AI-ASSISTED CODING

ASSIGNMENT-9.1

Documentation Generation: Automatic Documentation and  
Code Comments

NAME: VISHWAS

HALL-TICKET: 2403A510E4

BATCH NO: 05

**TASK-1:** 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.

**Prompt:** Give two simple python functions and generate google type doc string as in line comments.

**CODE GENERATED:**

def add\_numbers(a: int, b: int) -> int:

    """

    Adds two integers and returns the result.

    Args:

        a (int): The first integer.

        b (int): The second integer.

    Returns:

        int: The sum of a and b.

    Example:

        >>> add\_numbers(5, 7)

        12

    """

    return a + b

def is\_palindrome(word: str) -> bool:

    """

    Checks whether a given word is a palindrome.

    Args:

        word (str): The word to check.

    Returns:

        bool: True if the word is a palindrome, False otherwise.

    Example:

        >>> is\_palindrome("madam")

        True

    """

    return word == word[::-1]

# Example usage and outputs

result1 = add\_numbers(5, 7)

print("add\_numbers(5, 7) =", result1)  # Output: 12

result2 = is\_palindrome("madam")

print("is\_palindrome('madam') =", result2)  # Output: True

result3 = is\_palindrome("hello")

print("is\_palindrome('hello') =", result3)  # Output: False

**OUTPUT:**

**A black background with white text

AI-generated content may be incorrect.**

**OBSERVATION :**

The AI added **Google-style docstrings** to the given functions. Each function now has a clear description, parameters with type hints, return type, and an example usage. This makes the code easier to understand and maintain.

TASK -2: Use AI to add meaningful inline comments to a Python  
program explaining only complex logic parts.  
• Instructions:  
o Provide a Python script without comments to the AI.  
o Instruct AI to skip obvious syntax explanations and focus  
only on tricky or non-intuitive code sections.  
o Verify that comments improve code readability and  
maintainability.

PROMPT:

Add meaningful inline comments to the following Python script, explaining only the complex or non-intuitive parts of the code. Skip obvious syntax explanations. Provide the updated script with comments.

CODE GENERATED:

def find\_missing\_number(nums):

    """

    Finds the missing number in a list containing numbers from 0 to n with one missing.

    Args:

        nums (list of int): List of n numbers ranging from 0 to n with one missing.

    Returns:

        int: The missing number.

    """

    n = len(nums)

    total\_sum = n \* (n + 1) // 2  # Sum of all numbers from 0 to n using formula

    # The tricky part: sum(nums) gives the sum of present numbers, so subtracting

    # it from total\_sum directly gives the missing number

    return total\_sum - sum(nums)

# Example usage

numbers = [0, 1, 2, 4, 5]

missing = find\_missing\_number(numbers)

print("Missing number:", missing)  # Output: 3

OUTPUT:  


OBSERVATION:

The AI added helpful inline comments only to the tricky parts of the code. This makes it easier to understand the non-obvious logic without cluttering the code with explanations for basic syntax.

**TASK – 3:**

Use AI to create a module-level docstring summarizing the  
purpose, dependencies, and main functions/classes of a Python  
file.

• Instructions:  
o Supply the entire Python file to AI.  
o Instruct AI to write a single multi-line docstring at the top  
of the file.  
o Ensure the docstring clearly describes functionality and  
usage without rewriting the entire code.

PROMPT:

Write a module-level docstring for the following Python file. The docstring should summarize the file’s purpose, its dependencies, and list the main functions or classes. Include a brief usage example. Do not modify or rewrite the existing code.

CODE GENERATED BEFORE DOCSTRING:

def is\_prime(n):

    if n < 2:

        return False

    for i in range(2, int(n\*\*0.5) + 1):

        if n % i == 0:

            return False

    return True

AFTER ADDING DOCSTRING:

"""

Module: prime\_utils.py

Purpose:

    This module provides a utility function to check whether a given number is prime.

Dependencies:

    - None (uses only standard Python features)

Main Functions:

    - is\_prime(n: int) -> bool:

        Determines whether a given number is prime.

Usage:

    Import this module and call the function with appropriate arguments.

    Example:

        >>> is\_prime(7)

        True

        >>> is\_prime(10)

        False

"""

def is\_prime(n: int) -> bool:

    """Check if a number is prime.

    A prime number is a number greater than 1 that has no divisors other than 1 and itself.

    Args:

        n (int): The number to check.

    Returns:

        bool: True if n is prime, False otherwise.

    Example:

        >>> is\_prime(7)

        True

        >>> is\_prime(10)

        False

    """

    if n < 2:

        return False

    for i in range(2, int(n\*\*0.5) + 1):

        if n % i == 0:

            return False

    return True

# Example usage with dynamic input

if \_name\_ == "\_main\_":

    num = int(input("Enter a number to check if it is prime: "))

    if is\_prime(num):

        print(f"{num} is a prime number")

    else:

        print(f"{num} is not a prime number")

OBSERVATION:

The AI created a clear module-level docstring that describes what the Python file does, its dependencies, the main functions/classes, and gives a short usage example. This helps anyone reading the file understand its purpose quickly without diving into the code.

TASK -4:

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

PROMPT:

Convert the inline comments in this Python code into structured Google-style docstrings for each function. Keep the meaning of the comments intact and include descriptions for parameters and return values.

ORIGINAL CODE WITH IN-LINE COMMENTS:

def factorial(n):

    # Factorial of 0 is 1

    if n == 0:

        return 1

    # Initialize result to 1

    result = 1

    # Multiply result by each number from 1 to n

    for i in range(1, n + 1):

        result \*= i

    # Return the factorial of n

    return result

CODE TRANSFORMED TO DOCSTRING:

def factorial(n: int) -> int:

    """Compute the factorial of a non-negative integer.

    The factorial of a number n (denoted as n!) is the product of all positive

    integers less than or equal to n. By definition, factorial of 0 is 1.

    Args:

        n (int): A non-negative integer whose factorial is to be computed.

    Returns:

        int: Factorial of n.

    Example:

        >>> factorial(5)

        120

        >>> factorial(0)

        1

    """

    if n == 0:

        return 1

    result = 1

    for i in range(1, n + 1):

        result \*= i

    return result

OBSERVATION:

The AI successfully transformed the inline comments into clear Google-style docstrings. Each function now has a proper description with arguments and return details, making the code more readable and professional.

TASK -5:

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.

PROMPT:

Review the following Python code and update the existing docstrings so they accurately describe the current function behavior. Use Google-style formatting for the corrected docstrings

CODE INPUT WITH INACCURATE DOCSTRING :

def square\_number(x: int) -> int:

    """

    Multiplies two numbers and returns the result.   # ❌ Incorrect

    Args:

        x (int): The first number.

    Returns:

        int: The product of two numbers.             # ❌ Wrong description

    """

    return x \* x

def greet(name: str) -> str:

    """

    Returns the length of the given string.          # ❌ Incorrect

    Args:

        name (str): Input string.

    Returns:

        int: The length of the string.               # ❌ Wrong return type

    """

    return f"Hello, {name}!"

AI-CORRECTED DOCSTRING:

def square\_number(x: int) -> int:

    """

    Returns the square of a number.

    Args:

        x (int): The number to be squared.

    Returns:

        int: The square of the input number.

    Example:

        >>> square\_number(4)

        16

    """

    return x \* x

def greet(name: str) -> str:

    """

    Generates a greeting message for the given name.

    Args:

        name (str): The name of the person to greet.

    Returns:

        str: A greeting message including the person's name.

    Example:

        >>> greet("Alice")

        'Hello, Alice!'

    """

    return f"Hello, {name}!"

OBSERVATION:

The AI corrected the outdated docstrings to match what the functions actually do. Now, the docstrings are accurate, consistent with the code, and follow Google-style formatting, which improves clarity and reliability.

**TASK-6:**

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

VAGUE PROMPT:

def factorial(n: int) -> int:

    if n == 0 or n == 1:

        return 1

    return n \* factorial(n - 1)

"Add comments to this function"

Detailed Prompt:

Add Google-style docstrings with parameters, return types, and examples.

CODE WITH PRAGUE PROMPT:

def factorial(n: int) -> int:

    # Check if n is 0 or 1

    if n == 0 or n == 1:

        return 1

    # Recursive call to calculate factorial

    return n \* factorial(n - 1)

CODE WITH DETAILED PROMPT:

def factorial(n: int) -> int:

    """

    Calculates the factorial of a non-negative integer using recursion.

    Args:

        n (int): A non-negative integer for which to calculate the factorial.

    Returns:

        int: The factorial of the given integer.

    Example:

        >>> factorial(5)

        120

    """

    if n == 0 or n == 1:

        return 1

    return n \* factorial(n - 1)

OBSERVATION:

• The vague prompt produced only inline comments that explain what the code does step by step.

• The detailed prompt produced rich, structured documentation that is more useful for developers and aligns with industry standards.