AI Assisted Coding-10

Name : Tharun
HT NO : 2303A52386
Batch No : 40

Task Description #1 – Syntax and Logic Errors

Task: Use AI to identify and fix syntax and logic errors in a faulty Python script.

Prompt: > "Can you identify and fix the syntax and logic errors in this Python script? It's supposed to calculate the average of a list of marks, but it contains typos and syntax issues."

Code:

```
# Calculate average score of a student
def calc_average(marks):
    total = 0
    for m in marks:
        total += m
    average = total / len(marks)
    return average   # Fixed typo: 'avrage' to 'average'

marks = [85, 90, 78, 92]
# Fixed syntax error: added missing closing parenthesis
print("Average Score is ", calc_average(marks))
```

Output:

```
Average Score is  86.25
```

Explanation:

I found a couple of quick issues in the original code. First, there was a simple typo inside the function: it tried to return a variable named avrage instead of the correctly spelled average variable that we just calculated. Second, the print statement at the very bottom was missing its closing parenthesis. I fixed both of these, and now the code calculates and prints the average perfectly!

Task Description #2 – PEP 8 Compliance

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

Prompt: > "Please refactor this Python code to strictly follow PEP 8 style guidelines. Fix the spacing, indentation, and parameter naming conventions."

Code:

```
def area_of_rect(length, width):
    return length * width

print(area_of_rect(10, 20))
```

Output:

```
200
```

Explanation:

To make this code PEP 8 compliant, I expanded the single-line function into a standard block format by moving the return statement to a new line and adding proper indentation. I also added a blank line between the function definition and the function call, which PEP 8 recommends. Finally, I renamed the parameters from L, B to length, width. PEP 8 advises against using single letters like 'l' because it looks too much like the number '1', and using full words makes the code much easier to understand anyway!

Task Description #3 – Readability Enhancement

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

Prompt: > "Make this Python code more readable by replacing the single-letter variables with descriptive names, adding inline comments to explain what it does, and applying clear formatting. Do not change the underlying math logic."

Code:

```
def calculate_percentage(amount, percentage_rate):
    # Calculates the percentage of a given amount
    return amount * percentage_rate / 100

principal_amount = 200
rate_of_interest = 15

print(calculate_percentage(principal_amount, rate_of_interest))
```

Output:

```
30.0
```

Explanation:

The original code worked just fine, but it was really hard to guess what it was actually doing because it relied heavily on single letters like c, x, y, a, and b. I gave the function a descriptive name (calculate_percentage) and renamed the variables so that their purpose is obvious at a glance. I also spaced out the variables and added a helpful comment explaining what the function calculates. It's doing the exact same math, but now humans can easily read it.

Task Description #4 – Refactoring for Maintainability

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

Prompt: > "Refactor this repetitive Python script by breaking the print statements out into a reusable function that utilizes a loop to iterate over the list of students."

Code:

```
def welcome_students(student_list):
    for student in student_list:
        print("Welcome", student)

students = ["Alice", "Bob", "Charlie"]
welcome_students(students)
```

Output:

```
Welcome Alice
Welcome Bob
Welcome Charlie
```

Explanation:

Instead of hard-coding a print statement for every single index in the list, I created a reusable function called welcome_students. Inside this function, I used a for loop to go through the list one by one. This is a much better way to write the code because if you add twenty more students to the list later, you won't need to write twenty more print statements—the loop will handle everything automatically.

Task Description #5 – Performance Optimization

Task: Use AI to make the code run faster.

Prompt: > "Optimize this Python code to make it run faster and consume less memory. Please use list comprehensions instead of manually appending to a list inside a standard for-loop."

Code:

```
# Find squares of numbers using an optimized list comprehension
squares = [n**2 for n in range(1, 1000000)]
print(len(squares))
```

Output:

```
999999
```

Explanation:

The original code was doing a lot of unnecessary work. It first generated a massive list of numbers, created an empty list, and then used a slow for loop to append the squared numbers one by one. I combined all of those steps into a single line called a "list comprehension". Because list comprehensions are optimized in C under the hood of Python, this runs much faster and saves a lot of memory. It also looks much cleaner!

Task Description #6 – Complexity Reduction

Task: Use AI to simplify overly complex logic.

Prompt: > "Simplify this overly complex, deeply nested if-else logic for grading into a cleaner, flatter structure using elif statements."

Code:

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

# Testing the function
print(grade(75))
```

Output:

```
C
```

Explanation:

The original function suffered from what programmers call the "pyramid of doom"—deeply nested if and else statements that just keep pushing the code further to the right. It makes the code incredibly annoying to read and edit. I flattened the entire thing by replacing the nested blocks with elif (else if) statements. Now the logic reads naturally from top to bottom. Once it finds the first true condition, it returns the grade and exits the function instantly.