ASSIGNMENT-9.4

Name:N.Soumya

H NO:2303A51869

Batch-13

<u>Lab 9 – Documentation Generation: Automatic Documentation and Code Comments</u>

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

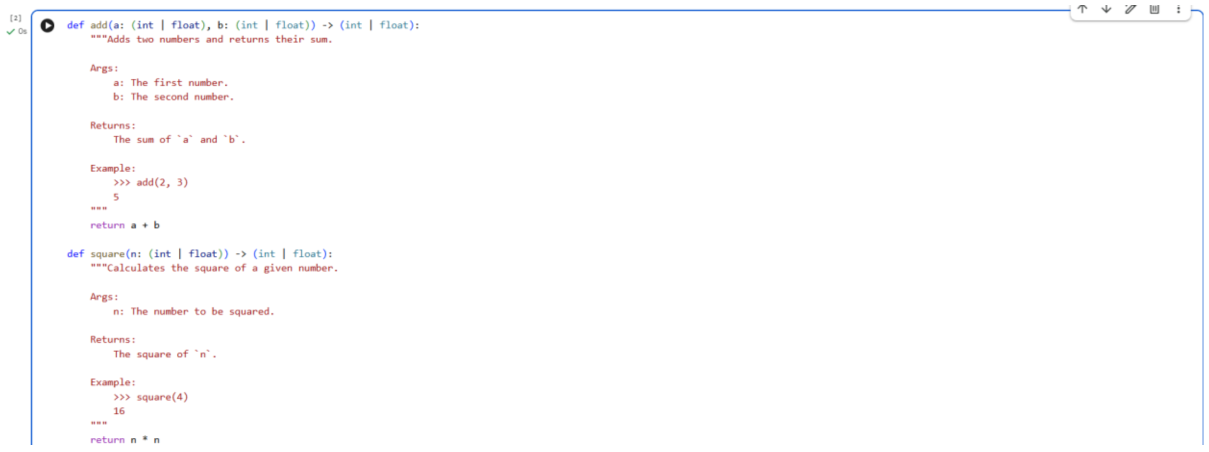
• Docstrings that clearly explain function behavior and usage

• Improved readability and usability of the codebase

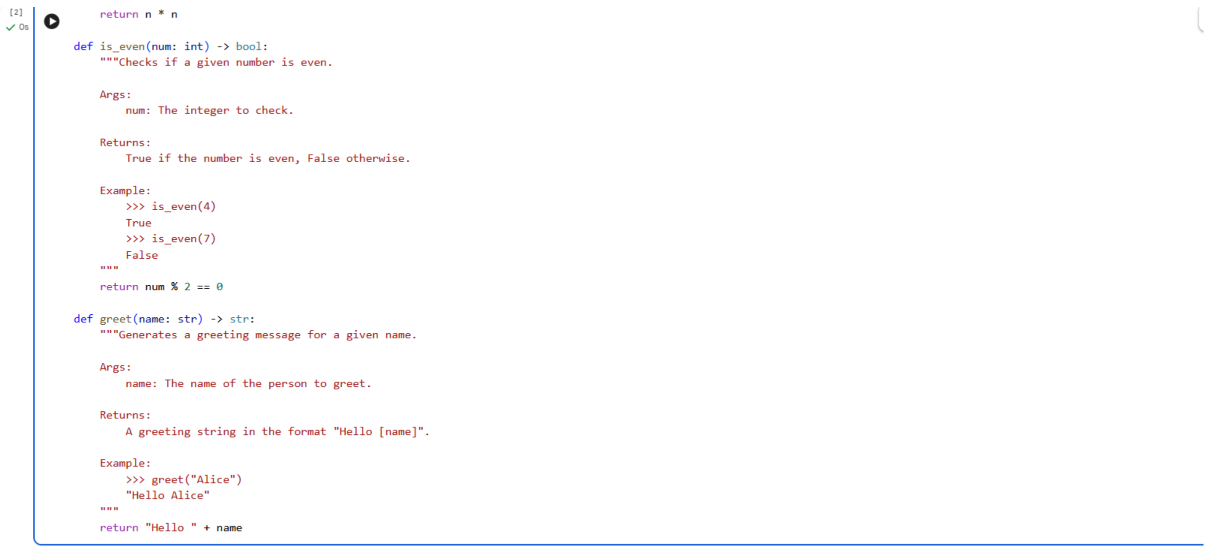
Prompt:

```
#Generate Google-style docstrings for the following Python functions.
def add(a, b):
    return a + b


def square(n):
    return n * n


def is_even(num):
    return num % 2 == 0


def greet(name):
    return "Hello " + name
```

Output Docstring



```
def add(a: (int | float), b: (int | float)) -> (int | float):
    """Adds two numbers and returns their sum.

    Args:
        a: The first number.
        b: The second number.

    Returns:
        The sum of `a` and `b`.

    Example:
        >>> add(2, 3)
        5
    """
    return a + b

def square(n: (int | float)) -> (int | float):
    """Calculates the square of a given number.

    Args:
        n: The number to be squared.

    Returns:
        The square of `n`.

    Example:
        >>> square(4)
        16
    """
    return n * n
```



```
    return n * n

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

    Args:
        num: The integer to check.

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

    Example:
        >>> is_even(4)
        True
        >>> is_even(7)
        False
    """
    return num % 2 == 0

def greet(name: str) -> str:
    """Generates a greeting message for a given name.

    Args:
        name: The name of the person to greet.

    Returns:
        A greeting string in the format "Hello [name]".

    Example:
        >>> greet("Alice")
        "Hello Alice"
    """
    return "Hello " + name
```

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

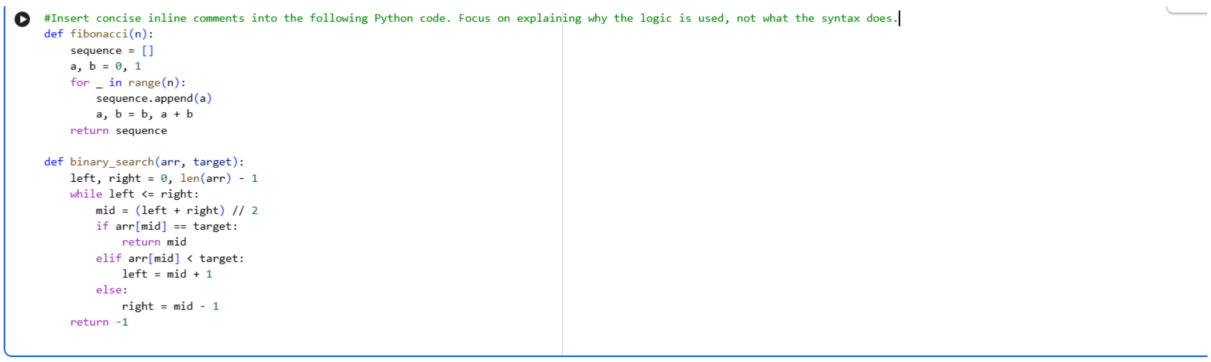
• A Python script with concise, meaningful inline comments

• Comments that explain why the logic exists, not what Python

syntax does

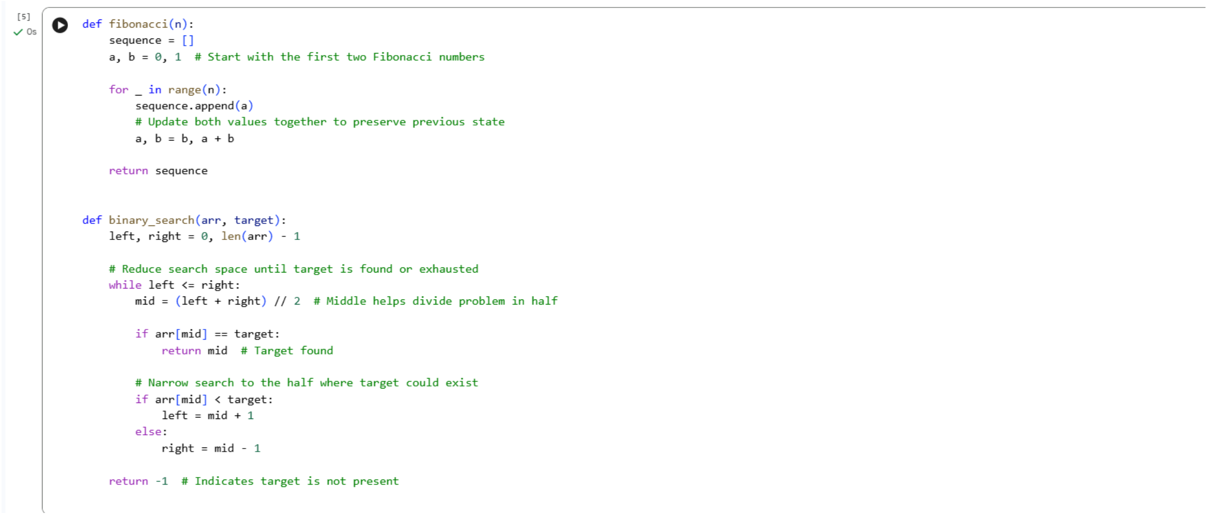
• Noticeable improvement in code readability

Prompt:



```python
#Insert concise inline comments into the following Python code. Focus on explaining why the logic is used, not what the syntax does.
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 Docstring



```python
def fibonacci(n):
    sequence = []
    a, b = 0, 1  # Start with the first two Fibonacci numbers

    for _ in range(n):
        sequence.append(a)
        # Update both values together to preserve previous state
        a, b = b, a + b

    return sequence


def binary_search(arr, target):
    left, right = 0, len(arr) - 1

    # Reduce search space until target is found or exhausted
    while left <= right:
        mid = (left + right) // 2  # Middle helps divide problem in half

        if arr[mid] == target:
            return mid  # Target found

        # Narrow search to the half where target could exist
        if arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1

    return -1  # Indicates target is not present
```

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 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.

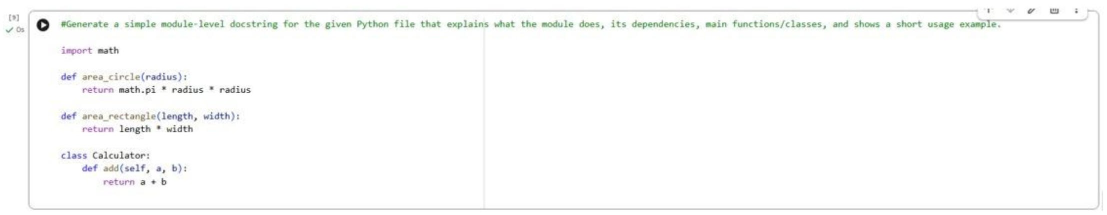
Expected Outcome

• A well-written multi-line module-level docstring

• Clear overview of what the module does and how to use it

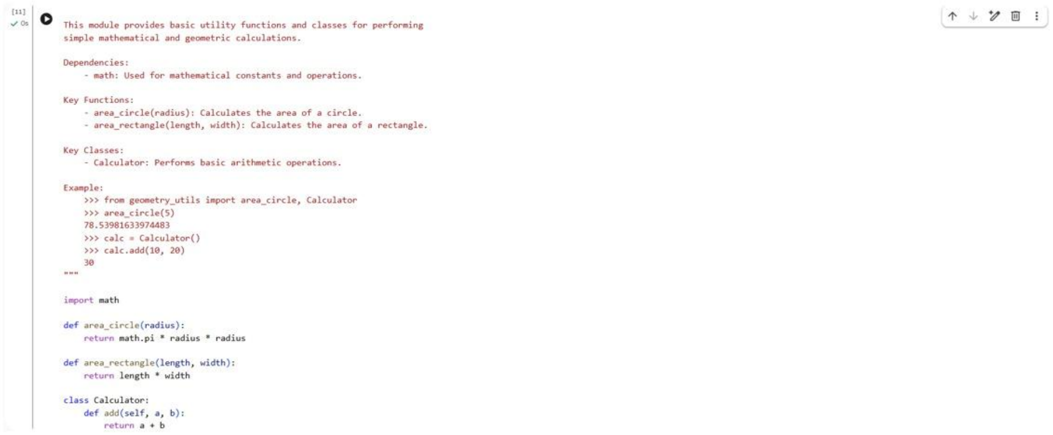
• Documentation suitable for real-world projects or repositories

Prompt:



```
[9]
✓ 0s
#Generate a simple module-level docstring for the given Python file that explains what the module does, its dependencies, main functions/classes, and shows a short usage example.

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
```

OUTPUT:



```
[11]
✓ 0s
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

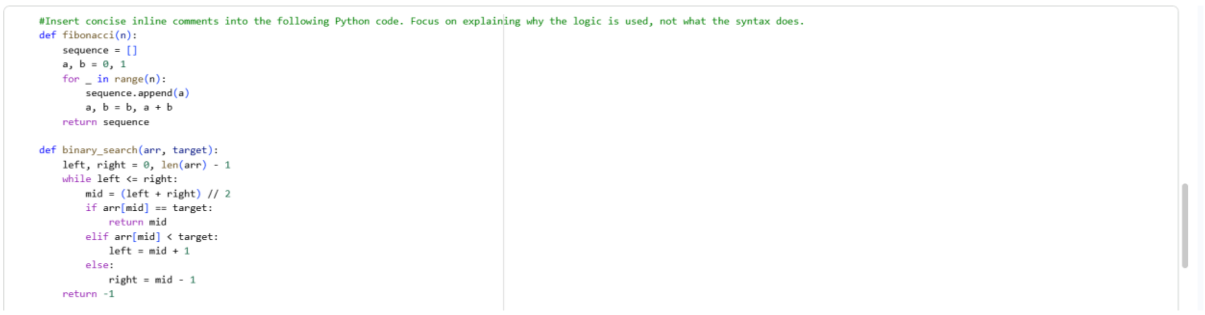
Expected Outcome

• Functions with clean, standardized docstrings

• Reduced clutter inside function bodies

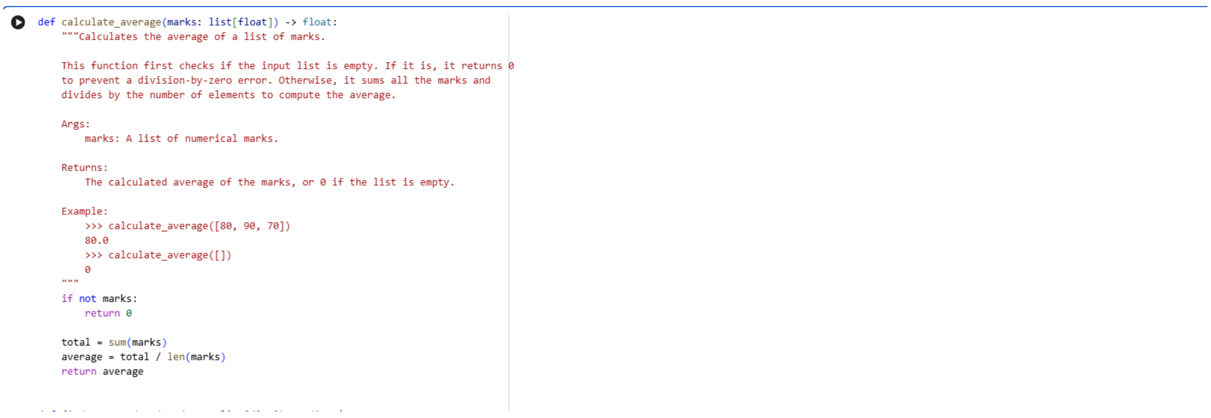
• Improved consistency across the codebase

PROMPT:

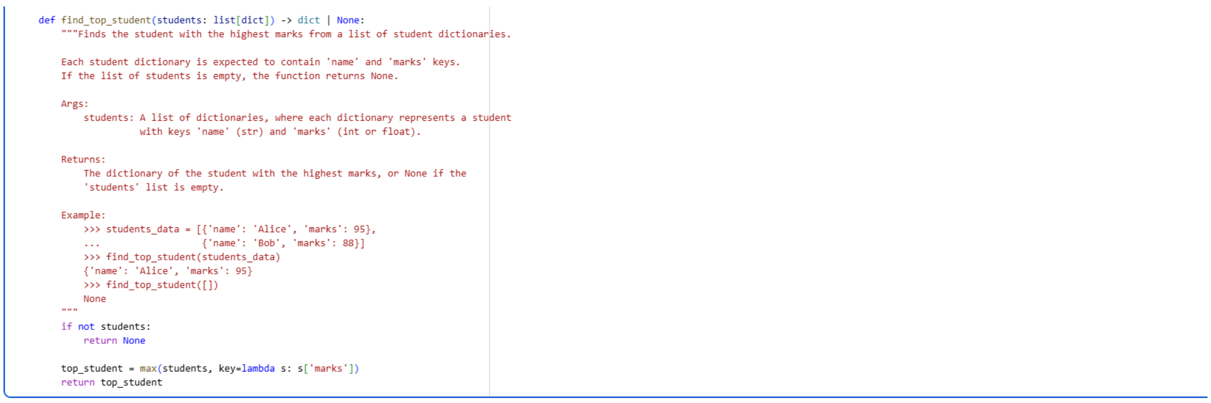


```python
#Insert concise inline comments into the following Python code. Focus on explaining why the logic is used, not what the syntax does.
def fibonacci(n):
    sequence = []
    a, b = 0, 1
    for _ in range(n):
        sequence.append(a)
        a, b = b, a + b
    return sequence

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
```

GOOGLE-STYLED DOCSTRING OUTPUT:



```python
def calculate_average(marks: list[float]) -> float:
    """Calculates the average of a list of marks.

    This function first checks if the input list is empty. If it is, it returns 0
    to prevent a division-by-zero error. Otherwise, it sums all the marks and
    divides by the number of elements to compute the average.

    Args:
        marks: A list of numerical marks.

    Returns:
        The calculated average of the marks, or 0 if the list is empty.

    Example:
        >>> calculate_average([80, 90, 70])
        80.0
        >>> calculate_average([])
        0
    """
    if not marks:
        return 0

    total = sum(marks)
    average = total / len(marks)
    return average
```



```python
def find_top_student(students: list[dict]) -> dict | None:
    """Finds the student with the highest marks from a list of student dictionaries.

    Each student dictionary is expected to contain 'name' and 'marks' keys.
    If the list of students is empty, the function returns None.

    Args:
        students: A list of dictionaries, where each dictionary represents a student
                  with keys 'name' (str) and 'marks' (int or float).

    Returns:
        The dictionary of the student with the highest marks, or None if the
        'students' list is empty.

    Example:
        >>> students_data = [{'name': 'Alice', 'marks': 95},
        ...                  {'name': 'Bob', 'marks': 88}]
        >>> find_top_student(students_data)
        {'name': 'Alice', 'marks': 95}
        >>> find_top_student([])
        None
    """
    if not students:
        return None

    top_student = max(students, key=lambda s: s['marks'])
    return top_student
```

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.

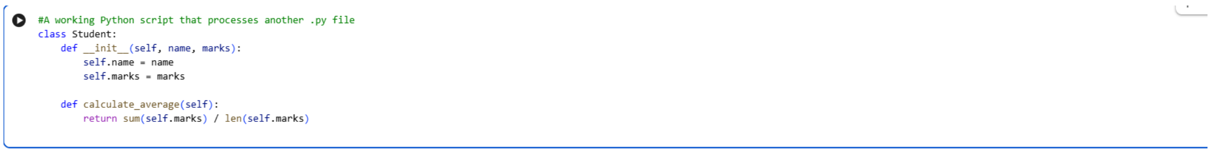
Expected Outcome

• A working Python script that processes another .py file

• Automatically inserted placeholder docstrings

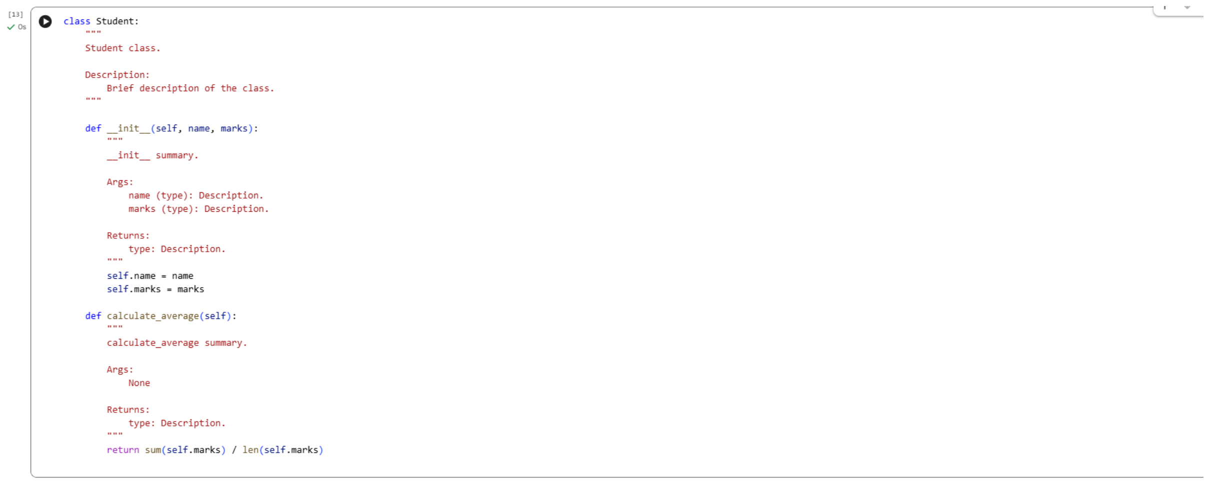
• Clear demonstration of how AI can assist in documentation automation

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)
```

OUTPUT



```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**.**