AIAC - Lab Assignment : 5.1 & 6 (Classes & Responsible AI)

Name : Nampally Nithin Varma

Enrollment No : 2303A52011

Batch : 31

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:

```
class Employee:
    def __init__(self, empid, empname,designation,basic_salar
y,exp):
        self.empid = empid
        self.empname = empname
        self.designation = designation
```

```
        self.basic_salary = basic_salary
        self.exp = exp
    def calculate_salary(self):
        if self.exp > 10:
            allowance = 0.2 * self.basic_salary
        elif 5<= self.exp <= 10:
            allowance = 0.1 * self.basic_salary
        elif self.exp < 5:
            allowance = 0.05 * self.basic_salary
        total_salary = self.basic_salary + allowance
        return total_salary
    #example usage
emp1 = Employee(101, "Alice", "Manager", 50000, 12 )
print(f"Total salary for {emp1.empname}: {emp1.calculate_sala
ry()}")
```

Output:

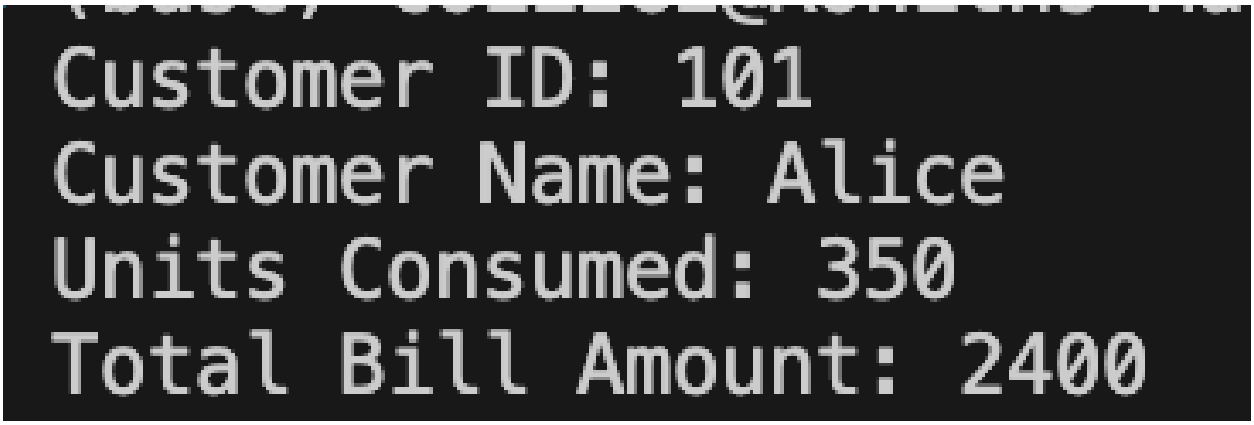
Task 2:
Electricity Bill Calculation- Create Python code that defines a class named `ElectricityBill` with attributes: `customer_id`, `name`, and `units_consumed`. Implement a method `display_details()` to print customer details, and a method `calculate_bill()` where:

- 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:

```
class ElectricityBill:
    def calculate_bill(self, customer_id, name, units_consume
d, rate_per_unit):
        self.customer_id = customer_id
        self.name = name
        self.units_consumed = units_consumed
        self.rate_per_unit = rate_per_unit

        if self.units_consumed <= 100:
            bill_amount = self.units_consumed * 5
        elif 101 <= self.units_consumed <= 300:
            bill_amount = (100 * 5) + ((self.units_consumed -
100) * 7)
        elif self.units_consumed > 300:
            bill_amount = (100 * 5) + (200 * 7) + ((self.unit
s_consumed - 300) * 10)
        return bill_amount
customer1= ElectricityBill()
customer1.calculate_bill(101, "Alice", 350, 0)
print(f"Customer ID: {customer1.customer_id}")
print(f"Customer Name: {customer1.name}")
print(f"Units Consumed: {customer1.units_consumed}")
print(f"Total Bill Amount: {customer1.calculate_bill(101, 'Al
ice', 350, 0)}")
```

Output:



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:

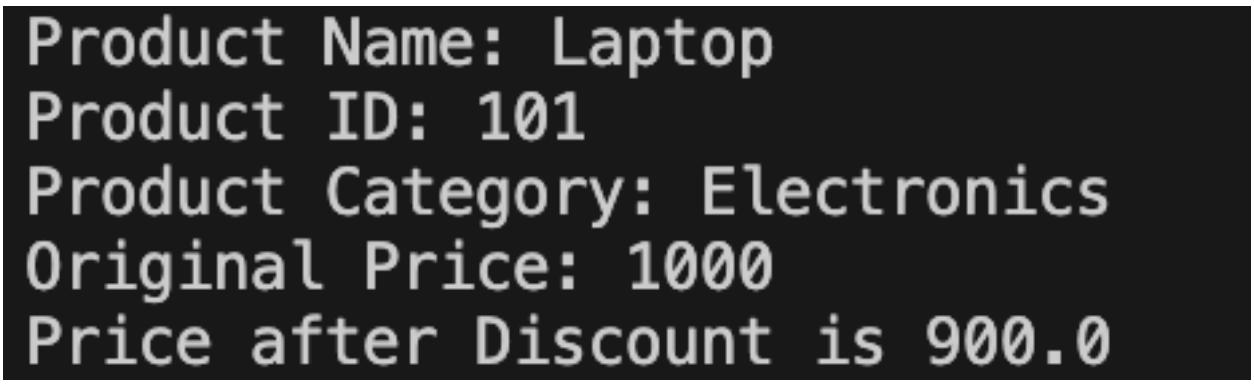
- Electronics → 10% discount
- Clothing → 15% discount
- Grocery → 5% discount
  Create at least one product object, display details, and print the final price after discount.

Code:

```
class Product:
    def __init__(self, product_id, product_name, price, categ
ory):
        self.product_id = product_id
        self.name = product_name
        self.price = price
        self.category = category

    def calculate_discount(self):
        if self.category == "Electronics":
```

```
            return self.price * 0.90   # 10% discount
        elif self.category == "Clothing":
            return self.price * 0.80   # 20% discount
        elif self.category == "Grocery":
            return self.price * 0.95   # 5% discount
        return self.price  # No discount


# Creating object (outside the class)
bill1 = Product(101, "Laptop", 1000, "Electronics")

discounted_price = bill1.calculate_discount()
print(f"Product Name: {bill1.name}")
print(f"Product ID: {bill1.product_id}")
print(f"Product Category: {bill1.category}")
print(f"Original Price: {bill1.price}")
print(f"Price after Discount is {discounted_price}")
```

Output:



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:

```
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_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}")

    def calculate_late_fee(self):
        if self.days_late <= 5:
            return self.days_late * 5
        elif self.days_late <= 10:
            return self.days_late * 7
        else:
            return self.days_late * 10


# Creating object (outside the class)
book1 = LibraryBook(201, "Python Programming", "Guido van Ros
sum", "Alice", 8)
```

```
# Displaying details
book1.display_details()

# Calculating and printing late fee
late_fee = book1.calculate_late_fee()
print(f"Late Fee: ₹{late_fee}")
```

Output:

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:

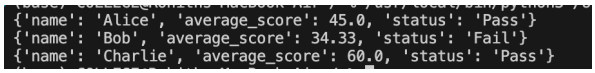
- Calculate the average score for each student
- Determine pass/fail status (pass ≥ 40)
- Return a summary report as a list of dictionaries
  Use Copilot suggestions as you build the function and format the output.

Code:

```
def student_report(student_data):
    report = []
```

```
    for name, marks in student_data.items():
        # Handle marks given as list or dictionary
        if isinstance(marks, dict):
            scores = marks.values()
        else:
            scores = marks

        average = sum(scores) / len(scores)
        status = "Pass" if average >= 40 else "Fail"

        report.append({
            "name": name,
            "average_score": round(average, 2),
            "status": status
        })

    return report
# Example usage
students = {
    "Alice": [45, 50, 40],
    "Bob": [30, 35, 38],
    "Charlie": {"Math": 60, "Science": 55, "English": 65}
}

summary = student_report(students)

for student in summary:
    print(student)
```

Output:



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:

```
class TaxiRide:
    def __init__(self, ride_id, driver_name, distance_km, wai
ting_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_details(self):
        print(f"Ride ID: {self.ride_id}")
        print(f"Driver Name: {self.driver_name}")
        print(f"Distance Travelled: {self.distance_km} km")
        print(f"Waiting Time: {self.waiting_time_min} minute
s")

    def calculate_fare(self):
        fare = 0
```

```
        # Distance fare calculation
        if self.distance_km <= 10:
            fare += self.distance_km * 15
        elif self.distance_km <= 30:
            fare += 10 * 15
            fare += (self.distance_km - 10) * 12
        else:
            fare += 10 * 15
            fare += 20 * 12
            fare += (self.distance_km - 30) * 10

        # Waiting charge
        fare += self.waiting_time_min * 2

        return fare


# Creating object (outside the class)
ride1 = TaxiRide(101, "Rahul", 35, 10)

# Displaying details
ride1.display_details()

# Calculating and printing total fare
total_fare = ride1.calculate_fare()
print(f"Total Fare: ₹{total_fare}")
```

Output:

```
Ride ID: 101
Driver Name: Rahul
Distance Travelled: 35 km
Waiting Time: 10 minutes
Total Fare: ₹460
```

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:

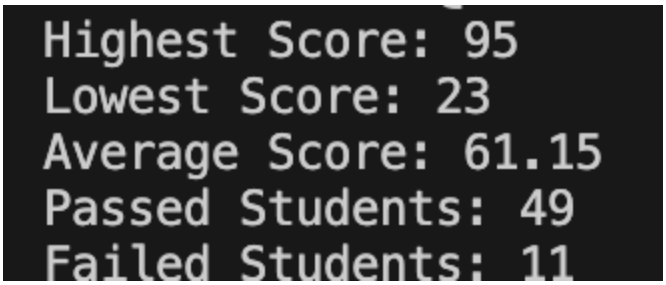
- Highest score in the class
- Lowest score in the class
- Class average score
- Number of students passed (score ≥ 40)
- Number of students failed (score < 40)
  Allow Copilot to assist with aggregations and logic

Code:

```
def statistics_subject(scores_list):
    if len(scores_list) != 60:
        print("Error: The list must contain exactly 60 studen
t scores.")
        return None

    highest_score = max(scores_list)
    lowest_score = min(scores_list)
    average_score = sum(scores_list) / len(scores_list)

    passed_students = sum(1 for score in scores_list if score
>= 40)
    failed_students = sum(1 for score in scores_list if score
```

```
< 40)

    return {
        "Highest Score": highest_score,
        "Lowest Score": lowest_score,
        "Average Score": average_score,
        "Passed Students": passed_students,
        "Failed Students": failed_students
    }


# Example usage
scores = [
    45, 67, 89, 34, 56, 78, 90, 23, 41, 55,
    62, 38, 49, 71, 84, 29, 65, 77, 58, 40,
    69, 73, 81, 92, 36, 47, 59, 68, 75, 88,
    33, 44, 52, 61, 70, 79, 85, 91, 28, 39,
    46, 54, 63, 72, 80, 87, 93, 31, 42, 50,
    57, 66, 74, 82, 86, 95, 27, 35, 48, 60
]

result = statistics_subject(scores)

if result:
    for key, value in result.items():
        print(f"{key}: {value}")
```

Ouput:



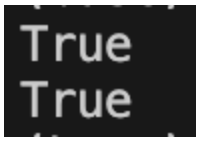
Task Description #8 (Transparency in Algorithm Optimization)
Task: Use AI to generate two solutions for checking prime numbers:
• Naive approach(basic)
• Optimized approach
Prompt:
“Generate Python code for two prime-checking methods and explain how the optimized version improves performance.”
Expected Output:
• Code for both methods.
• Transparent explanation of time complexity.
• Comparison highlighting efficiency improvements.

Code:

```
#Generate Python code for two prime-checking methods and expl
ain how the optimized version improves performance
def is_prime_basic(n):
    """Check if a number is prime using a basic method."""
    if n <= 1:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
def is_prime_optimized(n):
    """Check if a number is prime using an optimized metho
d."""
    if n <= 1:
        return False
```

```
    if n <= 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True
# Explanation of optimization:
# The basic method checks for factors from 2 to n-1, which re
sults in O(n) time complexity.
# The optimized method reduces the number of checks significa
ntly:
# 1. It first eliminates even numbers and multiples of 3 righ
t away.
# 2. It only checks for factors up to the square root of n, s
ince if n is divisible by any number larger than its square r
oot, the corresponding factor must be smaller than the square
root.
# 3. It checks only numbers of the form 6k ± 1, as all prime
numbers greater than 3 can be expressed in this form.
# These optimizations reduce the time complexity to O(√n), ma
king the optimized method much faster for larger numbers.

#example usage
print(is_prime_basic(17))  # Output: True
print(is_prime_optimized(17))  # Output: True
```

Output:



Task Description #9 (Transparency in Recursive Algorithms)
Objective: Use AI to generate a recursive function to calculate Fibonacci numbers.
Instructions:

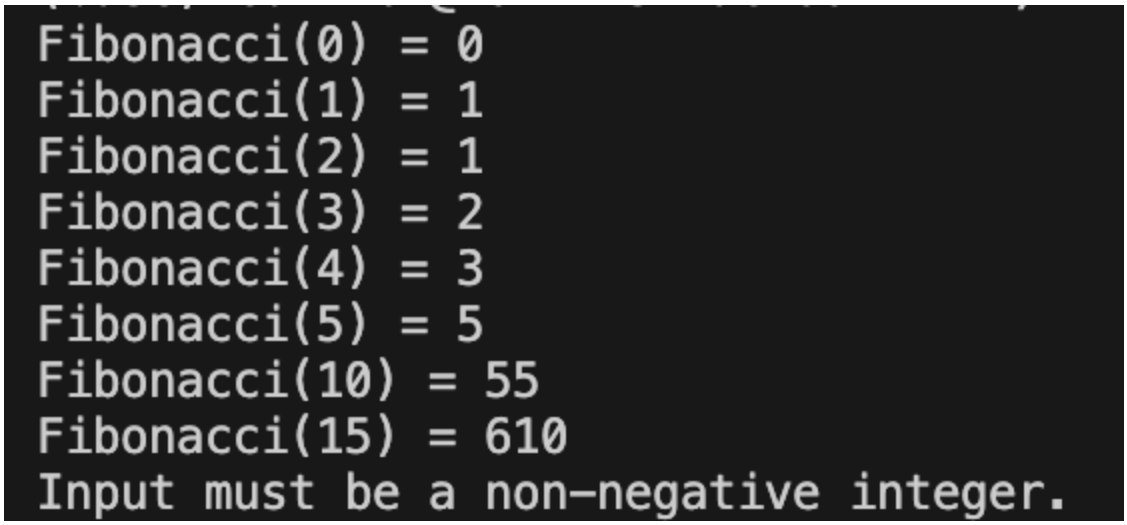
1. Ask AI to add clear comments explaining recursion.
2. Ask AI to explain base cases and recursive calls.
   Expected Output:
   - Well-commented recursive code.
   - Clear explanation of how recursion works.
   - Verification that explanation matches actual execution.

Code:

```
#generate well commented code for fibonacci series using recu
rsion and try to generate proper justification to handle test
cases also
def fibonacci(n):
    """
    This function returns the nth Fibonacci number using recu
rsion.

    Parameters:
    n (int): The position in the Fibonacci series to retriev
e. Must be a non-negative integer.

    Returns:
    int: The nth Fibonacci number.
```

```
    Raises:
    ValueError: If n is a negative integer.
    """

    # Check for invalid input
    if n < 0:
        raise ValueError("Input must be a non-negative intege
r.")

    # Base case: the first two Fibonacci numbers
    if n == 0:
        return 0
    elif n == 1:
        return 1

    # Recursive case: sum of the two preceding Fibonacci numb
ers
    return fibonacci(n - 1) + fibonacci(n - 2)
# Test cases to validate the function
if __name__ == "__main__":
    test_cases = [0, 1, 2, 3, 4, 5, 10, 15]

    for case in test_cases:
        print(f"Fibonacci({case}) = {fibonacci(case)}")

    # Testing invalid input
    try:
        print(fibonacci(-1))
    except ValueError as e:
        print(e)  # Expected output: Input must be a non-nega
tive integer.
```

Output:



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 behavior.

Code:

```
#Generate code with proper error handling and clear explanati
ons for each exception
class AIACError(Exception):
    """Base class for exceptions in AIAC module."""
    pass
class InvalidInputError(AIACError):
    """Exception raised for invalid inputs."""
```

```
    def __init__(self, message="The input provided is invali
d."):
        self.message = message
        super().__init__(self.message)
class CalculationError(AIACError):
    """Exception raised for errors during calculations."""
    def __init__(self, message="An error occurred during calc
ulation."):
        self.message = message
        super().__init__(self.message)
class DataNotFoundError(AIACError):
    """Exception raised when required data is not found."""
    def __init__(self, message="Required data not found."):
        self.message = message
        super().__init__(self.message)
# Example function demonstrating the use of custom exceptions
def perform_calculation(data):
    if data is None:
        raise DataNotFoundError("No data provided for calcula
tion.")
    if not isinstance(data, (int, float)):
        raise InvalidInputError("Input must be a number.")
    try:
        result = 100 / data  # Example calculation
    except ZeroDivisionError:
        raise CalculationError("Division by zero is not allow
ed.")
    return result
# Example usage
try:
    print(perform_calculation(0))
except AIACError as e:
    print(f"Error: {e}")
```

Output: