Name:Swetha Dixit

Course:AI Coding

Hall-Ticket:2303A52415

Batch:33

Assignment:10.2

Task:1

**Task Description -1(Error Detection and Correction)**

## Task:

Use AI to analyze a Python script and correct all syntax and logical errors.

## Sample Input Code:

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

## Expected Output-1:

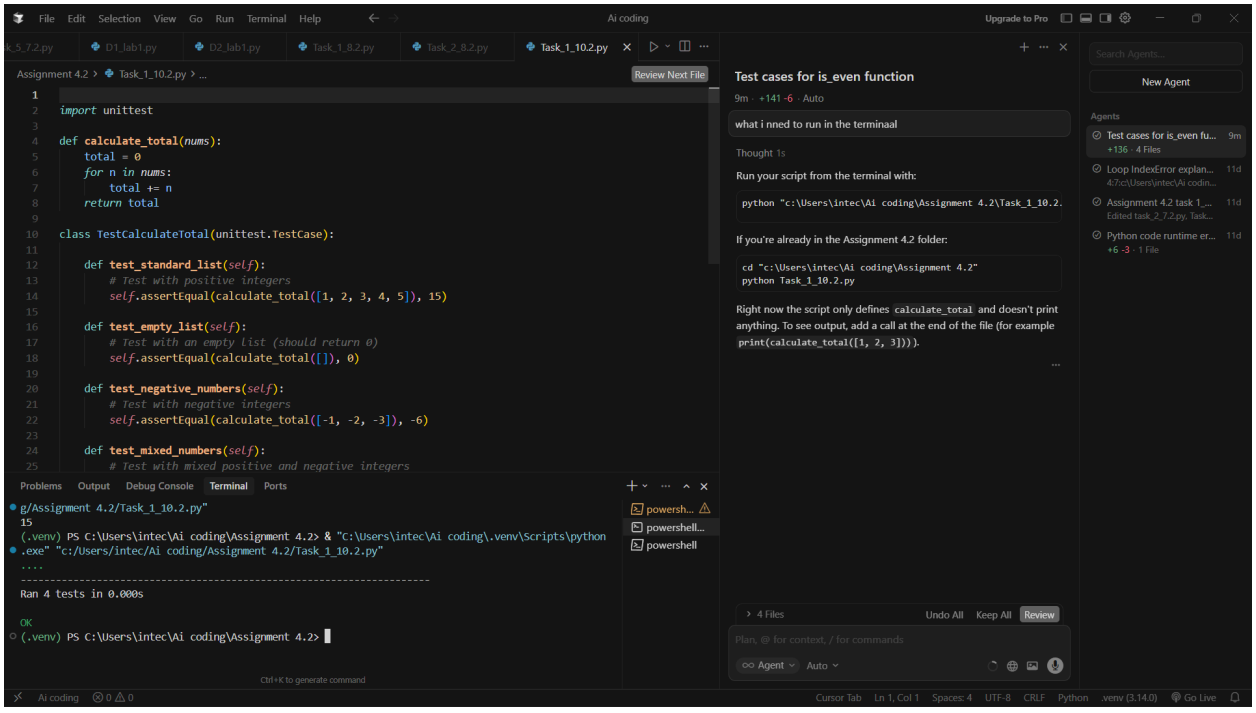
Corrected and executable Python code with

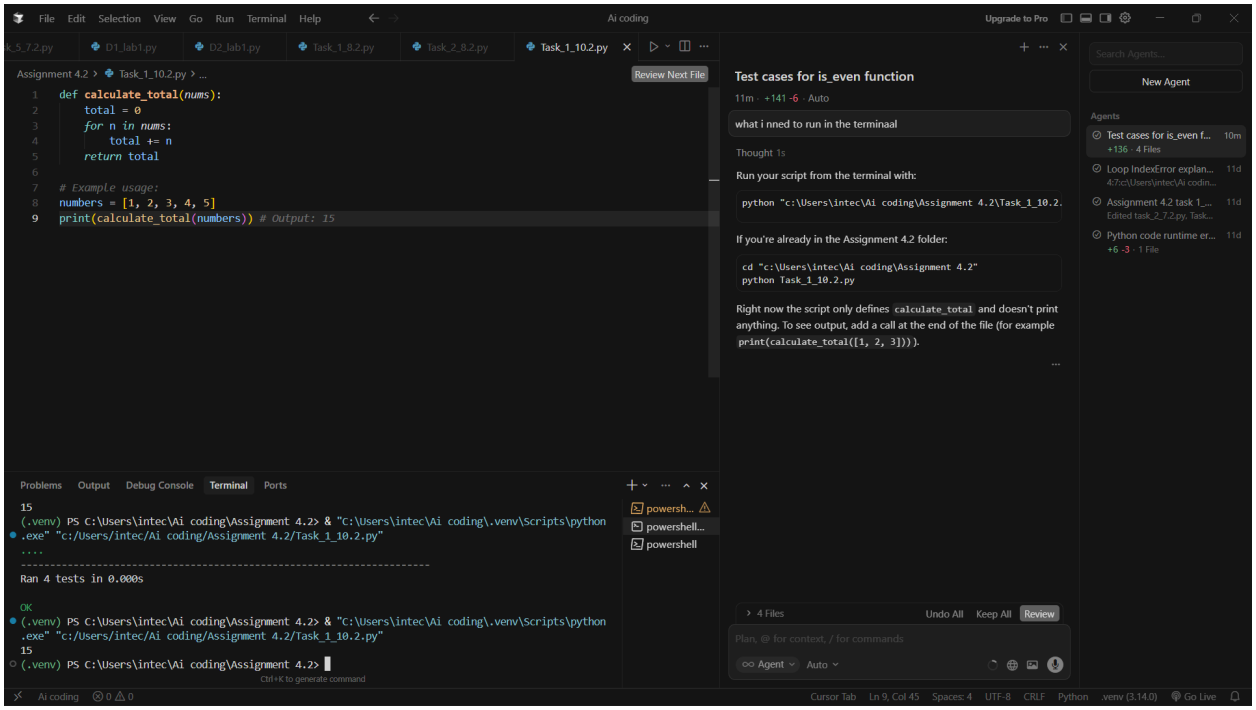
brief explanations of the identified syntax and logic errors.

Python Code:

```
import unittest



def calculate_total(nums):

    total = 0

    for n in nums:

        total += n

    return total



class TestCalculateTotal(unittest.TestCase):



    def test_standard_list(self):

        # Test with positive integers

        self.assertEqual(calculate_total([1, 2, 3, 4, 5]), 15)



    def test_empty_list(self):

        # Test with an empty list (should return 0)

        self.assertEqual(calculate_total([]), 0)
```

```
    def test_negative_numbers(self):
        # Test with negative integers
        self.assertEqual(calculate_total([-1, -2, -3]), -6)

    def test_mixed_numbers(self):
        # Test with mixed positive and negative integers
        self.assertEqual(calculate_total([10, -5, 2]), 7)

# Running the tests
if __name__ == '__main__':
    unittest.main(argv=['first-arg-is-ignored'], exit=False)
```






### Analysis of Errors

# Here is a breakdown of the specific syntax and logical errors identified and corrected:

### 1. Syntax Errors (Prevent the code from running)

- **Missing Colons:** The original code lacked colons (`:`) at the end of the function definition (`def ...`) and the loop statement (`for ...`). In Python, control flow statements must end with a colon.
    - *Fix:* Added `:` to lines 1 and 3.

### 2. Logical Errors (Code runs but fails or gives wrong results)

- **Undefined Variable in Return Statement:** The function attempted to `return total`, but the variable used to accumulate the sum was named `sum`. This would cause a `NameError` because `total` did not exist within the function's scope.
    - *Fix:* Updated the return statement to match the accumulator variable.
- **Shadowing Built-in Functions (Best Practice):** The original code used `sum` as a variable name. While valid, this "shadows" Python's built-in `sum()` function, which is bad practice and can lead to bugs later.
    - *Fix:* Renamed the variable from `sum` to `total`.

# Task:2**Task Description -2(Code Style Standardization)**

# Task:

Use AI to refactor Python code to comply with standard coding style guidelines.

# Sample Input Code:

```
def findSum(a,b):return a+b

print(findSum(5,10))
```

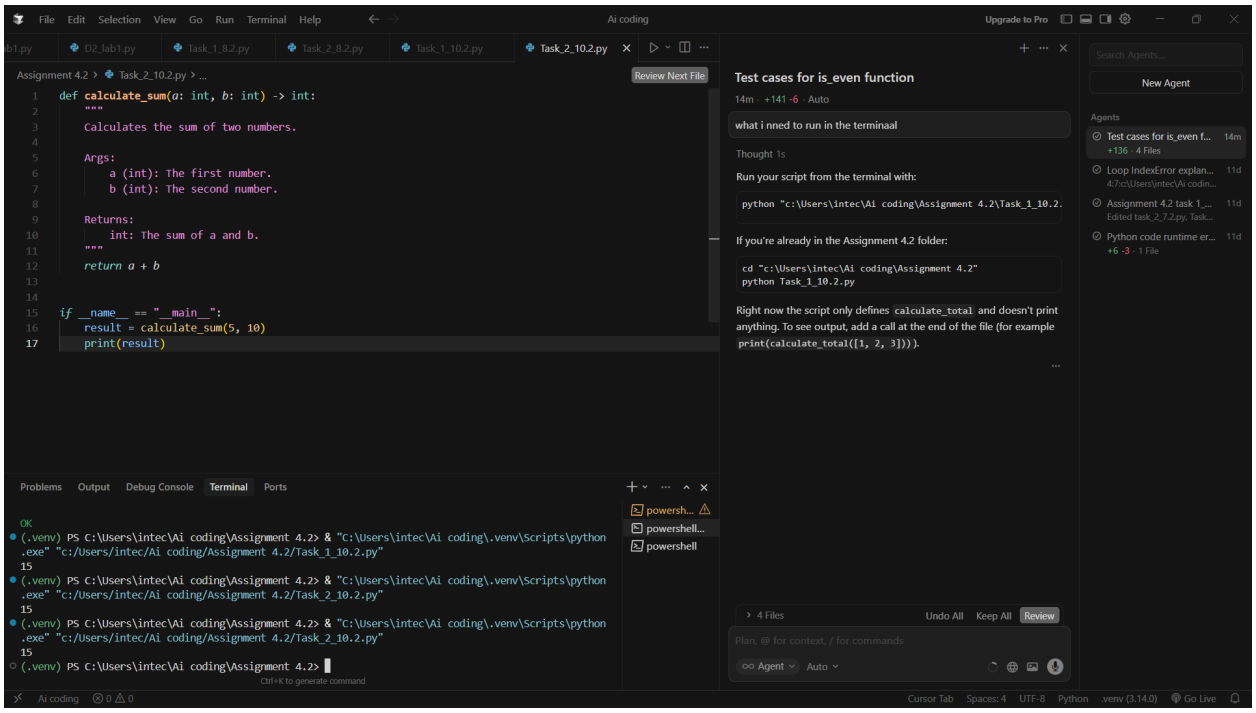
## Expected Output-2:

Well-structured, consistently formatted Python code following standard style conventions.

Python code:

```
def calculate_sum(a: int, b: int) -> int:
    """
    Calculates the sum of two numbers.

    Args:
        a (int): The first number.
        b (int): The second number.

    Returns:
        int: The sum of a and b.
    """
    return a + b
```

```python
if __name__ == "__main__":

    result = calculate_sum(5, 10)

    print(result)
```




## Key Style Improvements Applied (PEP 8)

- **Naming Convention (Snake Case):** Changed `findSum` (camelCase) to `calculate_sum` (snake_case). In Python, functions and variables should be lowercase with underscores.
- **Whitespace and Indentation:**
    - Moved the function body (`return a + b`) to a new line with a 4-space indentation.
    - Added spaces around operators (`+`) and after commas (e.g., `a, b`) to improve readability.

- **Documentation (Docstrings):** Added a docstring (`"""..."""`) to explain the function's purpose, arguments, and return value.
- **Type Hinting:** Added `: int` and `-> int` annotations. While not strictly mandatory in PEP 8, they are highly recommended in modern Python for clarity and debugging.
- **Execution Block:** Wrapped the executable code in `if __name__ == "__main__":`. This prevents the code from running automatically if this script is imported as a module into another program.

# Task:3

**Task Description -3(Code Clarity Improvement)**

## Task:

Use AI to improve code readability without changing its functionality.

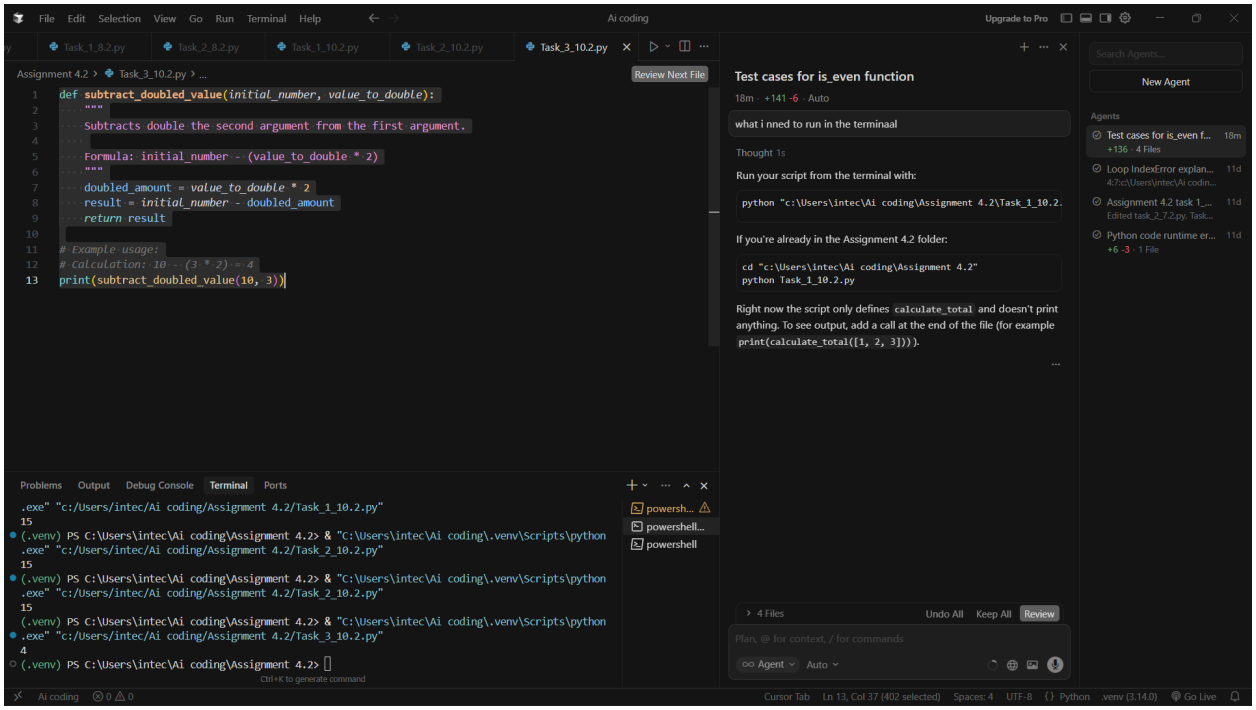
# Sample Input Code:

```
def f(x,y):

return x-y*2

print(f(10,3))
```

## Expected Output-3:

Python code rewritten with meaningful function and variable names, proper indentation, and improved clarity.

```
def subtract_doubled_value(initial_number, value_to_double):
    """
    Subtracts double the second argument from the first argument.

    Formula: initial_number - (value_to_double * 2)
    """
    doubled_amount = value_to_double * 2
    result = initial_number - doubled_amount
    return result


# Example usage:
# Calculation: 10 - (3 * 2) = 4
print(subtract_doubled_value(10, 3))
```




## Key Improvements Made

- **Meaningful Naming:**
  - Changed the function name `f` to `subtract_doubled_value` so the user knows exactly what the function does without reading the logic.
  - Changed `x` and `y` to `initial_number` and `value_to_double`. This prevents confusion about which number is being subtracted and which is being doubled.
- **Intermediate Variables:**
  - Introduced `doubled_amount` and `result`. Breaking the calculation into steps makes the logic easier to follow during debugging.
- **Spacing and Indentation:**
  - Added spaces around operators (`*`, `-`) for visual breathing room.
  - Corrected the indentation of the `return` statement (standard Python indentation is 4 spaces).

- **Documentation:**
    - Added a docstring to explain the formula being used.

Task:4**Task Description -4(Structural Refactoring)**

# Task:

Use AI to refactor repetitive code into reusable functions.

# Sample Input Code:

print("Hello Ram")

print("Hello Sita")

print("Hello Ravi")

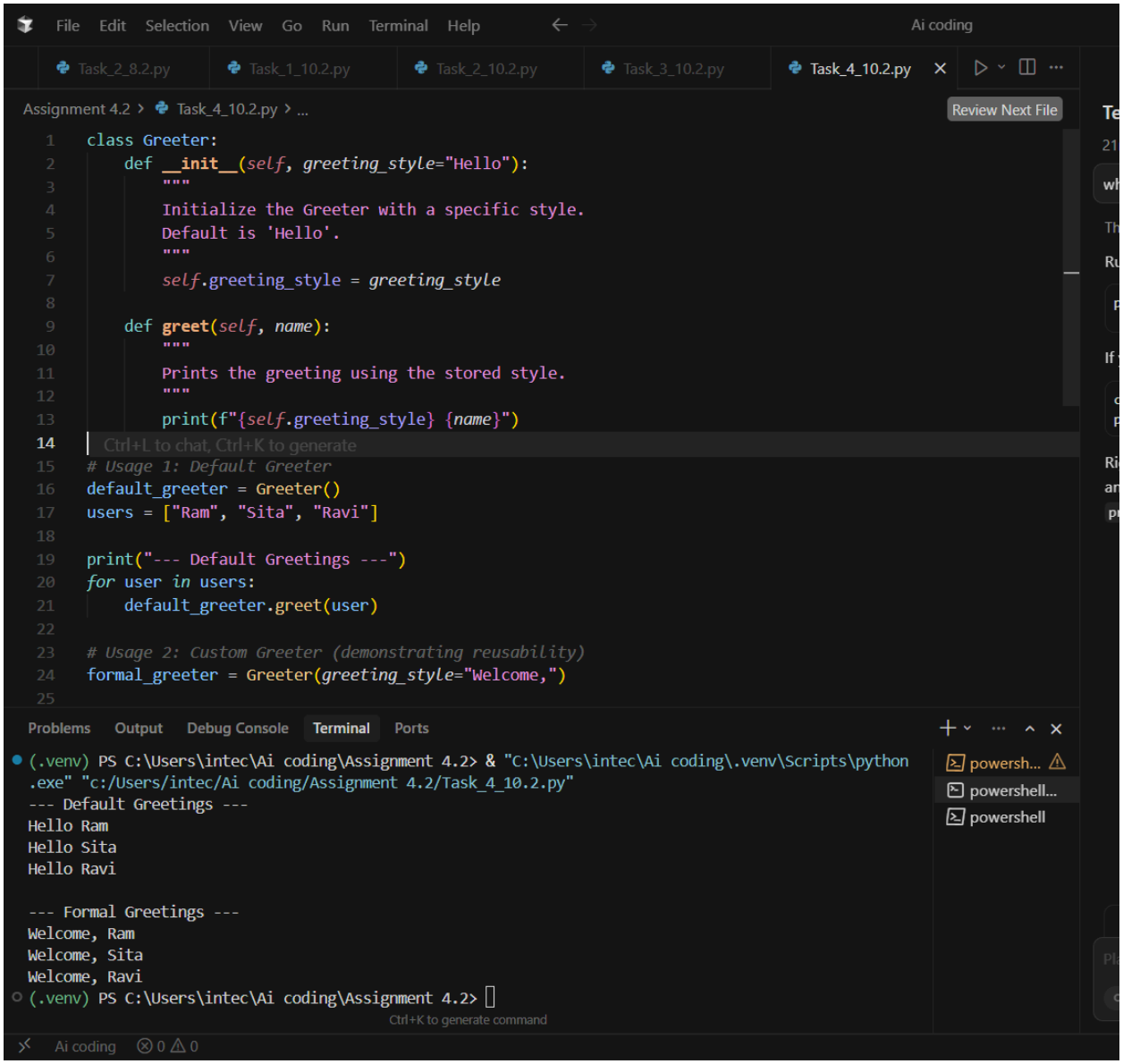
**Expected Output-4:**

Modular Python code using reusable functions to eliminate repetition

Python code:

```
class Greeter:

    def __init__(self, greeting_style="Hello"):

        """

        Initialize the Greeter with a specific style.

        Default is 'Hello'.

        """

        self.greeting_style = greeting_style
```

```
    def greet(self, name):
        """
        Prints the greeting using the stored style.
        """
        print(f"{self.greeting_style} {name}")

# Usage 1: Default Greeter
default_greeter = Greeter()
users = ["Ram", "Sita", "Ravi"]

print("--- Default Greetings ---")
for user in users:
    default_greeter.greet(user)

# Usage 2: Custom Greeter (demonstrating reusability)
formal_greeter = Greeter(greeting_style="Welcome,")

print("\n--- Formal Greetings ---")
for user in users:
    formal_greeter.greet(user)
```




**State Retention:** The `Greeter` remembers the `greeting_style`. You don't have to pass "Hello" or "Welcome" every time you call the function.

**Extensibility:** You could easily add methods like `change_style()` or a counter to track how many people were greeted without changing the external code structure.

Task:5**Task Description -5(Efficiency Enhancement)**

# Task:

Use AI to optimize Python code for better performance.

# Sample Input Code:

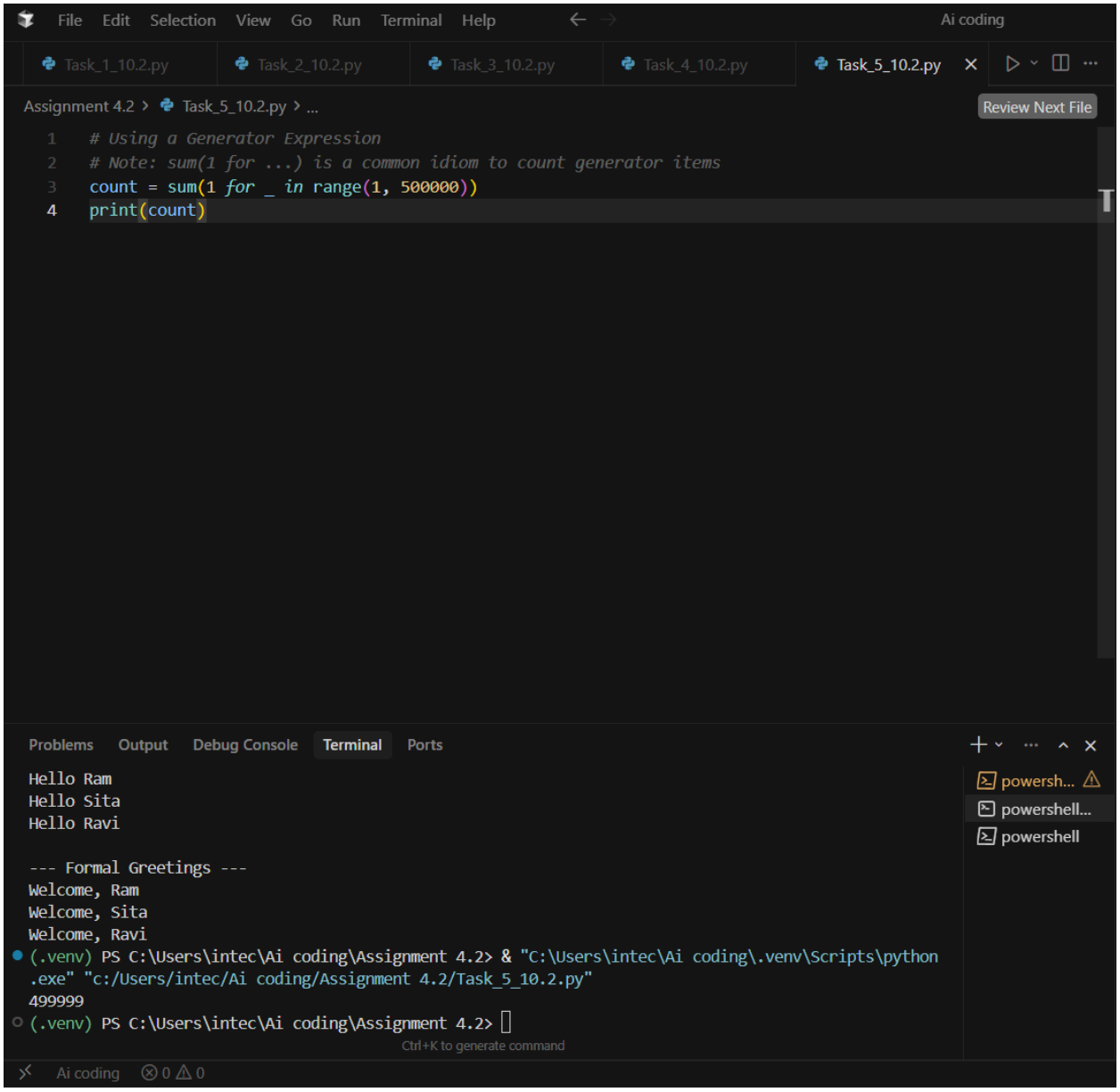
```
numbers = [ ]

for i in range(1, 500000):

        numbers.append(i * i)

print(len(numbers))
```

## Expected Output-5:

Optimized Python code that achieves the same result with improved performance.

Python code:

```
# Using a Generator Expression

# Note: sum(1 for ...) is a common idiom to count generator items

count = sum(1 for _ in range(1, 500000))

print(count)
```




**Original Code:** The `.append()` method is called 499,999 times. Each call involves a small amount of overhead that adds up.

**List Comprehension:** Pre-allocates memory chunks and avoids the repeated attribute lookup for `.append`.