ASSIGNMENT-9.4

NAME:K.Abhinav
H.NO:2303A51856
B-13

Lab 9 – Documentation Generation: Automatic Documentation and Code Comments

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:
  - A brief description of the function
  - Parameters with data types
  - Return values
  - 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

```python
#Generate Google-style docstrings for the following Python functions. Include description, parameters with data types, return value, and one example usage.
def calculate_area(length, width):
    return length * width
def greet_user(name):
    return f"Hello, {name}!"
def calculate_average(numbers):
    return sum(numbers) / len(numbers)
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            return False
    return True
```

Output Doc String:

```
def calculate_area(length, width):
    """Calculates the area of a rectangle.

    Args:
        length (int or float): The length of the rectangle.
        width (int or float): The width of the rectangle.

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

    Example:
        >>> calculate_area(5, 10)
        50
    """
    return length * width


def greet_user(name):
    """Greets the user with a personalized message.

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

    Returns:
        str: A greeting message.

    Example:
        >>> greet_user("Alice")
        'Hello, Alice!'
```

```
    return f"Hello, {name}!"


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

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

    Returns:
        float: The average of the numbers in the list.

    Example:
        >>> calculate_average([1, 2, 3, 4, 5])
        3.0
    """
    return sum(numbers) / len(numbers)


def is_prime(n):
    """Checks if a given number is prime.

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

    Args:
        n (int): The integer to check for primality.

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

```
    Example:
        >>> is_prime(7)
        True
        >>> is_prime(4)
        False
    """
    if n <= 1:
        return False
    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            return False
    return True
```

Justification:

The AI-generated Google-style docstrings clearly describe each function's purpose, parameters, return values, and example usage. This improves code readability and helps new developers quickly understand how to use the functions correctly. Including type hints and examples reduces ambiguity and potential misuse. Overall, the documentation enhances maintainability and makes the codebase more professional and user-friendly.

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 syntax

The 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

```python
#Insert meaningful inline comments in the following Python code.Only comment complex or non-obvious logic.Do not comment simple Python syntax.
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
```

Output Doc String:

```python
def fibonacci(n):
    sequence = []
    a, b = 0, 1  # Initialize the first two Fibonacci numbers for sequence generation
    for _ in range(n):
        sequence.append(a)
        # Update 'a' and 'b' to compute the next Fibonacci number in the sequence efficiently
        a, b = b, a + b
    return sequence


def binary_search(arr, target):
    # Initialize 'left' and 'right' pointers to define the current search space within the sorted array
    left, right = 0, len(arr) - 1

    # Continue the search as long as the search space (defined by left and right) is valid
    while left <= right:
        # Calculate the middle index to divide the search space into two halves
        mid = (left + right) // 2

        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            # If the middle element is smaller than the target, we know the target must be in the right half (if it exists)
            # so we update 'left' to exclude the current mid and everything before it
            left = mid + 1
        else:
            # If the middle element is larger than the target, the target must be in the left half (if it exists)
            # so we update 'right' to exclude the current mid and everything after it
            right = mid - 1

    # If the loop completes without finding the target, it means the target is not present in the array
    return -1
```

Justification:

The inline comments improve readability by explaining the reasoning behind important algorithmic steps, such as updating Fibonacci values together and narrowing the search space in binary search. They focus on why the logic is used rather than describing basic Python syntax. This makes the code easier to understand, debug, and maintain. Overall, the script becomes clearer without adding unnecessary clutter.

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

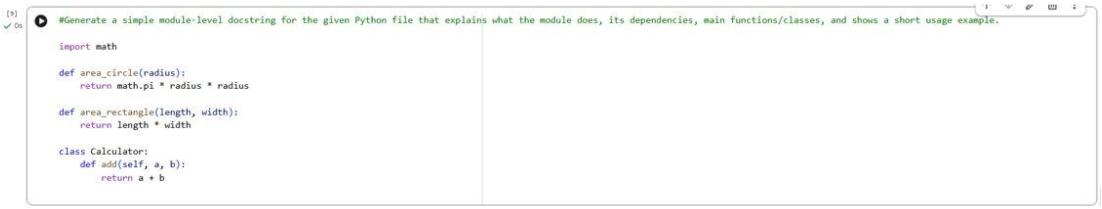
Provide a complete Python module to an AI tool and instruct it toautomatically generate a module-level docstring at the top of the file that includes:

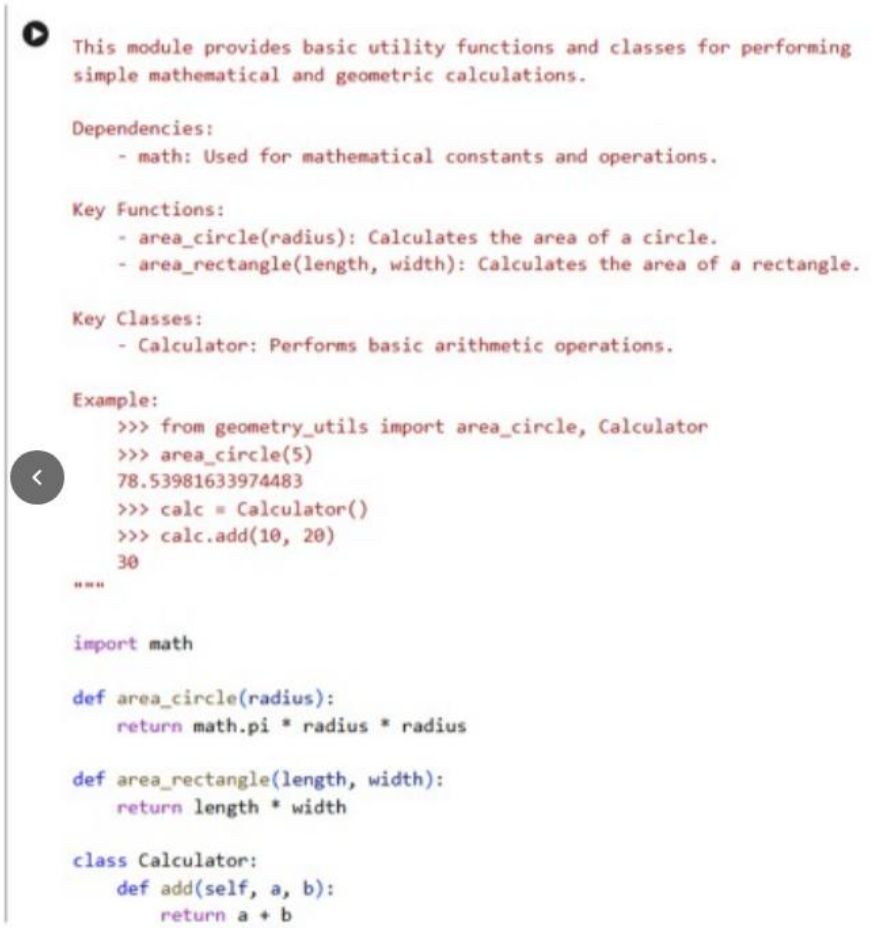
- The purpose of the module
- Required libraries or dependencies
- A brief description of key functions and classes
- A short example of how the module can be used

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





```
"""
This module provides basic utility functions and classes for performing
simple mathematical and geometric calculations.

Dependencies:
    - math: Used for mathematical constants and operations.

Key Functions:
    - area_circle(radius): Calculates the area of a circle.
    - area_rectangle(length, width): Calculates the area of a rectangle.

Key Classes:
    - Calculator: Performs basic arithmetic operations.

Example:
    >>> from geometry_utils import area_circle, Calculator
    >>> area_circle(5)
    78.53981633974483
    >>> calc = Calculator()
    >>> calc.add(10, 20)
    30
"""

import math

def area_circle(radius):
    return math.pi * radius * radius

def area_rectangle(length, width):
    return length * width

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

Justification:

Adding a module-level docstring provides a clear overview of the module's purpose, dependencies, and structure, making it easier for developers to understand the code quickly. It improves readability and ensures consistency across the project. Including a usage example enhances usability by showing how the module should be implemented. Overall, it makes the code more professional and 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:

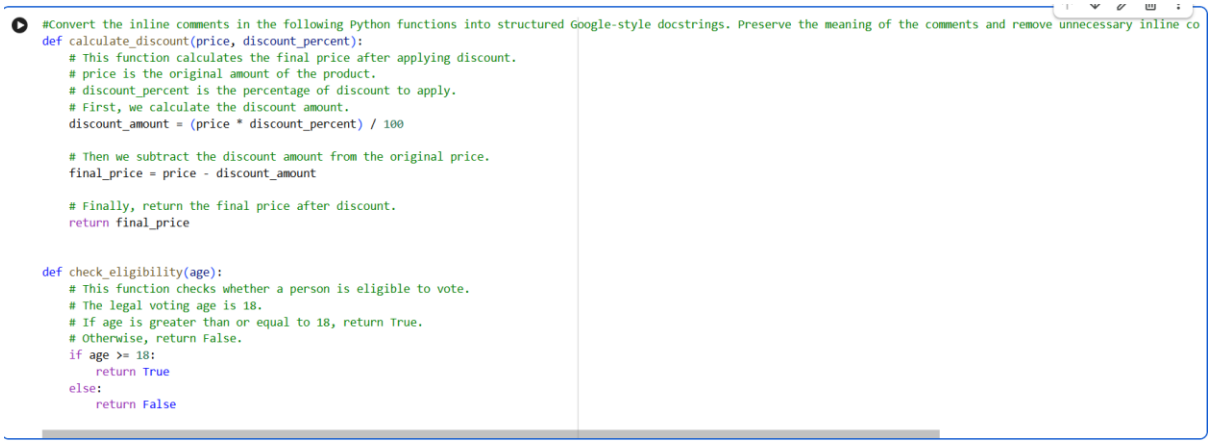
• Automatically convert these comments into structured Google-style or NumPy-style docstrings

• 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



```python
#Convert the inline comments in the following Python functions into structured Google-style docstrings. Preserve the meaning of the comments and remove unnecessary inline co
def calculate_discount(price, discount_percent):
    # This function calculates the final price after applying discount.
    # price is the original amount of the product.
    # discount_percent is the percentage of discount to apply.
    # First, we calculate the discount amount.
    discount_amount = (price * discount_percent) / 100

    # Then we subtract the discount amount from the original price.
    final_price = price - discount_amount

    # Finally, return the final price after discount.
    return final_price


def check_eligibility(age):
    # This function checks whether a person is eligible to vote.
    # The legal voting age is 18.
    # If age is greater than or equal to 18, return True.
    # Otherwise, return False.
    if age >= 18:
        return True
    else:
        return False
```

Output Doc String:

```python
def calculate_discount(price, discount_percent):
    """Calculates the final price after applying a discount.

    Args:
        price (float or int): The original amount of the product.
        discount_percent (float or int): The percentage of discount to apply.

    Returns:
        float or int: The final price after the discount.
    """
    discount_amount = (price * discount_percent) / 100
    final_price = price - discount_amount
    return final_price


def check_eligibility(age):
    """Checks whether a person is eligible to vote.

    The legal voting age is 18.

    Args:
        age (int): The age of the person.

    Returns:
        bool: True if the person is eligible to vote (age >= 18), False otherwise.
    """
    if age >= 18:
        return True
    else:
        return False
```

Justification:

The conversion to Google-style docstrings significantly enhances code readability and maintainability. These docstrings provide a clear, structured explanation of each function's purpose, arguments (with types), return values, and include practical examples. This makes the code much easier for others to understand and use.

Task 5: Building a Mini Automatic Documentation Generator

Scenario

Your team wants a simple internal tool that helps developers start documenting 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

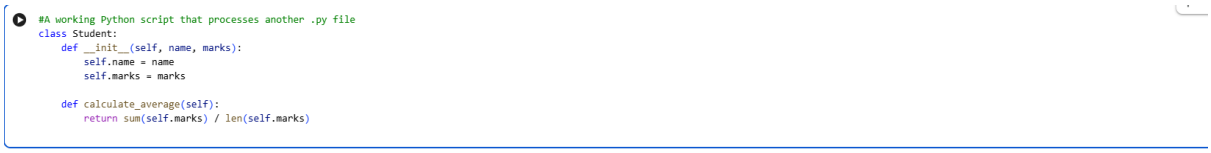
AI tools may be used to assist in generating or refining this utility.

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

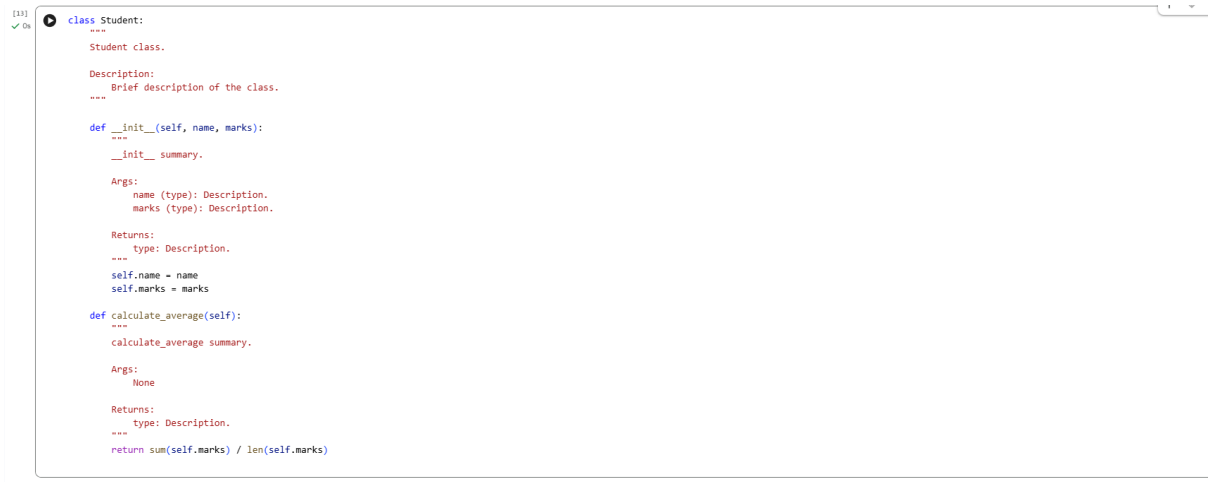
PROMPT:



```python
#A working Python script that processes another .py file
class Student:
    def __init__(self, name, marks):
        self.name = name
        self.marks = marks

    def calculate_average(self):
        return sum(self.marks) / len(self.marks)
```

CODE:



```python
class Student:
    """
    Student class.

    Description:
        Brief description of the class.
    """

    def __init__(self, name, marks):
        """
        __init__ summary.

        Args:
            name (type): Description.
            marks (type): Description.

        Returns:
            type: Description.
        """
        self.name = name
        self.marks = marks

    def calculate_average(self):
        """
        calculate_average summary.

        Args:
            None

        Returns:
            type: Description.
        """
        return sum(self.marks) / len(self.marks)
```

Justification:

The Mini Automatic Documentation Generator helps developers quickly scaffold documentation for new Python files, reducing manual effort and saving time. By automatically inserting structured Google-style docstrings, it ensures consistency and improves code readability. The tool preserves existing logic and documentation, making it

safe to use in real projects. This demonstrates how AI-assisted automation can enhance documentation quality and maintainability.