Assignment-10.1

2303A51304

Batch-05

Task-01: Use AI to identify and fix syntax and logic errors in a faulty

Prompt:

Identify all syntax errors that prevent the code from running.

Identify all logic errors that produce incorrect results.

Rewrite the code so that it is fully correct, runnable, and follows proper Python indentation and naming conventions.

Faulty Code:

```
# Calculate average score of a student

def calc_average(marks):

total = 0

for m in marks:

total += m

average = total / len(marks)

return avrage

marks = [85, 90, 78, 92]

print("Average Score is ", calc_average(marks)
```

Expected Output:Corrected Python code that runs without errors

Code:

```
marks = [85, 90, 78, 92]
print("Average Score is ", calc_average(marks)
Expected Output:
Corrected Python code that runs without errors'''
# ------------------ Corrected Code ------------------
def calc_average(marks):
    """
    Calculate the average score of a student.
    Args:
    marks (list): A list of marks obtained by the student.
    Returns:
    float: The average score.
    """
    total = 0
    for m in marks:
        total += m
    average = total / len(marks)
    return average  # Fixed typo from 'avrage' to 'average'
# Sample marks for a student
marks = [85, 90, 78, 92]
# Print the average score
print("Average Score is ", calc_average(marks))
```

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  POSTMAN CONSOLE

Average Score is  86.25
PS C:\Users\pooji\OneDrive\Desktop\AI_Assistance_coding>
```

Explanation:

The function calc_average takes a list of student marks as input.A variable total is initialized to store the sum of all marks.

A for loop iterates through the list and adds each mark to total.

The average is calculated by dividing the total marks by the number of subjects using len(marks).

The corrected variable name average is returned (fixing the earlier typo avrage). The function is called with sample marks, and the final average score is printed correctly.

Task-02: Use AI to refactor Python code to follow PEP 8 style guidelines.

Prompt:

Follow PEP 8 style

Sample Input Code:

def area_of_rect(L,B) : return L*B

print(area_of_rect(10,20))

Give Well-formatted PEP 8-compliant corrected Python code.

Code:

```
'''Follow PEP 8 style
Sample Input Code:
def area_of_rect(L,B) : return L*B
print(area_of_rect(10,20))
Expected Output:
Well-formatted PEP 8-compliant Python code.'''
# ------------------ PEP 8 Compliant Code ------------------
def area_of_rectangle(length, breadth):
    """
    Calculate the area of a rectangle.
    Args:
    length (float): The length of the rectangle.
    breadth (float): The breadth of the rectangle.
    Returns:
    float: The area of the rectangle.
    """
    return length * breadth
# Sample input
print(area_of_rectangle(10, 20))
```

Output:

```
rs/pooji/OneDrive/Desktop/AI_Assistance_coding/Assignment-10.1/task02.
200
PS C:\Users\pooji\OneDrive\Desktop\AI_Assistance_coding>
```

Explanation:

The function name area_of_rectangle follows snake_case, as required by PEP 8.

Parameter names length and breadth are descriptive instead of single-letter variables.

Proper indentation and line spacing improve readability and maintainability.

A docstring is added to explain the purpose, arguments, and return value of the function.

The calculation logic remains simple: multiplying length and breadth to get the area.

The function call is clearly separated from the definition, following standard Python style.

Task-03: Use AI to make code more readable without changing its logic

Prompt:

You are given a Python program that works correctly but is poorly readable due to unclear variable names,

lack of comments, and improper formatting.

Improve the readability of the code without changing its logic or output.

Replace unclear variable and function names with descriptive names.

Format the code clearly using proper indentation and spacing according to PEP 8 standards.

Do not modify the logic, calculations, or output of the program.

Input Code:

```
def c(x,y):
return x*y/100
a=200
b=15
print(c(a,b))
```

Output:

Readable, well-formatted Python code

Same functionality and output as the original program

Code:

```
Do not modify the logic, calculations, or output of the program.
Input Code:
def c(x,y):
return x*y/100
a=200
b=15
print(c(a,b))
Output:
Readable, well-formatted Python code
Same functionality and output as the original program'''
# ----------------- Improved Readability Code ------------------
def calculate_percentage(value, percentage):
    """
    Calculate the percentage of a given value.
    Args:
    value (float): The total value.
    percentage (float): The percentage to calculate.
    Returns:
    float: The calculated percentage of the value.
    """
    return value * percentage / 100
# Sample input values
total_value = 200
percentage_to_calculate = 15
# Print the calculated percentage
print(calculate_percentage(total_value, percentage_to_calculate))
```

Output:

```
/Desktop/AI_Assistance_coding/lab_test/task-03.py
/Desktop/AI_Assistance_coding/lab_test/task-03.py
30.0
PS C:\Users\pooji\OneDrive\Desktop\AI_Assistance_coding>
```

Explanation:

The function name calculate_percentage clearly describes what the function does, improving readability.

The parameters value and percentage replace unclear variable names x and y.

Proper indentation and spacing are applied to follow **PEP 8** formatting rules.

A docstring is added to explain the purpose, inputs, and output of the function. Variable names total_value and percentage_to_calculate make the code self-explanatory.

Task-04: Use AI to break repetitive or long code into reusable functions.

Prompt:

You are given a Python program that contains repetitive code statements.

Identify the repeated logic in the program.

Refactor the code by creating reusable functions to eliminate repetition.Ensure the refactored code is modular, readable, and follows PEP 8 standards.

Maintain the same functionality and output as the original program.

Do not change the logic or results—only improve structure and reusability.

Input Code:

```
students = ["Alice", "Bob", "Charlie"]

print("Welcome", students[0])
```

print("Welcome", students[1])

print("Welcome", students[2])

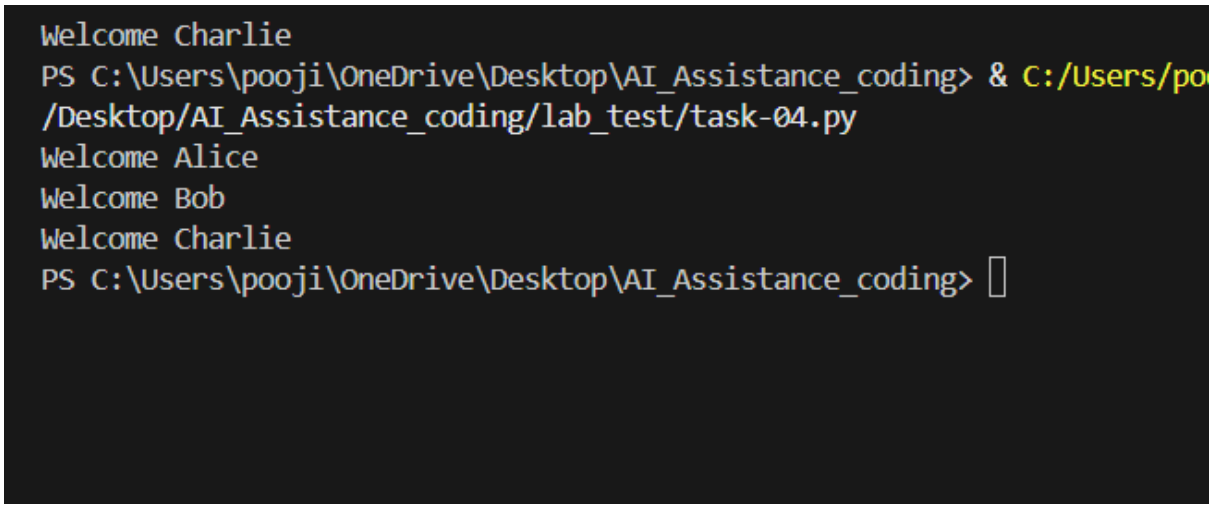
Expected Output:

Modular Python code using reusable functions. Same output as the original program

Code:

```
'''You are given a Python program that contains repetitive code statements.
Identify the repeated logic in the program.
Refactor the code by creating reusable functions to eliminate repetition.
Ensure the refactored code is modular, readable, and follows PEP 8 standards.
Maintain the same functionality and output as the original program.
Do not change the logic or results—only improve structure and reusability.
Input Code:
students = ["Alice", "Bob", "Charlie"]
print("Welcome", students[0])
print("Welcome", students[1])
print("Welcome", students[2])
Expected Output:
Modular Python code using reusable functions
Same output as the original program'''
# ------------------ Refactored Code with Reusable Function ------------------
def welcome_students(student_list):
    """
    Print a welcome message for each student in the list.
    Args:
    student_list (list): A list of student names.
    """
    for student in student_list:
        print("Welcome", student)
# Sample list of students
students = ["Alice", "Bob", "Charlie"]
# Call the function to welcome students
welcome_students(students)
```

Output:



Explanation:

The repeated print("Welcome", student) statements are replaced with a single reusable function.

The function welcome_students encapsulates the welcome logic, making the code modular.

A for loop is used to handle any number of students instead of hardcoding indices. Descriptive function and variable names improve readability and follow PEP 8 standards. A docstring explains the purpose and input of the function**.**

Task-05: Use AI to make the code run faster.

Prompt:

You are given a Python program that is functionally correct but inefficient in terms of performance.

Analyze the code and identify performance bottlenecks.

Optimize the code to run faster without changing its logic or output.

Use efficient Python constructs such as list comprehensions or vectorized operations where appropriate.

Ensure the optimized code is readable and follows PEP 8 standards.

Do not change the result or behavior of the program—only improve execution speed

Input Code:

Find squares of numbers

nums = [i for i in range(1,1000000)]

squares = []

for n in nums:

    squares.append(n**2)

print(len(squares))

Expected Output:

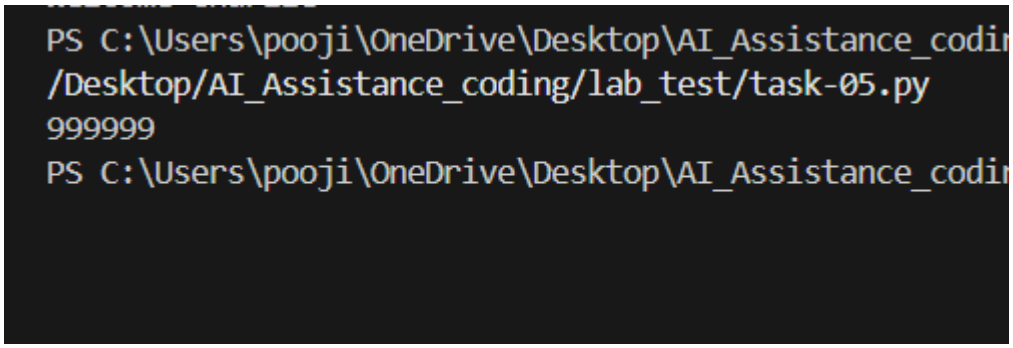
Optimized Python code using list comprehensions or faster constructs

Same output as the original program

Code:

```
for n in nums:
    squares.append(n**2)
print(len(squares))
Expected Output:
Optimized Python code using list comprehensions or faster constructs
Same output as the original program'''
# ------------------ Optimized Code Using List Comprehension ------------------
# Find squares of numbers using list comprehension for better performance
nums = [i for i in range(1, 1000000)]
squares = [n**2 for n in nums]  # Using list comprehension to calculate squares
print(len(squares))
```

Output:



Explanation:

The original code was slow because it used a manual loop with repeated append() calls.

List comprehension replaces the loop, which is faster and more memory-efficient in Python.

The calculation logic (n**2) and output remain exactly the same.

The code now performs the operation in a single, optimized expression. Proper spacing and comments improve readability while following PEP 8 standards

Task-06: Use AI to simplify overly complex logic.

Prompt:

You are given a Python function that produces correct results but uses overly complex and deeply nested conditional logic.

Analyze the existing logic and identify unnecessary nesting.

Simplify the code using cleaner control structures such as elif statements or a dictionary-based approach.

Ensure the simplified version is readable, maintainable, and follows PEP 8 standards.

Maintain the same logic and output as the original code.

Do not change the grading rules or return values—only improve clarity and structure.

Input Code:

```
def grade(score):
    if score >= 90:
        return "A"
    else:
        if score >= 80:
            return "B"
        else:
            if score >= 70:
                return "C"
            else:
                if score >= 60:
                    return "D"
                else:
                    return "F"
```

Expected Output:

Cleaner and simplified Python logic

Same grading output as the original program

Code:

```
                return "F"

Expected Output:
Cleaner and simplified Python logic
Same grading output as the original program'''
# ------------------ Simplified Code Using Elif Statements ------------------
def grade(score):
    """
    Determine the grade based on the score.
    Args:
    score (float): The score to evaluate.
    Returns:
    str: The grade corresponding to the score.
    """
    if score >= 90:
        return "A"
    elif score >= 80:  # Changed from nested if to elif
        return "B"
    elif score >= 70:  # Changed from nested if to elif
        return "C"
    elif score >= 60:  # Changed from nested if to elif
        return "D"
    else:
        return "F"  # No change needed here, as it's the final else case
# Sample input to test the function
print(grade(95))  # Expected output: "A"
print(grade(85))  # Expected output: "B"
print(grade(75))  # Expected output: "C"
print(grade(65))  # Expected output: "D"
print(grade(55))  # Expected output: "F"
```

Output:

```
999999
PS C:\Users\pooji\OneDrive\Desktop\AI_Assistance_coding> & C:/Users/pooji/AppData/Local/Python/py
/Desktop/AI_Assistance_coding/lab_test/task-06.py
A
B
C
D
F
PS C:\Users\pooji\OneDrive\Desktop\AI_Assistance_coding>
                                                    Ln 51, Col 1
```

Explanation:

The deeply nested if–else blocks are replaced with a flat if–elif–else structure. Using elif removes unnecessary nesting while preserving the same grading logic.

Each condition is checked in descending order, making the flow easy to follow.

A docstring is added to explain the function’s purpose, input, and output.

Readability and maintainability are improved without changing any grading rules.