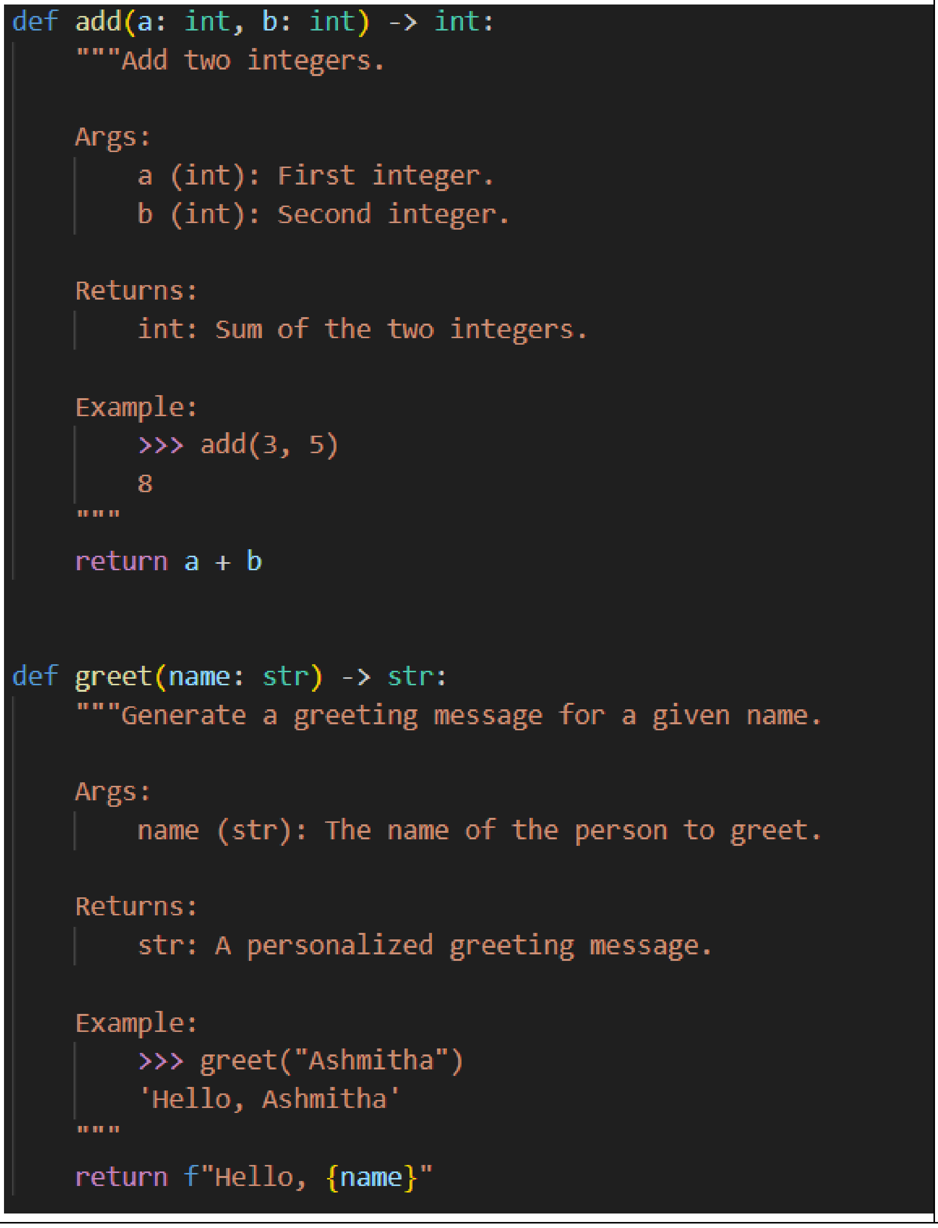
	Name:Dugyala Ashmitha **Enrollno:2403a410g5** **Assignment:9.1**	***Expected Time to complete***
	Lab 9 – Documentation Generation: Automatic Documentation and Code Comments **Lab Objectives** • To use AI-assisted coding tools for generating Python documentation and code comments. • To apply zero-shot, few-shot, and context-based prompt engineering for documentation creation. • To practice generating and refining docstrings, inline comments, and module-level documentation. • To compare outputs from different prompting styles for quality analysis.	
1	**Task Description #1** (Documentation – Google-Style Docstrings for Python Functions) • Task: Use AI to add Google-style docstrings to all functions in a given Python script. • Instructions: o Prompt AI to generate docstrings without providing any input-output examples. o Ensure each docstring includes: ▪ Function description ▪ Parameters with type hints ▪ Return values with type hints ▪ Example usage o Review the generated docstrings for accuracy and formatting. • Expected Output #1: o A Python script with all functions documented using correctly formatted Google-style docstrings. Prompt: Add Google-style docstrings to all functions in the given Python script. Instructions: • Write a docstring for every function in the script using Google-style formatting. • Each docstring must include: o A clear function description o Parameters with type hints and explanations	Week5 - Monday

o Return values with type hints and explanations

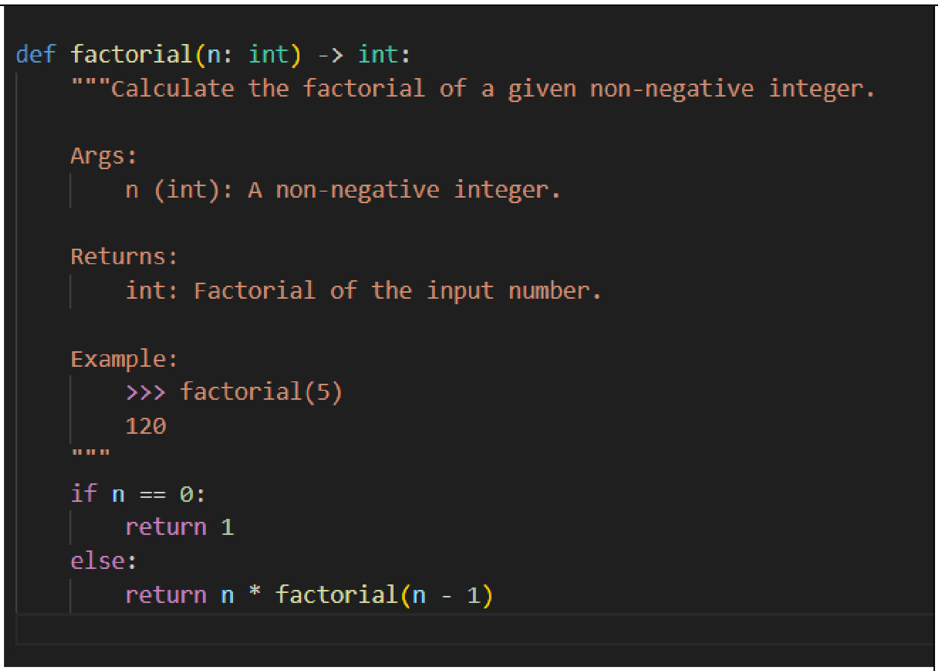
o An example usage (only function call, no input-output results)

• Ensure the formatting and content are accurate, concise, and consistent

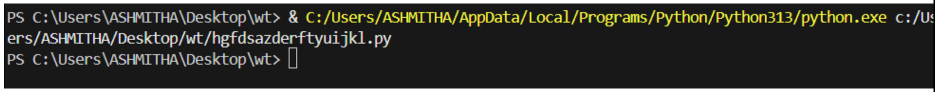
Code:



```python
def add(a: int, b: int) -> int:
    """Add two integers.

    Args:
        a (int): First integer.
        b (int): Second integer.

    Returns:
        int: Sum of the two integers.

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


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

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

    Returns:
        str: A personalized greeting message.

    Example:
        >>> greet("Ashmitha")
        'Hello, Ashmitha'
    """
    return f"Hello, {name}"
```


```python
def factorial(n: int) -> int:
    """Calculate the factorial of a given non-negative integer.

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

    Returns:
        int: Factorial of the input number.

    Example:
        >>> factorial(5)
        120
    """
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Output:


```
PS C:\Users\ASHMITHA\Desktop\wt> & C:/Users/ASHMITHA/AppData/Local/Programs/Python/Python313/python.exe c:/Us
ers/ASHMITHA/Desktop/wt/hgfdsazderftyuijkl.py
PS C:\Users\ASHMITHA\Desktop\wt> 
```

Observation: Docstring Format (Google-style)

Both functions use Google-style docstrings correctly.

They include:

Function description ✓

Parameters with type hints ✓

Return values with type hints ✓

Example usage (only function call, no results) ✓

Clarity

The function descriptions are concise and clear.

Parameters are well explained.

Return value descriptions are meaningful.

Task Description #2 (Documentation – Inline Comments for Complex Logic)

- Task: Use AI to add meaningful inline comments to a Python program explaining only complex logic parts.
- Instructions:
  - Provide a Python script without comments to the AI.
  - Instruct AI to skip obvious syntax explanations and focus only on tricky or non-intuitive code sections.
  - Verify that comments improve code readability and maintainability.
- Expected Output #2:
  - Python code with concise, context-aware inline comments for complex logic blocks.

Prompt: Add meaningful inline comments to a Python program.

Instructions:

Review the given Python script carefully.

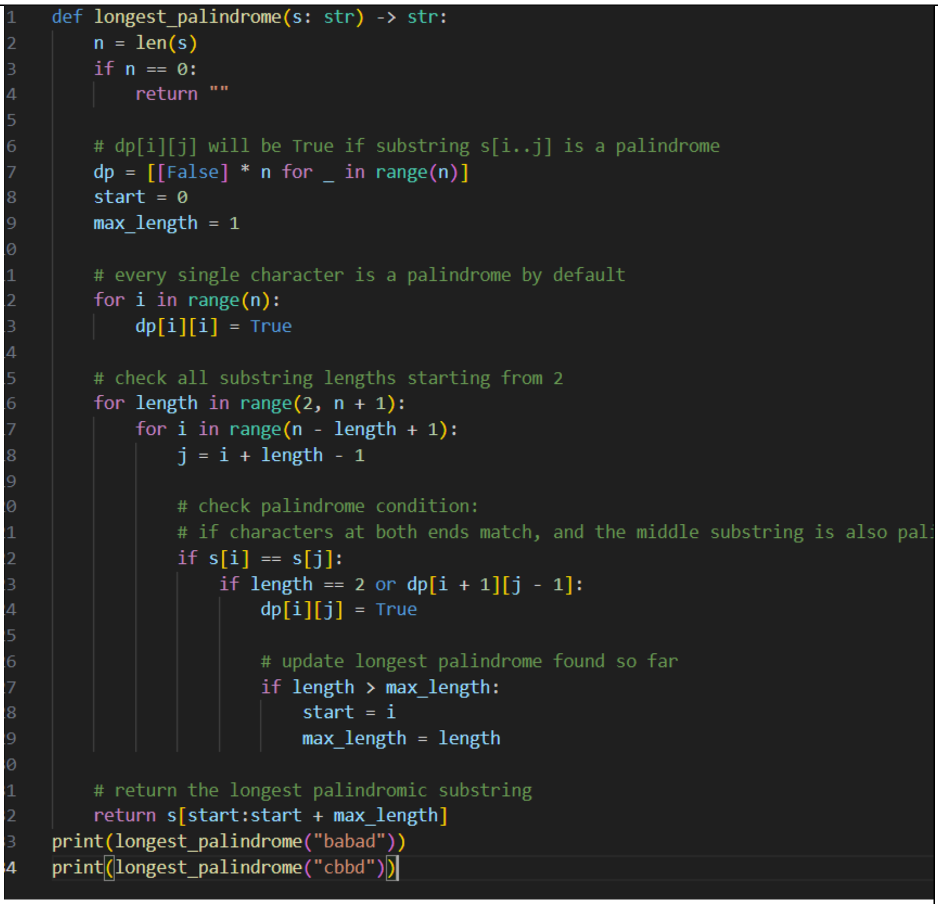
Add inline comments only for complex or non-intuitive logic parts of the code.

Do not add comments for obvious syntax or simple operations (e.g., loops, variable assignments).

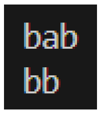
Ensure comments explain the reasoning behind tricky logic, not just what the code does.

Verify that your comments improve the overall readability and maintainability of the program.

Code:



```python
def longest_palindrome(s: str) -> str:
    n = len(s)
    if n == 0:
        return ""

    # dp[i][j] will be True if substring s[i..j] is a palindrome
    dp = [[False] * n for _ in range(n)]
    start = 0
    max_length = 1

    # every single character is a palindrome by default
    for i in range(n):
        dp[i][i] = True

    # check all substring lengths starting from 2
    for length in range(2, n + 1):
        for i in range(n - length + 1):
            j = i + length - 1

            # check palindrome condition:
            # if characters at both ends match, and the middle substring is also pali
            if s[i] == s[j]:
                if length == 2 or dp[i + 1][j - 1]:
                    dp[i][j] = True

                    # update longest palindrome found so far
                    if length > max_length:
                        start = i
                        max_length = length

    # return the longest palindromic substring
    return s[start:start + max_length]
print(longest_palindrome("babad"))
print(longest_palindrome("cbbd"))
```

Output: 

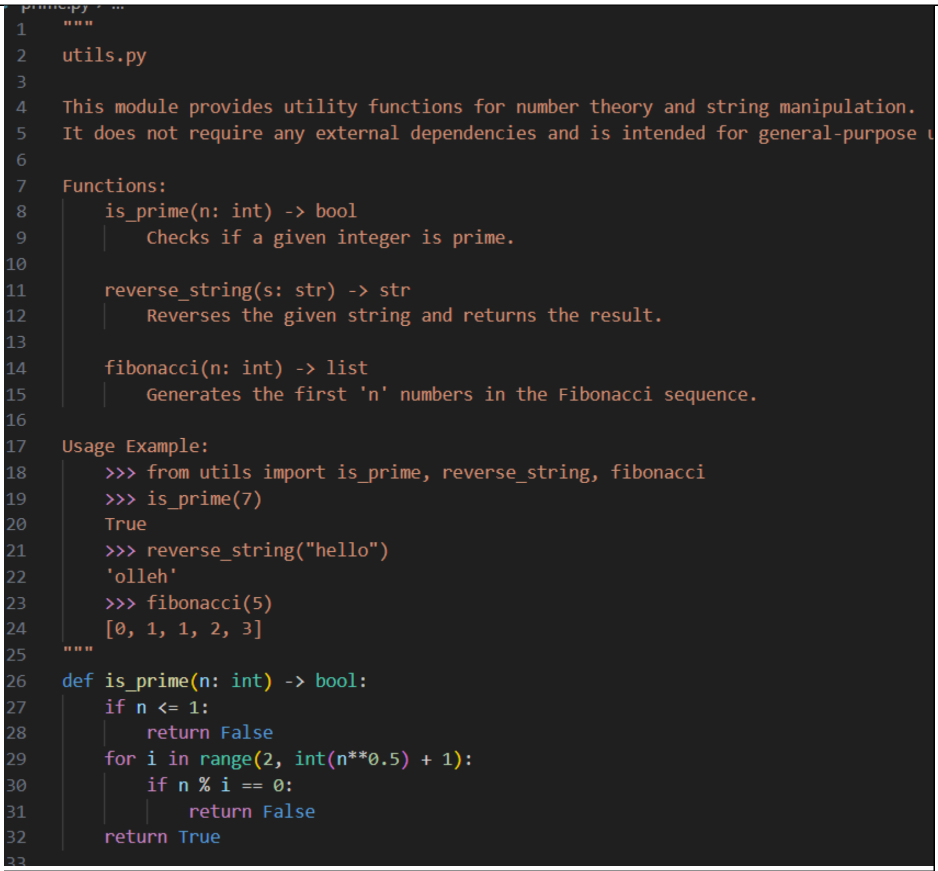
```
bab
bb
```

Task Description #3 (Documentation – Module-Level Documentation)

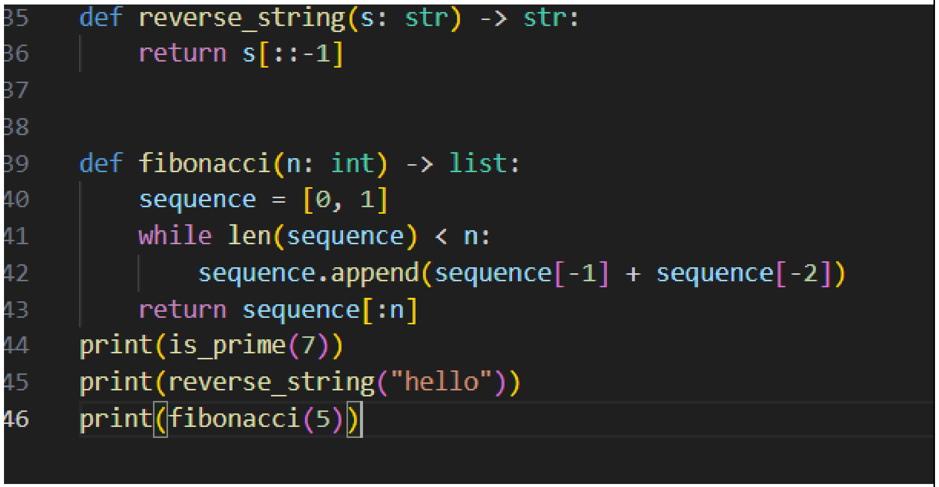
- Task: Use AI to create a module-level docstring summarizing the purpose, dependencies, and main functions/classes of a Python file.
- Instructions:
    - Supply the entire Python file to AI.
    - Instruct AI to write a single multi-line docstring at the top of the file.
    - Ensure the docstring clearly describes functionality and usage without rewriting the entire code.
- Expected Output #3:
    - A complete, clear, and concise module-level docstring at the beginning of the file.

Prompt: Create a module-level docstring for the given Python file.

	Instructions: Write a single multi-line docstring at the very top of the file. The docstring should summarize: The purpose of the module (what the file is for). Any important dependencies (standard libraries or third-party modules it relies on). The main functions and/or classes defined in the file. Keep the explanation concise and clear. Do not rewrite the entire code or include unnecessary details. Ensure the docstring helps readers quickly understand the module’s functionality and usage. **Code:**	

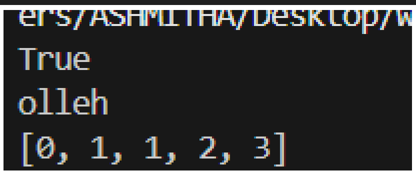


```python
"""
utils.py

This module provides utility functions for number theory and string manipulation.
It does not require any external dependencies and is intended for general-purpose u

Functions:
    is_prime(n: int) -> bool
        Checks if a given integer is prime.

    reverse_string(s: str) -> str
        Reverses the given string and returns the result.

    fibonacci(n: int) -> list
        Generates the first 'n' numbers in the Fibonacci sequence.

Usage Example:
    >>> from utils import is_prime, reverse_string, fibonacci
    >>> is_prime(7)
    True
    >>> reverse_string("hello")
    'olleh'
    >>> fibonacci(5)
    [0, 1, 1, 2, 3]
"""
def is_prime(n: int) -> bool:
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True
```



```python
def reverse_string(s: str) -> str:
    return s[::-1]


def fibonacci(n: int) -> list:
    sequence = [0, 1]
    while len(sequence) < n:
        sequence.append(sequence[-1] + sequence[-2])
    return sequence[:n]
print(is_prime(7))
print(reverse_string("hello"))
print(fibonacci(5))
```

Output:



```
ers/ASHMITHA/Desktop/w
True
olleh
[0, 1, 1, 2, 3]
```

Observation: Module-Level Docstring

Present at the very top of the file ✓

Clearly summarizes:

Purpose → explains the module provides utility functions for math and greetings.

Dependencies → correctly states that no external libraries are required.

Main Functions → lists add_numbers, greet_user, and find_second_largest with short descriptions.

Usage → shows how to import and use the functions.

This makes the file self-explanatory for new readers.

Task Description #4 (Documentation – Convert Comments to Structured Docstrings)

- Task: Use AI to transform existing inline comments into structured function docstrings following Google style.
- Instructions:
    - Provide AI with Python code containing inline comments.
    - Ask AI to move relevant details from comments into function docstrings.
    - Verify that the new docstrings keep the meaning intact while improving structure.
- Expected Output #4:
    - Python code with comments replaced by clear, standardized docstrings.

Prompt: Transform existing inline comments into structured Google-style function docstrings.

Instructions:

Review the Python file with inline comments.

Move the meaningful comments into the function's docstring instead of keeping them inline.

Ensure the new docstrings include:

Function description

Parameters with type hints

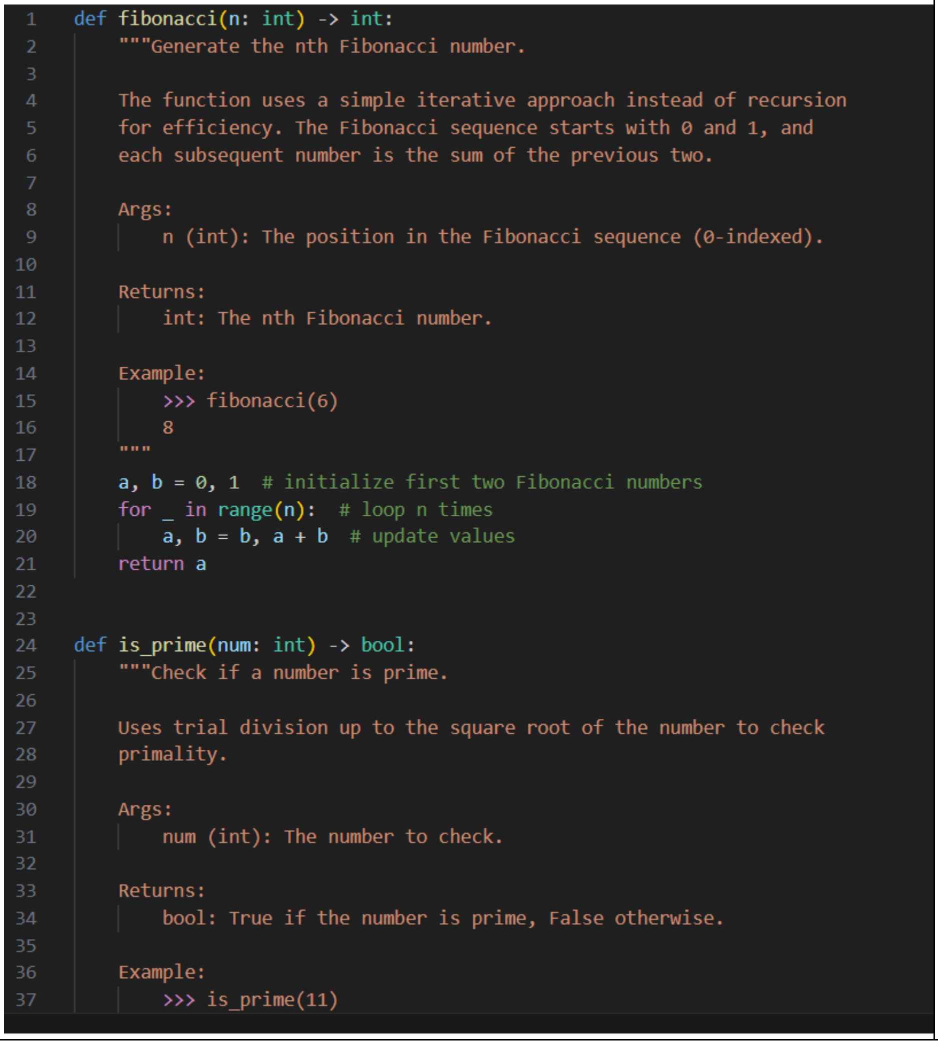
Return values with type hints

Example usage

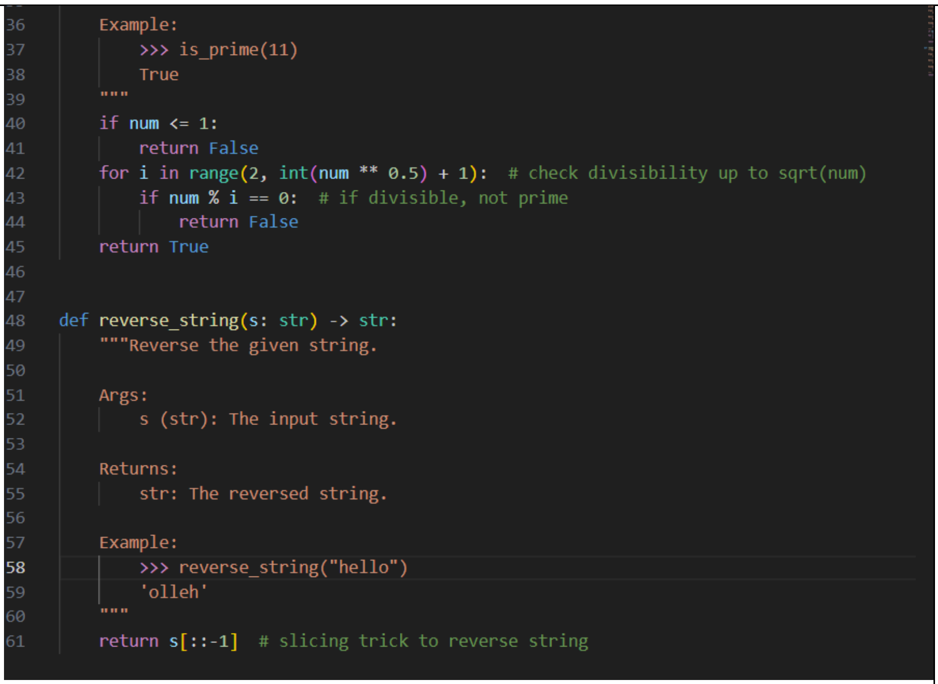
Keep the intent of the original comments intact.

Verify that the new documentation is clearer and more structured than the inline comments.

Code:

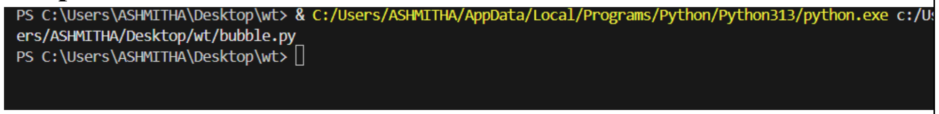


```python
def fibonacci(n: int) -> int:
    """Generate the nth Fibonacci number.

    The function uses a simple iterative approach instead of recursion
    for efficiency. The Fibonacci sequence starts with 0 and 1, and
    each subsequent number is the sum of the previous two.

    Args:
        n (int): The position in the Fibonacci sequence (0-indexed).

    Returns:
        int: The nth Fibonacci number.

    Example:
        >>> fibonacci(6)
        8
    """
    a, b = 0, 1  # initialize first two Fibonacci numbers
    for _ in range(n):  # loop n times
        a, b = b, a + b  # update values
    return a


def is_prime(num: int) -> bool:
    """Check if a number is prime.

    Uses trial division up to the square root of the number to check
    primality.

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

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

    Example:
        >>> is_prime(11)
```



```
    Example:
        >>> is_prime(11)
        True
    """
    if num <= 1:
        return False
    for i in range(2, int(num ** 0.5) + 1):  # check divisibility up to sqrt(num)
        if num % i == 0:  # if divisible, not prime
            return False
    return True


def reverse_string(s: str) -> str:
    """Reverse the given string.

    Args:
        s (str): The input string.

    Returns:
        str: The reversed string.

    Example:
        >>> reverse_string("hello")
        'olleh'
    """
    return s[::-1]  # slicing trick to reverse string
```

Output:



```
PS C:\Users\ASHMITHA\Desktop\wt> & C:/Users/ASHMITHA/AppData/Local/Programs/Python/Python313/python.exe c:/Us
ers/ASHMITHA/Desktop/wt/bubble.py
PS C:\Users\ASHMITHA\Desktop\wt> 
```

Observation: Inline Comments Removed

The inline comments in the input version explained each step separately.

In the output version, those details are consolidated into a structured Google-style docstring.

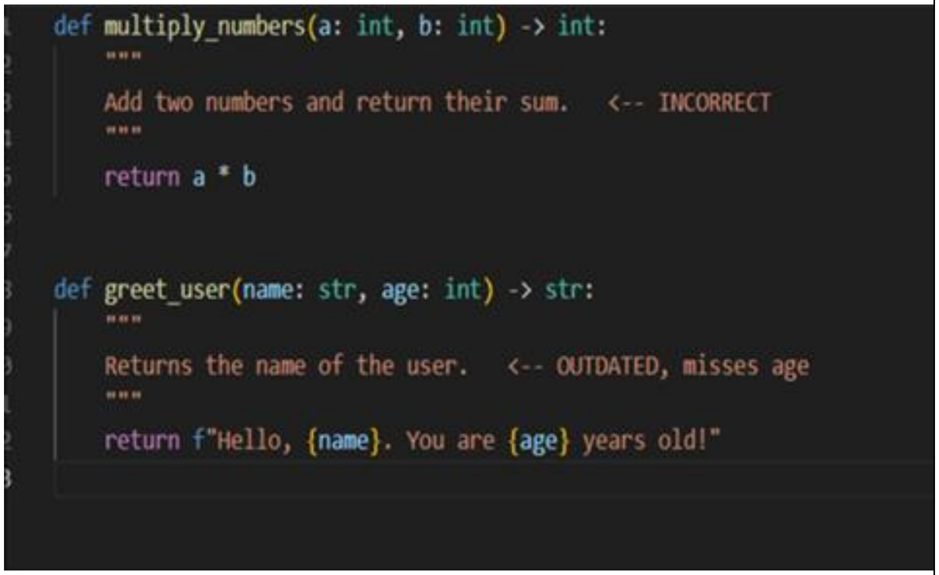
Improved Readability

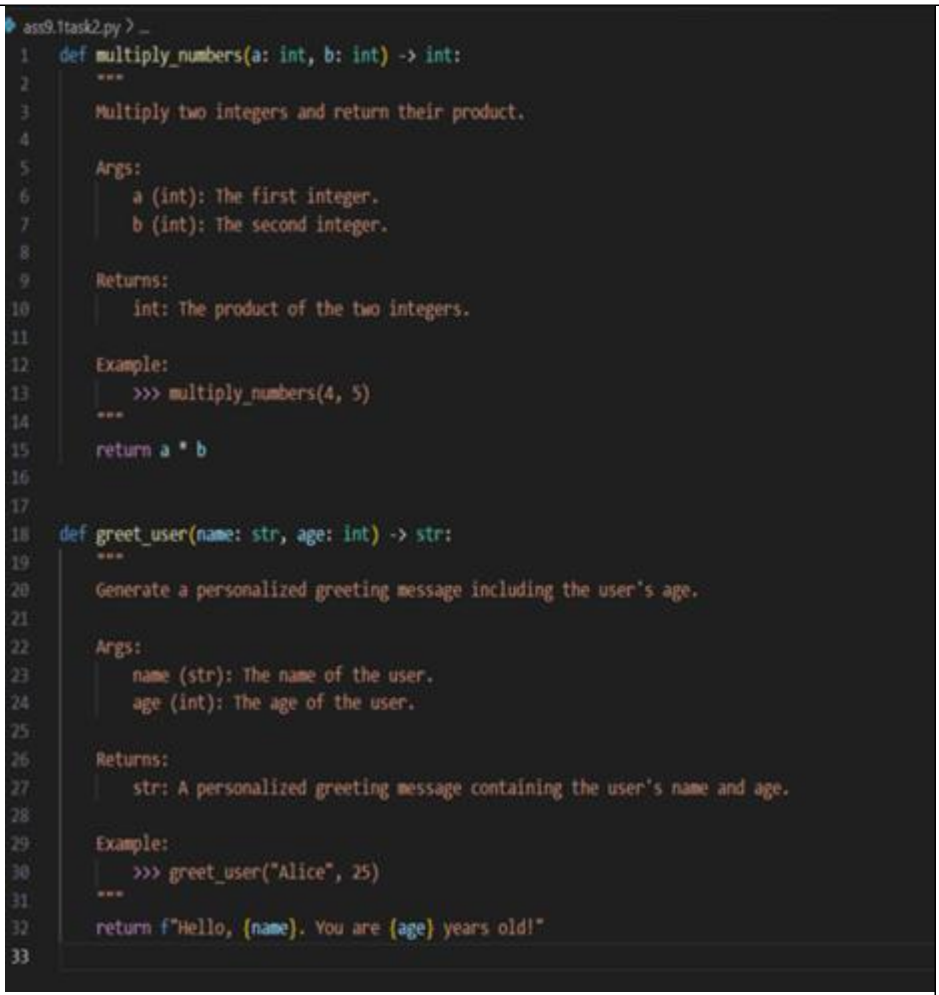
Instead of having scattered comments, the function has a single entry-point docstring.

Readers understand the purpose and logic without scanning multiple lines.

Task Description #5 (Documentation – Review and Correct Docstrings)

- Task: Use AI to identify and correct inaccuracies in existing docstrings.
- Instructions:

	◦ Provide Python code with outdated or incorrect docstrings.
	◦ Instruct AI to rewrite each docstring to match the current code behavior.
	◦ Ensure corrections follow Google-style formatting.
	• Expected Output #5:
	◦ Python file with updated, accurate, and standardized docstrings.
	PROMPT: Identify and correct inaccuracies in existing Python function docstrings. Instructions: • Review the provided Python file with outdated or incorrect docstrings. • Update each docstring so it accurately reflects the current code behavior. • Ensure all updated docstrings follow Google-style formatting and include: o Function description o Parameters with type hints o Return values with type hints o Example usage • Do not change the function's logic, only the docstrings.
	CODE: (with inaccurate docstrings)
	
	`def multiply_numbers(a: int, b: int) -> int:`
	`    """`
	`    Add two numbers and return their sum.   <-- INCORRECT`
	`    """`
	`    return a * b`
	
	`def greet_user(name: str, age: int) -> str:`
	`    """`
	`    Returns the name of the user.   <-- OUTDATED, misses age`
	`    """`
	`    return f"Hello, {name}. You are {age} years old!"`
	CODE: (with corrected docstrings)	

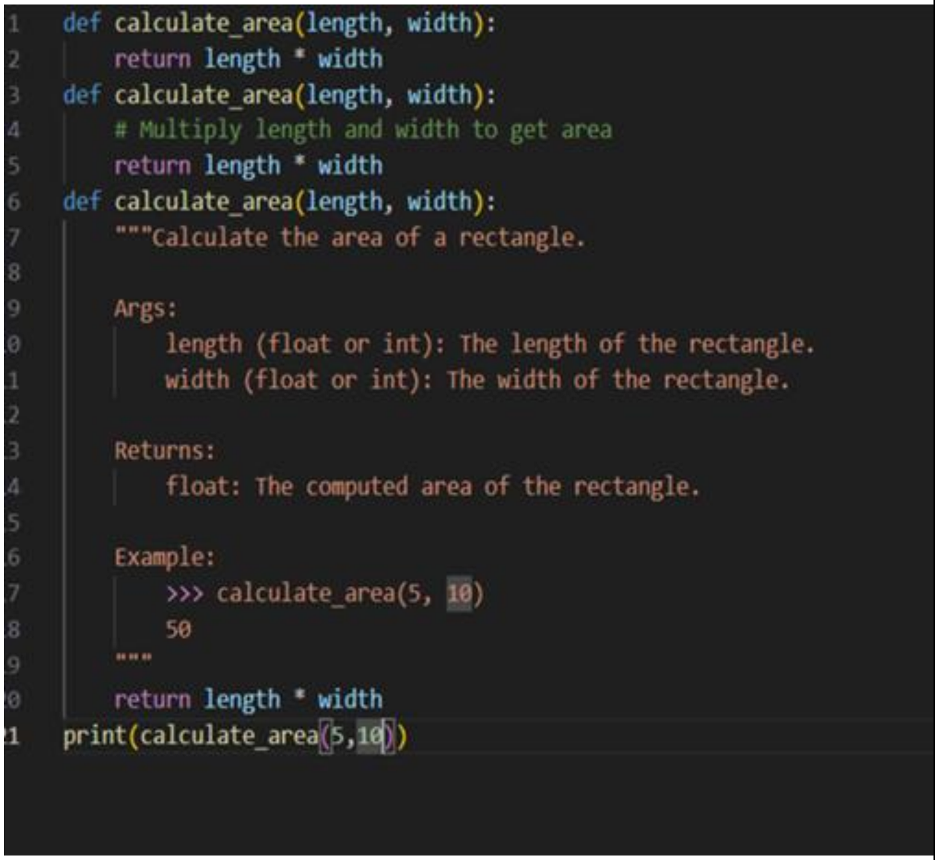


```python
def multiply_numbers(a: int, b: int) -> int:
    """
    Multiply two integers and return their product.

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

    Returns:
        int: The product of the two integers.

    Example:
        >>> multiply_numbers(4, 5)
    """
    return a * b


def greet_user(name: str, age: int) -> str:
    """
    Generate a personalized greeting message including the user's age.

    Args:
        name (str): The name of the user.
        age (int): The age of the user.

    Returns:
        str: A personalized greeting message containing the user's name and age.

    Example:
        >>> greet_user("Alice", 25)
    """
    return f"Hello, {name}. You are {age} years old!"
```

Observation:
Inline Comments Removed The inline comments in the input version explained each step separately. In the output version, those details are consolidated into a structured Google-style docstring. Improved Readability Instead of having scattered comments, the function has a single entry point docstring. Readers understand the purpose and logic without scanning multiple lines.

Task Description #6 (Documentation – Prompt Comparison Experiment)

- Task: Compare documentation output from a vague prompt and a detailed prompt for the same Python function.
- Instructions:
  - Create two prompts: one simple ("Add comments to this function") and one detailed ("Add Google-style docstrings with parameters, return types, and examples").
  - Use AI to process the same Python function with both

	prompts. ◦ Analyze and record differences in quality, accuracy, and completeness. • Expected Output #6: ◦ A comparison table showing the results from both prompts with observations. PROMPT: • Vague Prompt Simulation: Pretend the user asked only: "Add comments to this function." Provide the output for this vague request. • Detailed Prompt Simulation: Pretend the user asked instead: "Add Google-style docstrings with parameters, return types, and examples." Provide the output for this detailed request. • Comparison Task: Create a comparison table showing the outputs from both prompts side by side. Include observations on quality, accuracy, and completeness. **CODE(VAGUE PROMPT):**  See code below. **Output:no** **Observation:** **• A vague prompt gives minimal results (single comment, lacks structure). • A detailed prompt produces high-quality, structured documentation (parameters, return types, examples). • Detailed**	

```python
def calculate_area(length, width):
    return length * width
def calculate_area(length, width):
    # Multiply length and width to get area
    return length * width
def calculate_area(length, width):
    """Calculate the area of a rectangle.

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

    Returns:
        float: The computed area of the rectangle.

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

	prompting is especially important in collaborative coding, large projects, and generating professional docs.	