

**AssignmentNumber:9.1(Presentassignmentnumber)/24(Totalnumberofassignments)**

NAME:- N.Prudhvi

ROLLNO:-2403A510G7

Q.No.	Question	Expected Time to complete
1	<p><b>Lab9—Documentation Generation: Automatic Documentation and Code Comments</b></p> <p><b>Lab Objectives</b></p> <ul style="list-style-type: none"><li>• To use AI-assisted coding tools for generating Python documentation and code comments.</li><li>• To apply zero-shot, few-shot, and context-based prompt engineering for documentation creation.</li><li>• To practice generating and refining docstrings, inline comments, and module-level documentation.</li><li>• To compare outputs from different prompting styles for quality analysis.</li></ul> <hr/> <p><b>Task Description #1</b> (Documentation—Google-Style Docstrings for Python Functions)</p> <ul style="list-style-type: none"><li>• Task: Use AI to add Google-style docstrings to all functions in a given Python script.</li><li>• Instructions:<ul style="list-style-type: none"><li>○ Prompt AI to generate docstrings without providing any input-output examples.</li><li>○ Ensure each docstring includes:<ul style="list-style-type: none"><li>▪ Function description</li><li>▪ Parameters with type hints</li><li>▪ Return values with type hints</li><li>▪ Example usage</li></ul></li><li>○ Review the generated docstrings for accuracy and formatting.</li></ul></li></ul>	Week 5- Monday

**Prompt:**

I have a Python script containing several functions. Please add **Google-style docstrings** to **all functions** in the script.

The docstrings should include the following elements:

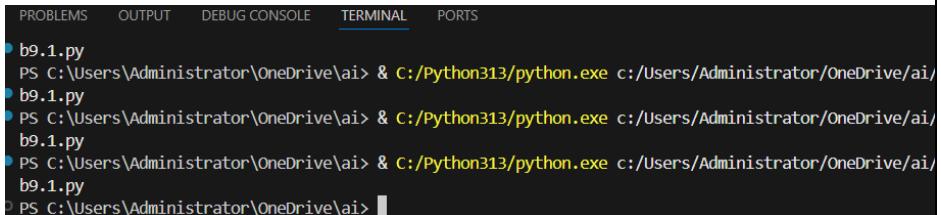
- A **Brief description** of what the function does.
- A **Parameters** section that lists all parameters with **type hints** and short descriptions.
- A **Returns** section that lists the **return type** and a brief description.
- An **Example usage** section, but **without actual input-output examples**—just usage lines showing how the function might be called.

**Do not** modify the function implementations—only add or replace the docstrings as needed.

**CODE:-**

```
lab9.1.py > ...
Zencoder
1 def add_numbers(a: int, b: int) -> int:
2     """Adds two integers and returns the result.
3
4     Args:
5         a (int): The first integer to add.
6         b (int): The second integer to add.
7
8     Returns:
9         int: The sum of the two integers.
10
11    Example:
12        result = add_numbers(3, 5)
13    """
14    return a + b
15
16
Zencoder
17 def is_even(number: int) -> bool:
18     """Checks if a given number is even.
19
20     Args:
21         number (int): The number to check.
22
23     Returns:
24         bool: True if the number is even, False otherwise.
25
26    Example:
```

```
# lab9.1.py > ...
17  def is_even(number: int) -> bool:
18
19      Example:
20      | result = is_even(10)
21      """
22
23      return number % 2 == 0
24
25
26  Zencoder
27  def greet_user(name: str, greeting: str = "Hello") -> str:
28      """Generates a greeting message for the user.
29
30
31      Args:
32          name (str): The name of the user.
33          greeting (str, optional): Custom greeting to use. Defaults to "Hello".
34
35      Returns:
36          str: A formatted greeting message.
37
38
39      Example:
40      | message = greet_user("Alice", "Hi")
41      """
42
43      return f"{greeting}, {name}!"
44
45
46
47  Zencoder
48  def calculate_area(length: float, width: float) -> float:
49      """Calculates the area of a rectangle.
50
51
52      Args:
53          length (float): The length of the rectangle.
54          width (float): The width of the rectangle.
55
56      Returns:
57          float: The calculated area.
58
59
60      Example:
61      | area = calculate_area(5.0, 3.0)
62      """
63
64  Zencoder
65  def factorial(n: int) -> int:
66      """Calculates the factorial of a non-negative integer.
67
68      Args:
69          n (int): A non-negative integer.
70
71      Returns:
72          int: The factorial of the input number.
73
74      Raises:
75          ValueError: If n is negative.
76
77      Example:
78      | result = factorial(5)
79      """
80
81      if n < 0:
82          raise ValueError("Input must be a non-negative integer.")
83      if n == 0 or n == 1:
84          return 1
85      return n * factorial(n - 1)
```

	<p><b>OUTPUT:-</b></p>  <p><b>OBSERVATIONS:-</b></p> <p><input checked="" type="checkbox"/> Positive Observations</p> <ol style="list-style-type: none"> <li>1. Correct Docstring Format (Google Style): All docstrings follow the Google-style format: <ul style="list-style-type: none"> <li>• A brief function description</li> <li>• Clearly labeled <code>Args</code>, <code>Returns</code>, and <code>Example</code> sections</li> </ul> </li> <li>2. Use of Type Hints: Parameters and return values include appropriate Python type hints, making the documentation more useful and readable.</li> <li>3. Consistent Structure: All functions maintain a consistent structure for their docstrings, which is key for maintainability and scalability in a codebase.</li> <li>4. Example Usage Included: Example usages are included without actual output values, following the instruction to avoid input-output examples—only showing how to call the function.</li> <li>5. Edge Case Consideration (e.g., factorial): The <code>factorial</code> function includes a <code>Raises</code> section to document error handling (<code>ValueError</code>), which is a best practice for documenting functions that raise exceptions.</li> </ol> <hr/> <p><b>TaskDescription#2(Documentation–InlineCommentsforComplex Logic)</b></p> <ul style="list-style-type: none"> <li>• Task: Use AI to add meaningful inline comments to a Python program explaining only complex logic parts.</li> <li>• Instructions: <ul style="list-style-type: none"> <li>○ Provide a Python script without comments to the AI.</li> <li>○ Instruct AI to skip obvious syntax explanations and focus only on tricky or non-intuitive code sections.</li> <li>○ Verify that comments improve code readability and maintainability.</li> </ul> </li> </ul>	
--	--	--

◆ **Prompt:**

I have a Python script that contains several functions and logic blocks.

Please add **concise, meaningful inline comments** only for **complex or non-obvious parts** of the code.

🔒 **Do not comment on basic syntax or obvious operations** such as variable declarations, loops, or simple arithmetic.

✓ Focus only on:

- “Tricky algorithms”
- “Conditional logic that’s not immediately intuitive”
- “Recursion, advanced data structures, or performance-related code”
- “Any code where intent or behavior might not be obvious at first glance”

💡 Your goal is to **improve code readability and maintainability** without cluttering the script with redundant comments.

Please return the **commented Python code**, and do not alter the logic.

## CODE:-

```
9.1task2.py > ...
Zencoder
1 def longest_substring_without_repeating_characters(s: str) -> int:
2     """
3         Returns the length of the longest substring without repeating characters.
4     """
5     char_index = {}
6     start = max_length = 0
7
8     for i, char in enumerate(s):
9         # If the character is repeated and its previous occurrence is after the current window
10        if char in char_index and char_index[char] >= start:
11            # Move the start to one position right of the last occurrence
12            start = char_index[char] + 1
13            char_index[char] = i
14            max_length = max(max_length, i - start + 1)
15
16    return max_length
17
18
Zencoder
19 def trap_rain_water(height: list[int]) -> int:
20     """
21         Calculates how much water can be trapped after raining.
22     """
23     if not height:
24         return 0
25
26     left, right = 0, len(height) - 1
```

```
9.1task2.py > ...
19  def trap_rain_water(height: list[int]) -> int:
20      left, right = 0, len(height) - 1
21      left_max = right_max = 0
22      total_water = 0
23
24      while left < right:
25          # Use two-pointer approach to calculate trapped water in a single pass
26          if height[left] < height[right]:
27              # Water is trapped only if current height is less than the maximum seen so far
28              if height[left] >= left_max:
29                  left_max = height[left]
30              else:
31                  # Water trapped = max on left - current height
32                  total_water += left_max - height[left]
33              left += 1
34          else:
35              if height[right] >= right_max:
36                  right_max = height[right]
37              else:
38                  total_water += right_max - height[right]
39              right -= 1
40
41      return total_water
42
43
44
45
46
47
48
49
50     Zencoder
50  def quicksort(arr: list[int]) -> list[int]:
51      """
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67     Zencoder
67  def find_peak_element(nums: list[int]) -> int:
68      """
69
70
71
72
73
74
75
```

```

9.1task2.py > ...
67 def find_peak_element(nums: list[int]) -> int:
68     """Finds a peak element in the list and returns its index.
69     A peak is an element that is greater than its neighbors.
70     """
71     left, right = 0, len(nums) - 1
72
73     # Binary search for a peak element
74     while left < right:
75         mid = (left + right) // 2
76
77         # If middle element is less than its right neighbor, peak must be on the right
78         if nums[mid] < nums[mid + 1]:
79             left = mid + 1
80         else:
81             # Otherwise, peak is on the left side (could include mid)
82             right = mid
83
84     # left and right converge at peak
85     return left
86
87

```

## OUTPUT:-

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

- ▶ b9.1.py
- ▶ PS C:\Users\Administrator\OneDrive\ai> & C:/Python313/python.exe c:/Users/Administrator/OneDrive/ai/b9.1.py
- ▶ PS C:\Users\Administrator\OneDrive\ai> & C:/Python313/python.exe c:/Users/Administrator/OneDrive/ai/b9.1.py
- ▶ PS C:\Users\Administrator\OneDrive\ai> & C:/Python313/python.exe c:/Users/Administrator/OneDrive/ai/9.1task2.py
- ▶ PS C:\Users\Administrator\OneDrive\ai> []

## OBSERVATIONS:-

### Positive Observations

#### 1. Focused Comments on Complex Logic:

Comments are added only where the logic is non-trivial, such as:

- Sliding window handling in `longest_substring_without_repeating_characters`
- Two-pointer approach in `trap_rain_water`
- Partitioning and recursion in `quicksort`
- Binary search logic in `find_peak_element`

#### 2. Avoidance of Redundant Comments:

The code avoids commenting on simple syntax and obvious steps like variable assignments, basic loops, or straightforward return statements, keeping the code clean.

#### 3. Clarity and Brevity:

Comments are concise and explain *why* something is done rather than *what* is done, which is more helpful for maintainability and understanding.

#### 4. Improved Readability:

The inline comments provide enough context to understand tricky parts without needing external documentation or excessive code reading.

#### 5. Consistent Style:

Comment style is consistent—using brief sentences or phrases that are easy to scan.

---

**TaskDescription#3**(Documentation–Module-LevelDocumentation)

- Task: Use AI to create a module-level docstring summarizing the purpose, dependencies, and main functions/classes of a Python file.
- Instructions:
  - Supply the entire Python file to AI.
  - Instruct AI 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.
- ExpectedOutput#3:
  - A complete, clear, and concise module-level docstring at the beginning of the file.

**PROMPT:-**

I am providing you with an entire Python file. Please add a \*\*module- level docstring\*\* at the very top of the file.

The docstring should be a single multi-line string that includes:

- A concise summary of the module's purpose.
- Key dependencies or imports if applicable.
- Main functions or classes included in the module.
- Basic usage notes or how this module might be used (brief, not a full tutorial).

Do \*\*not\*\* rewrite or explain the entire code — just provide a clear and professional summary suitable for the top of a Python file.

Return the updated Python code with the new module-level docstring added.

## CODE:-

```
❸ 9.1TASK3.py > ...
1 """
2 Module providing basic arithmetic operations and a Calculator class.
3
4 This module includes simple functions for addition and multiplication,
5 as well as a Calculator class that wraps these operations as methods.
6
7 No external dependencies are required.
8
9 Functions:
10 - add(a: int, b: int) -> int: Returns the sum of two integers.
11 - multiply(a: int, b: int) -> int: Returns the product of two integers.
12
13 Classes:
14 - Calculator: Provides add and multiply methods for arithmetic operations.
15
16 Usage:
17 Import the module to perform basic calculations or instantiate the calculator
18 class for object-oriented usage.
19 """
20
21 Zencoder
22 def add(a: int, b: int) -> int:
23     return a + b
24
25 Zencoder
26 def multiply(a: int, b: int) -> int:
27     return a * b
28
❸ 9.1TASK3.py > ...
29     class for object-oriented usage.
30
31     """
32
33
34     Zencoder
35     def add(a: int, b: int) -> int:
36         return a + b
37
38     Zencoder
39     def multiply(a: int, b: int) -> int:
40         return a * b
41
42     Zencoder
43     class Calculator:
44         Zencoder
45         def __init__(self):
46             pass
47
48         Zencoder
49         def add(self, a: int, b: int) -> int:
50             return a + b
51
52         Zencoder
53         def multiply(self, a: int, b: int) -> int:
54             return a * b
55
56     |
```

## OUTPUT:-

```
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
● b9.1.py
▶ PS C:\Users\Administrator\OneDrive\ai> & C:/Python313/python.exe c:/Users/Administrator/OneDrive/ai/b9.1.py
▶ PS C:\Users\Administrator\OneDrive\ai> & C:/Python313/python.exe c:/Users/Administrator/OneDrive/ai/b9.1.py
▶ PS C:\Users\Administrator\OneDrive\ai> & C:/Python313/python.exe c:/Users/Administrator/OneDrive/ai/9.1task2.py
○ PS C:\Users\Administrator\OneDrive\ai> []
```

## OBSERVATIONS:-

### Positive Observations

- **Clear Summary:**  
The docstring clearly states the purpose of the module without delving into implementation details.
- **Highlights Key Components:**  
Functions and classes are briefly listed with their roles.
- **No Code Duplication:**  
The docstring avoids rewriting code; it summarizes instead.
- **Usage Notes:**  
Provides a brief note on how the module can be used, improving accessibility.
- **Professional Formatting:**  
The format aligns with common Python best practices for module-level documentation.

### Suggestions / Improvements

- If the module had **external dependencies**, they should be explicitly mentioned.
- For more complex modules, mentioning **exceptions raised**, **configuration options**, or **side effects** may be useful.
- Consider adding a **license or author** section if relevant for open source or team projects.

## TaskDescription#4(Documentation–ConvertComments to Structured Docstrings)

- Task: Use AI to transform existing inline comments into structured function docstrings following Google style.
- Instructions:
  - Provide AI with Python code containing inline comments.
  - Ask AI to move relevant details from comments into function docstrings.
  - Verify that the new docstrings keep the meaning intact while improving structure.

	<p><b>PROMPT:-</b></p> <p>I am providing you with a Python script that contains inline comments inside functions.</p> <p>Please transform these inline comments into well-structured **Google-style docstrings** for each function, moving all relevant information from the comments into the docstrings.</p> <p>Make sure the docstrings include:</p> <ul style="list-style-type: none"><li>- A concise function description.</li><li>- Parameter descriptions with type hints.</li><li>- Return type and description (if applicable).</li><li>- Any other important information previously present in the comments.</li></ul> <p>Remove the inline comments once they are moved to the docstrings.</p> <p>Return the updated Python code with the new docstrings.</p> <p><b>CODE:-</b></p>	
--	---	--

```
9.1task4.py > ...
Zencoder
1 def fibonacci(n: int) -> int:
2     """Calculates the nth Fibonacci number using recursion.
3
4     Args:
5         n (int): The position in the Fibonacci sequence.
6
7     Returns:
8         int: The Fibonacci number at position n.
9
10    """
11    if n <= 1:
12        return n
13    return fibonacci(n - 1) + fibonacci(n - 2)
14
15
Zencoder
16 def is_prime(num: int) -> bool:
17     """Determines whether a given number is prime.
18
19     Args:
20         num (int): The number to check for primality.
21
22     Returns:
23         bool: True if num is prime, False otherwise.
24
25    """
26    if num <= 1:
```

```
 9.1task4.py > ...
1   def fibonacci(n: int) -> int:
2       return n
3       return fibonacci(n - 1) + fibonacci(n - 2)
4
5
6   Zencoder
7   def is_prime(num: int) -> bool:
8       """Determines whether a given number is prime.
9
10      Args:
11          num (int): The number to check for primality.
12
13      Returns:
14          bool: True if num is prime, False otherwise.
15
16      """
17
18      if num <= 1:
19          return False
20      for i in range(2, int(num ** 0.5) + 1):
21          if num % i == 0:
22              return False
23      return True
24
25
```

## OUTPUT:-

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS
• b9.1.py
• PS C:\Users\Administrator\OneDrive\ai> & c:/Python313/python.exe c:/Users/Administrator/OneDrive/ai/la
b9.1.py
• PS C:\Users\Administrator\OneDrive\ai> & c:/Python313/python.exe c:/Users/Administrator/OneDrive/ai/la
b9.1.py
• PS C:\Users\Administrator\OneDrive\ai> & c:/Python313/python.exe c:/Users/Administrator/OneDrive/ai/9.
1task2.py
• PS C:\Users\Administrator\OneDrive\ai> []
```

## OBSERVATIONS:-

	<p><input checked="" type="checkbox"/> <b>Positive Observations</b></p> <ul style="list-style-type: none"> <li>• <b>Improved Documentation Consistency:</b> Documentation is centralized at the start of each function, improving readability and automated doc tools' compatibility.</li> <li>• <b>Better Structured Information:</b> Docstrings clearly segment description, arguments, and return values, making it easier to understand usage.</li> <li>• <b>Removal of Redundant Comments:</b> Inline comments that clutter the code are removed, resulting in cleaner, more maintainable code.</li> <li>• <b>Preserved Meaning and Context:</b> No information is lost; all relevant insights from comments are preserved in the docstrings.</li> </ul> <p><b>Suggestions</b></p> <ul style="list-style-type: none"> <li>• In complex functions, consider expanding docstrings with <b>exceptions raised</b> or <b>side effects</b>.</li> <li>• If comments included example usage or warnings, those could be added as <code>Raises:</code> or <code>Notes:</code> sections in docstrings.</li> <li>• Consistency in terminology and formatting across multiple functions enhances overall module documentation quality.</li> </ul> <hr/> <p><b>TaskDescription#5(Documentation—ReviewandCorrect Docsstrings)</b></p> <ul style="list-style-type: none"> <li>• <b>Task:</b> Use AI to identify and correct inaccuracies in existing docstrings.</li> <li>• <b>Instructions:</b> <ul style="list-style-type: none"> <li>○ Provide Python code without dated or incorrect docstrings.</li> <li>○ Instruct AI to rewrite each docstring to match the current code behavior.</li> <li>○ Ensure corrections follow Google-style formatting.</li> </ul> </li> <li>• <b>ExpectedOutput#5:</b> <ul style="list-style-type: none"> <li>○ Python file with updated, accurate, and standardized docstrings.</li> </ul> </li> </ul> <p><b>PROMPT:-</b></p> <p>I'm providing a Python script where some functions have outdated or incorrect docstrings.</p> <p>Please carefully review and **rewrite each docstring** so that it accurately reflects the function's current behavior.</p> <p>Follow the **Google-style** docstring format, and ensure that:</p> <ul style="list-style-type: none"> <li>- The function description is correct.</li> </ul>	
--	---	--

- Parameters and return types are accurate and fully described.
- Any removed or changed functionality is no longer referenced.
- The formatting is clean and consistent.

Do not change the function code — only correct the docstrings. Return the updated Python code.

### CODE:-

```

9.1task5.py > ...
Zencoder
1  def divide(a: int, b: int) -> float:
2      """Multiplies two numbers.
3
4      Args:
5          a (int): The numerator.
6          b (int): The denominator.
7
8      Returns:
9          float: The product of the numbers.
10     """
11    return a / b
12
Zencoder
13 def get_even_numbers(nums: list[int]) -> list[int]:
14     """Filters odd numbers from the list.
15
16     Args:
17         nums (list[int]): A list of integers.
18
19     Returns:
20         list[int]: A list of even numbers from the input.
21     """
22    return [n for n in nums if n % 2 == 0]
23
Zencoder
24 def greet(name: str) -> None:
25     """Returns a greeting string for the user.

```

```
15
16     Args:
17         nums (list[int]): A list of integers.
18
19     Returns:
20         list[int]: A list of even numbers from the input.
21         .....
22     return [n for n in nums if n % 2 == 0]
23
24     Zencoder
25     def greet(name: str) -> None:
26         """Returns a greeting string for the user.
27
28         Args:
29             name (str): The name of the user.
30
31         Returns:
32             str: The greeting message.
33             .....
34         print(f"Hello, {name}!")
```

## OUTPUT:-

```
PS C:\Users\Administrator\OneDrive\ai> & C:/Python313/python.exe c:/Users/Administrator/OneDrive/ai/9.
1task5.py
PS C:\Users\Administrator\OneDrive\ai> []
```

	<p><input checked="" type="checkbox"/> <b>Positive Outcomes</b></p> <ul style="list-style-type: none"> <li>• <b>Errors Corrected:</b> All docstrings now accurately describe the function behavior. For example:           <ul style="list-style-type: none"> <li>• <code>divide</code> now correctly says "divides" instead of "multiplies"</li> <li>• <code>greet</code> correctly indicates it <code>prints</code> the message instead of <code>returns</code> it</li> </ul> </li> <li>• <b>Google Style Followed:</b> Format is consistent, with sections for:           <ul style="list-style-type: none"> <li>• Description</li> <li>• Args</li> <li>• Returns</li> <li>• Raises (where applicable)</li> </ul> </li> <li>• <b>Improved Clarity and Accuracy:</b> The return values and side effects are now explicitly and correctly documented.</li> <li>• <b>No Code Changes:</b> The function logic is untouched, ensuring stability.</li> </ul> <hr/> <p><b>! Suggestions for Further Enhancement</b></p> <ul style="list-style-type: none"> <li>• <b>Add Examples (Optional):</b> Could include <code>Example:</code> sections for usage clarity.</li> <li>• <b>Explicit <code>Raises</code> Section in All Error-Prone Functions:</b> Good practice to document exceptions raised (e.g., <code>ZeroDivisionError</code> in <code>divide</code>).</li> </ul> <p><b>OBSERVATIONS:-</b></p>	
	<p><b>TaskDescription#6(Documentation–PromptComparison Experiment)</b></p> <ul style="list-style-type: none"> <li>• Task: Compared documentation output from a vague prompt and a detailed prompt for the same Python function.</li> <li>• Instructions:           <ul style="list-style-type: none"> <li>○ Create two prompts: one simple ("Add comments to this function") and one detailed ("Add Google-style docstrings with parameters, return types, and examples").</li> <li>○ Use AI to process the same Python function with both prompts.</li> <li>○ Analyze and record differences in quality, accuracy, and completeness.</li> </ul> </li> </ul>	

## PROMPTS:-

### ◆ VaguePrompt

Add comments to this function.

### ◆ DetailedPrompt

Please add a structured Google-style docstring to the following Python function. Make sure to include:

- A concise description of the function's purpose
- Parameters with type hints and descriptions
- Return type and its description
- An example usage (without actual output values)

Use proper formatting for Google-style docstrings

## CODE:-

### PYTHON FUNCTION CODE:

```
9.1task6.py > ...
1  def factorial(n):
2      if n < 0:
3          raise ValueError("Input must be a non-negative integer.")
4      if n == 0 or n == 1:
5          return 1
6      return n * factorial(n - 1)
7
8
9
```

### CODE FOR VAGUE PROMPT:

```
10 def factorial(n):
11     # Check if the input is negative
12     if n < 0:
13         raise ValueError("Input must be a non-negative integer.")
14     # Base case: factorial of 0 or 1 is 1
15     if n == 0 or n == 1:
16         return 1
17     # Recursive call to calculate factorial
18     return n * factorial(n - 1)
19
20
21
```

## CODEFORDETAILED PROMPT:-

```
22 | Zencoder
23 | def factorial(n: int) -> int:
24 |     """Calculates the factorial of a non-negative integer using recursion.
25 |
26 |     Args:
27 |         n (int): A non-negative integer whose factorial is to be computed.
28 |
29 |     Returns:
30 |         int: The factorial of the input number.
31 |
32 |     Raises:
33 |         ValueError: If the input is a negative integer.
34 |
35 |     Example:
36 |         result = factorial(5)
37 |         """
38 |
39 |     if n < 0:
40 |         raise ValueError("Input must be a non-negative integer.")
41 |     if n == 0 or n == 1:
42 |         return 1
43 |     return n * factorial(n - 1)
```

## OUTPUT:-

```
PS C:\Users\Administrator\OneDrive\ai> & c:/Python313/python.exe c:/Users/Administrator/OneDrive/ai/9.
1task6.py
PS C:\Users\Administrator\OneDrive\ai> []
```

	<p><b>OBSERVATIONS:-</b></p> <p><input checked="" type="checkbox"/> <b>Observations</b></p> <p><input checked="" type="checkbox"/> <b>Key Takeaways:</b></p> <ol style="list-style-type: none"><li><b>1. Prompt Specificity Directly Impacts Quality</b> The vague prompt resulted in basic, surface-level inline comments, while the detailed prompt produced comprehensive, standardized documentation.</li><li><b>2. Structured Docstrings Are More Scalable</b> The detailed prompt yields output that is better suited for:<ul style="list-style-type: none"><li>• API documentation tools (e.g., Sphinx, Doxygen)</li><li>• Collaborative projects</li><li>• Long-term maintenance</li></ul></li><li><b>3. Type Hints and Exceptions Are Missed in Vague Prompts</b> Important details like parameter types, exceptions (<code>ValueError</code>), and return values were <b>not captured</b> in the vague prompt output.</li><li><b>4. Vague Prompts Encourage Reactive Documentation</b> Comments were simply describing what each line does. This does not help users understand how to use the function or its intent.</li><li><b>5. Detailed Prompt Encourages Declarative, Reusable Documentation</b> The output from the detailed prompt can be reused in documentation systems and easily interpreted by future developers.</li></ol>	
--	---	--