Name:Guggilla Anuja id:2403A51101 batch:06

SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE		DEPARTMENT OF COMPUTER SCIENCE ENGINEERING	
Program Name: B. Tech		**Assignment Type: Lab**	**Academic Year:**2025-2026
Course Coordinator Name	Venkataramana Veeramsetty		
Instructor(s) Name	Dr. V. Venkataramana (Co-ordinator) Dr. T. Sampath Kumar Dr. Pramoda Patro Dr. Brij Kishor Tiwari Dr.J.Ravichander Dr. Mohammand Ali Shaik Dr. Anirodh Kumar Mr. S.Naresh Kumar Dr. RAJESH VELPULA Mr. Kundhan Kumar Ms. Ch.Rajitha Mr. M Prakash Mr. B.Raju Intern 1 (Dharma teja) Intern 2 (Sai Prasad) Intern 3 (Sowmya) NS_2 (Mounika)		
Course Code	24CS002PC215	**Course Title**	AI Assisted Coding
Year/Sem	II/I	**Regulation**	R24
Date and Day of Assignment	Week5 - Monday	**Time(s)**	
Duration	2 Hours	**Applicable to Batches**	
AssignmentNumber: 9.1(Present assignment number)/**24**(Total number of assignments)			

Q.No.	Question	*Expected Time to complete*
1	**Lab 9 – Documentation Generation: Automatic Documentation and Code Comments** **Lab Objectives** • To use AI-assisted coding tools for generating Python documentation and code comments. • To apply zero-shot, few-shot, and context-based prompt	Week5 - Monday

	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.	

	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.
	• Expected Output #1:
	  o A Python script with all functions documented using correctly formatted Google-style docstrings.
	
	Prompt:
	Add Google-style docstrings to all functions in the given Python script .Each docstring should include a brief description of what the function does, all parameters with type hints and descriptions, the return value with type hint and description, and an example usage of the function. Ensure that all docstrings are properly formatted according to the Google Python Style Guide. Output the complete Python script with the docstrings added to every function.
	CODE:	

2-1 > ai > docstring.py > ...

```python
def add_numbers(a: int, b: int) -> int:
    """Adds two integers and returns the result.

    Args:
        a (int): The first integer to add.
        b (int): The second integer to add.

    Returns:
        int: The sum of a and b.

    Example:
        >>> add_numbers(2, 3)
        5
    """
    return a + b
def is_even(n: int) -> bool:
    """Checks if a number is even.

    Args:
        n (int): The number to check.

    Returns:
        bool: True if n is even, False otherwise.

    Example:
        >>> is_even(4)
        True
        >>> is_even(5)
        False
    """
    return n % 2 == 0
def factorial(n: int) -> int:
    """Calculates the factorial of a non-negative integer.

    Args:
        n (int): The non-negative integer to compute the factorial of.
```

2-1 > ai > docstring.py > ...

```python
def factorial(n: int) -> int:
    Returns:
        int: The factorial of n.

    Example:
        >>> factorial(5)
        120
    """
    if n < 0:
        raise ValueError("n must be a non-negative integer")
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result
def reverse_string(s: str) -> str:
    """Reverses the given string.

    Args:
        s (str): The string to reverse.

    Returns:
        str: The reversed string.

    Example:
        >>> reverse_string("hello")
        'olleh'
    """
    return s[::-1]
def average(numbers: list[float]) -> float:
    """Calculates the average of a list of numbers.

    Args:
        numbers (list[float]): A list of numbers.

    Returns:
        float: The average of the numbers.
```

```
73
74      Example:
75          >>> average([1.0, 2.0, 3.0])
76          2.0
77      """
78      if not numbers:
79          raise ValueError("The list of numbers cannot be empty")
80      return sum(numbers) / len(numbers)
81  # Function calls with example outputs
82  print("add_numbers(2, 3):", add_numbers(2, 3))
83  print("is_even(4):", is_even(4))
84  print("is_even(5):", is_even(5))
85  print("factorial(5):", factorial(5))
86  print("reverse_string('hello'):", reverse_string("hello"))
87  print("average([1.0, 2.0, 3.0]):", average([1.0, 2.0, 3.0]))
```

OUTPUT:

```
PS C:\Users\allur\Desktop\B.Tech> python -u "c:\Users\allur\Desktop\B.Tech\2-1\ai\docstring.py"
add_numbers(2, 3): 5
is_even(4): True
is_even(5): False
factorial(5): 120
reverse_string('hello'): olleh
average([1.0, 2.0, 3.0]): 2.0
PS C:\Users\allur\Desktop\B.Tech>
```

OBSERVATION:

Google-style docstrings were added to all functions, clearly describing their purpose, parameters with type hints, return values, and example usage. The script is now more readable, maintainable, and consistent, with no changes to the original functionality.

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:
  - 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.
- Expected Output #2:
  - Python code with concise, context-aware inline comments for complex logic blocks.

PROMPT:

CODE:

2-1 > ai > docstring.py > ...

```python
def fibonacci(n: int) -> list[int]:
    """Generates the Fibonacci sequence up to the nth term.

    Args:
        n (int): The number of terms to generate.

    Returns:
        list[int]: A list containing the Fibonacci sequence up to n term

    Example:
        >>> fibonacci(5)
        [0, 1, 1, 2, 3]
    """
    if n <= 0:
        return []
    sequence = [0]
    if n == 1:
        return sequence
    sequence.append(1)
    for i in range(2, n):
        # Each term is the sum of the two preceding terms
        sequence.append(sequence[-1] + sequence[-2])
    return sequence


def find_max(numbers: list[int]) -> int:
    """Finds the maximum value in a list of integers.

    Args:
        numbers (list[int]): The list of integers.

    Returns:
        int: The maximum integer in the list.

    Example:
        >>> find_max([1, 5, 3, 9, 2])
        9
```

2-1 > ai > docstring.py > ...

```python
def find_max(numbers: list[int]) -> int:
    if not numbers:
        raise ValueError("The list cannot be empty")
    max_num = numbers[0]
    for num in numbers[1:]:
        if num > max_num:
            max_num = num  # Update max_num if a larger value is found
    return max_num


def count_vowels(s: str) -> int:
    """Counts the number of vowels in a string.

    Args:
        s (str): The input string.

    Returns:
        int: The number of vowels in the string.

    Example:
        >>> count_vowels("hello world")
        3
    """
    vowels = set("aeiouAEIOU")
    count = 0
    for char in s:
        if char in vowels:
            count += 1  # Increment count for each vowel found
    return count


def merge_dicts(dict1: dict, dict2: dict) -> dict:
    """Merges two dictionaries into one. If there are duplicate keys, values from dict2 overwrite dict1.

    Args:
        dict1 (dict): The first dictionary.
        dict2 (dict): The second dictionary.

    Returns:
        dict: The merged dictionary.

    Example:
        >>> merge_dicts({'a': 1, 'b': 2}, {'b': 3, 'c': 4})
        {'a': 1, 'b': 3, 'c': 4}
    """
    merged = dict1.copy()  # Start with a copy to avoid mutating the original
    merged.update(dict2)   # Overwrite with dict2's values where keys overlap
    return merged


def unique_elements(lst: list) -> list:
    """Returns a list of unique elements, preserving the original order.

    Args:
        lst (list): The input list.

    Returns:
        list: A list containing only unique elements from the input, in order of first appearance.

    Example:
        >>> unique_elements([1, 2, 2, 3, 1, 4])
        [1, 2, 3, 4]
    """
    seen = set()
    unique = []
    for item in lst:
        if item not in seen:
            seen.add(item)  # Track seen elements to ensure uniqueness
            unique.append(item)
    return unique
```

```
# Example function calls
print("fibonacci(7):", fibonacci(7))
print("find_max([10, 3, 7, 22, 5]):", find_max([10, 3, 7, 22, 5]))
print("count_vowels('Artificial Intelligence'):", count_vowels("Artificial Intelligence"))
print("merge_dicts({'x': 1, 'y': 2}, {'y': 3, 'z': 4}):", merge_dicts({'x': 1, 'y': 2}, {'y': 3, 'z': 4}))
print("unique_elements([1, 2, 2, 3, 1, 4]):", unique_elements([1, 2, 2, 3, 1, 4]))
```

OUTPUT:

```
PS C:\Users\allur\Desktop\B.Tech> python -u "c:\Users\allur\Desktop\B.Tech\2-1\ai\docstring.py"
fibonacci(7): [0, 1, 1, 2, 3, 5, 8]
find_max([10, 3, 7, 22, 5]): 22
count_vowels('Artificial Intelligence'): 10
merge_dicts({'x': 1, 'y': 2}, {'y': 3, 'z': 4}): {'x': 1, 'y': 3, 'z': 4}
unique_elements([1, 2, 2, 3, 1, 4]): [1, 2, 3, 4]
PS C:\Users\allur\Desktop\B.Tech>
```

OBSERVATION:

Meaningful inline comments were added to only the complex or non-intuitive parts of the Python script. These comments clarify tricky logic, such as Fibonacci term calculation, updating the maximum value, set usage for fast membership checks, preserving order in unique elements, and avoiding dictionary mutation. The script's readability and maintainability are improved without cluttering straightforward code.

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:
  - 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.
- Expected Output #3:
  - A complete, clear, and concise module-level docstring at the beginning of the file.

PROMPT:

create a module-level docstring for the following Python file. The docstring should be a single multi-line comment at the top of the file that summarizes the purpose of the module, its dependencies, and the main functions or classes included. Do not rewrite the code or change its logic. Ensure the docstring clearly describes the functionality and how the module can be used. Output the complete Python file with the module-level docstring added at the beginning.

CODE:

2-1 > ai > docstring.py > ...

```python
def gcd(a: int, b: int) -> int:
    """Computes the greatest common divisor (GCD) of two integers using the Euclidean algorithm.

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

    Returns:
        int: The greatest common divisor of a and b.

    Example:
        >>> gcd(48, 18)
        6
    """
    while b != 0:
        a, b = b, a % b  # Replace a with b and b with the remainder
    return abs(a)


def flatten(nested_list: list) -> list:
    """Flattens a nested list into a single list of values.

    Args:
        nested_list (list): A list that may contain other lists as elements.

    Returns:
        list: A flat list containing all values from the nested structure.

    Example:
        >>> flatten([1, [2, [3, 4], 5], 6])
        [1, 2, 3, 4, 5, 6]
    """
    flat = []
    for item in nested_list:
        if isinstance(item, list):
            flat.extend(flatten(item))  # Recursively flatten sublists
        else:
```

2-1 > ai > docstring.py > flatten

```python
def flatten(nested_list: list) -> list:
            else:
                flat.append(item)
        return flat


def is_palindrome(s: str) -> bool:
    """Checks if a string is a palindrome, ignoring case and non-alphanumeric characters.

    Args:
        s (str): The string to check.

    Returns:
        bool: True if s is a palindrome, False otherwise.

    Example:
        >>> is_palindrome("A man, a plan, a canal: Panama")
        True
    """
    filtered = ''.join(c.lower() for c in s if c.isalnum())
    return filtered == filtered[::-1]


def transpose_matrix(matrix: list[list]) -> list[list]:
    """Transposes a 2D matrix (list of lists).

    Args:
        matrix (list[list]): The matrix to transpose.

    Returns:
        list[list]: The transposed matrix.

    Example:
        >>> transpose_matrix([[1, 2], [3, 4], [5, 6]])
        [[1, 3, 5], [2, 4, 6]]
    """
    if not matrix or not matrix[0]:
```

2-1 > ai > docstring.py > transpose_matrix

```python
def transpose_matrix(matrix: list[list]) -> list[list]:
    """
    if not matrix or not matrix[0]:
        return []
    # Use zip to transpose rows and columns
    return [list(row) for row in zip(*matrix)]


def chunk_list(lst: list, size: int) -> list[list]:
    """Splits a list into chunks of a specified size.

    Args:
        lst (list): The list to split.
        size (int): The size of each chunk.

    Returns:
        list[list]: A list of chunks, where each chunk is a list.

    Example:
        >>> chunk_list([1, 2, 3, 4, 5], 2)
        [[1, 2], [3, 4], [5]]
    """
    if size <= 0:
        raise ValueError("Chunk size must be positive")
    return [lst[i:i + size] for i in range(0, len(lst), size)]


# Example function calls
print("gcd(48, 18):", gcd(48, 18))
print("flatten([1, [2, [3, 4], 5], 6]):", flatten([1, [2, [3, 4], 5], 6]))
print("is_palindrome('A man, a plan, a canal: Panama'):", is_palindrome("A man, a plan, a canal: Panama"))
print("transpose_matrix([[1, 2], [3, 4], [5, 6]]):", transpose_matrix([[1, 2], [3, 4], [5, 6]]))
print("chunk_list([1, 2, 3, 4, 5], 2):", chunk_list([1, 2, 3, 4, 5], 2))
```

OUTPUT:

```
PS C:\Users\allur\Desktop\B.Tech> python -u "c:\Users\allur\Desktop\B.Tech\2-1\ai\docstring.py"
gcd(48, 18): 6
flatten([1, [2, [3, 4], 5], 6]): [1, 2, 3, 4, 5, 6]
is_palindrome('A man, a plan, a canal: Panama'): True
transpose_matrix([[1, 2], [3, 4], [5, 6]]): [[1, 3, 5], [2, 4, 6]]
chunk_list([1, 2, 3, 4, 5], 2): [[1, 2], [3, 4], [5]]
```

OBSERVATION:

The module-level docstring summarizes the file's purpose, main functions, and dependencies, including utilities for math (gcd), lists (flatten, chunk_list), strings (is_palindrome), and matrices (transpose_matrix). It improves readability, aids maintainability, and provides developers a clear overview of the module's functionality without needing to examine each function individually.

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:
  - 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.
- Expected Output #4:
  - Python code with comments replaced by clear, standardized docstrings.

PROMPT:

Convert the inline comments in the Python script into Google-style docstrings for each function. Preserve the meaning while structuring the information clearly. Each docstring should include a description, parameters with type hints, return values with type hints, and example usage if relevant. Output the complete Python code with comments replaced by well-formatted, standardized docstrings without changing the original functionality.

CODE:

```
> ai > docstring.py > chunk_list
def gcd(a: int, b: int) -> int:
    """Computes the greatest common divisor (GCD) of two integers using the Euclidean algorithm.

    The function repeatedly replaces the pair (a, b) with (b, a % b) until b becomes zero.
    The absolute value of a at this point is the GCD.

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

    Returns:
        int: The greatest common divisor of a and b.

    Example:
        >>> gcd(48, 18)
        6
    """
    while b != 0:
        a, b = b, a % b
    return abs(a)


def flatten(nested_list: list) -> list:
    """Flattens a nested list into a single list of values.

    Recursively traverses the input list and any sublists, collecting all non-list elements
    into a single flat list.

    Args:
        nested_list (list): A list that may contain other lists as elements.

    Returns:
        list: A flat list containing all values from the nested structure.

    Example:
        >>> flatten([1, [2, [3, 4], 5], 6])
        [1, 2, 3, 4, 5, 6]
```

```python
def flatten(nested_list: list) -> list:
    """
    flat = []
    for item in nested_list:
        if isinstance(item, list):
            flat.extend(flatten(item))
        else:
            flat.append(item)
    return flat


def is_palindrome(s: str) -> bool:
    """Checks if a string is a palindrome, ignoring case and non-alphanumeric character

    The function filters out non-alphanumeric characters and compares the string to its

    Args:
        s (str): The string to check.

    Returns:
        bool: True if s is a palindrome, False otherwise.

    Example:
        >>> is_palindrome("A man, a plan, a canal: Panama")
        True
    """
    filtered = ''.join(c.lower() for c in s if c.isalnum())
    return filtered == filtered[::-1]


def transpose_matrix(matrix: list[list]) -> list[list]:
    """Transposes a 2D matrix (list of lists).

    Converts rows to columns and columns to rows using the zip function.

    Args:
        matrix (list[list]): The matrix to transpose.
```

```python
def transpose_matrix(matrix: list[list]) -> list[list]:
    Returns:
        list[list]: The transposed matrix.

    Example:
        >>> transpose_matrix([[1, 2], [3, 4], [5, 6]])
        [[1, 3, 5], [2, 4, 6]]
    """
    if not matrix or not matrix[0]:
        return []
    return [list(row) for row in zip(*matrix)]


def chunk_list(lst: list, size: int) -> list[list]:
    """Splits a list into chunks of a specified size.

    Divides the input list into sublists, each of length 'size'. The last chunk may be shorter
    if the list length is not a multiple of the chunk size.

    Args:
        lst (list): The list to split.
        size (int): The size of each chunk.

    Returns:
        list[list]: A list of chunks, where each chunk is a list.

    Raises:
        ValueError: If the chunk size is not positive.

    Example:
        >>> chunk_list([1, 2, 3, 4, 5], 2)
        [[1, 2], [3, 4], [5]]
    """
    if size <= 0:
        raise ValueError("Chunk size must be positive")
    return [lst[i:i + size] for i in range(0, len(lst), size)]
```

```python
# Example function calls
print("gcd(48, 18):", gcd(48, 18))
print("flatten([1, [2, [3, 4], 5], 6]):", flatten([1, [2, [3, 4], 5], 6]))
print("is_palindrome('A man, a plan, a canal: Panama'):", is_palindrome("A man, a plan, a canal: Panama"))
print("transpose_matrix([[1, 2], [3, 4], [5, 6]]):", transpose_matrix([[1, 2], [3, 4], [5, 6]]))
print("chunk_list([1, 2, 3, 4, 5], 2):", chunk_list([1, 2, 3, 4, 5], 2))
```

OUTPUT:

```
PS C:\Users\allur\Desktop\B.Tech> python -u "c:\Users\allur\Desktop\B.Tech\2-1\ai\docstring.py"
gcd(48, 18): 6
flatten([1, [2, [3, 4], 5], 6]): [1, 2, 3, 4, 5, 6]
is_palindrome('A man, a plan, a canal: Panama'): True
transpose_matrix([[1, 2], [3, 4], [5, 6]]): [[1, 3, 5], [2, 4, 6]]
chunk_list([1, 2, 3, 4, 5], 2): [[1, 2], [3, 4], [5]]
PS C:\Users\allur\Desktop\B.Tech>
```

OBSERVATION:

All inline comments have been successfully converted into structured Google-style docstrings. Each function now clearly explains its purpose, logic, parameters, return values, and provides example usage. This enhances readability, maintainability, and consistency while preserving the original functionality of the code.

	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.
	• Expected Output #5:
	  o Python file with updated, accurate, and standardized docstrings.
	PROMPT:
	Review the provided Python code and correct any outdated or inaccurate docstrings. Rewrite each docstring to accurately reflect the current function behavior, following Google-style formatting. Ensure they clearly describe the function's purpose, parameters with type hints, return values, and example usage if relevant. Output the full Python file with all docstrings updated and consistent with the code.
	CODE:	

ai > docstring.py > ...

```python
def sum_of_squares(numbers: list[int]) -> int:
    """Calculates the sum of the squares of a list of integers.

    Args:
        numbers (list[int]): A list of integers.

    Returns:
        int: The sum of the squares of the input integers.

    Example:
        >>> sum_of_squares([1, 2, 3])
        14
    """
    return sum(x ** 2 for x in numbers)


def capitalize_words(sentence: str) -> str:
    """Capitalizes the first letter of each word in a sentence.

    Args:
        sentence (str): The input sentence.

    Returns:
        str: The sentence with each word capitalized.

    Example:
        >>> capitalize_words("hello world")
        'Hello World'
    """
    return ' '.join(word.capitalize() for word in sentence.split())


def filter_even(numbers: list[int]) -> list[int]:
    """Filters and returns only the even numbers from a list of integers.

    Args:
        numbers (list[int]): A list of integers.
```

```
    """
    return ' '.join(word.capitalize() for word in sentence.split())


def filter_even(numbers: list[int]) -> list[int]:
    """Filters and returns only the even numbers from a list of integers.

    Args:
        numbers (list[int]): A list of integers.

    Returns:
        list[int]: A list containing only the even integers from the input.

    Example:
        >>> filter_even([1, 2, 3, 4, 5, 6])
        [2, 4, 6]
    """
    return [x for x in numbers if x % 2 == 0]


# Example function calls
print("sum_of_squares([1, 2, 3]):", sum_of_squares([1, 2, 3]))
print("capitalize_words('hello world'):", capitalize_words("hello world"))
print("filter_even([1, 2, 3, 4, 5, 6]):", filter_even([1, 2, 3, 4, 5, 6]))
```

OUTPUT:

```
PS C:\Users\allur\Desktop\B.Tech> python -u "c:
sum_of_squares([1, 2, 3]): 14
capitalize_words('hello world'): Hello World
filter_even([1, 2, 3, 4, 5, 6]): [2, 4, 6]
PS C:\Users\allur\Desktop\B.Tech>
```

OBSERVATION:

All docstrings accurately describe the current behavior of the functions, following Google-style formatting. Each docstring clearly explains the function's purpose, input parameters with type hints, return values, and provides example usage. This ensures the code is readable, maintainable, and consistent with its actual functionality, improving clarity for developers and users.

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:
    - 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.
- Expected Output #6:
    - A comparison table showing the results from both prompts with observations.

PROMPT:

Compare documentation generated by AI for the same Python function using two prompts: a vague prompt ("Add comments to this function") and a detailed prompt ("Add Google-style docstrings with parameters, return types, and examples"). Analyze differences in quality, accuracy, and completeness of the documentation. Provide a comparison table showing results from both prompts along with observations

TABLE:

Aspect	**Vague Prompt:** ***"Add comments to this function"***	**Detailed Prompt:** ***"Add Google-style docstrings with parameters, return types, and examples"***	**Observations**
Format	Single-line or inline comments (e.g., # Adds two numbers)	Structured Google-style docstring with sections: Description, Args, Returns, Example	Detailed prompt enforces a standard, professional format.
Clarity	Very brief, assumes prior knowledge	Clear explanation of function purpose, parameter meaning, return value, and example usage	Detailed prompt is more beginner-friendly.
Accuracy	Correct but superficial (just says "adds two numbers")	Correct, with precise details on parameter types and return type	Both are accurate, but detailed prompt adds type specificity.
Completeness	Only describes purpose	Includes description, parameters, return type, and usage example	Detailed prompt ensures comprehensive documentation.
Usefulness	Limited for quick reference	Useful for developers, API users, and tools like Sphinx/ReadTheDocs	Detailed prompt output is professional and ready for integration.

OBSERVATION:

The vague prompt produces minimal, surface-level comments that do

	not explain parameters, return values, or usage. The detailed prompt yields complete Google-style docstrings, improving clarity, usability, and maintainability. Detailed prompts ensure documentation is accurate, structured, and developer-friendly.	