Assingment:-9.1

-Ai assistant cording

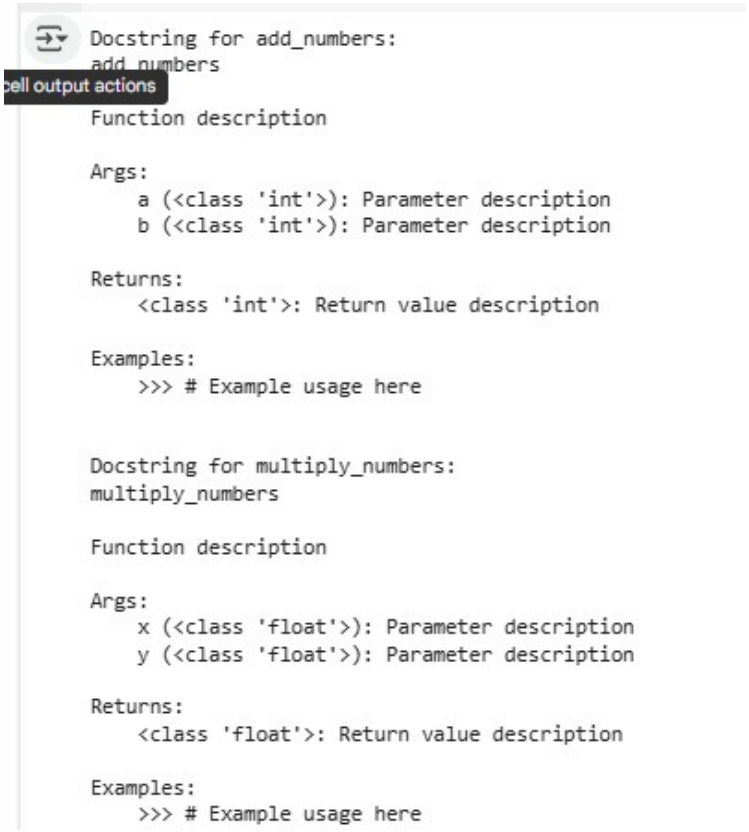
NAME:-Aashutosh Kumar

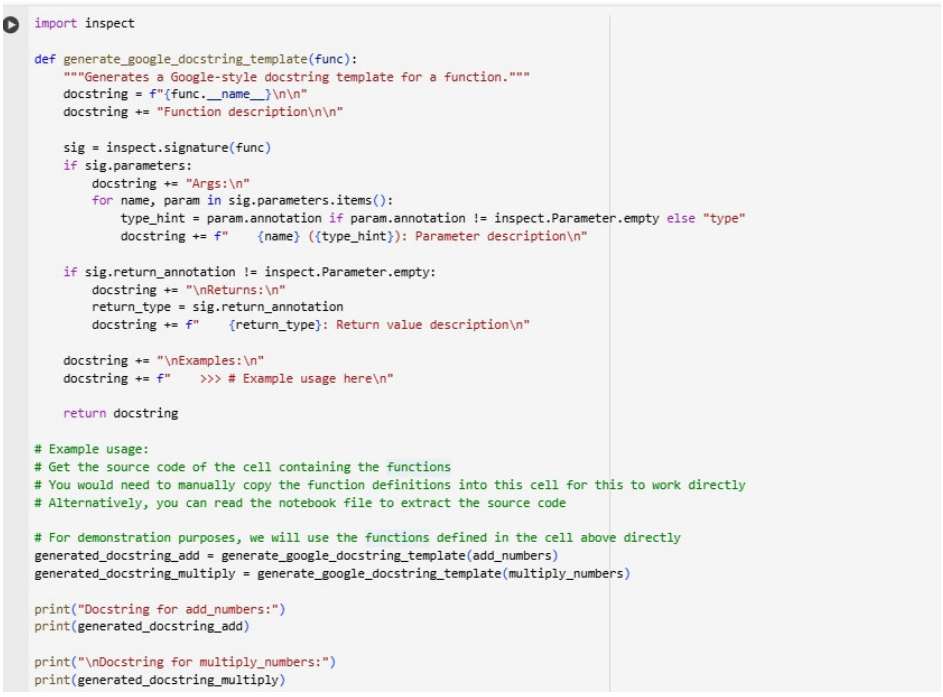
HALLTKT:-2403A51316

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:
  - Prompt AI to generate docstrings without providing any input-output examples.
  - Ensure each docstring includes:
    - Function description
    - Parameters with type hints
    - Return values with type hints
    - Example usage
  - Review the generated docstrings for accuracy and formatting.
- Expected Output #1:
  - A Python script with all functions documented using correctly formatted Google-style docstrings.

ANS:-





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.

ANS:- Python script give by me:

```
def complex_function(data):
  """
  This function processes a list of numbers.
  """
  processed_data = []
  for i in range(len(data)):
    if i % 2 == 0:
      processed_data.append(data[i] * 2)
    else:
      processed_data.append(data[i] + 10)
  return processed_data

# Example usage
my_data = [1, 2, 3, 4, 5]
result = complex_function(my_data)
print(result)
```

Snippet after skip obvious syntax explanations and focus only on tricky or nonintuitive code sections.

```python
def complex_function(data: list[int]) -> list[int]:
    processed_data = []
    for i in range(len(data)):
        # Apply different transformation depending on whether index is even or odd
        if i % 2 == 0:
            # Even index → double the element
            processed_data.append(data[i] * 2)
        else:
            # Odd index → add 10 to the element
            processed_data.append(data[i] + 10)
    return processed_data


# Example usage
my_data = [1, 2, 3, 4, 5]
result = complex_function(my_data)
print(result)
```

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

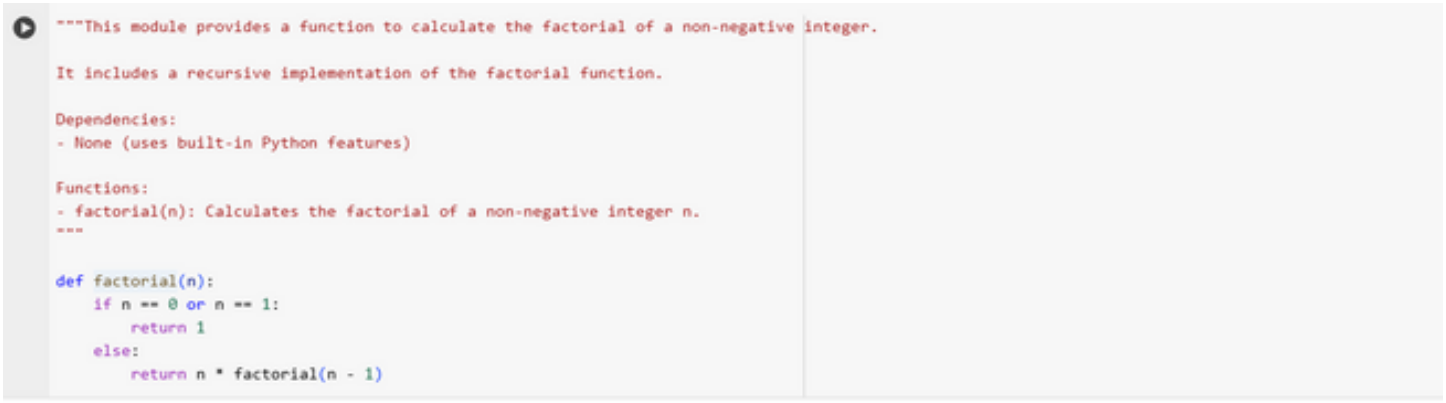
- Task: Use AI to create a module-level docstring summarizing the purpose, dependencies, and main functions/classes of a Python file.
- Instructions:
  - Supply the entire Python file to AI.
  - Instruct AI to write a single multi-line docstring at the top of the file.
  - Ensure the docstring clearly describes functionality and usage without rewriting the entire code.

ANS:-

📌 Example (before prompt)

python Copy Edit

```python
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```



```
"""This module provides a function to calculate the factorial of a non-negative integer.

It includes a recursive implementation of the factorial function.

Dependencies:
- None (uses built-in Python features)

Functions:
- factorial(n): Calculates the factorial of a non-negative integer n.
"""

def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

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.

ANS:-

Step 1:-Taking python code that already has **inline comments**.

```python
def complex_function(data: list[int]) -> list[int]:
    processed_data = []
    for i in range(len(data)):
        # Apply different transformation depending on whether index is even or odd
        if i % 2 == 0:
            # Even index → double the element
            processed_data.append(data[i] * 2)
        else:
            # Odd index → add 10 to the element
            processed_data.append(data[i] + 10)
    return processed_data
```

Step 2:-Move the relevant inline comments into structured Google-style docstrings, and remove those comments from the body.

```python
def complex_function(data: list[int]) -> list[int]:
    """Process a list of numbers with alternating transformations.

    Even-indexed elements are doubled, while odd-indexed elements have 10 added.

    Args:
        data (list[int]): A list of integers to be processed.

    Returns:
        list[int]: A new list where even-indexed numbers are doubled and
        odd-indexed numbers are incremented by 10.

    Examples:
        >>> complex_function([1, 2, 3, 4, 5])
        [2, 12, 6, 14, 10]
    """
    processed_data = []
    for i in range(len(data)):
        if i % 2 == 0:
            processed_data.append(data[i] * 2)
        else:
            processed_data.append(data[i] + 10)
    return processed_data
```

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

- Task: Use AI to identify and correct inaccuracies in existing docstrings.
- Instructions:
  - Provide Python code with outdated or incorrect docstrings.
  - Instruct AI to rewrite each docstring to match the current code behavor.

ANS:-

BEFORE: I provided a outdated and incorrect docstring.

```
def gcd(a: int, b: int) -> int:
    """Return the sum of two numbers.

    Args:
        a (int): First number.
        b (int): Second number.

    Returns:
        int: The sum of a and b.
    """
    while b != 0:
        a, b = b, a % b
    return a
```

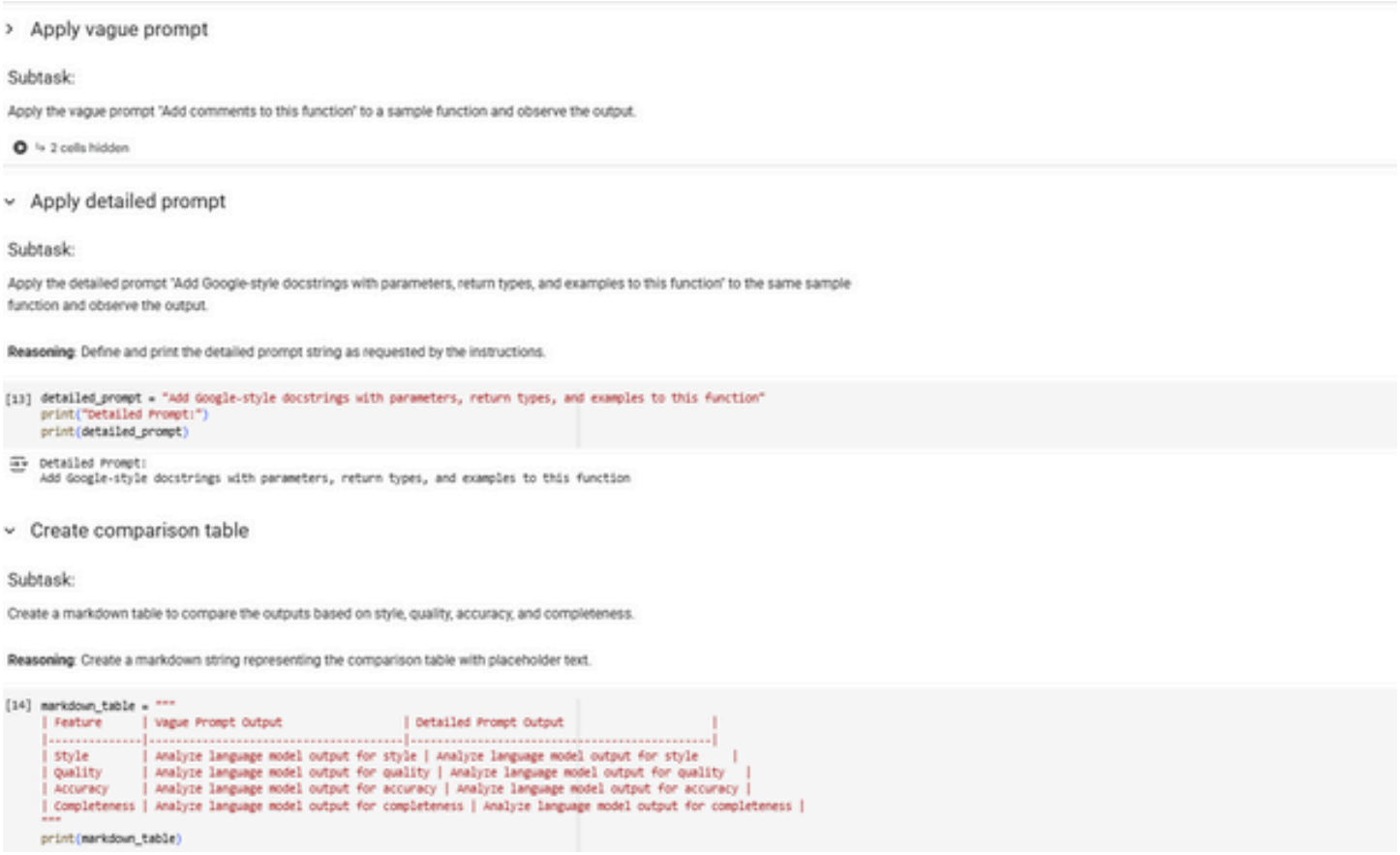
After instructing

```
def gcd(a: int, b: int) -> int:
    """Compute the greatest common divisor (GCD) of two integers.

    Uses the Euclidean algorithm: repeatedly replaces (a, b) with (b, a % b)
    until b becomes zero. The final value of a is the GCD.

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

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

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

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.

ANS:-



Create comparison table

Subtask:

Create a markdown table to compare the outputs based on style, quality, accuracy, and completeness.

↳ 2 cells hidden

Summary:

Data Analysis Key Findings

- A sample Python function `add_numbers` with an existing docstring was defined for testing.
- A "vague prompt" ("Add comments to this function") and a "detailed prompt" ("Add Google-style docstrings with parameters, return types, and examples to this function") were prepared.
- A markdown table was created with columns for "Feature", "Vague Prompt Output", and "Detailed Prompt Output" and rows for "Style", "Quality", "Accuracy", and "Completeness" to facilitate the comparison of the language model outputs.

Insights or Next Steps

- The next step is to actually apply both the vague and detailed prompts to the sample function using a language model and populate the comparison table with the resulting outputs and analysis.
- Analyzing the populated table will provide insights into how prompt specificity affects the style, quality, accuracy, and completeness of generated documentation.

```
[14] markdown_table = """
     | Feature      | Vague Prompt Output                  | Detailed Prompt Output                       |
     |--------------|--------------------------------------|----------------------------------------------|
     | Style        | Analyze language model output for style | Analyze language model output for style     |
     | Quality      | Analyze language model output for quality | Analyze language model output for quality   |
     | Accuracy     | Analyze language model output for accuracy | Analyze language model output for accuracy |
     | Completeness | Analyze language model output for completeness | Analyze language model output for completeness |
     """
     print(markdown_table)
```

```
     | Feature      | Vague Prompt Output                  | Detailed Prompt Output                       |
     |--------------|--------------------------------------|----------------------------------------------|
     | Style        | Analyze language model output for style | Analyze language model output for style     |
     | Quality      | Analyze language model output for quality | Analyze language model output for quality   |
     | Accuracy     | Analyze language model output for accuracy | Analyze language model output for accuracy |
     | Completeness | Analyze language model output for completeness | Analyze language model output for completeness |
```


Summary:

Data Analysis Key Findings

- A sample Python function `add_numbers` with an existing docstring was defined for testing.
- A "vague prompt" ("Add comments to this function") and a "detailed prompt" ("Add Google-style docstrings with parameters, return types, and examples to this function") were prepared.
- A markdown table was created with columns for "Feature", "Vague Prompt Output", and "Detailed Prompt Output" and rows for "Style", "Quality", "Accuracy", and "Completeness" to facilitate the comparison of the language model outputs.

Insights or Next Steps

- The next step is to actually apply both the vague and detailed prompts to the sample function using a language model and populate the comparison table with the resulting outputs and analysis.
- Analyzing the populated table will provide insights into how prompt specificity affects the style, quality, accuracy, and completeness of generated documentation.