AI ASSISTANT CODING - ASSIGNMENT-9.3

Name: Valaboju Harshavardhanu **Roll No:** 2303A52354 **Batch:** 41

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

Prompt used:

#Generate a calculator with multiple functions. Using a python script without any docstrings.

#Generate Google-style function docstrings for each function where it should clearly explain which function is doing what and what values it returning and a brief explanation of every function.

Code:

Without Docstrings:

```
def add(*args):
    return sum(args)

def subtract(num1, num2):
    return num1 - num2

def multiply(*args):
    result = 1
    for num in args:
        result *= num
    return result

def divide(num1, num2):
    if num2 == 0:
        return "Error: Cannot divide by zero!"
    return num1 / num2

def power(base, exponent):
    return base ** exponent

def calculator():
    print("\n--- Simple Calculator ---")
    print("Select operation:")
    print("1. Add (n numbers)")
    print("2. Subtract (2 numbers)")
    print("3. Multiply (n numbers)")
    print("4. Divide (2 numbers)")
    print("5. Power (2 numbers)")
    print("6. Exit")
```

```
    while True:
        choice = input("Enter choice(1/2/3/4/5/6): ")

        if choice in ('1', '3'):
            nums_str = input("Enter numbers separated by space: ")
            try:
                numbers = [float(x) for x in nums_str.split()]
            except ValueError:
                print("Invalid input. Please enter valid numbers.")
                continue
            if choice == '1':
                print(f"Result: {add(*numbers)}")
            else: # choice == '3'
                print(f"Result: {multiply(*numbers)}")
        elif choice in ('2', '4', '5'):
            try:
                num1 = float(input("Enter first number: "))
                num2 = float(input("Enter second number: "))
            except ValueError:
                print("Invalid input. Please enter valid numbers.")
                continue

            if choice == '2':
                print(f"Result: {subtract(num1, num2)}")
            elif choice == '4':
                print(f"Result: {divide(num1, num2)}")
            else: # choice == '5'
                print(f"Result: {power(num1, num2)}")
        elif choice == '6':
            print("Exiting calculator. Goodbye!")
            break
        else:
            print("Invalid input. Please enter a valid choice.")

# To run the calculator, uncomment the line below:
calculator()
```

With Docstrings:

```
def add(*args):
    """Performs addition on an arbitrary number of input values.

    This function takes any number of numerical arguments and returns their sum.

    Args:
        *args: Variable length argument list of numbers to be added.

    Returns:
        float or int: The sum of all provided numbers.
    """
    return sum(args)

def subtract(num1, num2):
    """Performs subtraction of two numbers.

    This function subtracts the second number from the first number.

    Args:
        num1 (float or int): The first number (minuend).
        num2 (float or int): The second number (subtrahend).

    Returns:
        float or int: The result of num1 - num2.
    """
    return num1 - num2
```

```python
def multiply(*args):
    """Performs multiplication on an arbitrary number of input values.
    This function takes any number of numerical arguments and returns their product.
    Args:
        *args: Variable length argument list of numbers to be multiplied.
    Returns:
        float or int: The product of all provided numbers.
    """
    result = 1
    for num in args:
        result *= num
    return result
def divide(num1, num2):
    """Performs division of two numbers, handling division by zero.
    This function divides the first number by the second number. It returns
    an error message if the divisor is zero.
    Args:
        num1 (float or int): The numerator.
        num2 (float or int): The denominator.
    Returns:
        float or str: The result of num1 / num2, or an error string if num2 is zero.
    """
    if num2 == 0:
        return "Error: Cannot divide by zero!"
    return num1 / num2

def power(base, exponent):
    """Calculates the power of a base number raised to an exponent.
    This function computes base ** exponent.
    Args:
        base (float or int): The base number.
        exponent (float or int): The exponent.
    Returns:
        float or int: The result of base raised to the power of exponent.
    """
    return base ** exponent

def calculator():
    """A simple command-line calculator program.
    This function presents a menu of operations to the user (add, subtract,
    multiply, divide, power) and takes numerical input to perform the chosen
    calculation. It continues to run until the user chooses to exit.
    Includes basic input validation for numbers.
    """
    print("\n--- Simple Calculator ---")
    print("Select operation:")
    print("1. Add (n numbers)")
    print("2. Subtract (2 numbers)")
    print("3. Multiply (n numbers)")
    print("4. Divide (2 numbers)")
    print("5. Power (2 numbers)")
    print("6. Exit")
```

```python
    while True:
        choice = input("Enter choice(1/2/3/4/5/6): ")

        if choice in ('1', '3'):
            nums_str = input("Enter numbers separated by space: ")
            try:
                numbers = [float(x) for x in nums_str.split()]
            except ValueError:
                print("Invalid input. Please enter valid numbers.")
                continue
            if choice == '1':
                print(f"Result: {add(*numbers)}")
            else: # choice == '3'
                print(f"Result: {multiply(*numbers)}")
        elif choice in ('2', '4', '5'):
            try:
                num1 = float(input("Enter first number: "))
                num2 = float(input("Enter second number: "))
            except ValueError:
                print("Invalid input. Please enter valid numbers.")
                continue

            if choice == '2':
                print(f"Result: {subtract(num1, num2)}")
            elif choice == '4':
                print(f"Result: {divide(num1, num2)}")
            else: # choice == '5'
                print(f"Result: {power(num1, num2)}")
        elif choice == '6':
            print("Exiting calculator. Goodbye!")
            break
        else:
            print("Invalid input. Please enter a valid choice.")

# To run the calculator, uncomment the line below:
calculator()
```

Output:

```
...
    --- Simple Calculator ---
    Select operation:
    1. Add (n numbers)
    2. Subtract (2 numbers)
    3. Multiply (n numbers)
    4. Divide (2 numbers)
    5. Power (2 numbers)
    6. Exit
    Enter choice(1/2/3/4/5/6): 2
    Enter first number: 5
    Enter second number: 2
    Result: 3.0
    Enter choice(1/2/3/4/5/6): 6
    Exiting calculator. Goodbye!
```

Justification:

Providing code both with and without docstrings, offers flexibility. Docstrings are crucial for readability and maintenance in collaborative projects, while a version without them might be preferred for concise examples or when code size is a critical factor. Generating them in separate cells allows for easy comparison or use in different scenarios without altering previous work

Task 2: Enhancing Readability Through AI-Generated Inline

Comments.

Prompt used:

#Generate a python script in which it contains loops, conditional logic and Algorithms without inline comments or docstrings.

#Generate inline comments only for complex or non- obvious logic but on syntax of the function.

Code:

Without Docstrings:

```python
def find_primes_up_to_n(limit):
    if limit < 2:
        return []

    primes = []
    for num in range(2, limit + 1):
        is_prime = True

        if num == 2:
            is_prime = True
        elif num % 2 == 0:
            is_prime = False
        else:
            for i in range(3, int(num**0.5) + 1, 2):
                if num % i == 0:
                    is_prime = False
                    break

        if is_prime:
            primes.append(num)

    return primes

while True:
    try:
        user_limit_str = input("Enter a number to find primes up to (e.g., 20): ")
        user_limit = int(user_limit_str)
        if user_limit < 0:
            print("Please enter a non-negative number.")
        else:
            break
    except ValueError:
        print("Invalid input. Please enter an integer.")

print(f"\nFinding prime numbers up to {user_limit}...")
prime_numbers = find_primes_up_to_n(user_limit)
print(f"Prime numbers up to {user_limit} are: {prime_numbers}")
```

With inline comments:

```python
def find_primes_up_to_n(limit):
    """Finds all prime numbers up to a given limit.
    Args:
        limit (int): The upper bound (inclusive) for finding prime numbers.
    Returns:
        list: A list of prime numbers found within the range [2, limit].
    """
    if limit < 2:
        return []
    primes = []
    # Outer loop: Iterate through each number from 2 up to the limit
    for num in range(2, limit + 1):
        is_prime = True  # Assume the current number is prime initially
        # Conditional Logic 1: Check for special case of 2 (first prime)
        if num == 2:
            is_prime = True
        # Conditional Logic 2: If number is even and greater than 2, it's not prime
        elif num % 2 == 0:
            is_prime = False
        else:
            # Algorithm: Primality test for odd numbers
            # Inner loop: Check for divisibility from 3 up to the square root of num
            # We only need to check odd divisors up to sqrt(num)
            # because if num has a divisor larger than its square root, it must also have one smaller.
            # We start from 3 and increment by 2 (checking only odd numbers).
            for i in range(3, int(num**0.5) + 1, 2):
                # Conditional Logic 3: If num is divisible by i, it's not prime
                if num % i == 0:
                    is_prime = False
                    break  # No need to check further divisors if one is found
```

```
        # Conditional Logic 4: If after all checks, is_prime is still True, add to list
        if is_prime:
            primes.append(num)
    return primes
# --- Main execution block ---
while True:
    try:
        user_limit_str = input("Enter a number to find primes up to (e.g., 20): ")
        user_limit = int(user_limit_str)
        if user_limit < 0:
            print("Please enter a non-negative number.")
        else:
            break
    except ValueError:
        print("Invalid input. Please enter an integer.")
print(f"\nFinding prime numbers up to {user_limit}...")
prime_numbers = find_primes_up_to_n(user_limit)
print(f"Prime numbers up to {user_limit} are: {prime_numbers}")
```

Output:

```
Enter a number to find primes up to (e.g., 20): 15

Finding prime numbers up to 15...
Prime numbers up to 15 are: [2, 3, 5, 7, 11, 13]
```

Justification:

This last version demonstrates adding inline comments selectively, focusing only on complex or non-obvious logic, which is a common practice for maintaining code clarity without over-commenting basic syntax. This approach caters to different readability needs during development.

Task 3: Generating Module-Level Documentation for a Python Package.

Prompt used:

#Generate a python script in which it contains loops, conditional logic and Algorithms without inline comments or docstrings.

#Generate a module-level docstring at the top of the file that includes The purpose of the module ,Required libraries , a brief description and short example

Code:

Without module-level docstring:

```python
def add(*args):
    return sum(args)

def subtract(num1, num2):
    return num1 - num2

def multiply(*args):
    result = 1
    for num in args:
        result *= num
    return result

def divide(num1, num2):
    if num2 == 0:
        return "Error: Cannot divide by zero!"
    return num1 / num2

def power(base, exponent):
    return base ** exponent

def calculator():
    print("\n--- Simple Calculator ---")
    print("Select operation:")
    print("1. Add (n numbers)")
    print("2. Subtract (2 numbers)")
    print("3. Multiply (n numbers)")
    print("4. Divide (2 numbers)")
    print("5. Power (2 numbers)")
    print("6. Exit")
```

```python
    while True:
        choice = input("Enter choice(1/2/3/4/5/6): ")

        if choice in ('1', '3'):
            nums_str = input("Enter numbers separated by space: ")
            try:
                numbers = [float(x) for x in nums_str.split()]
            except ValueError:
                print("Invalid input. Please enter valid numbers.")
                continue
            if choice == '1':
                print(f"Result: {add(*numbers)}")
            else: # choice == '3'
                print(f"Result: {multiply(*numbers)}")
        elif choice in ('2', '4', '5'):
            try:
                num1 = float(input("Enter first number: "))
                num2 = float(input("Enter second number: "))
            except ValueError:
                print("Invalid input. Please enter valid numbers.")
                continue

            if choice == '2':
                print(f"Result: {subtract(num1, num2)}")
            elif choice == '4':
                print(f"Result: {divide(num1, num2)}")
            else: # choice == '5'
                print(f"Result: {power(num1, num2)}")
        elif choice == '6':
            print("Exiting calculator. Goodbye!")
            break
        else:
            print("Invalid input. Please enter a valid choice.")

# To run the calculator, uncomment the line below:
calculator()
```

With module-level docstring:

```
"""A simple command-line calculator module.
This module provides basic arithmetic operations (addition, subtraction, multiplication, division, and exponentiation)
through individual functions and a main interactive calculator interface.
It's designed for clarity and reusability, demonstrating fundamental Python concepts like functions, loops,
conditional logic, and user input handling.

Required Libraries/Dependencies:
    - No external libraries are required; only standard Python built-in functions are used.

Key Functions:
    - `add(*args)`: Sums an arbitrary number of values.
    - `subtract(num1, num2)`: Subtracts two numbers.
    - `multiply(*args)`: Multiplies an arbitrary number of values.
    - `divide(num1, num2)`: Divides two numbers, with zero-division handling.
    - `power(base, exponent)`: Calculates the power of a base to an exponent.
    - `calculator()`: The main interactive function to run the calculator.

Usage Example:
    To use the interactive calculator:
    >>> from __main__ import calculator # or import calculator if saved as a module
    >>> calculator()
    # Follow the on-screen prompts for operations.
"""
def add(*args):
    """Performs addition on an arbitrary number of input values.
    This function takes any number of numerical arguments and returns their sum.
    Args:
        *args: Variable length argument list of numbers to be added.
    Returns:
        float or int: The sum of all provided numbers.
    """
    return sum(args)
def subtract(num1, num2):
    """Performs subtraction of two numbers.
    This function subtracts the second number from the first number.
    Args:
        num1 (float or int): The first number (minuend).
        num2 (float or int): The second number (subtrahend).
    Returns:
        float or int: The result of num1 - num2.
    """
    return num1 - num2
def multiply(*args):
    """Performs multiplication on an arbitrary number of input values.
    This function takes any number of numerical arguments and returns their product.
    Args:
        *args: Variable length argument list of numbers to be multiplied.

    Returns:
        float or int: The product of all provided numbers.
    """
```

```
    result = 1
    for num in args:
        result *= num
    return result
def divide(num1, num2):
    """Performs division of two numbers, handling division by zero.
    This function divides the first number by the second number. It returns
    an error message if the divisor is zero.
    Args:
        num1 (float or int): The numerator.
        num2 (float or int): The denominator.
    Returns:
        float or str: The result of num1 / num2, or an error string if num2 is zero.
    """
    if num2 == 0:
        return "Error: Cannot divide by zero!"
    return num1 / num2
def power(base, exponent):
    """Calculates the power of a base number raised to an exponent.
    This function computes base ** exponent.
    Args:
        base (float or int): The base number.
        exponent (float or int): The exponent.
    Returns:
        float or int: The result of base raised to the power of exponent.
    """
    return base ** exponent
```

```python
def calculator():
    """A simple command-line calculator program.
    This function presents a menu of operations to the user (add, subtract,
    multiply, divide, power) and takes numerical input to perform the chosen
    calculation. It continues to run until the user chooses to exit.
    Includes basic input validation for numbers.
    """
    print("\n--- Simple Calculator ---")
    print("Select operation:")
    print("1. Add (n numbers)")
    print("2. Subtract (2 numbers)")
    print("3. Multiply (n numbers)")
    print("4. Divide (2 numbers)")
    print("5. Power (2 numbers)")
    print("6. Exit")
    while True:
        choice = input("Enter choice(1/2/3/4/5/6): ")
        if choice in ('1', '3'):
            nums_str = input("Enter numbers separated by space: ")
            try:
                numbers = [float(x) for x in nums_str.split()]
            except ValueError:
                print("Invalid input. Please enter valid numbers.")
                continue
            if choice == '1':
                print(f"Result: {add(*numbers)}")
            else: # choice == '3'
                print(f"Result: {multiply(*numbers)}")
        elif choice in ('2', '4', '5'):
            try:
                num1 = float(input("Enter first number: "))
                num2 = float(input("Enter second number: "))
            except ValueError:
                print("Invalid input. Please enter valid numbers.")
                continue
            if choice == '2':
                print(f"Result: {subtract(num1, num2)}")
            elif choice == '4':
                print(f"Result: {divide(num1, num2)}")
            else: # choice == '5'
                print(f"Result: {power(num1, num2)}")
        elif choice == '6':
            print("Exiting calculator. Goodbye!")
            break
        else:
            print("Invalid input. Please enter a valid choice.")

# Example of how to run the calculator:
# if __name__ == "__main__":
#     calculator()
```

Output:

```
--- Simple Calculator ---
Select operation:
1. Add (n numbers)
2. Subtract (2 numbers)
3. Multiply (n numbers)
4. Divide (2 numbers)
5. Power (2 numbers)
6. Exit
Enter choice(1/2/3/4/5/6): 2
Enter first number: 5
Enter second number: 2
Result: 3.0
Enter choice(1/2/3/4/5/6): 6
Exiting calculator. Goodbye!
```

Justification:

This task helps make the code easier for people to understand the moment they open the file. Instead of guessing what the module does, developers get a clear idea of its purpose, requirements, and how to use it. It saves time for new team members and reduces confusion. Overall, it makes the code more friendly, professional, and ready to be shared.

Task 4: Converting Developer Comments into Structured Docstrings.

Prompt used:

#Create a simple Python function with detailed inline comments explaining its purpose, arguments, and logic. Then, convert these inline comments into a structured Google-style docstring and remove the original inline comments. Finally, summarize the benefits of using docstrings for code readability and maintainability

Code:

With inline comments:

```python
def calculate_rectangle_area(length, width):
    # Purpose: This function calculates the area of a rectangle.

    # Argument: length (float or int) - Represents the length of the rectangle.
    # Argument: width (float or int) - Represents the width of the rectangle.

    # Logic: The area of a rectangle is calculated by multiplying its length by its width.
    area = length * width

    # Returns: float or int - The calculated area of the rectangle.
    return area

# Example usage of the function
# Define the dimensions of a rectangle
rect_length = 10
rect_width = 5

# Call the function to calculate the area
calculated_area = calculate_rectangle_area(rect_length, rect_width)

# Print the result
print(f"The area of a rectangle with length {rect_length} and width {rect_width} is: {calculated_area}")

# Another example
rect_length_float = 7.5
rect_width_float = 3.2
calculated_area_float = calculate_rectangle_area(rect_length_float, rect_width_float)
print(f"The area of a rectangle with length {rect_length_float} and width {rect_width_float} is: {calculated_area_float}")
```

With Google-style docstrings:

```python
def calculate_rectangle_area(length, width):
    """
    Calculates the area of a rectangle.
    This function takes the length and width of a rectangle and returns its area.
    Args:
        length (float or int): The length of the rectangle.
        width (float or int): The width of the rectangle.
    Returns:
        float or int: The calculated area of the rectangle (length * width).
    """
    area = length * width
    return area
# Example usage of the function
rect_length = 10
rect_width = 5
calculated_area = calculate_rectangle_area(rect_length, rect_width)
print(f"The area of a rectangle with length {rect_length} and width {rect_width} is: {calculated_area}")
rect_length_float = 7.5
rect_width_float = 3.2
calculated_area_float = calculate_rectangle_area(rect_length_float, rect_width_float)
print(f"The area of a rectangle with length {rect_length_float} and width {rect_width_float} is: {calculated_area_float}")
```

Output:

```
The area of a rectangle with length 10 and width 5 is: 50
The area of a rectangle with length 7.5 and width 3.2 is: 24.0
```

Justification:

The benefits of using docstrings were highlighted, including improved readability, enhanced maintainability, automated documentation generation, interactive help support, facilitated code collaboration, and self-contained information within the code.

Task 5: Building a Mini Automatic Documentation Generator.

Prompt used:

#Design a simple Python utility that reads a .py file, automatically detects functions and classes, and inserts placeholder Google-style docstrings for each detected function or class. The goal is documentation scaffolding, not perfect documentation. The solution should be suitable for execution in Google Colab.

Code:

Tool-Mini Automatic Documentation Generator.

```
%%writefile auto_doc_generator.py
import ast
def generate_google_docstring(name, node_type, args=None):
    if node_type == "class":
        return f'''"""
{name} class.
Attributes:
    TODO: Describe class attributes
"""
'''
    else:
        params = ""
        if args:
            params = "\n".join([f"    {arg}: TODO" for arg in args])
        return f'''"""
{name} function.
Args:
{params if params else "    None"}

Returns:
    TODO
"""
'''
```

```
class DocstringAdder(ast.NodeTransformer):

    def visit_FunctionDef(self, node):
        if ast.get_docstring(node) is None:
            args = [arg.arg for arg in node.args.args]
            doc = generate_google_docstring(node.name, "function", args)
            node.body.insert(0, ast.Expr(value=ast.Constant(doc)))
        return node
    def visit_ClassDef(self, node):
        if ast.get_docstring(node) is None:
            doc = generate_google_docstring(node.name, "class")
            node.body.insert(0, ast.Expr(value=ast.Constant(doc)))
        return node
def process_file(input_file, output_file):
    with open(input_file, "r", encoding="utf-8") as f:
        tree = ast.parse(f.read())

    tree = DocstringAdder().visit(tree)
    ast.fix_missing_locations(tree)

    updated_code = ast.unparse(tree)

    with open(output_file, "w", encoding="utf-8") as f:
        f.write(updated_code)

    print("Docstrings inserted successfully")

if __name__ == "__main__":
    process_file("input.py", "output.py")
```

```
Writing auto_doc_generator.py
```

Input file (undocumented code):

```
%%writefile input.py
def add(a, b):
    return a + b

class Calculator:
    def square(self, x):
        return x * x
```

```
Overwriting input.py
```

Tool:

```
!python auto_doc_generator.py
```

```
Docstrings inserted successfully
```

Output file (documented code):

```
!cat output.py
```

```
def add(a, b):
    '''"""
add function.

Args:
    a: TODO
    b: TODO

Returns:
    TODO
"""
'''
    return a + b

class Calculator:
    '''"""
Calculator class.

Attributes:
    TODO: Describe class attributes
"""
'''

    def square(self, x):
        return x * x
```

Justification:

Using AI-assisted logic, the above tool automatically reads Python files, detects functions and classes, and inserts Google-style docstrings, demonstrating documentation automation. This reduces the manual effort required from developers and helps maintain consistent documentation standards. The tool acts as a starting point for developers to quickly understand and improve code documentation