AI-Assignment-10.1

NAME: Karthikeya
HT.NO:2303A51306
Batch:05

Task Description #1 – Syntax and Logic Errors
Prompt:
Task: Use AI to identify and fix syntax and logic errors in a faulty [Python script].
Sample Input Code:

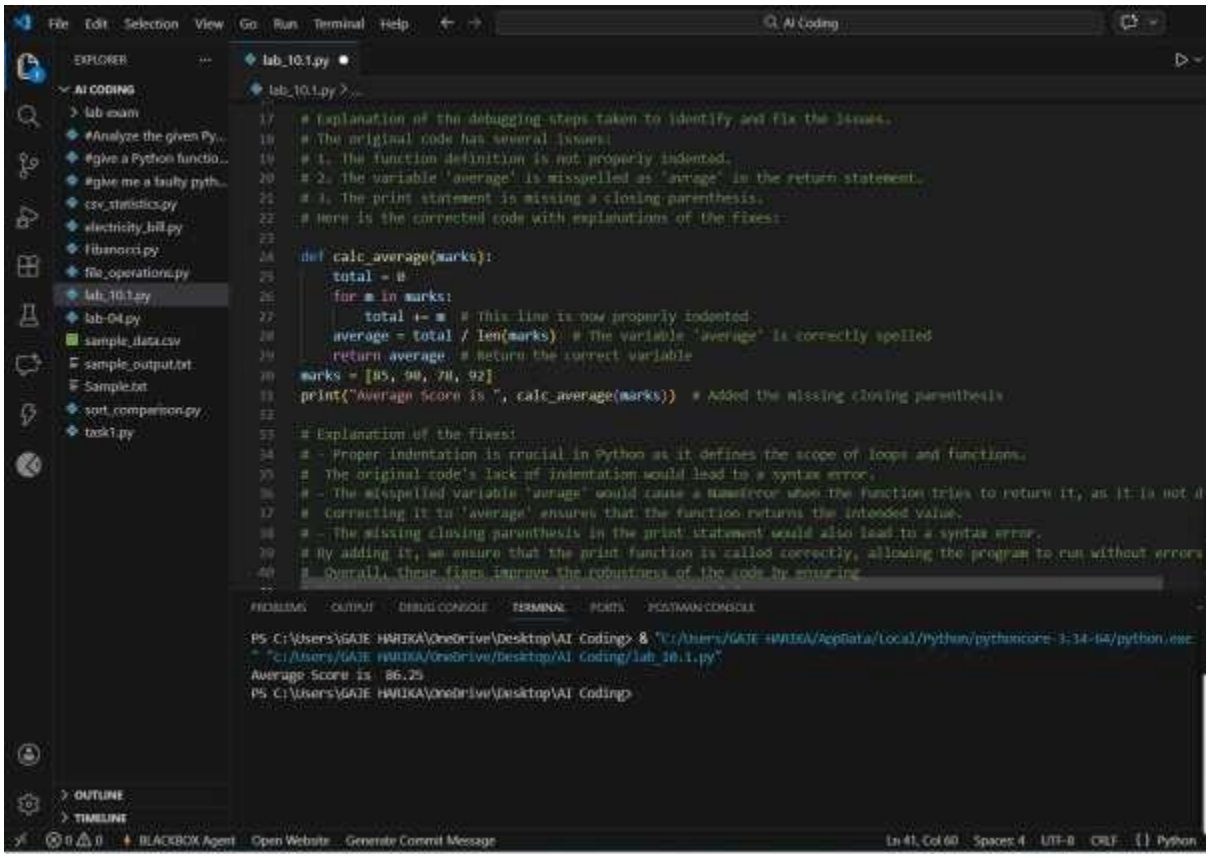
```
# Calculate average score of a student
def calc_average(marks):
total = 0
for m in marks:
total += m
average = total / len(marks)
return avrage # Typo here
marks = [85, 90, 78, 92]
print("Average Score is ", calc_average(marks)
```

Expected Output:
• Corrected and runnable Python code with explanations of the fixes.

Refactored Code:

```
def calc_average(marks):
    total = 0
    for m in marks:
        total += m
    average = total / len(marks)
    return average
marks = [85, 90, 78, 92]
print("Average Score is ", calc_average(marks))
```

Output:
Average Score is 86.25



Explanation of the fixes:

- Proper indentation is crucial in Python as it defines the scope of loops and functions. The original code's lack of indentation would lead to a syntax error.

- The misspelled variable 'avrage' would cause a NameError when the function tries to return it, as it is not defined. Correcting it to 'average' ensures that the function returns the intended value.

- The missing closing parenthesis in the print statement would also lead to a syntax error. By adding it, we ensure that the print function is called correctly, allowing the program to run without errors.Overall, these fixes improve the robustness of the code by ensuring it is syntactically correct and functions as intended.

Task Description #2 – PEP 8 Compliance

Prompt:

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))

Expected Output:

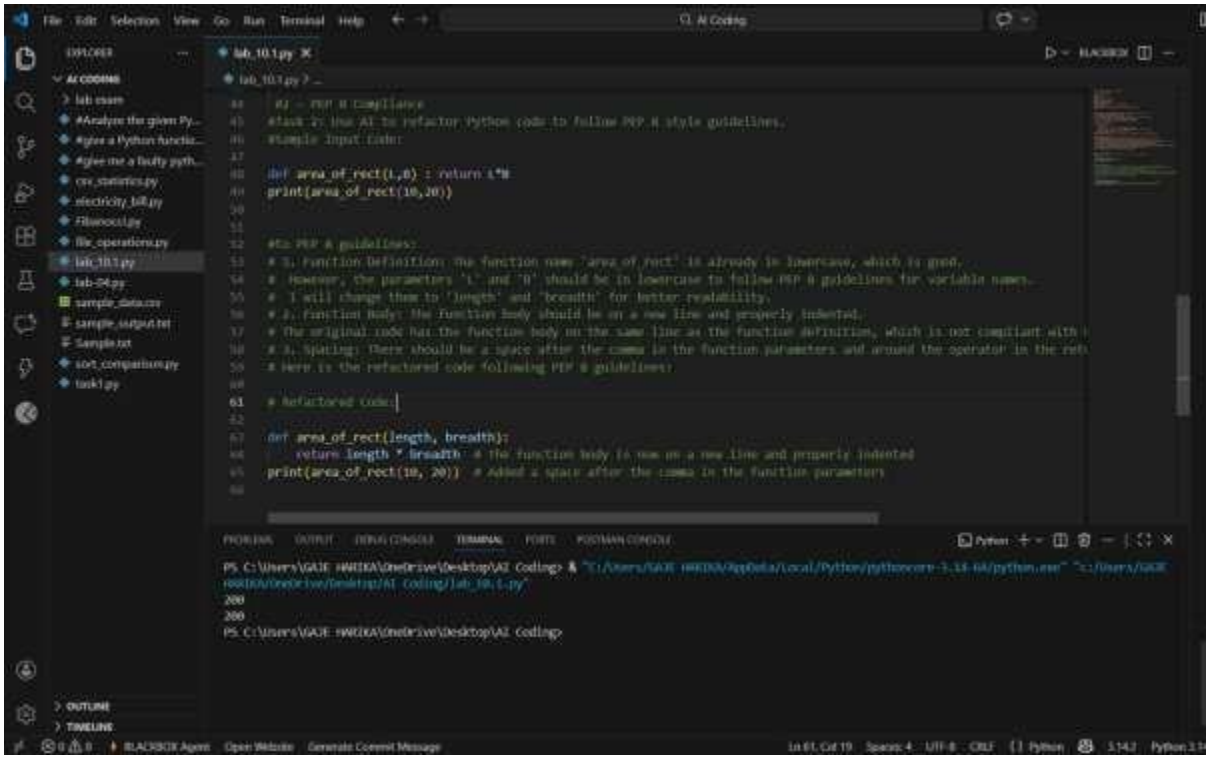
• Well-formatted PEP 8-compliant Python code.

Refactored Code:

def area_of_rect(length, breadth):

```
    return length * breadth
print(area_of_rect(10, 20))
```

Output:



Explanation of the fixes:

1. Function Definition: The function name 'area_of_rect' is already in lowercase, which is good. However, the parameters 'L' and 'B' should be in lowercase to follow PEP 8 guidelines for variable names. I will change them to 'length' and 'breadth' for better readability.

2. Function Body: The function body should be on a new line and properly indented. The original code has the function body on the same line as the function definition, which is not compliant with PEP 8.

3. Spacing: There should be a space after the comma in the function parameters and around the operator in the return statement for better readability.

Task Description

#3 – Readability Enhancement

Prompt:

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))
```

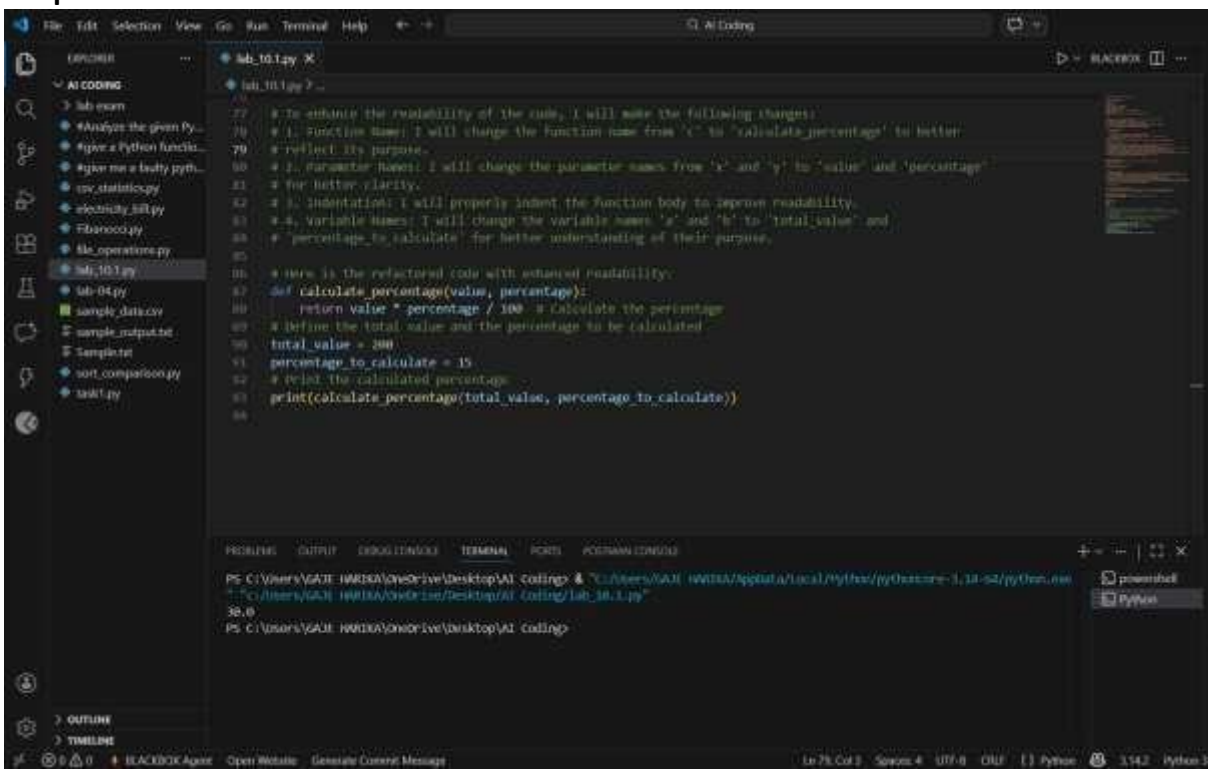
Expected Output:

• Python code with descriptive variable names, inline comments, and clear formatting.

Refactored Code:

```
def calculate_percentage(value, percentage):
    return value * percentage / 100
total_value = 200
percentage_to_calculate = 15
# Print the calculated percentage
print(calculate_percentage(total_value, percentage_to_calculate))
```

Output:



Explanation of the fixes:

1. Function Name: I will change the function name from 'c' to 'calculate_percentage' to better reflect its purpose.
2. Parameter Names: I will change the parameter names from 'x' and 'y' to 'value' and 'percentage' for better clarity.
3. Indentation: I will properly indent the function body to improve readability.
4. Variable Names: I will change the variable names 'a' and 'b' to 'total_value' and 'percentage_to_calculate' for better understanding of their purpose.

Task Description #4 – Refactoring for Maintainability

Prompt:

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])
```

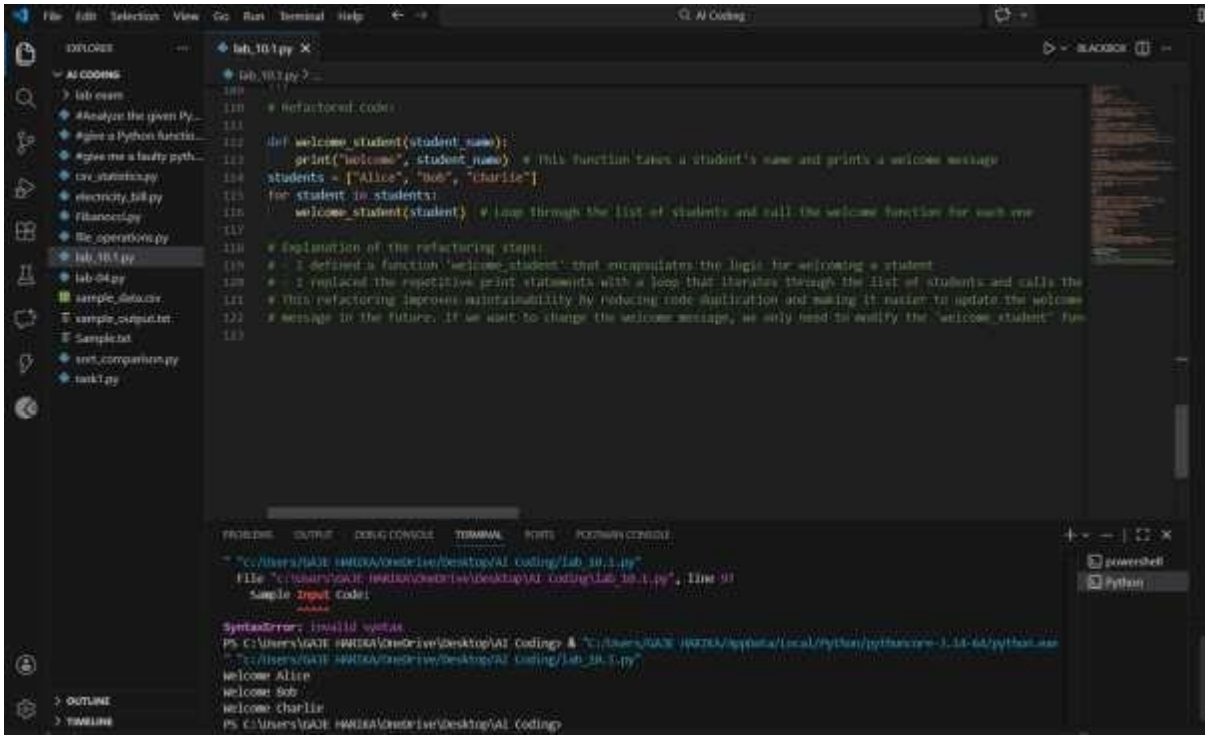
Expected Output:

• Modular code with reusable functions.

Refactored Code:

```
def welcome_student(student_name):
    print("Welcome", student_name)
students = ["Alice", "Bob", "Charlie"]
for student in students:
    welcome_student(student)
```

Output:



Explanation of the fixes:

- I defined a function 'welcome_student' that encapsulates the logic for welcoming a student
- I replaced the repetitive print statements with a loop that iterates through the list of students and calls the 'welcome_student' function for each student. This refactoring improves maintainability by reducing code duplication and making it easier to update the welcome message in the future. If we want to change the welcome message, we only need to modify the 'welcome_student' function, and it will automatically apply to all students without needing to change multiple lines of code.

Task Description #5 – Performance Optimization

Prompt:

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))
```

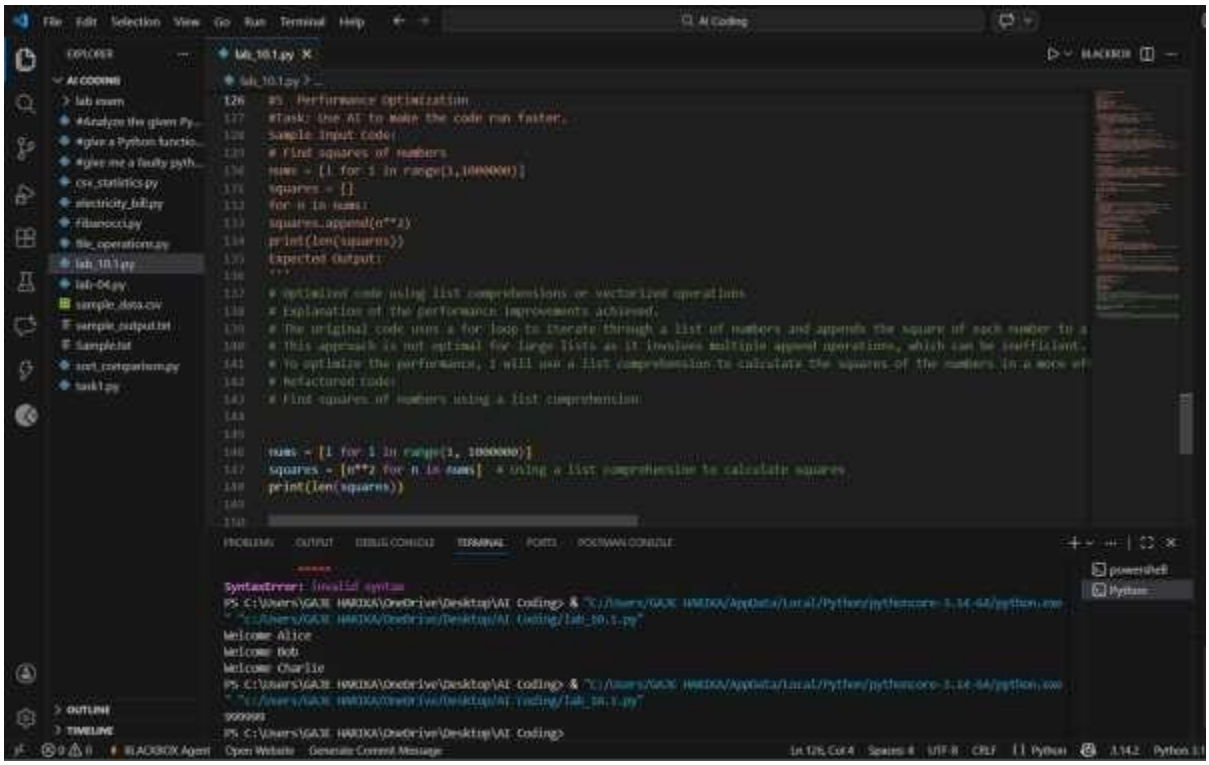
Expected Output:

• Optimized code using list comprehensions or vectorized operations.

Refactored Code:

```
nums = [i for i in range(1, 1000000)]
squares = [n**2 for n in nums]
print(len(squares))
```

Output:



Explanation of the fixes:

- The list comprehension allows us to create the list of squares in a single line of code, which is more efficient than using a for loop with append.
- List comprehensions are optimized in Python and can be significantly faster than traditional loops, especially when dealing with large datasets.
- By using a list comprehension, we reduce the overhead of multiple append operations and improve the overall performance of the code, making it run faster when calculating the squares of a large list of numbers.

Task Description #6 – Complexity Reduction

Prompt:

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"
```

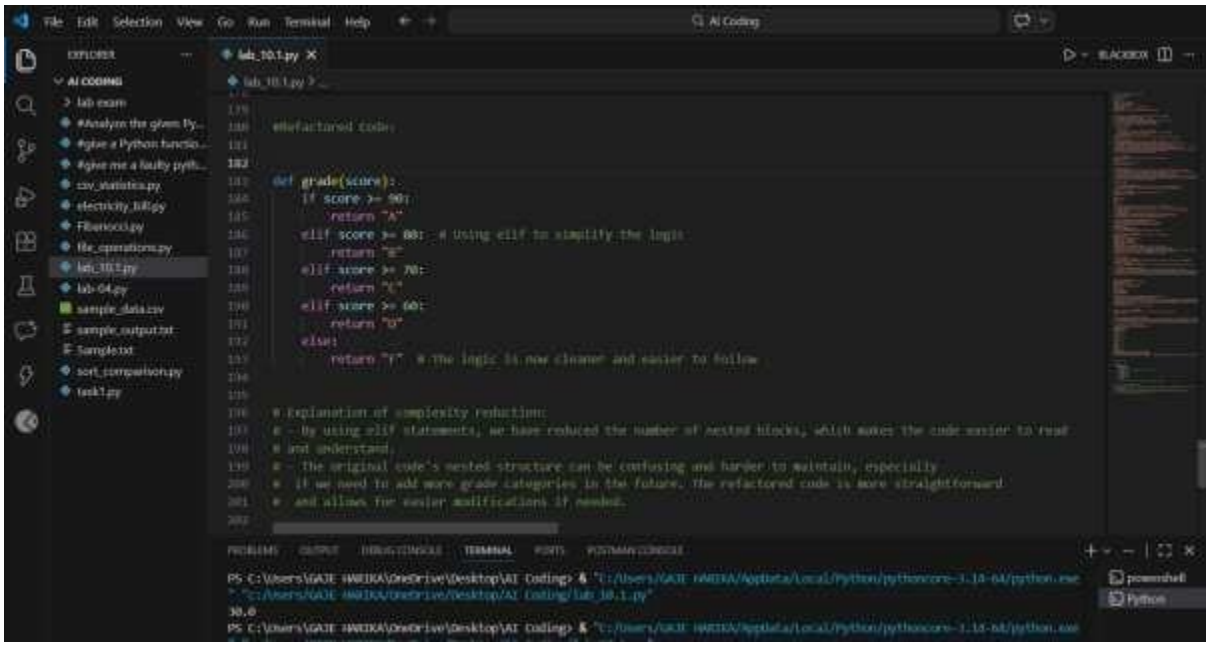
Expected Output:

• Cleaner logic using elif or dictionary mapping.

Refactored 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"
```

Output:



Explanation of the fixes:

- By using elif statements, we have reduced the number of nested blocks, which makes the code easier to read and understand.

- The original code's nested structure can be confusing and harder to maintain, especially if we need to add more grade categories in the future. The refactored code is more straightforward and allows for easier modifications if needed.