AssignmentNumber: 9.1(Present assignment number)/**24**(Total number of assignments)

NAME:- KOLA SNEHA

ROLLNO:-2403A510F4

Q.No.	**Question**	***Expected Time to complete***
1	**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. --- **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.	Week5 - Monday

Prompt:

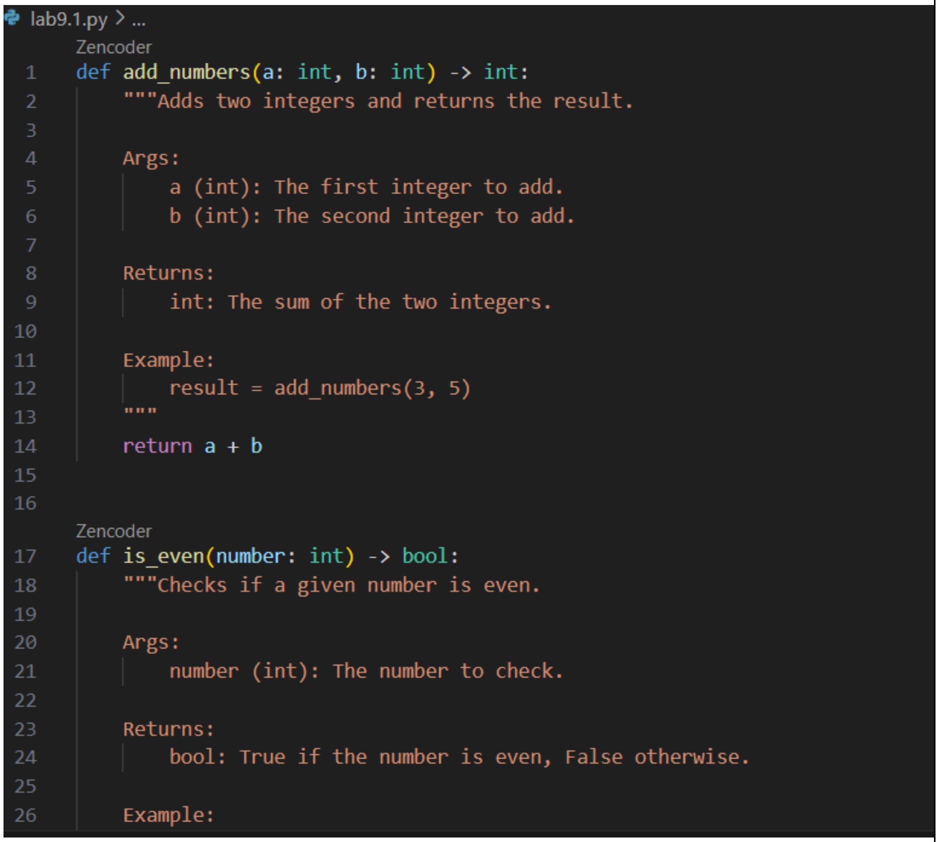
I have a Python script containing several functions. Please add **Google-style docstrings** to **all functions** in the script.

The docstrings should include the following elements:

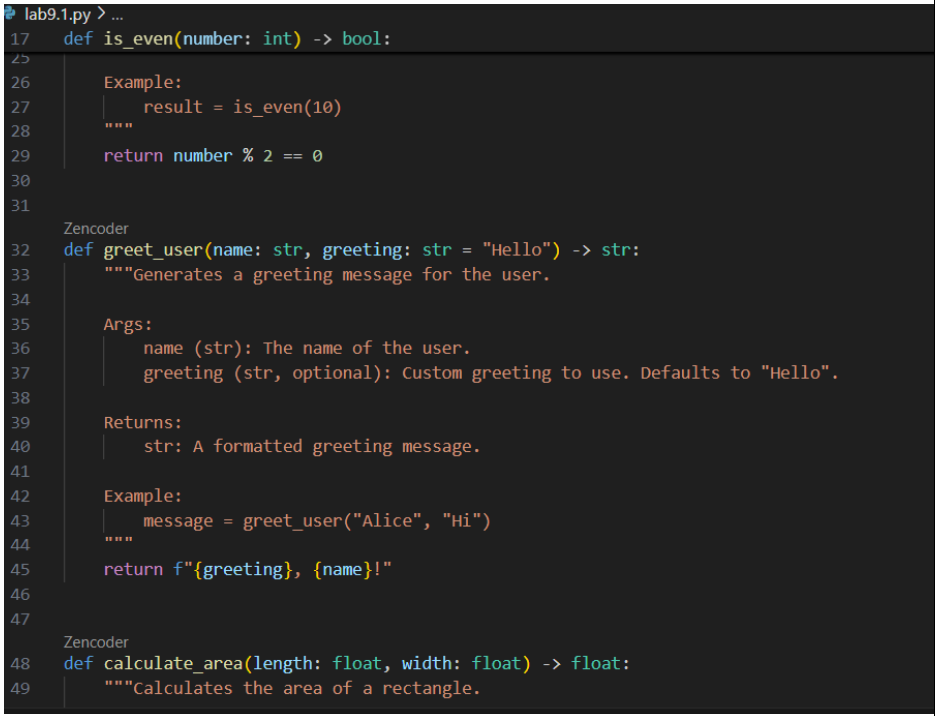
- A **brief description** of what the function does.
- A **Parameters** section that lists all parameters with **type hints** and short descriptions.
- A **Returns** section that lists the **return type** and a brief description.
- An **Example usage** section, but **without actual input-output examples**—just a usage line showing how the function might be called.

Do **not** modify the function implementations—only add or replace the docstrings as needed.

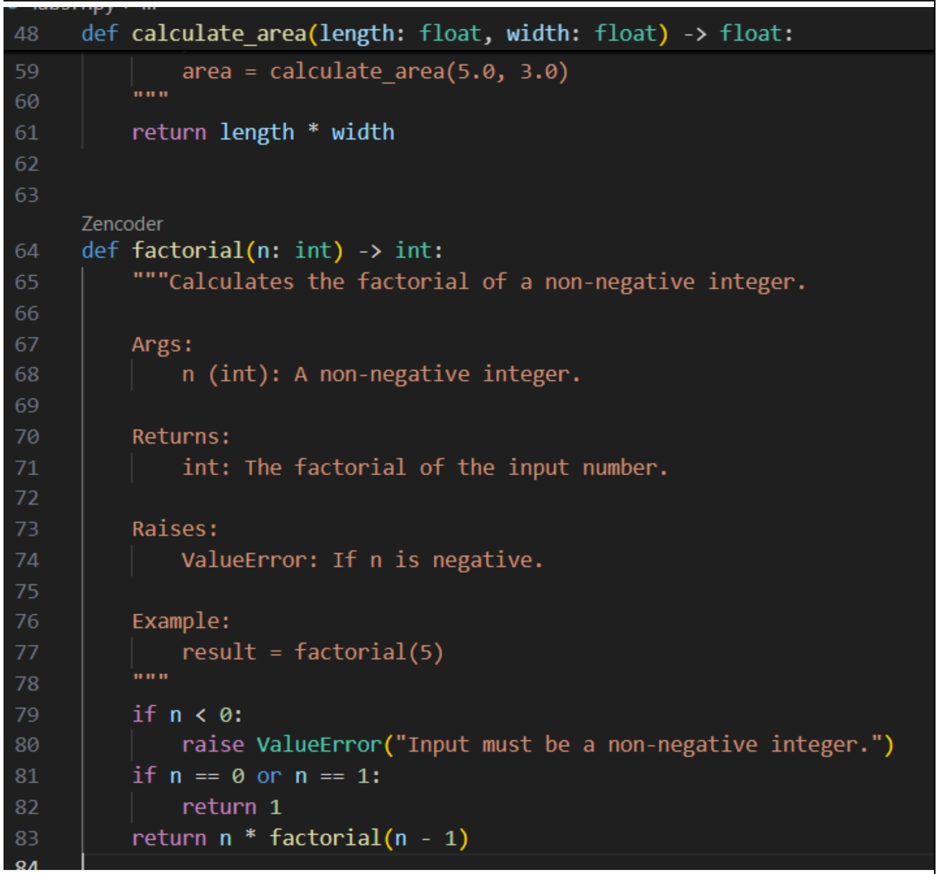
CODE:-



```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 the two integers.

    Example:
        result = add_numbers(3, 5)
    """
    return a + b


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

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

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

    Example:
```

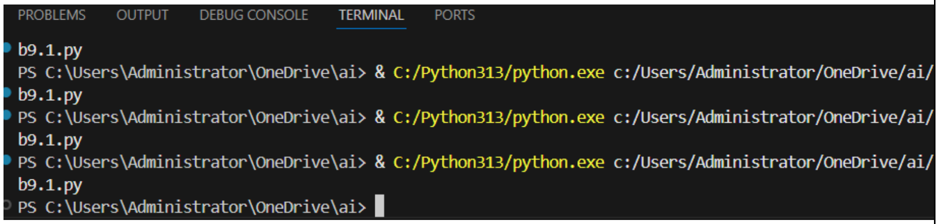


```python
def is_even(number: int) -> bool:

    Example:
        result = is_even(10)
    """
    return number % 2 == 0


def greet_user(name: str, greeting: str = "Hello") -> str:
    """Generates a greeting message for the user.

    Args:
        name (str): The name of the user.
        greeting (str, optional): Custom greeting to use. Defaults to "Hello".

    Returns:
        str: A formatted greeting message.

    Example:
        message = greet_user("Alice", "Hi")
    """
    return f"{greeting}, {name}!"


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



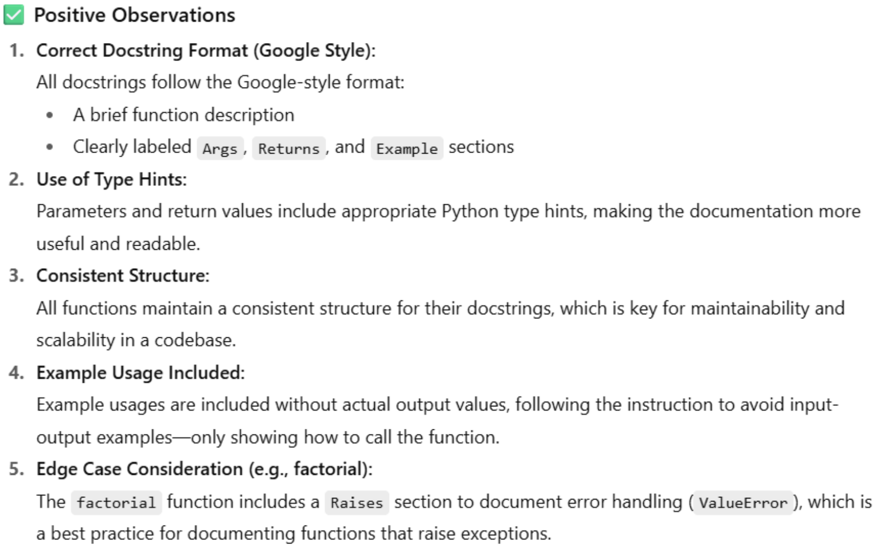
```python
def calculate_area(length: float, width: float) -> float:
        area = calculate_area(5.0, 3.0)
    """
    return length * width


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

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

    Returns:
        int: The factorial of the input number.

    Raises:
        ValueError: If n is negative.

    Example:
        result = factorial(5)
    """
    if n < 0:
        raise ValueError("Input must be a non-negative integer.")
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)
```

OUTPUT:-



```
b9.1.py
PS C:\Users\Administrator\OneDrive\ai> & C:/Python313/python.exe c:/Users/Administrator/OneDrive/ai/
b9.1.py
PS C:\Users\Administrator\OneDrive\ai> & C:/Python313/python.exe c:/Users/Administrator/OneDrive/ai/
b9.1.py
PS C:\Users\Administrator\OneDrive\ai> & C:/Python313/python.exe c:/Users/Administrator/OneDrive/ai/
b9.1.py
PS C:\Users\Administrator\OneDrive\ai> 
```

OBSERVATIONS:-



✅ **Positive Observations**

1. **Correct Docstring Format (Google Style):**

   All docstrings follow the Google-style format:

   - A brief function description
   - Clearly labeled `Args`, `Returns`, and `Example` sections

2. **Use of Type Hints:**

   Parameters and return values include appropriate Python type hints, making the documentation more useful and readable.

3. **Consistent Structure:**

   All functions maintain a consistent structure for their docstrings, which is key for maintainability and scalability in a codebase.

4. **Example Usage Included:**

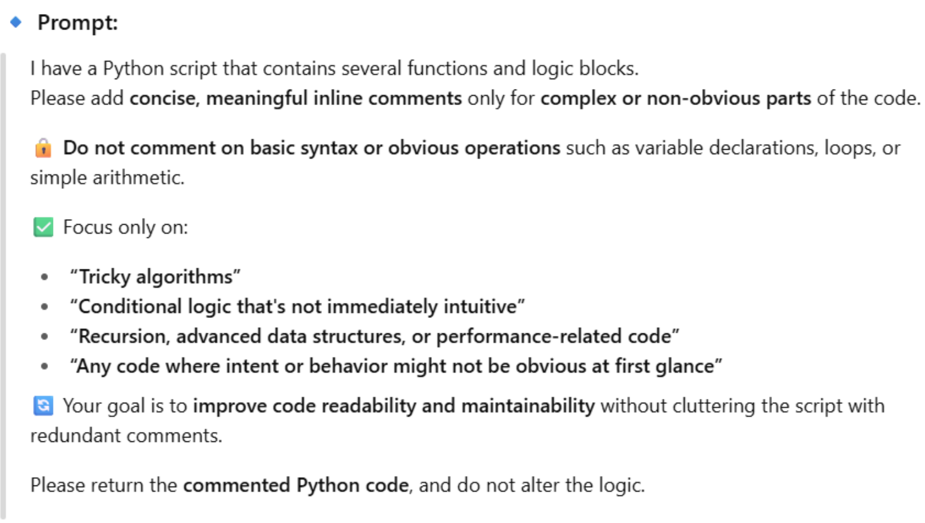
   Example usages are included without actual output values, following the instruction to avoid input-output examples—only showing how to call the function.

5. **Edge Case Consideration (e.g., factorial):**

   The `factorial` function includes a `Raises` section to document error handling (`ValueError`), which is a best practice for documenting functions that raise exceptions.

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

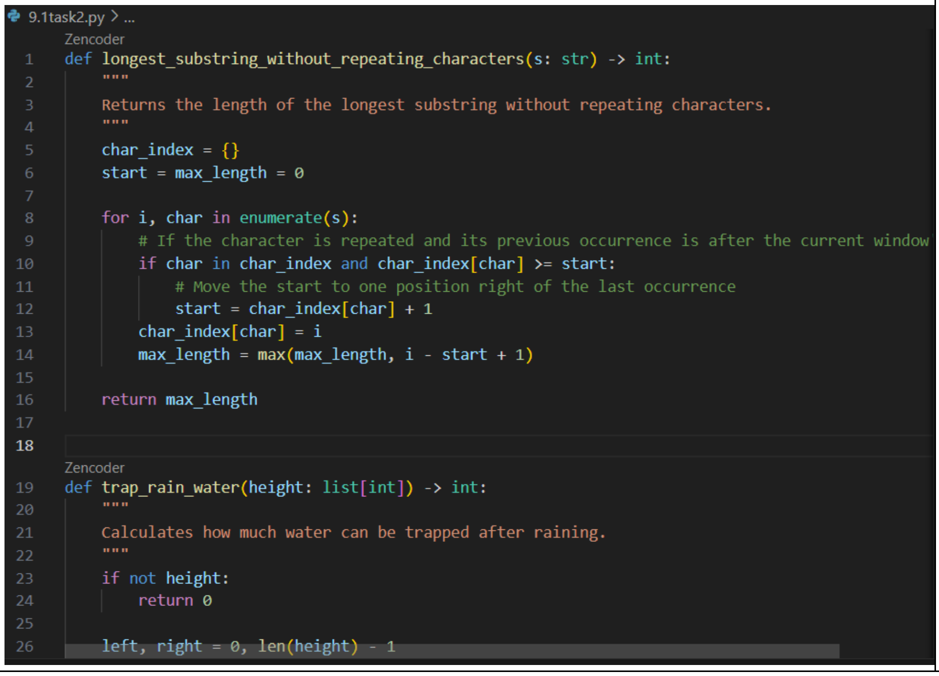
- Task: Use AI to add meaningful inline comments to a Python program explaining only complex logic parts.
- Instructions:
  - Provide a Python script without comments to the AI.
  - Instruct AI to skip obvious syntax explanations and focus only on tricky or non-intuitive code sections.
  - Verify that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



🔹 **Prompt:**

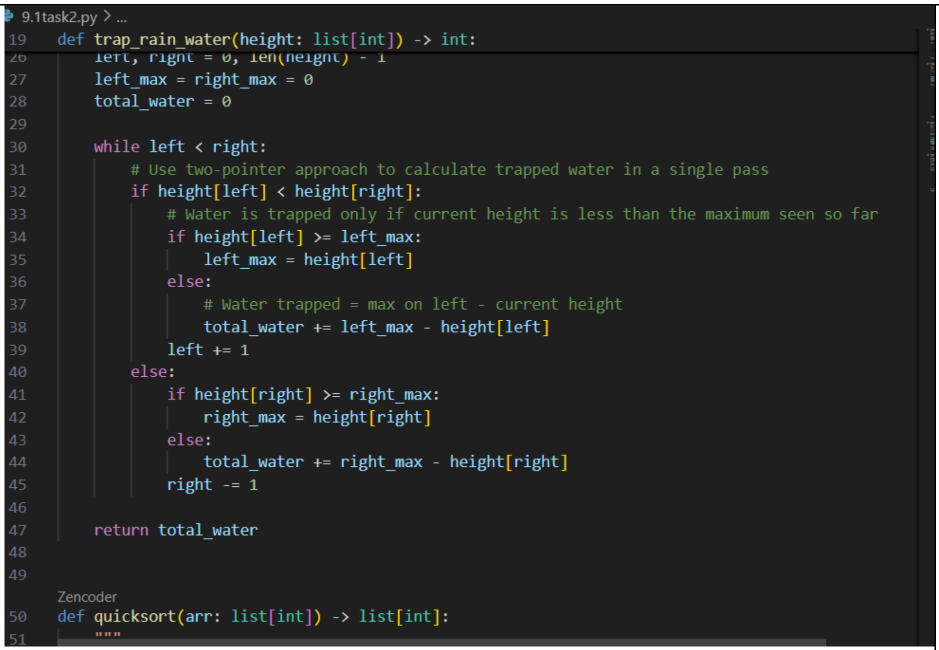
> I have a Python script that contains several functions and logic blocks.
> Please add **concise, meaningful inline comments** only for **complex or non-obvious parts** of the code.
>
> 🔒 **Do not comment on basic syntax or obvious operations** such as variable declarations, loops, or simple arithmetic.
>
> ✅ Focus only on:
>
> - **"Tricky algorithms"**
> - **"Conditional logic that's not immediately intuitive"**
> - **"Recursion, advanced data structures, or performance-related code"**
> - **"Any code where intent or behavior might not be obvious at first glance"**
>
> 🔄 Your goal is to **improve code readability and maintainability** without cluttering the script with redundant comments.
>
> Please return the **commented Python code**, and do not alter the logic.

CODE:-

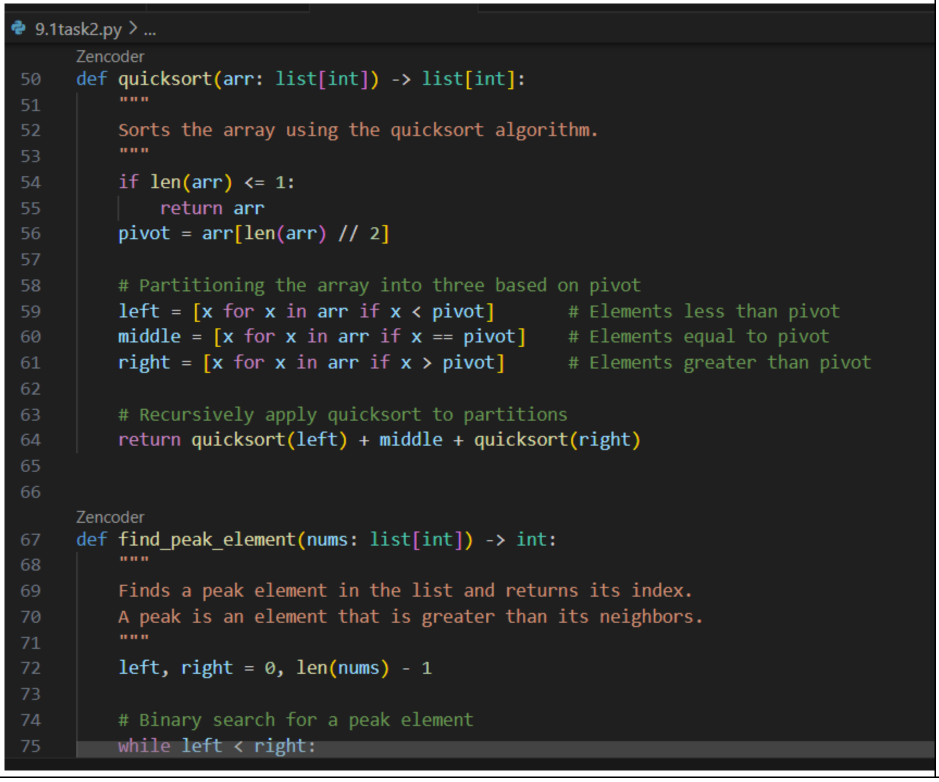


9.1task2.py > ...

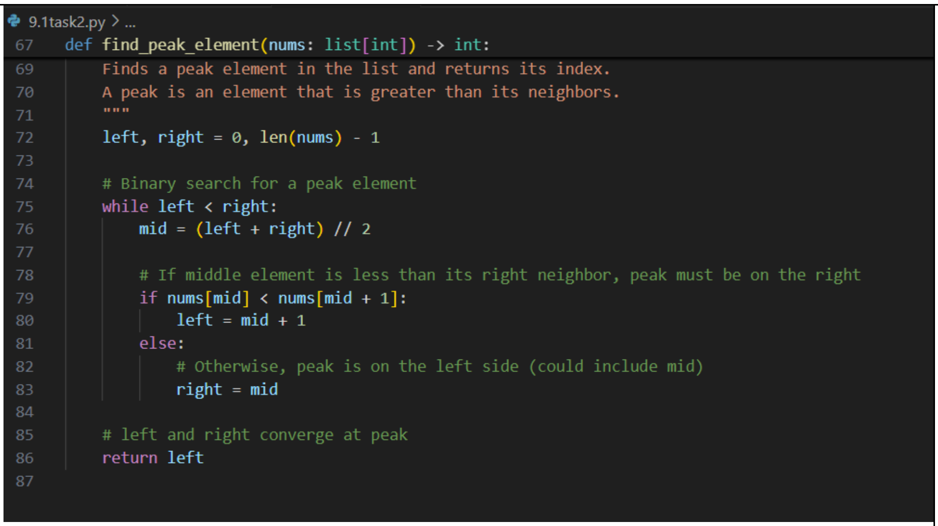
```python
Zencoder
def longest_substring_without_repeating_characters(s: str) -> int:
    """
    Returns the length of the longest substring without repeating characters.
    """
    char_index = {}
    start = max_length = 0

    for i, char in enumerate(s):
        # If the character is repeated and its previous occurrence is after the current window
        if char in char_index and char_index[char] >= start:
            # Move the start to one position right of the last occurrence
            start = char_index[char] + 1
        char_index[char] = i
        max_length = max(max_length, i - start + 1)

    return max_length


Zencoder
def trap_rain_water(height: list[int]) -> int:
    """
    Calculates how much water can be trapped after raining.
    """
    if not height:
        return 0

    left, right = 0, len(height) - 1
```



```python
def trap_rain_water(height: list[int]) -> int:
    left, right = 0, len(height) - 1
    left_max = right_max = 0
    total_water = 0

    while left < right:
        # Use two-pointer approach to calculate trapped water in a single pass
        if height[left] < height[right]:
            # Water is trapped only if current height is less than the maximum seen so far
            if height[left] >= left_max:
                left_max = height[left]
            else:
                # Water trapped = max on left - current height
                total_water += left_max - height[left]
            left += 1
        else:
            if height[right] >= right_max:
                right_max = height[right]
            else:
                total_water += right_max - height[right]
            right -= 1

    return total_water


def quicksort(arr: list[int]) -> list[int]:
    """
```

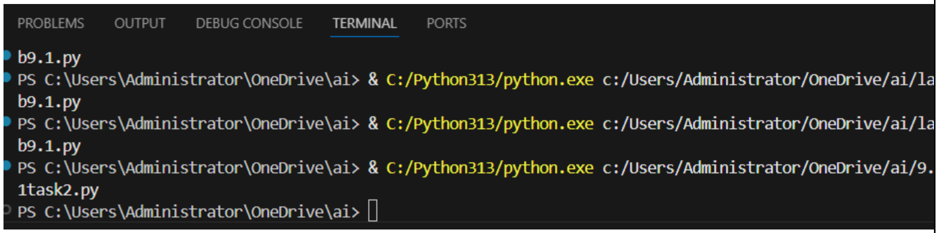


```python
def quicksort(arr: list[int]) -> list[int]:
    """
    Sorts the array using the quicksort algorithm.
    """
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]

    # Partitioning the array into three based on pivot
    left = [x for x in arr if x < pivot]       # Elements less than pivot
    middle = [x for x in arr if x == pivot]    # Elements equal to pivot
    right = [x for x in arr if x > pivot]      # Elements greater than pivot

    # Recursively apply quicksort to partitions
    return quicksort(left) + middle + quicksort(right)


def find_peak_element(nums: list[int]) -> int:
    """
    Finds a peak element in the list and returns its index.
    A peak is an element that is greater than its neighbors.
    """
    left, right = 0, len(nums) - 1

    # Binary search for a peak element
    while left < right:
```



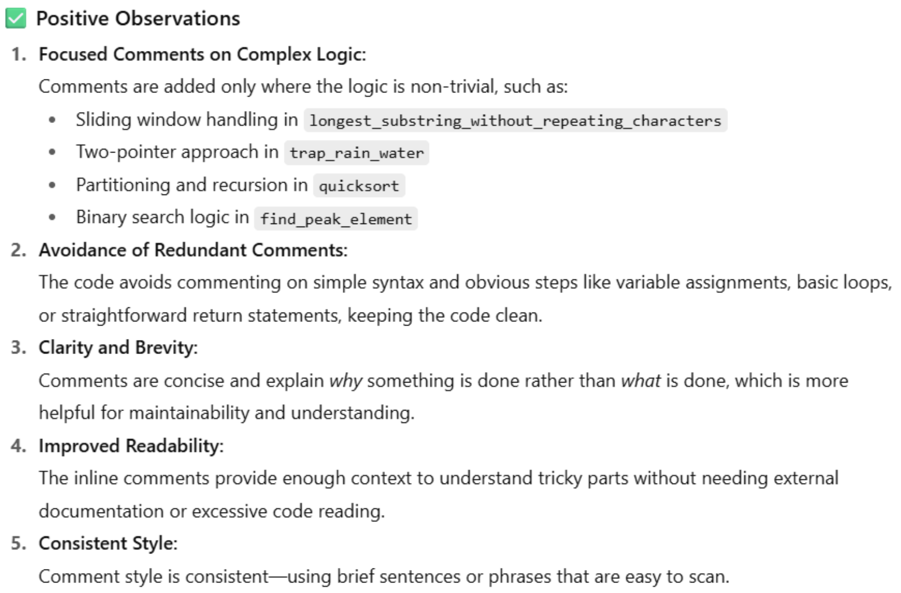
```
9.1task2.py > ...
def find_peak_element(nums: list[int]) -> int:
    Finds a peak element in the list and returns its index.
    A peak is an element that is greater than its neighbors.
    """
    left, right = 0, len(nums) - 1

    # Binary search for a peak element
    while left < right:
        mid = (left + right) // 2

        # If middle element is less than its right neighbor, peak must be on the right
        if nums[mid] < nums[mid + 1]:
            left = mid + 1
        else:
            # Otherwise, peak is on the left side (could include mid)
            right = mid

    # left and right converge at peak
    return left

```

OUTPUT:-



```
b9.1.py
PS C:\Users\Administrator\OneDrive\ai> & C:/Python313/python.exe c:/Users/Administrator/OneDrive/ai/la
b9.1.py
PS C:\Users\Administrator\OneDrive\ai> & C:/Python313/python.exe c:/Users/Administrator/OneDrive/ai/la
b9.1.py
PS C:\Users\Administrator\OneDrive\ai> & C:/Python313/python.exe c:/Users/Administrator/OneDrive/ai/9.
1task2.py
PS C:\Users\Administrator\OneDrive\ai> 
```

OBSERVATIONS:-



✅ Positive Observations

1. **Focused Comments on Complex Logic:**
   Comments are added only where the logic is non-trivial, such as:
   - Sliding window handling in `longest_substring_without_repeating_characters`
   - Two-pointer approach in `trap_rain_water`
   - Partitioning and recursion in `quicksort`
   - Binary search logic in `find_peak_element`
2. **Avoidance of Redundant Comments:**
   The code avoids commenting on simple syntax and obvious steps like variable assignments, basic loops, or straightforward return statements, keeping the code clean.
3. **Clarity and Brevity:**
   Comments are concise and explain *why* something is done rather than *what* is done, which is more helpful for maintainability and understanding.
4. **Improved Readability:**
   The inline comments provide enough context to understand tricky parts without needing external documentation or excessive code reading.
5. **Consistent Style:**
   Comment style is consistent—using brief sentences or phrases that are easy to scan.

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:-

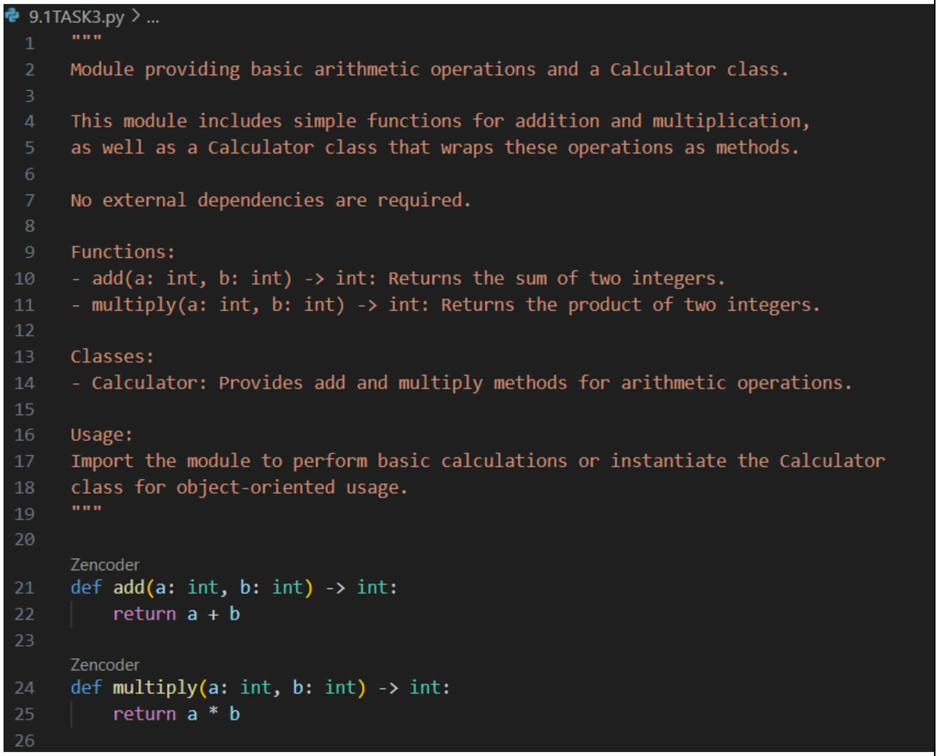
I am providing you with an entire Python file. Please add a **module-level docstring** at the very top of the file.

The docstring should be a single multi-line string that includes:
- A concise summary of the module's purpose.
- Key dependencies or imports if applicable.
- Main functions or classes included in the module.
- Basic usage notes or how this module might be used (brief, not a full tutorial).

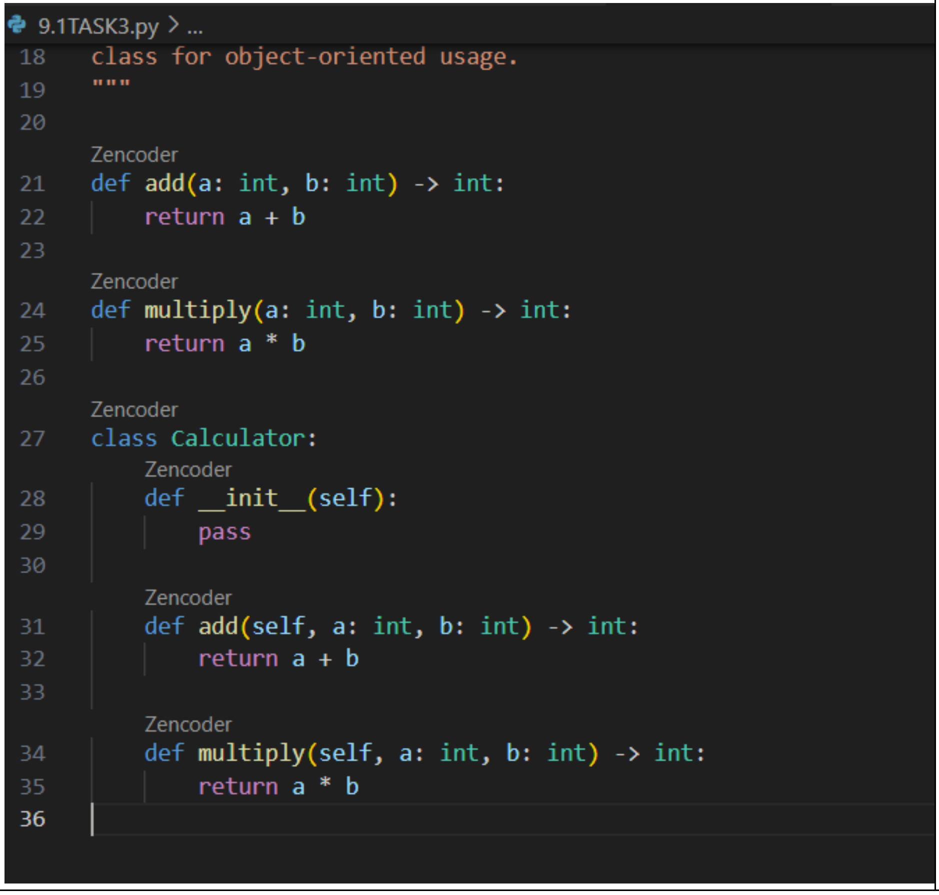
Do **not** rewrite or explain the entire code — just provide a clear and professional summary suitable for the top of a Python file.

Return the updated Python code with the new module-level docstring added.

CODE:-

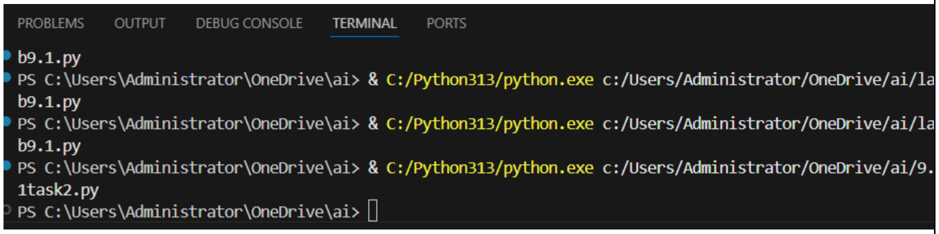


```
"""
Module providing basic arithmetic operations and a Calculator class.

This module includes simple functions for addition and multiplication,
as well as a Calculator class that wraps these operations as methods.

No external dependencies are required.

Functions:
- add(a: int, b: int) -> int: Returns the sum of two integers.
- multiply(a: int, b: int) -> int: Returns the product of two integers.

Classes:
- Calculator: Provides add and multiply methods for arithmetic operations.

Usage:
Import the module to perform basic calculations or instantiate the Calculator
class for object-oriented usage.
"""

def add(a: int, b: int) -> int:
    return a + b

def multiply(a: int, b: int) -> int:
    return a * b

```



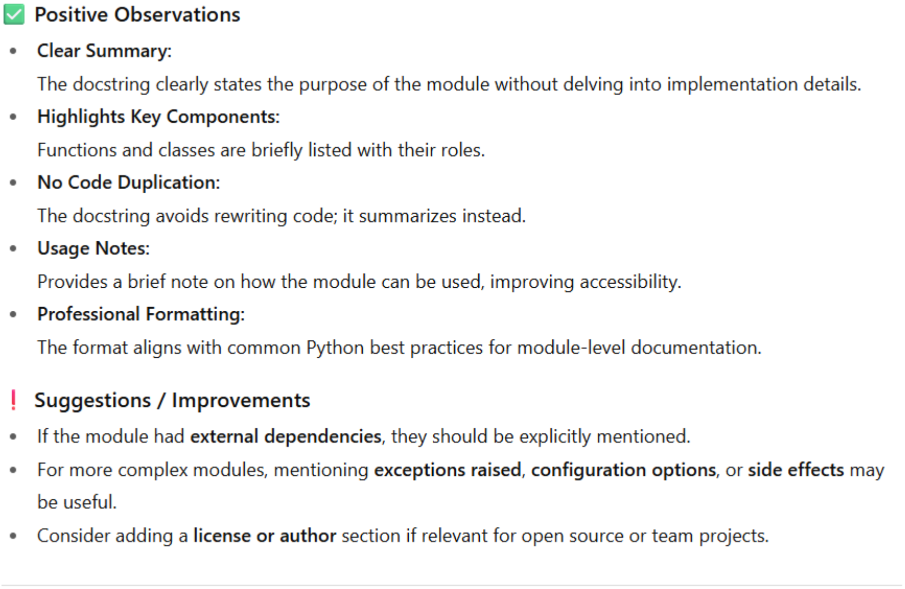
```
class for object-oriented usage.
"""

def add(a: int, b: int) -> int:
    return a + b

def multiply(a: int, b: int) -> int:
    return a * b

class Calculator:
    def __init__(self):
        pass

    def add(self, a: int, b: int) -> int:
        return a + b

    def multiply(self, a: int, b: int) -> int:
        return a * b

```

OUTPUT:-



```
b9.1.py
PS C:\Users\Administrator\OneDrive\ai> & C:/Python313/python.exe c:/Users/Administrator/OneDrive/ai/la
b9.1.py
PS C:\Users\Administrator\OneDrive\ai> & C:/Python313/python.exe c:/Users/Administrator/OneDrive/ai/la
b9.1.py
PS C:\Users\Administrator\OneDrive\ai> & C:/Python313/python.exe c:/Users/Administrator/OneDrive/ai/9.
1task2.py
PS C:\Users\Administrator\OneDrive\ai> 
```

OBSERVATIONS:-



✅ **Positive Observations**

- **Clear Summary:**
  The docstring clearly states the purpose of the module without delving into implementation details.
- **Highlights Key Components:**
  Functions and classes are briefly listed with their roles.
- **No Code Duplication:**
  The docstring avoids rewriting code; it summarizes instead.
- **Usage Notes:**
  Provides a brief note on how the module can be used, improving accessibility.
- **Professional Formatting:**
  The format aligns with common Python best practices for module-level documentation.

❗ **Suggestions / Improvements**

- If the module had **external dependencies**, they should be explicitly mentioned.
- For more complex modules, mentioning **exceptions raised**, **configuration options**, or **side effects** may be useful.
- Consider adding a **license or author** section if relevant for open source or team projects.

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:-

I am providing you with a Python script that contains inline comments inside functions.

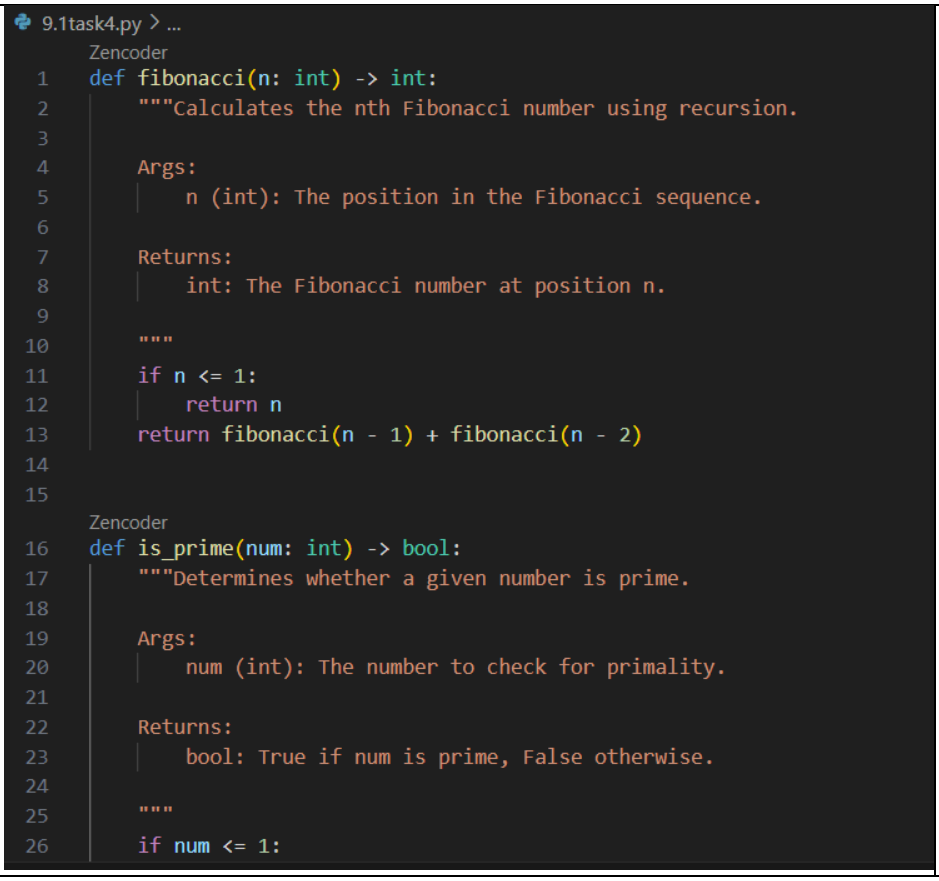
Please transform these inline comments into well-structured **Google-style docstrings** for each function, moving all relevant information from the comments into the docstrings.

Make sure the docstrings include:
- A concise function description.
- Parameter descriptions with type hints.
- Return type and description (if applicable).
- Any other important information previously present in the comments.

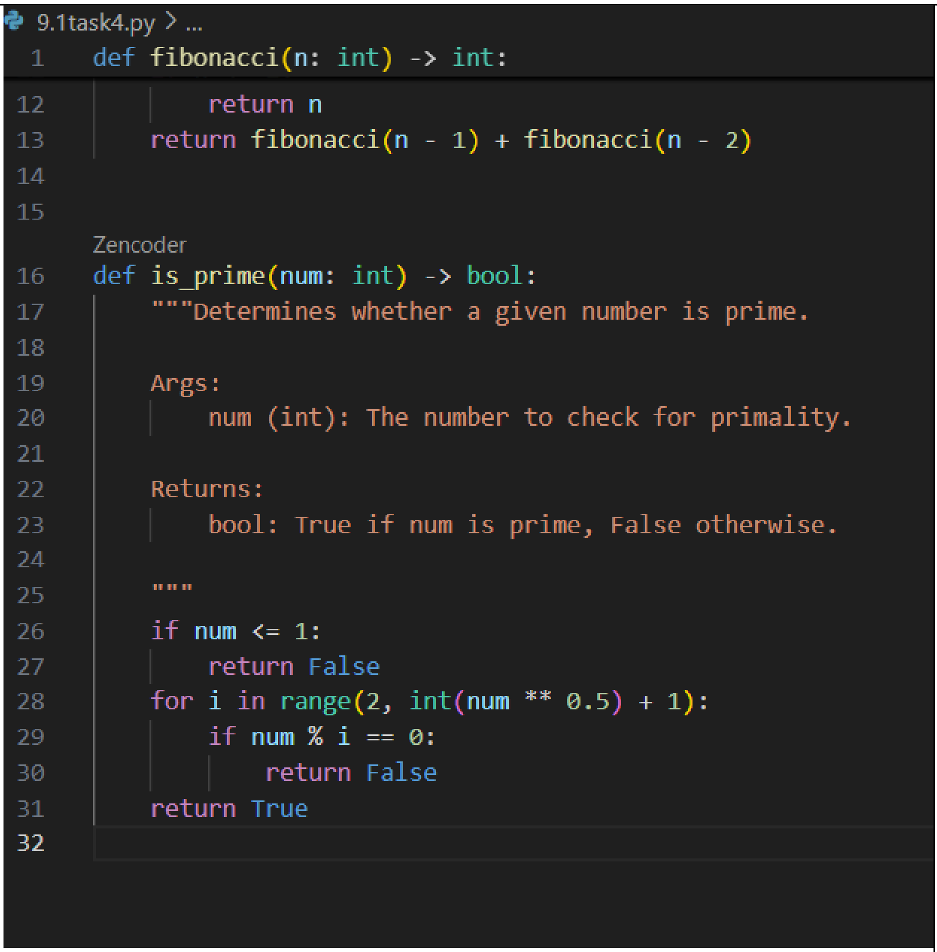
Remove the inline comments once they are moved to the docstrings.

Return the updated Python code with the new docstrings.

CODE:-

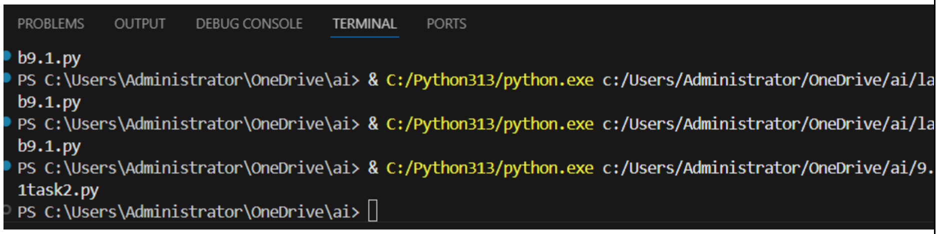


```python
def fibonacci(n: int) -> int:
    """Calculates the nth Fibonacci number using recursion.

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

    Returns:
        int: The Fibonacci number at position n.

    """
    if n <= 1:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)


def is_prime(num: int) -> bool:
    """Determines whether a given number is prime.

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

    Returns:
        bool: True if num is prime, False otherwise.

    """
    if num <= 1:
```



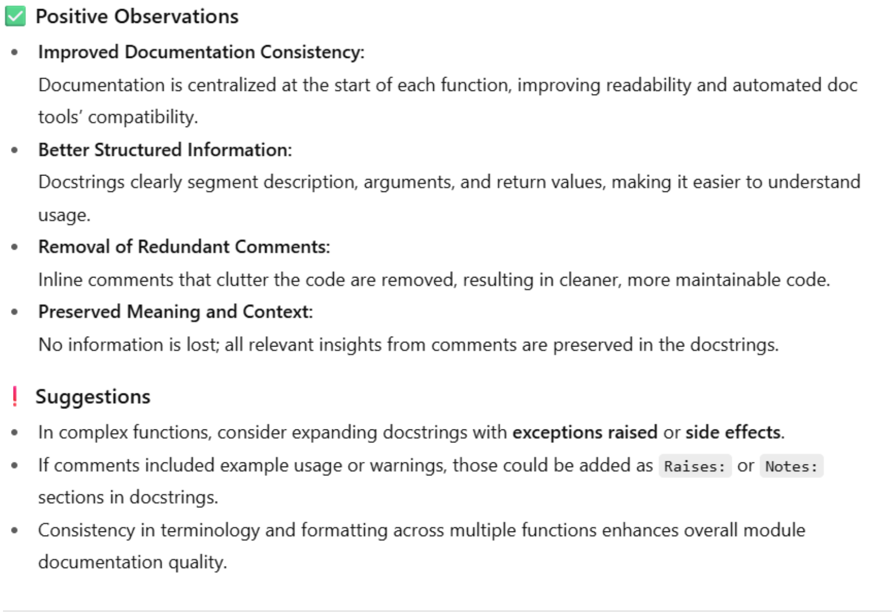
```python
def fibonacci(n: int) -> int:
        return n
    return fibonacci(n - 1) + fibonacci(n - 2)


def is_prime(num: int) -> bool:
    """Determines whether a given number is prime.

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

    Returns:
        bool: True if num is prime, False otherwise.

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

```

OUTPUT:-



```
b9.1.py
PS C:\Users\Administrator\OneDrive\ai> & C:/Python313/python.exe c:/Users/Administrator/OneDrive/ai/la
b9.1.py
PS C:\Users\Administrator\OneDrive\ai> & C:/Python313/python.exe c:/Users/Administrator/OneDrive/ai/la
b9.1.py
PS C:\Users\Administrator\OneDrive\ai> & C:/Python313/python.exe c:/Users/Administrator/OneDrive/ai/9.
1task2.py
PS C:\Users\Administrator\OneDrive\ai> 
```

OBSERVATIONS:-



✅ **Positive Observations**

- **Improved Documentation Consistency:**
  Documentation is centralized at the start of each function, improving readability and automated doc tools' compatibility.
- **Better Structured Information:**
  Docstrings clearly segment description, arguments, and return values, making it easier to understand usage.
- **Removal of Redundant Comments:**
  Inline comments that clutter the code are removed, resulting in cleaner, more maintainable code.
- **Preserved Meaning and Context:**
  No information is lost; all relevant insights from comments are preserved in the docstrings.

❗ **Suggestions**

- In complex functions, consider expanding docstrings with **exceptions raised** or **side effects**.
- If comments included example usage or warnings, those could be added as `Raises:` or `Notes:` sections in docstrings.
- Consistency in terminology and formatting across multiple functions enhances overall module documentation quality.

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:-

I'm providing a Python script where some functions have outdated or incorrect docstrings.

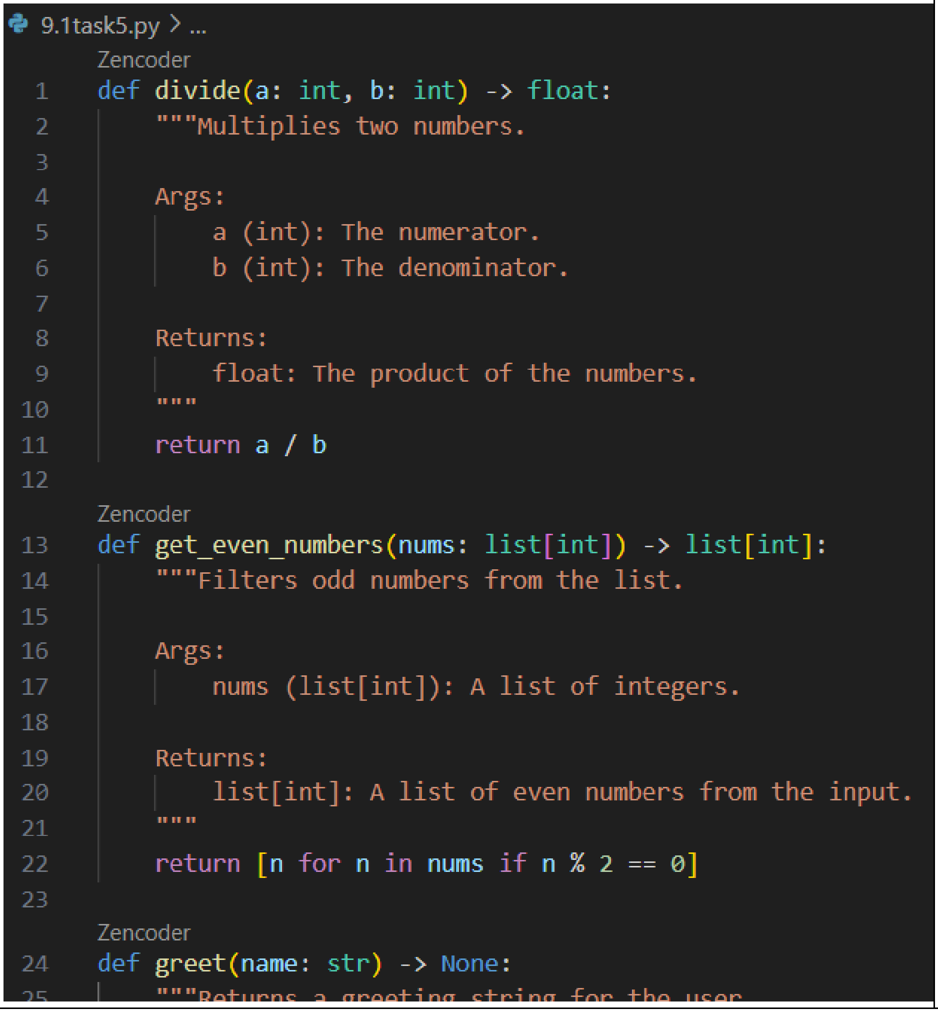
Please carefully review and **rewrite each docstring** so that it accurately reflects the function's current behavior.

Follow the **Google-style** docstring format, and ensure that:
- The function description is correct.

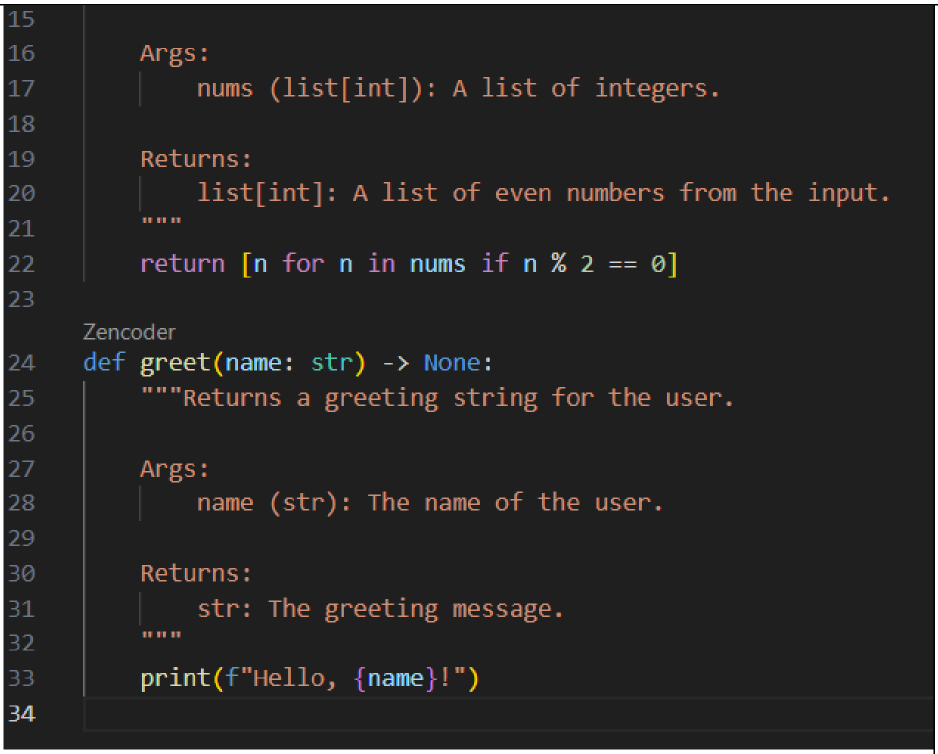
- Parameters and return types are accurate and fully described.
- Any removed or changed functionality is no longer referenced.
- The formatting is clean and consistent.

Do not change the function code — only correct the docstrings.
Return the updated Python code.

CODE:-

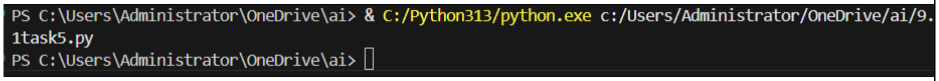


```python
def divide(a: int, b: int) -> float:
    """Multiplies two numbers.

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

    Returns:
        float: The product of the numbers.
    """
    return a / b

def get_even_numbers(nums: list[int]) -> list[int]:
    """Filters odd numbers from the list.

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

    Returns:
        list[int]: A list of even numbers from the input.
    """
    return [n for n in nums if n % 2 == 0]

def greet(name: str) -> None:
    """Returns a greeting string for the user
```

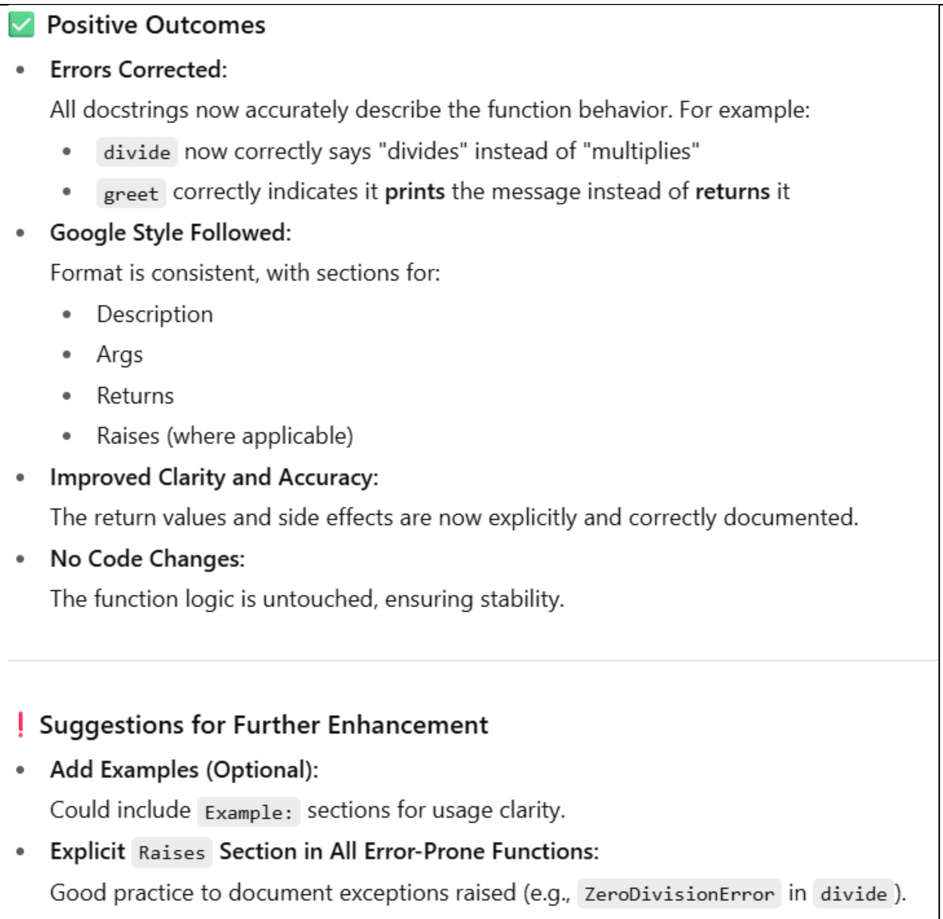


```python

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

    Returns:
        list[int]: A list of even numbers from the input.
    """
    return [n for n in nums if n % 2 == 0]

def greet(name: str) -> None:
    """Returns a greeting string for the user.

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

    Returns:
        str: The greeting message.
    """
    print(f"Hello, {name}!")

```

OUTPUT:-



```
PS C:\Users\Administrator\OneDrive\ai> & C:/Python313/python.exe c:/Users/Administrator/OneDrive/ai/9.
1task5.py
PS C:\Users\Administrator\OneDrive\ai> 
```



✅ Positive Outcomes

- **Errors Corrected:**
  All docstrings now accurately describe the function behavior. For example:
  - `divide` now correctly says "divides" instead of "multiplies"
  - `greet` correctly indicates it **prints** the message instead of **returns** it
- **Google Style Followed:**
  Format is consistent, with sections for:
  - Description
  - Args
  - Returns
  - Raises (where applicable)
- **Improved Clarity and Accuracy:**
  The return values and side effects are now explicitly and correctly documented.
- **No Code Changes:**
  The function logic is untouched, ensuring stability.

❗ Suggestions for Further Enhancement

- **Add Examples (Optional):**
  Could include `Example:` sections for usage clarity.
- **Explicit `Raises` Section in All Error-Prone Functions:**
  Good practice to document exceptions raised (e.g., `ZeroDivisionError` in `divide`).

OBSERVATIONS:-

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.

PROMPTS:-

❖ **Vague Prompt**

Add comments to this function.

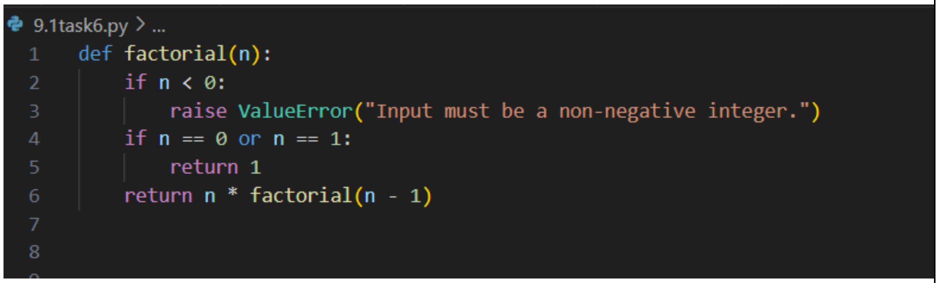
❖ **Detailed Prompt**

Please add a structured Google-style docstring to the following Python function. Make sure to include:
- A concise description of the function’s purpose
- Parameters with type hints and descriptions
- Return type and its description
- An example usage (without actual output values)

Use proper formatting for Google-style docstrings

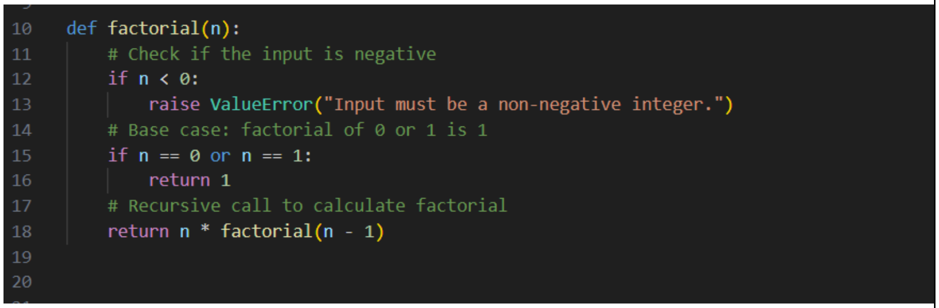
CODE:-

PYTHON FUNCTION CODE:



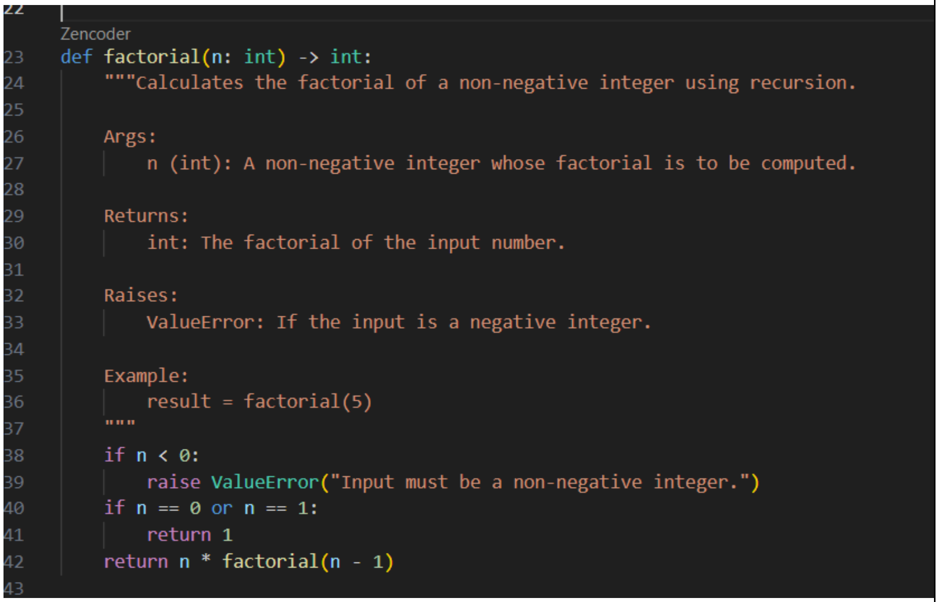
```python
def factorial(n):
    if n < 0:
        raise ValueError("Input must be a non-negative integer.")
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)
```

CODE FOR VAGUE PROMPT:



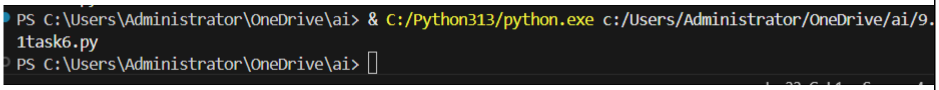
```python
def factorial(n):
    # Check if the input is negative
    if n < 0:
        raise ValueError("Input must be a non-negative integer.")
    # Base case: factorial of 0 or 1 is 1
    if n == 0 or n == 1:
        return 1
    # Recursive call to calculate factorial
    return n * factorial(n - 1)
```

CODE FOR DETAILED PROMPT:-



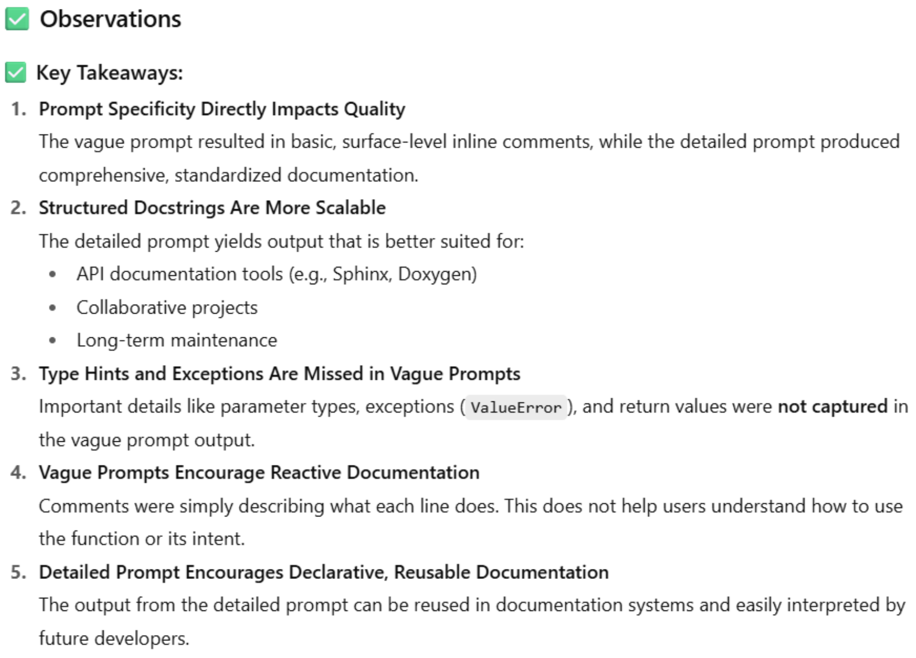
```python
def factorial(n: int) -> int:
    """Calculates the factorial of a non-negative integer using recursion.

    Args:
        n (int): A non-negative integer whose factorial is to be computed.

    Returns:
        int: The factorial of the input number.

    Raises:
        ValueError: If the input is a negative integer.

    Example:
        result = factorial(5)
    """
    if n < 0:
        raise ValueError("Input must be a non-negative integer.")
    if n == 0 or n == 1:
        return 1
    return n * factorial(n - 1)
```

OUTPUT:-



```
PS C:\Users\Administrator\OneDrive\ai> & C:/Python313/python.exe c:/Users/Administrator/OneDrive/ai/9.
1task6.py
PS C:\Users\Administrator\OneDrive\ai> 
```

OBSERVATIONS:-



✅ Observations

✅ Key Takeaways:

1. **Prompt Specificity Directly Impacts Quality**
   The vague prompt resulted in basic, surface-level inline comments, while the detailed prompt produced comprehensive, standardized documentation.
2. **Structured Docstrings Are More Scalable**
   The detailed prompt yields output that is better suited for:
   - API documentation tools (e.g., Sphinx, Doxygen)
   - Collaborative projects
   - Long-term maintenance
3. **Type Hints and Exceptions Are Missed in Vague Prompts**
   Important details like parameter types, exceptions (`ValueError`), and return values were **not captured** in the vague prompt output.
4. **Vague Prompts Encourage Reactive Documentation**
   Comments were simply describing what each line does. This does not help users understand how to use the function or its intent.
5. **Detailed Prompt Encourages Declarative, Reusable Documentation**
   The output from the detailed prompt can be reused in documentation systems and easily interpreted by future developers.