

<p>K.Akshay</p> <p>2303A51817</p>			
AssignmentNumber: 9.4 (Present assignment number)/24(Total number of assignments)			
Q.No.	Question	<i>Expected Time to complete</i>	
1	<p>Lab 9 – 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. <p>Lab Outcomes</p> <ul style="list-style-type: none"> • Generate structured code documentation using AI tools • Apply appropriate documentation styles to different code contexts • Improve code readability through selective commenting • Convert informal developer comments into professional documentation • Analyze and refine AI-generated documentation 	Week 5	
	<p>Task 1: Auto-Generating Function Documentation in a Shared Codebase</p> <p>Scenario</p> <p>You have joined a development team where several utility functions are already implemented, but the code lacks proper documentation. New team members are struggling to understand how these functions should be used.</p> <p>Task Description</p> <p>You are given a Python script containing multiple functions without any docstrings.</p> <p>Using an AI-assisted coding tool:</p>		

- Ask the AI to automatically generate **Google-style function docstrings** for each function
- Each docstring should include:
 - A brief description of the function
 - Parameters with data types
 - Return values
 - At least one example usage (if applicable)

Experiment with different prompting styles (zero-shot or context-based) to observe quality differences.

Expected Outcome

- A Python script with well-structured Google-style docstrings
- Docstrings that clearly explain function behavior and usage
- Improved readability and usability of the codebase

Prompt:-

M Generate Google-style docstrings for all functions in the following Python script.
 Each docstring should include: - A short description of the function -
 Parameters with their data types - Return value with its type - At least one
 example usage (if applicable) Do not change the function logic. Only add
 properly formatted docstrings inside each function.

Code:-

```
▶ def add(a, b):
    """
    Adds two numbers together.

    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.

    Example:
        >>> add(2, 3)
        5
    """
    return a + b
def factorial(n):
    """
    Calculates the factorial of a given non-negative integer.

    Args:
        n (int): A non-negative integer.

    Returns:
        int: The factorial of n.

    Example:
        >>> factorial(5)
        120
    """
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)
```

```
▶ def is_even(number):
    """
    Checks whether a number is even.

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

    Returns:
        bool: True if the number is even, otherwise False.

    Example:
        >>> is_even(4)
        True
    """
    return number % 2 == 0

def find_max(numbers):
    """
    Returns the maximum value from a list of numbers.

    Args:
        numbers (list of int or float): A list of numeric values.

    Returns:
        int or float: The largest number in the list.

    Example:
        >>> find_max([3, 7, 2, 9])
        9
    """
    return max(numbers)

print("Addition:", add(10, 5))
print("Factorial:", factorial(5))
print("Is Even:", is_even(8))
print("Maximum:", find_max([1, 4, 9, 2]))
```

Output:-

```
...  Addition: 15
    Factorial: 120
    Is Even: True
    Maximum: 9
```

Explanation:-

This code adds professional Google-style docstrings to functions. It explains what each function does, its parameters, return values, and gives an example usage. This makes the code easier for new developers to understand and use correctly.

Task 2: Enhancing Readability Through AI-Generated Inline Comments

Scenario

A Python program contains complex logic that works correctly but is difficult to understand at first glance. Future maintainers may find it hard to debug or extend this code.

Task Description

You are provided with a Python script containing:

- Loops
- Conditional logic
- Algorithms (such as Fibonacci sequence, sorting, or searching)

Use AI assistance to:

- Automatically insert **inline comments only for complex or non-obvious logic**
- Avoid commenting on trivial or self-explanatory syntax

The goal is to improve clarity without cluttering the code.

Expected Outcome

- A Python script with concise, meaningful inline comments
- Comments that explain *why* the logic exists, not *what* Python syntax does
- Noticeable improvement in code readability

Prompt:-

- M** generate a python code where it has complex logic where it works correctly but it is difficult to understand at first glance. so provide a python script containing loops,conditional logic and algorithms.give inline comments for complex logics

Code:-

```

❶ def mysterious_processor(data_sequence):
    """Processes a sequence of numbers through a series of cryptic transformations.

    The function filters, conditionally transforms, sorts, and then
    performs a cumulative operation based on multiple conditions.

    Args:
        data_sequence (list of int): A list of integer numbers to process.

    Returns:
        int: A single integer result after all transformations and aggregations.

    Example:
        >>> mysterious_processor([1, 2, 3, 6, 7, 8, 9, 10, 12, 15, 20])
        154000
        ...
        intermediate_storage = []

        # First pass: Filter and conditionally transform numbers
        for item in data_sequence:
            # Check if the number is even and greater than 5
            if item % 2 == 0 and item > 5:
                processed_val = 0 # Initialize for scope
                # Apply a different transformation based on divisibility by 3
                if item % 3 == 0: # If divisible by 3, square it
                    processed_val = item ** 2
                else: # Otherwise, cube it
                    processed_val = item ** 3
                intermediate_storage.append(processed_val)

            # Sort the intermediate results in reverse order (descending)
            intermediate_storage.sort(reverse=True)

        final_culmination = 0
        # Second pass: Cumulative aggregation based on another set of conditions
        for index, val in enumerate(intermediate_storage):
            # For even indices, perform addition; for odd indices, perform multiplication
            if index % 2 == 0:

                # Sort the intermediate results in reverse order (descending)
                intermediate_storage.sort(reverse=True)

                final_culmination = 0
                # Second pass: Cumulative aggregation based on another set of conditions
                for index, val in enumerate(intermediate_storage):
                    # For even indices, perform addition; for odd indices, perform multiplication
                    if index % 2 == 0:
                        final_culmination += val # Adding for even indices
                    else:
                        # Apply a special condition for multiplication: if value is greater than 1000
                        if val > 1000:
                            final_culmination *= (val // 100) # Multiply by value divided by 100 (integer division)
                        else:
                            final_culmination += (val * 2) # Add double the value otherwise
                    # Ensure final culmination doesn't exceed a certain threshold, reset if it does
                    if final_culmination > 1_000_000:
                        final_culmination = final_culmination % 100000 + 10000 # Reset with modulo and offset

            return final_culmination

    # Example Usage:
    input_numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
    result = mysterious_processor(input_numbers)
    print(f"The mysterious processing result is: {result}")

    input_numbers_2 = [1, 2, 3, 6, 7, 8, 9, 10, 12, 15, 20]
    result_2 = mysterious_processor(input_numbers_2)
    print(f"The mysterious processing result for a different input is: {result_2}")

```

Output:-

*** The mysterious processing result is: 23800
 The mysterious processing result for a different input is: 10836

Explanation:-

This function takes a list of numbers and first filters out only the even numbers that are greater than five. It then transforms these selected numbers by squaring them if they are divisible by three, or cubing them if they are not. The numbers are sorted in descending order. The function then goes through this sorted list and calculates a cumulative total: it adds the numbers at even indices and either multiplies or adds the

	<p>numbers at odd indices depending on their size. If the total exceeds a certain then resets.. the function returns the final cumulative value.</p>	
	<p>Task 3: Generating Module-Level Documentation for a Python Package</p> <p>Scenario</p> <p>Your team is preparing a Python module to be shared internally (or uploaded to a repository). Anyone opening the file should immediately understand its purpose and structure.</p> <p>Task Description</p> <p>Provide a complete Python module to an AI tool and instruct it to automatically generate a module-level docstring at the top of the file that includes:</p> <ul style="list-style-type: none"> • The purpose of the module • Required libraries or dependencies • A brief description of key functions and classes • A short example of how the module can be used <p>Focus on clarity and professional tone.</p> <p>Expected Outcome</p> <ul style="list-style-type: none"> • A well-written multi-line module-level docstring • Clear overview of what the module does and how to use it • Documentation suitable for real-world projects or repositories <p>Prompt:-</p> <p> Generate a professional module-level docstring for the following Python module. The docstring should be placed at the top of the file and include: - The purpose of the module - Required libraries or dependencies - A brief description of key functions and classes - A short example showing how the module can be used Keep the tone clear, professional, and suitable for a real-world project or repository.</p> <p>Code:-</p>	

► r"""
This module provides advanced numerical data processing capabilities through its primary function, `mysterious_processor`. It performs multi-stage transformations on sequences of integers, including conditional filtering, value transformations based on divisibility rules, sorting, and a custom cumulative aggregation algorithm.

Dependencies:
This module uses only Python built-in data types and functions.
No external libraries are required.

Key Functions:
mysterious_processor(data_sequence: list[int]) -> int
Processes a list of integers through filtering, transformation, sorting, and aggregation steps to return a single integer result.

Example:

```
input_data = [1, 2, 3, 6, 7, 8, 9, 10, 12, 15, 20]
result = mysterious_processor(input_data)
print(result)

another_input = [1, 2, 3, ..., 25]
another_result = mysterious_processor(another_input)
print(another_result)
....
```

Output:-

... 'nThis module provides advanced numerical data processing capabilities through its primary function, `mysterious_processor`. It performs multi-stage transformations on sequences of integers, including conditional filtering, value transformations based on divisibility rules, sorting, and a custom cumulative aggregation algorithm.'nDependencies:'n This module uses only Python built-in data types and functions.'n No external libraries are required.'nKey Functions:'n **mysterious_processor(data_sequence: list[int]) -> int**'n Processes a list of integers through filtering, transformation, sorting, and aggregation steps to return a single integer result.'nExample:'n input_data = [1, 2, 3, 6, 7, 8, 9, 10, 12, 15, 20]'n result = mysterious_processor(input_data)'n print(result)'n another_input = [1, 2, 3, ..., 25]'n another_result = mysterious_processor(another_input)'n print(another_result)'n' n

Explanation:-

This code block is a documentation string for an entire python module. It clearly describes what the module does, specifies that it requires no external libraries, briefly introduces its main function, and provides examples of how to use that function, making it easy for anyone to understand the module's purpose and functionality at a glance.

Task 4: Converting Developer Comments into Structured Docstrings

Scenario

In a legacy project, developers have written long explanatory comments inside functions instead of proper docstrings. The team now wants to standardize documentation.

Task Description

You are given a Python script where functions contain detailed inline comments explaining their logic.

Use AI to:

- Automatically convert these comments into structured **Google-style or NumPy-style docstrings**
- Preserve the original meaning and intent of the comments
- Remove redundant inline comments after conversion

Expected Outcome

- Functions with clean, standardized docstrings
- Reduced clutter inside function bodies
- Improved consistency across the codebase

Prompt:-

M Convert the inline comments in the following Python functions into proper Google-style docstrings. Keep the meaning the same, remove unnecessary inline comments after conversion, and do not change the function logic.

Code:-

```
def mysterious_processor(data_sequence):
    """Processes a sequence of numbers through a series of cryptic transformations.

    The function filters, conditionally transforms, sorts, and then
    performs a cumulative operation based on multiple conditions.

    Args:
        data_sequence (list of int): A list of integer numbers to process.

    Returns:
        int: A single integer result after all transformations and aggregations.

    Process:
        1. **First Pass (Filtering and Transformation)**:
            - Iterates through `data_sequence`.
            - For each 'item', if it's even and greater than 5:
                - If 'item' is divisible by 3, it's squared ('item ** 2').
                - Otherwise (if not divisible by 3), it's cubed ('item ** 3').
            - These processed values are collected into 'intermediate_storage'.
        2. **Sorting**: The 'intermediate_storage' list is sorted in reverse (descending) order.
        3. **Second Pass (Cumulative Aggregation)**:
            - Initializes 'final_culmination' to 0.
            - Iterates through the sorted 'intermediate_storage' with both index and value:
                - If the 'index' is even:
                    - If 'val' is greater than 1000, 'final_culmination' is multiplied by 'val' integer-divided by 100 ('val // 100').
                    - Otherwise, 'final_culmination' is incremented by 'val' multiplied by 2 ('val * 2').
            - **Threshold Reset**: If 'final_culmination' ever exceeds 1,000,000, it's reset using a modulo operation ('% 100000') a
    Example:
        >>> mysterious_processor([1, 2, 3, 6, 7, 8, 9, 10, 12, 15, 20])
        154000
    """
    intermediate_storage = []

    for item in data_sequence:
        if item % 2 == 0 and item > 5:
            processed_val = 0
```

```

for item in data_sequence:
    if item % 2 == 0 and item > 5:
        processed_val = 0
        if item % 3 == 0:
            processed_val = item ** 2
        else:
            processed_val = item ** 3
        intermediate_storage.append(processed_val)

intermediate_storage.sort(reverse=True)

final_culmination = 0
for index, val in enumerate(intermediate_storage):
    if index % 2 == 0:
        final_culmination += val
    else:
        if val > 1000:
            final_culmination *= (val // 100)
        else:
            final_culmination += (val * 2)
    if final_culmination > 1_000_000:
        final_culmination = final_culmination % 100000 + 10000

return final_culmination

# Example Usage:
input_numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25]
result = mysterious_processor(input_numbers)
print(f"The mysterious processing result is: {result}")

input_numbers_2 = [1, 2, 3, 6, 7, 8, 9, 10, 12, 15, 20]
result_2 = mysterious_processor(input_numbers_2)
print(f"The mysterious processing result for a different input is: {result_2}")

```

Output:-

```

*** The mysterious processing result is: 23800
The mysterious processing result for a different input is: 10836

```

Explanation:-

I converted the detailed inline comments within the `mysterious_processor` function into a comprehensive, Google-style docstring. This involved moving the explanations of the function's complex steps into a new 'Process:' section within the docstring and then removing the now redundant inline comments, all while ensuring the function's original behavior remained perfectly intact.

Task 5: Building a Mini Automatic Documentation Generator

Scenario

Your team wants a simple internal tool that helps developers start documenting new Python files quickly, without writing documentation from scratch.

Task Description

Design a small Python utility that:

- Reads a given `.py` file
- Automatically detects:
 - Functions
 - Classes
- Inserts **placeholder Google-style docstrings** for each detected function or class

AI tools may be used to assist in generating or refining this utility.
Note: The goal is **documentation scaffolding**, not perfect documentation.

Expected Outcome

- A working Python script that processes another .py file
- Automatically inserted placeholder docstrings
- Clear demonstration of how AI can assist in documentation automation

Prompt:-

M Help me build a simple Python script that reads another .py file, detects functions and classes, and inserts placeholder Google-style docstrings inside them. The script should: - Detect "def" and "class" definitions - Insert a basic Google-style placeholder docstring - Not modify existing logic - Save the updated file Keep the solution simple and readable.

Code:-

```
▶ # Define sample code for input.py
sample_code_for_input = '''
import math

def calculate_area(radius):
    return math.pi * radius**2

class Circle:
    def __init__(self, radius):
        self.radius = radius

    def get_circumference(self):
        return 2 * math.pi * self.radius

    def is_valid(self):
        return self.radius > 0
...

# Write the sample code to 'input.py'
with open('input.py', 'w') as f:
    f.write(sample_code_for_input)

print("Created 'input.py' with sample code.")

# Run the docstring insertion function
add_placeholder_docstrings("input.py", "output.py")

print("\n--- Content of input.py (Original) ---")
with open('input.py', 'r') as f:
    print(f.read())

print("\n--- Content of output.py (With Docstrings) ---")
with open('output.py', 'r') as f:
    print(f.read())
```

Output:-

```
... Created 'input.py' with sample code.
Documentation scaffolding completed successfully!

--- Content of input.py (Original) ---

import math

def calculate_area(radius):
    return math.pi * radius**2

class Circle:
    def __init__(self, radius):
        self.radius = radius

    def get_circumference(self):
        return 2 * math.pi * self.radius

    def is_valid(self):
        return self.radius > 0

--- Content of output.py (With Docstrings) ---

import math

def calculate_area(radius):
    """
    Short description of calculate_area.

    Args:
        # Add parameter descriptions here

    Returns:
        # Add return description here
    """
    return math.pi * radius**2
```

```

class Circle:
    """
    Short description of Circle class.
    """
    def __init__(self, radius):
        """
        Short description of __init__.

        Args:
            # Add parameter descriptions here

        Returns:
            # Add return description here
        """
        self.radius = radius

    def get_circumference(self):
        """
        Short description of get_circumference.

        Args:
            # Add parameter descriptions here

        Returns:
            # Add return description here
        """
        return 2 * math.pi * self.radius

    def is_valid(self):
        """
        Short description of is_valid.

        Args:
            # Add parameter descriptions here

        Returns:
            # Add return description here
        """
        return self.radius > 0

```

Explanation:-

I wrote a simple Python script containing a few functions and a class in a Python file called input.py. Subsequently, I used the add placeholder docstrings script on the input.py file. The script scanned the input.py file and detected all the functions and the class defined in the script. The script subsequently added basic Google Style placeholder docstrings to the script. After that, I checked the original input.py and the newly created output.py. From the code, you could see obviously that the docstrings were added correctly into the new file, and the code logic inside had not been changed at all.

Note: Report should be submitted a word document for all tasks in a single document with prompts, comments & code explanation, and output and if required, screenshots

--	--	--