AI-ASSISSTED CODING

NAME:SANIYA **ASSIGNMENT:9.1**

BATCH:05

BRANCH:CSE

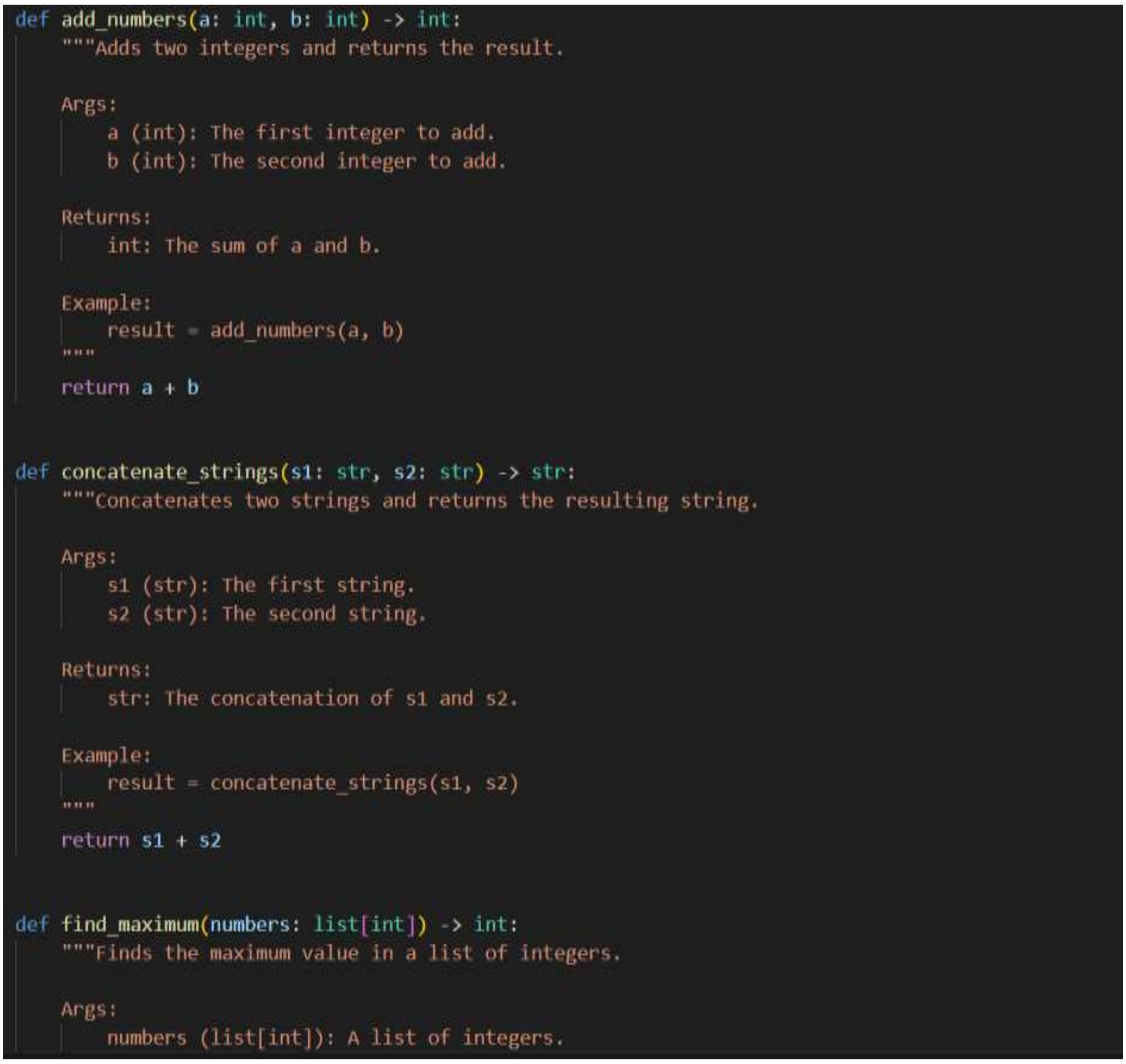
HT.NO:2403A510E7

TASK 1:

- ### PROMPT:

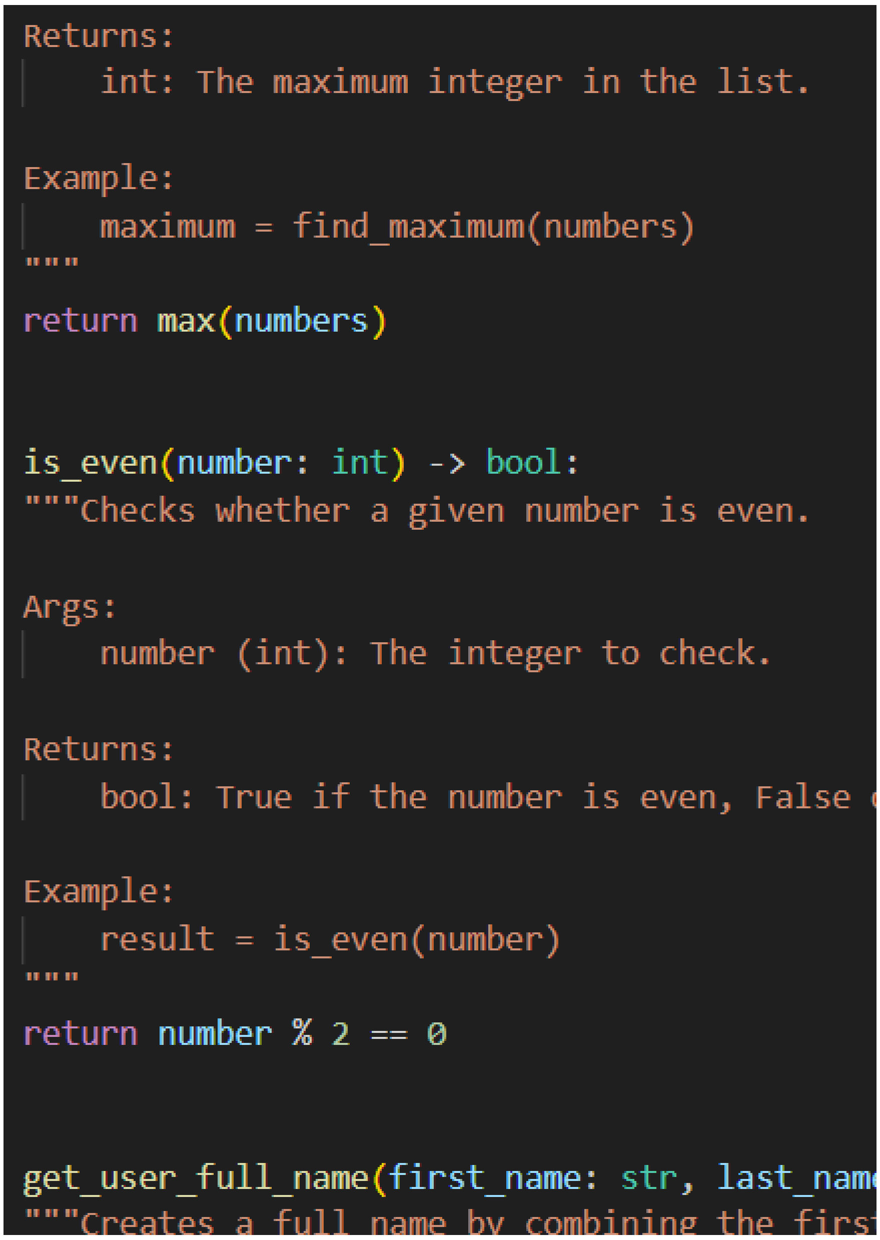
  Write Google-style docstrings for all the functions in a given Python script. Each docstring should include a brief description of the function, the parameters with their type hints, the return values with type hints, and an example usage section. The docstrings must follow the exact Google-style formatting standards, and no explicit input-output examples with real values should be provided. Apply this consistently to every function in the script

- ### CODE GENERATED:



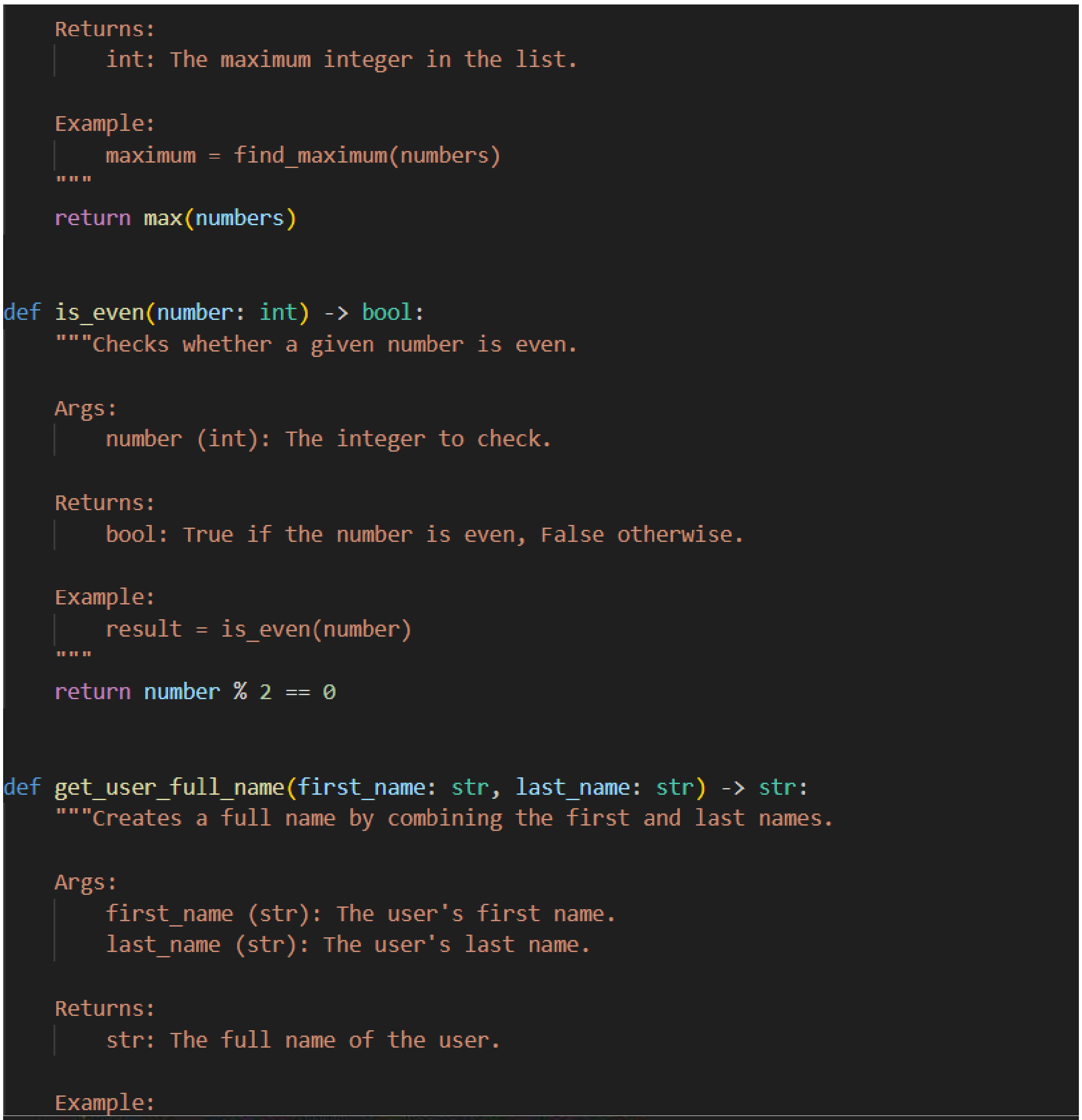
```python
def add_numbers(a: int, b: int) -> int:
    """Adds two integers and returns the result.

    Args:
        a (int): The first integer to add.
        b (int): The second integer to add.

    Returns:
        int: The sum of a and b.

    Example:
        result = add_numbers(a, b)
    """
    return a + b


def concatenate_strings(s1: str, s2: str) -> str:
    """Concatenates two strings and returns the resulting string.

    Args:
        s1 (str): The first string.
        s2 (str): The second string.

    Returns:
        str: The concatenation of s1 and s2.

    Example:
        result = concatenate_strings(s1, s2)
    """
    return s1 + s2


def find_maximum(numbers: list[int]) -> int:
    """Finds the maximum value in a list of integers.

    Args:
        numbers (list[int]): A list of integers.
```

➢



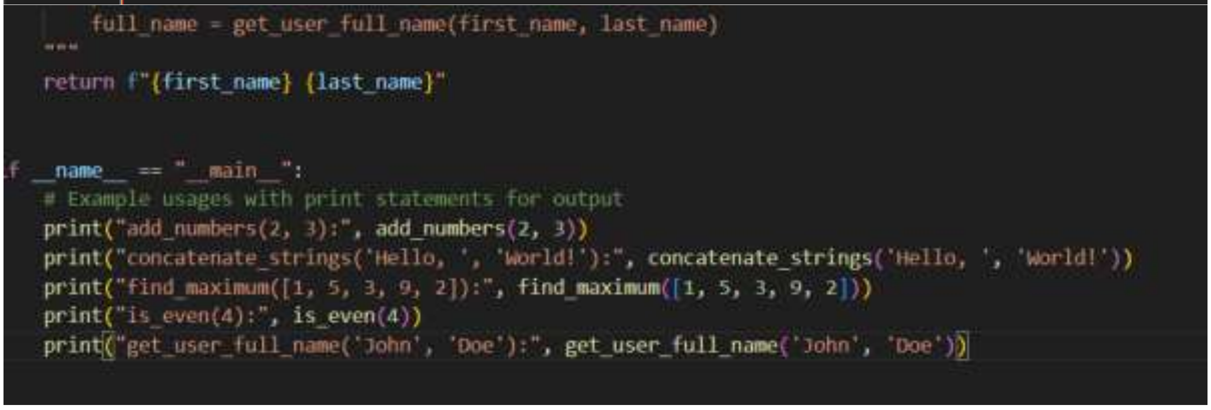
```
Returns:
    int: The maximum integer in the list.

Example:
    maximum = find_maximum(numbers)
"""
return max(numbers)


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

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

Returns:
    bool: True if the number is even, False 

Example:
    result = is_even(number)
"""
return number % 2 == 0


get_user_full_name(first_name: str, last_name
"""Creates a full name by combining the first
```

➢



```
    Returns:
        int: The maximum integer in the list.

    Example:
        maximum = find_maximum(numbers)
    """
    return max(numbers)


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

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

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

    Example:
        result = is_even(number)
    """
    return number % 2 == 0


def get_user_full_name(first_name: str, last_name: str) -> str:
    """Creates a full name by combining the first and last names.

    Args:
        first_name (str): The user's first name.
        last_name (str): The user's last name.

    Returns:
        str: The full name of the user.

    Example:
```

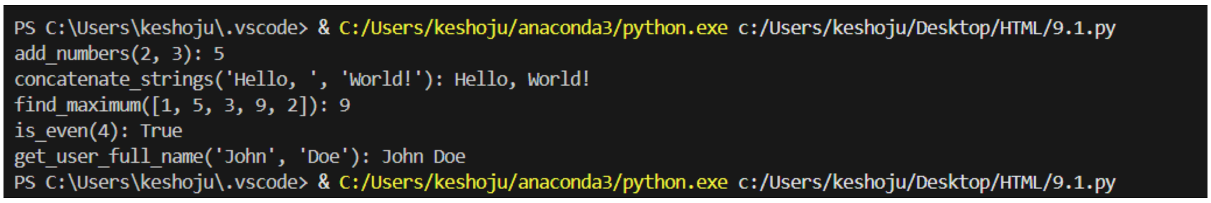
➢



```
        full_name = get_user_full_name(first_name, last_name)
    """
    return f"{first_name} {last_name}"


f __name__ == "__main__":
    # Example usages with print statements for output
    print("add_numbers(2, 3):", add_numbers(2, 3))
    print("concatenate_strings('Hello, ', 'World!'):", concatenate_strings('Hello, ', 'World!'))
    print("find_maximum([1, 5, 3, 9, 2]):", find_maximum([1, 5, 3, 9, 2]))
    print("is_even(4):", is_even(4))
    print("get_user_full_name('John', 'Doe'):", get_user_full_name('John', 'Doe'))
```

➢

➢ OUTPUT:



```
PS C:\Users\keshoju\.vscode> & C:/Users/keshoju/anaconda3/python.exe c:/Users/keshoju/Desktop/HTML/9.1.py
add_numbers(2, 3): 5
concatenate_strings('Hello, ', 'World!'): Hello, World!
find_maximum([1, 5, 3, 9, 2]): 9
is_even(4): True
get_user_full_name('John', 'Doe'): John Doe
PS C:\Users\keshoju\.vscode> & C:/Users/keshoju/anaconda3/python.exe c:/Users/keshoju/Desktop/HTML/9.1.py
```

➢ OBSERVATION:

This code defines five utility functions for basic operations: adding numbers, concatenating strings, finding the maximum in a list, checking if a number is even, and combining first and last names. Each function includes type hints and docstrings. The main block demonstrates their usage with print statements.

❖ TASK 2:

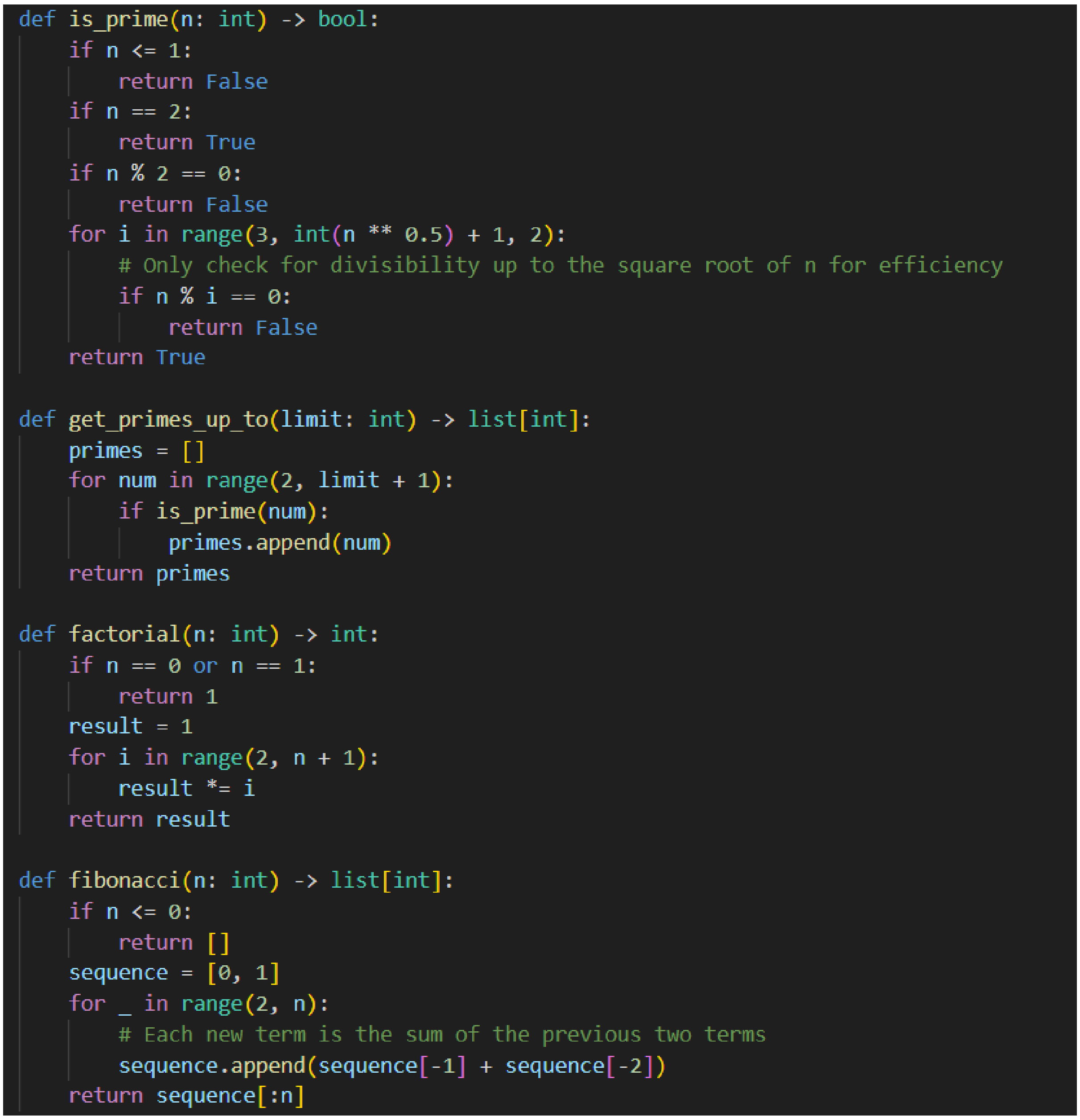
PROMPT:

Can you add meaningful inline comments to this Python script, explaining only the complex or non-intuitive parts of the code? Skip obvious syntax explanations and focus on tricky logic sections. The comments should be concise, context-aware, and improve code readability and maintainability.

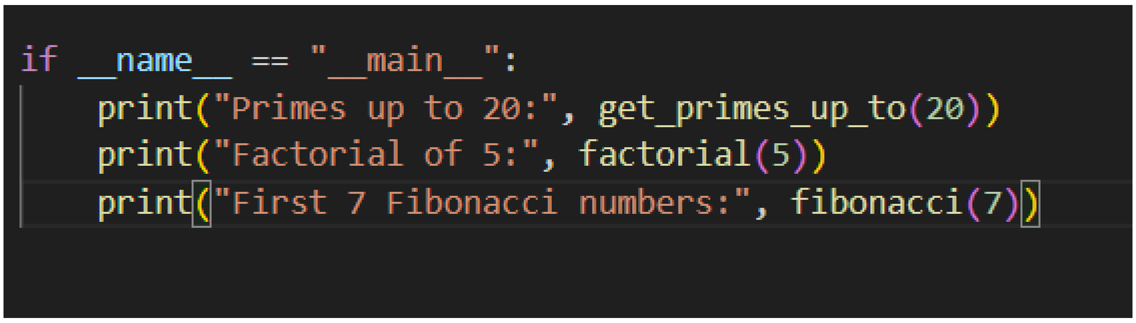
Expected Output:

A Python script with clear, concise inline comments for complex logic blocks.

- CODE GENERATED:

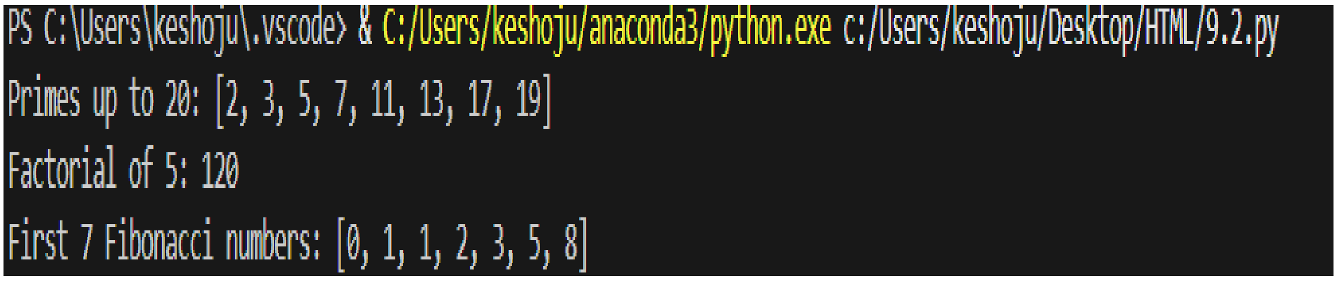


```python
def is_prime(n: int) -> bool:
    if n <= 1:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False
    for i in range(3, int(n ** 0.5) + 1, 2):
        # Only check for divisibility up to the square root of n for efficiency
        if n % i == 0:
            return False
    return True

def get_primes_up_to(limit: int) -> list[int]:
    primes = []
    for num in range(2, limit + 1):
        if is_prime(num):
            primes.append(num)
    return primes

def factorial(n: int) -> int:
    if n == 0 or n == 1:
        return 1
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result

def fibonacci(n: int) -> list[int]:
    if n <= 0:
        return []
    sequence = [0, 1]
    for _ in range(2, n):
        # Each new term is the sum of the previous two terms
        sequence.append(sequence[-1] + sequence[-2])
    return sequence[:n]
```



```python
if __name__ == "__main__":
    print("Primes up to 20:", get_primes_up_to(20))
    print("Factorial of 5:", factorial(5))
    print("First 7 Fibonacci numbers:", fibonacci(7))
```

OUTPUT:



```
PS C:\Users\keshoju\.vscode> & C:/Users/keshoju/anaconda3/python.exe c:/Users/keshoju/Desktop/HTML/9.2.py
Primes up to 20: [2, 3, 5, 7, 11, 13, 17, 19]
Factorial of 5: 120
First 7 Fibonacci numbers: [0, 1, 1, 2, 3, 5, 8]
```

OBSERVATION:

This code provides functions to check if a number is prime, generate primes up to a limit, compute factorials, and generate Fibonacci sequences. The main block demonstrates each function. The code is efficient, uses type hints, and is easy to read.

❖ TASK-3:

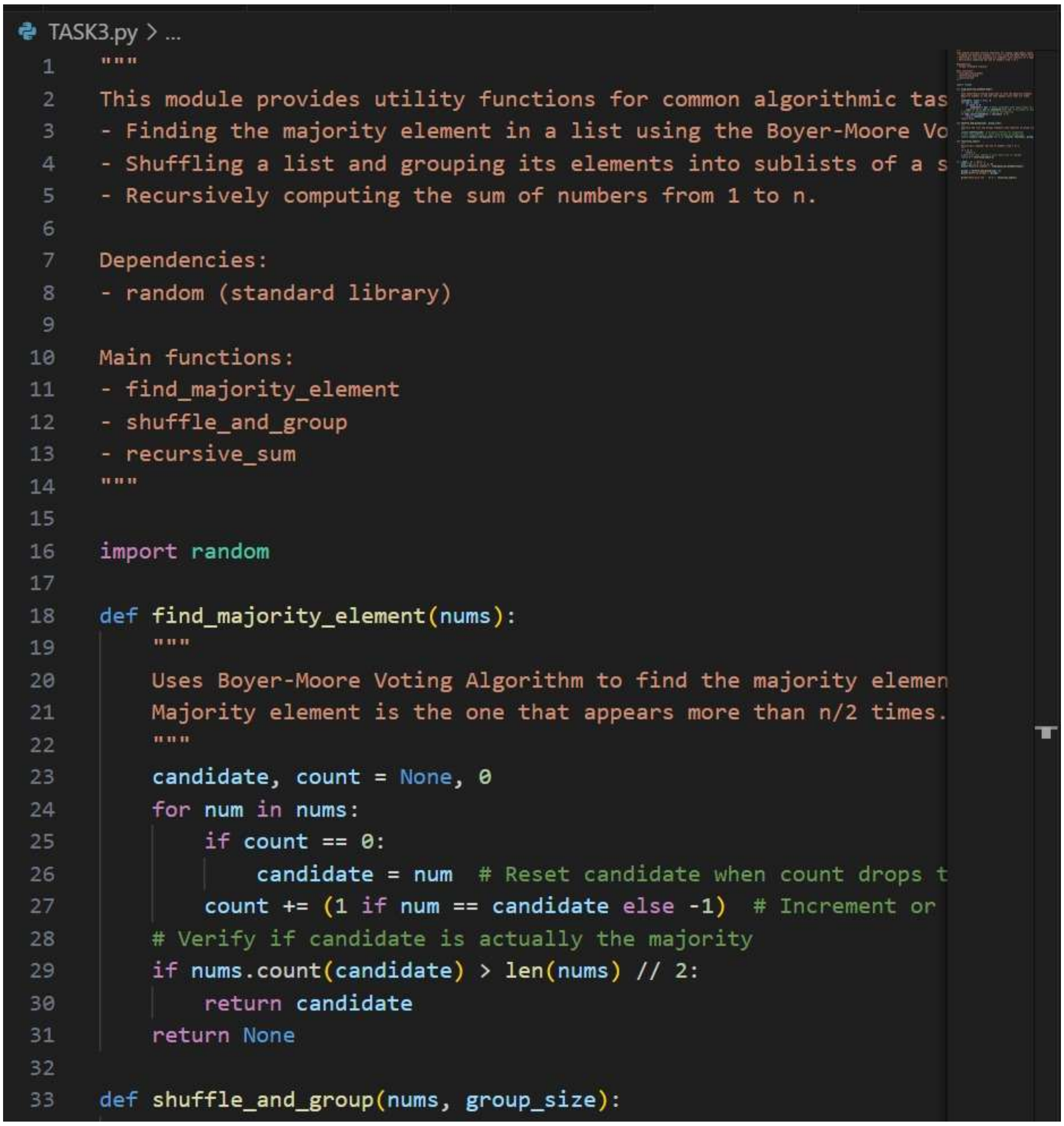
PROMPT:

Can you add a module-level docstring at the top of this Python file that summarizes its purpose, mentions any dependencies, and briefly describes the main functions or classes it contains? The docstring should be a concise multi-line description that improves readability without rewriting or duplicating the code.

Expected Output #3:

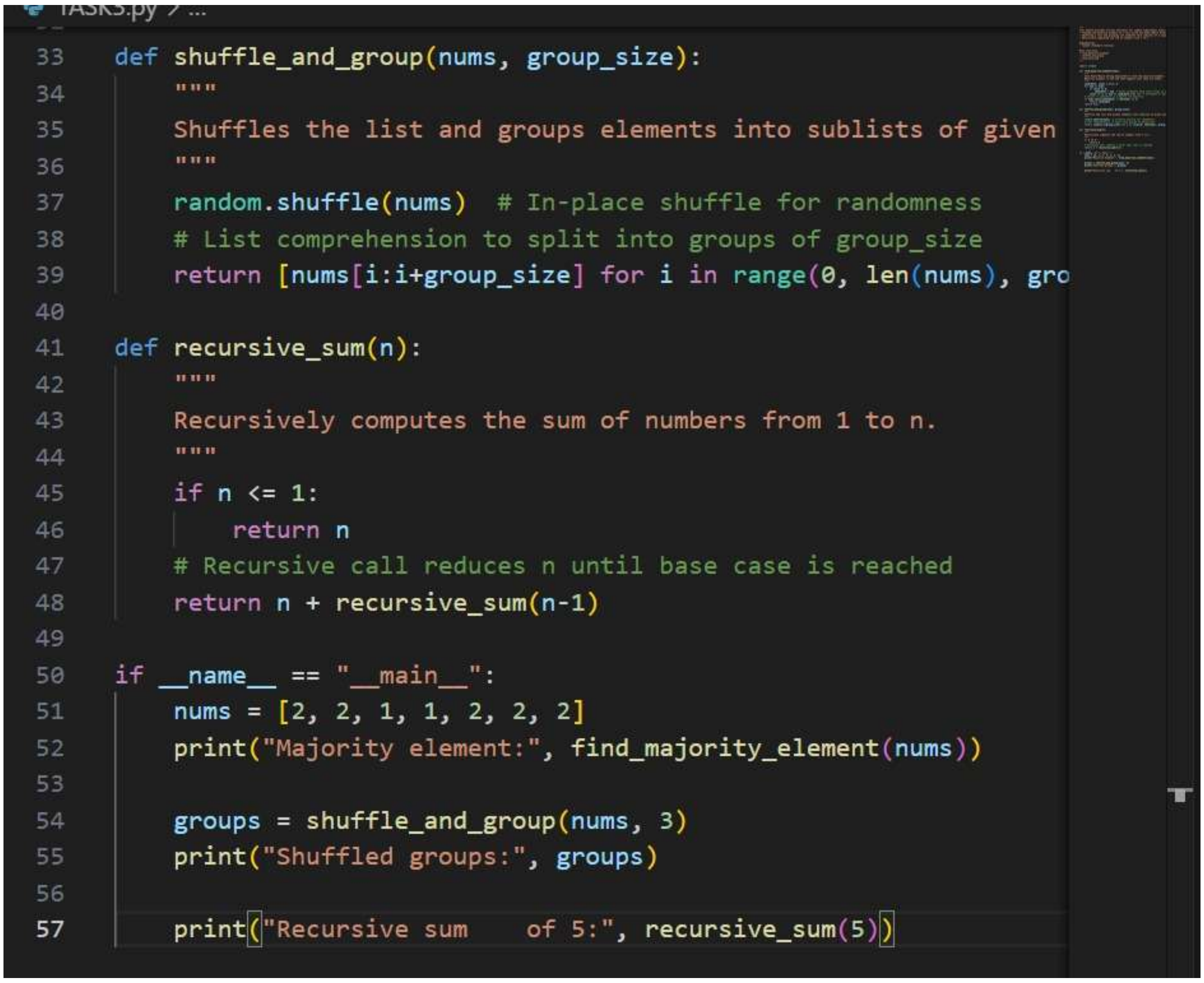
A complete, clear, and concise module-level docstring placed at the beginning of the file.

CODE GENERATED:



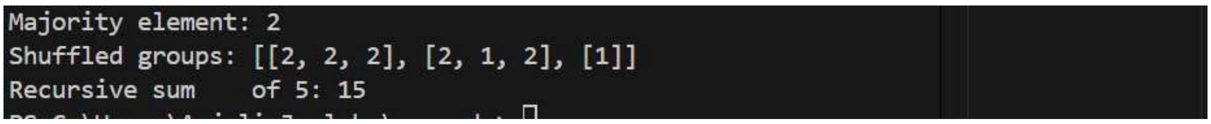
TASK3.py > ...

```
"""
This module provides utility functions for common algorithmic tas
- Finding the majority element in a list using the Boyer-Moore Vo
- Shuffling a list and grouping its elements into sublists of a s
- Recursively computing the sum of numbers from 1 to n.

Dependencies:
- random (standard library)

Main functions:
- find_majority_element
- shuffle_and_group
- recursive_sum
"""

import random

def find_majority_element(nums):
    """
    Uses Boyer-Moore Voting Algorithm to find the majority elemen
    Majority element is the one that appears more than n/2 times.
    """
    candidate, count = None, 0
    for num in nums:
        if count == 0:
            candidate = num  # Reset candidate when count drops t
        count += (1 if num == candidate else -1)  # Increment or
    # Verify if candidate is actually the majority
    if nums.count(candidate) > len(nums) // 2:
        return candidate
    return None

def shuffle_and_group(nums, group_size):
```



```python
TASKS.py > ...
def shuffle_and_group(nums, group_size):
    """
    Shuffles the list and groups elements into sublists of given
    """
    random.shuffle(nums)  # In-place shuffle for randomness
    # List comprehension to split into groups of group_size
    return [nums[i:i+group_size] for i in range(0, len(nums), gro

def recursive_sum(n):
    """
    Recursively computes the sum of numbers from 1 to n.
    """
    if n <= 1:
        return n
    # Recursive call reduces n until base case is reached
    return n + recursive_sum(n-1)

if __name__ == "__main__":
    nums = [2, 2, 1, 1, 2, 2, 2]
    print("Majority element:", find_majority_element(nums))

    groups = shuffle_and_group(nums, 3)
    print("Shuffled groups:", groups)

    print("Recursive sum    of 5:", recursive_sum(5))
```

OUTPUT:



```
Majority element: 2
Shuffled groups: [[2, 2, 2], [2, 1, 2], [1]]
Recursive sum    of 5: 15
```

OBSERVATION:

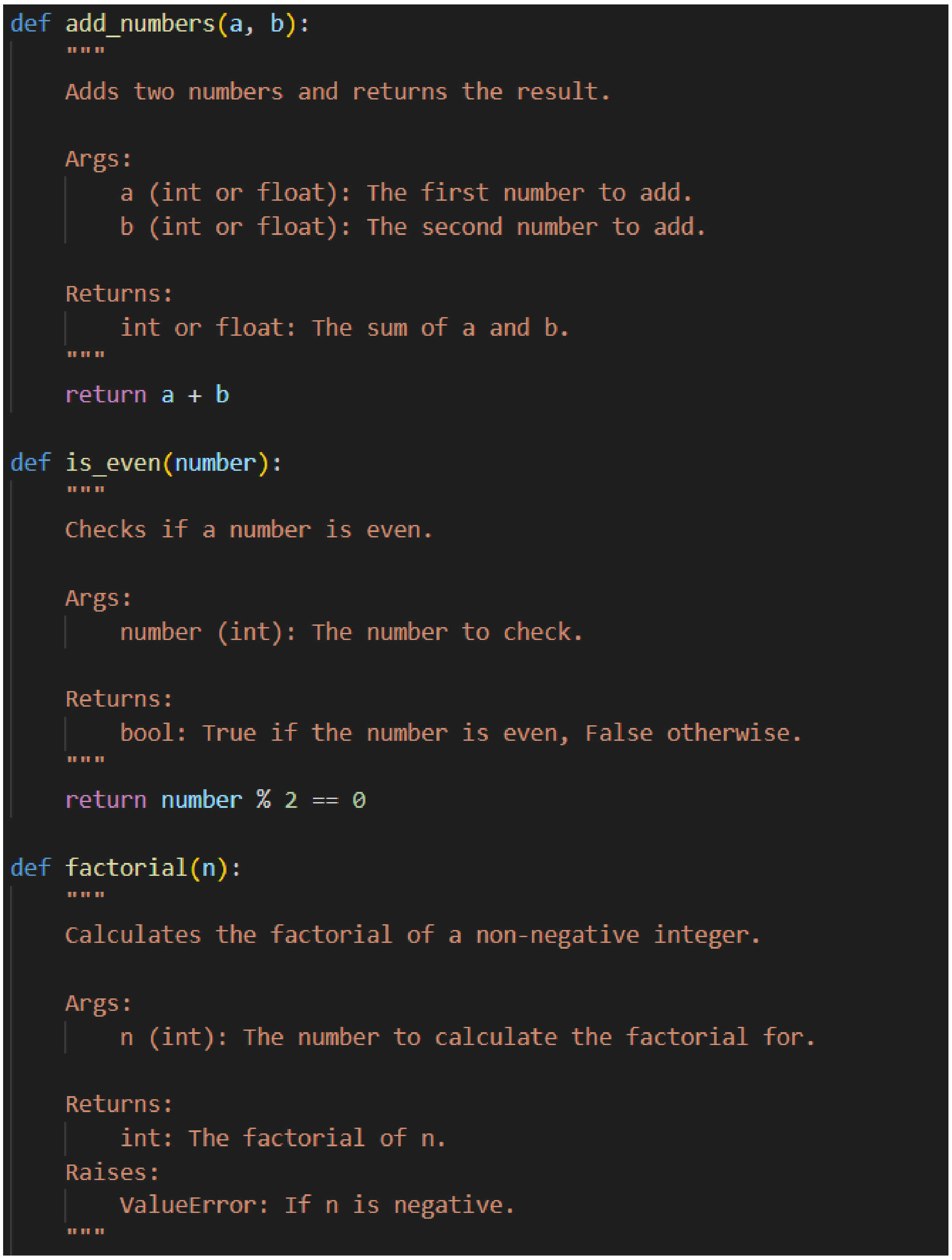
This module provides a concise overview of its purpose, lists any key dependencies, and briefly describes the main functions or classes it contains. It serves as a quick reference for understanding the file’s role within the project without duplicating or rewriting the actual code.

TASK -4 :

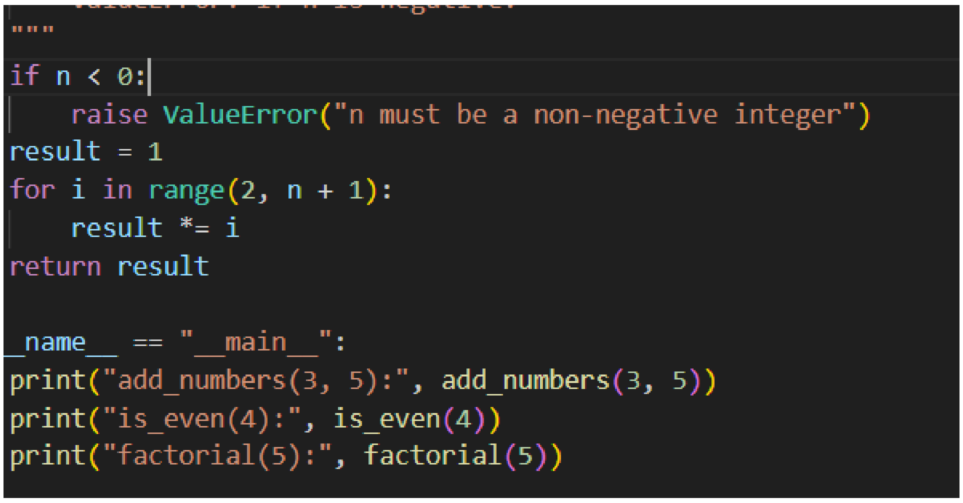
PROMPT:

Python file with inline comments explaining the logic inside each function. Please convert these inline comments into structured function docstrings using the Google style format. Each function’s docstring should include a clear summary of what the function does, the arguments with their types and descriptions under an "Args:" section, and the return values with type and meaning under a "Returns:" section. Ensure that the docstrings preserve the meaning of the inline comments while improving readability and consistency. After creating the docstrings, remove the redundant inline comments so that the final output is clean Python code with only standardized Google-style docstrings.

CODE GENERATED:

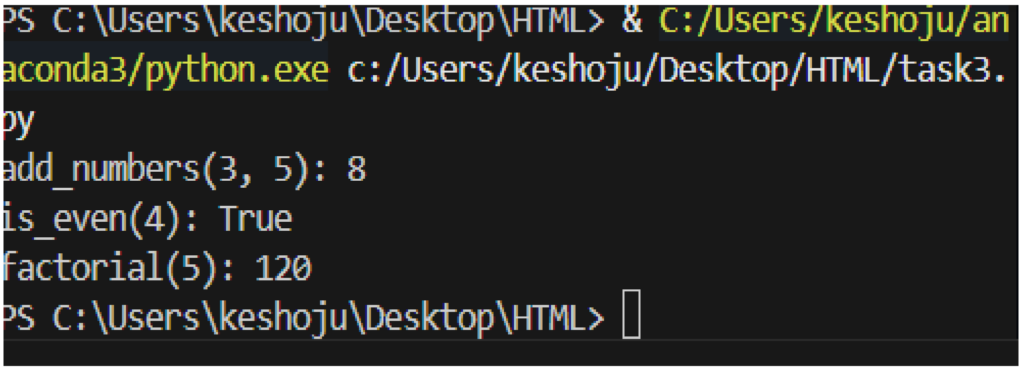


```python
def add_numbers(a, b):
    """
    Adds two numbers and returns the result.

    Args:
        a (int or float): The first number to add.
        b (int or float): The second number to add.

    Returns:
        int or float: The sum of a and b.
    """
    return a + b

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

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

    Returns:
        bool: True if the number is even, False otherwise.
    """
    return number % 2 == 0

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

    Args:
        n (int): The number to calculate the factorial for.

    Returns:
        int: The factorial of n.
    Raises:
        ValueError: If n is negative.
    """
```



```
"""
if n < 0:
    raise ValueError("n must be a non-negative integer")
result = 1
for i in range(2, n + 1):
    result *= i
return result

_name__ == "__main__":
print("add_numbers(3, 5):", add_numbers(3, 5))
print("is_even(4):", is_even(4))
print("factorial(5):", factorial(5))
```

OUTPUT:



```
PS C:\Users\keshoju\Desktop\HTML> & C:/Users/keshoju/an
aconda3/python.exe c:/Users/keshoju/Desktop/HTML/task3.
py
add_numbers(3, 5): 8
is_even(4): True
factorial(5): 120
PS C:\Users\keshoju\Desktop\HTML> 
```

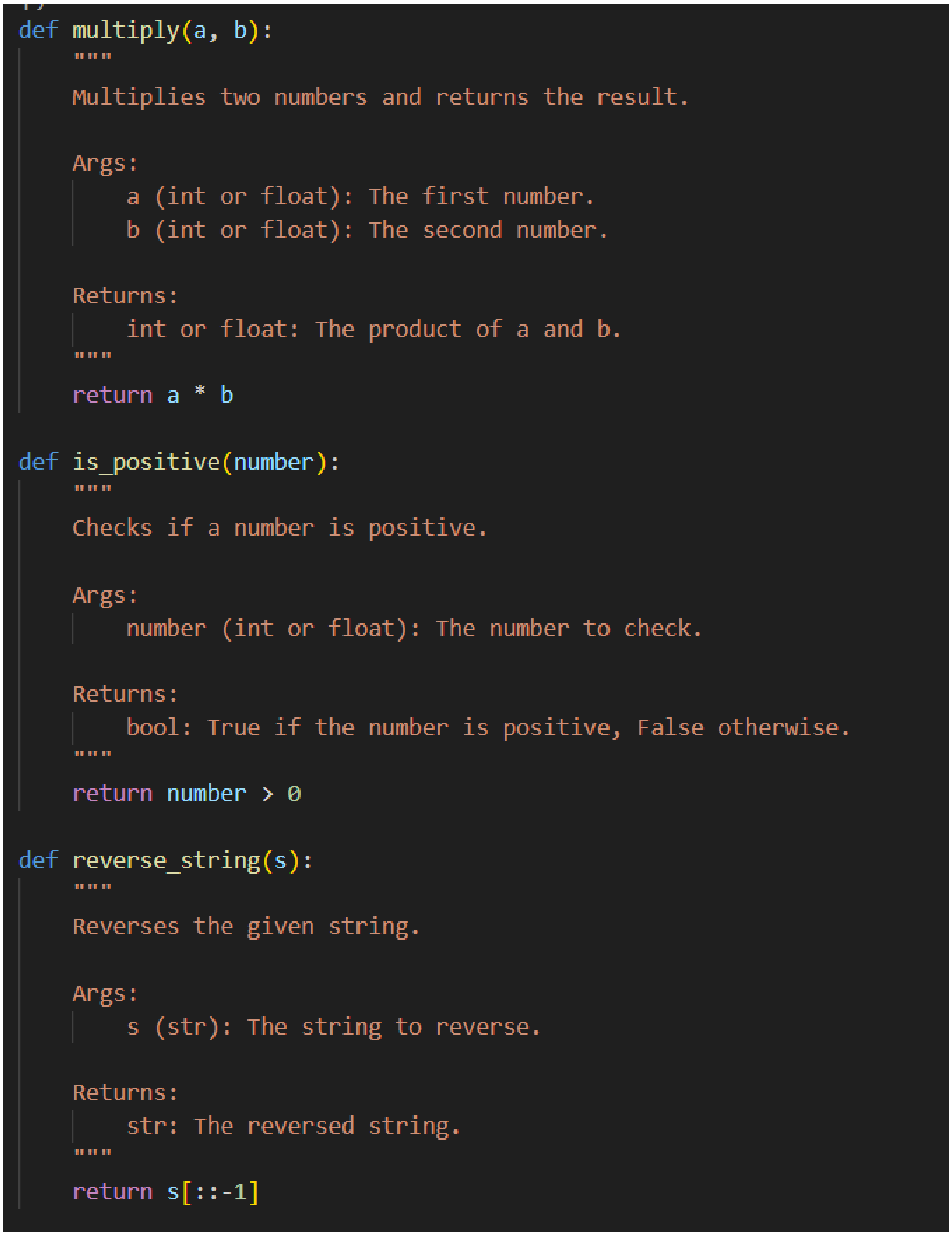
OBSERVATION:

This code defines and demonstrates three basic mathematical functions—addition, even number checking, and factorial calculation—using clear, well-documented Python functions with example outputs when run as a script.

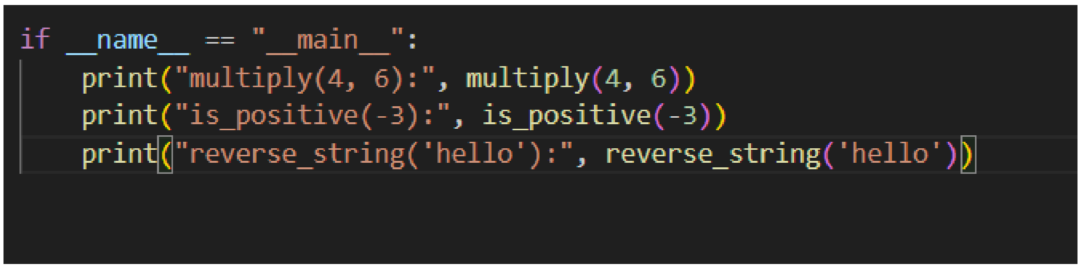
❖ TASK-5

PROMPT:

Python file where the existing docstrings are outdated or inaccurate. Please carefully review each function and class, compare the current code behavior with its docstring, and then rewrite the docstrings so they are correct and consistent. Use the Google style docstring format, including a summary line, an "Args:" section with parameter types and descriptions, and a "Returns:" section with return type and description. The final output should be the same Python file, but with updated, accurate, and standardized docstrings that fully reflect what the code actually does.

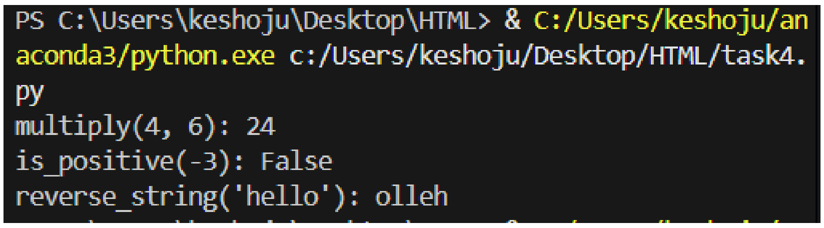


```python
def multiply(a, b):
    """
    Multiplies two numbers and returns the result.

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

    Returns:
        int or float: The product of a and b.
    """
    return a * b

def is_positive(number):
    """
    Checks if a number is positive.

    Args:
        number (int or float): The number to check.

    Returns:
        bool: True if the number is positive, False otherwise.
    """
    return number > 0

def reverse_string(s):
    """
    Reverses the given string.

    Args:
        s (str): The string to reverse.

    Returns:
        str: The reversed string.
    """
    return s[::-1]
```



```python
if __name__ == "__main__":
    print("multiply(4, 6):", multiply(4, 6))
    print("is_positive(-3):", is_positive(-3))
    print("reverse_string('hello'):", reverse_string('hello'))
```

OUTPUT:



```
PS C:\Users\keshoju\Desktop\HTML> & C:/Users/keshoju/an
aconda3/python.exe c:/Users/keshoju/Desktop/HTML/task4.
py
multiply(4, 6): 24
is_positive(-3): False
reverse_string('hello'): olleh
```

CONCLUSION:

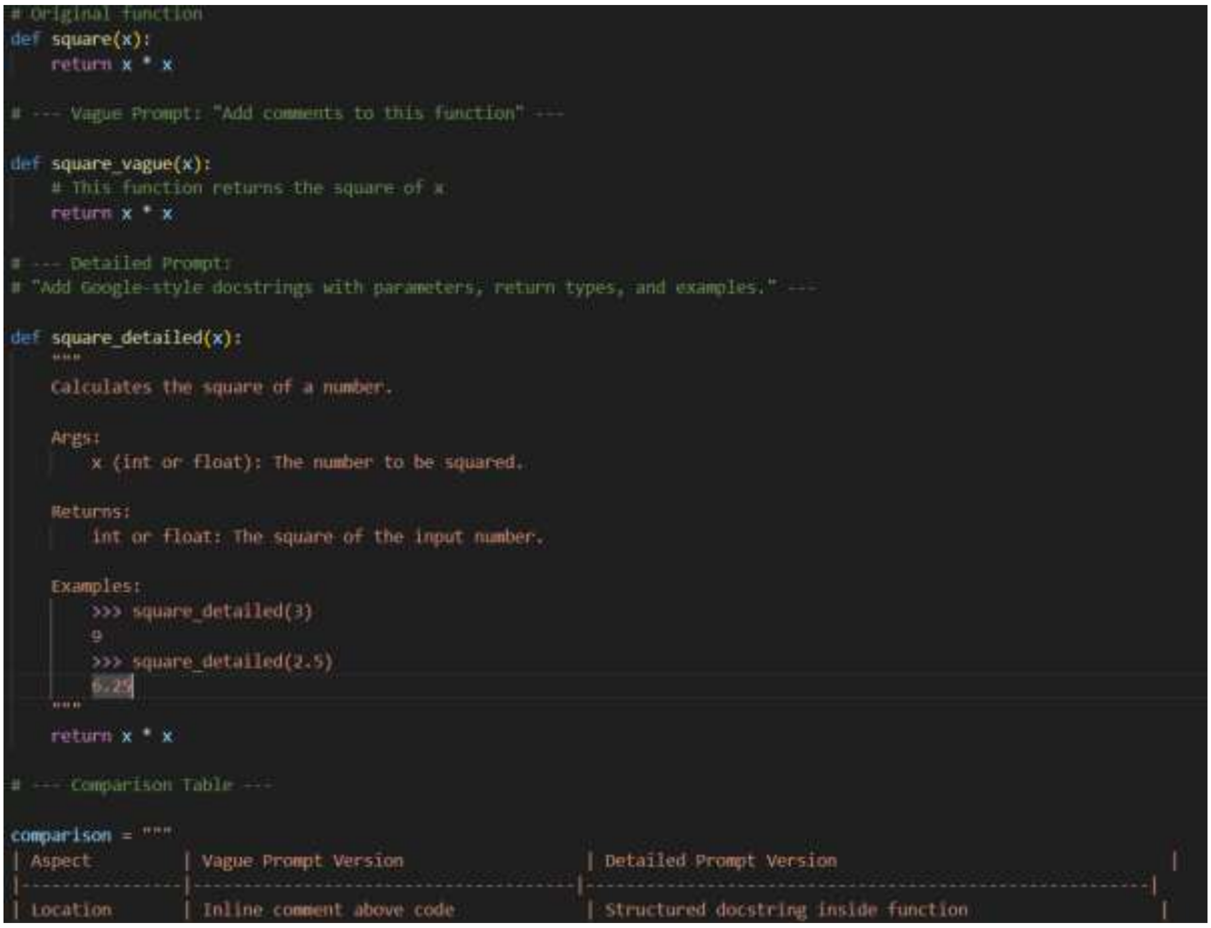
This code provides three utility functions for multiplying numbers, checking if a number is positive, and reversing a string, each with clear Google-style docstrings and example outputs when run as a script.

❖ TASK – 6

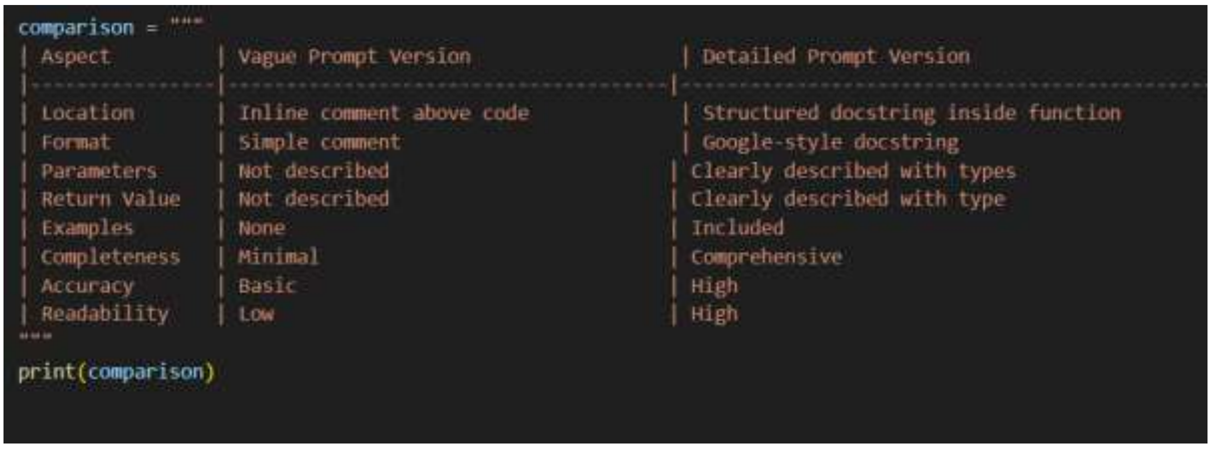
PROMPT:

Take the same Python function and process it with two different prompts: one vague prompt ("Add comments to this function") and one detailed prompt ("Add Google-style docstrings with parameters, return types, and examples"). For each case, show the resulting documentation applied to the function. Then, create a comparison table that highlights the differences in quality, accuracy, and completeness between the vague prompt and the detailed prompt. The expected output should include both versions of the documented function followed by a clear comparison table with observations.

CODE GENERATED:

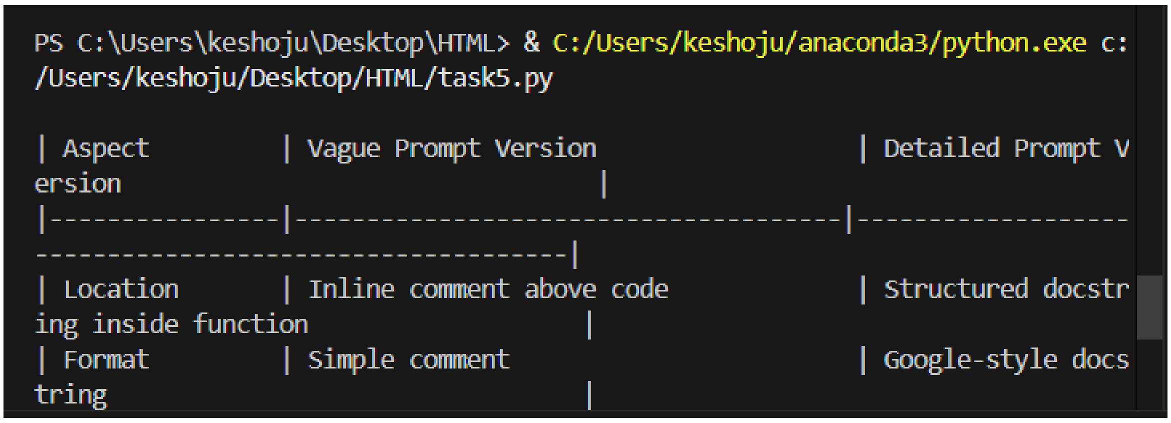


```python
# Original function
def square(x):
    return x * x

# --- Vague Prompt: "Add comments to this function" ---

def square_vague(x):
    # This function returns the square of x
    return x * x

# --- Detailed Prompt:
# "Add Google-style docstrings with parameters, return types, and examples." ---

def square_detailed(x):
    """
    Calculates the square of a number.

    Args:
        x (int or float): The number to be squared.

    Returns:
        int or float: The square of the input number.

    Examples:
        >>> square_detailed(3)
        9
        >>> square_detailed(2.5)
        6.25
    """
    return x * x

# --- Comparison Table ---

comparison = """
| Aspect        | Vague Prompt Version                  | Detailed Prompt Version                                    |
|---------------|---------------------------------------|------------------------------------------------------------|
| Location      | Inline comment above code             | Structured docstring inside function                       |
```

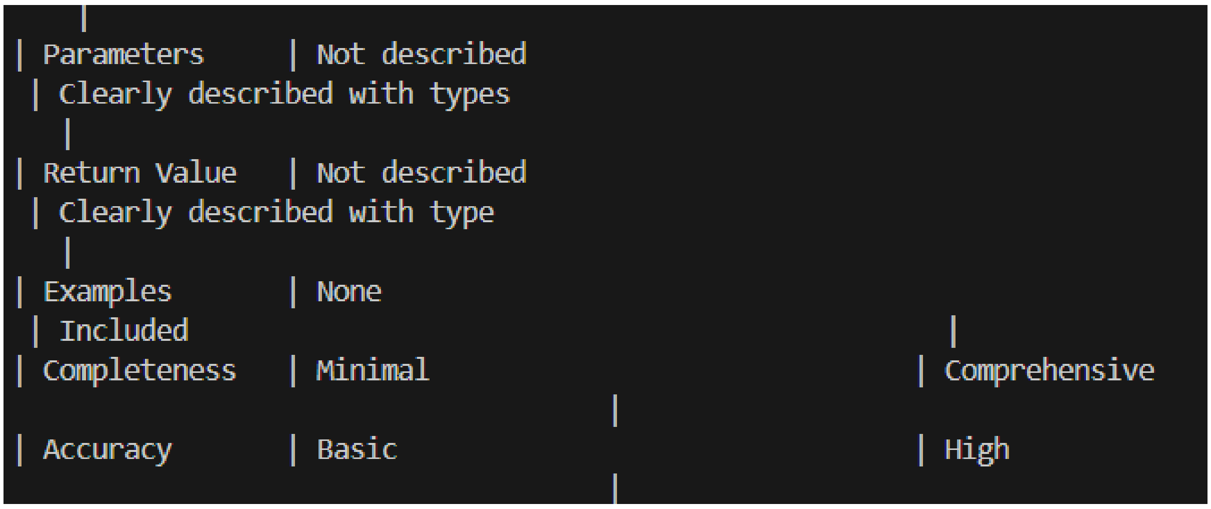


```python
comparison = """
| Aspect        | Vague Prompt Version                  | Detailed Prompt Version
|---------------|---------------------------------------|-----------------------------------------
| Location      | Inline comment above code             | Structured docstring inside function
| Format        | Simple comment                        | Google-style docstring
| Parameters    | Not described                        | Clearly described with types
| Return Value  | Not described                        | Clearly described with type
| Examples      | None                                 | Included
| Completeness  | Minimal                              | Comprehensive
| Accuracy      | Basic                                | High
| Readability   | Low                                  | High
"""
print(comparison)
```

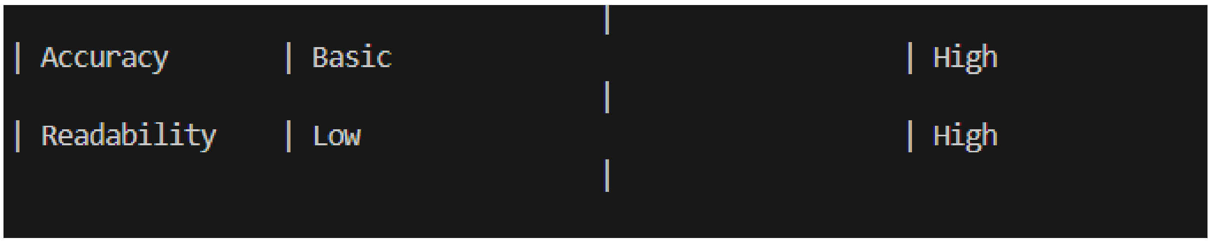
OUTPUT:



```
PS C:\Users\keshoju\Desktop\HTML> & C:/Users/keshoju/anaconda3/python.exe c:
/Users/keshoju/Desktop/HTML/task5.py

| Aspect          | Vague Prompt Version                   | Detailed Prompt V
ersion                                  |
|---------------|------------------------------------|-------------------
-----------------------------------|
| Location        | Inline comment above code              | Structured docstr
ing inside function                    |
| Format          | Simple comment                         | Google-style docs
tring                                  |
```



```
    |
| Parameters      | Not described
  | Clearly described with types
    |
| Return Value    | Not described
  | Clearly described with type
    |
| Examples        | None
  | Included                                          |
| Completeness    | Minimal                                       | Comprehensive
                                        |
| Accuracy        | Basic                                         | High
                                        |
```



```
                                        |
| Accuracy        | Basic                                         | High
                                        |
| Readability     | Low                                           | High
                                        |
```

OBSERVATION:

The comparison demonstrates that a vague prompt results in minimal and less informative documentation, while a detailed prompt produces comprehensive, accurate, and user-friendly docstrings. Using detailed prompts leads to higher quality, more maintainable, and more understandable code documentation.