SR UNIVERSITY

*School of Computer Science and Artificial Intelligence*

## *Lab Assignment # 10.2*

| | |
|---|---|
| *Program* | *: B. Tech (CSE)* |
| *Specialization* | *:* |
| *Course Title* | *: AI Assisted coding* |
| *Course Code* | *:* |
| *Semester* | *: II* |
| *Academic Session* | *: 2025-2026* |
| *Name of Student* | *: P Abhinav* |
| *Enrollment No.* | *: 2403A51L38* |
| *Batch No.* | *: 52* |
| *Date* | *:10-06-2026* |

### *Task 1:* Use AI to analyze a Python script and correct all syntax and logical errors.

**Prompt:** Analyze the given Python code, identify all syntax and logical errors, correct them, and explain the issues briefly.

**Sample Input Code(Contains Syntactical or Logical Errors):**

```
def calculate_total(nums)
sum = 0
for n in nums
sum += n
return total
```

```
  File "/tmp/ipython-input-2382688864.py", line 1
    def calculate_total(nums)
                               ^
SyntaxError: expected ':'
```

**Corrected Code:**

```
def calculate_total(nums):
    total = 0
    for n in nums:
        total += n
    return total
```

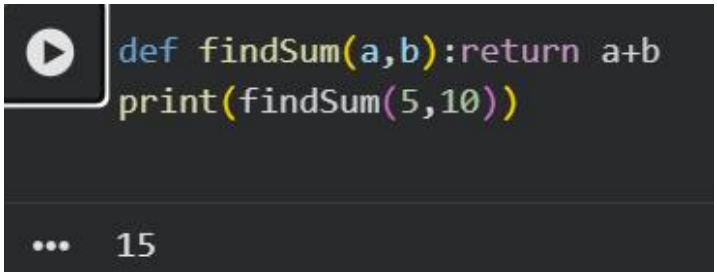
### Code Explanation:

The given code contained multiple syntax and logical errors that prevented it from executing correctly. The function definition was missing a colon, which is required in Python to indicate the start of a function block. The for loop statement also lacked a colon, causing another syntax error. Proper indentation was missing throughout the code, violating Python's indentation-based block structure. The variable name sum was used, which shadows Python's built-in sum() function and can lead to confusion or unexpected behavior. Another major logical error was returning a variable named total that was never defined in the original code. This would result in a runtime error if execution reached that line. To fix these issues, colons were added where required, indentation was corrected, and the variable was consistently renamed to total. These changes ensure logical correctness, syntactic validity, and better coding practice. After correction, the function now executes successfully and correctly computes the sum of the list elements.

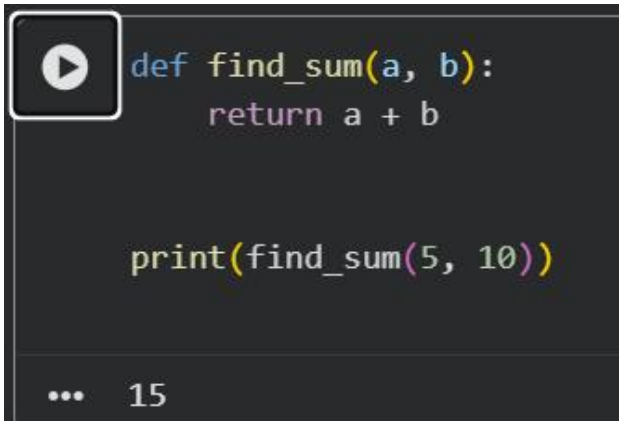
### Task 2: Use AI to refactor Python code to comply with standard coding style guidelines.

**Prompt**: Explain the decision-making logic and loop behavior only. Skip basic syntax explanations.

### Sample Input Code(Contains Syntactical or Logical Errors):




### Corrected Code:



### Explanation:

Although the original code executed correctly, it did not follow Python's recommended PEP 8 style guidelines. The function name was written in camelCase, whereas Python conventions prefer snake_case for function names. There was no spacing between parameters or around operators, which reduced readability. Writing the entire function definition on a single line made the code harder to understand and maintain. The refactored version separates the function logic onto multiple lines and adds proper spacing to improve clarity. A blank line was included between the function definition and the function call to enhance visual structure. These stylistic changes make the code easier to read, debug, and extend in the future. Adhering to PEP 8 standards is especially important in collaborative environments where code readability matters. The functionality remains unchanged while the code quality is significantly improved.

### Task 3: Code Clarity Improvement

**Prompt:** Use AI to improve code readability without changing its functionality.

### Sample Input Code(Contains Syntactical or Logical Errors):

```
def f(x,y):
return x-y*2
print(f(10,3))
```

```
  File "/tmp/ipython-input-1651372911.py", line 2
    return x-y*2
    ^
IndentationError: expected an indented block after function definition on line 1
```

### Corrected Code:

```
def calculate_difference(value1, value2):
    return value1 - value2 * 2


print(calculate_difference(10, 3))
```

```
4
```

### Explanation:

The original code suffered from poor readability due to unclear naming and incorrect indentation. The function name f provided no indication of the operation being performed. Similarly, the variable names x and y did not describe their purpose, making the logic harder to understand. Missing indentation caused a syntax error, preventing execution. In the improved version, the function name clearly describes the operation being performed, and the parameter names are more descriptive. Proper indentation and spacing were added to follow Python syntax rules. These changes make the code self-explanatory and reduce the need for additional comments. Improving clarity is essential for debugging, collaboration, and long-term maintenance. The functionality and output remain exactly the same, ensuring correctness while improving readability and comprehension.

### Task 4: Structural Refactoring

**Prompt:** Refactor the following repetitive print statements into a reusable Python function.

### Sample Input Code(Contains Syntactical or Logical Errors):

```
print("Hello Ram")
print("Hello Sita")
print("Hello Ravi")
```

```
Hello Ram
Hello Sita
Hello Ravi
```

### Corrected Code:

```
def greet(name):
    print(f"Hello {name}")

greet("Ram")
greet("Sita")
greet("Ravi")
```

```
Hello Ram
Hello Sita
Hello Ravi
```

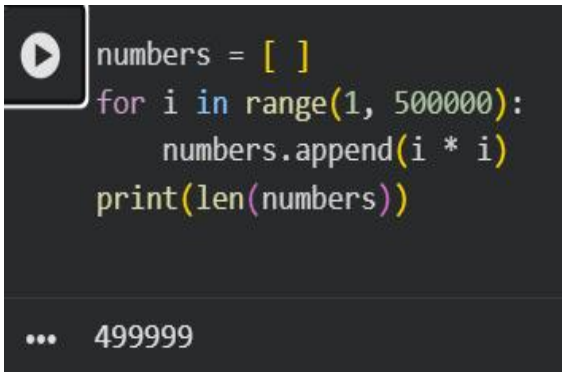
**Explanation:**

The original code contained repeated print statements that performed nearly identical operations. This repetition increases code length and makes maintenance more difficult. If the greeting message needed to be updated, each line would have to be changed individually. By introducing a reusable function, the greeting logic is centralized in one place. This approach follows the DRY principle and improves modularity. The function makes the code easier to extend, as additional names can be greeted with a single function call. The use of formatted strings improves readability and flexibility. Structural refactoring such as this enhances scalability and reduces redundancy. This design is considered a best practice in software engineering and leads to cleaner, more maintainable code.

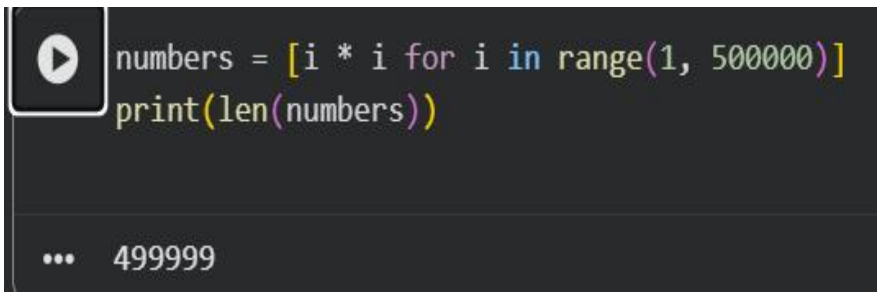
**Task 5: Efficiency Enhancement**

**Prompt:** Optimize the following Python code to improve performance while maintaining the same output.

**Sample Input Code(Contains Syntactical or Logical Errors):**




**Corrected Code:**



## Explanation:

The original code used a traditional for loop combined with the append method to build a list, which is slower and more verbose for large datasets. This approach requires multiple method calls and additional overhead. The optimized version replaces the loop with a list comprehension, which is more efficient and concise. List comprehensions are faster because they are optimized internally by Python's interpreter. The optimized code performs the same computation while reducing execution time and improving readability. Fewer lines of code also reduce the chance of errors. This optimization is particularly beneficial when handling large amounts of data. The output remains unchanged, ensuring correctness while achieving better performance and cleaner code.

## Conclusion:

This laboratory experiment theoretically demonstrates how artificial intelligence can be applied as an effective tool for improving software quality. The use of AI for code review enables systematic detection of syntax errors, logical flaws, and inefficient programming patterns. By enforcing coding standards such as PEP 8, AI contributes to uniformity and professionalism in Python programs. The experiment highlights the theoretical importance of readability and maintainability in long-term software development. Structural refactoring illustrates how abstraction and modularity reduce redundancy and enhance reusability. Efficiency optimization emphasizes the role of computational thinking in selecting appropriate language constructs. The study reinforces core software engineering principles such as clarity, scalability, and robustness. AI-assisted feedback supports developers in understanding errors rather than merely correcting them. Theoretical evaluation of AI suggestions ensures human oversight in decision-making. This balance prevents blind dependence on automated tools. The lab also demonstrates how prompt engineering influences the quality of AI output. Proper prompts lead to targeted and meaningful improvements. Overall, the experiment validates AI as a supportive, not substitutive, component of programming practice. Such integration promotes disciplined coding habits. Theoretical knowledge gained through this lab is applicable to real-world software engineering scenarios.