Lab Assignment-9.4

Hall.No:2303A51873

Name:R.Vineesha

Batch:14

Task 1: Auto-Generating Function Documentation in a Shared Codebase

Scenario

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.

Task Description

You are given a Python script containing multiple functions without any docstrings.

Using an AI-assisted coding tool:

• Ask the AI to automatically generate Google-style function docstrings for each function

• Each docstring should include:

o A brief description of the function

o Parameters with data types

o Return values

o 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

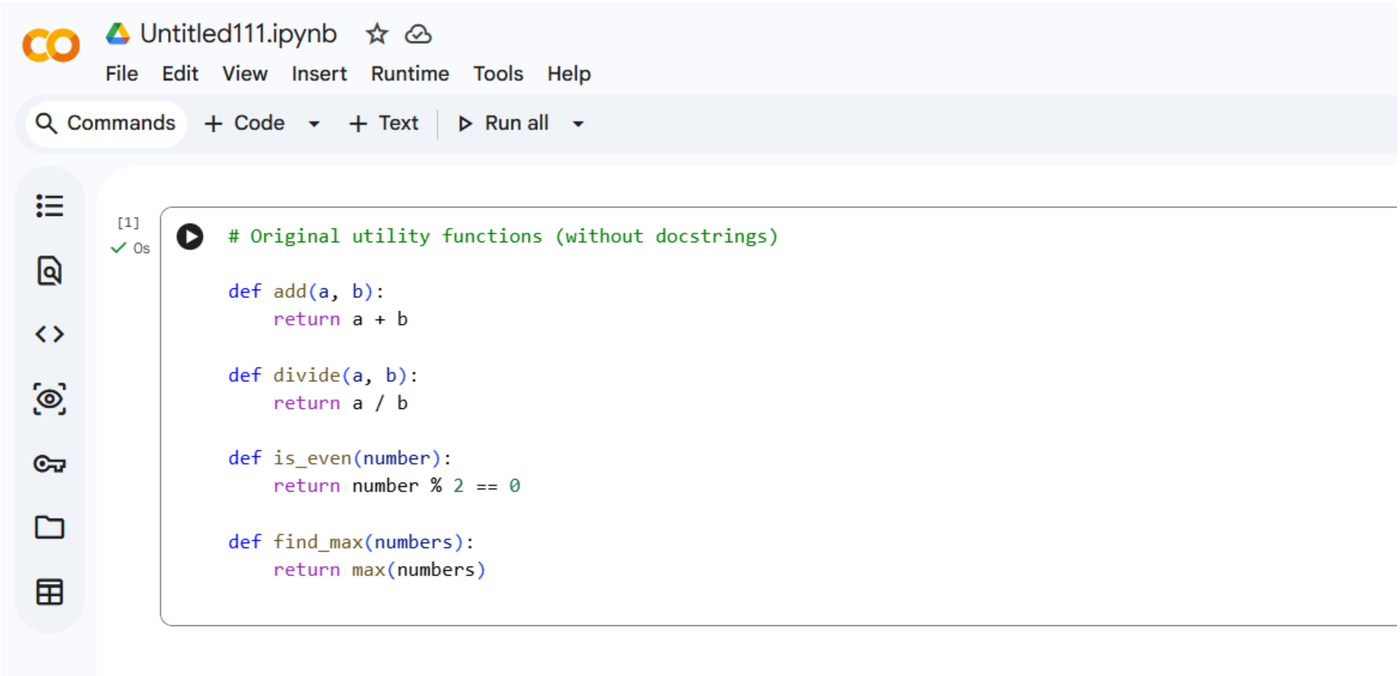
• Docstrings that clearly explain function behavior and usage

• Improved readability and usability of the codebase

Original Python Code (Without Docstrings)

```
# Original utility functions (without docstrings)

def add(a, b):

   return a + b

def divide(a, b):

   return a / b

def is_even(number):
```

```
    return number % 2 == 0
def find_max(numbers):
    return max(numbers)
```



Docstrings (Zero-Shot Prompt)

Prompt:

Generate Google-style docstrings for the following Python functions.

Include:

- Brief description

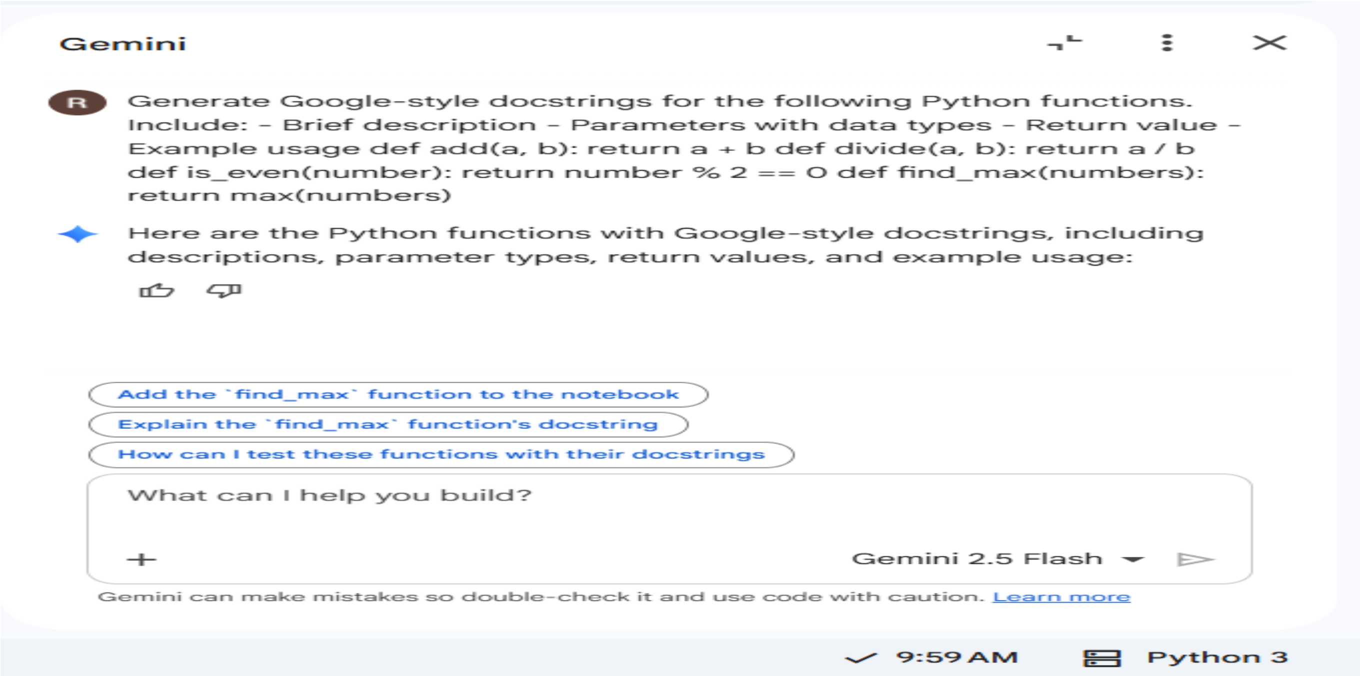
- Parameters with data types

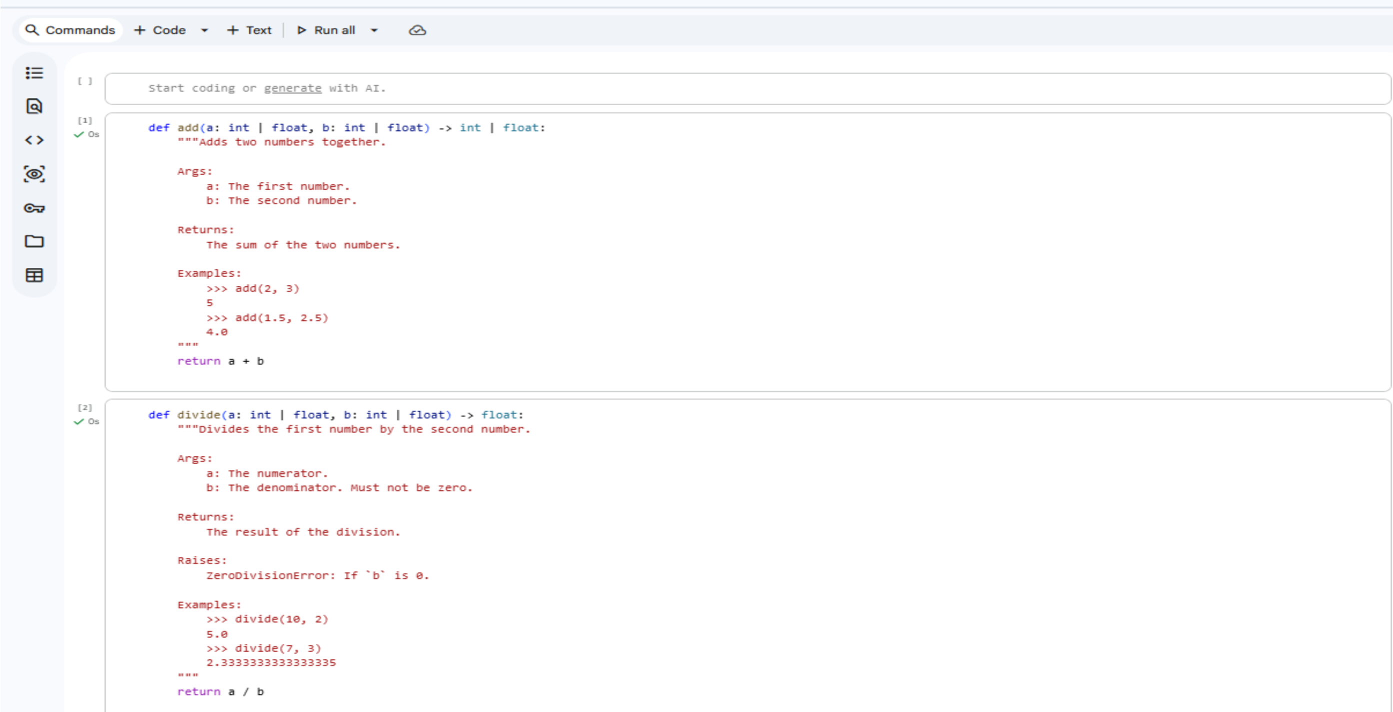
- Return value

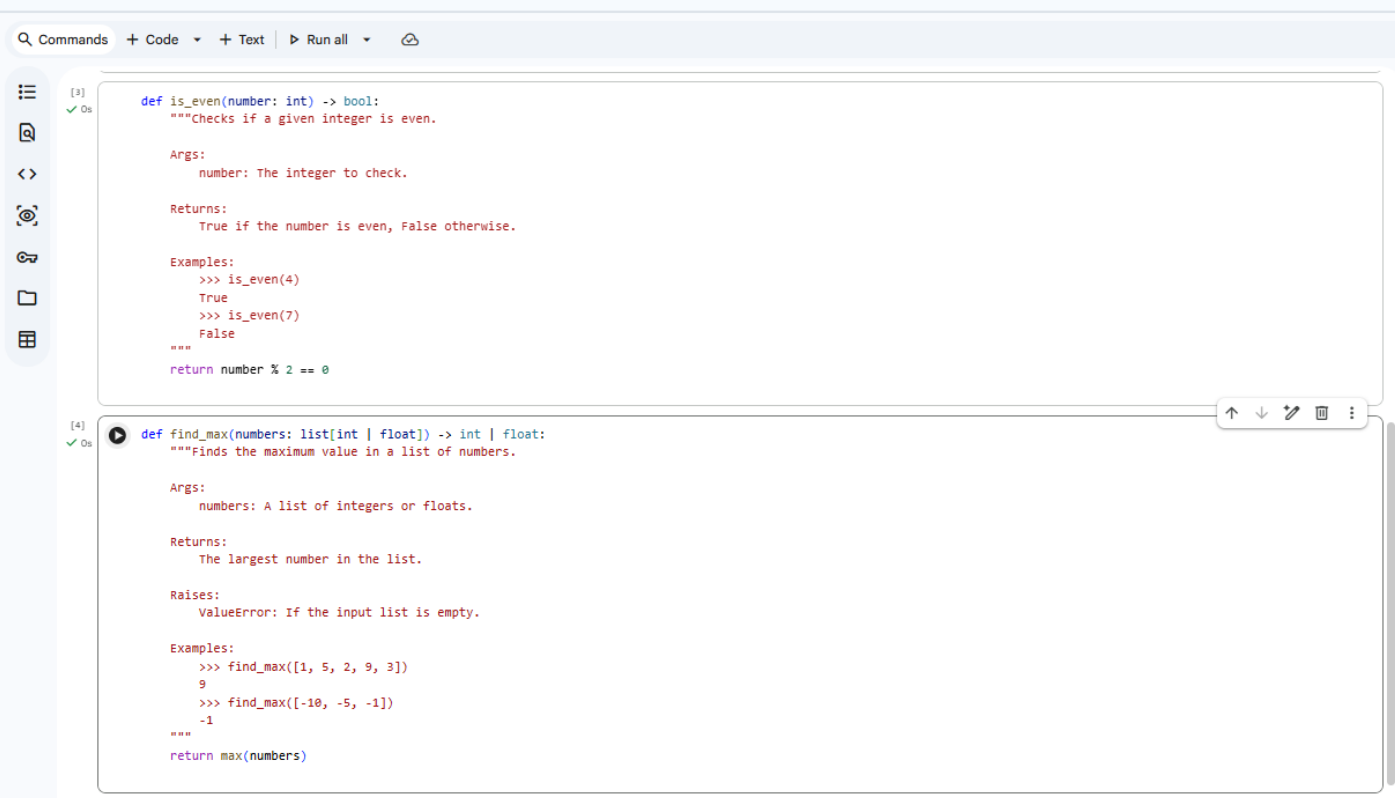
- Example usage

```
def add(a, b):
    return a + b
def divide(a, b):
    return a / b
def is_even(number):
    return number % 2 == 0
def find_max(numbers):
```

return max(numbers)







```python
def is_even(number: int) -> bool:
    """Checks if a given integer is even.

    Args:
        number: The integer to check.

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

    Examples:
        >>> is_even(4)
        True
        >>> is_even(7)
        False
    """
    return number % 2 == 0
```

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

    Args:
        numbers: A list of integers or floats.

    Returns:
        The largest number in the list.

    Raises:
        ValueError: If the input list is empty.

    Examples:
        >>> find_max([1, 5, 2, 9, 3])
        9
        >>> find_max([-10, -5, -1])
        -1
    """
    return max(numbers)
```

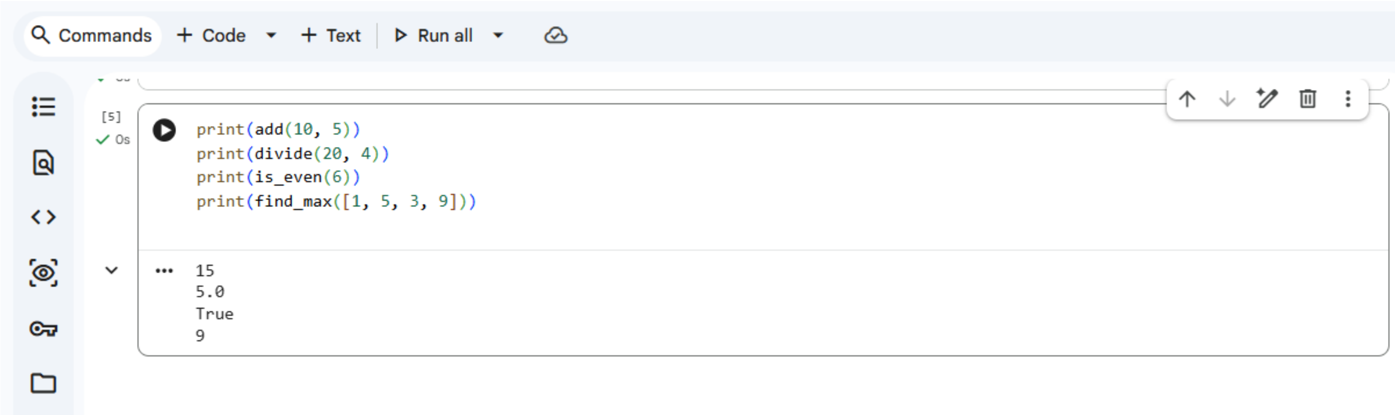
Test the Functions

print(add(10, 5))

print(divide(20, 4))

print(is_even(6))

print(find_max([1, 5, 3, 9]))



```python
print(add(10, 5))
print(divide(20, 4))
print(is_even(6))
print(find_max([1, 5, 3, 9]))
```

```
15
5.0
True
9
```

Context-Based Prompt (For Comparison)

You are a senior Python developer working in a shared enterprise codebase.zWrite professional, beginner-friendly Google-style docstrings for the following functions. Make the documentation clear and production-ready.

```python
def add(a: int | float, b: int | float) -> int | float:
    """Calculates the sum of two numerical values.

    This function takes two numbers (integers or floats) and returns their sum.
    It's a fundamental arithmetic operation.

    Args:
        a: The first number, which can be an integer or a float.
        b: The second number, which can also be an integer or a float.

    Returns:
        The sum of `a` and `b`, maintaining the appropriate numerical type
        (e.g., if one input is float, the result will be float).

    Examples:
        >>> add(2, 3)
        5
        >>> add(1.5, 2.5)
        4.0
        >>> add(-10, 5)
        -5
    """
    return a + b
```

```python
def divide(a: int | float, b: int | float) -> float:
    """Performs floating-point division of one number by another.

    This function calculates the quotient when `a` (numerator) is divided by
    `b` (denominator). The result will always be a float.
    It's crucial that the denominator `b` is not zero to avoid a `ZeroDivisionError`.

    Args:
        a: The numerator, an integer or a float.
        b: The denominator, an integer or a float. Must not be zero.

    Returns:
        The floating-point result of `a` divided by `b`.

    Raises:
        ZeroDivisionError: If `b` is 0, as division by zero is mathematically
                           undefined and raises a runtime error in Python.

    Examples:
        >>> divide(10, 2)
        5.0
        >>> divide(7, 3)
        2.3333333333333335
        >>> divide(10.0, 0.5)
        20.0
    """
    return a / b
```

Files

```python
def is_even(number: int) -> bool:
    """Determines if a given integer is an even number.

    An even number is an integer that is exactly divisible by 2.
    This function uses the modulo operator to check for divisibility.

    Args:
        number: The integer to check for evenness.

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

    Examples:
        >>> is_even(4)
        True
        >>> is_even(7)
        False
        >>> is_even(0) # Zero is considered an even number
        True
        >>> is_even(-2)
        True
    """
    return number % 2 == 0
```

```python
def find_max(numbers: list[int | float]) -> int | float:
    """Identifies and returns the largest numerical element within a list.

    This function iterates through a list of numbers (integers or floats)
    and returns the highest value found. It will raise an error if the
    provided list is empty, as there would be no maximum element.

    Args:
        numbers: A list containing integers or floats.
                 The list should not be empty.

    Returns:
        The single largest number present in the input list.

    Raises:
        ValueError: If the input `numbers` list is empty, as `max()` cannot
                    operate on an empty sequence.

    Examples:
        >>> find_max([1, 5, 2, 9, 3])
        9
        >>> find_max([-10, -5, -1])
        -1
        >>> find_max([3.14, 2.71, 1.618])
        3.14
    """
    return max(numbers)
```

Observation

In this experiment, AI was used to generate Google-style docstrings for existing Python functions. The zero-shot prompt produced correct but basic documentation. When a context-based prompt was used, the docstrings became more detailed, professional, and clearer. It was observed that giving proper context improves the quality and readability of AI-generated documentation.

Zero-shot prompting generated simple and correct docstrings.

Context-based prompting produced more professional and detailed documentation.

The context-based prompt improved readability and clarity.

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 syntaxThe goal is to improve clarity without cluttering the code.

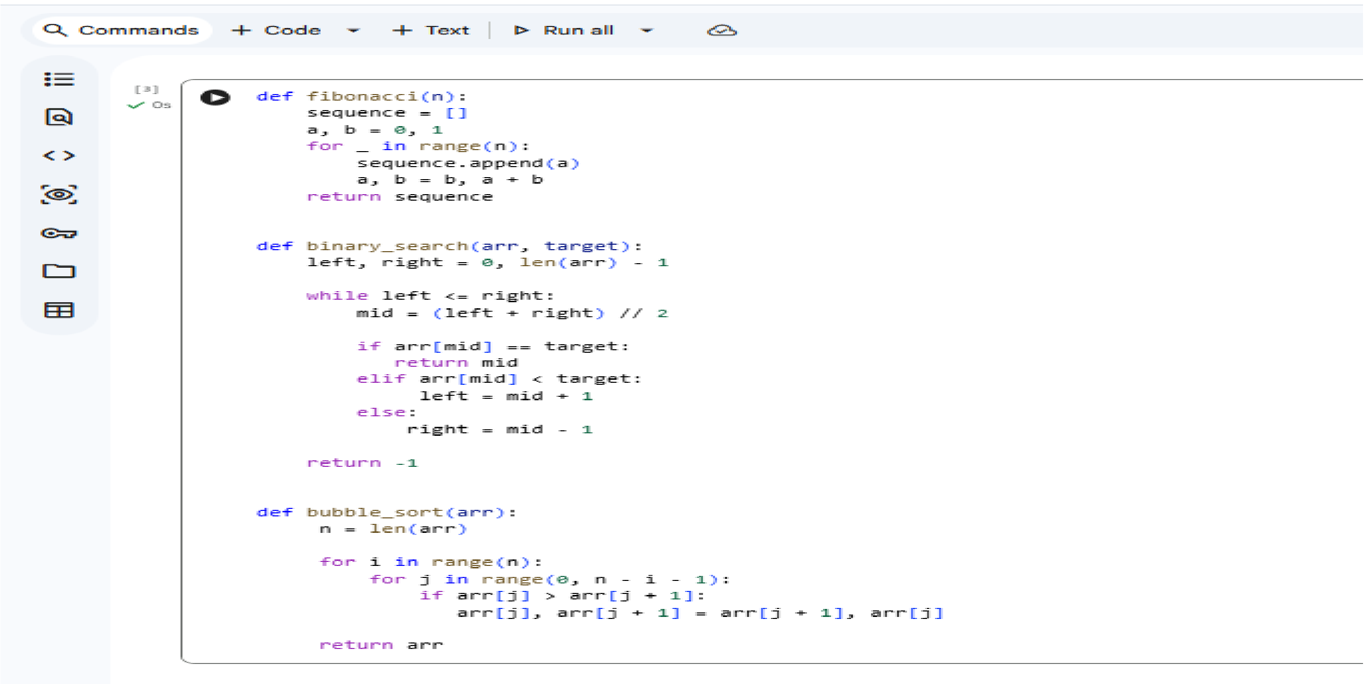
Expected Outcome

- 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

Original Code (Without Comments)

```
def fibonacci(n):
    sequence = []
    a, b = 0, 1
    for _ in range(n):
        sequence.append(a)
        a, b = b, a + b
    return sequence
```

```
def binary_search(arr, target):
    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 bubble_sort(arr):
    n = len(arr)
     for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
     return arr
```




```
def fibonacci(n):
    sequence = []
    a, b = 0, 1
    for _ in range(n):
        sequence.append(a)
        a, b = b, a + b
    return sequence


def binary_search(arr, target):
    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 bubble_sort(arr):
     n = len(arr)

     for i in range(n):
         for j in range(0, n - i - 1):
             if arr[j] > arr[j + 1]:
                 arr[j], arr[j + 1] = arr[j + 1], arr[j]

     return arr
```

AI to Generate Inline Comments

Prompt

Insert concise and meaningful inline comments into the following Python code. Only comment on complex or non-obvious logic.Do NOT comment on trivial syntax. Explain WHY the logic is used, not what Python syntax does.

AI-Generated Code

```python
def fibonacci(n):
    sequence = []
    a, b = 0, 1  # Initialize the first two Fibonacci numbers
    for _ in range(n):
        sequence.append(a)
        a, b = b, a + b  # Calculate the next Fibonacci number: new 'a' is old 'b', new 'b' is old 'a' + old 'b'
    return sequence


def binary_search(arr, target):
    left, right = 0, len(arr) - 1  # Initialize pointers for the search range

    while left <= right:  # Continue as long as there's a valid search space
        mid = (left + right) // 2  # Calculate the middle index to split the search space

        if arr[mid] == target:
            return mid  # Target found at the middle index
        elif arr[mid] < target:
            left = mid + 1  # Target is in the right half, so discard the left half and middle
        else:
            right = mid - 1  # Target is in the left half, so discard the right half and middle

    return -1  # Target not found after exhausting the search space


def bubble_sort(arr):
    n = len(arr)

    for i in range(n):  # Outer loop for passes; 'i' counts how many elements are already in their correct sorted position at the end
        # Last 'i' elements are already in place, so the inner loop's range shrinks
        for j in range(0, n - i - 1):  # Inner loop to compare adjacent elements and bubble up the largest
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]  # Swap elements if they are in the wrong order to move larger elements towards the end

    return arr
```

Test the Program

print(fibonacci(7))

print(binary_search([1, 3, 5, 7, 9], 5))

print(bubble_sort([5, 2, 9, 1]))

```python
print(fibonacci(7))
print(binary_search([1, 3, 5, 7, 9], 5))
print(bubble_sort([5, 2, 9, 1]))
```

```
[0, 1, 1, 2, 3, 5, 8]
2
[1, 2, 5, 9]
```

Observation

In this experiment, AI was used to insert inline comments into complex Python logic. The AI successfully added concise comments explaining the purpose of loops, conditions, and algorithm decisions. Trivial syntax was not commented, avoiding unnecessary clutter. The readability and maintainability of the code improved significantly without affecting functionality.

Task 3: Generating Module-Level Documentation for a Python Package
Scenario
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.
Task Description
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:
• 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 Focus on clarity and professional tone.

Expected Outcome
• 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

Python Module (Without Module Docstring)

```
import math
class Calculator:
    def add(self, a, b):
        return a + b
def subtract(self, a, b):
        return a - b
def multiply(self, a, b):
        return a * b
def divide(self, a, b):
        if b == 0:
            raise ValueError("Cannot divide by zero")
        return a / b
def factorial(n):
    if n < 0:
        raise ValueError("Negative numbers not allowed")
    return math.factorial(n)
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            return False
    return True
```

Commands + Code + Text Run all Saving...

Gemini

[6]

```python
import math


class Calculator:
    def add(self, a, b):
        return a + b

    def subtract(self, a, b):
        return a - b

    def multiply(self, a, b):
        return a * b

    def divide(self, a, b):
        if b == 0:
            raise ValueError("Cannot divide by zero")
        return a / b


def factorial(n):
    if n < 0:
        raise ValueError("Negative numbers not allowed")
    return math.factorial(n)


def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            return False
    return True
```

AI to Generate Module-Level Docstring

prompt:

Generate a professional multi-line module-level docstring for the following Python module.

The docstring must include:

- Purpose of the module

- Required libraries/dependencies

- Brief description of key classes and functions

- Short example usage

Maintain a professional tone suitable for a real-world repository.

Commands + Code + Text Run all

[9]

```python
import math


class Calculator:
    def add(self, a, b):
        return a + b

    def subtract(self, a, b):
        return a - b

    def multiply(self, a, b):
        return a * b

    def divide(self, a, b):
        if b == 0:
            raise ValueError("Cannot divide by zero")
        return a / b


def factorial(n):
    if n < 0:
        raise ValueError("Negative numbers not allowed")
    return math.factorial(n)


def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            return False
    return True
```

Test the Module

calc = Calculator()

print(calc.multiply(4, 5))

print(factorial(5))

print(is_prime(11))

[10]

```python
calc = Calculator()
print(calc.multiply(4, 5))
print(factorial(5))
print(is_prime(11))
```

```
20
120
True
```

Observation

In this experiment, AI was used to generate a module-level docstring for a complete Python module. The generated documentation clearly described the purpose, dependencies, key components, and example usage. The professional tone and structured format improved clarity and made the module suitable for real-world repositories.

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

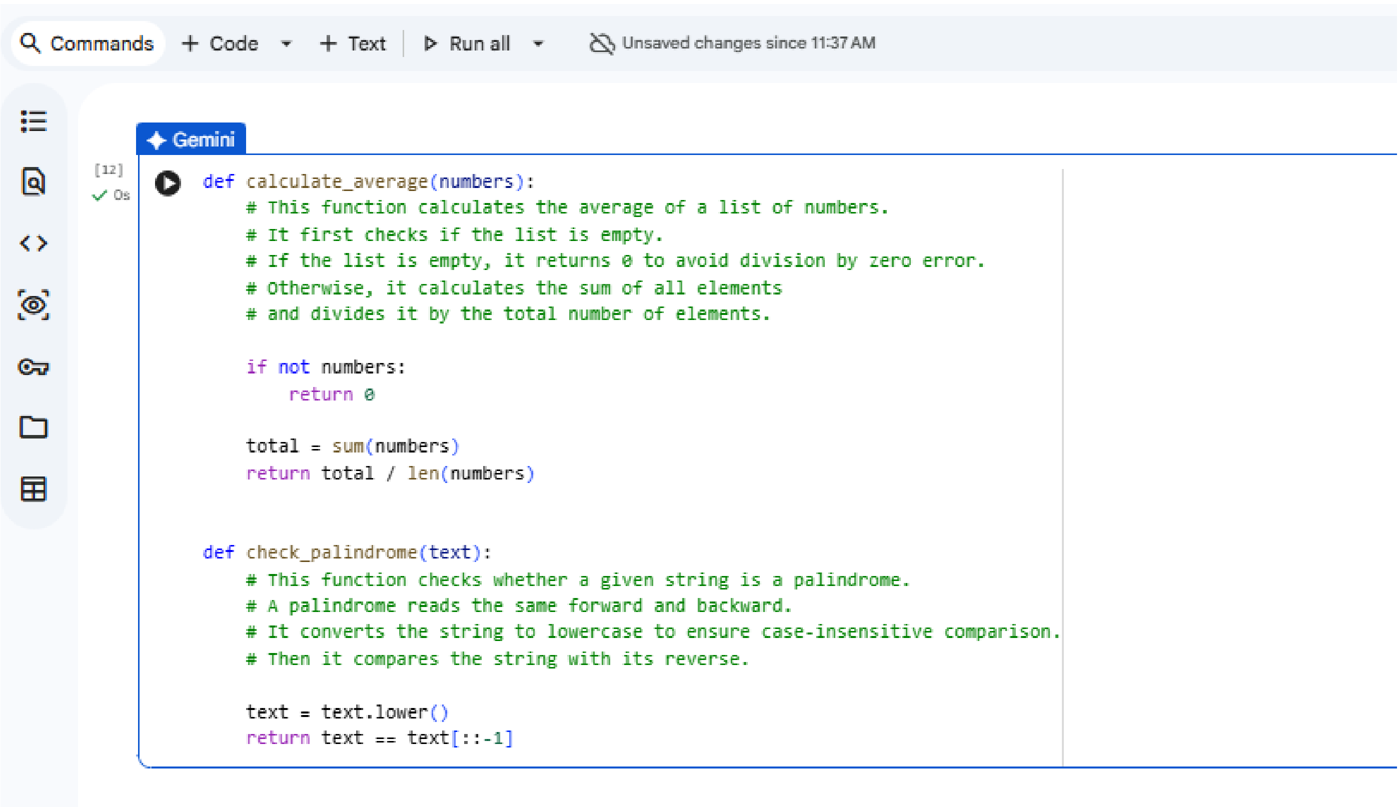
Expected Outcome

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

Legacy Code (With Long Inline Comments)

```
def calculate_average(numbers):

    # This function calculates the average of a list of numbers.

    # It first checks if the list is empty.

    # If the list is empty, it returns 0 to avoid division by zero error.

    # Otherwise, it calculates the sum of all elements

    # and divides it by the total number of elements.

    if not numbers:

        return 0
```

```
    total = sum(numbers)
    return total / len(numbers)
def check_palindrome(text):
    # This function checks whether a given string is a palindrome.
    # A palindrome reads the same forward and backward.
    # It converts the string to lowercase to ensure case-insensitive comparison.
    # Then it compares the string with its reverse.
    text = text.lower()
    return text == text[::-1]
```



AI to Convert Comments into Docstrings

Convert the inline comments inside the following Python functions into structured Google-style docstrings. Preserve the original meaning. Remove redundant inline comments after conversion.Keep the function logic unchanged.

```python
def calculate_average(numbers):
    """Calculates the average of a list of numbers.

    This function computes the arithmetic mean of a list of numerical values.
    It handles the edge case of an empty list by returning 0 to prevent division by zero errors.

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

    Returns:
        float: The average of the numbers in the list. Returns 0.0 if the list is empty.
    """
    if not numbers:
        return 0

    total = sum(numbers)
    return total / len(numbers)


def check_palindrome(text):
    """Checks if a given string is a palindrome.

    A palindrome is a word, phrase, or sequence that reads the same backward as forward.
    This function performs a case-insensitive check by converting the input string to lowercase
    before comparing it with its reversed version.

    Args:
        text (str): The string to be checked.

    Returns:
        bool: True if the string is a palindrome, False otherwise.
    """
    text = text.lower()
    return text == text[::-1]
```

Test the Functions

print(calculate_average([10, 20, 30]))

print(check_palindrome("Madam"))

```python
print(calculate_average([10, 20, 30]))
print(check_palindrome("Madam"))
```

```
20.0
True
```

Observation

In this experiment, AI was used to convert legacy inline comments into structured Google-style docstrings. The original meaning of the comments was preserved while redundant inline comments were removed. The resulting code became cleaner, more standardized, and easier to maintain. This improved documentation consistency across the codebase.

Task 5: Building a Mini Automatic Documentation Generator

Scenario

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

Task Description

Design a small Python utility that:

• Reads a given .py file

• Automatically detects:

o Functions

o 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

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

Create a Sample Python File (To Process)


```python
# Create a sample Python file

sample_code = """
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def greet(self):
        return f"Hello, my name is {self.name}"

def add(a, b):
    return a + b

def multiply(x, y):
    return x * y
"""

with open("sample_module.py", "w") as f:
    f.write(sample_code)

print("Sample file created successfully!")
```

Build the Documentation Generator Utility

```
Sample file created successfully!
```

```python
import ast

def generate_placeholder_docstring(name, obj_type):
    """
    Generates a basic Google-style placeholder docstring.
    """
    if obj_type == "function":
        return f'''    """
    TODO: Describe the purpose of the function `{name}`.

    Args:
        TODO: Add parameter descriptions.

    Returns:
        TODO: Describe return value.
    """\n'''

    elif obj_type == "class":
        return f'''    """
    TODO: Describe the purpose of the class `{name}`.

    Attributes:
        TODO: Describe class attributes.

    Methods:
        TODO: Describe key methods.
    """\n'''

def insert_docstrings(file_path):
    with open(file_path, "r") as file:
        lines = file.readlines()

    tree = ast.parse("".join(lines))
    new_lines = lines.copy()

    # Offset helps adjust line numbers after inserting docstrings
    offset = 0

    for node in ast.walk(tree):
        if isinstance(node, ast.FunctionDef):
            docstring = generate_placeholder_docstring(node.name, "function")
            insert_position = node.body[0].lineno - 1 + offset
            new_lines.insert(insert_position, docstring)
            offset += 1

        elif isinstance(node, ast.ClassDef):
            docstring = generate_placeholder_docstring(node.name, "class")
            insert_position = node.body[0].lineno - 1 + offset
            new_lines.insert(insert_position, docstring)
            offset += 1

    # Save modified file
    output_file = "documented_" + file_path
    with open(output_file, "w") as file:
        file.writelines(new_lines)

    t(f"Documentation scaffolding added! Saved as {output_file}")
```

Run cell (Ctrl+Enter)
cell executed since last change

executed by RAVULA VINEESHA
3:55 PM (8 minutes ago)

Run the Generator

```
[6]
insert_docstrings("sample_module.py")
```

```
Documentation scaffolding added! Saved as documented_sample_module.py
```

```
[7]
with open("documented_sample_module.py", "r") as f:
    print(f.read())
```

```
...
class Student:
    """
    TODO: Describe the purpose of the class `Student`.

    Attributes:
        TODO: Describe class attributes.

    Methods:
        TODO: Describe key methods.
    """
    def __init__(self, name, age):
        self.name = name
        self.age = age
    """
    TODO: Describe the purpose of the function `__init__`.

    Args:
        TODO: Add parameter descriptions.

    Returns:
        TODO: Describe return value.
    """

    def greet(self):
        return f"Hello, my name is {self.name}"

    """
    TODO: Describe the purpose of the function `greet`.

    Args:
        TODO: Add parameter descriptions.

    Returns:
        TODO: Describe return value.
    """
def add(a, b):
    """
    TODO: Describe the purpose of the function `add`.

    Args:
        TODO: Add parameter descriptions.

    Returns:
        TODO: Describe return value.
    """
    return a + b

def multiply(x, y):
    """
    TODO: Describe the purpose of the function `multiply`.

    Args:
        TODO: Add parameter descriptions.

    Returns:
        TODO: Describe return value.
    """
```

Observation

AI-assisted development helped design a mini documentation generator using Python's AST module. The script successfully detected functions and classes and inserted placeholder Google-style docstrings. This demonstrates how AI can assist in automating documentation scaffolding in real-world projects.