AI ASSISTED CODING
ASSIGNMENT - 5.1 & 6.1

Name:B.SiriChandana
HTNo:2303A52004
Batch:31

Question:

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_salary,exp)
:
        self.empid=empid
        self.empname=empname
        self.designation=designation
        self.basic_salary=basic_salary
        self.exp=exp
    def display_deteils(self):
        print("Employee ID:",self.empid)
        print("Employee Name:",self.empname)
        print("Designation:",self.designation)
        print("Basic Salary:",self.basic_salary)
        print("Experience:",self.exp)
```

```python
    def calculate_allowance(self):
        if self.exp>10:
            allowance=0.2*self.basic_salary
        elif 5<=self.exp<=10:
            allowance=0.1*self.basic_salary
        else:
            allowance=0.05*self.basic_salary
        total_salary = self.basic_salary+allowance
        print("Allowance is:",allowance)
        print(f"Total Salary is:",total_salary)
# Example usage
empobj=Employee(101,"John Doe","Manager",50000,12)
empobj.display_deteils()
empobj.calculate_allowance()
```

Output:

```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\employee.py"
Employee ID: 101
Employee Name: John Doe
Designation: Manager
Basic Salary: 50000
Experience: 12
Allowance is: 10000.0
Total Salary is: 60000.0
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
```

Justification:

This program uses a class to neatly store employee details and display them using object-oriented principles.
Allowance is calculated correctly using conditional statements based on the employee's experience.

Question:

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 __init__(self,customer_id,name,units_consumed):
        self.customer_id=customer_id
        self.name=name
        self.units_consumed=units_consumed
    def display_details(self):
        print("Customer ID:", self.customer_id)
        print("Customer Name:", self.name)
        print("Units Consumed:", self.units_consumed)
    def calculate_bill(self):
        if self.units_consumed <= 100:
            rate_per_unit = 5
        elif 101 <= self.units_consumed <= 300:
            rate_per_unit = 7
        else:
            rate_per_unit = 10
        total_bill = self.units_consumed * rate_per_unit
        print("Total Electricity Bill for", self.name,
"is:", total_bill)

# Example usage
billobj = ElectricityBill(201, "Alice Smith", 250)
billobj.display_details()
billobj.calculate_bill()
```

Output:

```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\tempCodeRunnerFile.py"
Customer ID: 201
Customer Name: Alice Smith
Units Consumed: 250
Total Electricity Bill for Alice Smith is: 1750
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
```

Justification:

The class structure helps organize customer details and electricity usage efficiently.
Conditional logic ensures the bill amount is calculated accurately based on unit slabs.

Question:

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,category):
        self.product_id=product_id
        self.product_name=product_name
        self.price=price
        self.category=category
    def display_product_info(self):
        print("Product ID:", self.product_id)
        print("Product Name:", self.product_name)
        print("Price:", self.price)
        print("Category:", self.category)
    def calculate_discount(self):
        if self.category.lower() == "electronics":
```

```python
            discount_rate = 0.10  # 10% discount
        elif self.category.lower() == "clothing":
            discount_rate = 0.15  # 15% discount
        else:
            discount_rate = 0.05  # 5% discount
        discount_amount = self.price * discount_rate
        discounted_price = self.price - discount_amount
        print("Discounted Price for", self.product_name,
"is:", discounted_price)
# Example usage
productobj = Product(101, "Smartphone", 500,
"Electronics")
productobj.display_product_info()
productobj.calculate_discount()
```

Output:

```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\Product.py"
Product Name: Smartphone
Price: 500
Category: Electronics
Discounted Price for Smartphone is: 450.0
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
```

Justification:

The program applies object-oriented concepts to manage product information clearly. Discounts are correctly applied using category-based conditions to compute the final price.

Question:

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_book_info(self):
        print("Book ID:", self.book_id)
        print("Title:", self.title)
        print("Author:", self.author)
        print("Borrower:", self.borrower)
        print("Days Late:", self.days_late)
    def calculate_late_fee(self):
        if self.days_late <= 5:
            late_fee_per_day = 5
        elif 6 <= self.days_late <= 10:
            late_fee_per_day = 7
        else:
            late_fee_per_day = 10
        total_late_fee = self.days_late *
late_fee_per_day
        print("Total Late Fee for", self.title, "is:",
total_late_fee)
# Example usage
bookobj = LibraryBook(301, "The Great Gatsby", "F. Scott
Fitzgerald", "John Doe", 8)
bookobj.display_book_info()
bookobj.calculate_late_fee()
```

Output:

```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\tempCodeRunnerFile.py"
Book ID: 301
Title: The Great Gatsby
Author: F. Scott Fitzgerald
Borrower: John Doe
Days Late: 8
Total Late Fee for The Great Gatsby is: 56
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
```

Justification:

This code models library book details using a class for better data organization.
Late fees are calculated accurately using conditional checks on the number of late days.

Question:
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.

Prompt:

```
#Write a Python function student_report(student_data)
that accepts a dictionary of student names mapped to a
list of marks. For each student, calculate the average
score, determine pass/fail (pass ≥ 40), and return a
summary report as a list of dictionaries. Include clear
comments and sample output formatting.
```

Code:

```
from numpy import average


def student_report(student_data):
    """
```

```
    Generate a summary report for students based on their
marks.

    Parameters:
    student_data (dict): A dictionary where keys are
student names (str) and values are lists of marks (list
of int).

    Returns:
    list: A list of dictionaries, each containing the
student's name, average score, and pass/fail status.
    """

    report = []

    for student, marks in student_data.items():
        # Input validation: Check if marks is a list
        if not isinstance(marks, list) or not
all(isinstance(mark, (int, float)) for mark in marks):
            raise ValueError(f"Marks for {student} must
be a list of numbers.")

        # Calculate average score
        average_score = sum(marks) / len(marks) if marks
else 0

        # Determine pass/fail status
        status = "Pass" if average_score >= 40 else
"Fail"
        # Append the student's report to the list
        report.append({
            "student": student,
```

```
            "average_score": average_score,
            "status": status
        })

    return report
# Example usage:
if __name__ == "__main__":
    # Sample student data for demonstration
    sample_student_data = {
        "Alice": [45, 78, 89],
        "Bob": [34, 23, 40],
        "Charlie": [90, 92, 85],
        "David": [30, 25, 20]
    }

    # Call the student_report function and print the
results
    report = student_report(sample_student_data)
    for student_info in report:
        print(f"Student: {student_info['student']},
Average Score: {student_info['average_score']:.2f},
Status: {student_info['status']}")
```

Output:

```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\tempCodeRunnerFile.py"
Student: Alice, Average Score: 70.67, Status: Pass
Student: Bob, Average Score: 32.33, Status: Fail
Student: Charlie, Average Score: 89.00, Status: Pass
Student: David, Average Score: 25.00, Status: Fail
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
```

Justification:

The function processes student data efficiently using dictionaries and loops.
It correctly computes averages and determines pass/fail status based on defined criteria.

Question:
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,waiting_tim
e_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_info(self):
        print("Ride ID:", self.ride_id)
        print("Driver Name:", self.driver_name)
        print("Distance (km):", self.distance_km)
        print("Waiting Time (min):",
self.waiting_time_min)
    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 * waiting_charge_per_min)
        print("Total Fare for the ride with", 
self.driver_name, "is:", total_fare)
# Example usage
rideobj = TaxiRide(401, "Bob Johnson", 18, 5)
rideobj.display_ride_info()
rideobj.calculate_fare()
```

Output:

```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\TaxiRide.py"
Ride ID: 401
Driver Name: Bob Johnson
Distance (km): 18
Waiting Time (min): 5
Total Fare for the ride with Bob Johnson is: 226
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
```

Justification:

The program uses a class to represent taxi ride details in a structured way.
Fare calculation follows distance slabs and includes waiting charges accurately.

Question:

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

Prompt:

```
# Write a Python function statistics_subject(scores_list) 
that accepts a list of 60 student scores and returns:
```

```
highest score, lowest score, class average, number passed
(≥40), number failed (<40). Include clear comments and
input validation.
```

Code:

```python
def statistics_subject(scores_list):
    """
    Calculate statistics for a list of student scores.

    Parameters:
    scores_list (list): A list of 60 student scores
(integers between 0 and 100).

    Returns:
    dict: A dictionary containing the highest score,
lowest score, class average,
          number of students passed (score >= 40), and
number of students failed (score < 40).
    """

    # Input validation: Check if the input is a list
    if not isinstance(scores_list, list):
        raise ValueError("Input must be a list.")

    # Input validation: Check if the list contains
exactly 60 scores
    if len(scores_list) != 60:
        raise ValueError("The list must contain exactly
60 scores.")

    # Input validation: Check if all scores are integers
between 0 and 100
```

```
    for score in scores_list:
        if not isinstance(score, int) or score < 0 or
score > 100:
            raise ValueError("All scores must be integers
between 0 and 100.")

    # Calculate highest score
    highest_score = max(scores_list)

    # Calculate lowest score
    lowest_score = min(scores_list)

    # Calculate class average
    class_average = sum(scores_list) / len(scores_list)

    # Calculate number of students passed (score >= 40)
    num_passed = sum(1 for score in scores_list if score
>= 40)

    # Calculate number of students failed (score < 40)
    num_failed = sum(1 for score in scores_list if score
< 40)

    # Return the statistics as a dictionary
    return {
        "highest_score": highest_score,
        "lowest_score": lowest_score,
        "class_average": class_average,
        "num_passed": num_passed,
        "num_failed": num_failed
    }
# Example usage:
```

```python
if __name__ == "__main__":
    # Sample list of 60 student scores for demonstration
    sample_scores = [55, 78, 90, 34, 67, 88, 45, 23, 76,
89,
                     92, 41, 39, 60, 73, 84, 95, 100, 12,
56,
                     43, 38, 77, 81, 69, 50, 29, 66, 72,
85,
                     91, 47, 30, 64, 58, 79, 82, 94, 37,
40,
                     53, 68, 74, 86, 99, 22, 15, 48, 61,
71,
                     80, 87, 93, 36, 44, 54, 65, 70, 83,
57]

    # Call the statistics_subject function and print the
results
    stats = statistics_subject(sample_scores)
    print("Statistics for the class:")
    print(f"Highest Score: {stats['highest_score']}")
    print(f"Lowest Score: {stats['lowest_score']}")
    print(f"Class Average: {stats['class_average']:.2f}")
    print(f"Number Passed: {stats['num_passed']}")
    print(f"Number Failed: {stats['num_failed']}")
```

Output:

```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\scores.py"
Statistics for the class:
Highest Score: 100
Lowest Score: 12
Class Average: 63.25
Number Passed: 49
Number Failed: 11
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
```

Justification:

This function validates input data and computes class statistics using built-in operations.
It accurately identifies pass/fail counts and calculates highest, lowest, and average scores.

Question:

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.

Prompt:

```
#Generate Python code for two prime-checking methods: a naive approach and an optimized approach. Include clear comments, explain the time complexity of each, and compare how the optimized version improves performance.
```

Code:

```
# Naive approach to check if a number is prime
def is_prime_naive(n):
    """
    Check if a number is prime using the naive approach.

    Parameters:
    n (int): The number to be checked for primality.

    Returns:
    bool: True if n is prime, False otherwise.

    Time Complexity:
    O(n) - In the worst case, we check all numbers from 2 to n-1.
```

```python
    """
    if n <= 1:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
# Optimized approach to check if a number is prime
def is_prime_optimized(n):
    """
    Check if a number is prime using an optimized
approach.

    Parameters:
    n (int): The number to be checked for primality.

    Returns:
    bool: True if n is prime, False otherwise.

    Time Complexity:
    O(√n) - We only check for factors up to the square
root of n.
    """
    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
# Example usage:
if __name__ == "__main__":
    number = int(input("Enter a positive integer to check
for primality: "))

    # Check using naive approach
    if is_prime_naive(number):
        print(f"{number} is prime (naive approach).")
    else:
        print(f"{number} is not prime (naive approach).")

    # Check using optimized approach
    if is_prime_optimized(number):
        print(f"{number} is prime (optimized approach).")
    else:
        print(f"{number} is not prime (optimized
approach).")

    # Print which method is faster and why
    print(
        "optimized approach is faster for larger numbers:
O(√n) checks vs O(n) in the naive approach."
    )

    """
    Comparison:
    The naive approach has a time complexity of O(n),
which means that as the input number increases, the time
```

```
taken to check for primality increases linearly. This can
be inefficient for large numbers.
    The optimized approach reduces the time complexity to
O(√n) by eliminating even numbers and checking only up to
the square root of n. This significantly improves
performance, especially for larger numbers, making it
much faster and more efficient.
    """
```

Output:

```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\prime.py"
Enter a positive integer to check for primality: 5
5 is prime (naive approach).
5 is prime (optimized approach).
Optimized approach is faster for larger numbers: O(√n) checks vs O(n) in the naive approach.
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
```

Justification:

The naive method checks all possible divisors, making it simple but inefficient.
The optimized method improves performance by reducing checks to √n, making it faster.

Question:
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.

Prompt:

```
#generate well commented code for fibonacci series using
recursion in python
```

Code:

```
def fibonacci(n):
    """
```

```
    Calculate the nth Fibonacci number using recursion.

    Parameters:
    n (int): The position in the Fibonacci sequence to
retrieve.

    Returns:
    int: The nth Fibonacci number.
    """
    # Base case: the first two Fibonacci numbers are 0
and 1
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        # Recursive case: sum of the two preceding
Fibonacci numbers
        return fibonacci(n - 1) + fibonacci(n - 2)
# Example usage:
if __name__ == "__main__":
    num=int(input("Enter a positive integer: "))
    print(f"The {num}th Fibonacci number is:
{fibonacci(num)}")
```

Output:

```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\fibo.py"
  Enter a positive integer: 5
  The 5th Fibonacci number is: 5
PS C:\Users\bogas\OneDrive\Desktop\AIAC>
```

Justification:

The recursive function clearly demonstrates how Fibonacci numbers are generated.
Base cases prevent infinite recursion, while recursive calls build the solution correctly.

Question:
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.

Prompt:
```
# Generate a well-commented Python program that reads a
user-specified file, processes data, and includes proper
error handling with clear explanations for each exception
```

Code:
```
def read_file(file_path):
    """
    Reads the contents of a file and processes the data.

    Parameters:
    file_path (str): The path to the file to be read.

    Returns:
    str: The contents of the file if successful, or an
error message if an exception occurs.
    """
    try:
        # Attempt to open the file in read mode
        with open(file_path, 'r') as file:
            # Read the contents of the file
            data = file.read()
            # Process the data (for demonstration, we
will just return it)
```

```
            return data
    except FileNotFoundError:
        # This exception is raised when the specified file does not exist
        return f"Error: The file '{file_path}' was not found."
    except PermissionError:
        # This exception is raised when there are insufficient permissions to read the file
        return f"Error: You do not have permission to read the file '{file_path}'."
    except Exception as e:
        # Catch any other exceptions that may occur and return a generic error message
        return f"An unexpected error occurred: {str(e)}"
# Example usage:
if __name__ == "__main__":
    # Prompt the user to enter the file path
    file_path = input("Enter the path of the file to read: ")
    # Call the read_file function and print the result
    result = read_file(file_path)
    print(result)
```

Output:

```
PS C:\Users\bogas\OneDrive\Desktop\AIAC> python -u "c:\Users\bogas\OneDrive\Desktop\AIAC\tempCodeRunnerFile.py"
Enter the path of the file to read: sample.txt
Hello, World!
This is a sample text file.
File operations in Python are easy!

PS C:\Users\bogas\OneDrive\Desktop\AIAC>
```

Justification:

The program uses try–except blocks to handle file-related errors safely.
Each exception is clearly explained, ensuring reliable execution without crashing.