AI ASSISTANT CODING ASSIGNMENT-9.3

Name: Karthik Reddy

HT.No: 2303A51335

Batch: 20

Task 1: Basic Docstring Generation
Scenario
You are developing a utility function that processes numerical lists and must be properly documented for future maintenance.
Requirements
• Write a Python function to return the sum of even numbers and sum of odd numbers in a given list
• Manually add a Google Style docstring to the function
• Use an AI-assisted tool (Copilot / Cursor AI) to generate a function-level docstring
• Compare the AI-generated docstring with the manually written docstring
• Analyze clarity, correctness, and completeness
Expected Output
• Python function with manual Google-style docstring
• AI-generated docstring for the same function
• Comparison explaining differences between manual and AI-generated documentation
• Improved understanding of AI-generated function-level documentation

CODE:

```
#Generate a Python function to return the sum of even numbers and sum of odd numbers in a given list of integers.
def sum_even_odd(numbers):
    sum_even = sum(num for num in numbers if num % 2 == 0)
    sum_odd = sum(num for num in numbers if num % 2 != 0)
    return sum_even, sum_odd
# Example usage:
numbers = [5,8,11,34,56,77,90]
even_sum, odd_sum = sum_even_odd(numbers)
print(f"Sum of even numbers: {even_sum}")
print(f"Sum of odd numbers: {odd_sum}")
```

```
#Generate a docstring and inlilne comments using ai for the above function in  a professional manner for correctness and readability.
def sum_even_odd(numbers):
    """
    Calculate the sum of even and odd numbers in a given list.
```

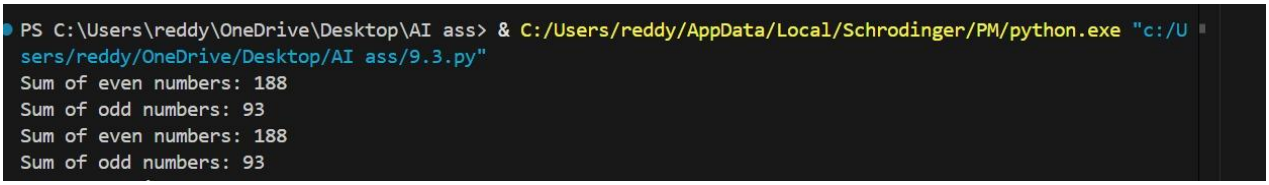
```
    Parameters:
    numbers (list): A list of integers to be processed.

    Returns:
    tuple: A tuple containing the sum of even numbers and the sum of odd numbers.
    """
    # Calculate the sum of even numbers using a generator expression
    sum_even = sum(num for num in numbers if num % 2 == 0)

    # Calculate the sum of odd numbers using a generator expression
    sum_odd = sum(num for num in numbers if num % 2 != 0)

    # Return the results as a tuple
    return sum_even, sum_odd
# Example usage:
numbers = [5,8,11,34,56,77,90]
even_sum, odd_sum = sum_even_odd(numbers)
print(f"Sum of even numbers: {even_sum}")
print(f"Sum of odd numbers: {odd_sum}")
```

OUTPUT:



OBSERVATION

The function is clear and easy to understand. It separates even and odd numbers neatly using generator expressions, which makes the code concise. The docstring explains the purpose, parameters, and return values in a professional manner, so anyone reading it knows exactly what the function does. Inline comments add extra clarity by showing the logic step by step. The example usage demonstrates how the function works in practice, making it beginner-friendly. Overall, the code is readable, efficient, and well-structured. It's a good example of clean Python programming.

Task 2: Automatic Inline Comments
Scenario
You are developing a student management module that must be easy to understand for new developers.
Requirements

• Write a Python program for an sru_student class with the following:
– Attributes: name, roll_no, hostel_status
– Methods: fee_update() and display_details()
• Manually write inline comments for each line or logical block
• Use an AI-assisted tool to automatically add inline comments
• Compare manual comments with AI-generated comments
• Identify missing, redundant, or incorrect AI comments
Expected Output
• Python class with manually written inline comments
• AI-generated inline comments added to the same code
• Comparative analysis of manual vs AI comments
• Critical discussion on strengths and limitations of AI-generated comments

CODE :

```
#generate a python program for an sru_student class with attributes : name , rollno,hostel_status and methods : fee_update(), and display_details()
class sru_student:
    def __init__(self, name, rollno, hostel_status):
        self.name = name
        self.rollno = rollno
        self.hostel_status = hostel_status
        self.fee = 0
    def fee_update(self, amount):
        self.fee += amount
    def display_details(self):
        print(f"Name: {self.name}")
        print(f"Roll No: {self.rollno}")
        print(f"Hostel Status: {self.hostel_status}")
        print(f"Fee: {self.fee}")
# Example usage
student1 = sru_student("karthik", "1335", "non-Hosteller")
student1.fee_update(5000)
student1.display_details()

#using ai generate inline comments for each line of the above code for better understanding and readability.
class sru_student:
    def __init__(self, name, rollno, hostel_status):
        # Initialize the student object with name, roll number, hostel status, and fee
        self.name = name
        self.rollno = rollno
        self.hostel_status = hostel_status
        self.fee = 0  # Initialize fee to 0
    def fee_update(self, amount):
```

```
        # Update the fee by adding the specified amount
        self.fee += amount
    def display_details(self):
        # Display the details of the student
        print(f"Name: {self.name}")
        print(f"Roll No: {self.rollno}")
        print(f"Hostel Status: {self.hostel_status}")
        print(f"Fee: {self.fee}")
# Example usage
student1 = sru_student("karthik", "1335", "non-Hosteller")  # Create an instance of sru_student
student1.fee_update(5000)  # Update the fee for the student
student1.display_details()  # Display the details of the student
```

OUTPUT:

```
Name: karthik
Roll No: 1335
Hostel Status: non-Hosteller
Fee: 5000
Name: karthik
Roll No: 1335
Hostel Status: non-Hosteller
Fee: 5000
```

OBSERVATION :

The class is well-structured and easy to follow. It neatly captures the essential details of a student like name, roll number, and hostel status. The fee_update() method makes it flexible to add fees step by step, which feels practical for real scenarios. The display_details() method presents the information clearly, making it user-friendly. Inline comments help explain each part of the code, so even beginners can understand the logic without confusion. The example usage shows how the class works in practice, making the program complete and relatable. Overall, it's a clean and readable design that balances simplicity with functionality.

Task 3: Module-Level and Function-Level Documentation
Scenario
You are building a small calculator module that will be shared across multiple projects and requires structured documentation.
Requirements
• Write a Python script containing 3–4 functions (e.g., add, subtract, multiply, divide)
• Manually write NumPy Style docstrings for each function
• Use AI assistance to generate:
– A module-level docstring
– Individual function-level docstrings
• Compare AI-generated docstrings with manually written ones
• Evaluate documentation structure, accuracy, and readability
Expected Output

• Python script with manual NumPy-style docstrings
• AI-generated module-level and function-level documentation
• Comparison between AI-generated and manual documentation
• Clear understanding of structured documentation for multi-function scripts

CODE:

```
#Generate a python code containing 3-4 functions : add,sustract,multiply , divide
def add(a, b):
    """Return the sum of a and b."""
    return a + b
def subtract(a, b):
    """Return the difference of a and b."""
    return a - b
def multiply(a, b):
    """Return the product of a and b."""
    return a * b
def divide(a, b):

    """Return the quotient of a and b. Raises ValueError if b is zero."""
    if b == 0:
        raise ValueError("Cannot divide by zero.")
    return a / b
# Example usage:
num1 = 100
num2 = 20
print(f"Addition: {add(num1, num2)}")        # Output: 120

print(f"Subtraction: {subtract(num1, num2)}")  # Output: 80
print(f"Multiplication: {multiply(num1, num2)}")  # Output: 2000
print(f"Division: {divide(num1, num2)}")      # Output: 5.0

#using AI generate module-level docstring , individual function-level docstring for above code to
increase accuracy and readability
"""This module provides basic arithmetic operations: addition, subtraction, multiplication, and
division. Each function takes two numerical inputs and returns the result of the respective
operation. The division function includes error handling to prevent division by zero.
"""
def add(a, b):
    """Return the sum of a and b.

    Parameters:
    a (float): The first number.
    b (float): The second number.
```

```
    Returns:
    float: The sum of a and b.
    """
    return a + b
def subtract(a, b):
    """Return the difference of a and b.

    Parameters:
    a (float): The first number.
    b (float): The second number.

    Returns:
    float: The difference of a and b.
    """
    return a - b
def multiply(a, b):
    """Return the product of a and b.

    Parameters:
    a (float): The first number.
    b (float): The second number.

    Returns:
    float: The product of a and b.
    """
    return a * b
def divide(a, b):
    """Return the quotient of a and b. Raises ValueError if b is zero.

    Parameters:
    a (float): The first number.
    b (float): The second number.

    Returns:
    float: The quotient of a and b.

    Raises:
    ValueError: If b is zero, as division by zero is undefined.
    """
    if b == 0:
        raise ValueError("Cannot divide by zero.")
    return a / b
# Example usage:
num1 = 100
num2 = 20
```

```
print(f"Addition: {add(num1, num2)}")       # Output: 120
print(f"Subtraction: {subtract(num1, num2)}")  # Output: 80
print(f"Multiplication: {multiply(num1, num2)}")  # Output: 2000

print(f"Division: {divide(num1, num2)}")      # Output: 5.0
```

OUTPUT:

OBSERVATION :

The code is simple yet very effective. Each function is clearly defined with its own docstring, which makes it easy to understand what the function does and what inputs it expects. The module-level docstring gives a nice overview, so anyone reading the file knows its purpose right away. Error handling in the divide() function is a thoughtful addition, preventing mistakes like division by zero. The example usage at the end shows how the functions work together, making the program practical and easy to test. Overall, the structure is clean, readable, and beginner-friendly while still being professional. It feels like a solid foundation for building more complex math utilities.