ASSIGNMENT-9.4

NAME:-P.Likitha

Batch:-43

HTNO:-2303A52393

Task 1 – Prompt

Generate google-style docstrings for the following Python functions.

Each docstring should include:

- A brief description of the function

- Parameters with data types and short explanations

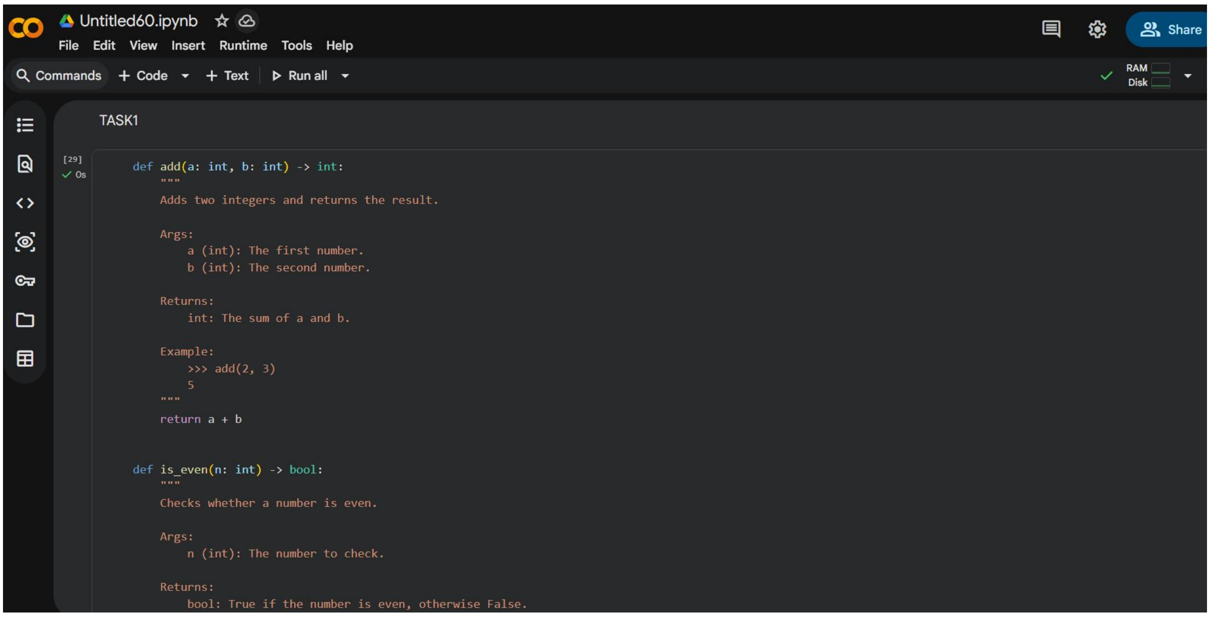
- Return values with data types

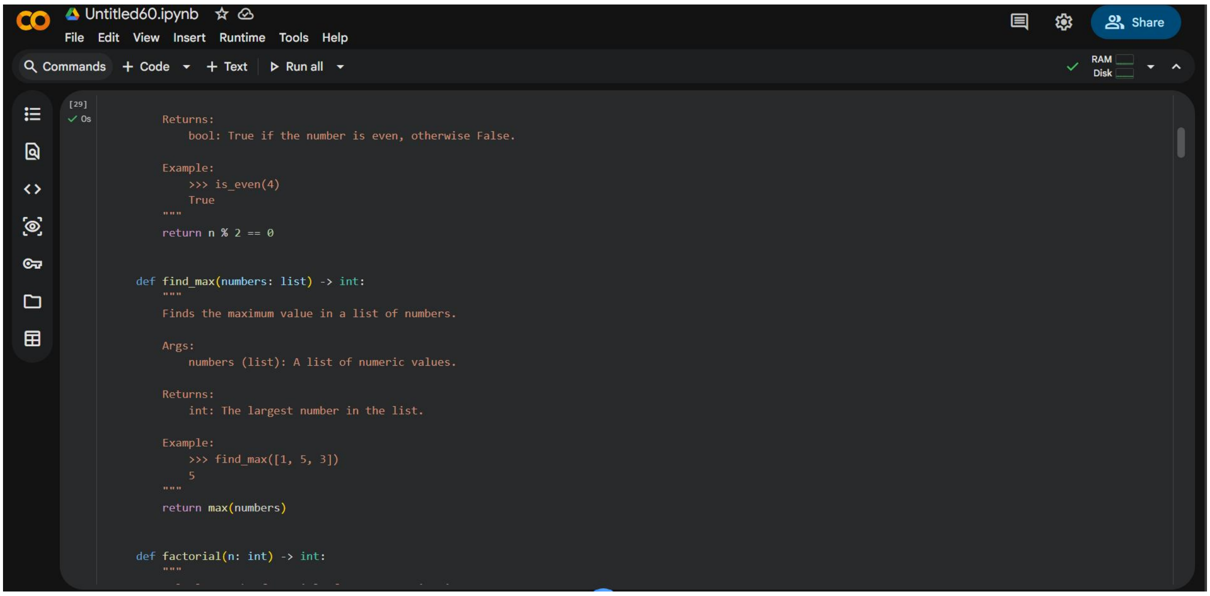
- At least one example usage in doctest format (if applicable)

Do not modify the original function logic.

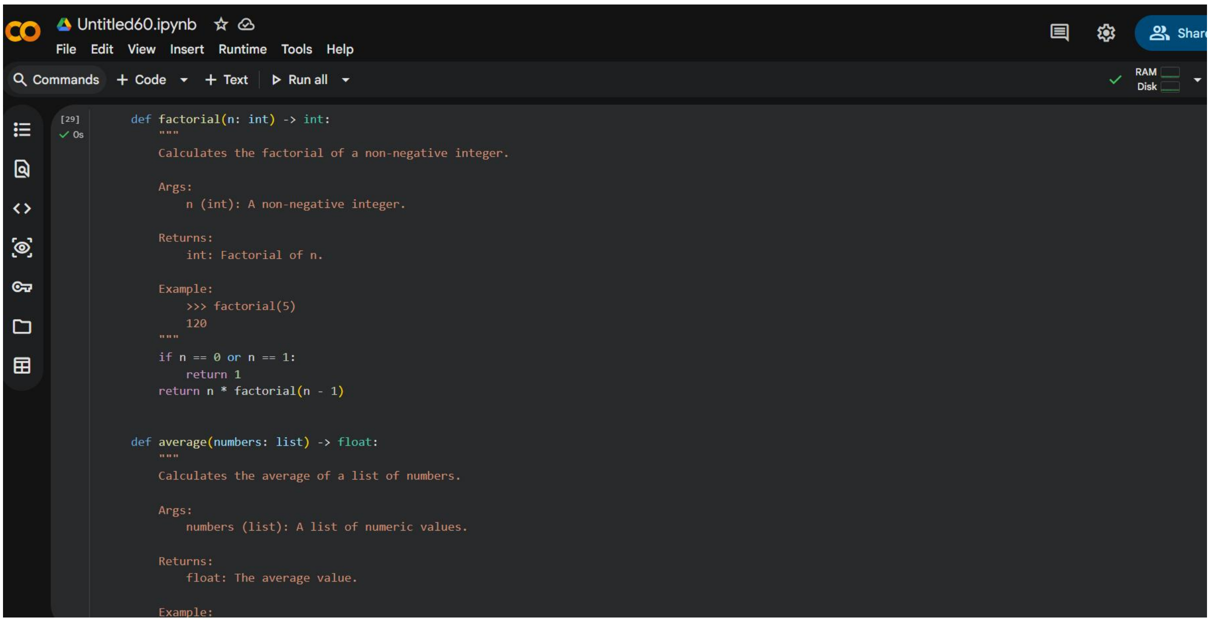
Ensure the formatting follows the Google Python Style Guide.

Keep the tone professional and clear for use in a shared codebase.

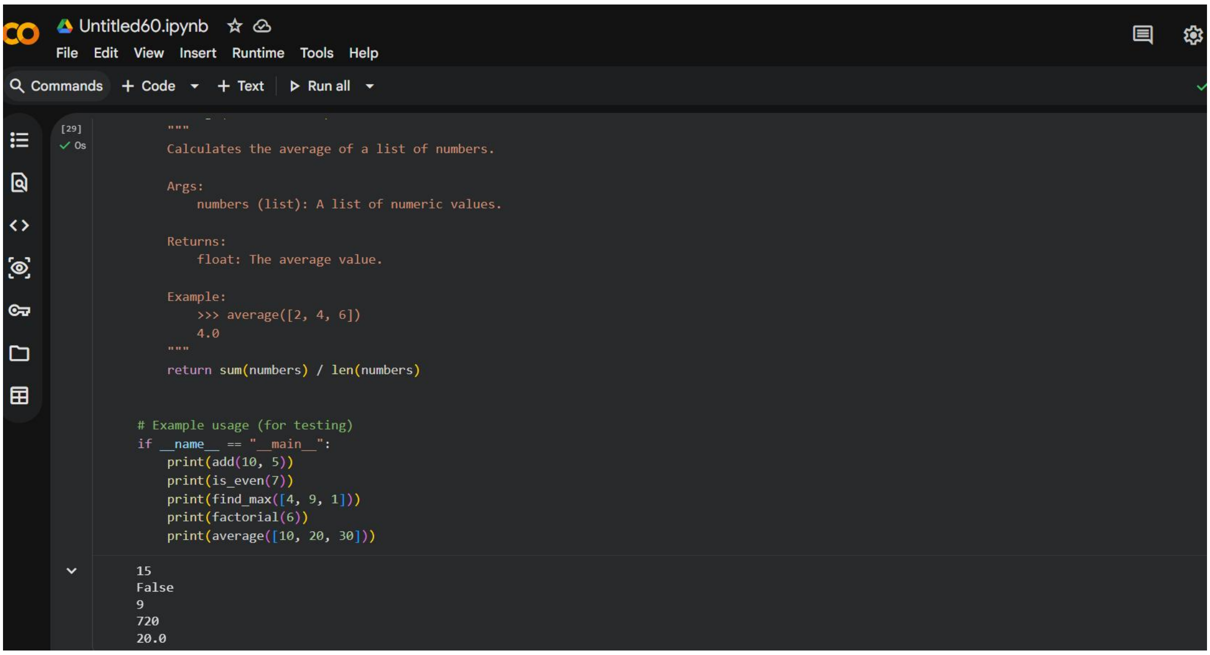




```python
    Returns:
        bool: True if the number is even, otherwise False.

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


def find_max(numbers: list) -> int:
    """
    Finds the maximum value in a list of numbers.

    Args:
        numbers (list): A list of numeric values.

    Returns:
        int: The largest number in the list.

    Example:
        >>> find_max([1, 5, 3])
        5
    """
    return max(numbers)


def factorial(n: int) -> int:
    """
```



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

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

    Returns:
        int: Factorial of n.

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


def average(numbers: list) -> float:
    """
    Calculates the average of a list of numbers.

    Args:
        numbers (list): A list of numeric values.

    Returns:
        float: The average value.

    Example:
```



Observation

- AI successfully generated well-structured Google-style docstrings.
- Function purpose, parameters, return values, and examples are clearly explained.
- Code readability and usability for new developers improved significantly.

Task2

You are a senior Python developer reviewing a codebase for readability improvements.

The following Python script contains complex logic, including loops, conditional statements, and algorithms. The code works correctly but lacks clarity.

Your task:

- Insert concise inline comments ONLY where the logic is complex or non-obvious.

- Explain WHY the logic exists, not what basic Python syntax does.

- Avoid commenting on trivial or self-explanatory lines.

- Do not modify the original logic or structure.

- Keep comments professional and minimal to avoid clutter.

Return the improved version of the script with meaningful inline comments added.

TASK2

```python
def fibonacci(n):
    fib = [0, 1]

    # Build sequence iteratively to avoid recursion overhead
    for i in range(2, n):
        # Each value depends on previous two values (core Fibonacci relation)
        fib.append(fib[i - 1] + fib[i - 2])

    return fib[:n]


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

    # Continue narrowing search space until pointers cross
    while left <= right:
        mid = (left + right) // 2

        # Decide which half to discard based on comparison
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            # Target must be in right half (array assumed sorted)
            left = mid + 1
        else:
            # Target must be in left half
```

```python
            # Target must be in left half
            right = mid -
    return -1
def bubble_sort(arr):
    n = len(arr)
    # Outer loop controls number of passes
    for i in range(n):
        swapped = False  # Track if any swap occurs to allow early stop
        # Ignore last i elements since they are already sorted
        for j in range(0, n - i - 1):
            # Swap only when elements are out of order
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        # If no swaps happened, array is already sorted → optimization
        if not swapped:
            break
    return arr
# Example usage
if __name__ == "__main__":
    print("Fibonacci:", fibonacci(10))

    sorted_list = bubble_sort([5, 2, 9, 1, 7])
    print("Sorted:", sorted_list)

    print("Binary search index:", binary_search(sorted_list, 7))
```

```
Fibonacci: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
Sorted: [1, 2, 5, 7, 9]
Binary search index: 3
```

Observation

- Inline comments were added only to complex and non-obvious logic.
- Trivial Python syntax was left uncommented, avoiding clutter.
- The code became easier to understand and maintain.

Task 3

You are a Python documentation expert preparing a module for internal or public use.

Analyze the following complete Python module and generate a professional multi-line module-level docstring to be placed at the top of the file.

The docstring must include:

- A clear explanation of the module's purpose

- Required libraries or dependencies

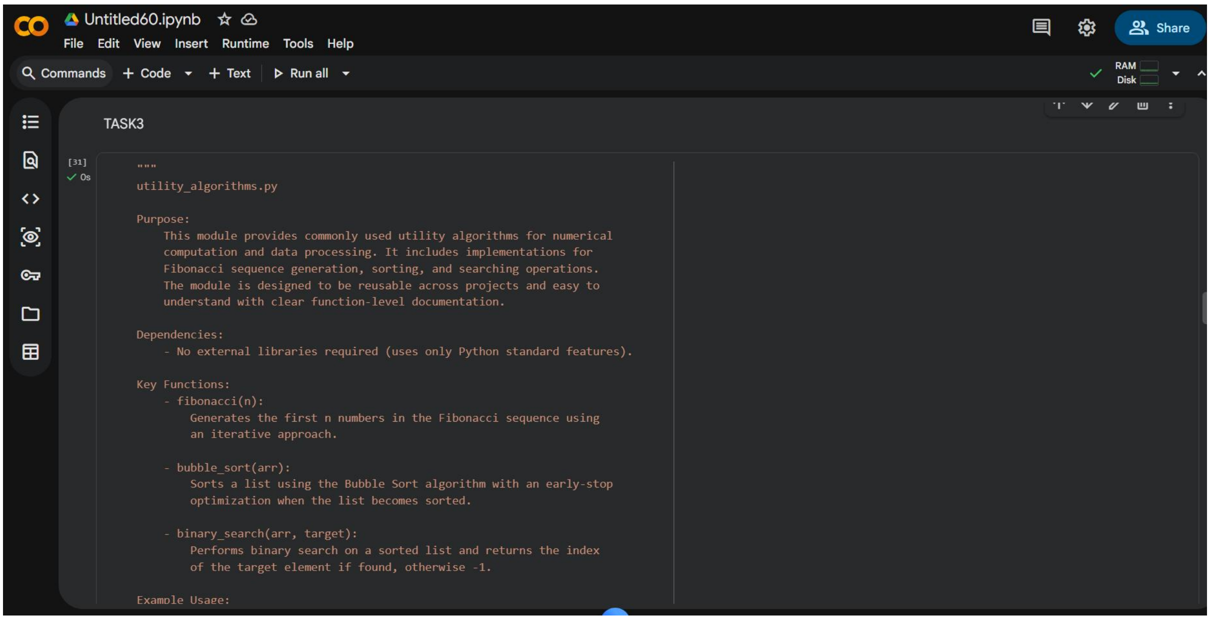
- A brief description of key functions and classes

- A short example demonstrating how to use the module

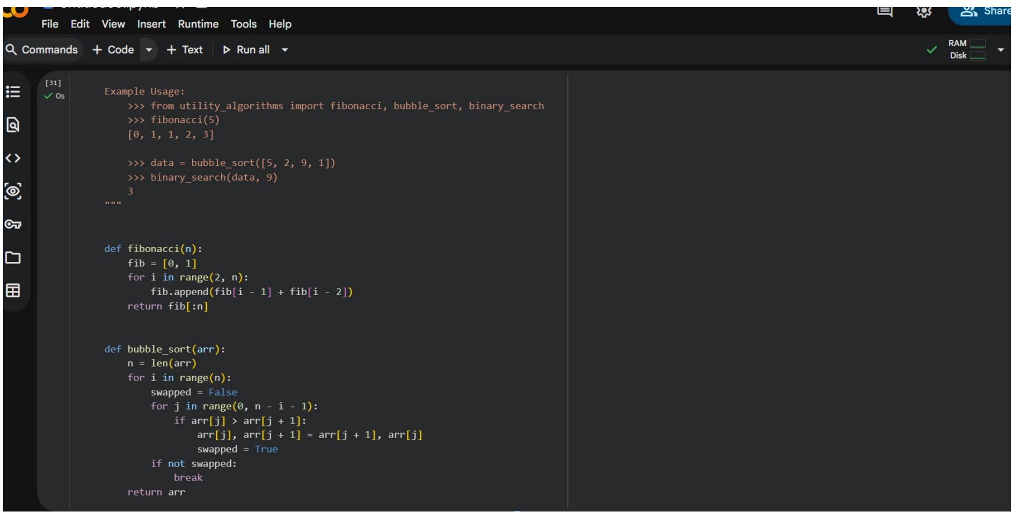
- Professional tone suitable for production code

Do not modify the existing code.

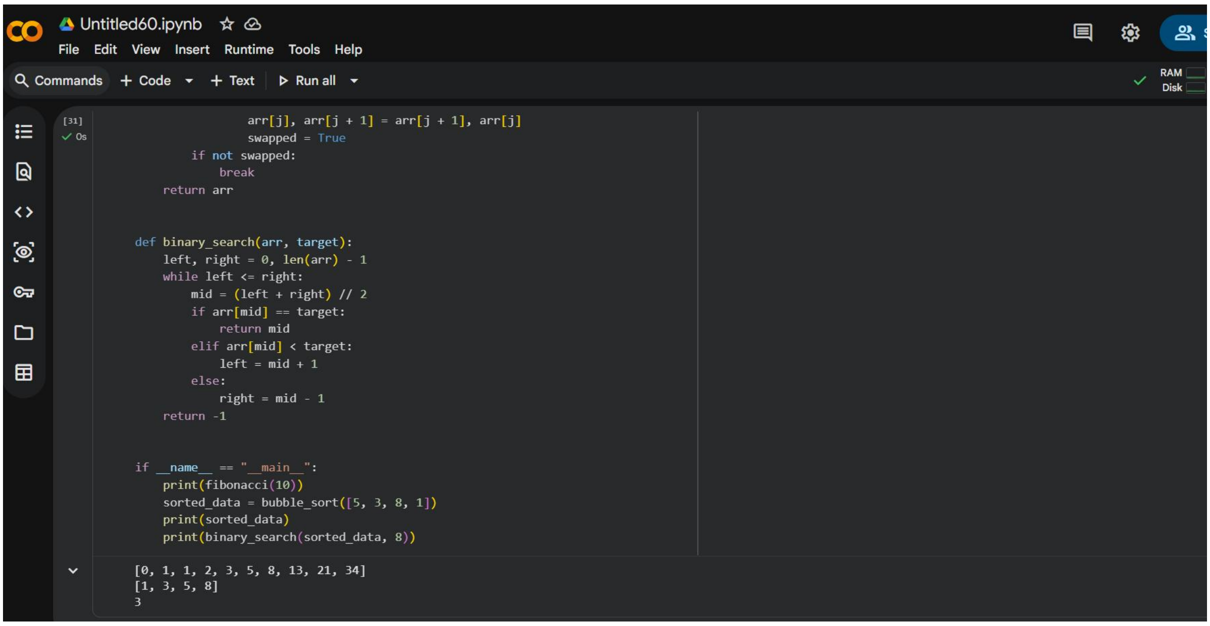
Only generate the module-level docstring.




```
"""
utility_algorithms.py

Purpose:
    This module provides commonly used utility algorithms for numerical
    computation and data processing. It includes implementations for
    Fibonacci sequence generation, sorting, and searching operations.
    The module is designed to be reusable across projects and easy to
    understand with clear function-level documentation.

Dependencies:
    - No external libraries required (uses only Python standard features).

Key Functions:
    - fibonacci(n):
        Generates the first n numbers in the Fibonacci sequence using
        an iterative approach.

    - bubble_sort(arr):
        Sorts a list using the Bubble Sort algorithm with an early-stop
        optimization when the list becomes sorted.

    - binary_search(arr, target):
        Performs binary search on a sorted list and returns the index
        of the target element if found, otherwise -1.

Example Usage:
```




```
Example Usage:
    >>> from utility_algorithms import fibonacci, bubble_sort, binary_search
    >>> fibonacci(5)
    [0, 1, 1, 2, 3]

    >>> data = bubble_sort([5, 2, 9, 1])
    >>> binary_search(data, 9)
    3
"""


def fibonacci(n):
    fib = [0, 1]
    for i in range(2, n):
        fib.append(fib[i - 1] + fib[i - 2])
    return fib[:n]


def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        if not swapped:
            break
    return arr
```



Observation

- A clear and professional module-level docstring was generated.
- The purpose and structure of the module are understandable at first glance.
- Documentation is suitable for real-world and shared codebases.

Task 4

You are a Python refactoring specialist helping standardize documentation in a legacy project.

The following script contains detailed inline comments inside functions explaining their behavior. These should be converted into structured docstrings.

Your task:

- Convert relevant explanatory comments into proper Google-style (or NumPy-style) docstrings.

- Preserve the original meaning and intent.

- Include:

  - Description

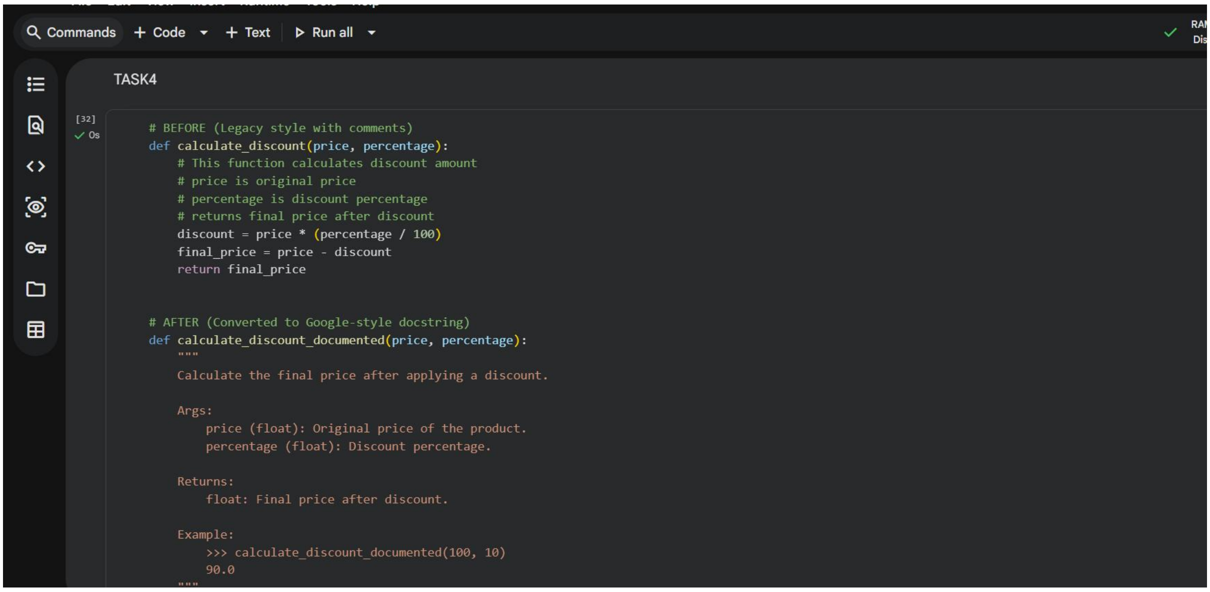
  - Parameters with types

  - Return values with types

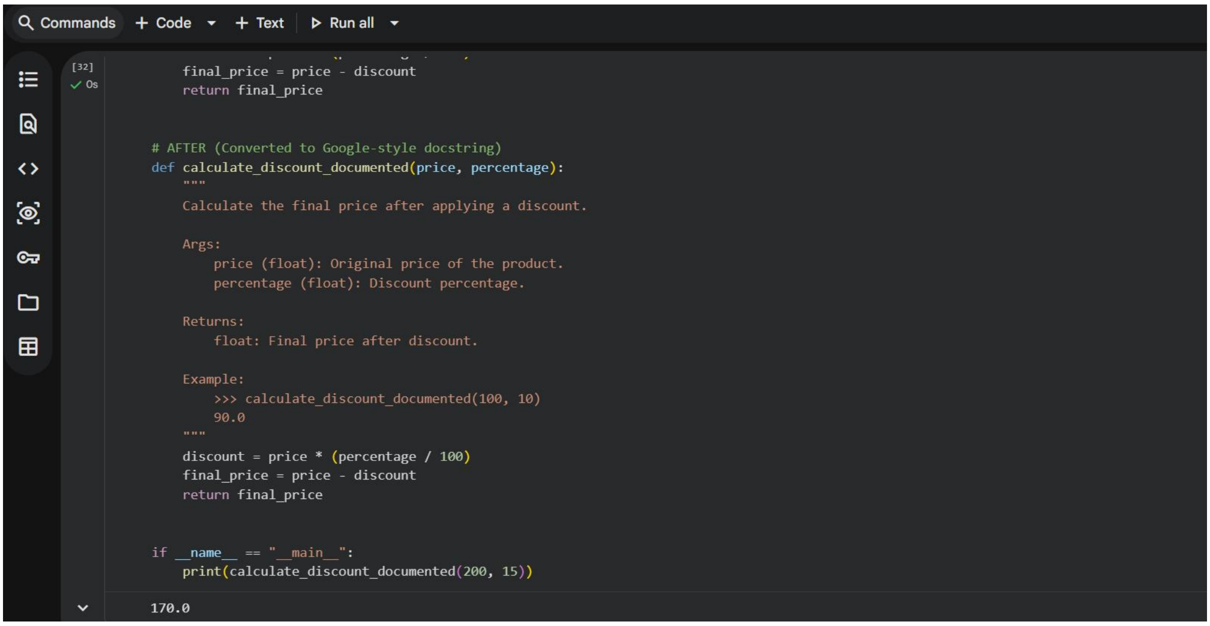
- Remove redundant inline comments after conversion.

- Do not modify function logic.

Return the cleaned and standardized version of the
script.




```python
# BEFORE (Legacy style with comments)
def calculate_discount(price, percentage):
    # This function calculates discount amount
    # price is original price
    # percentage is discount percentage
    # returns final price after discount
    discount = price * (percentage / 100)
    final_price = price - discount
    return final_price


# AFTER (Converted to Google-style docstring)
def calculate_discount_documented(price, percentage):
    """
    Calculate the final price after applying a discount.

    Args:
        price (float): Original price of the product.
        percentage (float): Discount percentage.

    Returns:
        float: Final price after discount.

    Example:
        >>> calculate_discount_documented(100, 10)
        90.0
    """
```




```python
    final_price = price - discount
    return final_price


# AFTER (Converted to Google-style docstring)
def calculate_discount_documented(price, percentage):
    """
    Calculate the final price after applying a discount.

    Args:
        price (float): Original price of the product.
        percentage (float): Discount percentage.

    Returns:
        float: Final price after discount.

    Example:
        >>> calculate_discount_documented(100, 10)
        90.0
    """
    discount = price * (percentage / 100)
    final_price = price - discount
    return final_price


if __name__ == "__main__":
    print(calculate_discount_documented(200, 15))
```

```
170.0
```

Observation

- Detailed inline comments were successfully converted into structured docstrings.
- Redundant comments inside function bodies were removed.
- Code consistency and cleanliness improved.

Task 5

You are a Python developer building an internal documentation scaffolding tool.Design a small Python utility that:

- Reads a given .py file

- Automatically detects all functions and classes

- Inserts placeholder Google-style docstrings if none exist

- Preserves indentation and formatting

- Does not modify existing docstrings

The generated placeholder docstrings should include:

- Short description placeholder - Args section

- Returns section (if applicable)

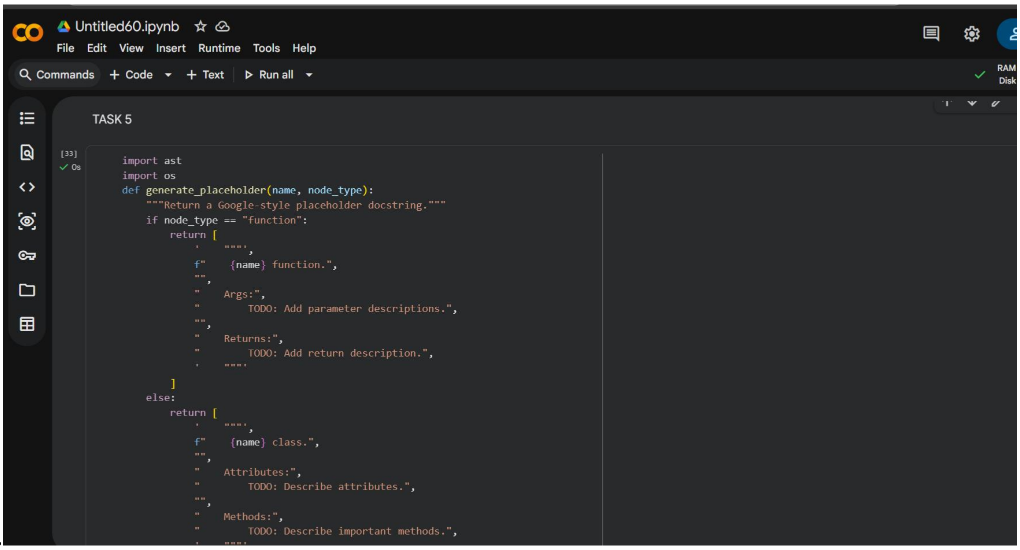
The goal is documentation scaffolding, not perfect documentation.

Provide:

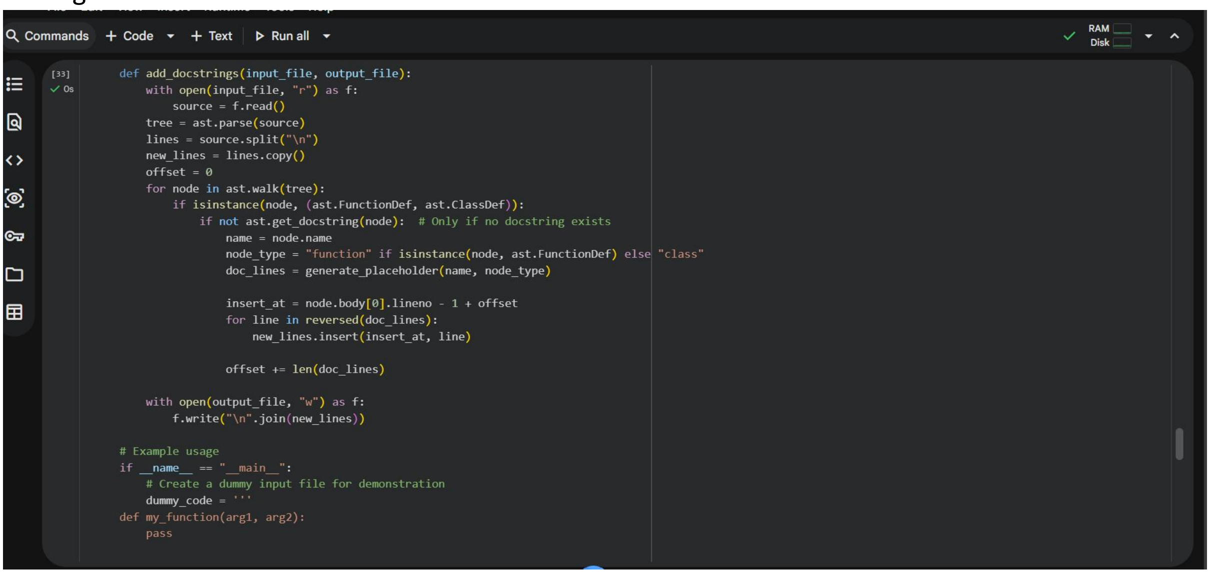
1. A complete working Python script

2. Clear explanation of how it works

3. Example of how to run it
Ensure the solution is clean, readable, and suitable for internal tooling.




```
import ast
import os
def generate_placeholder(name, node_type):
    """Return a Google-style placeholder docstring."""
    if node_type == "function":
        return [
            '    """',
            f"    {name} function.",
            "",
            "    Args:",
            "        TODO: Add parameter descriptions.",
            "",
            "    Returns:",
            "        TODO: Add return description.",
            '    """'
        ]
    else:
        return [
            '    """',
            f"    {name} class.",
            "",
            "    Attributes:",
            "        TODO: Describe attributes.",
            "",
            "    Methods:",
            "        TODO: Describe important methods.",
            '    """'
```




```
def add_docstrings(input_file, output_file):
    with open(input_file, "r") as f:
        source = f.read()
    tree = ast.parse(source)
    lines = source.split("\n")
    new_lines = lines.copy()
    offset = 0
    for node in ast.walk(tree):
        if isinstance(node, (ast.FunctionDef, ast.ClassDef)):
            if not ast.get_docstring(node):  # Only if no docstring exists
                name = node.name
                node_type = "function" if isinstance(node, ast.FunctionDef) else "class"
                doc_lines = generate_placeholder(name, node_type)

                insert_at = node.body[0].lineno - 1 + offset
                for line in reversed(doc_lines):
                    new_lines.insert(insert_at, line)

                offset += len(doc_lines)

    with open(output_file, "w") as f:
        f.write("\n".join(new_lines))

# Example usage
if __name__ == "__main__":
    # Create a dummy input file for demonstration
    dummy_code = '''
def my_function(arg1, arg2):
    pass
```

```
def my_function(arg1, arg2):
    pass

class MyClass:
    def __init__(self):
        pass

def another_func():
    """Existing docstring"""
    pass
'''
    with open("input.py", "w") as f:
        f.write(dummy_code)

    add_docstrings("input.py", "output.py")
    print("Documentation scaffolding completed successfully!")

    # Optionally, print the content of the output file
    print("\nContent of output.py:")
    with open("output.py", "r") as f:
        print(f.read())

    # Clean up the dummy files
    os.remove("input.py")
    os.remove("output.py")
```

```
Documentation scaffolding completed successfully!

Content of output.py:
```

```
def my_function(arg1, arg2):
    """
    my_function function.

    Args:
        TODO: Add parameter descriptions.

    Returns:
        TODO: Add return description.
    """
    pass

class MyClass:
    """
    MyClass class.

    Attributes:
        TODO: Describe attributes.

    Methods:
        TODO: Describe important methods.
    """
    def __init__(self):
    """
    __init__ function.

    Args:
        TODO: Add parameter descriptions.

    Returns:
        TODO: Add return description.
    """
        pass
```

```
    __init__ function.

    Args:
        TODO: Add parameter descriptions.

    Returns:
        TODO: Add return description.
    """
        pass

def another_func():
    """Existing docstring"""
    pass
```

Observation

- The documentation generator correctly detected functions in the Python file.
- Placeholder Google-style docstrings were generated automatically.
- The tool demonstrates how AI can assist in documentation automation.