Assignment 10.1

Code Review and Quality: Using AI to Improve Code Quality and Readability

Task 1

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

Sample Input 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 # Typo here
marks = [85, 90, 78, 92]
print("Average Score is ", calc_average(marks)
```

Prompt:

You are given a faulty Python script with syntax and logic errors.

Tasks:

1. Identify all the syntax and logic errors in the given code.
2. Correct the code so that it runs successfully.
3. Provide explanations for each fix you made.

Faulty Python Code:

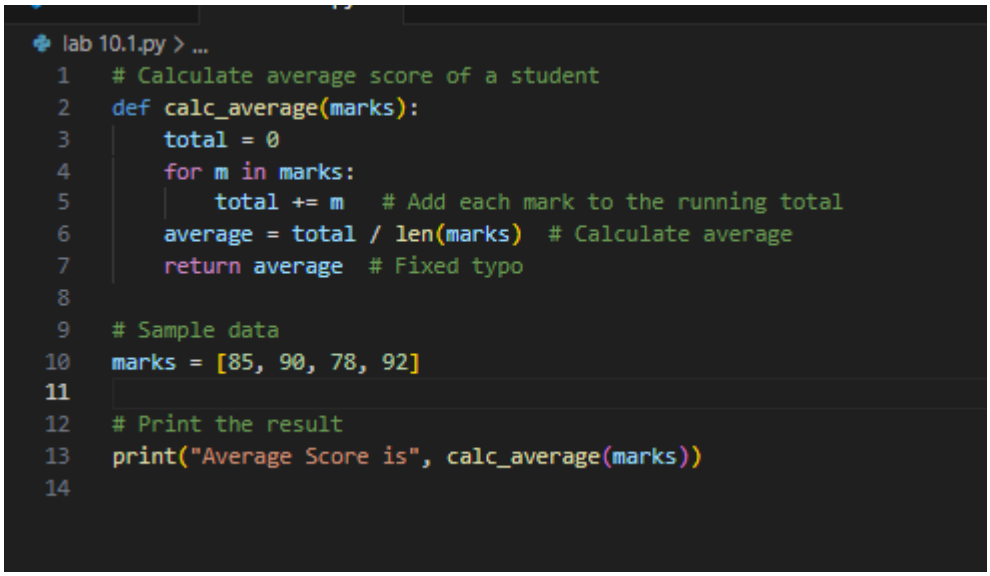
```
# Calculate average score of a student

def calc_average(marks):

total = 0

for m in marks:

total += m

average = total / len(marks)
```

return avrage # Typo here

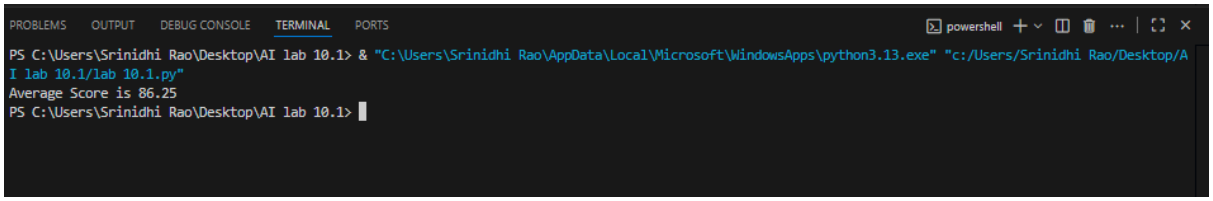
marks = [85, 90, 78, 92]

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

code generated:



Output:



Fixes & Explanations:

1. Indentation Errors
    - Original code had no indentation inside the function.
    - Fixed by properly indenting total = 0, the for loop, and the return statement.
2. Typo in Variable Name
    - Original: return avrage (misspelled).
    - Fixed to return average.
3. Missing Parenthesis in print Function

- Original:
- print("Average Score is ", calc_average(marks)
  - One closing) missing.
- Fixed by adding the missing parenthesis.

4. Code Readability
   - Added comments for clarity.
   - Separated marks declaration and print statement for better readability.

Observations:

1.A function is created to calculate the average of marks.

2.It first sets total = 0.

3.It goes through each mark in the list and adds it to the total.

4.After adding all marks, it divides the total by the number of marks to find the average.

5.The function then returns this average value.

6.A list of marks [85, 90, 78, 92] is given.

7.Finally, the program prints the average score of these marks.

Task 2

PEP 8 Compliance
Task: Use AI to refactor Python code to follow PEP 8 style guidelines.
Sample Input Code:
*def area_of_rect(L,B):return L*B*
print(area_of_rect(10,20))

Prompt:

You are given Python code that does not follow PEP 8 style guidelines.

Tasks:

1. Refactor the code so it follows PEP 8 formatting rules (proper naming, indentation, spacing, and readability).

2. Keep the functionality the same.
3. Provide the corrected code.

Faulty Python Code:

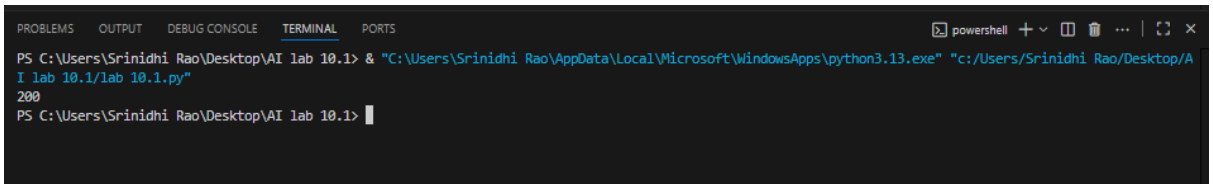
*def area_of_rect(L,B):return L*B*

print(area_of_rect(10,20))

Code generated:

```
def area_of_rectangle(length, breadth):
    """Calculate the area of a rectangle."""
    return length * breadth


print(area_of_rectangle(10, 20))
```

Output:



Fixes Made (PEP 8 Compliance)

1. Function Name
    - Changed area_of_rect → area_of_rectangle (use descriptive lowercase with underscores).
2. Variable Names
    - Changed L → length, B → breadth (avoid single-letter uppercase variable names).
3. Formatting
    - Put return on a new line.
    - Added indentation inside the function.
    - Added two blank lines before the print statement (PEP 8 recommends two blank lines after function definitions).

4. Docstring
    - Added a short docstring to explain the function purpose.
5. Spacing
    - Added spaces after commas in function calls: area_of_rectangle(10, 20).

Observation:

1. A function named area_of_rectangle is created.

2.The function takes two inputs: length and breadth.

3.It multiplies these two numbers to find the area of a rectangle.

4.A short docstring is added to explain what the function does.

5.The program then calls the function with 10 and 20.

6.Finally, it prints the result of the calculation.

Task 3

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

Sample Input Code:

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

Prompt:

You are given Python code that is functional but hard to read because of short names and no comments.

Tasks:

Refactor the code to make it more readable without changing its logic.

Use descriptive variable names instead of single letters.

Add a docstring to the function.

Add inline comments where needed.

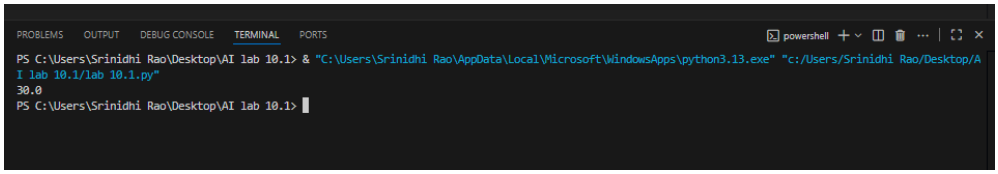
Input Code:

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

Code generated:

```python
def calculate_percentage(amount, percent):
    """Calculate the percentage of a given amount."""
    return amount * percent / 100


# Example values
total_amount = 200   # The main amount
percentage = 15      # The percentage to calculate

# Print the result of the percentage calculation
print(calculate_percentage(total_amount, percentage))
```

Output:



Fixes & Explanations:

1.Function Name

- Changed c → calculate_percentage (clear and descriptive).

2.Variable Names

- Changed a → total_amount, b → percentage (self-explanatory).

3.Docstring

- Added a docstring inside the function to describe its purpose.

4.Inline Comments

- Added comments to explain the role of variables and the print statement.

5.Readability

- Proper indentation, spacing, and clear structure.

Observations:

1.A function called calculate_percentage is defined.

2.It takes two inputs:

- amount → the main value.
- percent → the percentage to be calculated.

3.The function multiplies amount and percent, then divides by 100 to get the percentage value.

4.Two variables are created:

- total_amount = 200
- percentage = 15

5.The function is called with these values.

6.The program prints the result of the calculation.

Task 4

Refactoring for Maintainability
Task: Use AI to break repetitive or long code into reusable functions.
Sample Input Code:
students = ["Alice", "Bob", "Charlie"]
print("Welcome", students[0])
print("Welcome", students[1])
print("Welcome", students[2])

Prompt:

You are given Python code that contains repetitive print statements.

Tasks:

1. Refactor the code to improve maintainability.

2. Break the repeated logic into a reusable function.
3. Use a loop to handle multiple students instead of repeating lines.
4. Add a docstring to the function.

Input Code:

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

print("Welcome", students[0])

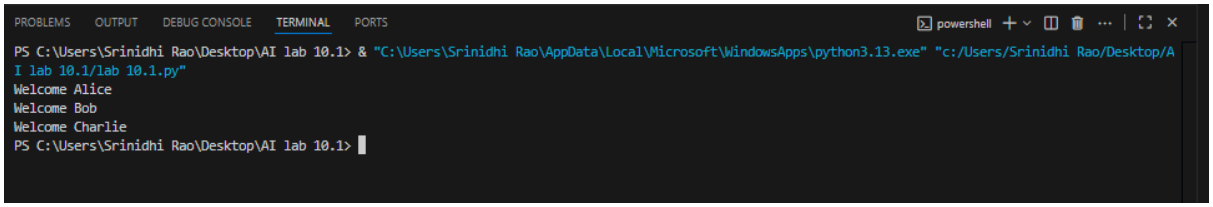
print("Welcome", students[1])

print("Welcome", students[2])

Code generated:

```python
def welcome_student(name):
    """Print a welcome message for a student."""
    print("Welcome", name)


# List of students
students = ["Alice", "Bob", "Charlie"]

# Loop through each student and call the function
for student in students:
    welcome_student(student)
```

Output:



Fixes & Explanations:

1) Reusable Function

- Created welcome_student(name) so the greeting logic is not repeated.

2) Loop Instead of Repetition

- Replaced 3 separate print statements with a for loop that goes through all students.

3) Docstring

- Added a docstring to describe what the function does.

4) Maintainability

- Now if we add more students, we don't need to copy-paste lines; the loop and function handle it automatically.

Observation:

1. A function named welcome_student is created.

- It takes one input: the student's name.
- It prints a welcome message for that student.

2.A list of students is defined: ["Alice", "Bob", "Charlie"].

3.A for loop goes through each student in the list.

4.For each student, the welcome_student function is called.

Task 5

Performance Optimization
Task: Use AI to make the code run faster.

Sample 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))
```

Prompt:

You are given Python code that works but is **not optimized for performance**.

Tasks:

1. Refactor the code to run faster.

2. Replace explicit loops with **list comprehensions** or **vectorized operations** (if NumPy is allowed).
3. Keep the output the same.
4. Ensure the code is clean and Pythonic.

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))
```

Code generated:

```
# Find squares of numbers (optimized using list comprehension)
squares = [n ** 2 for n in range(1, 1_000_000)]

print(len(squares))
```

Output:

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS                                    powershell
PS C:\Users\Srinidhi Rao\Desktop\AI lab 10.1> & "C:\Users\Srinidhi Rao\AppData\Local\Microsoft\WindowsApps\python3.13.exe" "c:/Users/Srinidhi Rao/Desktop/A
I lab 10.1/lab 10.1.py"
999999
PS C:\Users\Srinidhi Rao\Desktop\AI lab 10.1>
```

Fixes & Explanations:

1. **Removed Extra List Creation**
   - Original code first created nums list, then iterated again.
   - Optimized code calculates squares directly, avoiding an extra pass.
2. **Used List Comprehension**
   - Replaced the for loop with a list comprehension: [n ** 2 for n in range(...)].
   - This is faster and more Pythonic.

3. **Underscore in Large Numbers**
    - Used 1_000_000 for readability (PEP 8 compliant)

Observation:

1.The program finds the squares of numbers from 1 up to 999,999.

2.Instead of creating a list first and then looping, it directly builds the list of squares using a list comprehension.

squares = [n ** 2 for n in range(1, 1_000_000)]

3.This makes the code shorter, faster, and more memory-efficient.

4.Finally, it prints the length of the list, which is 999999.

Task 6

Complexity Reduction
Task: Use AI to simplify overly complex logic.

Sample 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"
```

Prompt:

You are given Python code that contains overly complex nested if-else logic.

Tasks:

1. Simplify the logic so that the code is easier to read and maintain.

2. Replace deep nesting with cleaner structures like elif chains or dictionary mapping.
3. Keep the functionality the same (grading system).
4. Add a docstring to explain the function.

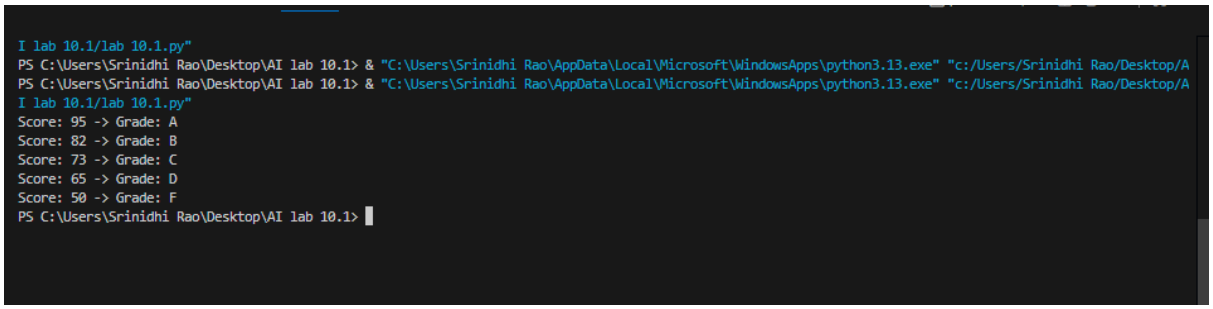
Input Code:

```
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"
```

Code generated:

```python
def grade(score):
    """Return the grade for a given score."""
    if score >= 90:
        return "A"
    elif score >= 80:
        return "B"
    elif score >= 70:
        return "C"
    elif score >= 60:
        return "D"
    else:
        return "F"


# Example usage
scores = [95, 82, 73, 65, 50]

for s in scores:
    print(f"Score: {s} -> Grade: {grade(s)}")
```

Output:



Fixes & Explanations:

1.Reduced Nested if-else

- Original code had many nested else blocks, making it hard to read.
- Replaced with elif chain (much cleaner).

2.Alternative Dictionary Mapping

- Another approach: store thresholds in a dictionary and check them in order.
- This makes it easier to extend or modify grading rules.

3.Added Docstrings

- Explained the function purpose

Observation:

1. A function grade(score) is defined.
    - It takes one input: a **score** (number).
    - It returns a letter grade based on the score.
2. The grading rules are:
    - 90 or above → "A"
    - 80–89 → "B"
    - 70–79 → "C"
    - 60–69 → "D"
    - Below 60 → "F"
3. A list of scores [95, 82, 73, 65, 50] is used as test examples.
4. For each score, the function is called and the result is printed.
5. The output shows the score alongside its corresponding grade.