Assignment-10.1

Ht.No : 2303A51584

Batch: 05

Task Description #1 – Syntax and Logic Errors

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

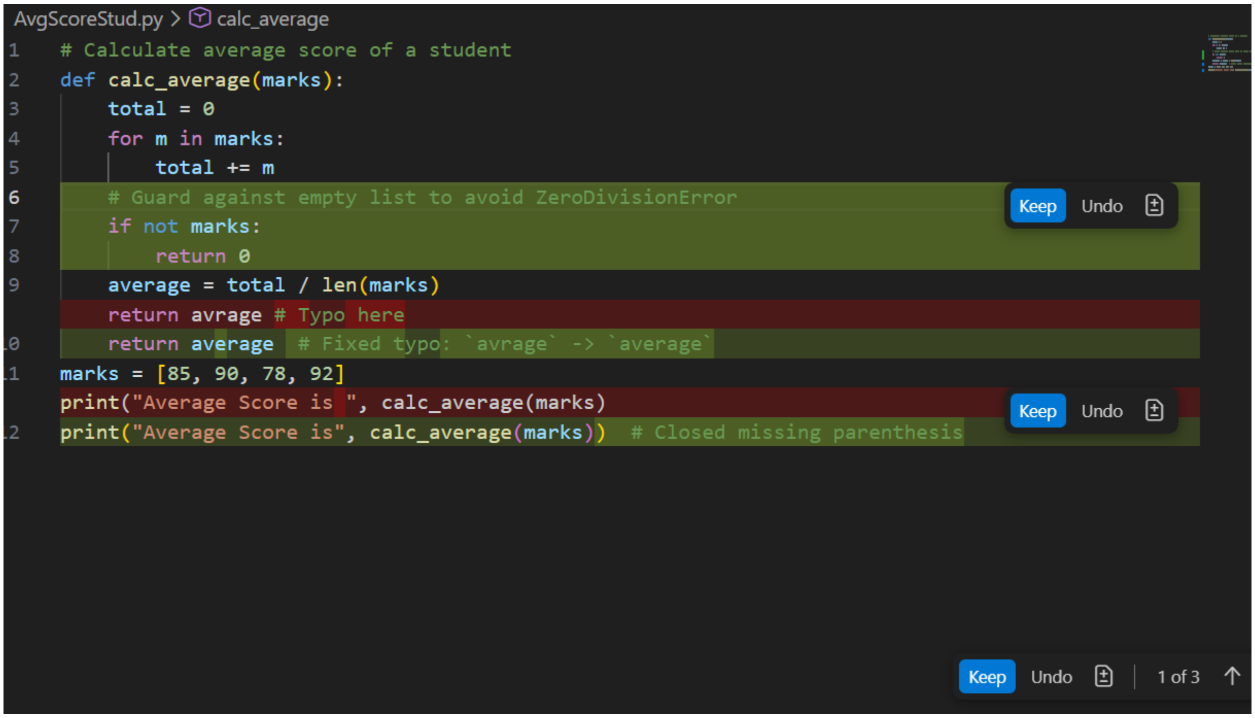
Sample Input Code:

```
# 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)
```

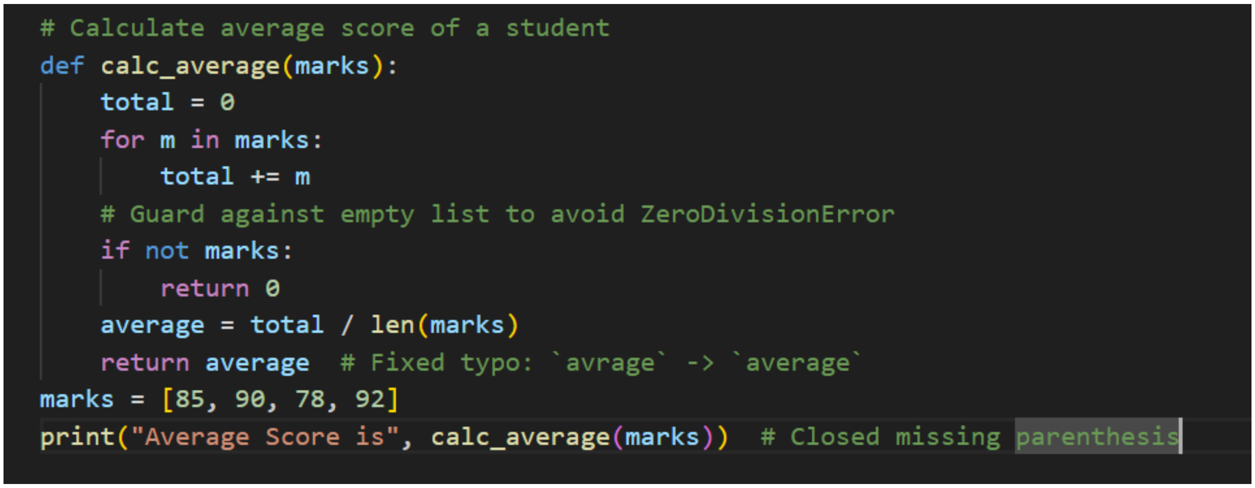
Prompt:

My current code is giving errors. So, identify and fix syntax and logic errors in the code. Explain the fixed(debugged) lines with comments.

Code:

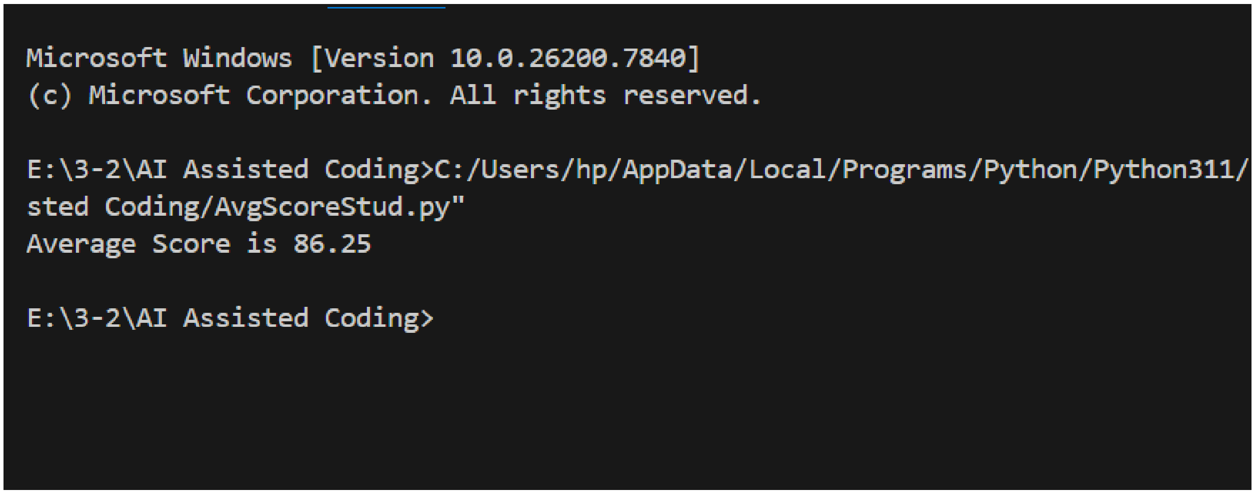


```
# Calculate average score of a student
def calc_average(marks):
    total = 0
    for m in marks:
        total += m
    # Guard against empty list to avoid ZeroDivisionError
    if not marks:
        return 0
    average = total / len(marks)
    return avrage # Typo here
    return average  # Fixed typo: `avrage` -> `average`
marks = [85, 90, 78, 92]
print("Average Score is ", calc_average(marks)
print("Average Score is", calc_average(marks))  # Closed missing parenthesis
```



```
# Calculate average score of a student
def calc_average(marks):
    total = 0
    for m in marks:
        total += m
    # Guard against empty list to avoid ZeroDivisionError
    if not marks:
        return 0
    average = total / len(marks)
    return average  # Fixed typo: `avrage` -> `average`
marks = [85, 90, 78, 92]
print("Average Score is", calc_average(marks))  # Closed missing parenthesis
```

Output:



```
Microsoft Windows [Version 10.0.26200.7840]
(c) Microsoft Corporation. All rights reserved.

E:\3-2\AI Assisted Coding>C:/Users/hp/AppData/Local/Programs/Python/Python311/
sted Coding/AvgScoreStud.py"
Average Score is 86.25

E:\3-2\AI Assisted Coding>
```

Explanation:

Indentation Error

- Statements inside the function were not indented.
- Python requires proper indentation to define function blocks.

Variable Name Typo

- You wrote return avrage instead of return average.
- This causes a NameError because avrage is not defined.

Missing Parenthesis

- The print() statement was missing a closing).
- This causes a SyntaxError.

Block Structure Issue

- The for loop body was not indented properly.

- Python cannot identify which statements belong inside theloop.

Task Description #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:

Refactor Python code to follow PEP 8 style guidelines by,

Improving formatting and spacing

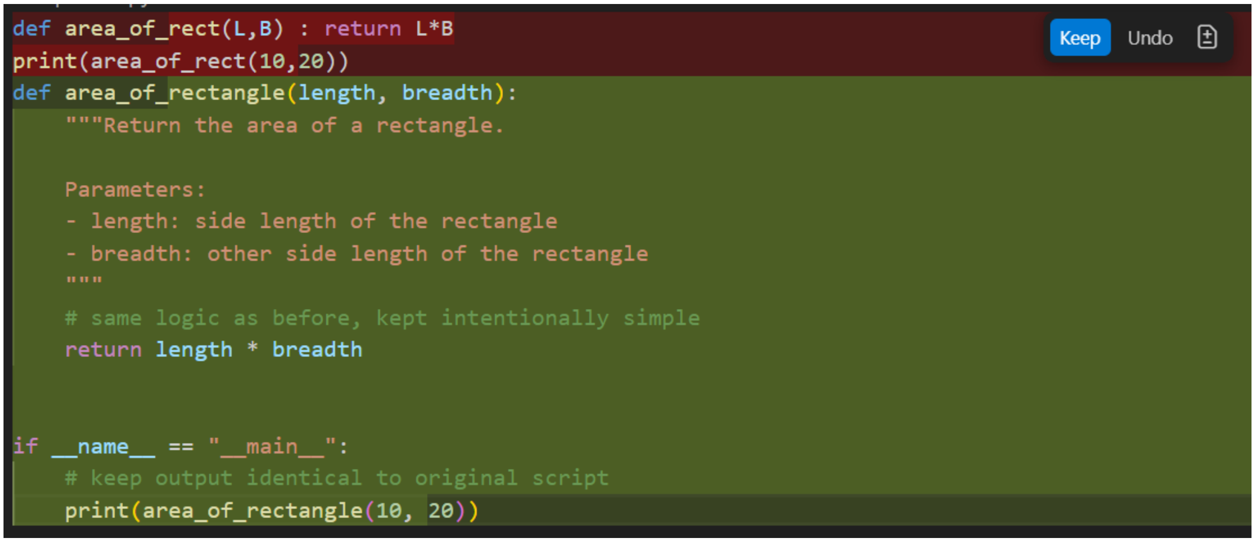
Use proper function and variable naming conventions

Add appropriate line breaks

Ensure readability and clean structure

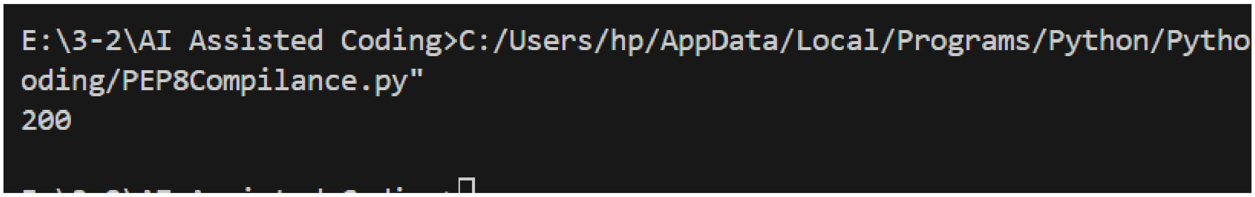
Do not change the logic or output

Code:



```
def area_of_rect(L,B) : return L*B
print(area_of_rect(10,20))
def area_of_rectangle(length, breadth):
    """Return the area of a rectangle.

    Parameters:
    - length: side length of the rectangle
    - breadth: other side length of the rectangle
    """
    # same logic as before, kept intentionally simple
    return length * breadth


if __name__ == "__main__":
    # keep output identical to original script
    print(area_of_rectangle(10, 20))
```

Output:



```
E:\3-2\AI Assisted Coding>C:/Users/hp/AppData/Local/Programs/Python/Pytho
oding/PEP8Compilance.py"
200
```

Explanation:

- Renamed c → calculate_percentage and parameters/vars to describe their roles.
- Added a docstring and comments for clarity.
- Fixed indentation and spacing so the code is easy to read.
- Logic unchanged; it still prints 30.0 for the given inputs.

Task Description #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))
```

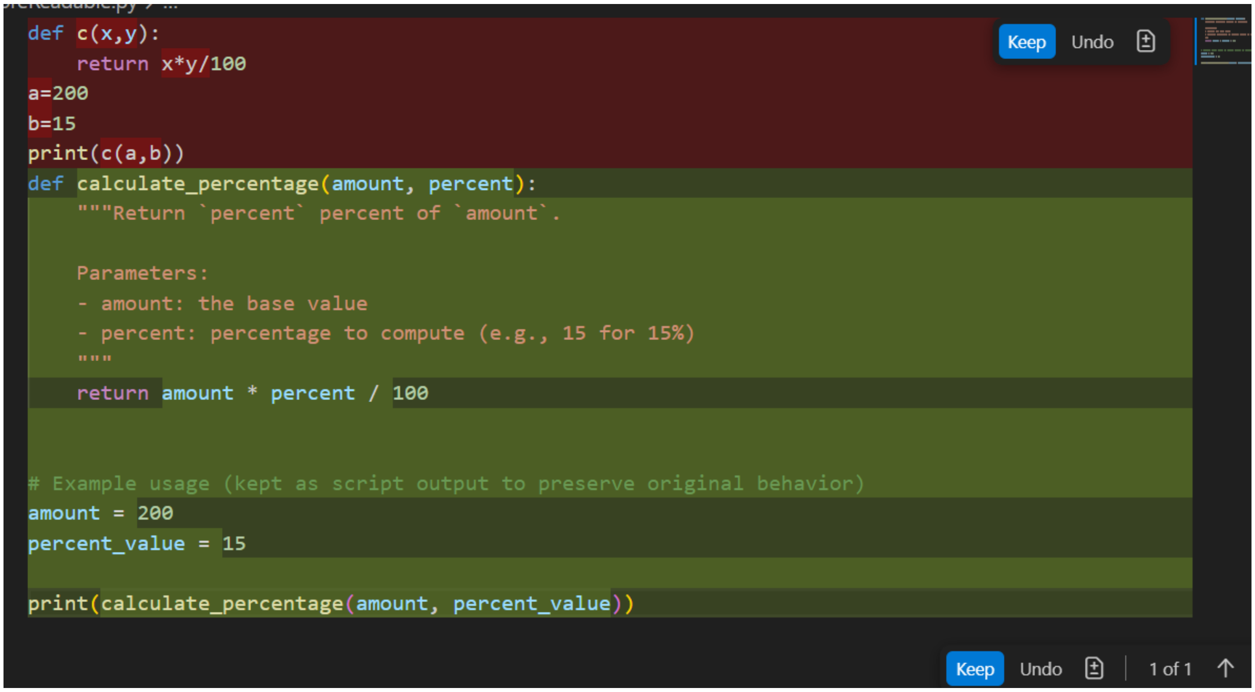
Expected Output:

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

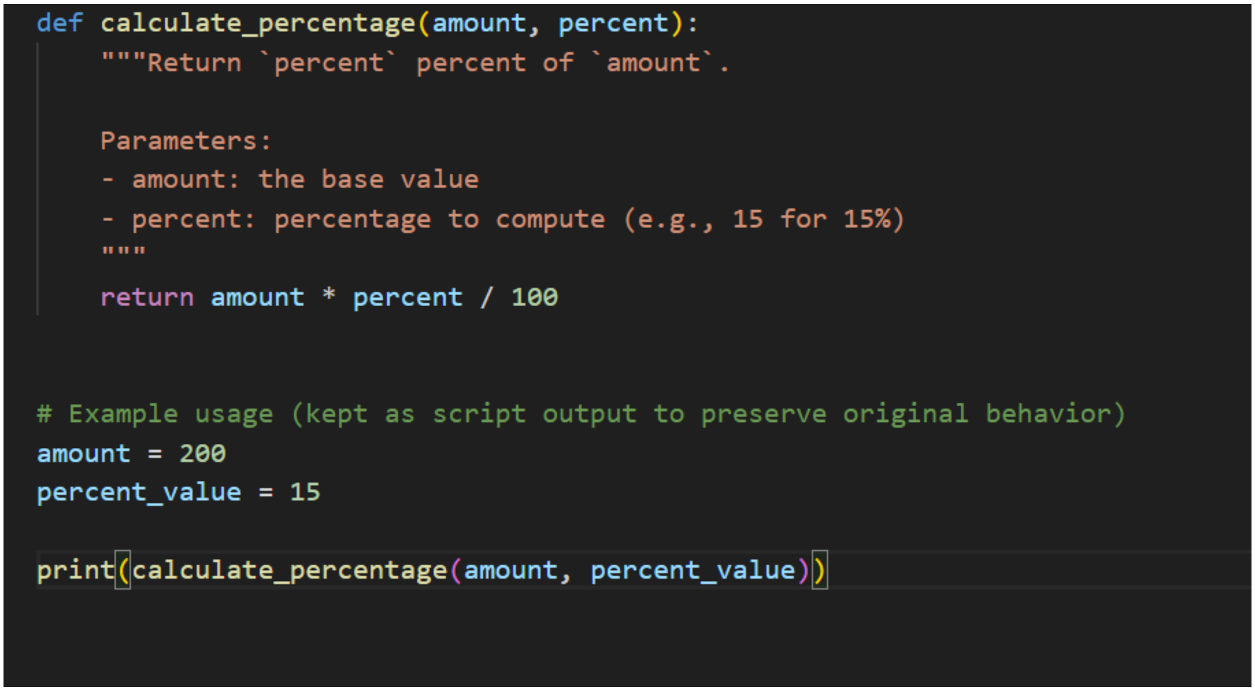
Prompt:

Improve the readability of the Python code without changing its logic. Use meaningful variable and function names, with proper indentation, and add simple comments.

Code:

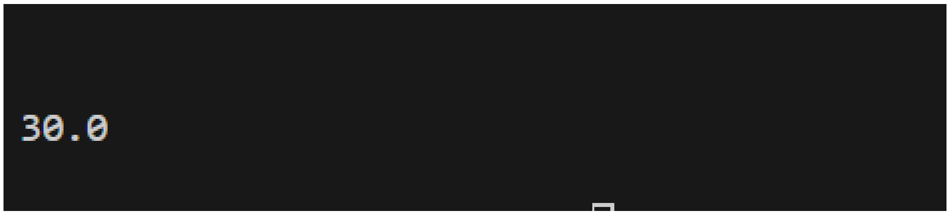


```
def c(x,y):
    return x*y/100
a=200
b=15
print(c(a,b))
def calculate_percentage(amount, percent):
    """Return `percent` percent of `amount`.

    Parameters:
    - amount: the base value
    - percent: percentage to compute (e.g., 15 for 15%)
    """
    return amount * percent / 100


# Example usage (kept as script output to preserve original behavior)
amount = 200
percent_value = 15

print(calculate_percentage(amount, percent_value))
```



```
def calculate_percentage(amount, percent):
    """Return `percent` percent of `amount`.

    Parameters:
    - amount: the base value
    - percent: percentage to compute (e.g., 15 for 15%)
    """
    return amount * percent / 100


# Example usage (kept as script output to preserve original behavior)
amount = 200
percent_value = 15

print(calculate_percentage(amount, percent_value))
```

Output:



```
30.0
```

Explanation:

- Pulled the welcome message into a greet function.
- Created welcome_all to loop through any student list.
- Data (names) now lives in the list, not hard coded print calls.
- Output stays the same but structure is easier to extend.

Task Description #4 – Refactoring for Maintainability

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

functions.

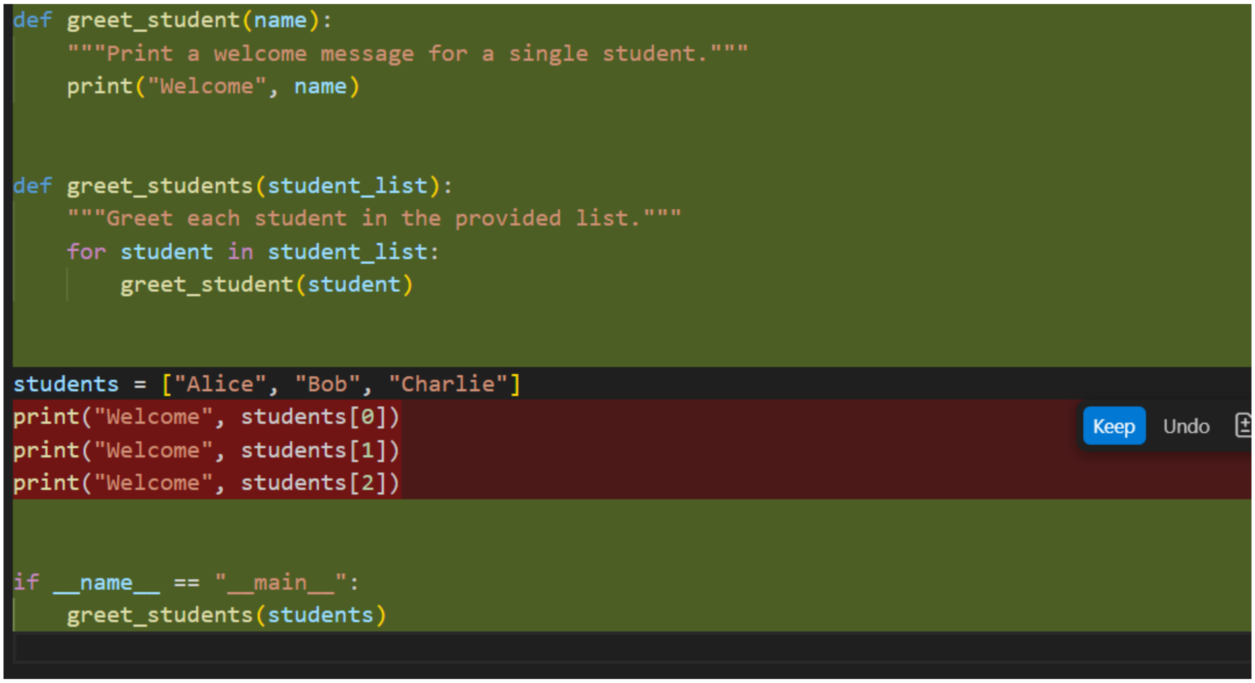
Sample Input Code:

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

print("Welcome", students[0])

print("Welcome", students[1])

print("Welcome", students[2])
```

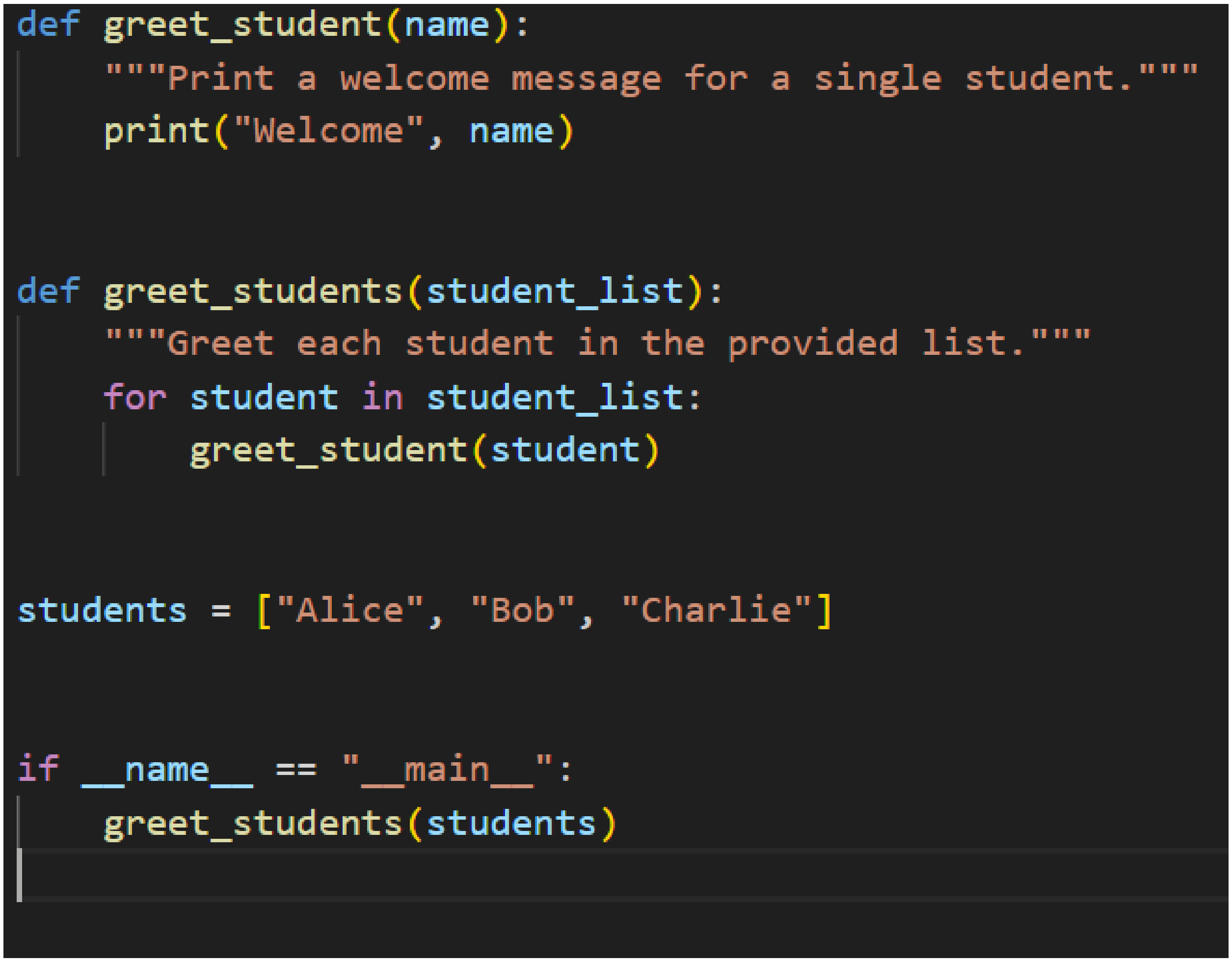
Prompt:

Refactor the Python code to improve maintainability. Break repetitive or long code into reusable functions.

Code:

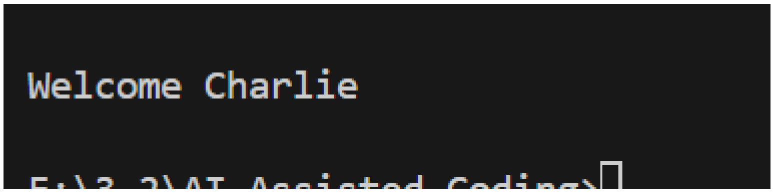


```
def greet_student(name):
    """Print a welcome message for a single student."""
    print("Welcome", name)


def greet_students(student_list):
    """Greet each student in the provided list."""
    for student in student_list:
        greet_student(student)


students = ["Alice", "Bob", "Charlie"]
print("Welcome", students[0])
print("Welcome", students[1])
print("Welcome", students[2])


if __name__ == "__main__":
    greet_students(students)
```



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


def greet_students(student_list):
    """Greet each student in the provided list."""
    for student in student_list:
        greet_student(student)


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


if __name__ == "__main__":
    greet_students(students)
```

Output:



```
Welcome Charlie

E:\3-2\AI Assisted Coding>
```

Explanation:

- Pulled the welcome message into a greet function.
- Created welcome_all to loop through any student list.
- Data (names) now lives in the list, not hard coded print calls.
- Output stays the same but structure is easier to extend.

Task Description #5 – Performance Optimization

Task: Use AI to make the code run faster.

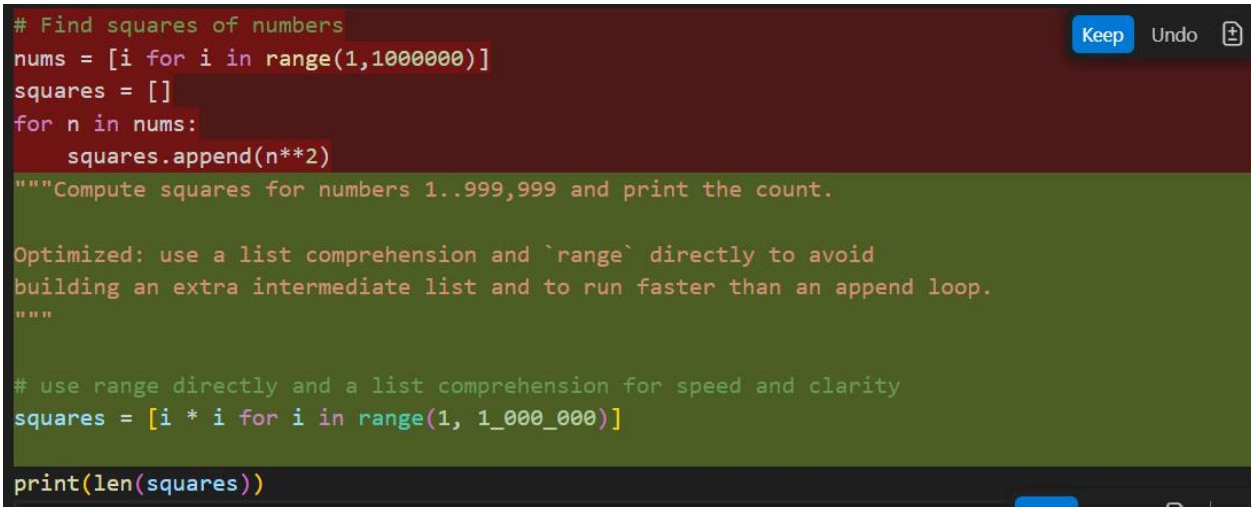
Sample Input Code:

```
# Find squares of numbers
nums = [i for i in range(1,1000000)]
squares = []
for n in nums:
squares.append(n**2)
print(len(squares))
```

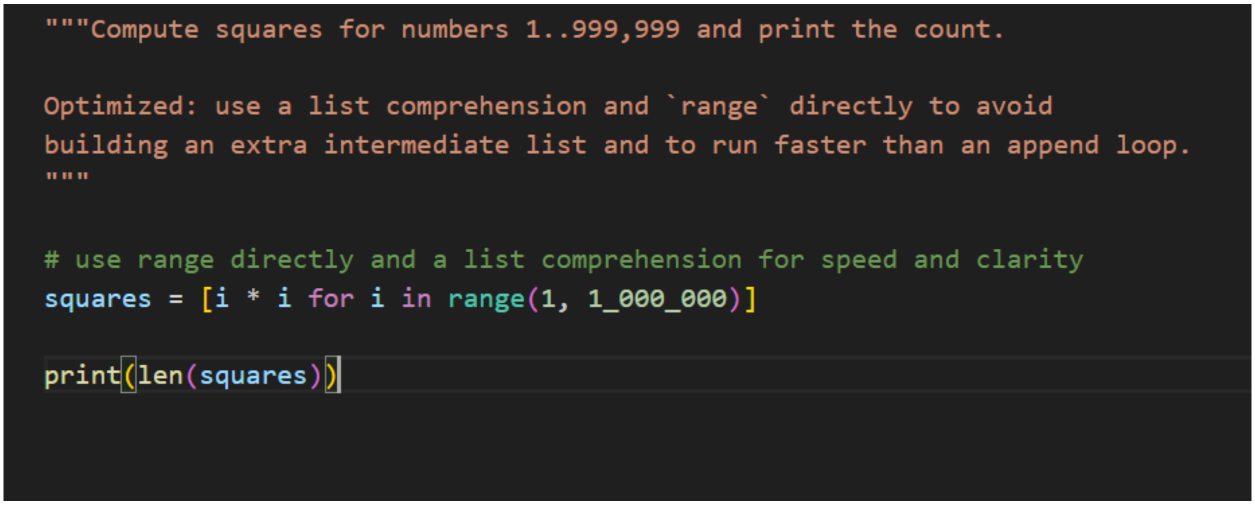
Prompt:

Optimize the python code in List comprehensions or vectorizedoperations to make it run faster.

Code:

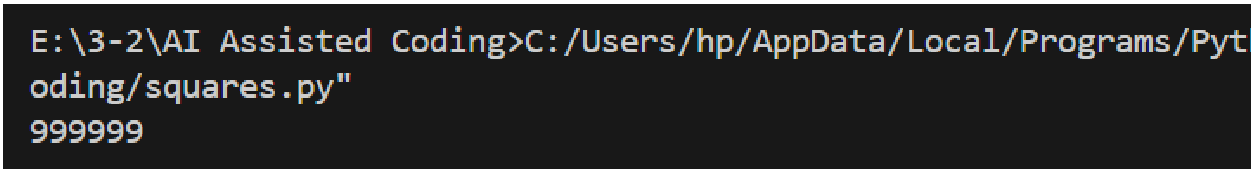


```
# Find squares of numbers
nums = [i for i in range(1,1000000)]
squares = []
for n in nums:
    squares.append(n**2)
"""Compute squares for numbers 1..999,999 and print the count.

Optimized: use a list comprehension and `range` directly to avoid
building an extra intermediate list and to run faster than an append loop.
"""

# use range directly and a list comprehension for speed and clarity
squares = [i * i for i in range(1, 1_000_000)]

print(len(squares))
```



```
"""Compute squares for numbers 1..999,999 and print the count.

Optimized: use a list comprehension and `range` directly to avoid
building an extra intermediate list and to run faster than an append loop.
"""

# use range directly and a list comprehension for speed and clarity
squares = [i * i for i in range(1, 1_000_000)]

print(len(squares))
```

Output:



```
E:\3-2\AI Assisted Coding>C:/Users/hp/AppData/Local/Programs/Pyt
oding/squares.py"
999999
```

Explanation:

Used range() instead of creating a list

- range(1, 1000000) generates numbers only when needed.
- This reduces memory usage compared to storing all numbers in a list.

Replaced for loop and append() with list comprehension

- List comprehensions execute faster than traditional loops.
- They reduce overhead caused by repeated function calls.

Simplified the calculation

- Used n * n instead of n**2.
- Multiplication is faster than exponentiation.

Reduced code complexity

- Fewer lines of code make the program cleaner and easier to read.
- Improves maintainability and execution speed.

Improved overall performance

- The optimized code runs faster and uses less memory.
- Suitable for handling large datasets efficiently.

Task Description #6 – Complexity Reduction

Task: Use AI to simplify overly complex logic.

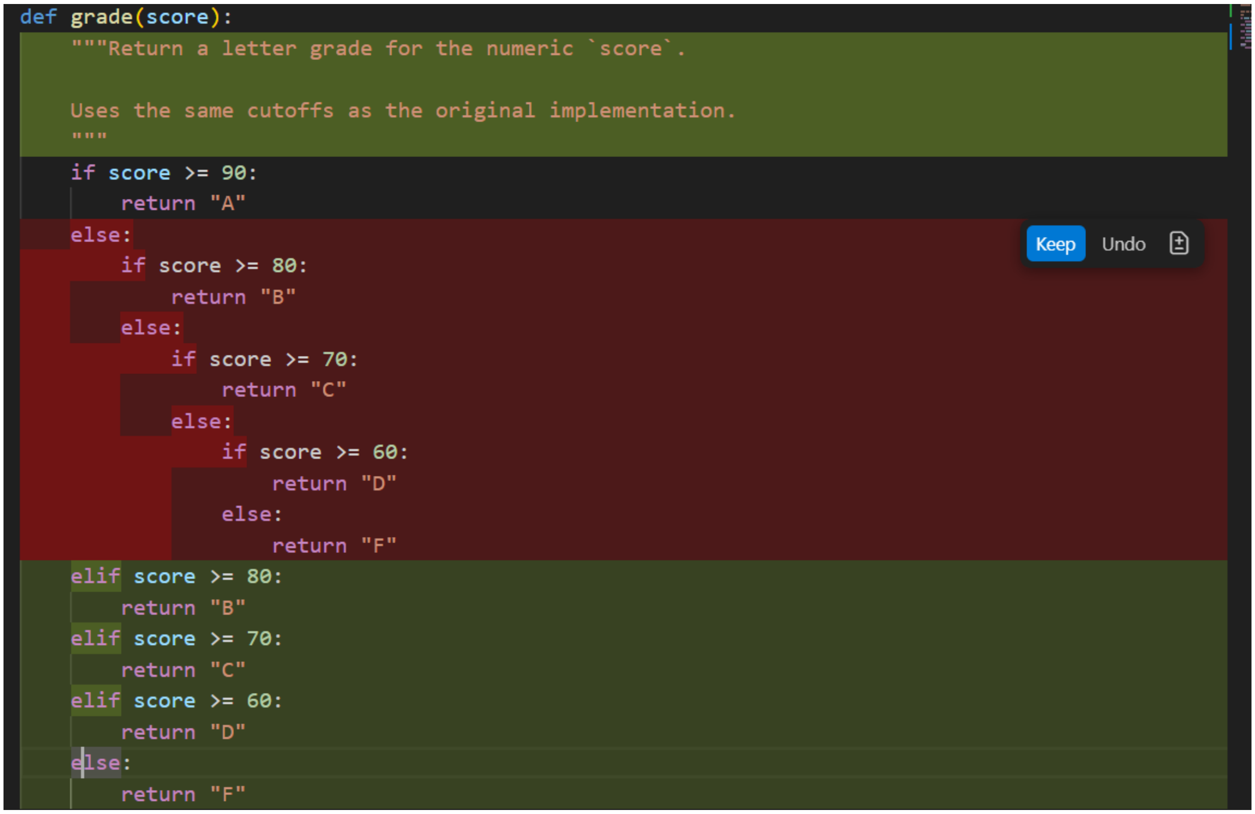
Sample Input Code:

```
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"
```

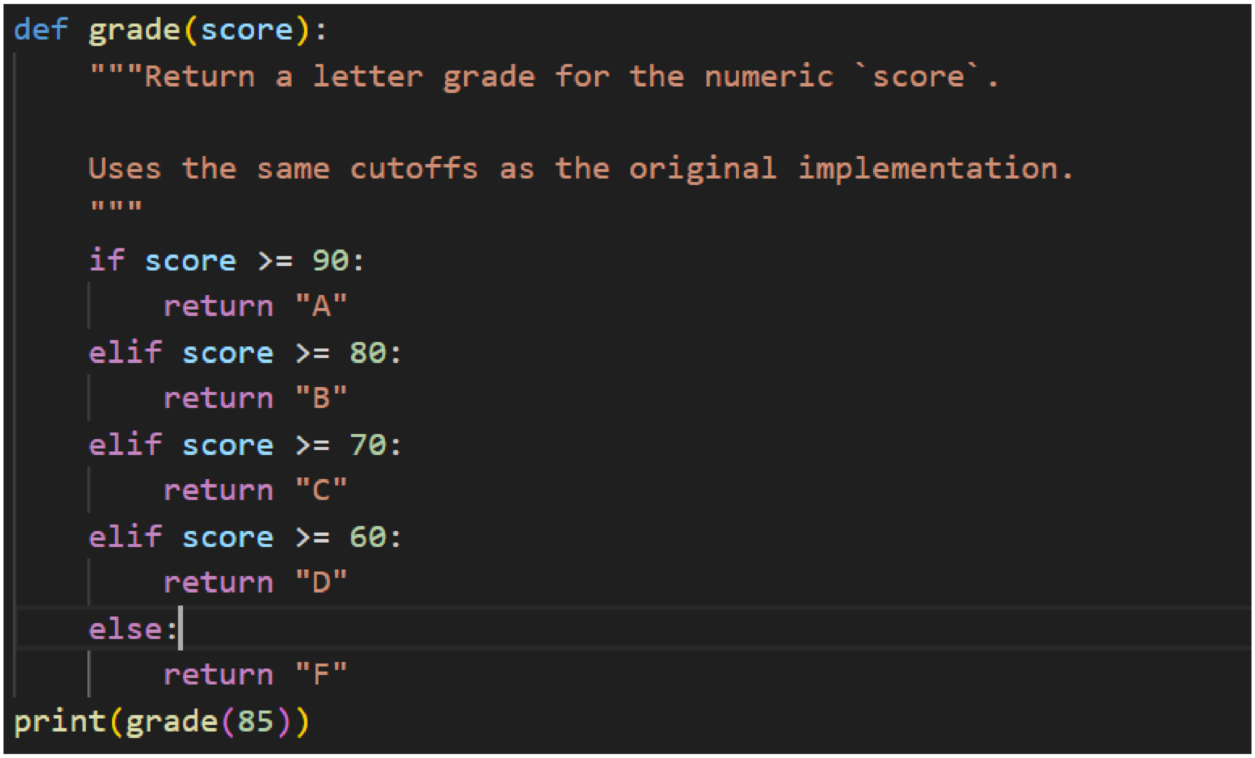
Prompt:

Simplify the over complex logic and make it run effeciently.

Code:

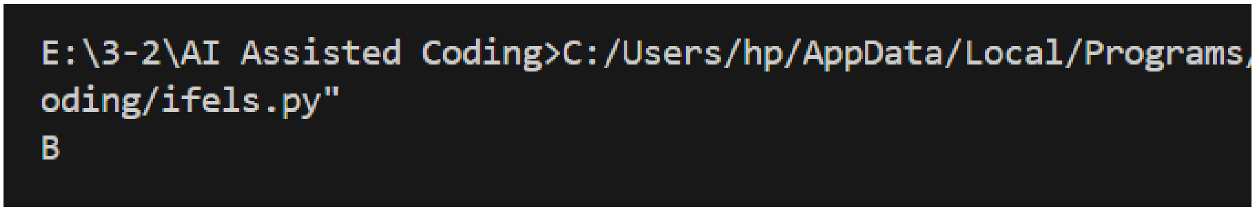


```
def grade(score):
    """Return a letter grade for the numeric `score`.

    Uses the same cutoffs as the original implementation.
    """
    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"
    elif score >= 80:
        return "B"
    elif score >= 70:
        return "C"
    elif score >= 60:
        return "D"
    else:
        return "F"
```



```
def grade(score):
    """Return a letter grade for the numeric `score`.

    Uses the same cutoffs as the original implementation.
    """
    if score >= 90:
        return "A"
    elif score >= 80:
        return "B"
    elif score >= 70:
        return "C"
    elif score >= 60:
        return "D"
    else:
        return "F"
print(grade(85))
```

Output:



```
E:\3-2\AI Assisted Coding>C:/Users/hp/AppData/Local/Programs
oding/ifels.py"
B
```

Explanation:

Replaced nested if-else blocks with elif

- Reduces unnecessary nesting.
- Makes the logic easier to understand.

Improved code readability

- The grading conditions are now clearly ordered.
- Each condition is checked only when the previous one fails.

Reduced logical complexity

- Eliminates deep indentation levels.
- Easier to debug and maintain.

Same functionality with fewer lines

- Output remains unchanged.
- Code is more concise and clean.

Better performance and maintainability

- Less branching improves logical flow.
- Suitable for future updates or modifications.