<u>T. VIKAS</u>

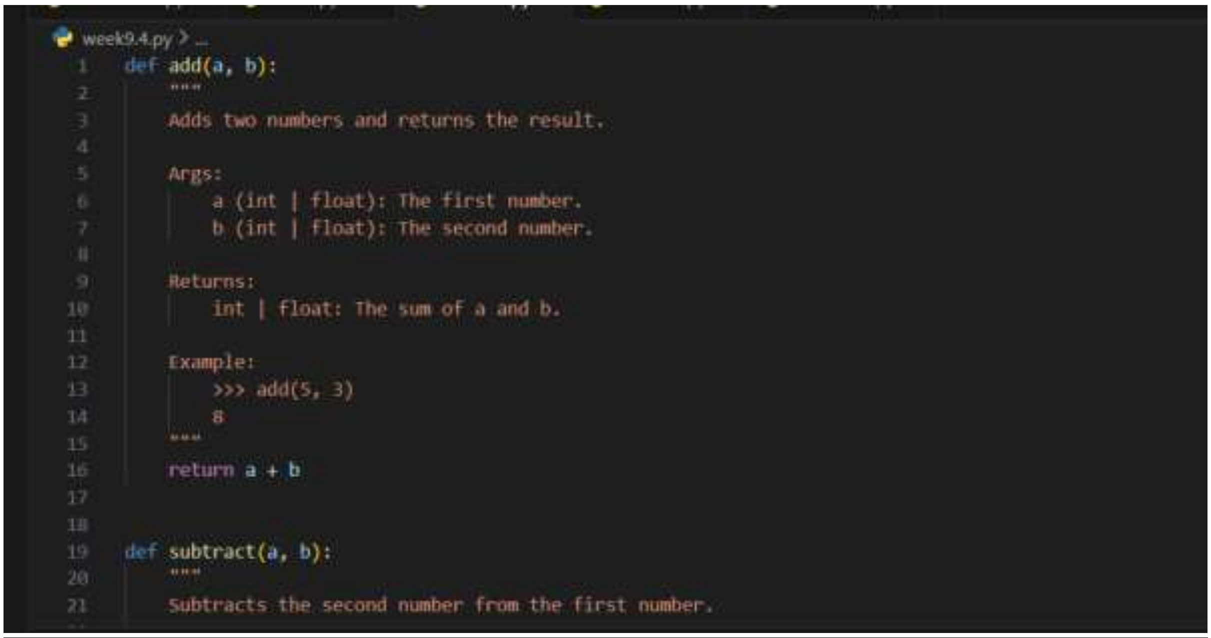
<u>2303A52281</u>

<u>BATCH 43</u>

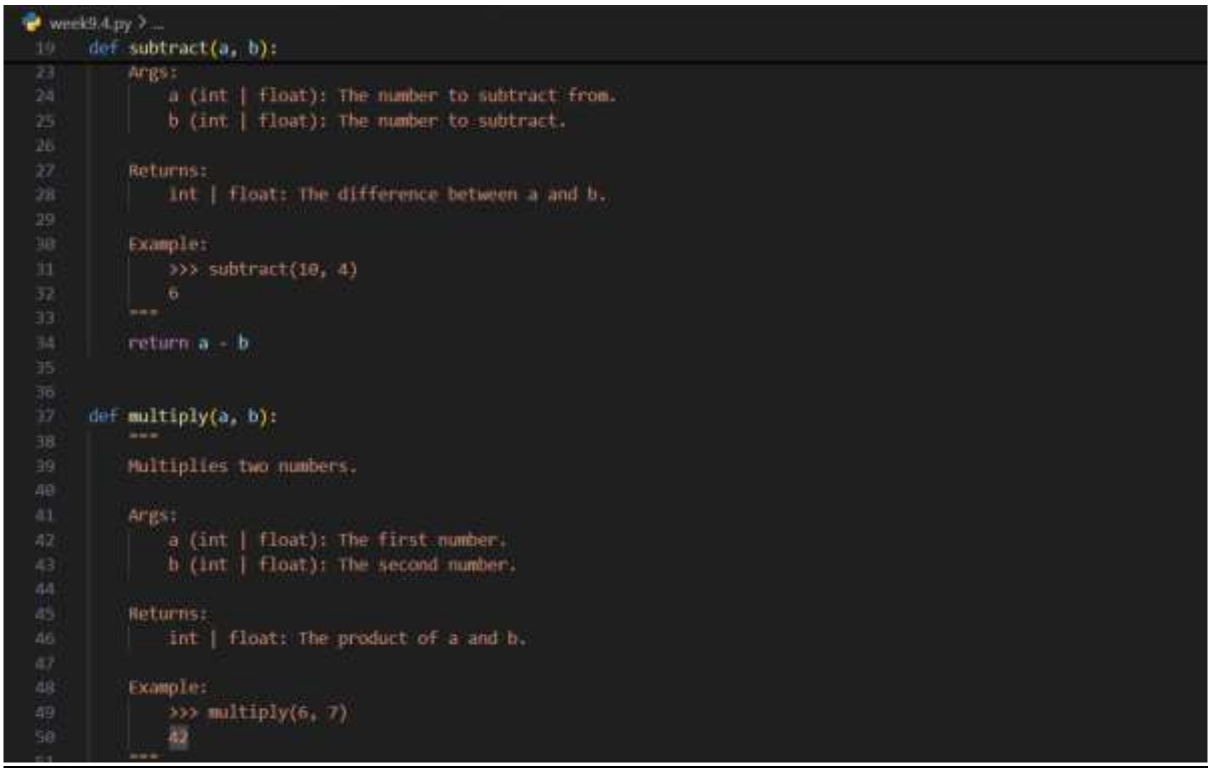
<u>WEEK 9.4</u>

<u>TASK 1</u>: Auto-Generating Function Documentation in a Shared Codebase

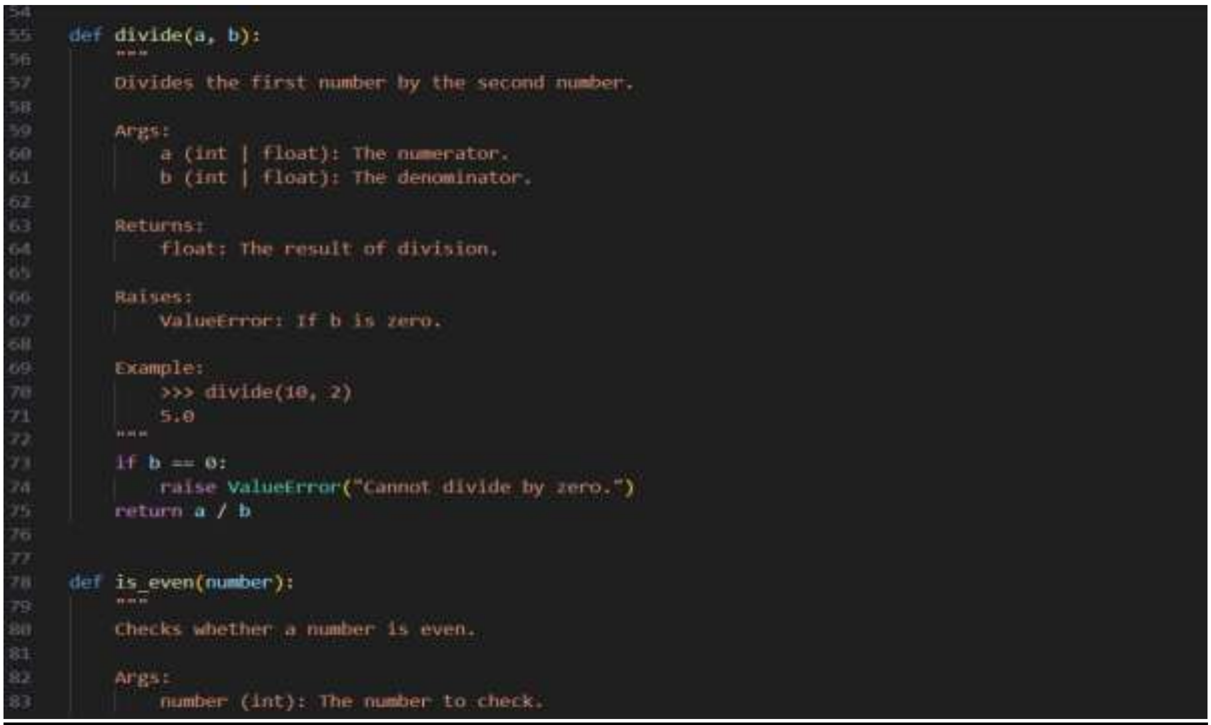
CODE:



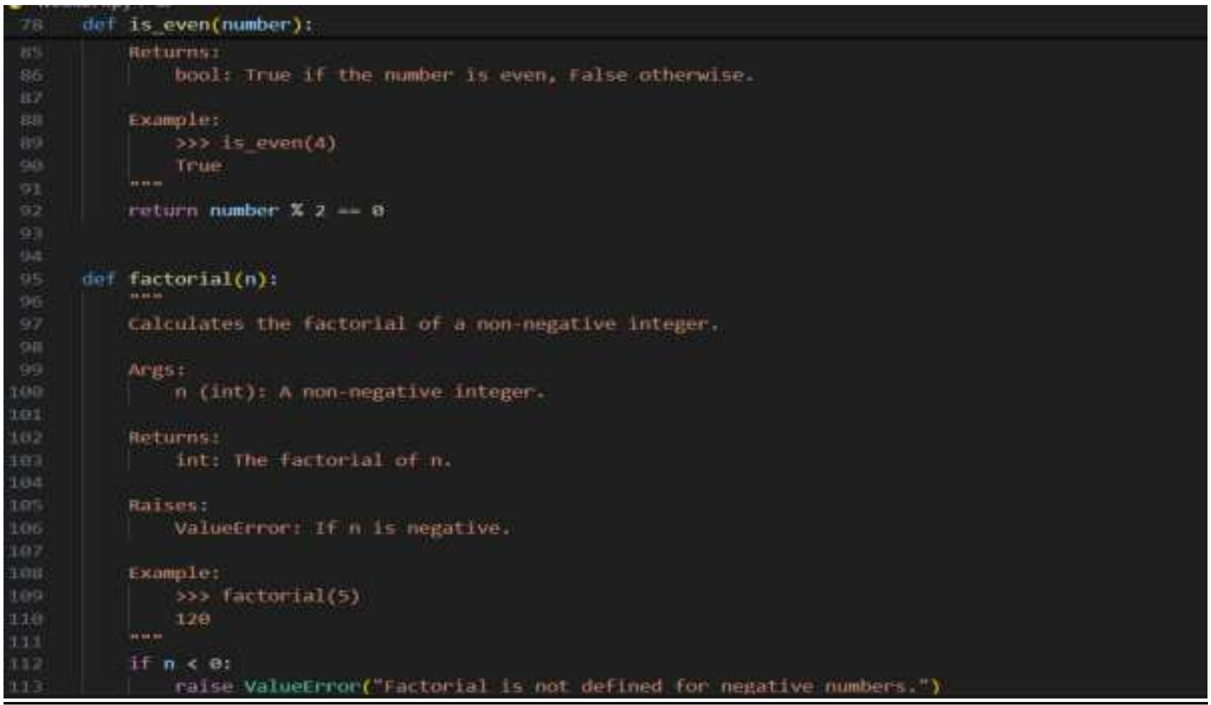
```python
def add(a, b):
    """
    Adds two numbers and returns the result.

    Args:
        a (int | float): The first number.
        b (int | float): The second number.

    Returns:
        int | float: The sum of a and b.

    Example:
        >>> add(5, 3)
        8
    """
    return a + b


def subtract(a, b):
    """
    Subtracts the second number from the first number.
```



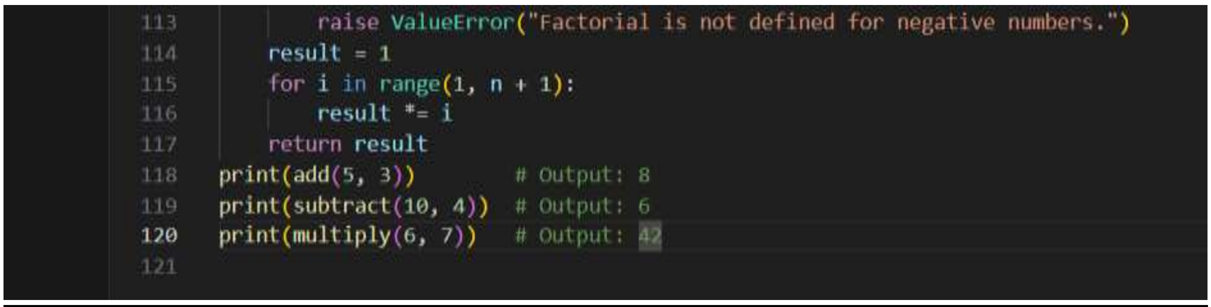
```python
def subtract(a, b):
    Args:
        a (int | float): The number to subtract from.
        b (int | float): The number to subtract.

    Returns:
        int | float: The difference between a and b.

    Example:
        >>> subtract(10, 4)
        6
    """
    return a - b


def multiply(a, b):
    """
    Multiplies two numbers.

    Args:
        a (int | float): The first number.
        b (int | float): The second number.

    Returns:
        int | float: The product of a and b.

    Example:
        >>> multiply(6, 7)
        42
    """
```



```python

def divide(a, b):
    """
    Divides the first number by the second number.

    Args:
        a (int | float): The numerator.
        b (int | float): The denominator.

    Returns:
        float: The result of division.

    Raises:
        ValueError: If b is zero.

    Example:
        >>> divide(10, 2)
        5.0
    """
    if b == 0:
        raise ValueError("Cannot divide by zero.")
    return a / b


def is_even(number):
    """
    Checks whether a number is even.

    Args:
        number (int): The number to check.
```

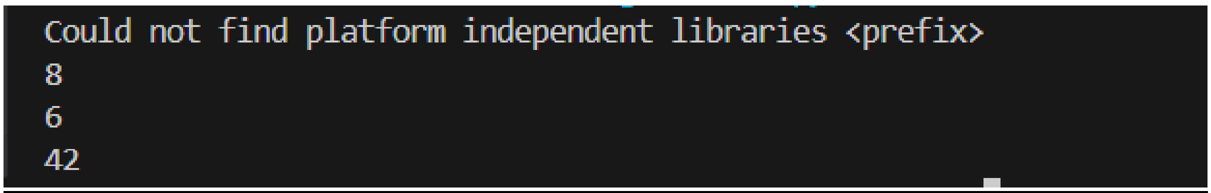


```python
def is_even(number):
    Returns:
        bool: True if the number is even, False otherwise.

    Example:
        >>> is_even(4)
        True
    """
    return number % 2 == 0


def factorial(n):
    """
    Calculates the factorial of a non-negative integer.

    Args:
        n (int): A non-negative integer.

    Returns:
        int: The factorial of n.

    Raises:
        ValueError: If n is negative.

    Example:
        >>> factorial(5)
        120
    """
    if n < 0:
        raise ValueError("Factorial is not defined for negative numbers.")
```



```python
        raise ValueError("Factorial is not defined for negative numbers.")
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
print(add(5, 3))        # Output: 8
print(subtract(10, 4))  # Output: 6
print(multiply(6, 7))   # Output: 42

```

Output:



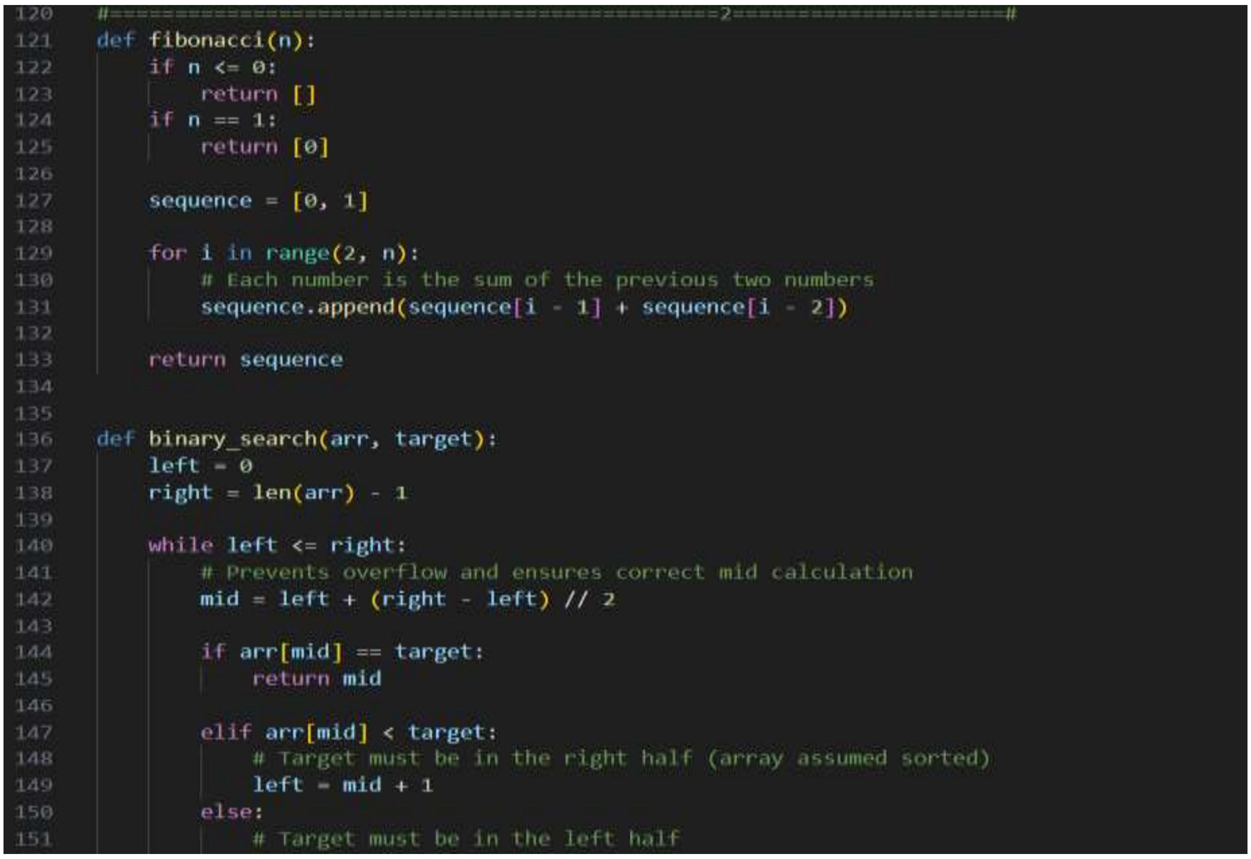
```
Could not find platform independent libraries <prefix>
8
6
42
```

Summary:

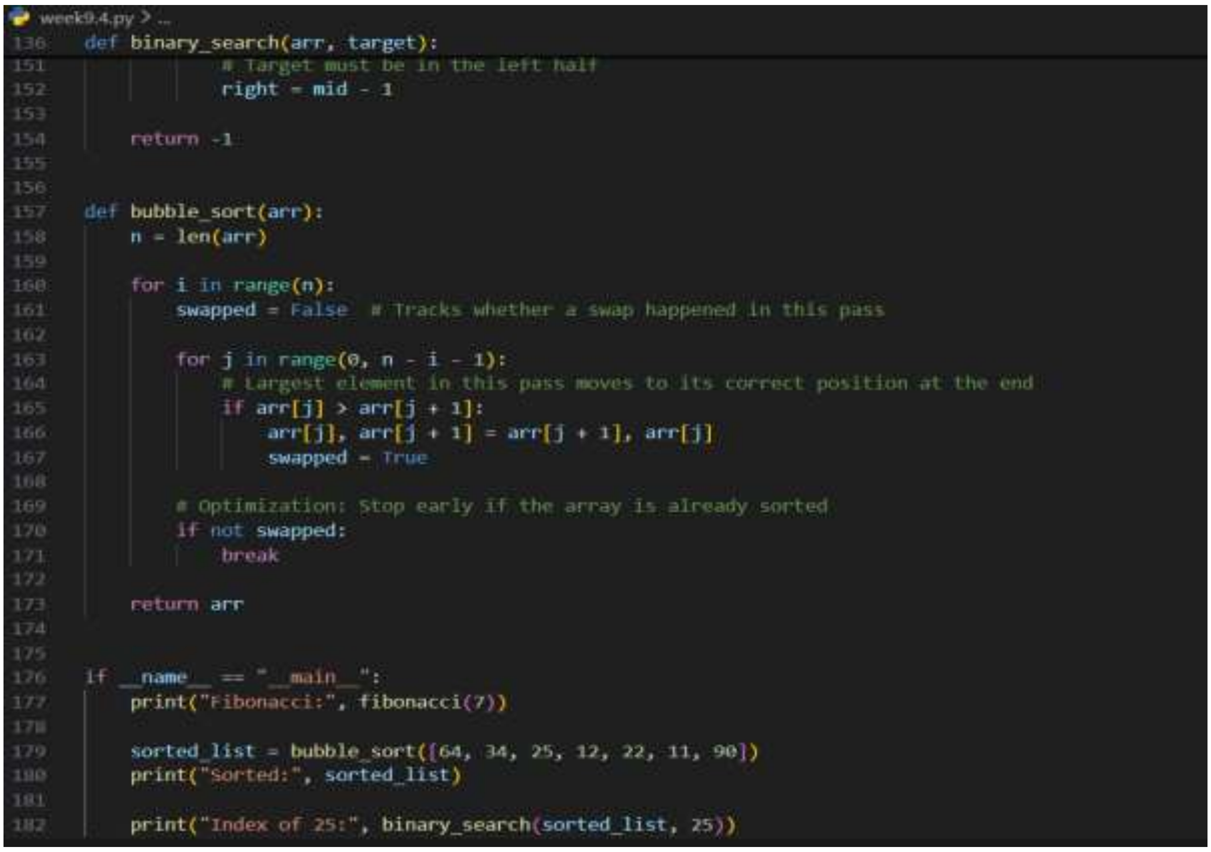
This task demonstrates how AI-assisted tools can be used to automatically generate well-structured Google-style docstrings for an existing Python codebase that lacks documentation. By adding clear function descriptions, parameter types, return values, exceptions, and example usages, the readability and maintainability of the code significantly improve. The experiment with zero-shot and context-based prompting shows that more detailed and guided prompts produce higher-quality documentation. Overall, AI-driven documentation enhances team collaboration, simplifies onboarding for new developers, and ensures consistent documentation standards across the shared codebase.

TASK 2: Enhancing Readability Through AI-Generated InlineComments

CODE:



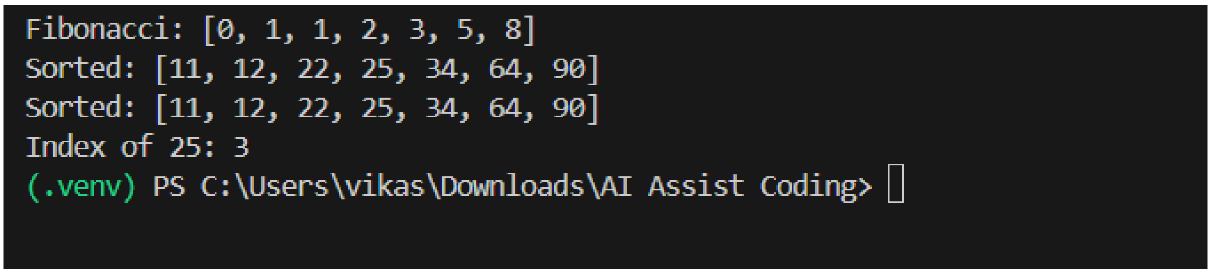
```python
#===============================================2======================#
def fibonacci(n):
    if n <= 0:
        return []
    if n == 1:
        return [0]

    sequence = [0, 1]

    for i in range(2, n):
        # Each number is the sum of the previous two numbers
        sequence.append(sequence[i - 1] + sequence[i - 2])

    return sequence


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

    while left <= right:
        # Prevents overflow and ensures correct mid calculation
        mid = left + (right - left) // 2

        if arr[mid] == target:
            return mid

        elif arr[mid] < target:
            # Target must be in the right half (array assumed sorted)
            left = mid + 1
        else:
            # Target must be in the left half
```



week9.4.py > ...

```python
def binary_search(arr, target):
            # Target must be in the left half
            right = mid - 1

    return -1


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

    for i in range(n):
        swapped = False  # Tracks whether a swap happened in this pass

        for j in range(0, n - i - 1):
            # Largest element in this pass moves to its correct position at the end
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True

        # Optimization: Stop early if the array is already sorted
        if not swapped:
            break

    return arr


if __name__ == "__main__":
    print("Fibonacci:", fibonacci(7))

    sorted_list = bubble_sort([64, 34, 25, 12, 22, 11, 90])
    print("Sorted:", sorted_list)

    print("Index of 25:", binary_search(sorted_list, 25))
```

Output:



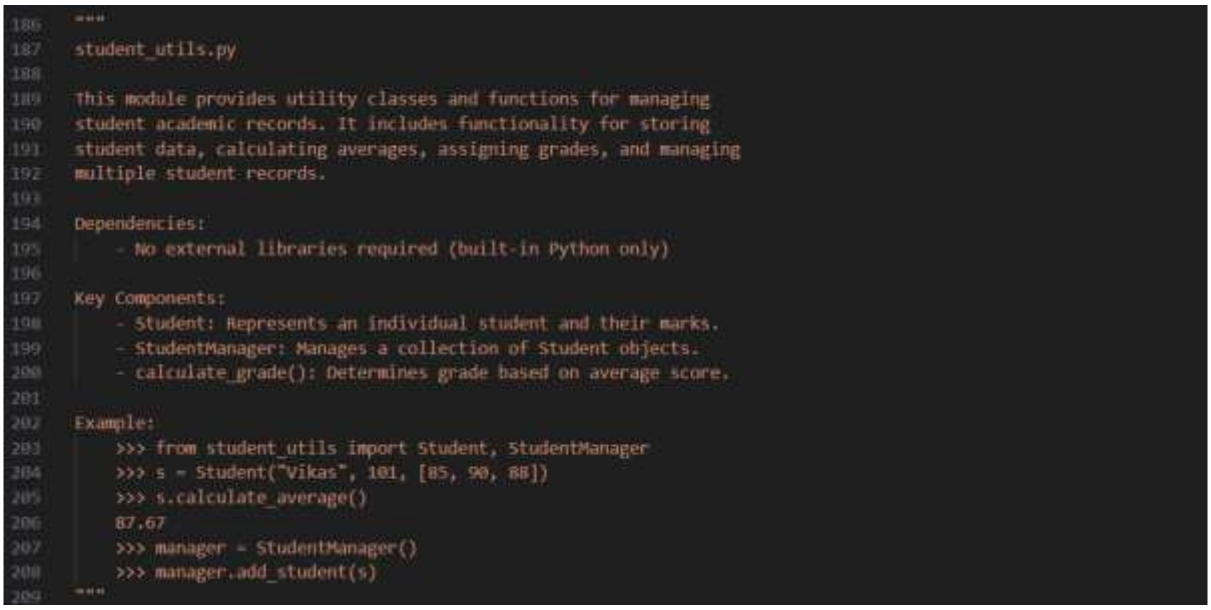
```
Fibonacci: [0, 1, 1, 2, 3, 5, 8]
Sorted: [11, 12, 22, 25, 34, 64, 90]
Sorted: [11, 12, 22, 25, 34, 64, 90]
Index of 25: 3
(.venv) PS C:\Users\vikas\Downloads\AI Assist Coding>
```

Summary:

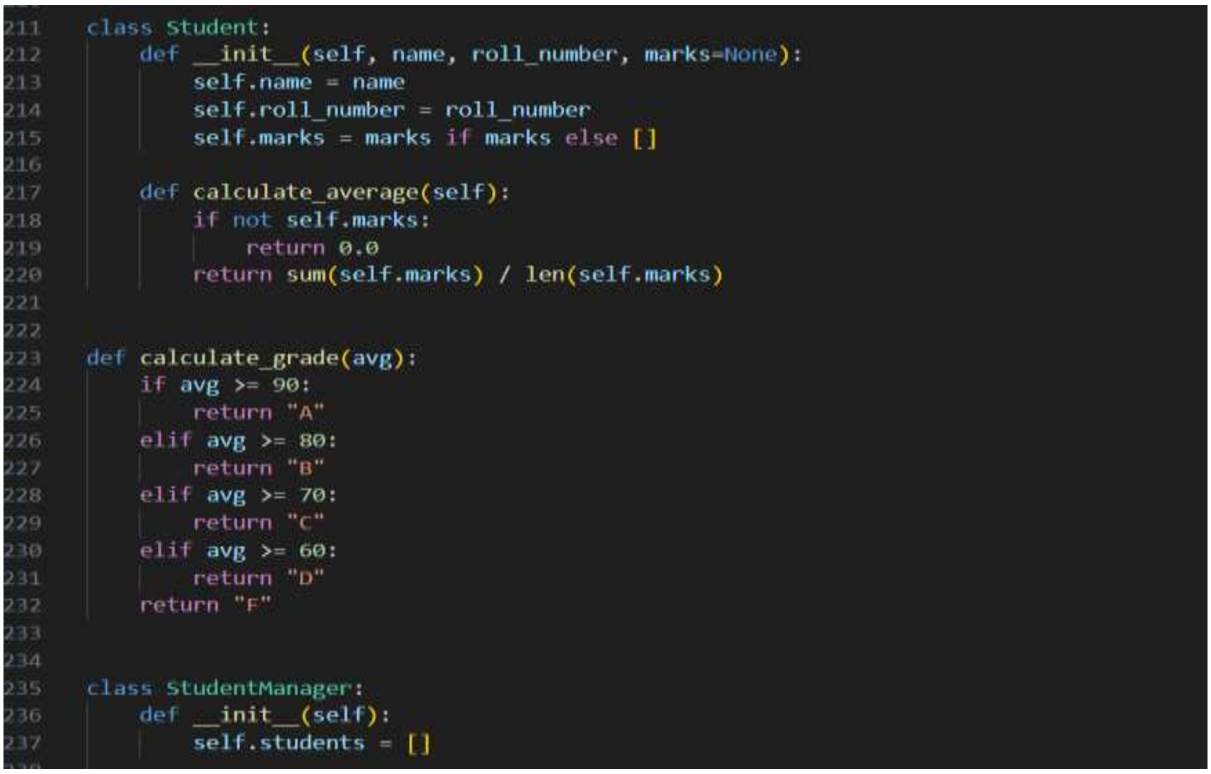
This task demonstrates how AI-generated inline comments can significantly improve code readability without overwhelming the script with unnecessary explanations. By adding concise comments only to complex or non-obvious logic—such as optimization strategies, algorithmic decisions, and boundary handling—the clarity of the program is enhanced while maintaining clean structure. The resulting code is easier to understand, debug, and maintain, making it more accessible for future developers working in a shared codebase.

<u>TASK 3</u>: Generating Module-Level Documentation for a Python Package

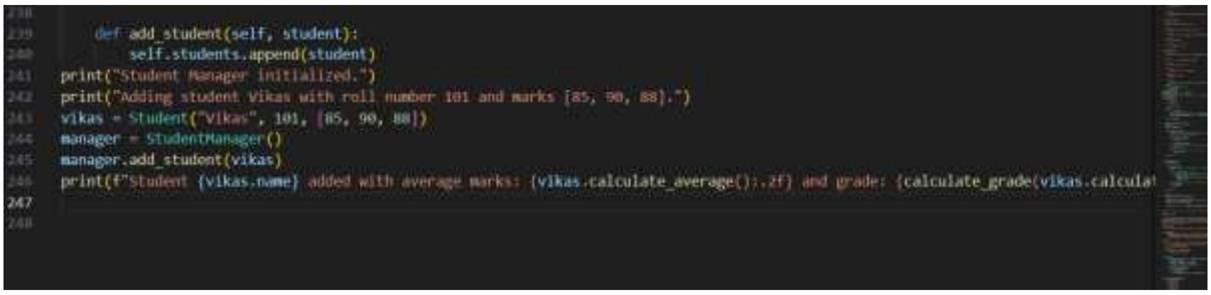
Code:



```
"""
student_utils.py

This module provides utility classes and functions for managing
student academic records. It includes functionality for storing
student data, calculating averages, assigning grades, and managing
multiple student records.

Dependencies:
    - No external libraries required (built-in Python only)

Key Components:
    - Student: Represents an individual student and their marks.
    - StudentManager: Manages a collection of Student objects.
    - calculate_grade(): Determines grade based on average score.

Example:
    >>> from student_utils import Student, StudentManager
    >>> s = Student("Vikas", 101, [85, 90, 88])
    >>> s.calculate_average()
    87.67
    >>> manager = StudentManager()
    >>> manager.add_student(s)
"""
```

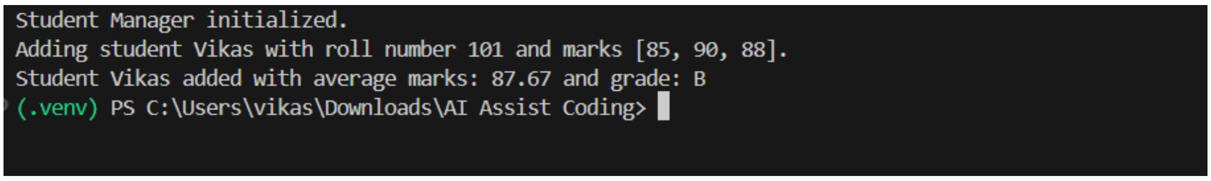


```
class Student:
    def __init__(self, name, roll_number, marks=None):
        self.name = name
        self.roll_number = roll_number
        self.marks = marks if marks else []

    def calculate_average(self):
        if not self.marks:
            return 0.0
        return sum(self.marks) / len(self.marks)


def calculate_grade(avg):
    if avg >= 90:
        return "A"
    elif avg >= 80:
        return "B"
    elif avg >= 70:
        return "C"
    elif avg >= 60:
        return "D"
    return "F"


class StudentManager:
    def __init__(self):
        self.students = []
```



```

    def add_student(self, student):
        self.students.append(student)
print("Student Manager initialized.")
print("Adding student Vikas with roll number 101 and marks [85, 90, 88].")
vikas = Student("Vikas", 101, [85, 90, 88])
manager = StudentManager()
manager.add_student(vikas)
print(f"Student {vikas.name} added with average marks: {vikas.calculate_average():.2f} and grade: {calculate_grade(vikas.calcula
```

Output:



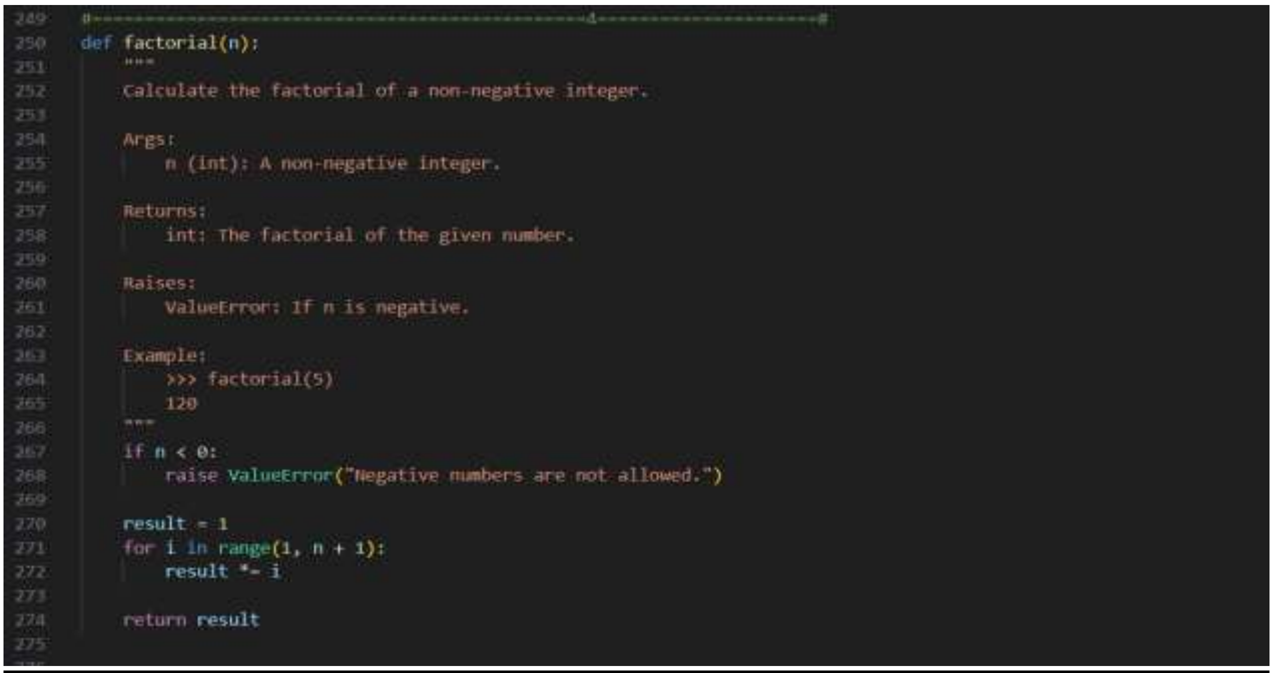
```
Student Manager initialized.
Adding student Vikas with roll number 101 and marks [85, 90, 88].
Student Vikas added with average marks: 87.67 and grade: B
(.venv) PS C:\Users\vikas\Downloads\AI Assist Coding>
```

Summary:

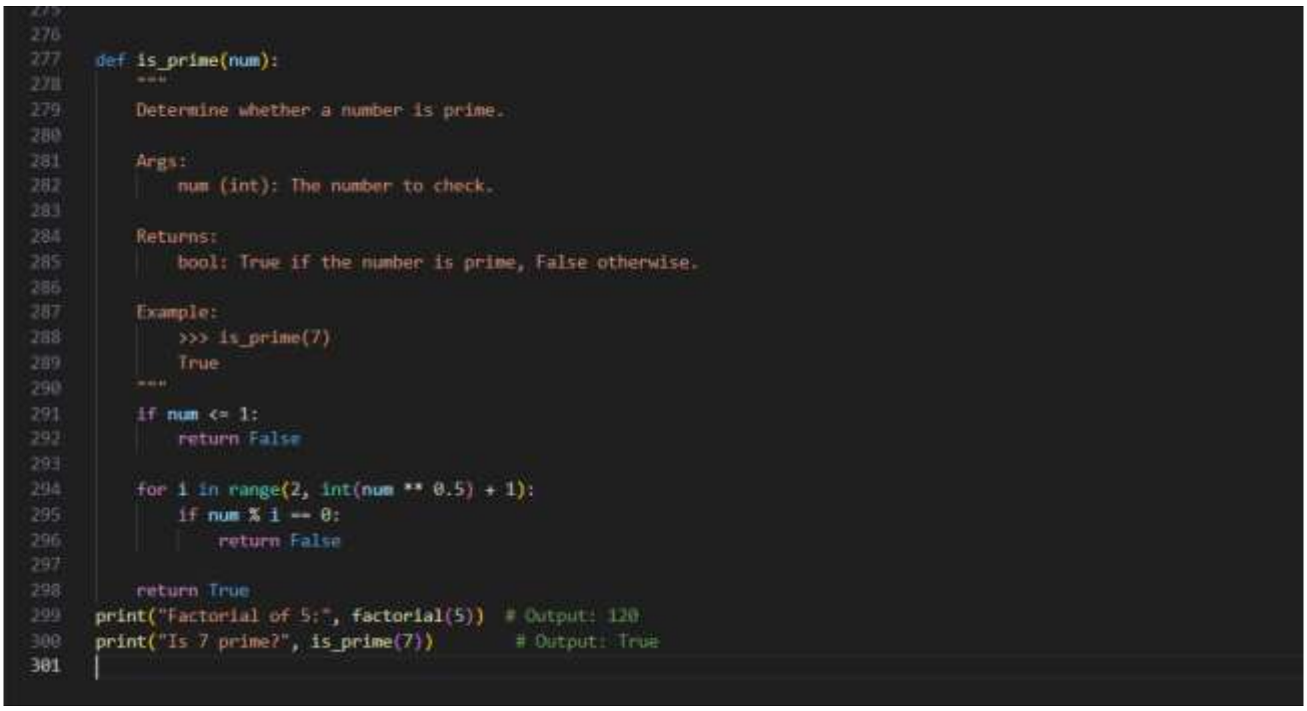
This task demonstrates how AI can generate a professional module-level docstring that clearly describes the purpose, structure, dependencies, and usage of a Python module. The documentation provides an immediate overview for developers, improving discoverability and usability. Such structured documentation is suitable for real-world repositories and internal team sharing.

TASK 4: Converting Developer Comments into Structured Docstrings

CODE:

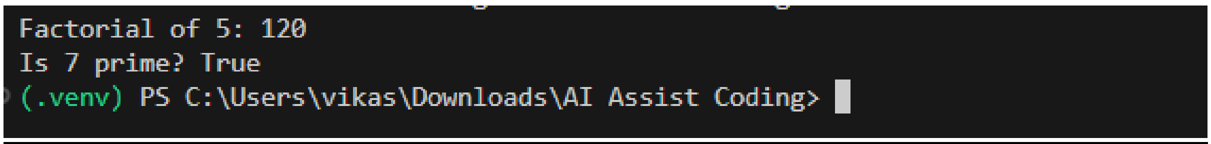


```python
#-------------------------------------------------4---------------------#
def factorial(n):
    """
    Calculate the factorial of a non-negative integer.

    Args:
        n (int): A non-negative integer.

    Returns:
        int: The factorial of the given number.

    Raises:
        ValueError: If n is negative.

    Example:
        >>> factorial(5)
        120
    """
    if n < 0:
        raise ValueError("Negative numbers are not allowed.")

    result = 1
    for i in range(1, n + 1):
        result *= i

    return result

```



```
def is_prime(num):
    """
    Determine whether a number is prime.

    Args:
        num (int): The number to check.

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

    Example:
        >>> is_prime(7)
        True
    """
    if num <= 1:
        return False

    for i in range(2, int(num ** 0.5) + 1):
        if num % i == 0:
            return False

    return True
print("Factorial of 5:", factorial(5))  # Output: 120
print("Is 7 prime?", is_prime(7))        # Output: True
```

Output:



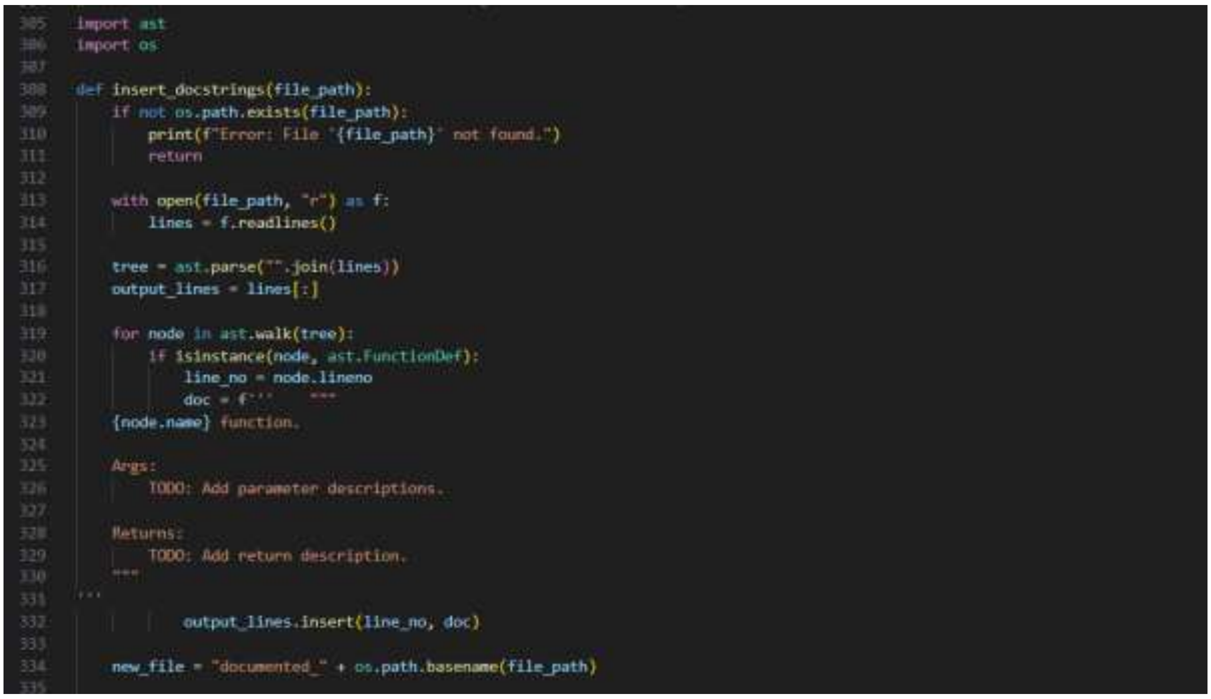
```
Factorial of 5: 120
Is 7 prime? True
(.venv) PS C:\Users\vikas\Downloads\AI Assist Coding>
```

Summary:

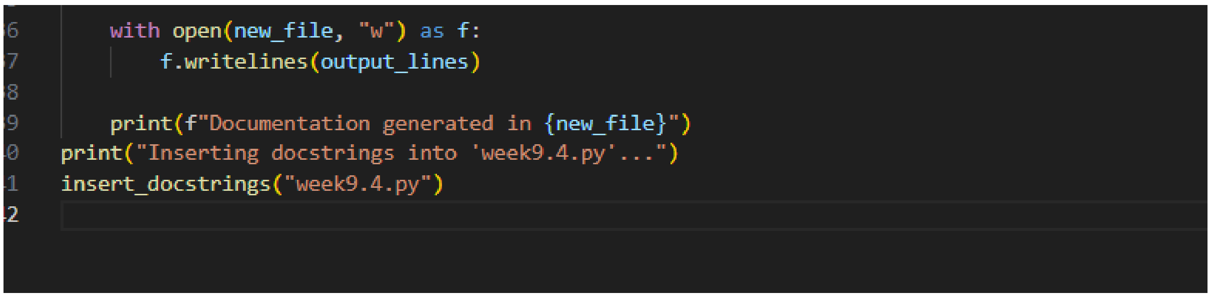
This task shows how AI can convert lengthy inline comments into structured, standardized Google-style docstrings. The result reduces clutter inside function bodies while preserving original intent. The code becomes cleaner, more professional, and consistent across the project, improving maintainability and readability.

Task5: Mini Automatic Documentation Generator

Code:

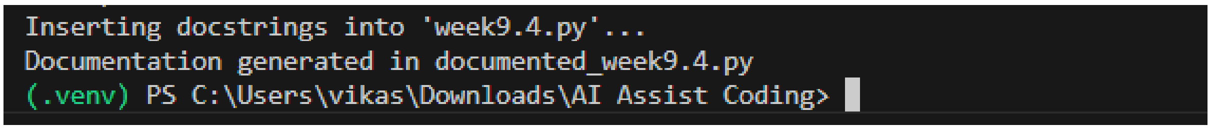


```python
import ast
import os

def insert_docstrings(file_path):
    if not os.path.exists(file_path):
        print(f"Error: File '{file_path}' not found.")
        return

    with open(file_path, "r") as f:
        lines = f.readlines()

    tree = ast.parse("".join(lines))
    output_lines = lines[:]

    for node in ast.walk(tree):
        if isinstance(node, ast.FunctionDef):
            line_no = node.lineno
            doc = f'''    """
    {node.name} function.

    Args:
        TODO: Add parameter descriptions.

    Returns:
        TODO: Add return description.
    """
'''
            output_lines.insert(line_no, doc)

    new_file = "documented_" + os.path.basename(file_path)
```



```python
    with open(new_file, "w") as f:
        f.writelines(output_lines)

    print(f"Documentation generated in {new_file}")
print("Inserting docstrings into 'week9.4.py'...")
insert_docstrings("week9.4.py")
```

Output:



```
Inserting docstrings into 'week9.4.py'...
Documentation generated in documented_week9.4.py
(.venv) PS C:\Users\vikas\Downloads\AI Assist Coding>
```

Summary:

This task builds a simple internal documentation utility that scans a Python file, detects functions and classes using the ast module, and inserts placeholder Google-style docstrings automatically. It helps developers quickly scaffold documentation for new files, encouraging standardized documentation practices while saving time. This approach can be extended into a more advanced internal documentation tool for real-world development teams.