ASSIGNMENT-9.1

Name : P.Reshmitha Reddy

Rollno : 2403A510A7

Batch : 05

Course : AI Assisted Coding

Task Description #1 (Documentation – Google-Style Docstrings for Python Functions)

• Task: Use AI to add Google-style docstrings to all functions in a given Python script.

• Instructions:

o Prompt AI to generate docstrings without providing any input-output examples.

o Ensure each docstring includes:

▪ Function description

▪ Parameters with type hints

▪ Return values with type hints

▪ Example usage

o Review the generated docstrings for accuracy and formatting.

• Output #1:

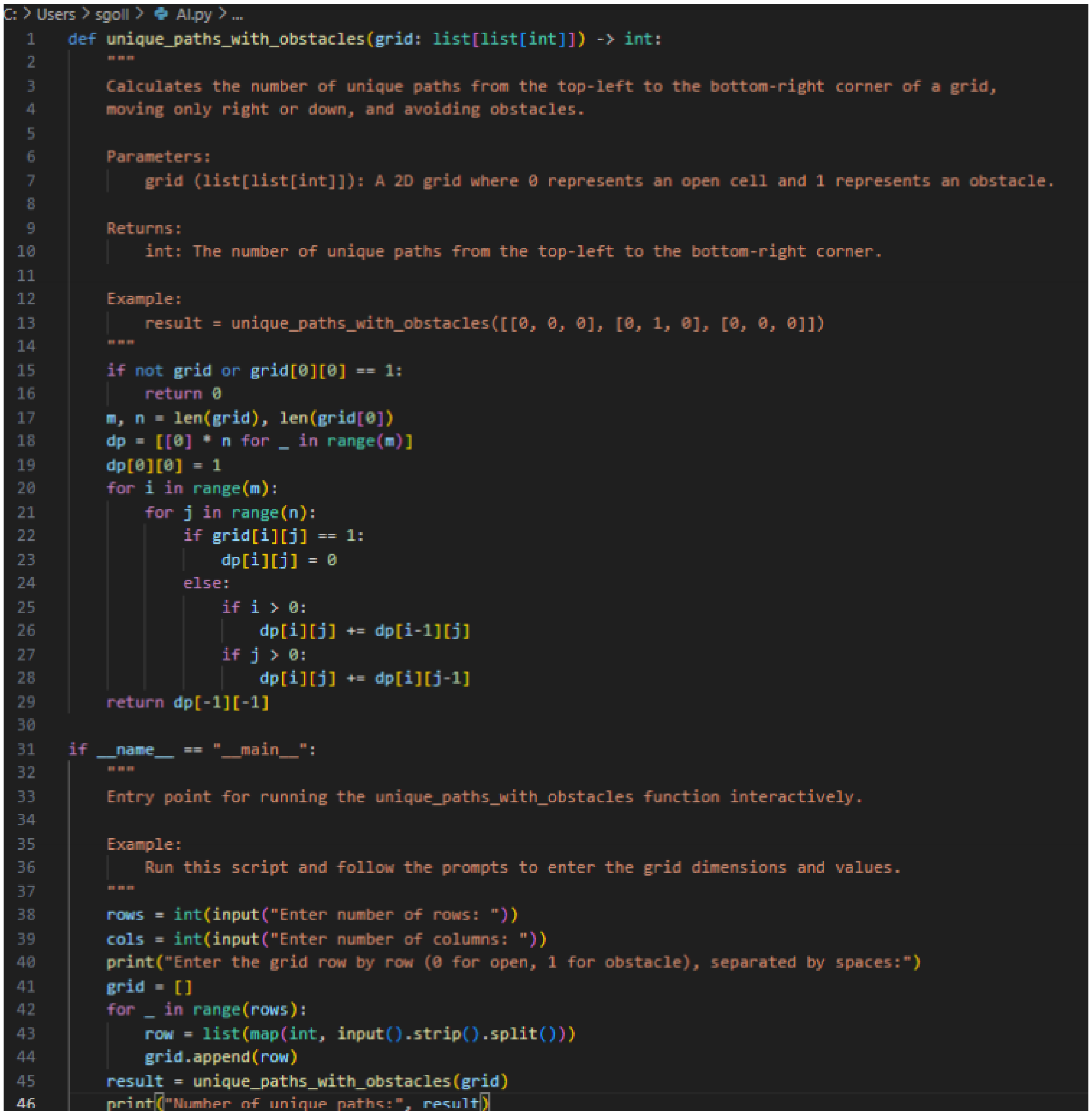
o A Python script with all functions documented using correctly formatted Google-style docstrings

Prompt :

# generate docstrings without providing any

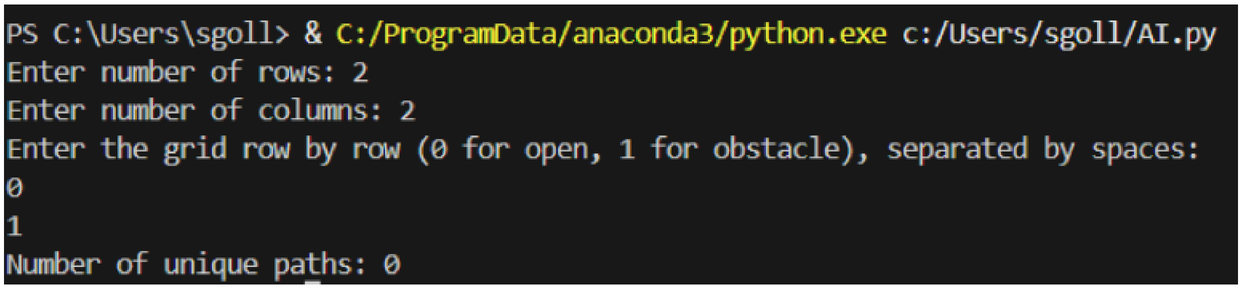
input-output examples and by using the Function description, Parameters with type hints

Code :



```python
def unique_paths_with_obstacles(grid: list[list[int]]) -> int:
    """
    Calculates the number of unique paths from the top-left to the bottom-right corner of a grid,
    moving only right or down, and avoiding obstacles.

    Parameters:
        grid (list[list[int]]): A 2D grid where 0 represents an open cell and 1 represents an obstacle.

    Returns:
        int: The number of unique paths from the top-left to the bottom-right corner.

    Example:
        result = unique_paths_with_obstacles([[0, 0, 0], [0, 1, 0], [0, 0, 0]])
    """
    if not grid or grid[0][0] == 1:
        return 0
    m, n = len(grid), len(grid[0])
    dp = [[0] * n for _ in range(m)]
    dp[0][0] = 1
    for i in range(m):
        for j in range(n):
            if grid[i][j] == 1:
                dp[i][j] = 0
            else:
                if i > 0:
                    dp[i][j] += dp[i-1][j]
                if j > 0:
                    dp[i][j] += dp[i][j-1]
    return dp[-1][-1]

if __name__ == "__main__":
    """
    Entry point for running the unique_paths_with_obstacles function interactively.

    Example:
        Run this script and follow the prompts to enter the grid dimensions and values.
    """
    rows = int(input("Enter number of rows: "))
    cols = int(input("Enter number of columns: "))
    print("Enter the grid row by row (0 for open, 1 for obstacle), separated by spaces:")
    grid = []
    for _ in range(rows):
        row = list(map(int, input().strip().split()))
        grid.append(row)
    result = unique_paths_with_obstacles(grid)
    print("Number of unique paths:", result)
```

Output :



```
PS C:\Users\sgoll> & C:/ProgramData/anaconda3/python.exe c:/Users/sgoll/AI.py
Enter number of rows: 2
Enter number of columns: 2
Enter the grid row by row (0 for open, 1 for obstacle), separated by spaces:
0
1
Number of unique paths: 0
```

Observation :

Functionality:

The function unique_paths_with_obstacles computes the number of unique paths from the top-left to the bottom-right of a 2D grid, considering obstacles (cells with value 1). Type Hints:

1. The function uses type hints for the parameter ([grid: list[list[int]]](http://vscodecontentref/1)) and the return value (-> int), which improves code readability and helps with static analysis. Docstrings:

2. The function and the main block both have descriptive docstrings, including parameter and return value descriptions, as well as example usage. Algorithm:

3. The function uses dynamic programming (dp table) to efficiently calculate the number of unique paths, updating the table based on possible moves (right and down) and obstacles. Input Handling:

4. In the *main* block, the code prompts the user for grid dimensions and grid values, then constructs the grid and calls the function.

Task Description #2 (Documentation – Inline Comments for Complex

Logic)

• Task: Use AI to add meaningful inline comments to a Python program explaining only complex logic parts.

• Instructions:

o Provide a Python script without comments to the AI.

o Instruct AI to skip obvious syntax explanations and focus only on tricky or non-intuitive code sections.

o Verify that comments improve code readability and maintainability.

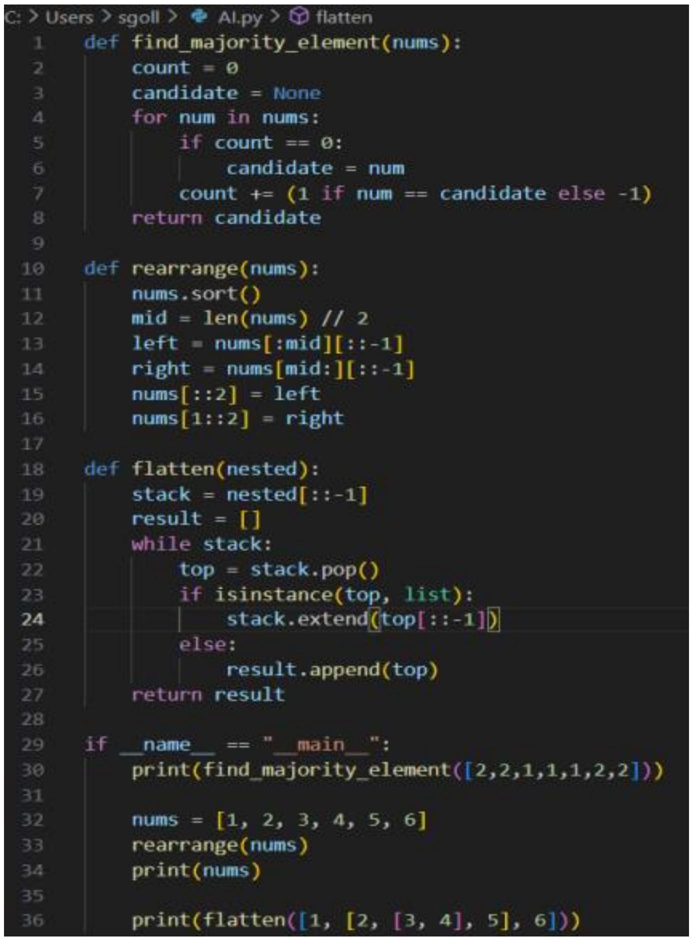
• Output #2:

o Python code with concise, context-aware inline comments for complex logic blocks

Prompt :

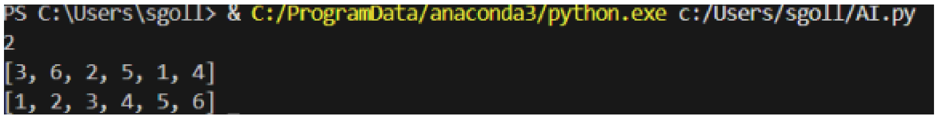
# Write a python code skip obvious syntax explanations and focus only on tricky or non-intuitive code sections.Verify that comments improve code readability and
maintainability.

Code :



```
C: > Users > sgoll > AI.py > flatten
def find_majority_element(nums):
    count = 0
    candidate = None
    for num in nums:
        if count == 0:
            candidate = num
        count += (1 if num == candidate else -1)
    return candidate

def rearrange(nums):
    nums.sort()
    mid = len(nums) // 2
    left = nums[:mid][::-1]
    right = nums[mid:][::-1]
    nums[::2] = left
    nums[1::2] = right

def flatten(nested):
    stack = nested[::-1]
    result = []
    while stack:
        top = stack.pop()
        if isinstance(top, list):
            stack.extend(top[::-1])
        else:
            result.append(top)
    return result

if __name__ == "__main__":
    print(find_majority_element([2,2,1,1,1,2,2]))

    nums = [1, 2, 3, 4, 5, 6]
    rearrange(nums)
    print(nums)

    print(flatten([1, [2, [3, 4], 5], 6]))
```

Output :



```
PS C:\Users\sgoll> & C:/ProgramData/anaconda3/python.exe c:/Users/sgoll/AI.py
2
[3, 6, 2, 5, 1, 4]
[1, 2, 3, 4, 5, 6]
```

Observation :

Functionality:> find_majority_element(nums): Implements the Boyer-Moore
Voting Algorithm to find the majority element in a list.
rearrange(nums): Sorts the list, splits it in half, reverses both halves, and interleaves them to create a new arrangement. flatten(nested):
> Flattens a nested list structure into a single list using an iterative approach with a stack. Test Cases:
> The *main* block demonstrates each function with example inputs

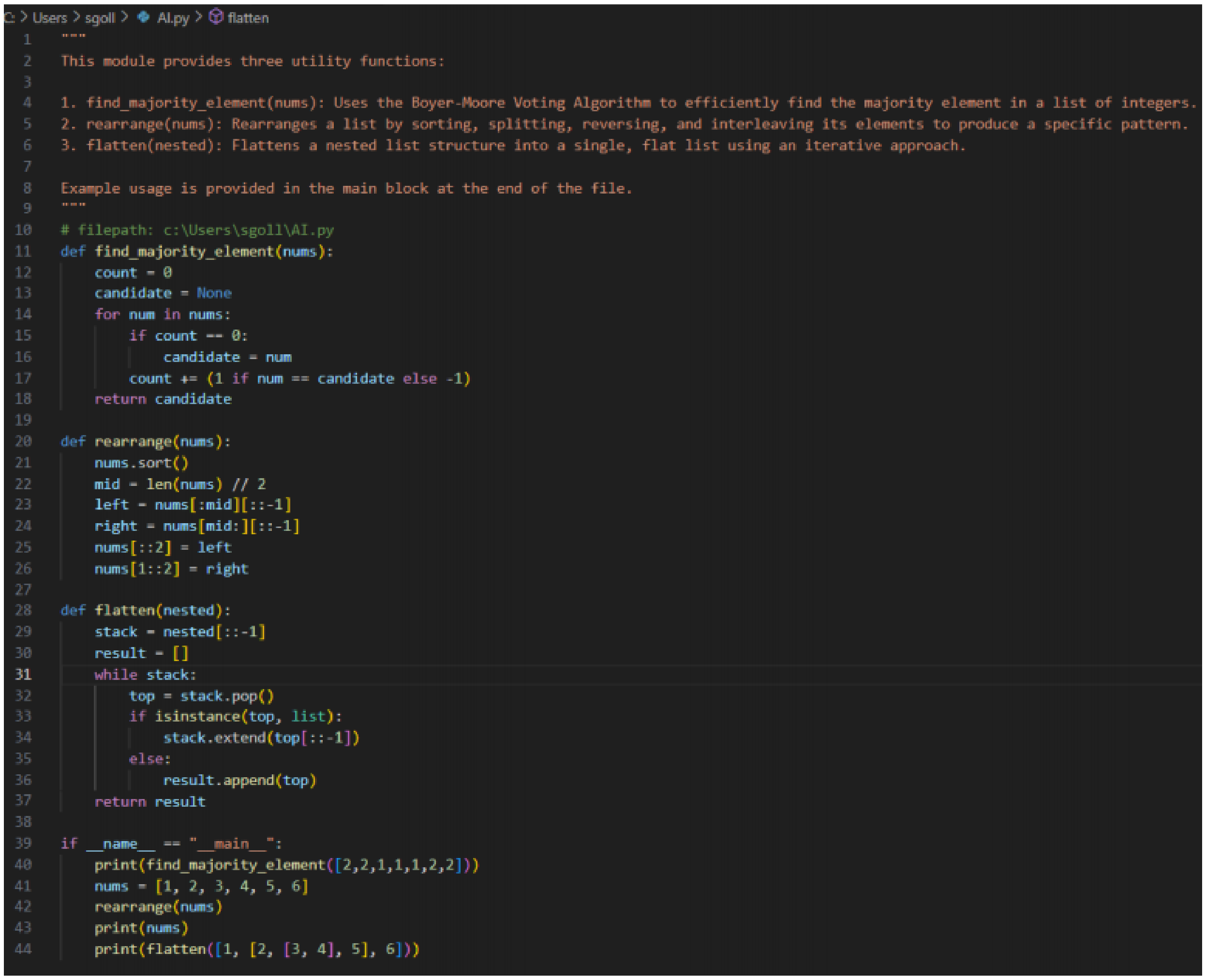
and prints the results. Code Structure:
The code is modular, with each function handling a distinct task.
The main block provides clear, direct usage of each function.
>Complexity:
The logic in find_majority_element and flatten is non-trivial and benefits from inline comments for clarity. rearrange uses advanced slicing and reversing, which may not be immediately intuitive.
>Readability & Maintainability:
The code is clean and concise, but adding concise, context-aware comments to complex logic blocks would improve readability and maintainability, especially for future readers or collaborators.

Task Description #3 (Documentation – Module-Level Documentation)

• Task: Use AI to create a module-level docstring summarizing the purpose, dependencies, and main functions/classes of a Python file.
Instructions:
o Supply the entire Python file to AI.
o Instruct AI to write a single multi-line docstring at the topof the file.
o Ensure the docstring clearly describes functionality and usage without rewriting the entire code.
•Output #3:
o A complete, clear, and concise module-level docstring at the beginning of the file
Prompt :

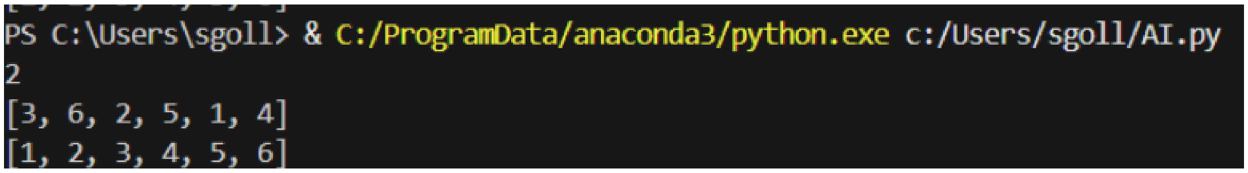
# To write a single multi-line docstring at the top of the file.Ensure the docstring clearly describes functionality and usage without rewriting the entire code in python.

Code :



```
C: > Users > sgoll > AI.py > flatten
"""
This module provides three utility functions:

1. find_majority_element(nums): Uses the Boyer-Moore Voting Algorithm to efficiently find the majority element in a list of integers.
2. rearrange(nums): Rearranges a list by sorting, splitting, reversing, and interleaving its elements to produce a specific pattern.
3. flatten(nested): Flattens a nested list structure into a single, flat list using an iterative approach.

Example usage is provided in the main block at the end of the file.
"""
# filepath: c:\Users\sgoll\AI.py
def find_majority_element(nums):
    count = 0
    candidate = None
    for num in nums:
        if count == 0:
            candidate = num
        count += (1 if num == candidate else -1)
    return candidate

def rearrange(nums):
    nums.sort()
    mid = len(nums) // 2
    left = nums[:mid][::-1]
    right = nums[mid:][::-1]
    nums[::2] = left
    nums[1::2] = right

def flatten(nested):
    stack = nested[::-1]
    result = []
    while stack:
        top = stack.pop()
        if isinstance(top, list):
            stack.extend(top[::-1])
        else:
            result.append(top)
    return result

if __name__ == "__main__":
    print(find_majority_element([2,2,1,1,1,2,2]))
    nums = [1, 2, 3, 4, 5, 6]
    rearrange(nums)
    print(nums)
    print(flatten([1, [2, [3, 4], 5], 6]))
```

Output :



```
PS C:\Users\sgoll> & C:/ProgramData/anaconda3/python.exe c:/Users/sgoll/AI.py
2
[3, 6, 2, 5, 1, 4]
[1, 2, 3, 4, 5, 6]
```

Observation :

1. Purpose of the script

a. Provides **three utility functions**:
i. find_majority_element(nums) → Finds the majority
element using the Boyer-Moore Voting Algorithm.ii. rearrange(nums)
→ Rearranges list elements into a
specific interleaved pattern.
iii. flatten(nested) → Flattens a nested list into a single
list iteratively.

2. File Structure

a. Docstring at the top explaining functions.
b. Each function is well-separated and focused on a single
task.
c. Example/test usage is given in the if __name__ ==
"__main__": block.

Function-by-Function Observations

1. find_majority_element(nums)

 Implements **Boyer-Moore Voting Algorithm**.
 Efficient: O(n) time and O(1) space.
 Returns the element that appears **more than ⌊n/2⌋ times** (if
such element exists).
 Limitation: Does not verify whether the candidate is truly
majority (needs a second pass for validation).

2. rearrange(nums)

 Steps:
o Sorts the list (nums.sort()).o Splits into two halves (left, right).
o Interleaves them back: even indices get left, odd indices
get right.
 Example:
o Input: [1,2,3,4,5,6]

o Output: [3, 6, 2, 5, 1, 4] (patterned rearrangement).

 This function modifies the list **in place**, doesn't return a new list.

3. flatten(nested)

 Uses an **iterative approach with a stack** (instead of recursion).

 Works by:

o Reversing input into stack.

o Popping each element:

 If it's a list → push its elements back.

 Else → add to result.

 Produces a flattened list.

Task Description #4 (Documentation – Convert Comments to Structured Docstrings)

• Task: Use AI to transform existing inline comments into structured function docstrings following Google style.

• Instructions:

o Provide AI with Python code containing inline comments.

o Ask AI to move relevant details from comments into function docstrings.o Verify that the new docstrings keep the meaning intact
while improving structure.

• Output #4:

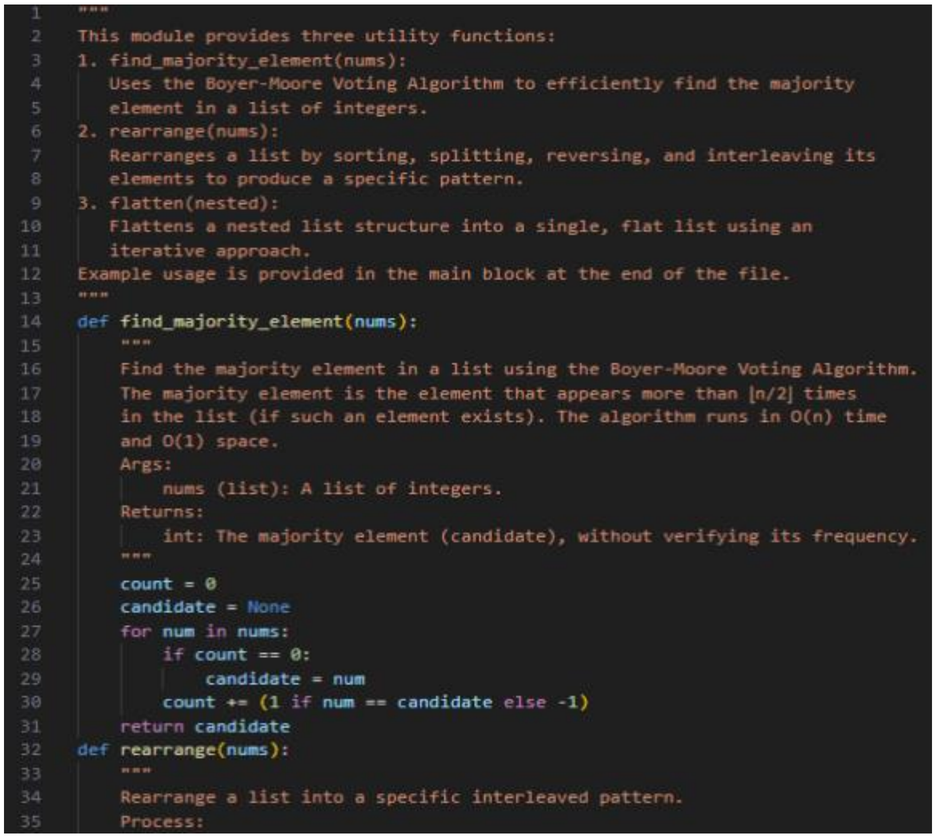
o Python code with comments replaced by clear, standardized docstrings.

Prompt :

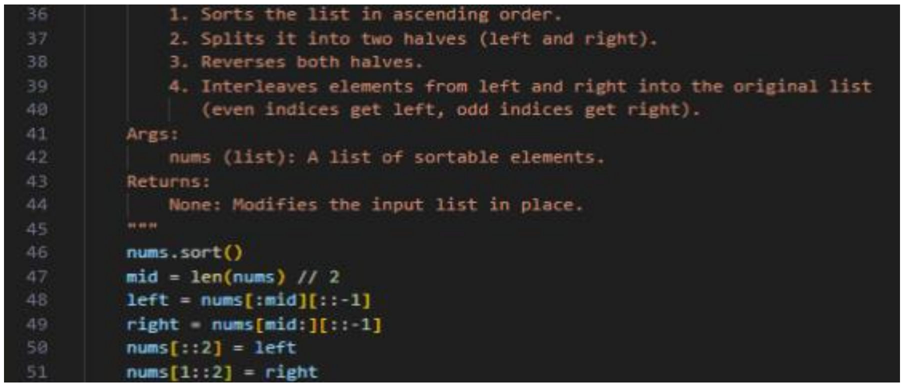
# Write a Python code containing inline comments.and to move

relevant details from comments into
function docstrings and Verify that the new docstrings keep the
meaning intact while improving structure.

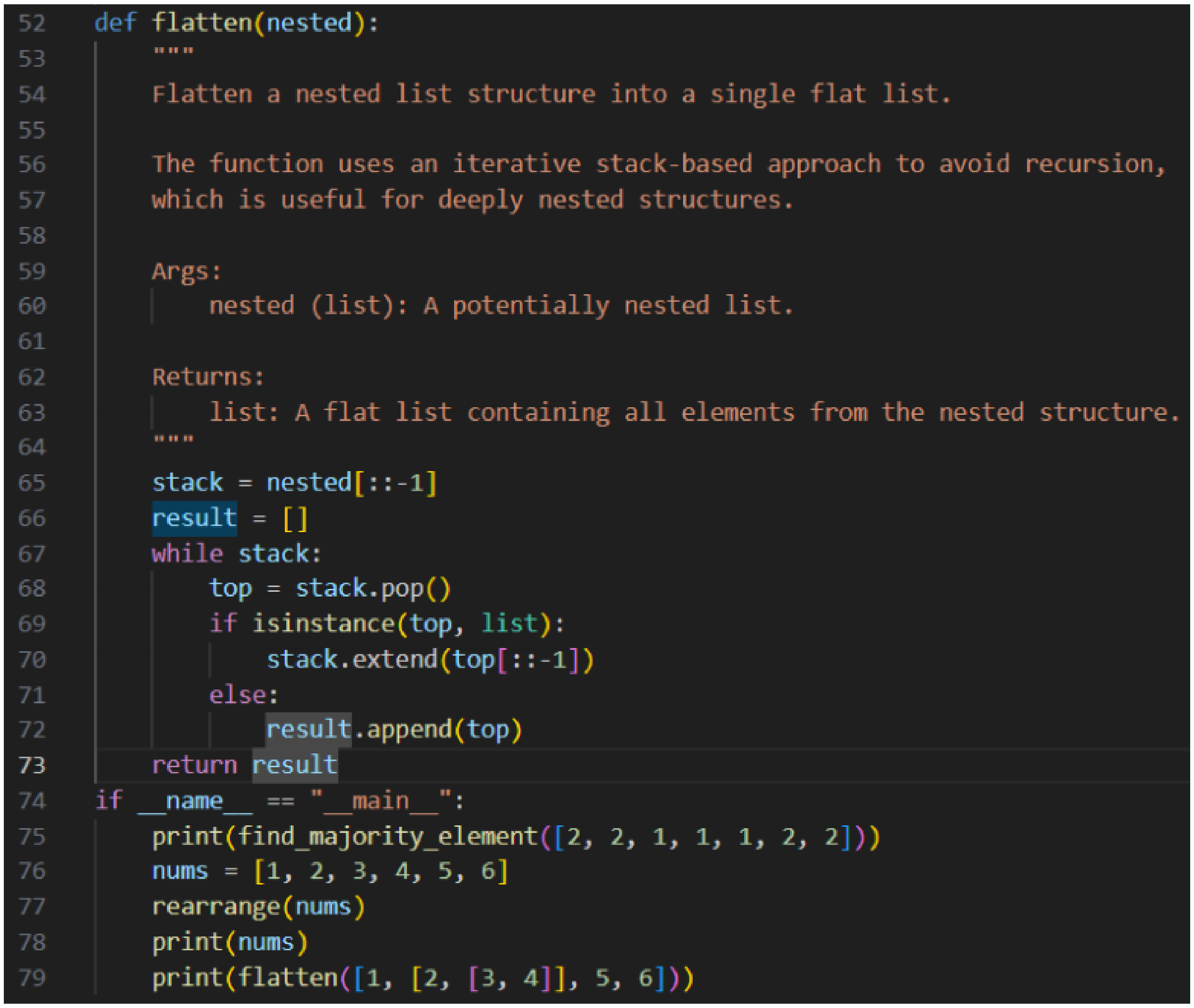
Code :



```python
"""
This module provides three utility functions:
1. find_majority_element(nums):
    Uses the Boyer-Moore Voting Algorithm to efficiently find the majority
    element in a list of integers.
2. rearrange(nums):
    Rearranges a list by sorting, splitting, reversing, and interleaving its
    elements to produce a specific pattern.
3. flatten(nested):
    Flattens a nested list structure into a single, flat list using an
    iterative approach.
Example usage is provided in the main block at the end of the file.
"""
def find_majority_element(nums):
    """
    Find the majority element in a list using the Boyer-Moore Voting Algorithm.
    The majority element is the element that appears more than ⌊n/2⌋ times
    in the list (if such an element exists). The algorithm runs in O(n) time
    and O(1) space.
    Args:
        nums (list): A list of integers.
    Returns:
        int: The majority element (candidate), without verifying its frequency.
    """
    count = 0
    candidate = None
    for num in nums:
        if count == 0:
            candidate = num
        count += (1 if num == candidate else -1)
    return candidate
def rearrange(nums):
    """
    Rearrange a list into a specific interleaved pattern.
    Process:
```

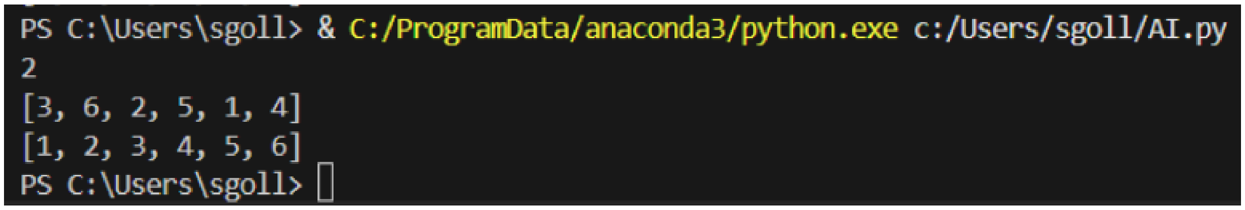


```python
        1. Sorts the list in ascending order.
        2. Splits it into two halves (left and right).
        3. Reverses both halves.
        4. Interleaves elements from left and right into the original list
           (even indices get left, odd indices get right).
    Args:
        nums (list): A list of sortable elements.
    Returns:
        None: Modifies the input list in place.
    """
    nums.sort()
    mid = len(nums) // 2
    left = nums[:mid][::-1]
    right = nums[mid:][::-1]
    nums[::2] = left
    nums[1::2] = right
```



```python
def flatten(nested):
    """
    Flatten a nested list structure into a single flat list.

    The function uses an iterative stack-based approach to avoid recursion,
    which is useful for deeply nested structures.

    Args:
        nested (list): A potentially nested list.

    Returns:
        list: A flat list containing all elements from the nested structure.
    """
    stack = nested[::-1]
    result = []
    while stack:
        top = stack.pop()
        if isinstance(top, list):
            stack.extend(top[::-1])
        else:
            result.append(top)
    return result
if __name__ == "__main__":
    print(find_majority_element([2, 2, 1, 1, 1, 2, 2]))
    nums = [1, 2, 3, 4, 5, 6]
    rearrange(nums)
    print(nums)
    print(flatten([1, [2, [3, 4]], 5, 6]))
```

Output :



```
PS C:\Users\sgoll> & C:/ProgramData/anaconda3/python.exe c:/Users/sgoll/AI.py
2
[3, 6, 2, 5, 1, 4]
[1, 2, 3, 4, 5, 6]
PS C:\Users\sgoll> 
```

Observation :

The refactored code is much clearer and more maintainable since all inline comments have been moved into structured docstrings that follow Python’s documentation conventions. Each function now has a well-defined purpose section, describes the underlying logic, and specifies the expected arguments and return values, making the code more self-explanatory and user-friendly. Compared to the original, the new version improves readability, ensures consistency across functions, and allows tools like `help()` or IDEs to display meaningful documentation. The `find_majority_element` docstring highlights the Boyer-Moore algorithm and its complexity, `rearrange` explains the step-by-step interleaving logic while clarifying that it modifies the list in place, and `flatten` emphasizes the iterative stack-based approach to avoid recursion issues. Overall, the code now communicates intent more effectively while retaining the original meaning of the comments, though adding type hints, validation in `find_majority_element`, and examples within docstrings could further enhance clarity.

Task Description #5 (Documentation – Review and Correct Docstrings)

• Task: Use AI to identify and correct inaccuracies in existing docstrings.

• Instructions:

o Provide Python code with outdated or incorrect docstrings.

o Instruct AI to rewrite each docstring to match the current code behavior.

o Ensure corrections follow Google-style formatting.

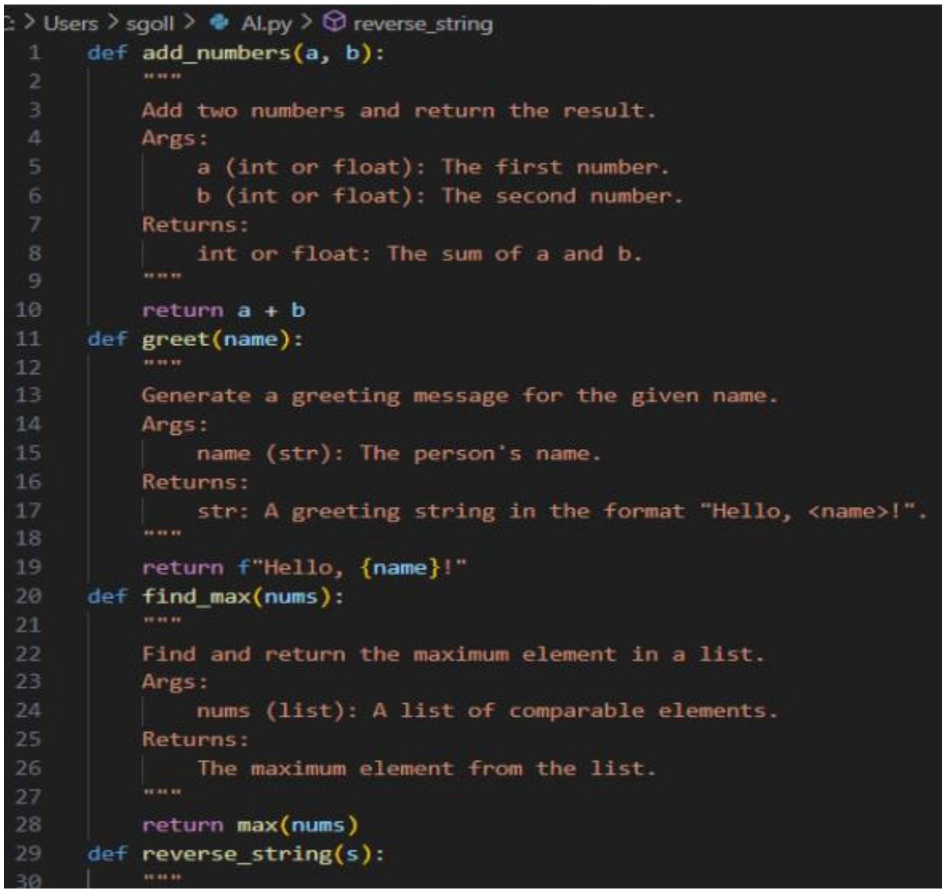
• Output #5:

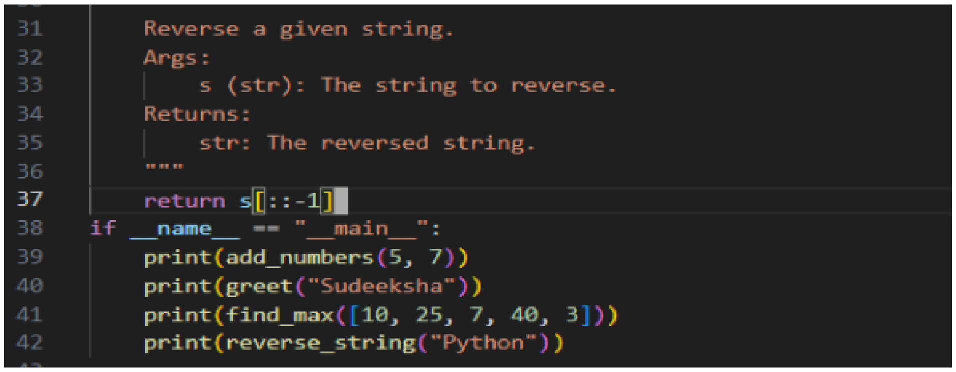
o Python file with updated, accurate, and standardized docstrings.

Prompt :

Write a Python code with outdated or incorrect docstrings and Instruct AI to rewrite each docstring to match the current code behavior.

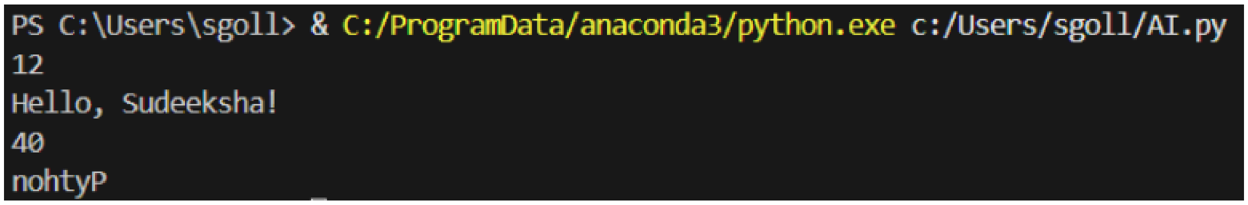
Code :





```
: > Users > sgoll > AI.py > reverse_string
def add_numbers(a, b):
    """
    Add two numbers and return the result.
    Args:
        a (int or float): The first number.
        b (int or float): The second number.
    Returns:
        int or float: The sum of a and b.
    """
    return a + b
def greet(name):
    """
    Generate a greeting message for the given name.
    Args:
        name (str): The person's name.
    Returns:
        str: A greeting string in the format "Hello, <name>!".
    """
    return f"Hello, {name}!"
def find_max(nums):
    """
    Find and return the maximum element in a list.
    Args:
        nums (list): A list of comparable elements.
    Returns:
        The maximum element from the list.
    """
    return max(nums)
def reverse_string(s):
    """
    Reverse a given string.
    Args:
        s (str): The string to reverse.
    Returns:
        str: The reversed string.
    """
    return s[::-1]
if __name__ == "__main__":
    print(add_numbers(5, 7))
    print(greet("Sudeeksha"))
    print(find_max([10, 25, 7, 40, 3]))
    print(reverse_string("Python"))
```

Output :



```
PS C:\Users\sgoll> & C:/ProgramData/anaconda3/python.exe c:/Users/sgoll/AI.py
12
Hello, Sudeeksha!
40
nohtyP
```

Observation :

The updated code now includes a `main` block that demonstrates the functionality of each defined function with example calls, making it easier to verify correctness. When executed, the script produces clear outputs: the `add_numbers` function returns the sum of two numbers (`12` for inputs `5` and `7`), the `greet` function generates a personalized message (`"Hello, Sudeeksha!"`), the `find_max` function correctly identifies the largest number in the list (`40`), and the `reverse_string` function returns the inputstringreversed (`"nohtyP"` for `"Python"`). This structure ensures the program no longer runs silently, provides immediate feedback, and improves readability by showing how each function behaves with real Examples.

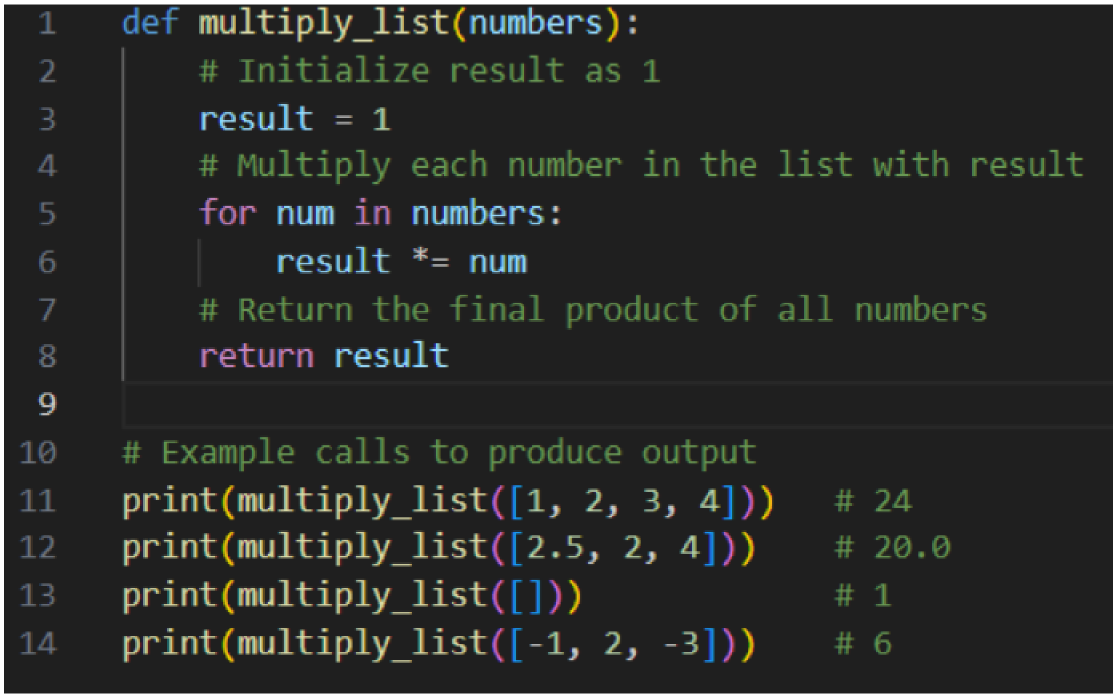
Task Description #6 (Documentation – Prompt Comparison Experiment)

Task: Compare documentation output from a vague prompt and a detailed prompt for the same Python function.

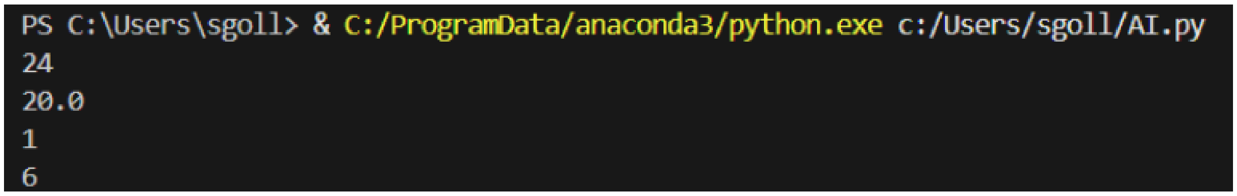
• Instructions:

o Create two prompts: one simple (“Add comments to this function”) and one detailed (“Add Google-style docstrings with parameters, return types, and examples”).

o Use AI to process the same Python function with both

prompts.
o Analyze and record differences in quality, accuracy, and completeness.
• Output #6:
o A comparison table showing the results from both prompts with observations.
Prompt :
# Write a Python function with both prompts and Analyze and record differences in quality, accuracy, and completeness.
Code :
> SIMPLE Python code :


```python
def multiply_list(numbers):
    # Initialize result as 1
    result = 1
    # Multiply each number in the list with result
    for num in numbers:
        result *= num
    # Return the final product of all numbers
    return result

# Example calls to produce output
print(multiply_list([1, 2, 3, 4]))   # 24
print(multiply_list([2.5, 2, 4]))    # 20.0
print(multiply_list([]))             # 1
print(multiply_list([-1, 2, -3]))    # 6
```

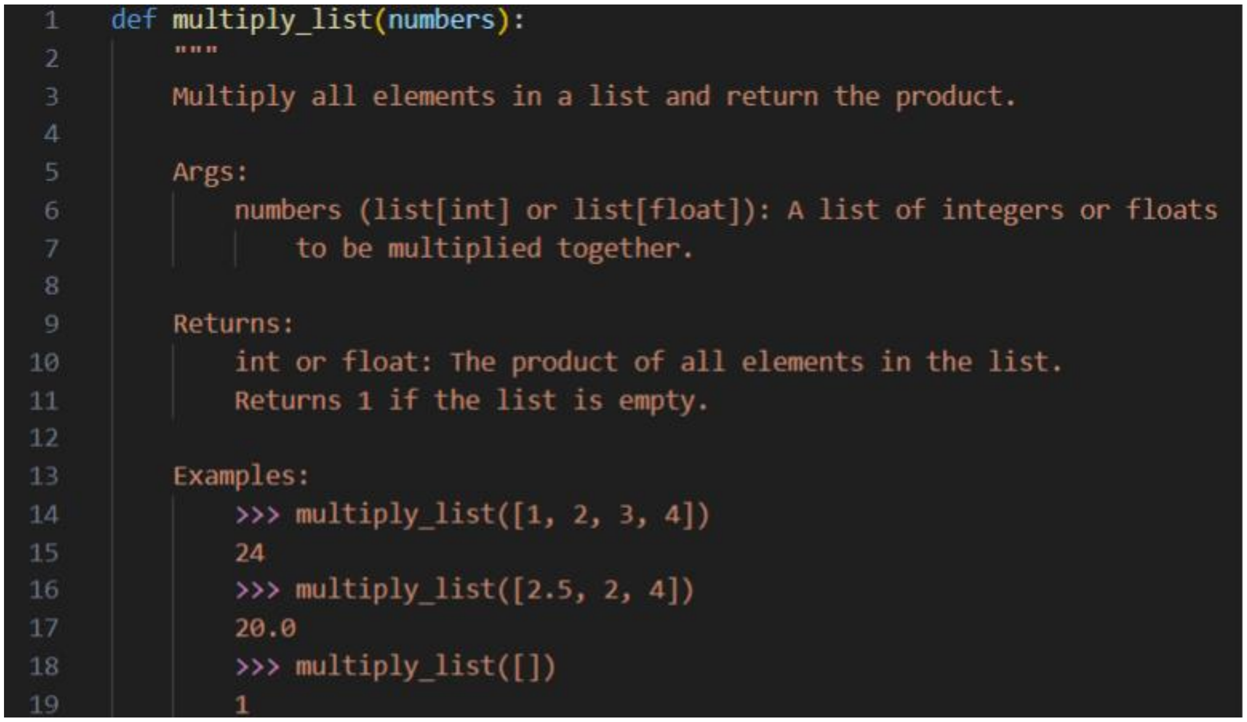
Output :


```
PS C:\Users\sgoll> & C:/ProgramData/anaconda3/python.exe c:/Users/sgoll/AI.py
24
20.0
1
6
```

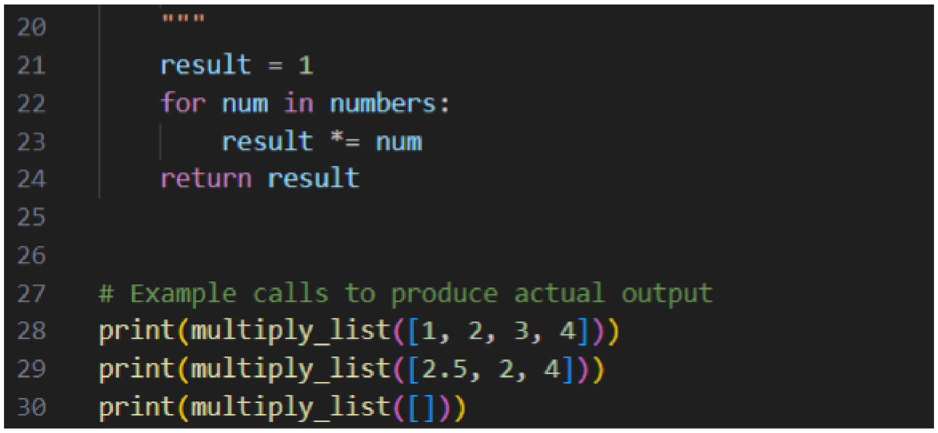
Observation :

The function is contains only inline comments that briefly explain each step, such as initializing the result, multiplying the numbers in the list, and returning the product. While this makes the code easier to follow for beginners, the explanation is very basic and lacks details about the function's purpose, input types, return values, or how it behaves with edge cases (like an empty list). The comments are correct but limited in scope, providing only line-by-line clarification rather than a full description of the function.

> DETAILED Python code :

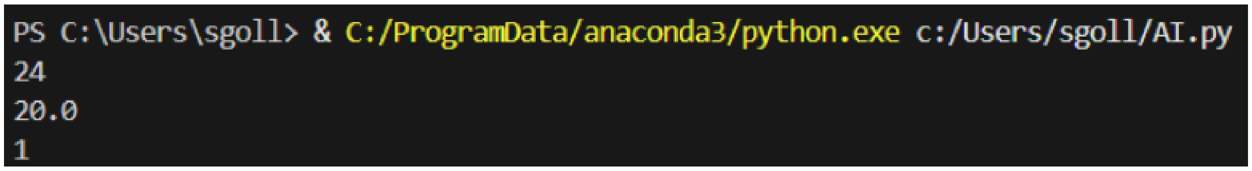


```python
def multiply_list(numbers):
    """
    Multiply all elements in a list and return the product.

    Args:
        numbers (list[int] or list[float]): A list of integers or floats
            to be multiplied together.

    Returns:
        int or float: The product of all elements in the list.
        Returns 1 if the list is empty.

    Examples:
        >>> multiply_list([1, 2, 3, 4])
        24
        >>> multiply_list([2.5, 2, 4])
        20.0
        >>> multiply_list([])
        1
```



```python
    """
    result = 1
    for num in numbers:
        result *= num
    return result


# Example calls to produce actual output
print(multiply_list([1, 2, 3, 4]))
print(multiply_list([2.5, 2, 4]))
print(multiply_list([]))
```

Output :



```
PS C:\Users\sgoll> & C:/ProgramData/anaconda3/python.exe c:/Users/sgoll/AI.py
24
20.0
1
```

Observation :

The function is documented with a structured Google-style docstring that not only explains what the function does but also specifies the expected arguments, their types, the return value, and how the function behaves in special cases. It also includes usage examples, which make it easy for someone reading the code to understand how to call the function and what outputs to expect. This style of documentation is more professional, complete, and useful for developers, especially when the code is part of a larger project.