LAB EVALUATION 2

**PRAVEER RAJ**

**230957002**

**ROLL N0: 1**

***WEEK 5:***

***Exercise 1:***

**Student Grades Management System:**

**Create a simple student grades management system which perform the following functions**

**(Use a dictionary where the keys are student names, and the values are lists of grades.):**

**• Add a student: Add a student's name and their grades for multiple subjects.**

**• Update a Grade: Update a specific grade for a student in each subject.**

**• Remove a student: Remove a student from the system.**

**• Get Average Grade: Calculate and return the average grade for a student across all**

**subjects.**

**• Get Subject Average: Calculate and return the average grade for a specific subject.**

**• List All Students: List all students with their average grades for each subject and**

**overall.**

**• Get Highest Grade: Find the highest grade in a specific subject**

class StudentGradesSystem:

def \_\_init\_\_(self):

self.students = {}

def add\_student(self, name, grades):

self.students[name] = grades

print(f"Student {name} added with grades: {grades}")

def update\_grade(self, name, subject\_index, new\_grade):

if name in self.students:

if 0 <= subject\_index < len(self.students[name]):

self.students[name][subject\_index] = new\_grade

print(f"Grade updated for {name} in subject {subject\_index + 1}: {new\_grade}")

else:

print(f"Subject index {subject\_index} is invalid.")

else:

print(f"Student {name} not found.")

def remove\_student(self, name):

if name in self.students:

del self.students[name]

print(f"Student {name} removed.")

else:

print(f"Student {name} not found.")

def get\_average\_grade(self, name):

if name in self.students:

grades = self.students[name]

avg = sum(grades) / len(grades) if grades else 0

return f"Average grade for {name}: {avg:.2f}"

else:

return f"Student {name} not found."

def get\_subject\_average(self, subject\_index):

total, count = 0, 0

for grades in self.students.values():

if 0 <= subject\_index < len(grades):

total += grades[subject\_index]

count += 1

if count > 0:

return f"Average grade for subject {subject\_index + 1}: {total / count:.2f}"

else:

return f"No data available for subject {subject\_index + 1}."

def list\_all\_students(self):

if self.students:

for name, grades in self.students.items():

overall\_avg = sum(grades) / len(grades) if grades else 0

print(f"{name}: Grades: {grades} | Overall Average: {overall\_avg:.2f}")

else:

print("No students available.")

def get\_highest\_grade(self, subject\_index):

highest\_grade = -1

highest\_student = None

for name, grades in self.students.items():

if 0 <= subject\_index < len(grades):

if grades[subject\_index] > highest\_grade:

highest\_grade = grades[subject\_index]

highest\_student = name

if highest\_student:

return f"Highest grade in subject {subject\_index + 1}: {highest\_grade} by {highest\_student}"

else:

return f"No data available for subject {subject\_index + 1}."

system = StudentGradesSystem()

system.add\_student("Alice", [85, 90, 78])

system.add\_student("Bob", [88, 76, 92])

system.add\_student("Charlie", [95, 85, 80])

system.update\_grade("Alice", 1, 95)

system.remove\_student("Bob")

print(system.get\_average\_grade("Alice"))

print(system.get\_subject\_average(1))

system.list\_all\_students()

print(system.get\_highest\_grade(2))

**OUTPUT:**

Student Alice added with grades: [85, 90, 78]

Student Bob added with grades: [88, 76, 92]

Student Charlie added with grades: [95, 85, 80]

Grade updated for Alice in subject 2: 95

Student Bob removed.

Average grade for Alice: 86.00

Average grade for subject 2: 90.00

Alice: Grades: [85, 95, 78] | Overall Average: 86.00

Charlie: Grades: [95, 85, 80] | Overall Average: 86.67

Highest grade in subject 3: 80 by Charlie

**Q2)Exercise 2: Employee Management System**

**Implement Employee Management System using nested dictionaries and lists and**

**implement following functions to handle different operations.**

**• add\_employee(): Adds a new employee or updates an existing employee's details.**

**• update\_salary(): Updates the salary of an existing employee.**

**• add\_performance\_score(): Adds a performance score to an employee's record.**

**• remove\_employee(): Removes an employee from the records.**

**• get\_average\_salary\_by\_department(): Computes the average salary of employees**

**in a specified department.**

**• get\_employee\_with\_highest\_performance(): Finds the employee with the highest**

**average performance score.**

**• list\_employees\_by\_department(): Lists all employees in a specified department.**

class EmployeeManagementSystem:

def \_\_init\_\_(self):

self.employees = {}

def add\_employee(self, employee\_id, name, department, salary, performance\_scores=None):

if performance\_scores is None:

performance\_scores = []

self.employees[employee\_id] = {

"name": name,

"department": department,

"salary": salary,

"performance\_scores": performance\_scores

}

print(f"Employee {name} added/updated.")

def update\_salary(self, employee\_id, new\_salary):

if employee\_id in self.employees:

self.employees[employee\_id]['salary'] = new\_salary

print(f"Salary updated for {self.employees[employee\_id]['name']}: {new\_salary}")

else:

print(f"Employee {employee\_id} not found.")

def add\_performance\_score(self, employee\_id, score):

if employee\_id in self.employees:

self.employees[employee\_id]['performance\_scores'].append(score)

print(f"Performance score {score} added for {self.employees[employee\_id]['name']}.")

else:

print(f"Employee {employee\_id} not found.")

def remove\_employee(self, employee\_id):

if employee\_id in self.employees:

print(f"Employee {self.employees[employee\_id]['name']} removed.")

del self.employees[employee\_id]

else:

print(f"Employee {employee\_id} not found.")

def get\_average\_salary\_by\_department(self, department):

total\_salary, count = 0, 0

for employee in self.employees.values():

if employee['department'] == department:

total\_salary += employee['salary']

count += 1

if count > 0:

return f"Average salary in {department}: {total\_salary / count:.2f}"

else:

return f"No employees found in {department}."

def get\_employee\_with\_highest\_performance(self):

highest\_avg = -1

top\_employee = None

for employee\_id, employee in self.employees.items():

scores = employee['performance\_scores']

if scores:

avg\_score = sum(scores) / len(scores)

if avg\_score > highest\_avg:

highest\_avg = avg\_score

top\_employee = employee

if top\_employee:

return f"Employee with highest performance: {top\_employee['name']} with average score {highest\_avg:.2f}"

else:

return "No performance scores available."

def list\_employees\_by\_department(self, department):

employees\_in\_dept = [emp['name'] for emp in self.employees.values() if emp['department'] == department]

if employees\_in\_dept:

print(f"Employees in {department}: {', '.join(employees\_in\_dept)}")

else:

print(f"No employees found in {department}.")

system = EmployeeManagementSystem()

system.add\_employee(1, "Alice", "HR", 50000, [4.5, 4.7])

system.add\_employee(2, "Bob", "IT", 60000, [4.0, 3.8])

system.add\_employee(3, "Charlie", "HR", 55000, [4.8, 4.9])

system.update\_salary(2, 65000)

system.add\_performance\_score(1, 4.6)

system.remove\_employee(3)

print(system.get\_average\_salary\_by\_department("HR"))

print(system.get\_employee\_with\_highest\_performance())

system.list\_employees\_by\_department("IT")

**OUTPUT:**

Student Alice added with grades: [85, 90, 78]

Student Bob added with grades: [88, 76, 92]

Student Charlie added with grades: [95, 85, 80]

Grade updated for Alice in subject 2: 95

Student Bob removed.

Average grade for Alice: 86.00

Average grade for subject 2: 90.00

Alice: Grades: [85, 95, 78] | Overall Average: 86.00

Charlie: Grades: [95, 85, 80] | Overall Average: 86.67

Highest grade in subject 3: 80 by Charlie

***WEEK 6:***

***Exercise 1:***

**You are tasked with designing a Library Management System for a local library. The**

**library has both EBooks (digital format) and Printed Books (physical copies), and the**

**system should allow members (both students and teachers) to borrow books. The library**

**also has a librarian who manages the addition and removal of books.**

**Your system should include the following features:**

**1. Book Management:**

**o Books can be either EBooks or Printed Books.**

**o Each book should have a title, author, and ISBN.**

**o EBooks should have a file format, while Printed Books should have a page**

**count.**

**2. Member Management:**

**o The library has members who can either be students or teachers.**

**o Each member has a name and member ID.**

**o Members should be able to borrow books.**

**3. Librarian Management:**

**o A librarian can add or remove books from the library.**

**o A librarian can be both a student and a teacher.**

**4. Library Operations:**

**o The system should allow the librarian to:**

**▪ Add new books to the library.**

**▪ Remove books from the library using their ISBN.**

**▪ Search for books by title or author.**

**Requirements:**

**Using Python and Object-Oriented Programming principles, implement the following:**

**1. Create a class hierarchy to represent books (including EBooks and Printed Books).**

**2. Create a class hierarchy to represent members (students and teachers).**

**3. Implement the functionalities for adding, removing, and searching for books.**

**4. Demonstrate the following types of inheritance:**

**o Single Inheritance for books.**

**o Multiple Inheritance for the librarian, who is both a student and a teacher.**

**o Hierarchical Inheritance for members (students and teachers).**

**Tasks:**

**1. Book Management:**

**o Define a base class Book with attributes for title, author, and ISBN.**

**o Define a subclass EBook that adds the attribute for file format.**

**o Define another subclass PrintedBook that adds the attribute for page count.**

**2. Member and Librarian Management:**

**o Define a base class Member with attributes for name and member ID.**

**o Define two subclasses: Student and Teacher, which inherit from Member.**

**o Create a Librarian class that inherits from both Student and Teacher (multiple**

**inheritance).**

**3. Library Class:**

**o Implement a Library class to manage the collection of books.**

**o Add methods to the Library class to:**

**▪ Add new books.**

**▪ Remove a book by its ISBN.**

**▪ Search for books by title or author.**

**4. Demonstration:**

**o Instantiate a library and add books (both EBooks and Printed Books) to it.**

**o Demonstrate searching for books using keywords.**

**o Show how a librarian can add and remove books from the system.**

class Book:

def \_\_init\_\_(self, title, author, isbn):

self.title = title

self.author = author

self.isbn = isbn

class EBook(Book):

def \_\_init\_\_(self, title, author, isbn, file\_format):

super().\_\_init\_\_(title, author, isbn)

self.file\_format = file\_format

class PrintedBook(Book):

def \_\_init\_\_(self, title, author, isbn, page\_count):

super().\_\_init\_\_(title, author, isbn)

self.page\_count = page\_count

class Member:

def \_\_init\_\_(self, name, member\_id):

self.name = name

self.member\_id = member\_id

class Student(Member):

pass

class Teacher(Member):

pass

class Librarian(Student, Teacher):

pass

class Library:

def \_\_init\_\_(self):

self.books = []

def add\_book(self, book):

self.books.append(book)

def remove\_book(self, isbn):

for book in self.books:

if book.isbn == isbn:

self.books.remove(book)

print(f"Book with ISBN {isbn} removed successfully.")

return

print(f"Book with ISBN {isbn} not found.")

def search\_book(self, keyword):

results = []

for book in self.books:

if keyword in book.title or keyword in book.author:

results.append(book)

return results

def main():

library = Library()

while True:

print("Library Management System")

print("1. Add Book")

print("2. Remove Book")

print("3. Search Book")

print("4. Exit")

choice = input("Enter your choice: ")

if choice == "1":

title = input("Enter book title: ")

author = input("Enter book author: ")

isbn = input("Enter book ISBN: ")

book\_type = input("Enter book type (EBook or PrintedBook): ")

if book\_type == "EBook":

file\_format = input("Enter file format: ")

book = EBook(title, author, isbn, file\_format)

elif book\_type == "PrintedBook":

page\_count = int(input("Enter page count: "))

book = PrintedBook(title, author, isbn, page\_count)

library.add\_book(book)

print("Book added successfully.")

elif choice == "2":

isbn = input("Enter ISBN of book to remove: ")

library.remove\_book(isbn)

elif choice == "3":

keyword = input("Enter keyword to search: ")

results = library.search\_book(keyword)

if results:

print("Search results:")

for book in results:

print(f"Title: {book.title}, Author: {book.author}, ISBN: {book.isbn}")

else:

print("No books found.")

elif choice == "4":

break

else:

print("Invalid choice. Please try again.")

if \_\_name\_\_ == "\_\_main\_\_":

main()

**OUTPUT:**

Library Management System

1. Add Book

2. Remove Book

3. Search Book

4. Exit

Enter your choice: 1

Enter book title: x

Enter book author: y

Enter book ISBN: z

Enter book type (EBook or PrintedBook): EBook

Enter file format: pdf

Book added successfully.

Library Management System

1. Add Book

2. Remove Book

3. Search Book

4. Exit

Enter your choice:

↑↓ for history. Search history with c-↑/c-↓

**Exercise 2:**

**Advanced E-Commerce System Utilizing Polymorphism**

**A rapidly growing online retail company is looking to upgrade its E-Commerce System.**

**They need the system to manage various types of products, allow users to add them to their**

**shopping carts, apply discounts, and handle different payment methods. To make the system**

**robust, flexible, and scalable, you decide to use Object-Oriented Programming (OOP)**

**principles.**

**Objectives:**

**1. Product Management:**

**o Products in the system belong to different categories such as Electronics and**

**Clothing. Each product category has its own discount logic.**

**o Discounts should be applied based on product types, demonstrating method**

**overriding.**

**2. Shopping Cart Management:**

**o Users should be able to add multiple items to their shopping carts and see the**

**total cost after discounts.**

**o The system should allow merging two shopping carts using operator**

**overloading.**

**3. Payment Processing:**

**o Customers should be able to process payments using various methods (e.g.,**

**credit card, PayPal). Even though Python does not natively support method**

**overloading, it should be simulated to handle different payment methods**

**efficiently.**

**Functional Requirements:**

**1. Products:**

**o Implement a base Product class that represents general products, containing**

**attributes like name and price.**

**o Create derived classes such as Electronics and Clothing that override the base**

**class method for calculating product discounts.**

**2. Shopping Cart:**

**o Implement a ShoppingCart class that can hold a collection of products.**

**o Overload the + operator to merge two shopping carts into one.**

**3. Payment Processing:**

**o Implement a PaymentProcessor class that simulates method overloading to**

**handle different payment methods (e.g., credit card and PayPal) using variable**

**arguments.**

class Product:

def \_\_init\_\_(self, name, price):

self.name = name

self.price = price

def calculate\_discount(self):

return self.price \* 0.1 # 10% discount as a placeholder

class Electronics(Product):

def calculate\_discount(self):

return self.price \* 0.15 # 15% discount for Electronics

class Clothing(Product):

def calculate\_discount(self):

return self.price \* 0.05 # 5% discount for Clothing

class ShoppingCart:

def \_\_init\_\_(self, products=None):

self.products = products or []

def add\_product(self, product):

self.products.append(product)

def calculate\_total(self):

total = sum(product.price for product in self.products)

discount = sum(product.calculate\_discount() for product in self.products)

return total - discount

def \_\_add\_\_(self, other):

combined\_cart = ShoppingCart()

combined\_cart.products = self.products + other.products

return combined\_cart

class PaymentProcessor:

def process\_payment(self, payment\_method, \*args, \*\*kwargs):

if payment\_method == "credit\_card":

self.process\_credit\_card\_payment(\*args, \*\*kwargs)

elif payment\_method == "paypal":

self.process\_paypal\_payment(\*args, \*\*kwargs)

else:

raise ValueError(f"Unsupported payment method: {payment\_method}")

def process\_credit\_card\_payment(self, amount, card\_number, expiration\_date, cvv):

# Implement credit card payment processing

pass

def process\_paypal\_payment(self, amount, paypal\_email, paypal\_password):

# Implement PayPal payment processing

pass

def main():

electronics = Electronics("Smartphone", 500)

clothing = Clothing("T-Shirt", 20)

cart1 = ShoppingCart()

cart1.add\_product(electronics)

cart1.add\_product(clothing)

cart2 = ShoppingCart()

cart2.add\_product(electronics)

print("Cart 1 total:", cart1.calculate\_total())

print("Cart 2 total:", cart2.calculate\_total())

merged\_cart = cart1 + cart2

print("Merged cart total:", merged\_cart.calculate\_total())

payment\_processor = PaymentProcessor()

payment\_processor.process\_payment("credit\_card", 500, "1234567890123456", "123", "2025")

payment\_processor.process\_payment("paypal", 500, "john.doe@example.com", "secret\_password")

if \_\_name\_\_ == "\_\_main\_\_":

main()

**OUTPUT:**

Cart 1 total: 444.0

Cart 2 total: 425.0

Merged cart total: 869.0

***WEEK 7:***

***Exercise 1:***

**Inventory Management System using Python Inheritance**

**Scenario: Designing an Inventory Management System**

**You are tasked with designing an Inventory Management System that handles various types**

**of products. Each product shares common attributes but has specific attributes based on the**

**category of the product. The system must support operations such as adding products,**

**calculating inventory value, applying discounts, and checking stock levels.**

**Problem Definition:**

**We need to manage three types of products:**

**1. Electronics (e.g., phones, laptops).**

**2. Clothing (e.g., shirts, pants).**

**3. Groceries (e.g., fruits, vegetables).**

**All product types share basic attributes like name, price, quantity, and SKU (Stock Keeping**

**Unit). However, they also have specific attributes:**

**• Electronics may have warranty period and brand.**

**• Clothing has attributes like size and material.**

**• Groceries include expiration date and organic status.**

**Additionally, the system needs to:**

**1. Add new products.**

**2. Update stock.**

**3. Calculate the total value of inventory.**

**4. Apply discounts based on product type.**

**5. Check if products are low in stock.**

class Product:

def \_\_init\_\_(self, name, price, quantity, sku):

self.name = name

self.price = price

self.quantity = quantity

self.sku = sku

def calculate\_value(self):

return self.price \* self.quantity

def is\_low\_stock(self, threshold=5):

return self.quantity < threshold

class Electronics(Product):

def \_\_init\_\_(self, name, price, quantity, sku, warranty\_period, brand):

super().\_\_init\_\_(name, price, quantity, sku)

self.warranty\_period = warranty\_period

self.brand = brand

class Clothing(Product):

def \_\_init\_\_(self, name, price, quantity, sku, size, material):

super().\_\_init\_\_(name, price, quantity, sku)

self.size = size

self.material = material

class Groceries(Product):

def \_\_init\_\_(self, name, price, quantity, sku, expiration\_date, organic\_status):

super().\_\_init\_\_(name, price, quantity, sku)

self.expiration\_date = expiration\_date

self.organic\_status = organic\_status

class Inventory:

def \_\_init\_\_(self):

self.products = {}

def add\_product(self, product):

if product.sku in self.products:

print("Product with this SKU already exists.")

else:

self.products[product.sku] = product

print(f"Added {product.name} to inventory.")

def update\_stock(self, sku, quantity):

if sku in self.products:

self.products[sku].quantity += quantity

print(f"Updated stock for {sku}. New quantity: {self.products[sku].quantity}.")

else:

print("Product not found.")

def calculate\_total\_value(self):

total\_value = sum(product.calculate\_value() for product in self.products.values())

return total\_value

def apply\_discount(self, sku, discount\_percentage):

if sku in self.products:

product = self.products[sku]

discount\_amount = product.price \* (discount\_percentage / 100)

product.price -= discount\_amount

print(f"Applied discount to {product.name}. New price: {product.price:.2f}.")

else:

print("Product not found.")

def check\_low\_stock(self):

low\_stock\_products = [product for product in self.products.values() if product.is\_low\_stock()]

return low\_stock\_products

# Example Usage with user-defined input

if \_\_name\_\_ == "\_\_main\_\_":

inventory = Inventory()

while True:

print("\n1. Add Product")

print("2. Update Stock")

print("3. Calculate Total Inventory Value")

print("4. Apply Discount")

print("5. Check Low Stock Items")

print("6. Exit")

choice = input("Enter your choice: ")

if choice == "1":

print("\nSelect Product Type:")

print("1. Electronics")

print("2. Clothing")

print("3. Groceries")

product\_type = input("Enter choice (1/2/3): ")

name = input("Enter product name: ")

price = float(input("Enter product price: "))

quantity = int(input("Enter product quantity: "))

sku = input("Enter SKU: ")

if product\_type == "1":

warranty\_period = input("Enter warranty period: ")

brand = input("Enter brand: ")

product = Electronics(name, price, quantity, sku, warranty\_period, brand)

inventory.add\_product(product)

elif product\_type == "2":

size = input("Enter size: ")

material = input("Enter material: ")

product = Clothing(name, price, quantity, sku, size, material)

inventory.add\_product(product)

elif product\_type == "3":

expiration\_date = input("Enter expiration date (YYYY-MM-DD): ")

organic\_status = input("Is it organic? (yes/no): ").lower() == "yes"

product = Groceries(name, price, quantity, sku, expiration\_date, organic\_status)

inventory.add\_product(product)

else:

print("Invalid product type.")

elif choice == "2":

sku = input("Enter SKU of the product to update stock: ")

quantity = int(input("Enter quantity to add: "))

inventory.update\_stock(sku, quantity)

elif choice == "3":

total\_value = inventory.calculate\_total\_value()

print(f"Total inventory value: ${total\_value:.2f}")

elif choice == "4":

sku = input("Enter SKU of the product to apply discount: ")

discount\_percentage = float(input("Enter discount percentage: "))

inventory.apply\_discount(sku, discount\_percentage)

elif choice == "5":

low\_stock\_items = inventory.check\_low\_stock()

if low\_stock\_items:

print("Low stock items:")

for item in low\_stock\_items:

print(f"{item.name} (Quantity: {item.quantity})")

else:

print("No low stock items.")

elif choice == "6":

break

else:

print("Invalid choice, please try again.")

**OUTPUT:**

1. Add Product

2. Update Stock

3. Calculate Total Inventory Value

4. Apply Discount

5. Check Low Stock Items

6. Exit

Enter your choice: 1

Select Product Type:

1. Electronics

2. Clothing

3. Groceries

Enter choice (1/2/3): 1

Enter product name: fRUITS

Enter product price: 500

Enter product quantity: 5

Enter SKU: 9

Enter warranty period: 1

Enter brand: NIOI

Added fRUITS to inventory.

1. Add Product

2. Update Stock

3. Calculate Total Inventory Value

4. Apply Discount

5. Check Low Stock Items

6. Exit

[ ]:

***Exercise 2:***

**Building a Payment Processing System**

**You are tasked with designing a Payment Processing System that handles multiple payment**

**methods (Credit Card, PayPal, Bank Transfer). Each payment method has unique steps**

**involved in processing payments, but they all share the common interface of processing a**

**payment and issuing a refund.**

**Problem Definition:**

**The system must support the following payment methods:**

**1. Credit Card Payment: Requires card number, expiry date, and CVV to process**

**payments.**

**2. PayPal Payment: Uses a PayPal account email and password.**

**3. Bank Transfer Payment: Processes payments using a bank account number and a sort**

**code.**

**Each payment method has:**

**• A method to process payments.**

**• A method to issue refunds.**

**• Error handling for failed payments.**

class PaymentMethod:

def process\_payment(self, amount):

raise NotImplementedError("This method should be overridden in subclasses")

def issue\_refund(self, amount):

raise NotImplementedError("This method should be overridden in subclasses")

class CreditCardPayment(PaymentMethod):

def \_\_init\_\_(self, card\_number, expiry\_date, cvv):

self.card\_number = card\_number

self.expiry\_date = expiry\_date

self.cvv = cvv

def process\_payment(self, amount):

if self.validate\_card():

print(f"Processing credit card payment of ${amount}")

else:

print("Credit card validation failed.")

def issue\_refund(self, amount):

print(f"Issuing credit card refund of ${amount}")

def validate\_card(self):

return True

class PayPalPayment(PaymentMethod):

def \_\_init\_\_(self, email, password):

self.email = email

self.password = password

def process\_payment(self, amount):

if self.authenticate():

print(f"Processing PayPal payment of ${amount}")

else:

print("PayPal authentication failed.")

def issue\_refund(self, amount):

print(f"Issuing PayPal refund of ${amount}")

def authenticate(self):

return True

class BankTransferPayment(PaymentMethod):

def \_\_init\_\_(self, account\_number, sort\_code):

self.account\_number = account\_number

self.sort\_code = sort\_code

def process\_payment(self, amount):

print(f"Processing bank transfer of ${amount}")

def issue\_refund(self, amount):

print(f"Issuing bank transfer refund of ${amount}")

# User-defined input usage

def main():

print("Select Payment Method:")

print("1. Credit Card")

print("2. PayPal")

print("3. Bank Transfer")

choice = input("Enter choice (1/2/3): ")

if choice == "1":

card\_number = input("Enter Credit Card Number: ")

expiry\_date = input("Enter Expiry Date (MM/YY): ")

cvv = input("Enter CVV: ")

amount = float(input("Enter Payment Amount: "))

credit\_card\_payment = CreditCardPayment(card\_number, expiry\_date, cvv)

credit\_card\_payment.process\_payment(amount)

refund\_choice = input("Would you like to issue a refund? (y/n): ")

if refund\_choice.lower() == 'y':

refund\_amount = float(input("Enter Refund Amount: "))

credit\_card\_payment.issue\_refund(refund\_amount)

elif choice == "2":

email = input("Enter PayPal Email: ")

password = input("Enter PayPal Password: ")

amount = float(input("Enter Payment Amount: "))

paypal\_payment = PayPalPayment(email, password)

paypal\_payment.process\_payment(amount)

refund\_choice = input("Would you like to issue a refund? (y/n): ")

if refund\_choice.lower() == 'y':

refund\_amount = float(input("Enter Refund Amount: "))

paypal\_payment.issue\_refund(refund\_amount)

elif choice == "3":

account\_number = input("Enter Bank Account Number: ")

sort\_code = input("Enter Sort Code: ")

amount = float(input("Enter Payment Amount: "))

bank\_transfer\_payment = BankTransferPayment(account\_number, sort\_code)

bank\_transfer\_payment.process\_payment(amount)

refund\_choice = input("Would you like to issue a refund? (y/n): ")

if refund\_choice.lower() == 'y':

refund\_amount = float(input("Enter Refund Amount: "))

bank\_transfer\_payment.issue\_refund(refund\_amount)

else:

print("Invalid choice, please try again.")

if \_\_name\_\_ == "\_\_main\_\_":

main()

**OUTPUT:**

Select Payment Method:

1. Credit Card

2. PayPal

3. Bank Transfer

Enter choice (1/2/3): 1

Enter Credit Card Number: 122

Enter Expiry Date (MM/YY): 10/27

Enter CVV: 805

Enter Payment Amount: 5000

Processing credit card payment of $5000.0

Would you like to issue a refund? (y/n):

**WEEK 8:**

**Exercise 1:**

File Handling in Python with Error and Exception Handling

You are tasked with building a simple data processing system for an e-commerce company

that handles product inventories and customer reviews. The company uses two types of files to

store data:

1. CSV Files: The product inventory is stored in a CSV file with the following columns:

Product IL), Product Name, Category, Price, and Stock Quantity.

2, JSON Files: Customer reviews for each product are stored in separate JSON files. Each

review file contains a list of reviews for a product, where each review has the following

fields: Review IL), User ID, Rating, Comment, and Date.

Your system needs to process these files to provide insights and updates on product availability,

pricing trends, and customer feedback. Additionally, the system should allow adding new

products and updating product information in the CSV file, while handling errors and

exceptions gracefully.

System Requirements:

1. Product Management (CSV File):

• Add New Products: Write a function to add a new product to the inventory CSV file.

• Update Stock and Price: Implement a function to update the stock quantity and price

of an existing product in the CSV file.

Check Product Availability: Write a function that reads the CSV file to check if a

product is in stock and, if so, how many units are available.

2. Customer Review Management (JSON Files):

Add Customer Review: Implement a function that adds a new customer review to the

corresponding JSON file for a product.

Average Rating Calculation: Write a function that reads a product's JSON review file

and calculates its average rating.

Review Search: Implement a search function that reads the JSON file and allows users

to search for reviews containing specific keywords in the comment section.

3. Data Analysis:

• Top Rated Products: Create a function that reads all product review JSON files,

calculates the average rating for each product, and lists the top 5 highest-rated products.

• Out of Stock Products: Write a function that reads the CSV inventory file and lists all

products that are out of stock.

• Price Trends: Implement a function to read the CSV file and identify products whose

prices have increased or decreased in the past month.

4. Error and Exception Handling:

• Implement error handling for situations such as:

File Not Found: When the CSV or JSON file is missing.

Invalid Data Format: If the data in the files does not match the expected

structure.

File Corruption: When a file contains corrupted data.

Read/Write Permissions: When the system does not have permission to read

or write to a file.

import csv

import json

import os

def add\_product(file\_path):

product\_id = input("Enter Product ID: ")

product\_name = input("Enter Product Name: ")

category = input("Enter Category: ")

price = float(input("Enter Price: "))

stock\_quantity = int(input("Enter Stock Quantity: "))

try:

with open(file\_path, mode='a', newline='') as file:

writer = csv.writer(file)

writer.writerow([product\_id, product\_name, category, price, stock\_quantity])

print("Product added successfully.")

except PermissionError:

print("Error: Permission denied when trying to write to the file.")

except Exception as e:

print(f"An unexpected error occurred: {e}")

def update\_product(file\_path):

product\_id = input("Enter Product ID to update: ")

new\_price = float(input("Enter New Price: "))

new\_stock\_quantity = int(input("Enter New Stock Quantity: "))

try:

updated\_rows = []

with open(file\_path, mode='r') as file:

reader = csv.reader(file)

for row in reader:

if row[0] == product\_id:

row[3] = str(new\_price)

row[4] = str(new\_stock\_quantity)

updated\_rows.append(row)

with open(file\_path, mode='w', newline='') as file:

writer = csv.writer(file)

writer.writerows(updated\_rows)

print("Product updated successfully.")

except FileNotFoundError:

print("Error: The CSV file was not found.")

except Exception as e:

print(f"An unexpected error occurred: {e}")

def check\_availability(file\_path):

product\_id = input("Enter Product ID to check availability: ")

try:

with open(file\_path, mode='r') as file:

reader = csv.reader(file)

for row in reader:

if row[0] == product\_id:

return f"Available stock: {row[4]}"

return "Product not found."

except FileNotFoundError:

print("Error: The CSV file was not found.")

def add\_review():

product\_id = input("Enter Product ID for the review: ")

review\_id = input("Enter Review ID: ")

user\_id = input("Enter User ID: ")

rating = int(input("Enter Rating (1-5): "))

comment = input("Enter Comment: ")

date = input("Enter Date (YYYY-MM-DD): ")

review = {

"Review ID": review\_id,

"User ID": user\_id,

"Rating": rating,

"Comment": comment,

"Date": date

}

file\_name = f"{product\_id}\_reviews.json"

try:

if not os.path.exists(file\_name):

with open(file\_name, 'w') as file:

json.dump([], file)

with open(file\_name, 'r+') as file:

reviews = json.load(file)

reviews.append(review)

file.seek(0)

json.dump(reviews, file, indent=4)

print("Review added successfully.")

except Exception as e:

print(f"An error occurred while adding the review: {e}")

def average\_rating(product\_id):

file\_name = f"{product\_id}\_reviews.json"

try:

with open(file\_name) as file:

reviews = json.load(file)

total\_rating = sum(review['Rating'] for review in reviews)

average = total\_rating / len(reviews) if reviews else 0

return f"Average Rating: {average:.2f}"

except FileNotFoundError:

return "Review file not found."

except json.JSONDecodeError:

return "Error: Invalid JSON format."

def search\_reviews():

product\_id = input("Enter Product ID to search reviews: ")

keyword = input("Enter keyword to search in comments: ")

file\_name = f"{product\_id}\_reviews.json"

try:

with open(file\_name) as file:

reviews = json.load(file)

matching\_reviews = [review for review in reviews if keyword.lower() in review['Comment'].lower()]

return matching\_reviews if matching\_reviews else "No matching reviews found."

except FileNotFoundError:

return "Review file not found."

def top\_rated\_products(product\_ids):

ratings = {}

for product\_id in product\_ids:

avg\_rating\_result = average\_rating(product\_id)

if isinstance(avg\_rating\_result, str) and "Average Rating" in avg\_rating\_result:

ratings[product\_id] = float(avg\_rating\_result.split(": ")[1])

top\_products = sorted(ratings.items(), key=lambda x: x[1], reverse=True)[:5]

return top\_products

def out\_of\_stock\_products(file\_path):

out\_of\_stock = []

try:

with open(file\_path) as file:

reader = csv.reader(file)

for row in reader:

if int(row[4]) == 0:

out\_of\_stock.append(row[1])

return out\_of\_stock if out\_of\_stock else ["No products are out of stock."]

except FileNotFoundError:

return "Inventory file not found."

def price\_trends(current\_file\_path, previous\_file\_path):

current\_prices = {}

previous\_prices = {}

try:

with open(current\_file\_path) as current\_file:

reader = csv.reader(current\_file)

for row in reader:

current\_prices[row[0]] = float(row[3])

with open(previous\_file\_path) as previous\_file:

reader = csv.reader(previous\_file)

for row in reader:

previous\_prices[row[0]] = float(row[3])

trends = {}

for product\_id in current\_prices.keys():

if product\_id in previous\_prices:

change = current\_prices[product\_id] - previous\_prices[product\_id]

trends[product\_id] = change

return trends

except FileNotFoundError as e:

return f"File not found: {e.filename}"

def main():

inventory\_file\_path = 'inventory.csv'

while True:

print("\nE-commerce Data Processing System")

print("1. Add Product")

print("2. Update Product")

print("3. Check Product Availability")

print("4. Add Customer Review")

print("5. Calculate Average Rating")

print("6. Search Reviews")

print("7. List Out of Stock Products")

print("8. Show Top Rated Products")

choice = input("\nSelect an option (or 'q' to quit): ")

if choice == '1':

add\_product(inventory\_file\_path)

elif choice == '2':

update\_product(inventory\_file\_path)

elif choice == '3':

print(check\_availability(inventory\_file\_path))

elif choice == '4':

add\_review()

elif choice == '5':

product\_id = input("Enter Product ID to calculate average rating: ")

print(average\_rating(product\_id))

elif choice == '6':

results = search\_reviews()

if isinstance(results, list):

for review in results:

print(review)

else:

print(results)

elif choice == '7':

out\_of\_stock\_list = out\_of\_stock\_products(inventory\_file\_path)

for item in out\_of\_stock\_list:

print(item)

elif choice == '8':

example\_product\_ids = ['P001', 'P002', 'P003', 'P004', 'P005']

top\_products\_list = top\_rated\_products(example\_product\_ids)

for product in top\_products\_list:

print(product)

elif choice.lower() == 'q':

break

else:

print("Invalid option. Please try again.")

if \_\_name\_\_ == "\_\_main\_\_":

main()

**OUPUT:**

E-commerce Data Processing System

1. Add Product

2. Update Product

3. Check Product Availability

4. Add Customer Review

5. Calculate Average Rating

6. Search Reviews

7. List Out of Stock Products

8. Show Top Rated Products

Select an option (or 'q' to quit): 1

Enter Product ID: 123

Enter Product Name: XYZ

Enter Category: GROCERY

Enter Price: 1000

Enter Stock Quantity: 9

Product added successfully.

E-commerce Data Processing System

1. Add Product

2. Update Product

3. Check Product Availability

4. Add Customer Review

5. Calculate Average Rating

6. Search Reviews

7. List Out of Stock Products

8. Show Top Rated Products

Select an option (or 'q' to quit): q

**WEEK 9:**

***Exercise 1:***

**Sales Performance Analysis of XYZ Company Data Overview:**

**The sales data of XYZ Company contains the following attributes for each transaction:   
• Date: The date of the sale.   
• Product Name: The name of the product sold.   
• Units Sold: The number of units sold in the transaction.   
• Revenue: The total revenue generated by the sale.   
• Region: The geographical region where the sale occurred.   
• Discount Offered (%): The percentage of discount offered on the sale.   
• Salesperson: The name of the salesperson responsible for the sale.   
Insert At least 20 data and create the dataframe.  
 1. What are the top 3 sales transactions with the highest revenue?   
2. How many units of each product were sold?   
3. What is the total revenue after applying discounts?   
4. Which sales transaction had the highest discount offered, and how much revenue did it generate after applying the discount?   
5. Which salesperson generated the highest total revenue?   
6. What is the average discount offered by each salesperson?   
7. How much revenue was generated in each region?   
8. In which region did Alice generate the highest sales?   
9. Which product generated the highest revenue per unit sold?   
10. How many transactions were rated as "High" performance?   
11. Which salesperson sold the most units in the North region without offering any discount?   
12. What is the average revenue per unit sold in each region for each product?   
13. Which salesperson has the highest average revenue after discounts, and how does it compare between regions?   
14. What is the cumulative total revenue over time for each salesperson?   
15. For each salesperson, rank the transactions by revenue, and find the top 2 transactions for each.   
16. How has the total revenue generated by each product changed over time? Show cumulative revenue for each product per day.   
17. Analyze how discounts affect revenue. For each product, what is the average revenue generated with a discount compared to without a discount?   
18. What is the weighted average discount offered by each salesperson, weighted by the revenue they generated?   
19. What percentage of the total revenue does each region contribute? Compare it to the total revenue per region.**

import pandas as pd

import numpy as np

np.random.seed(0)

data = {

"Date": pd.date\_range(start="2023-01-01", periods=20, freq='D'),

"Product Name": np.random.choice(['Product A', 'Product B', 'Product C'], 20),

"Units Sold": np.random.randint(1, 10, size=20),

"Revenue": np.random.uniform(100, 1000, size=20).round(2),

"Region": np.random.choice(['North', 'South', 'East', 'West'], 20),

"Discount Offered (%)": np.random.randint(0, 30, size=20),

"Salesperson": np.random.choice(['Alice', 'Bob', 'Charlie', 'David'], 20)

}

sales\_data = pd.DataFrame(data)

sales\_data['Revenue After Discount'] = sales\_data['Revenue'] \* (1 - sales\_data['Discount Offered (%)'] / 100)

top\_revenue\_transactions = sales\_data.nlargest(3, 'Revenue')

units\_sold\_per\_product = sales\_data.groupby('Product Name')['Units Sold'].sum()

total\_revenue\_after\_discount = sales\_data['Revenue After Discount'].sum()

max\_discount\_transaction = sales\_data.loc[sales\_data['Discount Offered (%)'].idxmax()]

highest\_revenue\_salesperson = sales\_data.groupby('Salesperson')['Revenue'].sum().idxmax()

average\_discount\_per\_salesperson = sales\_data.groupby('Salesperson')['Discount Offered (%)'].mean()

revenue\_per\_region = sales\_data.groupby('Region')['Revenue'].sum()

alice\_sales = sales\_data[sales\_data['Salesperson'] == 'Alice']

region\_highest\_alice = alice\_sales.groupby('Region')['Revenue'].sum().idxmax()

sales\_data['Revenue Per Unit'] = sales\_data['Revenue'] / sales\_data['Units Sold'].replace(0, np.nan)

highest\_revenue\_per\_unit\_product = sales\_data.loc[sales\_data['Revenue Per Unit'].idxmax()]

high\_performance\_transactions\_count = (sales\_data['Revenue'] > 500).sum()

north\_no\_discount\_sales = sales\_data[(sales\_data['Region'] == 'North') & (sales\_data['Discount Offered (%)'] == 0)]

top\_north\_salesperson\_no\_discount = north\_no\_discount\_sales.groupby('Salesperson')['Units Sold'].sum().idxmax()

average\_revenue\_per\_unit\_region\_product = sales\_data.groupby(['Region', 'Product Name']).apply(

lambda x: (x['Revenue'] / x['Units Sold'].replace(0, np.nan)).mean()

)

average\_revenue\_after\_discount = sales\_data.groupby('Salesperson')['Revenue After Discount'].mean()

highest\_avg\_revenue\_salesperson = average\_revenue\_after\_discount.idxmax()

sales\_data['Cumulative Revenue'] = sales\_data.groupby('Salesperson')['Revenue'].cumsum()

top\_transactions\_per\_salesperson = sales\_data.sort\_values('Revenue', ascending=False).groupby('Salesperson').head(2)

cumulative\_revenue\_product\_over\_time = sales\_data.groupby(['Date', 'Product Name'])['Revenue'].sum().groupby(level=1).cumsum()

avg\_revenue\_with\_discount = sales\_data[sales\_data['Discount Offered (%)'] > 0].groupby('Product Name')['Revenue After Discount'].mean()

avg\_revenue\_without\_discount = sales\_data[sales\_data['Discount Offered (%)'] == 0].groupby('Product Name')['Revenue'].mean()

discount\_impact\_analysis = pd.DataFrame({

'Avg Revenue With Discount': avg\_revenue\_with\_discount,

'Avg Revenue Without Discount': avg\_revenue\_without\_discount

}).fillna(0)

weighted\_average\_discount = (sales\_data['Discount Offered (%)'] \* sales\_data['Revenue']).groupby(sales\_data['Salesperson']).sum() / sales\_data.groupby('Salesperson')['Revenue'].sum()

total\_revenue = sales\_data['Revenue'].sum()

percentage\_revenue\_by\_region = (revenue\_per\_region / total\_revenue) \* 100

comparison\_total\_revenue\_per\_region = pd.DataFrame({

'Total Revenue': revenue\_per\_region,

'Percentage of Total Revenue': percentage\_revenue\_by\_region

})

print("Top 3 Sales Transactions by Revenue:\n", top\_revenue\_transactions)

print("\nUnits Sold for Each Product:\n", units\_sold\_per\_product)

print("\nTotal Revenue After Discounts:", total\_revenue\_after\_discount)

print("\nTransaction with Highest Discount Offered:\n", max\_discount\_transaction)

print("\nSalesperson with Highest Total Revenue:", highest\_revenue\_salesperson)

print("\nAverage Discount Offered by Each Salesperson:\n", average\_discount\_per\_salesperson)

print("\nRevenue Generated in Each Region:\n", revenue\_per\_region)

print("\nRegion where Alice Generated Highest Sales:", region\_highest\_alice)

print("\nProduct with Highest Revenue Per Unit Sold:\n", highest\_revenue\_per\_unit\_product[['Product Name', 'Revenue Per Unit']])

print("\nNumber of High Performance Transactions:", high\_performance\_transactions\_count)

print("\nSalesperson in North Region with Most Units Sold (No Discount):", top\_north\_salesperson\_no\_discount)

print("\nAverage Revenue Per Unit Sold in Each Region for Each Product:\n", average\_revenue\_per\_unit\_region\_product)

print("\nSalesperson with Highest Average Revenue After Discounts:", highest\_avg\_revenue\_salesperson)

print("\nCumulative Total Revenue Over Time for Each Salesperson:\n", sales\_data[['Date', 'Salesperson', 'Cumulative Revenue']])

print("\nTop 2 Transactions for Each Salesperson:\n", top\_transactions\_per\_salesperson)

print("\nCumulative Revenue Over Time for Each Product:\n", cumulative\_revenue\_product\_over\_time)

print("\nAverage Revenue With vs Without Discount:\n", discount\_impact\_analysis)

print("\nWeighted Average Discount Offered by Each Salesperson:\n", weighted\_average\_discount)

print("\nPercentage of Total Revenue by Region:\n", comparison\_total\_revenue\_per\_region)

**OUTPUT:**

Top 3 Sales Transactions by Revenue:

Date Product Name Units Sold Revenue Region Discount Offered (%) \

8 2023-01-09 Product A 4 949.37 West 9

1 2023-01-02 Product B 5 796.81 East 15

12 2023-01-13 Product B 4 727.87 North 0

Salesperson Revenue After Discount

8 Alice 863.9267

1 Charlie 677.2885

12 Charlie 727.8700

Units Sold for Each Product:

Product Name

Product A 30

Product B 30

Product C 25

Name: Units Sold, dtype: int32

Total Revenue After Discounts: 9376.795900000003

Transaction with Highest Discount Offered:

Date 2023-01-19 00:00:00

Product Name Product B

Units Sold 5

Revenue 383.89

Region West

Discount Offered (%) 29

Salesperson Charlie

Revenue After Discount 272.5619

Name: 18, dtype: object

Salesperson with Highest Total Revenue: Charlie

Average Discount Offered by Each Salesperson:

Salesperson

Alice 15.000000

Bob 15.500000

Charlie 9.375000

David 11.666667

Name: Discount Offered (%), dtype: float64

Revenue Generated in Each Region:

Region

East 2583.59

North 4283.50

South 338.10

West 3312.99

Name: Revenue, dtype: float64

Region where Alice Generated Highest Sales: West

Product with Highest Revenue Per Unit Sold:

Product Name Product A

Revenue Per Unit 703.57

Name: 15, dtype: object

Number of High Performance Transactions: 11

Salesperson in North Region with Most Units Sold (No Discount): Charlie

Average Revenue Per Unit Sold in Each Region for Each Product:

Region Product Name

East Product A 703.570000

Product B 106.389750

Product C 109.311667

North Product A 285.939444

Product B 142.408333

Product C 83.131250

South Product A 37.566667

West Product A 224.561250

Product B 277.679333

Product C 218.413333

dtype: float64

Salesperson with Highest Average Revenue After Discounts: Alice

Cumulative Total Revenue Over Time for Each Salesperson:

Date Salesperson Cumulative Revenue

0 2023-01-01 Charlie 338.10

1 2023-01-02 Charlie 1134.91

2 2023-01-03 Bob 510.54

3 2023-01-04 Alice 611.59

4 2023-01-05 David 116.91

5 2023-01-06 Bob 1166.41

6 2023-01-07 Bob 1817.30

7 2023-01-08 David 772.15

8 2023-01-09 Alice 1560.96

9 2023-01-10 Charlie 1848.55

10 2023-01-11 Charlie 2272.11

11 2023-01-12 David 1265.48

12 2023-01-13 Charlie 2999.98

13 2023-01-14 David 1419.68

14 2023-01-15 David 2119.77

15 2023-01-16 David 2823.34

16 2023-01-17 Charlie 3289.32

17 2023-01-18 Bob 2033.33

18 2023-01-19 Charlie 3673.21

19 2023-01-20 Charlie 4100.55

Top 2 Transactions for Each Salesperson:

Date Product Name Units Sold Revenue Region Discount Offered (%) \

8 2023-01-09 Product A 4 949.37 West 9

1 2023-01-02 Product B 5 796.81 East 15

12 2023-01-13 Product B 4 727.87 North 0

15 2023-01-16 Product A 1 703.57 East 4

14 2023-01-15 Product C 8 700.09 North 18

5 2023-01-06 Product C 6 655.87 East 16

6 2023-01-07 Product A 1 650.89 North 17

3 2023-01-04 Product B 1 611.59 West 21

Salesperson Revenue After Discount Revenue Per Unit Cumulative Revenue

8 Alice 863.9267 237.342500 1560.96

1 Charlie 677.2885 159.362000 1134.91

12 Charlie 727.8700 181.967500 2999.98

15 David 675.4272 703.570000 2823.34

14 David 574.0738 87.511250 2119.77

5 Bob 550.9308 109.311667 1166.41

6 Bob 540.2387 650.890000 1817.30

3 Alice 483.1561 611.590000 611.59

Cumulative Revenue Over Time for Each Product:

Date Product Name

2023-01-01 Product A 338.10

2023-01-02 Product B 796.81

2023-01-03 Product A 848.64

2023-01-04 Product B 1408.40

2023-01-05 Product B 1525.31

2023-01-06 Product C 655.87

2023-01-07 Product A 1499.53

2023-01-08 Product C 1311.11

2023-01-09 Product A 2448.90

2023-01-10 Product A 3162.54

2023-01-11 Product A 3586.10

2023-01-12 Product C 1804.44

2023-01-13 Product B 2253.18

2023-01-14 Product C 1958.64

2023-01-15 Product C 2658.73

2023-01-16 Product A 4289.67

2023-01-17 Product B 2542.52

2023-01-18 Product B 2758.55

2023-01-19 Product B 3142.44

2023-01-20 Product B 3569.78

Name: Revenue, dtype: float64

Average Revenue With vs Without Discount:

Avg Revenue With Discount Avg Revenue Without Discount

Product Name

Product A 579.388367 423.56

Product B 343.557657 727.87

Product C 468.826420 0.00

Weighted Average Discount Offered by Each Salesperson:

Salesperson

Alice 13.701645

Bob 15.566853

Charlie 8.502001

David 9.292278

dtype: float64

Percentage of Total Revenue by Region:

Total Revenue Percentage of Total Revenue

Region

East 2583.59 24.563090

North 4283.50 40.724726

South 338.10 3.214434

West 3312.99 31.497750