

Assignment Number: 9.1 (Present assignment number) / **24** (Total number of assignments)

NAME:- A.SREEMANTH REDDY

ROLLNO:-2403A51OG1

Q.No.	Question	Expected Time to complete
1	<p>Lab9–Documentation Generation: Automatic Documentation and Code Comments</p> <p>Lab Objectives</p> <ul style="list-style-type: none">• To use AI-assisted coding tools for generating Python documentation and code comments.• To apply zero-shot, few-shot, and context-based prompt engineering for documentation creation.• To practice generating and refining docstrings, inline comments, and module-level documentation.• To compare outputs from different prompting styles for quality analysis. <hr/> <p>Task Description #1 (Documentation–Google-Style Docstrings for Python Functions)</p> <ul style="list-style-type: none">• Task: Use AI to add Google-styled docstring to all functions in a given Python script.• Instructions:<ul style="list-style-type: none">○ Prompt AI to generate docstrings without providing any input-output examples.○ Ensure each docstring includes:<ul style="list-style-type: none">▪ Function description▪ Parameters with type hints▪ Return values with type hints▪ Example usage○ Review the generated docstrings for accuracy and formatting.	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 a usage line 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:
28     """
29     Example:
30     | result = is_even(10)
31     | """
32     return number % 2 == 0
33
34 Zencoder
35 def greet_user(name: str, greeting: str = "Hello") -> str:
36     """Generates a greeting message for the user.
37
38     Args:
39     | name (str): The name of the user.
40     | greeting (str, optional): Custom greeting to use. Defaults to "Hello".
41
42     Returns:
43     | str: A formatted greeting message.
44
45     Example:
46     | message = greet_user("Alice", "Hi")
47     | """
48     return f"{greeting}, {name}!"
49
50 Zencoder
51 def calculate_area(length: float, width: float) -> float:
52     """Calculates the area of a rectangle.
53
54     Args:
55     | length (float): The length of the rectangle.
56     | width (float): The width of the rectangle.
57
58     Returns:
59     | float: The area of the rectangle.
60
61     Example:
62     | area = calculate_area(5.0, 3.0)
63     | """
64     return length * width
65
66 Zencoder
67 def factorial(n: int) -> int:
68     """Calculates the factorial of a non-negative integer.
69
70     Args:
71     | n (int): A non-negative integer.
72
73     Returns:
74     | int: The factorial of the input number.
75
76     Raises:
77     | ValueError: If n is negative.
78
79     Example:
80     | result = factorial(5)
81     | """
82     if n < 0:
83         raise ValueError("Input must be a non-negative integer.")
84     if n == 0 or n == 1:
85         return 1
86     return n * factorial(n - 1)

```

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/
b9.1.py
PS C:\Users\Administrator\OneDrive\ai> |
```

OBSERVATIONS:-

✓ Positive Observations

1. Correct Docstring Format (Google Style):

All docstrings follow the Google-style format:

- A brief function description
- Clearly labeled `Args`, `Returns`, and `Example` sections

2. Use of Type Hints:

Parameters and return values include appropriate Python type hints, making the documentation more useful and readable.

3. Consistent Structure:

All functions maintain a consistent structure for their docstrings, which is key for maintainability and scalability in a codebase.

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.

5. Edge Case Consideration (e.g., `factorial`):

The `factorial` function includes a `Raises` section to document error handling (`ValueError`), which is a best practice for documenting functions that raise exceptions.

TaskDescription#2(Documentation–InlineCommentsforComplex Logic)

- Task: Use AI to add meaningful inline comments to a Python program explaining only complex logic parts.
- Instructions:
 - Provide a Python script without comments to the AI.
 - Instruct AI to skip obvious syntax explanations and focus only on tricky or non-intuitive code sections.
 - Verify that comments improve code readability and maintainability.

◆ 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 Zencoder
44 def quicksort(arr: list[int]) -> list[int]:
45     """
46

```

```

9.1task2.py > ...
47
48 Zencoder
49 def quicksort(arr: list[int]) -> list[int]:
50     """
51     Sorts the array using the quicksort algorithm.
52     """
53     if len(arr) <= 1:
54         return arr
55     pivot = arr[len(arr) // 2]
56
57     # Partitioning the array into three based on pivot
58     left = [x for x in arr if x < pivot]      # Elements less than pivot
59     middle = [x for x in arr if x == pivot]  # Elements equal to pivot
60     right = [x for x in arr if x > pivot]    # Elements greater than pivot
61
62     # Recursively apply quicksort to partitions
63     return quicksort(left) + middle + quicksort(right)
64
65
66 Zencoder
67 def find_peak_element(nums: list[int]) -> int:
68     """
69     Finds a peak element in the list and returns its index.
70     A peak is an element that is greater than its neighbors.
71     """
72     left, right = 0, len(nums) - 1
73
74     # Binary search for a peak element
75     while left < right:

```

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

✓ 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-linedocstring at the top of the file.
 - Ensure the docstring clearly describes functionality and usage without rewriting the entire code.
- Expected Output#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 docstrings should be a single multi-lined 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:-

```
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
```

```
9.1TASK3.py > ...
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
29 Zencoder
30 class Calculator:
31     Zencoder
32     def __init__(self):
33         pass
34
35     Zencoder
36     def add(self, a: int, b: int) -> int:
37         return a + b
38
39     Zencoder
40     def multiply(self, a: int, b: int) -> int:
41         return a * b
```

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:-

- ✓ 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–ConvertCommentsto Structured Docstrings)

- Task:UseAIto transformexistinginlinecommentsinto structured function docstrings following Google style.
- Instructions:
 - ProvideAIwithPythoncodecontaininginlinecomments.
 - AskAItomoverelevantdetailsfromcommentsinto function docstrings.
 - Verifythatthenewdocstringskeepth meaningintact while improving structure.

	<p>PROMPT:-</p> <p>Iamproviding youwithaPythonscriptthatcontainsinlinecomments inside functions.</p> <p>Pleasetransformtheseinlinecommentsintowell-structured**Google- style docstrings** for each function, moving all relevant information from the comments into the docstrings.</p> <p>Makesurethedocstringsinclude:</p> <ul style="list-style-type: none">- Aconcisefunctiondescription.- Parameterdescriptionswithtypehints.- Returntypeanddescription(ifapplicable).- Anyotherimportantinformationpreviouslypresentinthecomments. <p>Remove the inline comments once they are moved to the docstrings.</p> <p>Returnthe updatedPythoncode withthe new docstrings.</p> <p>CODE:-</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:
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:
27         return False
28     for i in range(2, int(num ** 0.5) + 1):
29         if num % i == 0:
30             return False
31     return True
32

```

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:-

✔ **Positive Observations**

- **Improved Documentation Consistency:**

Documentation is centralized at the start of each function, improving readability and automated doc tools' compatibility.

- **Better Structured Information:**

Docstrings clearly segment description, arguments, and return values, making it easier to understand usage.

- **Removal of Redundant Comments:**

Inline comments that clutter the code are removed, resulting in cleaner, more maintainable code.

- **Preserved Meaning and Context:**

No information is lost; all relevant insights from comments are preserved in the docstrings.

! **Suggestions**

- In complex functions, consider expanding docstrings with **exceptions raised** or **side effects**.
- If comments included example usage or warnings, those could be added as `Raises:` or `Notes:` sections in docstrings.
- Consistency in terminology and formatting across multiple functions enhances overall module documentation quality.

TaskDescription#5(Documentation–ReviewandCorrect Docstrings)

- **Task:** Use AI to identify and correct inaccuracies in existing docstrings.
- **Instructions:**
 - Provide Python code without dated or incorrect docstrings.
 - Instruct AI to rewrite each docstring to match the current code behavior.
 - Ensure corrections follow Google-style formatting.
- **Expected Output#5:**
 - Python file with updated, accurate, and standardized docstrings.

PROMPT:-

I'm providing a Python script where some functions have outdated or incorrect docstrings.

Please carefully review and ****rewrite each docstring**** so that it accurately reflects the function's current behavior.

Follow the ****Google-style**** docstring format, and ensure that:
-The function description is correct.

- 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
13
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
24
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> █
```


✔ Positive Outcomes

- Errors Corrected:

All docstrings now accurately describe the function behavior. For example:

- `divide` now correctly says "divides" instead of "multiplies"
- `greet` correctly indicates it **prints** the message instead of **returns** it

- Google Style Followed:

Format is consistent, with sections for:

- Description
- Args
- Returns
- Raises (where applicable)

- Improved Clarity and Accuracy:

The return values and side effects are now explicitly and correctly documented.

- No Code Changes:

The function logic is untouched, ensuring stability.

! Suggestions for Further Enhancement

- Add Examples (Optional):

Could include `Example:` sections for usage clarity.

- Explicit `Raises` Section in All Error-Prone Functions:

Good practice to document exceptions raised (e.g., `ZeroDivisionError` in `divide`).

OBSERVATIONS:-

TaskDescription#6(Documentation–PromptComparison Experiment)

- Task: Compared documentation output from a vague prompt and a detailed prompt for the same Python function.
- Instructions:
 - Create two prompts: one simple (“Add comments to this function”) and one detailed (“Add Google-style docstrings with parameters, return types, and examples”).
 - Use AI to process the same Python function with both prompts.
 - Analyze and record differences in quality, accuracy, and completeness.

PROMPTS:-

☐ **Vague Prompt**

Add comments to this function.

☐ **Detailed Prompt**

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     if n < 0:  
39         raise ValueError("Input must be a non-negative integer.")  
40     if n == 0 or n == 1:  
41         return 1  
42     return n * factorial(n - 1)  
43
```

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> |
```

OBSERVATIONS:-

Observations

Key Takeaways:

- 1. Prompt Specificity Directly Impacts Quality
The vague prompt resulted in basic, surface-level inline comments, while the detailed prompt produced comprehensive, standardized documentation.
- 2. Structured Docstrings Are More Scalable
The detailed prompt yields output that is better suited for:
 - API documentation tools (e.g., Sphinx, Doxygen)
 - Collaborative projects
 - Long-term maintenance
- 3. Type Hints and Exceptions Are Missed in Vague Prompts
Important details like parameter types, exceptions (`ValueError`), and return values were **not captured** in the vague prompt output.
- 4. Vague Prompts Encourage Reactive Documentation
Comments were simply describing what each line does. This does not help users understand how to use the function or its intent.
- 5. Detailed Prompt Encourages Declarative, Reusable Documentation
The output from the detailed prompt can be reused in documentation systems and easily interpreted by future developers.