AIAC ASSIGNMENT

2303A51098

Shivani

Batch – 02

Task 1:

Employee Data: Create Python code that defines a class named `Employee` with the following attributes: `empid`, `empname`, `designation`, `basic_salary`, and `exp`. Implement a method `display_details()` to print all employee details. Implement another method `calculate_allowance()` to determine additional allowance based on experience:

- If `exp > 10 years` → allowance = 20% of `basic_salary`
- If `5 ≤ exp ≤ 10 years` → allowance = 10% of `basic_salary`
- If `exp < 5 years` → allowance = 5% of `basic_salary`

Finally, create at least one instance of the `Employee` class, call the `display_details()` method, and print the calculated allowance.

Code:

```
#employee data
class Employee:
    def __init__(self, emp_id, name, designation, salary, experience):
        self.emp_id = emp_id
        self.name = name
        self.designation = designation
        self.salary = salary
        self.experience = experience
    def display_employee_details(self):
        print(f"Employee ID: {self.emp_id}")
        print(f"Name: {self.name}")
        print(f"Designation: {self.designation}")
        print(f"Salary: {self.salary}")
        print(f"Experience: {self.experience} years")
    def calculate_allowance(self):
        if self.experience > 10:
            allowance = 0.2 * self.salary
        elif 5< self.experience <=10:
            allowance = 0.1 * self.salary
        else:
            allowance = 0.05 * self.salary
        print(f"Allowance for {self.name} is: {allowance}")
        print(f"Total Salary including Allowance: {self.salary + allowanc
# Example usage
emp1 = Employee(101, "John Doe", "Software Engineer", 60000, 12)
emp1.display_employee_details()
emp1.calculate_allowance()
```

Output:

```
Employee ID: 101
Name: John Doe
Designation: Software Engineer
Salary: 60000
Experience: 12 years
Allowance for John Doe is: 12000.0
Total Salary including Allowance: 72000.0
```

Task 2:

Electricity Bill Calculation- Create Python code that defines a class named `ElectricityBill` with attributes: `customer_id`, `name`, and `units_consumed`. Implement a method `display_details()` to print customer details, and a method `calculate_bill()` where:

- Units ≤ 100 → ₹5 per unit
- 101 to 300 units → ₹7 per unit
- More than 300 units → ₹10 per unit

Create a bill object, display details, and print the total bill amount.

Code:

```python
#electricity bill
class Electricity:
    def __init__(self, customer_id, name, units_consumed):
        self.customer_id = customer_id
        self.name = name
        self.units_consumed = units_consumed
    def display_customer_details(self):
        print(f"Customer ID: {self.customer_id}")
        print(f"Name: {self.name}")
        print(f"Units Consumed: {self.units_consumed} kWh")
    def calculate_bill(self):
        if self.units_consumed <= 100:
            rate = 5
        elif 101 <= self.units_consumed <= 300:
            rate = 7
        else:
            rate = 10
        bill_amount = self.units_consumed * rate
        print(f"Electricity Bill for {self.name} is: ${bill_amount:.2f}")
# Example usage
customer1 = Electricity(201, "Alice Smith", 250)
customer1.display_customer_details()
customer1.calculate_bill()
```

Output:

```
Customer ID: 201
Name: Alice Smith
Units Consumed: 250 kWh
Electricity Bill for Alice Smith is: $1750.00
```

Task 3:

Product Discount Calculation- Create Python code that defines a class named `Product` with attributes: `product_id`, `product_name`, `price`, and `category`. Implement a method `display_details()` to print product details. Implement another method `calculate_discount()` where:

- Electronics → 10% discount
- Clothing → 15% discount
- Grocery → 5% discount

Create at least one product object, display details, and print the final price after discount.

Code:

```python
#Product discount
class Product:
    def __init__(self, product_id, name, price, category):
        self.product_id = product_id
        self.name = name
        self.price = price
        self.category = category
    def display_product_details(self):
        print(f"Product ID: {self.product_id}")
        print(f"Name: {self.name}")
        print(f"Price: ${self.price:.2f}")
        print(f"Category: {self.category}")
    def calculate_discount(self):
        if self.category.lower() == "electronics":
            discount = 0.10 * self.price
        elif self.category.lower() == "clothing":
            discount = 0.15 * self.price
        else:
            discount = 0.05 * self.price
        print(f"Discount for {self.name} is: ${discount:.2f}")
        print(f"Price after Discount: ${self.price - discount:.2f}")
# Example usage
product1 = Product(301, "Smartphone", 800, "Electronics")
product1.display_product_details()
product1.calculate_discount()
```

Output:

```
Product ID: 301
Name: Smartphone
Price: $800.00
Category: Electronics
Discount for Smartphone is: $80.00
Price after Discount: $720.00
```

Task 4:

Book Late Fee Calculation- Create Python code that defines a class named `LibraryBook` with attributes: `book_id`, `title`, `author`, `borrower`, and `days_late`. Implement a method `display_details()` to print book details, and a method `calculate_late_fee()` where:

- Days late ≤ 5 → ₹5 per day
- 6 to 10 days late → ₹7 per day
- More than 10 days late → ₹10 per day

Create a book object, display details, and print the late fee.

Code:

```python
#book late fee
class LibraryBook:
    def __init__(self, book_id, title, author, borrower, days_late):
        self.book_id = book_id
        self.title = title
        self.author = author
        self.borrower = borrower
        self.days_late = days_late
    def display_book_details(self):
        print(f"Book ID: {self.book_id}")
        print(f"Title: {self.title}")
        print(f"Author: {self.author}")
        print(f"Borrower: {self.borrower}")
        print(f"Days Late: {self.days_late} days")
    def calculate_late_fee(self):
        if self.days_late <= 5:
            fee_per_day = 5
        elif 6 <= self.days_late <= 10:
            fee_per_day = 7
        else:
            fee_per_day = 10
        total_fee = self.days_late * fee_per_day
        print(f"Late Fee for {self.borrower} is: ${total_fee:.2f}")
# Example usage
book1 = LibraryBook(401, "The Great Gatsby", "F. Scott Fitzgerald", "Bob
book1.display_book_details()
book1.calculate_late_fee()
```

Output:

```
3/python.exe c:/Users/shiva/OneDrive/Documents/python/library.py
Book ID: 401
Title: The Great Gatsby
Author: F. Scott Fitzgerald
Borrower: Bob Johnson
Days Late: 8 days
Late Fee for Bob Johnson is: $56.00
```

Task 5:

Student Performance Report - Define a function

`student_report(student_data)` that accepts a dictionary containing

student names and their marks. The function should:

- Calculate the average score for each student

- Determine pass/fail status (pass ≥ 40)

- Return a summary report as a list of dictionaries

Use Copilot suggestions as you build the function and format the

output.

Code:

```python
#student performance report
def student_performance_report(student_data):
    for student in student_data:
        name = student.get('name')
        grades = student.get('grades', [])
        if not grades:
            print(f"Student: {name} has no grades available.")
            continue
        average_grade = sum(grades) / len(grades)
        status = "Pass" if average_grade >= 50 else "Fail"
        print(f"Student: {name}")
        print(f"Grades: {grades}")
        print(f"Average Grade: {average_grade:.2f}")
        print(f"Status: {status}")
# Example usage
students = [
    {'name': 'Alice', 'grades': [85, 78, 92]},
    {'name': 'Bob', 'grades': [45, 56, 39]},
    {'name': 'Charlie', 'grades': [60, 70, 80]}
]
student_performance_report(students)
```

Output:

```
Student: Alice
Grades: [85, 78, 92]
Average Grade: 85.00
Status: Pass
Student: Bob
Grades: [45, 56, 39]
Average Grade: 46.67
Status: Fail
Student: Charlie
Grades: [60, 70, 80]
Average Grade: 70.00
Status: Pass
```

Task 6:

Taxi Fare Calculation-Create Python code that defines a class named `TaxiRide` with attributes: `ride_id`, `driver_name`, `distance_km`, and `waiting_time_min`. Implement a method `display_details()` to print ride details, and a method `calculate_fare()` where:

- ₹15 per km for the first 10 km
- ₹12 per km for the next 20 km
- ₹10 per km above 30 km
- Waiting charge: ₹2 per minute

Create a ride object, display details, and print the total fare.

Code:

```python
#taxi fare
class TaxiRide:
    def __init__(self, ride_id, driver_name, distance_km, waiting_time_min):
        self.ride_id = ride_id
        self.driver_name = driver_name
        self.distance_km = distance_km
        self.waiting_time_min = waiting_time_min
    def display_ride_details(self):
        print(f"Ride ID: {self.ride_id}")
        print(f"Driver Name: {self.driver_name}")
        print(f"Distance: {self.distance_km} km")
        print(f"Waiting Time: {self.waiting_time_min} minutes")
    def calculate_fare(self):
        if self.distance_km <= 10:
            fare_per_km = 15
        elif 11 <= self.distance_km <= 20:
            fare_per_km = 12
        else:
            fare_per_km = 10
        waiting_charge_per_min = 2
        total_fare = (self.distance_km * fare_per_km) + (self.waiting_time_min * wait
        print(f"Total Fare for the ride with {self.driver_name} is: ${total_fare:.2f}
# Example usage
ride1 = TaxiRide(101, "Carlos Mendez", 18, 15)
ride1.display_ride_details()
ride1.calculate_fare()
```

Output:

```
Ride ID: 101
Driver Name: Carlos Mendez
Distance: 18 km
Waiting Time: 15 minutes
Total Fare for the ride with Carlos Mendez is: $246.00
```

Task 7:

Statistics Subject Performance - Create a Python function

`statistics_subject(scores_list)` that accepts a list of 60 student scores

and computes key performance statistics. The function should return

the following:

- Highest score in the class

- Lowest score in the class

- Class average score

- Number of students passed (score ≥ 40)

- Number of students failed (score < 40)

Allow Copilot to assist with aggregations and logic

Code:

```python
#statistics performance
def statistics_subject(scores_list):
    if not scores_list:
        print("No scores available.")
        return
    average_score = sum(scores_list) / len(scores_list)
    highest_score = max(scores_list)
    lowest_score = min(scores_list)
    print(f"Scores: {scores_list}")
    print(f"Average Score: {average_score:.2f}")
    print(f"Highest Score: {highest_score}")
    print(f"Lowest Score: {lowest_score}")
    print('pass' if average_score >= 40 else 'fail')
# Example usage
scores = [88, 76, 92, 85, 69, 95, 73]
statistics_subject(scores)
```

Output:

```
Scores: [88, 76, 92, 85, 69, 95, 73]
Average Score: 82.57
Highest Score: 95
Lowest Score: 69
pass
```

Task Description #8 (Transparency in Algorithm Optimization)

Task: Use AI to generate two solutions for checking prime numbers:

• Naive approach(basic)

• Optimized approach

Prompt:

“Generate Python code for two prime-checking methods and explain how the optimized version improves performance.”

Expected Output:

• Code for both methods.

• Transparent explanation of time complexity.

• Comparison highlighting efficiency improvements.

Code:

```python
#Generate Python code for two prime-checking methods and explain how the optimized version improves p
#primeoptimal.py
import math
# Basic prime-checking method
def is_prime_basic(n):
    if n <= 1:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
# Optimized prime-checking method
def is_prime_optimized(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    for i in range(5, int(math.sqrt(n)) + 1, 6):
        if n % i == 0 or n % (i + 2) == 0:
            return False
    return True
# Example usage
number = 29
print(f"Basic method: Is {number} prime? {is_prime_basic(number)}")
print(f"Optimized method: Is {number} prime? {is_prime_optimized(number)}")
# Explanation of optimization:
# The optimized method improves performance by reducing the number of checks needed to determine if a
# Instead of checking all numbers up to n-1, it only checks up to the square root of n.
# Additionally, it skips even numbers and multiples of 3 after initial checks, significantly reducing
```

Output:

```
3/python.exe c:/Users/shiva/OneDrive/Do
Basic method: Is 29 prime? True
Optimized method: Is 29 prime? True
```

Task Description #9 (Transparency in Recursive Algorithms)

Objective: Use AI to generate a recursive function to calculate

Fibonacci numbers.

Instructions:

1. Ask AI to add clear comments explaining recursion.

2. Ask AI to explain base cases and recursive calls.

Expected Output:

• Well-commented recursive code.

• Clear explanation of how recursion works.

• Verification that explanation matches actual execution.

Code:

```python
#generate a python code for calculating fibonacci numbers using both a basic recursi
#add clear comments explaining recursion.
#explain base cases and recursive calls.

# Basic recursive method to calculate Fibonacci numbers
def fibonacci_basic(n):
    # Base cases: the first two Fibonacci numbers
    if n == 0:
        return 0
    elif n == 1:
        return 1
    # Recursive case: sum of the two preceding Fibonacci numbers
    else:
        return fibonacci_basic(n - 1) + fibonacci_basic(n - 2)
# Example usage of the basic method
n = 10
print(f"Basic method: Fibonacci number at position {n} is {fibonacci_basic(n)}")
# Explanation of recursion:
# In the basic recursive method, we define two base cases:
# 1. If n is 0, the Fibonacci number is 0.
# 2. If n is 1, the Fibonacci number is 1.
# For any other value of n, the function calls itself twice:
# once with (n-1) and once with (n-2), summing the results to get the Fibonacci numb
# This leads to an exponential number of calls, making it inefficient for larger n.
```

Output:

```
Basic method: Fibonacci number at position 10 is 55
PS C:\Users\shiva\OneDrive\Documents\python>
```

Task Description #10 (Transparency in Error Handling)

Task: Use AI to generate a Python program that reads a file and processes data.

Prompt:

“Generate code with proper error handling and clear explanations for each exception.”

Expected Output:

• Code with meaningful exception handling.

• Clear comments explaining each error scenario.

• Validation that explanations align with runtime behaviour.

Code:

```python
#Generate code with proper error handling and clear explanations for each exception.
# error.py # Custom exception for invalid input
class InvalidInputError(Exception):
    """Exception raised for invalid inputs."""
    pass
# Function to divide two numbers with error handling
def divide_numbers(num1, num2):
    """Divides num1 by num2 with error handling for invalid inputs and division by zero."""
    try:
        # Check if inputs are numbers
        if not isinstance(num1, (int, float)) or not isinstance(num2, (int, float)):
            raise InvalidInputError("Both inputs must be numbers.")
        # Check for division by zero
        if num2 == 0:
            raise ZeroDivisionError("Cannot divide by zero.")
        # Perform division
        result = num1 / num2
        return result
    except InvalidInputError as e:
        print(f"InvalidInputError: {e}")
    except ZeroDivisionError as e:
        print(f"ZeroDivisionError: {e}")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
# Example usage
print(divide_numbers(10, 2))  # Valid input
print(divide_numbers(10, 0))  # Division by zero
print(divide_numbers(10, 'a'))  # Invalid input
print(divide_numbers('x', 2))  # Invalid input
print(divide_numbers(10, 5))  # Another valid input
# Explanation of error handling:
# 1. We define a custom exception InvalidInputError to handle cases where the inputs are not numbers.
# 2. In the divide_numbers function, we check the types of the inputs and raise InvalidInputError if they a
# 3. We also check for division by zero and raise ZeroDivisionError if num2 is zero.
# 4. The try-except blocks catch these specific exceptions and print appropriate error messages.
# 5. A general exception handler is included to catch any unexpected errors that may occur.
```

Output:

```
ZeroDivisionError: Cannot divide by zero.
None
InvalidInputError: Both inputs must be numbers.
None
InvalidInputError: Both inputs must be numbers.
None
2.0
```