label.config(text="")
```

```
def filter_entries():
    threshold = int(entry_threshold.get())
    listbox.delete(0, tk.END)
    for entry in entries:
        if entry["total_units"] > threshold:
            listbox.insert(tk.END, f"{entry['machine_id']} |
{entry['equipment_type']} | {entry['total_units']} units")

# GUI Setup
root = tk.Tk()
root.title("Production Data Input")

# Input Fields
tk.Label(root, text="Machine ID").pack()
entry_machine_id = tk.Entry(root)
entry_machine_id.pack()

tk.Label(root, text="Equipment Type").pack()
entry_equipment_type = tk.Entry(root)
entry_equipment_type.pack()

tk.Label(root, text="Run Duration (days)").pack()
entry_duration = tk.Entry(root)
entry_duration.pack()

tk.Label(root, text="Units Produced Per Day").pack()
entry_units_per_day = tk.Entry(root)
entry_units_per_day.pack()

# Add Entry Button
tk.Button(root, text="Add Entry", command=add_entry).pack()
```

```
# Error Label
error_label = tk.Label(root, text="", fg="red")
error_label.pack()

# Listbox for Entries
listbox = tk.Listbox(root, width=50)
listbox.pack()

# Threshold Filter
tk.Label(root, text="Filter by Total Units Produced
(Threshold)").pack()
entry_threshold = tk.Entry(root)
entry_threshold.pack()
tk.Button(root, text="Filter", command=filter_entries).pack()

root.mainloop()
```

Demo 3: Fetching and Viewing Production Data

File Name: Day2_Script5_Demo_3.py

Description: Demonstrates a tkinter GUI that fetches and displays production data from a CSV file.

Code:

python

```
import tkinter as tk
import csv


def load_data():
    with
open("Training_Datasets/Day1/production_line_data.csv",
mode="r") as file:
        reader = csv.DictReader(file)
        listbox.delete(0, tk.END)
        for row in reader:
            listbox.insert(tk.END, f"{row['Machine_ID']} |
{row['Equipment_Type']} | {row['Units_Produced']} units")


# GUI Setup
root = tk.Tk()
root.title("Production Data Viewer")

# Load Button
tk.Button(root, text="Load Data", command=load_data).pack()

# Listbox for Data
listbox = tk.Listbox(root, width=60)
listbox.pack()

root.mainloop()
```

Summary of Scripts for Day 2

File Name	Purpose
Day2_Script1_Demo_1.py	Refactors XML parsing script with PEP8/PEP257 compliance.
Day2_Script2_Exercise_1.py	Refactoring exercise for XML parsing script.
Day2_Script3_Demo_2.py	Demonstrates basic tkinter GUI for data input.
Day2_Script4_Exercise_2.py	Enhances GUI with validation and filtering functionality.
Day2_Script5_Demo_3.py	Fetches and displays production data from a CSV file.

Day 3: Consolidated Demo and Exercise Solutions

Below are the consolidated demo scripts and exercise solutions for Day 3,

Demo 1: Basic REST API Interaction

File Name: Day3_Script1_Demo_1.py

Description: Demonstrates fetching and updating inventory data using a mock REST API with the requests library.

Code:

python

```
import requests


# Fetch data from API
def fetch_inventory():
    response = 
requests.get("https://jsonplaceholder.typicode.com/posts/1")
    if response.status_code == 200:
        print("Data fetched successfully:")
        print(response.json())
    else:
        print(f"Failed to fetch data. Status code: 
{response.status_code}")


# Update data using API
def update_inventory():
    payload = {"title": "Updated Inventory", "body": "Updated 
data for inventory.", "userId": 1}
    response = 
requests.put("https://jsonplaceholder.typicode.com/posts/1", 
json=payload)
    if response.status_code == 200:
        print("Data updated successfully:")
        print(response.json())
```

```
    else:
        print(f"Failed to update data. Status code:
{response.status_code}")


# Example Usage
if __name__ == "__main__":
    fetch_inventory()
    update_inventory()
```

Exercise 1: Fetch and Filter Data from REST API

File Name: Day3_Script2_Exercise_1.py

Description: Participants fetch data from a REST API, filter items based on a threshold, and display the results.

Code:

python

```
import requests


def fetch_and_filter_inventory(threshold):
    response =
requests.get("https://jsonplaceholder.typicode.com/posts")
    if response.status_code == 200:
        items = response.json()
        filtered_items = [item for item in items if
item["userId"] < threshold]
        print("Filtered Items:")
        for item in filtered_items:
            print(item)
    else:
        print(f"Failed to fetch data. Status code:
{response.status_code}")


# Example Usage
if __name__ == "__main__":
    fetch_and_filter_inventory(3)
```

Demo 2: Parsing and Filtering JSON Data

File Name: Day3_Script3_Demo_2.py

Description: Demonstrates how to parse inventory_levels.json and filter products below the reorder threshold.

Code:

python

```
import json


def filter_inventory(input_file, output_file, threshold):
    with open(input_file, mode="r") as infile:
        inventory = json.load(infile)


    filtered_inventory = [item for item in inventory if
item["Stock_Level"] < threshold]


    with open(output_file, mode="w") as outfile:
        json.dump(filtered_inventory, outfile, indent=4)


# Example Usage
if __name__ == "__main__":

filter_inventory("Training_Datasets/Day3/inventory_levels.json
", "filtered_inventory.json", 300)
    print("Filtered inventory saved to
filtered_inventory.json.")
```

Exercise 2: Parse and Merge Data

File Name: Day3_Script4_Exercise_2.py

Description: Participants parse supplier schedules from supplier_schedule.xml and merge them with filtered inventory data from filtered_inventory.json.

Code:

python

```
import json
import xml.etree.ElementTree as ET


def merge_data(xml_file, json_file, output_file):
  # Parse XML
  tree = ET.parse(xml_file)
  root = tree.getroot()
  suppliers = [
    {"Supplier_ID": schedule.find("Supplier_ID").text,
     "Material_Type": schedule.find("Material_Type").text,
     "Delivery_Date": schedule.find("Delivery_Date").text}
    for schedule in root.findall("Schedule")
  ]

    # Parse JSON
    with open(json_file, mode="r") as infile:
        inventory = json.load(infile)

    # Combine Data
    combined_data = {
        "Suppliers": suppliers,
        "Inventory": inventory
    }
```

```
    # Save Combined Data
    with open(output_file, mode="w") as outfile:
        json.dump(combined_data, outfile, indent=4)

# Example Usage
if __name__ == "__main__":
    merge_data(
        "Training_Datasets/Day3/supplier_schedule.xml",
        "filtered_inventory.json",
        "combined_data.json"
    )
    print("Combined data saved to combined_data.json.")
```

Demo 3: Querying CSV Data

File Name: Day3_Script5_Demo_3.py

Description: Demonstrates querying production_line_data.csv to filter machines with high production volumes.

Code:

python

```
import csv


def filter_high_production_machines(input_file, output_file,
threshold):
    with open(input_file, mode="r") as infile,
open(output_file, mode="w", newline="") as outfile:
        reader = csv.DictReader(infile)
        writer = csv.DictWriter(outfile,
fieldnames=reader.fieldnames)
        writer.writeheader()
        for row in reader:
            if int(row["Units_Produced"]) > threshold:
```

```
                writer.writerow(row)

# Example Usage
if __name__ == "__main__":
    filter_high_production_machines(
        "Training_Datasets/Day1/production_line_data.csv",
        "high_production_machines.csv",
        300
    )
    print("Filtered machines saved to
high_production_machines.csv.")
```

Summary of Scripts for Day 3

File Name	Purpose
Day3_Script1_Demo_1.py	Demonstrates REST API interaction for fetching and updating data.
Day3_Script2_Exercise_1.py	Fetches and filters data from a REST API.
Day3_Script3_Demo_2.py	Parses and filters JSON inventory data.
Day3_Script4_Exercise_2.py	Merges supplier data from XML with inventory data from JSON.
Day3_Script5_Demo_3.py	Filters high-production machines from a CSV file.

Day 4: Consolidated Demo and Exercise Solutions

Below are the consolidated demo scripts and exercise solutions for Day 4.

Demo 1: SQLite Database Initialization

File Name: Day4_Script1_Demo_1.py

Description: Demonstrates how to initialize an SQLite database and create tables for production data.

Code:

python

```
import sqlite3


def initialize_database():
    conn =
sqlite3.connect("Training_Datasets/Day4/production_data.db")
    cursor = conn.cursor()

    # Create Production Table
    cursor.execute("""
    CREATE TABLE IF NOT EXISTS Production (
        Machine_ID TEXT,
        Equipment_Type TEXT,
        Run_Start TEXT,
        Run_End TEXT,
        Units_Produced INTEGER,
        Product_Category TEXT,
        Product_Type TEXT
    )
    """)
    conn.commit()
    conn.close()
    print("Database initialized and table created.")
```

```
# Example Usage
if __name__ == "__main__":
    initialize_database()
```

Exercise 1: Inserting Data into SQLite

File Name: Day4_Script2_Exercise_1.py

Description: Participants will load data from production_line_data.csv and insert it into the Production table in SQLite.

Code:

python

```
import sqlite3
import csv

def insert_production_data(csv_file, db_file):
    conn = sqlite3.connect(db_file)
    cursor = conn.cursor()

    with open(csv_file, mode="r") as file:
        reader = csv.DictReader(file)
        for row in reader:
            cursor.execute("""
            INSERT INTO Production (Machine_ID,
Equipment_Type, Run_Start, Run_End, Units_Produced,
Product_Category, Product_Type)
            VALUES (?, ?, ?, ?, ?, ?, ?)
            """, (
                row["Machine_ID"], row["Equipment_Type"],
row["Run_Start"], row["Run_End"],
                int(row["Units_Produced"]),
row["Product_Category"], row["Product_Type"]
            ))
```

```
    conn.commit()
    conn.close()
    print("Data inserted successfully.")

# Example Usage
if __name__ == "__main__":

insert_production_data("Training_Datasets/Day1/production_line
_data.csv", "Training_Datasets/Day4/production_data.db")
```

Demo 2: Querying SQLite Database

File Name: Day4_Script3_Demo_2.py

Description: Demonstrates querying the SQLite database for total units produced by each machine.

Code:

python

```
import sqlite3

def query_total_production(db_file):
    conn = sqlite3.connect(db_file)
    cursor = conn.cursor()

    cursor.execute("""
    SELECT Machine_ID, SUM(Units_Produced) AS Total_Units
    FROM Production
    GROUP BY Machine_ID
    ORDER BY Total_Units DESC
    """)
    results = cursor.fetchall()
    conn.close()
```

```
    print("Total Production by Machine:")
    for row in results:
        print(f"Machine {row[0]}: {row[1]} units")

# Example Usage
if __name__ == "__main__":

query_total_production("Training_Datasets/Day4/production_data
.db")
```

Exercise 2: Logging Anomalies in Production

File Name: Day4_Script4_Exercise_2.py

Description: Participants log production anomalies such as units produced below a threshold using Python's logging module.

Code:

python

```
import sqlite3
import logging

# Configure logging
logging.basicConfig(filename="Training_Datasets/Day4/productio
n_logs.log", level=logging.INFO)

def log_production_anomalies(db_file, threshold):
    conn = sqlite3.connect(db_file)
    cursor = conn.cursor()

    cursor.execute("""
    SELECT Machine_ID, SUM(Units_Produced) AS Total_Units
    FROM Production
    GROUP BY Machine_ID
    """)
```

```
    results = cursor.fetchall()
    conn.close()

    for row in results:
        if row[1] < threshold:
            logging.warning(f"Anomaly detected: Machine
{row[0]} produced {row[1]} units (below threshold of
{threshold}).")
        else:
            logging.info(f"Machine {row[0]} production is
normal: {row[1]} units.")

# Example Usage
if __name__ == "__main__":

log_production_anomalies("Training_Datasets/Day4/production_da
ta.db", 500)
    print("Anomalies logged in production_logs.log.")
```

Demo 3: Capstone Project System

File Name: Day4_Script5_Demo_3.py

Description: Demonstrates a complete system with a tkinter GUI for data input, SQLite database for storage, and logging for monitoring.

Code:

python

```
import tkinter as tk
import sqlite3
import logging

# Configure logging
logging.basicConfig(filename="Training_Datasets/Day4/capstone_logs.log", level=logging.INFO)

# Initialize Database
def initialize_database():
    conn = sqlite3.connect("capstone_project.db")
    cursor = conn.cursor()

    cursor.execute("""
    CREATE TABLE IF NOT EXISTS Production (
        Machine_ID TEXT,
        Equipment_Type TEXT,
        Run_Start TEXT,
        Run_End TEXT,
        Units_Produced INTEGER
    )
    """)
    conn.commit()
    conn.close()
```

```
def insert_production(machine_id, equipment_type, run_start,
run_end, units_produced):
    conn = sqlite3.connect("capstone_project.db")
    cursor = conn.cursor()

    cursor.execute("""
    INSERT INTO Production (Machine_ID, Equipment_Type,
Run_Start, Run_End, Units_Produced)
    VALUES (?, ?, ?, ?, ?)
    """, (machine_id, equipment_type, run_start, run_end,
units_produced))

    conn.commit()
    conn.close()
    logging.info(f"Inserted production data: {machine_id},
{units_produced} units.")

def add_production_entry():
    machine_id = entry_machine_id.get()
    equipment_type = entry_equipment_type.get()
    run_start = entry_run_start.get()
    run_end = entry_run_end.get()
    units_produced = int(entry_units_produced.get())

    insert_production(machine_id, equipment_type, run_start,
run_end, units_produced)
    listbox.insert(tk.END, f"{machine_id} | {equipment_type} |
{units_produced} units")

# GUI Setup
root = tk.Tk()
root.title("Capstone Project: Production Management")

# Input Fields
```

```
tk.Label(root, text="Machine ID").pack()
entry_machine_id = tk.Entry(root)
entry_machine_id.pack()

tk.Label(root, text="Equipment Type").pack()
entry_equipment_type = tk.Entry(root)
entry_equipment_type.pack()

tk.Label(root, text="Run Start").pack()
entry_run_start = tk.Entry(root)
entry_run_start.pack()

tk.Label(root, text="Run End").pack()
entry_run_end = tk.Entry(root)
entry_run_end.pack()

tk.Label(root, text="Units Produced").pack()
entry_units_produced = tk.Entry(root)
entry_units_produced.pack()

tk.Button(root, text="Add Entry",
command=add_production_entry).pack()

# Listbox to Display Entries
listbox = tk.Listbox(root, width=60)
listbox.pack()

# Initialize Database and Start GUI
initialize_database()
root.mainloop()
```

Summary of Scripts for Day 4

File Name	Purpose
Day4_Script1_Demo_1.py	Initializes SQLite database and creates production table.
Day4_Script2_Exercise_1.py	Inserts production data from CSV into SQLite database.
Day4_Script3_Demo_2.py	Queries SQLite database for total units produced by each machine.
Day4_Script4_Exercise_2.py	Logs production anomalies based on threshold.
Day4_Script5_Demo_3.py	Capstone project system with GUI, database, and logging.