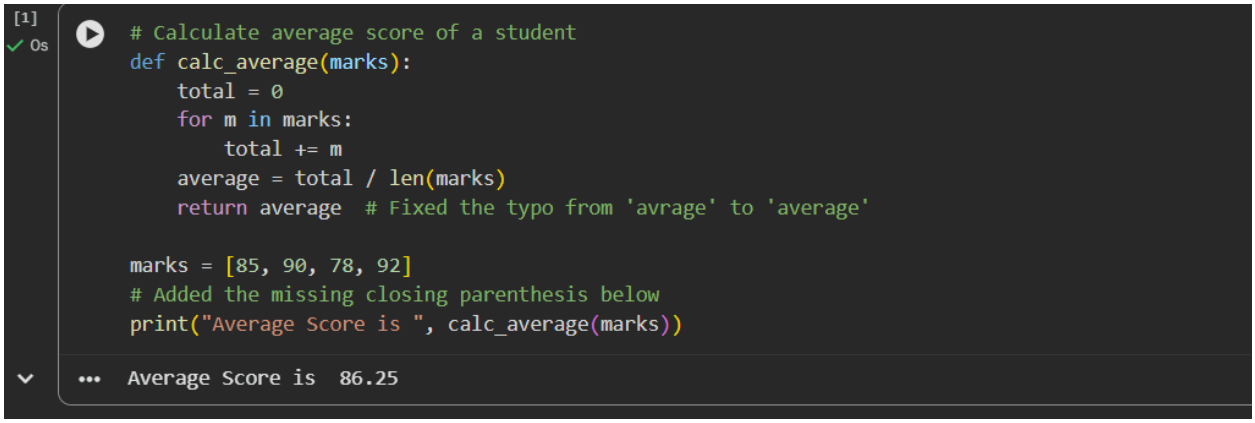
ASSIGNMENT 10.1

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

CODE:

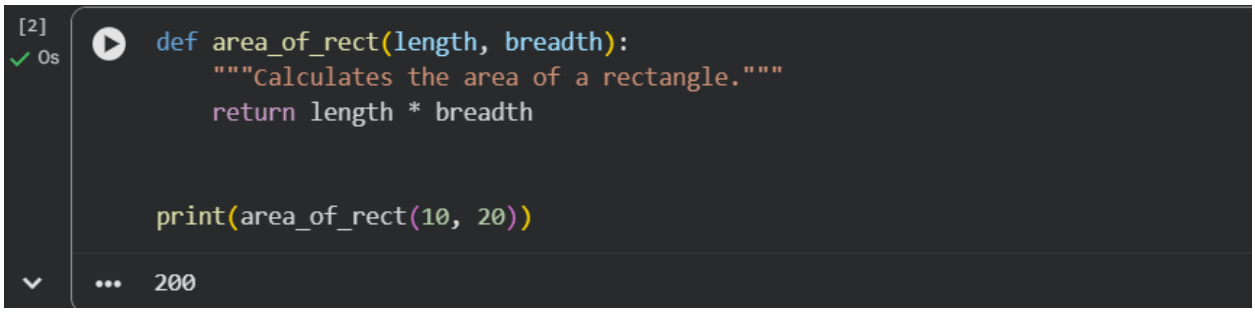


EXPLANATION:

- **Variable Name Consistency:** In the original script, the variable was defined as `average` but returned as `avrage`. Python is case-sensitive and literal, so it would have thrown a `NameError`. I aligned them so the function returns the correct calculated value.
- **Syntax Correction:** The final `print` statement was missing a closing parenthesis `)`. In Python, every opening bracket must have a matching closing one, or the interpreter will stop and throw a `SyntaxError`.
- **Indentation:** While the logic was mostly there, I ensured the function body is properly indented (4 spaces), which is how Python defines the scope of functions and loops.

Task Description #2 – PEP 8 Compliance
Task: Use AI to refactor Python code to follow PEP 8 style guidelines.

CODE:

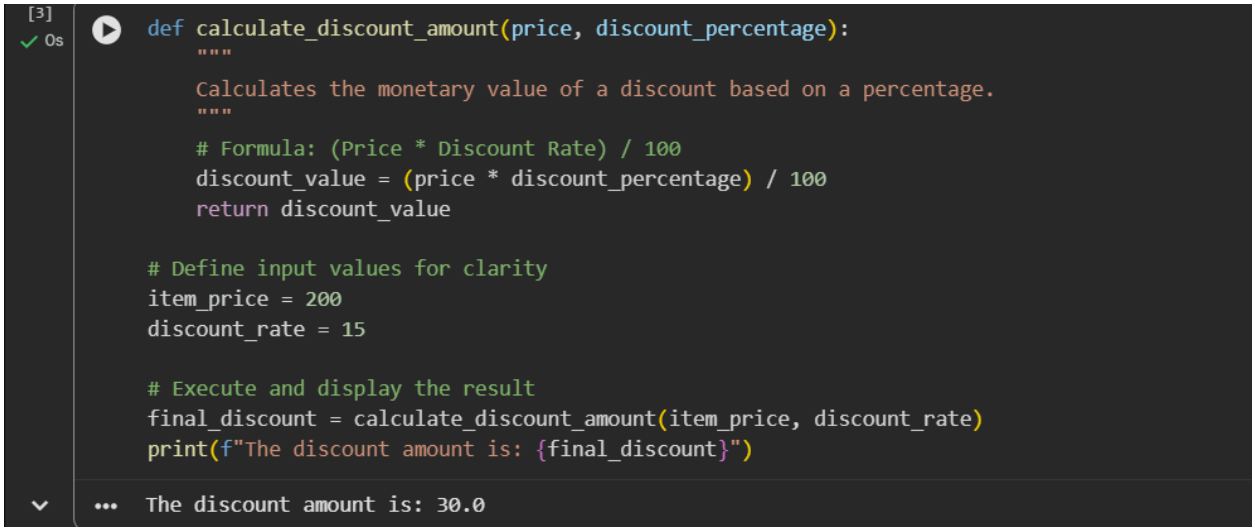


EXPLANATION:

- **Meaningful Naming: Changed single-letter variables `L` and `B` to `length` and `breadth`. PEP 8 encourages descriptive names to make the code's intent clear without needing extra comments.**
- **Whitespace and Spacing: * Removed the space before the colon in the function definition.**
- **Added a space after the comma in the parameters `(length, breadth)`.**
- **Added spaces around the multiplication operator (`*`) for better visual clarity.**
- **Vertical Structure: Moved the `return` statement to its own indented line. PEP 8 discourages "compound statements" (putting the entire function on one line) because it makes debugging harder.**
- **Blank Lines: Added two blank lines after the function definition to separate it from the rest of the script, as per the standard for top-level functions.**

Task Description #3 – Readability Enhancement
Task: Use AI to make code more readable without changing its logic

CODE:



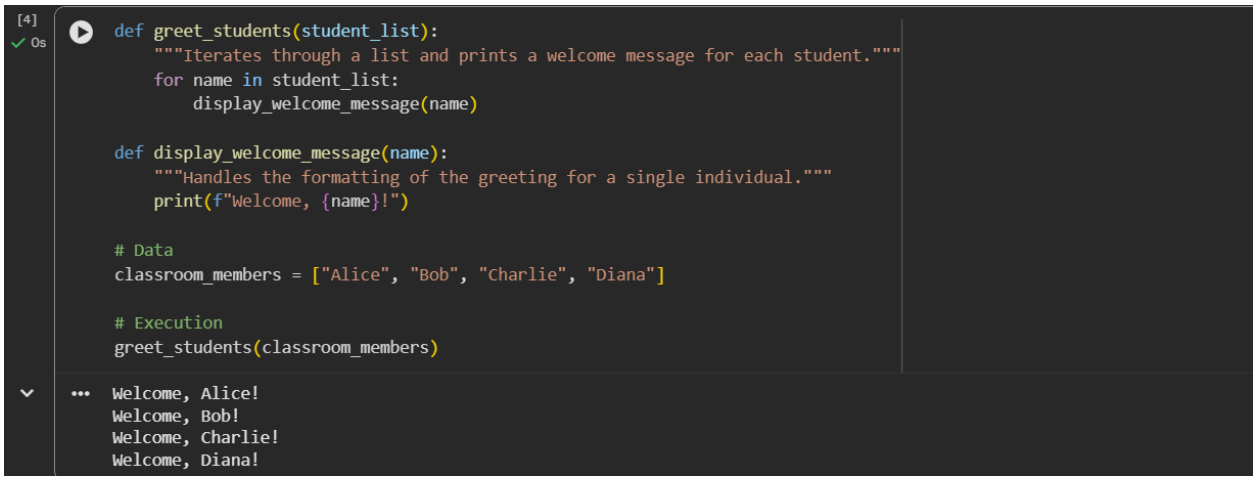
```
[3]
✓ 0s
def calculate_discount_amount(price, discount_percentage):
    """
    Calculates the monetary value of a discount based on a percentage.
    """
    # Formula: (Price * Discount Rate) / 100
    discount_value = (price * discount_percentage) / 100
    return discount_value

# Define input values for clarity
item_price = 200
discount_rate = 15

# Execute and display the result
final_discount = calculate_discount_amount(item_price, discount_rate)
print(f"The discount amount is: {final_discount}")

...  The discount amount is: 30.0
```

EXPLANATION:

- **Descriptive Naming:** Replaced ambiguous variables like `c`, `x`, and `y` with names that reflect their purpose (`calculate_discount_amount`, `price`, and `discount_percentage`). This removes the guesswork for anyone reading the code.
- **Documentation & Comments:** Added a **docstring** (the triple-quoted text) to explain the function's goal and an **inline comment** to explain the math.
- **Explicit Steps:** Instead of performing the math directly on the return line, I assigned it to a variable called `discount_value` first. This makes the logic easier to follow during debugging.
- **Formatted Output:** Used an **f-string** in the print statement to provide context to the numerical result, making the output user-friendly rather than just a raw number.

Task Description #4 – Refactoring for Maintainability
Task: Use AI to break repetitive or long code into reusable Functions.

CODE:



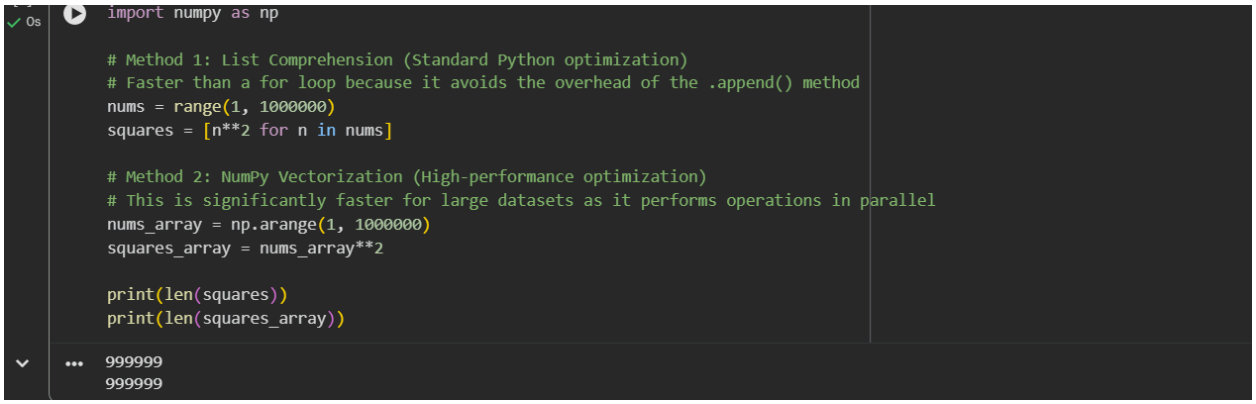
```
[4]
✓ 0s
def greet_students(student_list):
    """Iterates through a list and prints a welcome message for each student."""
    for name in student_list:
        display_welcome_message(name)

def display_welcome_message(name):
    """Handles the formatting of the greeting for a single individual."""
    print(f"Welcome, {name}!")

# Data
classroom_members = ["Alice", "Bob", "Charlie", "Diana"]

# Execution
greet_students(classroom_members)

...
Welcome, Alice!
Welcome, Bob!
Welcome, Charlie!
Welcome, Diana!
```

EXPLANATION:

- **Scalability: By using a `for` loop inside the function, the code now works for 3 students or 3,000 students without changing a single line of logic.**
- **Separation of Concerns: I broke the task into two functions. `greet_students` handles the logic of going through the list, while `display_welcome_message` handles the presentation of the string.**
- **Dry Principle (Don't Repeat Yourself): We removed the repetitive `print` statements. If you later decide to change "Welcome" to "Hello," you only have to change it in one place instead of three.**
- **Abstraction: The main script execution is now a simple function call, making the top-level code much cleaner and easier to read.**

Task Description #5 – Performance Optimization
Task: Use AI to make the code run faster.

CODE:



```
import numpy as np

# Method 1: List Comprehension (Standard Python optimization)
# Faster than a for loop because it avoids the overhead of the .append() method
nums = range(1, 1000000)
squares = [n**2 for n in nums]

# Method 2: NumPy Vectorization (High-performance optimization)
# This is significantly faster for large datasets as it performs operations in parallel
nums_array = np.arange(1, 1000000)
squares_array = nums_array**2

print(len(squares))
print(len(squares_array))
```

```
999999
999999
```

EXPLANATION:

- **Eliminating `.append()`: In the original code, Python has to look up the `append` method and resize the list in memory repeatedly. A List Comprehension performs this construction in a single, optimized step.**
- **Lazy Evaluation: Using `range()` instead of a pre-built list `[i for i in range(...)]` saves memory by generating numbers on the fly rather than storing a million integers twice.**

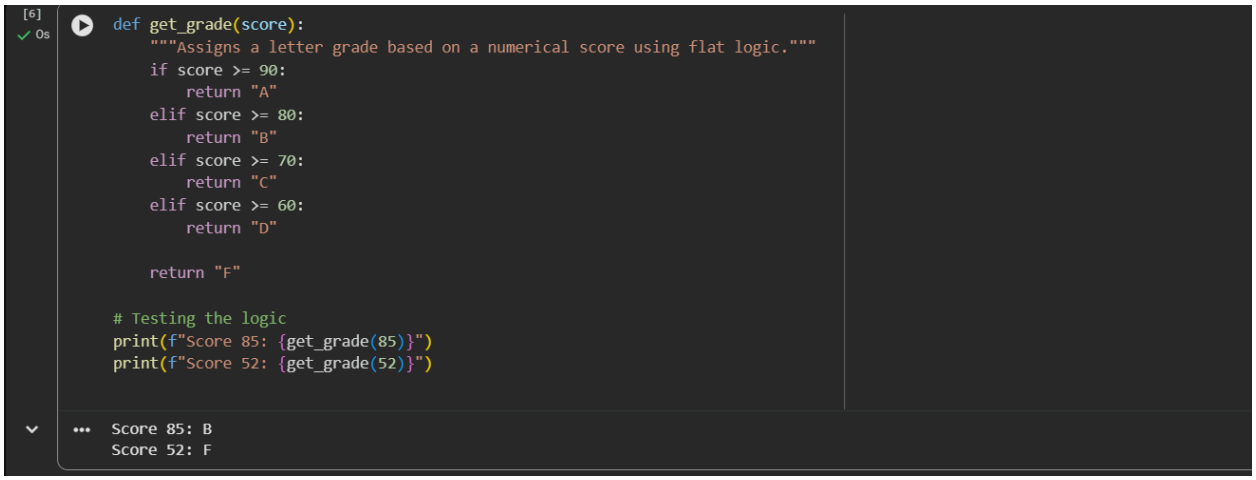
- **Vectorization with NumPy: By using NumPy arrays, we perform "SIMD" (Single Instruction, Multiple Data) operations. Instead of calculating each square one by one, the CPU processes blocks of numbers simultaneously.**

- **Memory Efficiency: The optimized versions reduce the "memory footprint," preventing the computer from slowing down due to high RAM usage when handling large datasets..**

Task Description #6 – Complexity Reduction

Task: Use AI to simplify overly complex logic.

CODE:



```
[6]
✓ 0s

def get_grade(score):
    """Assigns a letter grade based on a numerical score using flat logic."""
    if score >= 90:
        return "A"
    elif score >= 80:
        return "B"
    elif score >= 70:
        return "C"
    elif score >= 60:
        return "D"

    return "F"

# Testing the logic
print(f"Score 85: {get_grade(85)}")
print(f"Score 52: {get_grade(52)}")

...
Score 85: B
Score 52: F
```

EXPLANATION:

- **Flattening with `elif`:** By using `elif` (else-if), we remove the deep nesting. This tells Python to check the next condition only if the previous one was false, keeping all conditions at the same indentation level.
- **Early Returns:** Since `return` immediately exits the function, we don't need a final `else` for the "F" grade. If none of the above conditions are met, the code naturally falls through to the final return statement.
- **Readability (Cyclomatic Complexity):** The "mental load" required to follow the code is significantly reduced. In the original version, you had to keep track of four levels of "else" logic; here, you simply read from top to bottom.