**Name :- P.Nishanth**

**HTNO :-** 2403A52066

**ASSIGNMENT – 9.2**

# 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.

# CODE

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 the two integers.

Example:

>>> add\_numbers(3, 5)

8 """ return a + b def greet\_user(name: str) -> str:

"""Generates a greeting message for a user.

Args: name (str): The name of the user.

Returns:

str: A greeting message that includes the user's name.

Example:

>>> greet\_user("Alice")

'Hello, Alice!'

"""

return f"Hello, {name}!" def factorial(n: int) -> int:

"""Calculates the factorial of a given non-negative integer.

Uses a recursive approach.

Args:

n (int): A non-negative integer.

Returns:

int: The factorial of the input number.

Raises:

ValueError: If n is negative.

Example:

>>> factorial(5)

120 """ if n < 0:

raise ValueError("n must be a non-negative integer") if n == 0 or n == 1:

return 1 return n \* factorial(n - 1) print(add\_numbers(3, 5)) print(greet\_user("Alice")) print(factorial(5))

# OUTPUT

8

Hello, Alice!

120

# OBSERVATION

After adding **Google-style docstrings**, every function in the script is now **welldocumented**.



The docstrings clearly describe:

* **What the function does** (description).
* **What inputs it expects** (parameters with type hints).
* **What it returns** (return type).
* **How to use it** (example usage).
* **Possible errors** (Raises section in factorial).

 Running the functions produced correct outputs:

* add\_numbers(3, 5) → 8
* greet\_user("Alice") → "Hello, Alice!"
* factorial(5) → 120

The **documentation makes the code easier to understand and maintain**, especially for new users or collaborators.



Example usage inside the docstrings also serves as **inline testing** for correctness.

# 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. o Verify that comments improve code readability and maintainability.

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

# CODE

def is\_prime(n: int) -> bool: """Check if a number is prime."""

if n <= 1:

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

# Only check divisors up to sqrt(n), # because any larger factor would already

# have a corresponding smaller factor.

if n % i == 0: return False return True def fibonacci(n: int) -> list[int]:

"""Generate the first n Fibonacci numbers.""" sequence = [0, 1] while len(sequence) < n:

# Next number is sum of the last two numbers in the sequence. sequence.append(sequence[-1] + sequence[-2]) return sequence[:n] def binary\_search(arr: list[int], target: int) -> int: """Perform binary search on a sorted list.""" left, right = 0, len(arr) - 1 while left <= right:

mid = (left + right) // 2

# Compare middle element with the target if arr[mid] == target:

return mid elif arr[mid] < target: # Target is in the right half left = mid + 1 else:

# Target is in the left half right = mid - 1 return -1 def factorial(n: int) -> int: """Calculate factorial recursively.""" if n == 0 or n == 1:

return 1

# Recursive call reduces the problem size by 1 each step.

return n \* factorial(n - 1) print(is\_prime(7)) # Prime number print(is\_prime(12)) # Not prime print(fibonacci(7)) # First 7 Fibonacci numbers print(binary\_search([1, 3, 5, 7, 9], 7)) # Find element print(binary\_search([1, 3, 5, 7, 9], 4)) # Element not found print(factorial(5)) # Factorial of 5

# OUTPUT

True

False

[0, 1, 1, 2, 3, 5, 8]

3

-1

120

# OBSERVATION

 **nline comments** were added **only in complex logic sections**, such as:

* Explaining why prime checking stops at sqrt(n).
* Clarifying how Fibonacci numbers are generated.
* Showing how binary search adjusts search boundaries.
* Explaining recursion in factorial.

**Obvious syntax** (like return n + 1) was not commented, keeping the code clean.



Comments **improved readability** by giving context to tricky steps without cluttering.

Code runs correctly, producing expected outputs for all test cases.

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

# CODE

def is\_prime(n: int) -> bool: """Check if a number is prime."""

if n <= 1:

return False for i in range(2, int(n\*\*0.5) + 1): if n % i == 0: return False

return True def fibonacci(n: int) -> list[int]:

"""Generate the first n Fibonacci numbers.""" sequence = [0, 1] while len(sequence) < n: sequence.append(sequence[-1] + sequence[-2]) return sequence[:n] def binary\_search(arr: list[int], target: int) -> int: """Perform binary search on a sorted list.""" left, right = 0, len(arr) - 1 while left <= right:

mid = (left + right) // 2 if arr[mid] == target:

return mid elif arr[mid] < target:

left = mid + 1 else: right = mid - 1 return -1 def factorial(n: int) -> int: """Calculate factorial recursively.""" if n == 0 or n == 1:

return 1 return n \* factorial(n - 1) # Sample test calls print(is\_prime(11)) print(fibonacci(6)) print(binary\_search([2, 4, 6, 8, 10], 8)) print(factorial(5))

# OUTPUT

True

[0, 1, 1, 2, 3, 5]

3

120

# OBSERVATION

The **module-level docstring** at the top summarizes the **purpose, functions, dependencies, and usage** of the file.



It clearly lists available functions and their roles, making the module **self-explanatory** for future users.

No external dependencies are required, keeping the module lightweight.

Running the test calls verified correctness:

* is\_prime(11) → True
* fibonacci(6) → [0, 1, 1, 2, 3, 5]
* binary\_search([2,4,6,8,10], 8) → 3
* factorial(5) → 120

 The docstring improves **maintainability and usability**, serving as quick documentation for the entire file.

# 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:

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.

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

# CODE

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

if n <= 1: return False for i in range(2, int(n\*\*0.5) + 1): if n % i == 0: return False return True def fibonacci(n: int) -> list[int]:

sequence = [0, 1] while len(sequence) < n: sequence.append(sequence[-1] + sequence[-2]) return sequence[:n] def binary\_search(arr: list[int], target: int) -> int:

left, right = 0, len(arr) - 1 while left <= right:

mid = (left + right) // 2 if arr[mid] == target:

return mid elif arr[mid] < target:

left = mid + 1 else: right = mid - 1 return -1 def factorial(n: int) -> int: if n == 0 or n == 1:

return 1 return n \* factorial(n - 1)

# Sample test calls

print(is\_prime(7)) print(is\_prime(12)) print(fibonacci(7)) print(binary\_search([1, 3, 5, 7, 9], 7)) print(binary\_search([1, 3, 5, 7, 9], 4)) print(factorial(5))

# OUTPUT

True

False

[0, 1, 1, 2, 3, 5, 8]

3

-1

120

# OBSERVATION

nline comments were **converted into structured Google-style docstrings**, preserving their



meaning.

Each function now has:

* **Description** of what it does.
* **Args** with type hints.
* **Returns** with type hints.
* **Example usage** for clarity.

 The code executed successfully and produced expected results:

* is\_prime(7) → True
* is\_prime(12) → False
* fibonacci(7) → [0, 1, 1, 2, 3, 5, 8]
* binary\_search([...], 7) → 3
* binary\_search([...], 4) → -1
* factorial(5) → 120

 The new structure makes documentation **standardized and professional**, improving code **readability and reusability**.

# 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**.**

# CODE

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

if n <= 1:

return False for i in range(2, int(n\*\*0.5) + 1): if n % i == 0: return False return True def fibonacci(n: int) -> list[int]:

sequence = [0, 1] while len(sequence) < n:

sequence.append(sequence[-1] + sequence[-2]) return sequence[:n] def binary\_search(arr: list[int], target: int) -> int:

left, right = 0, len(arr) - 1 while left <= right:

mid = (left + right) // 2 if arr[mid] == target:

return mid elif arr[mid] < target: left = mid + 1 else: right = mid - 1 return -1 def factorial(n: int) -> int: if n < 0:

raise ValueError("n must be a non-negative integer") if n == 0 or n == 1:

return 1 return n \* factorial(n - 1) # Sample test calls print(is\_prime(11)) print(is\_prime(12)) print(fibonacci(6)) print(binary\_search([1, 3, 5, 7, 9], 7)) print(binary\_search([1, 3, 5, 7, 9], 4)) print(factorial(5))

# OUTPUT

True

False

[0, 1, 1, 2, 3, 5]

3

-1

120

# OBSERVATION

The original docstrings were reviewed and corrected to **accurately describe code behavior**.



Now every docstring follows **Google-style formatting** with correct:

* Function description
* Args with type hints
* Returns with type hints
* Example usage
* Error handling (Raises in factorial)

 Test calls confirm correctness:

* Prime check → True / False
* Fibonacci sequence generated correctly
* Binary search returned correct index or -1
* Factorial returned correct result 120

 The code is now **fully documented, standardized, and consistent with functionality**.

# 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: o A comparison table showing the results from both prompts with observations.

# CODE

# Function without documentation def square(n): return n \* n

# Output from vague prompt: "Add comments to this function" def square\_vague(n):

# multiply the number by itself return n \* n

# Output from detailed prompt: "Add Google-style docstrings with parameters, return types, and examples" def square\_detailed(n: int) -> int:

return n \* n # Sample test calls print(square(5)) print(square\_vague(6)) print(square\_detailed(7))

# OUTPUT

25

36

49

# OBSERVATION

1. The **vague prompt** added only a single inline comment, which is technically correct but lacks detail.
2. The **detailed prompt** generated a complete **Google-style docstring**, including description, parameters, return type, and example usage.
3. Running test calls confirmed correct outputs for all function versions.
4. This shows that **specific prompts lead to higher-quality documentation**, while vague prompts produce minimal results.