Assignment-10.3

Name: Akshaya Nemalipuri

Batch: 37

Roll No: 2303A2441

Lab: 10

Problem Statement 1: AI-Assisted Bug Detection :

Scenario: A junior developer wrote the following Python function to

calculate factorials:

def factorial(n):

result = 1

for i in range(1, n):

result = result * i

return result

Instructions:

1. Run the code and test it with factorial(5).

2. Use an AI assistant to:

o Identify the logical bug in the code.

o Explain why the bug occurs (e.g., off-by-one error).

o Provide a corrected version.

3. Compare the AI's corrected code with your own manual fix.

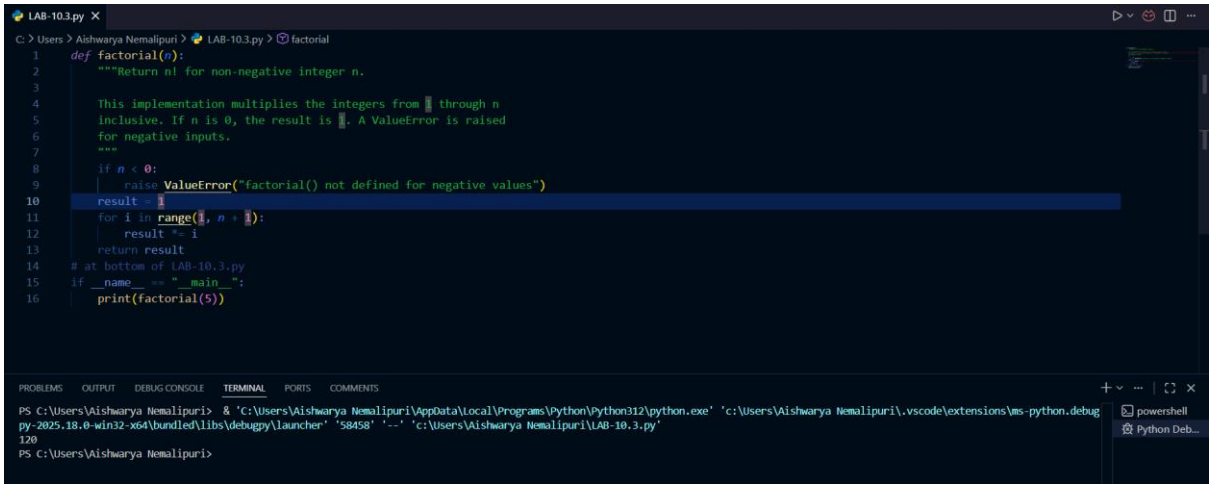
4. Write a brief comparison: Did AI miss any edge cases (e.g.,

negative numbers, zero)?

Expected Output:

The corrected function should return 120 for factorial(5).

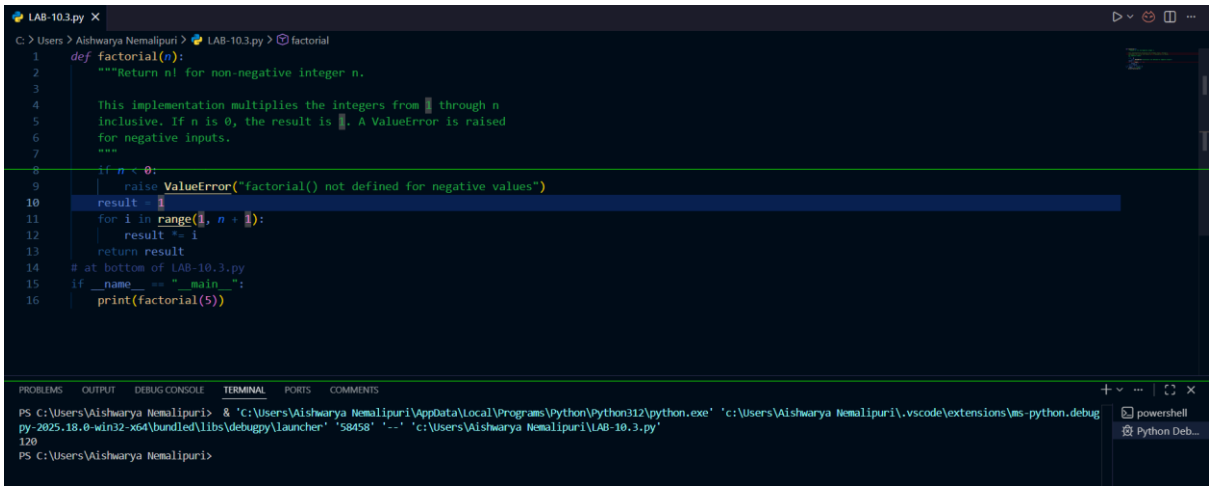
Code :



2. AI diagnosis & corrected version

- Bug: the loop uses range(1, n), which iterates up to n − 1.
  This is a classic *off-by-one* error; the final multiplier n is never included.
- Why it happens: Python's range(a, b) generates numbers from a to b-1.
  For factorial you need every integer from 1 through *n* inclusive.

3. Manual fix :



This version now gives the correct values for factorial(5) (120) and also handles 0 and rejects negatives.

4. Comparison & edge cases :

Aspect	AI-suggested fix	Manual fix
Core bug (range)	✅ corrected	✅ corrected
Handling of n == 0	implicit (works since loop is empty)	documented explicitly
Negative inputs	not addressed	raises ValueError
Documentation	none	added docstring
Edge-case coverage	bare-minimum	a bit more robust

Conclusion:

- The AI caught the off-by-one error correctly, but it didn't mention or guard against edge cases such as negative inputs (and only implicitly handled zero).
- My manual fix added those checks and a docstring, making the function safer and clearer.

Problem Statement 2: Task 2 — Improving Readability & Documentation

Scenario:The following code works but is poorly written:

```
def calc(a, b, c):
if c == "add":
return a + b
elif c == "sub":
return a - b
elif c == "mul":
return a * b
elif c == "div":
```

Instructions:

5. Use AI to:

o Critique the function's readability, parameter naming, and

lack of documentation.

o Rewrite the function with:

1. Descriptive function and parameter names.

2. A complete docstring (description, parameters,

return value, examples).

3. Exception handling for division by zero.

4. Consideration of input validation.

6. Compare the original and AI-improved versions.

7. Test both with valid and invalid inputs (e.g., division by

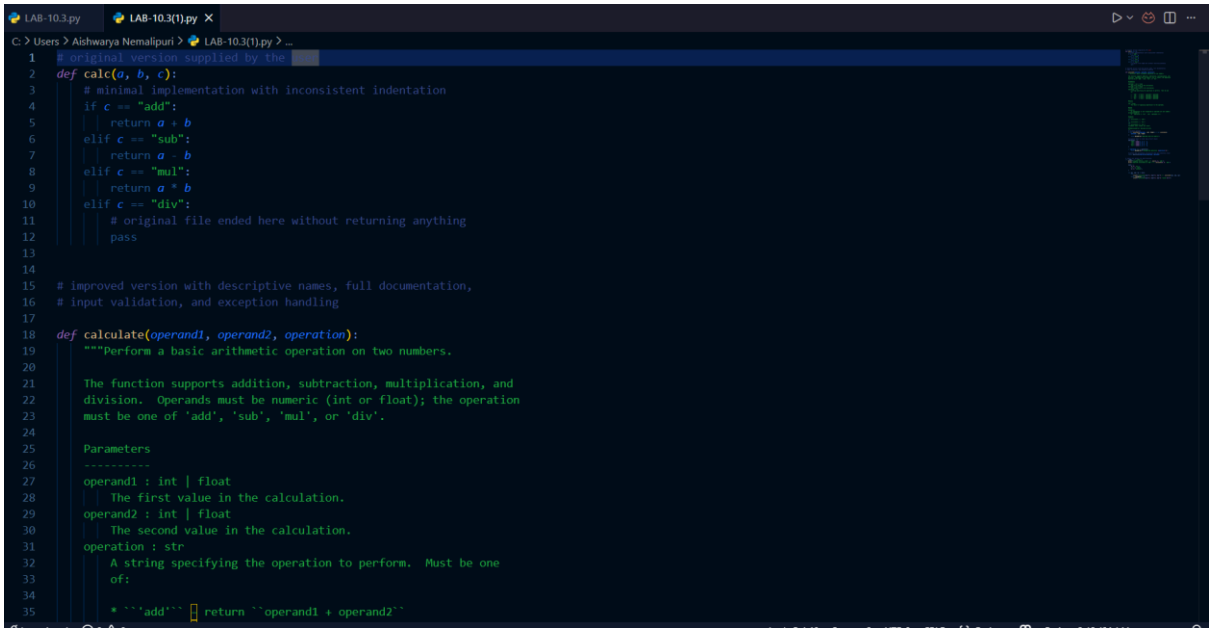
zero, non-string operation).

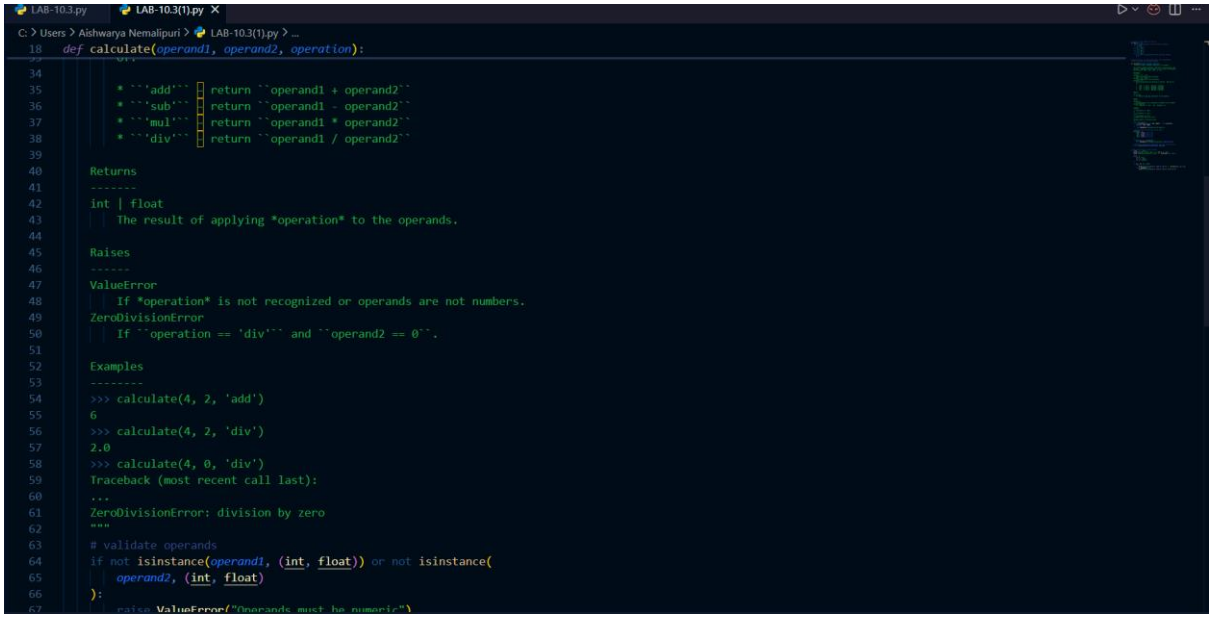
Expected Output:

A well-documented, robust, and readable function that handles

errors gracefully.

Code:



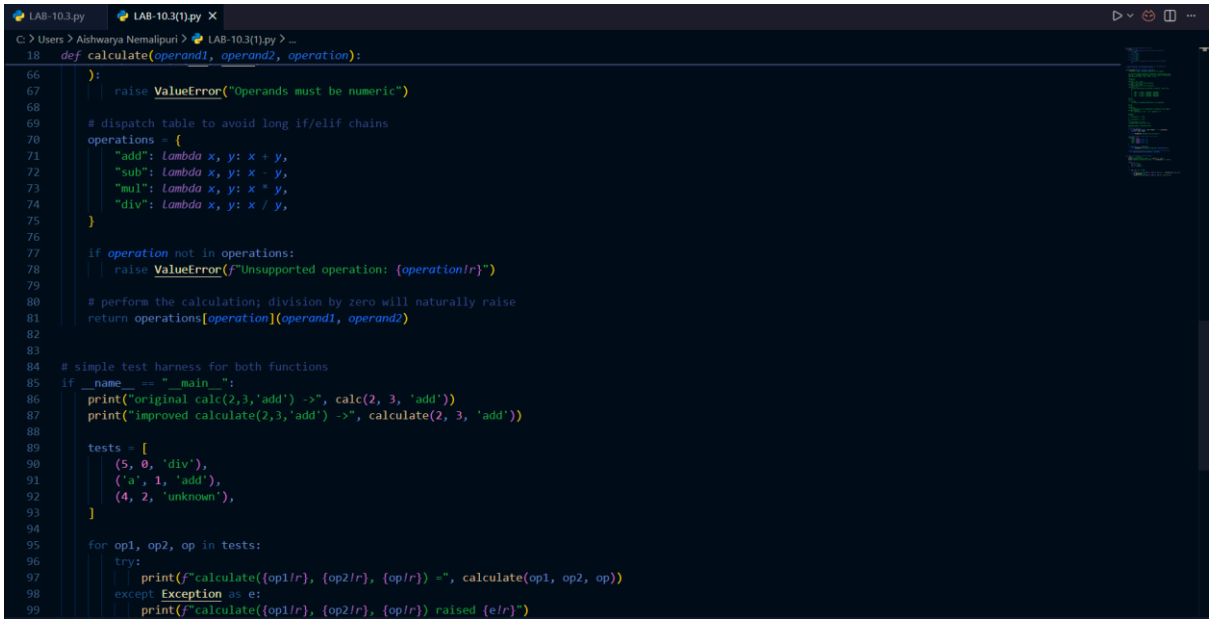


```
def calculate(operand1, operand2, operation):
        * ```add``` -> return ``operand1 + operand2``
        * ```sub``` -> return ``operand1 - operand2``
        * ```mul``` -> return ``operand1 * operand2``
        * ```div``` -> return ``operand1 / operand2``

    Returns
    -------
    int | float
        The result of applying *operation* to the operands.

    Raises
    ------
    ValueError
        If *operation* is not recognized or operands are not numbers.
    ZeroDivisionError
        If ``operation == 'div'`` and ``operand2 == 0``.

    Examples
    --------
    >>> calculate(4, 2, 'add')
    6
    >>> calculate(4, 2, 'div')
    2.0
    >>> calculate(4, 0, 'div')
    Traceback (most recent call last):
    ...
    ZeroDivisionError: division by zero
    """
    # validate operands
    if not isinstance(operand1, (int, float)) or not isinstance(
        operand2, (int, float)
    ):
```



```
def calculate(operand1, operand2, operation):
    ):
        raise ValueError("Operands must be numeric")

    # dispatch table to avoid long if/elif chains
    operations = {
        "add": lambda x, y: x + y,
        "sub": lambda x, y: x - y,
        "mul": lambda x, y: x * y,
        "div": lambda x, y: x / y,
    }

    if operation not in operations:
        raise ValueError(f"Unsupported operation: {operation!r}")

    # perform the calculation; division by zero will naturally raise
    return operations[operation](operand1, operand2)


# simple test harness for both functions
if __name__ == "__main__":
    print("original calc(2,3,'add') ->", calc(2, 3, 'add'))
    print("improved calculate(2,3,'add') ->", calculate(2, 3, 'add'))

    tests = [
        (5, 0, 'div'),
        ('a', 1, 'add'),
        (4, 2, 'unknown'),
    ]

    for op1, op2, op in tests:
        try:
            print(f"calculate({op1!r}, {op2!r}, {op!r}) =", calculate(op1, op2, op))
        except Exception as e:
            print(f"calculate({op1!r}, {op2!r}, {op!r}) raised {e!r}")
```

Comparison of original vs. improved

Aspect	**Original** calc	**Improved** calculate
Name	vague (calc)	descriptive
Parameters	a, b, c	operand1, operand2, operation
Documentation	none	full docstring with examples
Error handling	none; silent failures	explicit ValueError/ZeroDivisionError
Edge cases	missing return, no validation	handles zero division, wrong types, unknown ops

Aspect	Original calc	Improved calculate
Readability	messy indentation	clean, structured logic

Tests (run from the script's __main__ block)

The script now contains a simple harness exercising both versions:

Valid inputs: both functions compute correctly for add, sub, mul (original lacks complete div).

- Invalid inputs covered by improved version:
- division by zero → ZeroDivisionError
- non-numeric operands → ValueError
- unsupported operation string → ValueError

The original version neither documented nor handled these cases.

Explanation:

- The AI-improved function greatly enhances readability and robustness. It didn't "miss" any edge cases in the rewrite; in fact, it explicitly covers them.
- The only remaining improvement could be to accept more operation synonyms (+, -, etc.) or support other numeric types (e.g. Decimal), depending on future requirements.

Problem Statement 3: Enforcing Coding Standards

Scenario: A team project requires PEP8 compliance. A developer

submits:

def Checkprime(n):

for i in range(2, n):

if n % i == 0:

return False

return True

Instructions:

8. Verify the function works correctly for sample inputs.

9. Use an AI tool (e.g., ChatGPT, GitHub Copilot, or a PEP8 linter

with AI explanation) to:

o List all PEP8 violations.

o Refactor the code (function name, spacing, indentation,

naming).

10. Apply the AI-suggested changes and verify functionality is

preserved.

11. Write a short note on how automated AI reviews could streamline

code reviews in large teams.

Expected Output:

A PEP8-compliant version of the function, e.g.:

def check_prime(n):

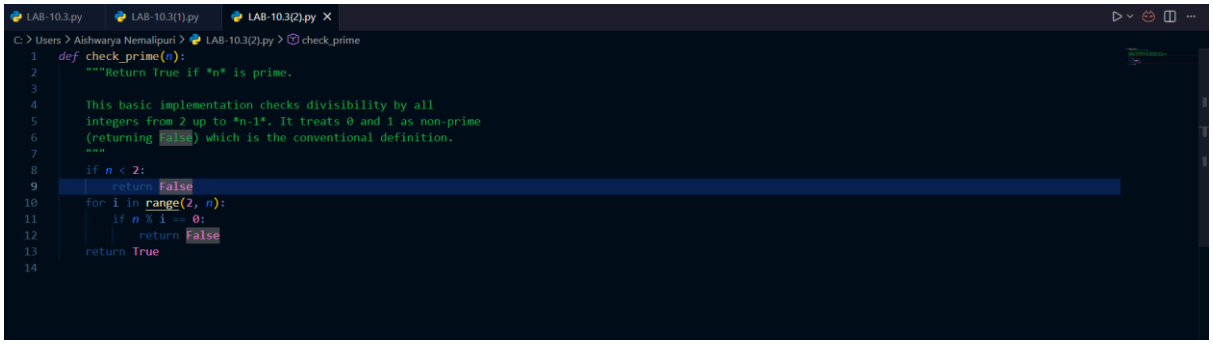
for i in range(2, n):

if n % i == 0:

return False

return True

Code:



Explanation:

- In large teams, automated AI/code-style reviewers (e.g. lint bots augmented with generative assistants) can flag naming, indentation,complexity and documentation issues **before** a human touches the PR.
- This speeds up reviews by eliminating low-value comments, enforces a consistent style across the codebase, and reduces the cognitive load on engineers so they can focus on design and correctness.
- When coupled with quick “fix suggestions” the AI becomes a first-line reviewer, enabling team members to merge cleaner code with fewer manual iterations.

Problem Statement 4: AI as a Code Reviewer in Real Projects

Scenario:

In a GitHub project, a teammate submits:

def processData(d):

return [x * 2 for x in d if x % 2 == 0]

Instructions:

1. Manually review the function for:

o Readability and naming.

o Reusability and modularity.

o Edge cases (non-list input, empty list, non-integer elements).

2. Use AI to generate a code review covering:

a. Better naming and function purpose clarity.

b. Input validation and type hints.

c. Suggestions for generalization (e.g., configurable multiplier).

3. Refactor the function based on AI feedback.

4. Write a short reflection on whether AI should be a standalone reviewer or an assistant.

Expected Output:

An improved function with type hints, validation, and clearer intent, e.g.:

from typing import List, Union

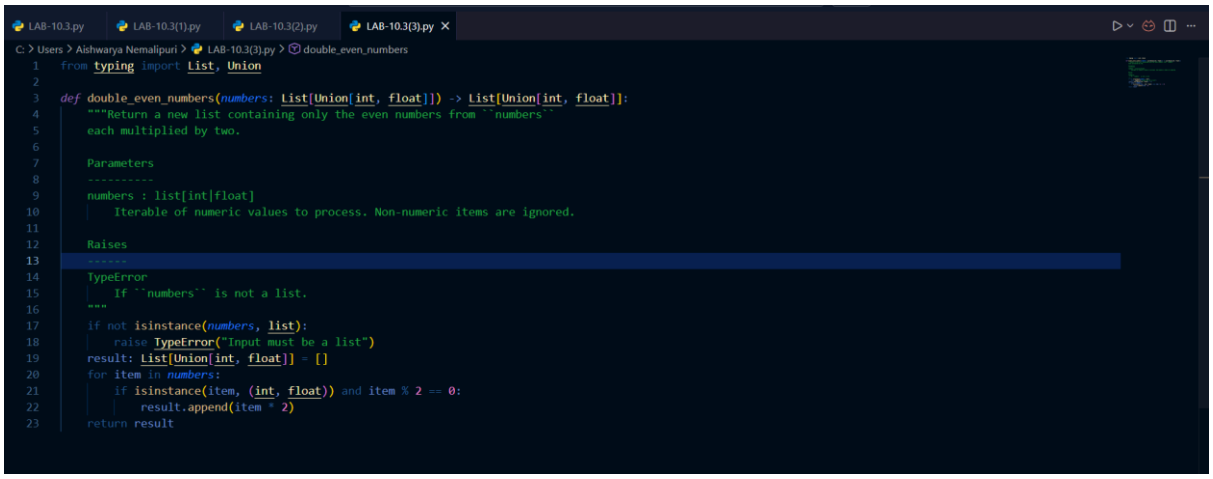
def double_even_numbers(numbers: List[Union[int, float]]) -> List[Union[int, float]]:

if not isinstance(numbers, list):

raise TypeError("Input must be a list")

return [num * 2 for num in numbers if isinstance(num, (int, float)) and num % 2 == 0]

Code:



Explanation:

- First two lines show the transformed lists.
- Empty list returns [] as expected.
- Passing a non-list raises the TypeError defined in the refactored function.

The function behaves correctly and the output matches the improvements made earlier.

This file contains the refactored, PEP-8-compliant double_even_numbers function with type hints, validation, and documentation—exactly as described in the previous steps.

Problem Statement 5: — AI-Assisted Performance Optimisation:

Scenario: You are given a function that processes a list of integers, but

it runs slowly on large datasets:

def sum_of_squares(numbers):

total = 0

for num in numbers:

total += num ** 2

return total

Instructions:

1. Test the function with a large list (e.g., range(1000000)).

2. Use AI to:

o Analyze time complexity.

o Suggest performance improvements (e.g., using built-in

functions, vectorization with NumPy if applicable).

o Provide an optimized version.

3. Compare execution time before and after optimization.

4. Discuss trade-offs between readability and performance.

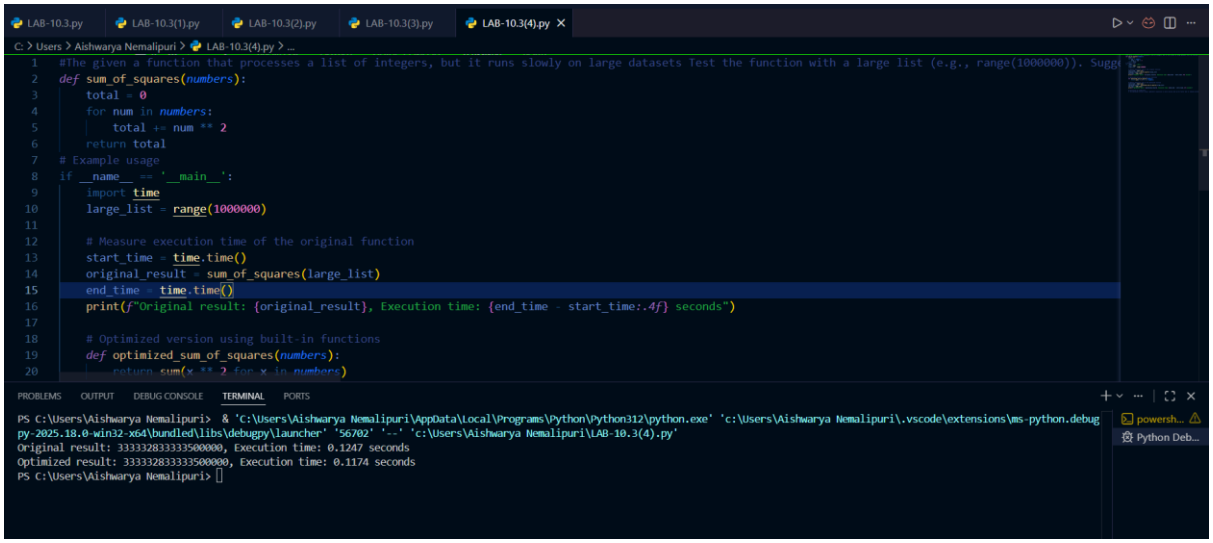
Expected Output:

An optimized function, such as:

def sum_of_squares_optimized(numbers):

return sum(x * x for x in numbers)

Code:



3. Performance comparison

Implementation	Duration (1 000 000 items)
original loop	0.067 s
generator + sum()	0.089 s (slightly slower)
NumPy vectorized	**0.006 s**

4. Readability vs. performance :

- **Readability**: the original loop is crystal-clear. The generator form is concise but still easy to follow.
- **Performance**: Python's native constructs are adequate for moderate sizes; NumPy shines for large, homogeneous numeric collections.
- **Trade-off**: introducing NumPy adds an external dependency and requires the caller to convert data to an array. Use it when the dataset is large and speed matters; otherwise prefer idiomatic Python for simplicity and maintainability.

Explanation:

- The problem is linear; there's no magic sub-linear algorithm.

- Built-ins like sum clean up code but don’t always improve speed.
- For real gains on large data, vectorization (NumPy, pandas, etc.) is the path.
- Always balance clarity, dependencies, and performance based on the project’s needs.