AI Assisted Coding Lab Ass-9.4

Name: G.SAI GANESH

Batch:13

2303A51848

Task 1: Auto-Generating Function Documentation in a Shared Codebase

Scenario

You have joined a development team where several utility functions are already implemented, but the code lacks proper documentation. New team members are struggling to understand how these functions should be used.

Task Description

You are given a Python script containing multiple functions without any docstrings.

Using an AI-assisted coding tool:

Ask the AI to automatically generate Google-style function docstrings for each function

Each docstring should include:

A brief description of the function

Parameters with data types

Return values

At least one example usage (if applicable)

Experiment with different prompting styles (zero-shot or context-based) to observe quality differences.

PROMPT:

You are a senior Python developer working in a collaborative development team.

The following Python script is part of a shared utility codebase, but it lacks proper documentation. New team members are struggling to understand how to use these functions.

Automatically generate well-structured Google-style docstrings for each function.

Each docstring must include:

- A brief and clear description of the function

- Parameters with data types and explanation

- Return value with data type and explanation

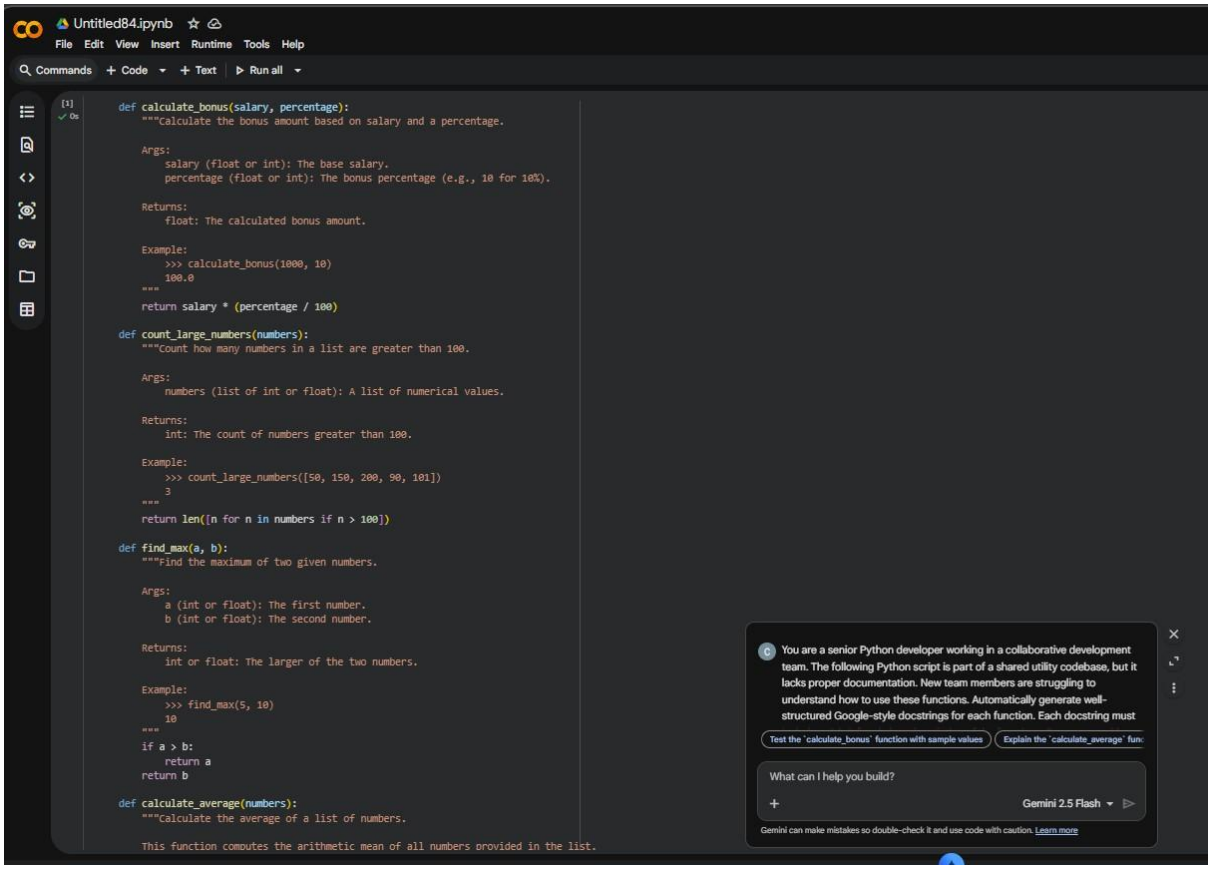
- At least one example usage (if applicable)

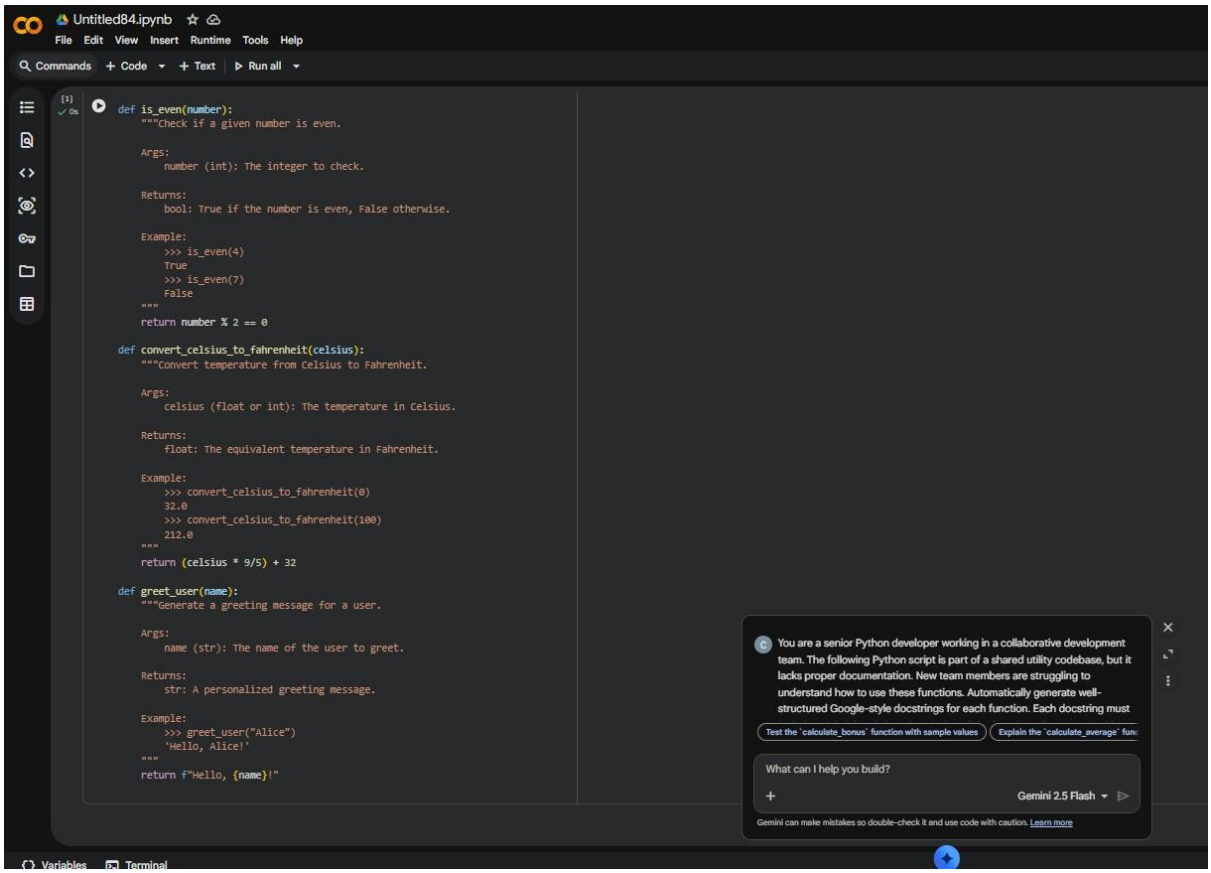
- Proper formatting consistent with Google-style documentation

Ensure the documentation improves readability and usability of the codebase.

Python Script:

```
def calculate_bonus(salary, percentage):
    return salary * (percentage / 100) def
count_large_numbers(numbers):
    return len([n for n in numbers if n > 100]) def
find_max(a, b):
    if a > b:        return a     return
b def
calculate_average(numbers):
total = sum(numbers)     return
total / len(numbers) def
is_even(number):
    return number % 2 == 0 def
convert_celsius_to_fahrenheit(celsius):
    return (celsius * 9/5) + 32 def
greet_user(name):
    return f"Hello, {name}!"
```





Task 2: Enhancing Readability Through AI-Generated Inline

Comments

Scenario

A Python program contains complex logic that works correctly but is difficult to understand at first glance. Future maintainers may find it hard to debug or extend this code.

Task Description

You are provided with a Python script containing:

- Loops
- Conditional logic
- Algorithms (such as Fibonacci sequence, sorting, or searching) Use AI assistance to:
- Automatically insert inline comments only for complex or non- obvious logic
- Avoid commenting on trivial or self-explanatory syntax The goal is to improve clarity without cluttering the code.

PROMPT:

You are a senior Python developer reviewing a complex Python script.

The following program contains loops, conditional logic, and algorithm implementations such as Fibonacci sequence, sorting, and searching.

Insert inline comments only for complex or non-obvious logic.

Avoid commenting on trivial or self-explanatory syntax.

Keep comments concise and meaningful.

Do not change the original code logic.

Here is the script: def

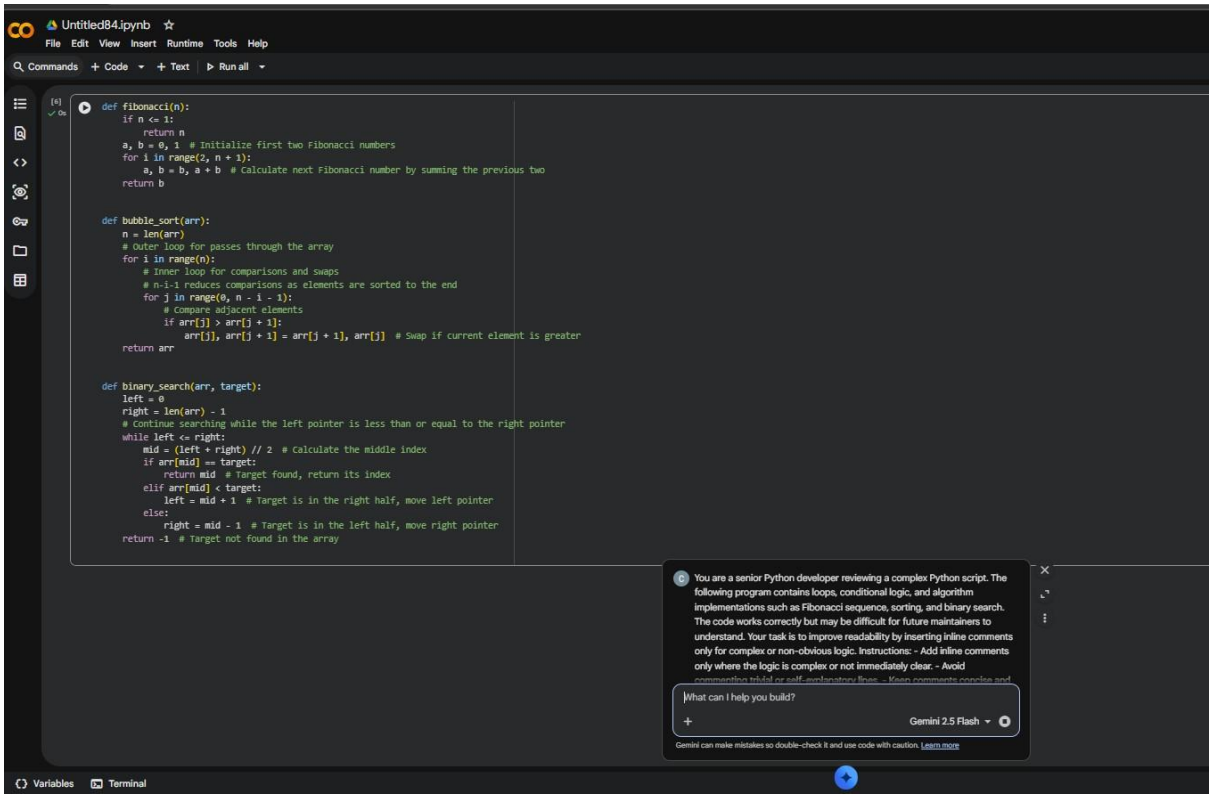
fibonacci(n):

if n <= 1:

return n a, b

= 0, 1 for i in

```
range(2, n +
1):        a, b =
b, a + b
return b



def bubble_sort(arr):     n =
len(arr)     for i in range(n):
for j in range(0, n - i - 1):
if arr[j] > arr[j + 1]:
              arr[j], arr[j + 1] = arr[j + 1], arr[j]
return arr



def binary_search(arr, target):
   left = 0     right = len(arr)
- 1     while left <= right:
mid = (left + right) // 2
if arr[mid] == target:
           return mid        elif
arr[mid] < target:
left = mid + 1         else:
right = mid - 1     return -
1
```



Task 3: Generating Module-Level Documentation for a Python Package

Scenario

Your team is preparing a Python module to be shared internally (or uploaded to a repository). Anyone opening the file should immediately understand its purpose and structure.

Task Description

Provide a complete Python module to an AI tool and instruct it to automatically generate a module-level docstring at the top of the file that includes:

- The purpose of the module
- Required libraries or dependencies
- A brief description of key functions and classes • A short example of how the module can be used

Focus on clarity and professional tone.

PROMPT:

You are a senior Python developer preparing a module for internal sharing and repository upload.

The following Python module needs a professional module-level docstring added at the very top of the file.

Automatically generate a well-structured module-level docstring that includes:

1. The overall purpose of the module

2. Required libraries or dependencies (if any)

3. A brief description of key functions and classes

4. A short example showing how the module can be used

5. Clear, professional tone and proper formatting

Do not modify the existing code.

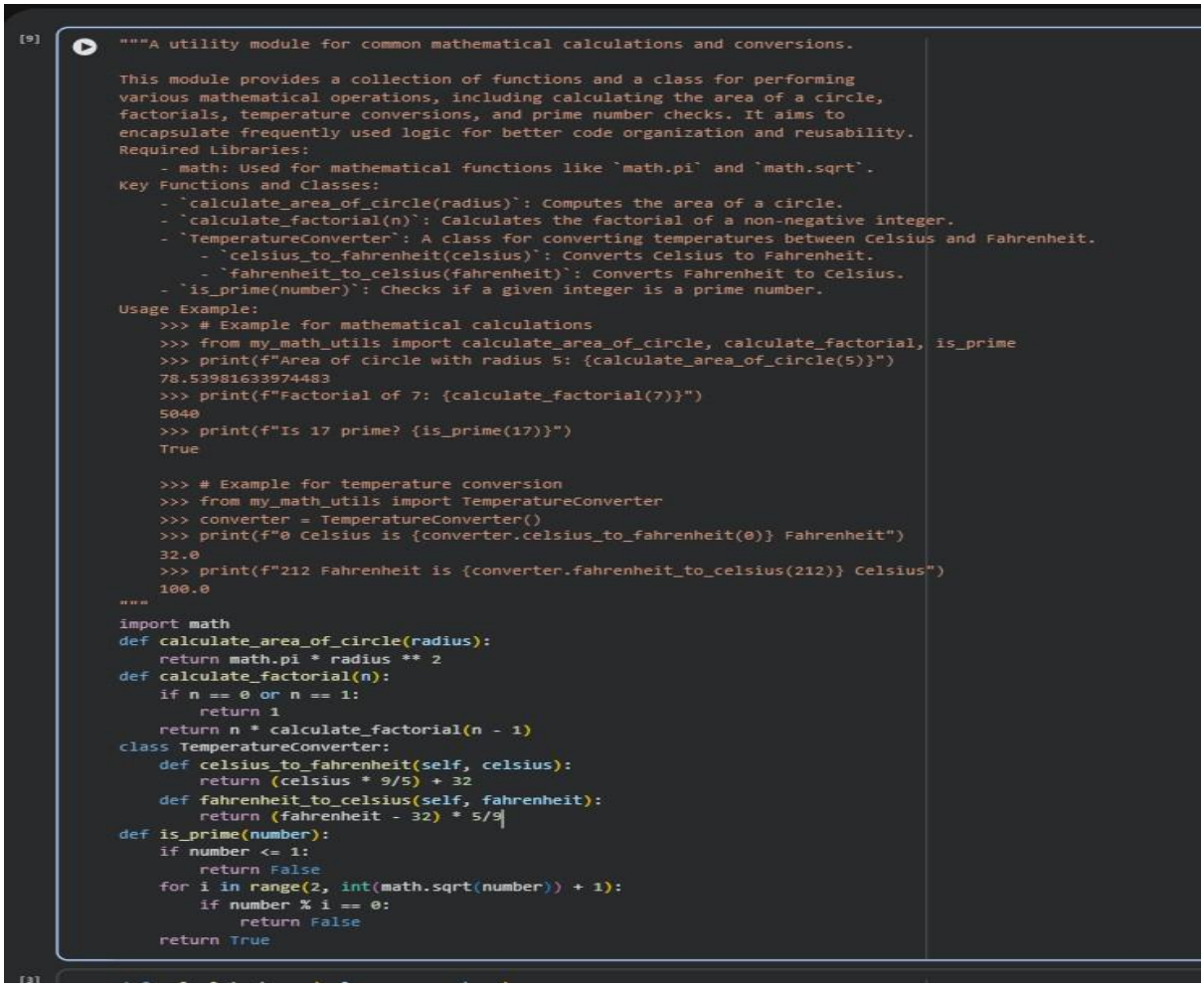
Only add the module-level docstring at the top of the file.

Python Module: import math def

calculate_area_of_circle(radius):

return math.pi * radius ** 2

```
def calculate_factorial(n):
if n == 0 or n == 1:
    return 1    return n *
calculate_factorial(n - 1) class
TemperatureConverter:
  def celsius_to_fahrenheit(self, celsius):
    return (celsius * 9/5) + 32    def
fahrenheit_to_celsius(self, fahrenheit):
    return (fahrenheit - 32) * 5/9 def
is_prime(number):
  if number <= 1:       return False    for i in
range(2, int(math.sqrt(number)) + 1):
    if number % i == 0:
      return False
return True
```



```
[9]
"""A utility module for common mathematical calculations and conversions.

This module provides a collection of functions and a class for performing
various mathematical operations, including calculating the area of a circle,
factorials, temperature conversions, and prime number checks. It aims to
encapsulate frequently used logic for better code organization and reusability.
Required Libraries:
    - math: Used for mathematical functions like `math.pi` and `math.sqrt`.
Key Functions and Classes:
    - `calculate_area_of_circle(radius)`: Computes the area of a circle.
    - `calculate_factorial(n)`: Calculates the factorial of a non-negative integer.
    - `TemperatureConverter`: A class for converting temperatures between Celsius and Fahrenheit.
        - `celsius_to_fahrenheit(celsius)`: Converts Celsius to Fahrenheit.
        - `fahrenheit_to_celsius(fahrenheit)`: Converts Fahrenheit to Celsius.
    - `is_prime(number)`: Checks if a given integer is a prime number.
Usage Example:
    >>> # Example for mathematical calculations
    >>> from my_math_utils import calculate_area_of_circle, calculate_factorial, is_prime
    >>> print(f"Area of circle with radius 5: {calculate_area_of_circle(5)}")
    78.53981633974483
    >>> print(f"Factorial of 7: {calculate_factorial(7)}")
    5040
    >>> print(f"Is 17 prime? {is_prime(17)}")
    True

    >>> # Example for temperature conversion
    >>> from my_math_utils import TemperatureConverter
    >>> converter = TemperatureConverter()
    >>> print(f"0 Celsius is {converter.celsius_to_fahrenheit(0)} Fahrenheit")
    32.0
    >>> print(f"212 Fahrenheit is {converter.fahrenheit_to_celsius(212)} Celsius")
    100.0
"""
import math
def calculate_area_of_circle(radius):
    return math.pi * radius ** 2
def calculate_factorial(n):
    if n == 0 or n == 1:
        return 1
    return n * calculate_factorial(n - 1)
class TemperatureConverter:
    def celsius_to_fahrenheit(self, celsius):
        return (celsius * 9/5) + 32
    def fahrenheit_to_celsius(self, fahrenheit):
        return (fahrenheit - 32) * 5/9
def is_prime(number):
    if number <= 1:
        return False
    for i in range(2, int(math.sqrt(number)) + 1):
        if number % i == 0:
            return False
    return True
```

Task 4: Converting Developer Comments into Structured Docstrings

Scenario

In a legacy project, developers have written long explanatory comments inside functions instead of proper docstrings. The team now wants to standardize documentation.

Task Description

You are given a Python script where functions contain detailed inline comments explaining their logic.

Use AI to:

- Automatically convert these comments into structured Google- style or NumPy-style docstrings
- Preserve the original meaning and intent of the comments
- Remove redundant inline comments after conversion

PROMPT:

You are a senior Python developer refactoring a legacy codebase.

The following Python script contains detailed explanatory inline comments inside functions instead of proper docstrings.

Your task is to:

1. **Convert the existing explanatory inline comments into structured Google-style docstrings.**
2. **Preserve the original meaning and intent of the comments.**
3. **Remove redundant inline comments after converting them into docstrings.**
4. **Do NOT modify the actual logic of the code.**
5. **Maintain professional formatting and consistency.**

Ensure each docstring includes:

- A brief description of the function

- Args section with parameter names and data types

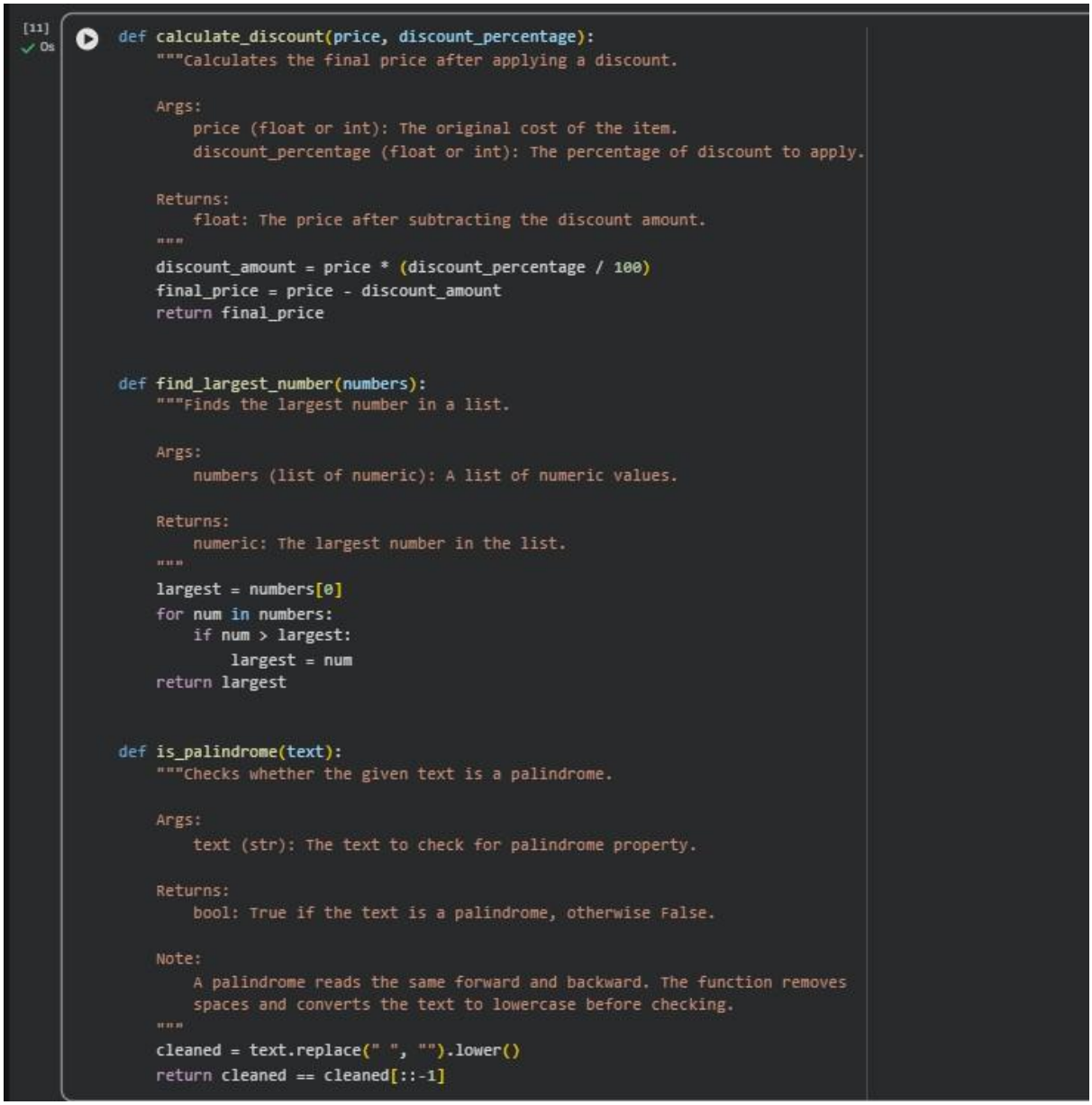
- Returns section with return data type

- Any important notes if necessary

Here is the Python script:

```
def calculate_discount(price, discount_percentage):
    # This function calculates the final price after applying a discount.
    # price is the original cost of the item.
    # discount_percentage is the percentage of discount to apply.
    # The function returns the price after subtracting the discount amount.

    discount_amount = price * (discount_percentage / 100)
final_price = price - discount_amount     return final_price
def find_largest_number(numbers):
```

```
   # This function finds the largest number in a list.
   # numbers is expected to be a list of numeric values.
   # It loops through each element and keeps track of the maximum value found.
   # It returns the largest number in the list.

   largest = numbers[0]
for num in numbers:
if num > largest:
largest = num     return
largest def
is_palindrome(text):
   # This function checks whether the given text is a palindrome.
   # A palindrome reads the same forward and backward.
   # It removes spaces and converts the text to lowercase before checking.
   # Returns True if the text is a palindrome, otherwise False.
cleaned = text.replace(" ", "").lower()     return cleaned ==
cleaned[::-1]
```



```
[11]
✓ 0s
def calculate_discount(price, discount_percentage):
    """Calculates the final price after applying a discount.

    Args:
        price (float or int): The original cost of the item.
        discount_percentage (float or int): The percentage of discount to apply.

    Returns:
        float: The price after subtracting the discount amount.
    """
    discount_amount = price * (discount_percentage / 100)
    final_price = price - discount_amount
    return final_price


def find_largest_number(numbers):
    """Finds the largest number in a list.

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

    Returns:
        numeric: The largest number in the list.
    """
    largest = numbers[0]
    for num in numbers:
        if num > largest:
            largest = num
    return largest


def is_palindrome(text):
    """Checks whether the given text is a palindrome.

    Args:
        text (str): The text to check for palindrome property.

    Returns:
        bool: True if the text is a palindrome, otherwise False.

    Note:
        A palindrome reads the same forward and backward. The function removes
        spaces and converts the text to lowercase before checking.
    """
    cleaned = text.replace(" ", "").lower()
    return cleaned == cleaned[::-1]
```

Task 5: Building a Mini Automatic Documentation Generator

Scenario

Your team wants a simple internal tool that helps developers start documenting new Python files quickly, without writing documentation from scratch.

Task Description

Design a small Python utility that:

- Reads a given .py file
- Automatically detects:

o Functions o

Classes

• Inserts placeholder Google-style docstrings for each detected function

or class

AI tools may be used to assist in generating or refining this utility. Note:

The goal is documentation scaffolding, not perfect documentation.

PROMPT:

You are a Python developer building a small internal tool for automatic documentation scaffolding.

Design a Python utility that:

1. Reads a given .py file.

2. Automatically detects:

- Functions

- Classes

3. Inserts placeholder Google-style docstrings for each detected function or class.

4. Does NOT aim to generate perfect documentation — only scaffolding placeholders.

5. Preserves the original code logic and formatting as much as possible.

6. Creates a new updated file instead of overwriting the original.

Requirements:

- Use standard Python libraries (such as ast if appropriate).

- Ensure placeholders include:

- Brief description section

- Args section (for functions)

- Returns section (for functions)

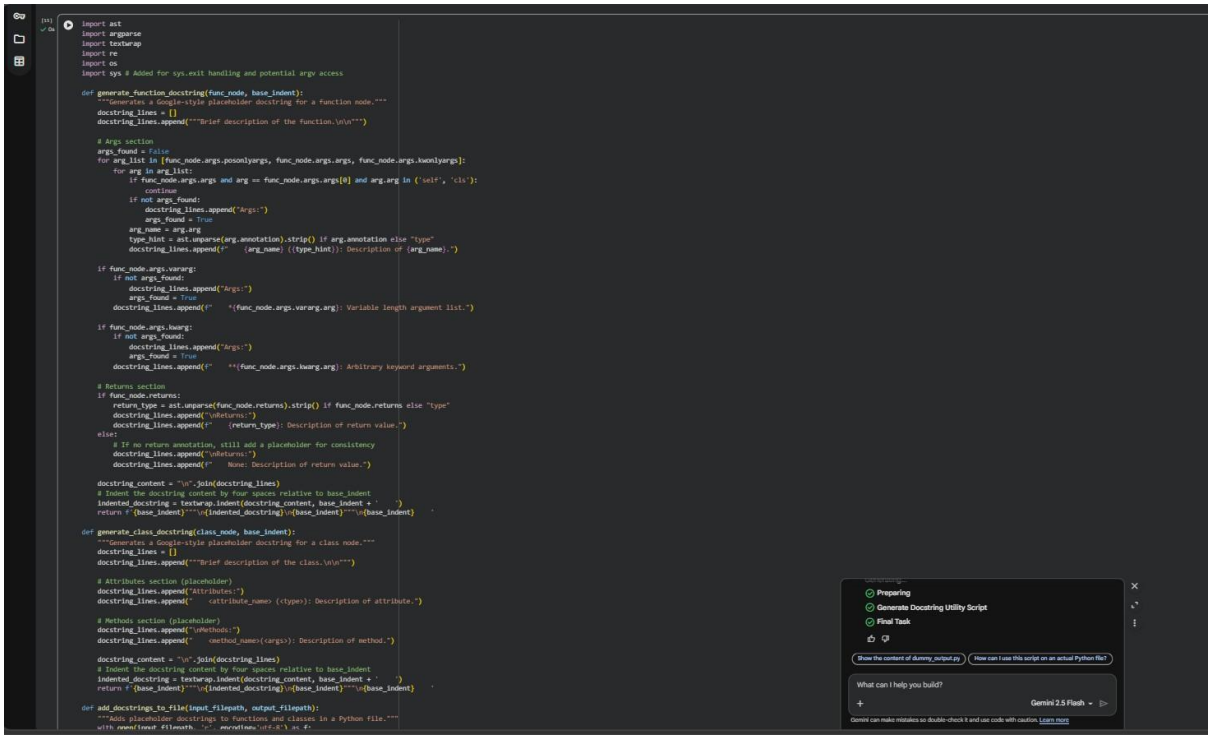
Attributes/Methods section (for classes)

- Maintain clean and professional formatting.

- Add docstrings only if they are missing.

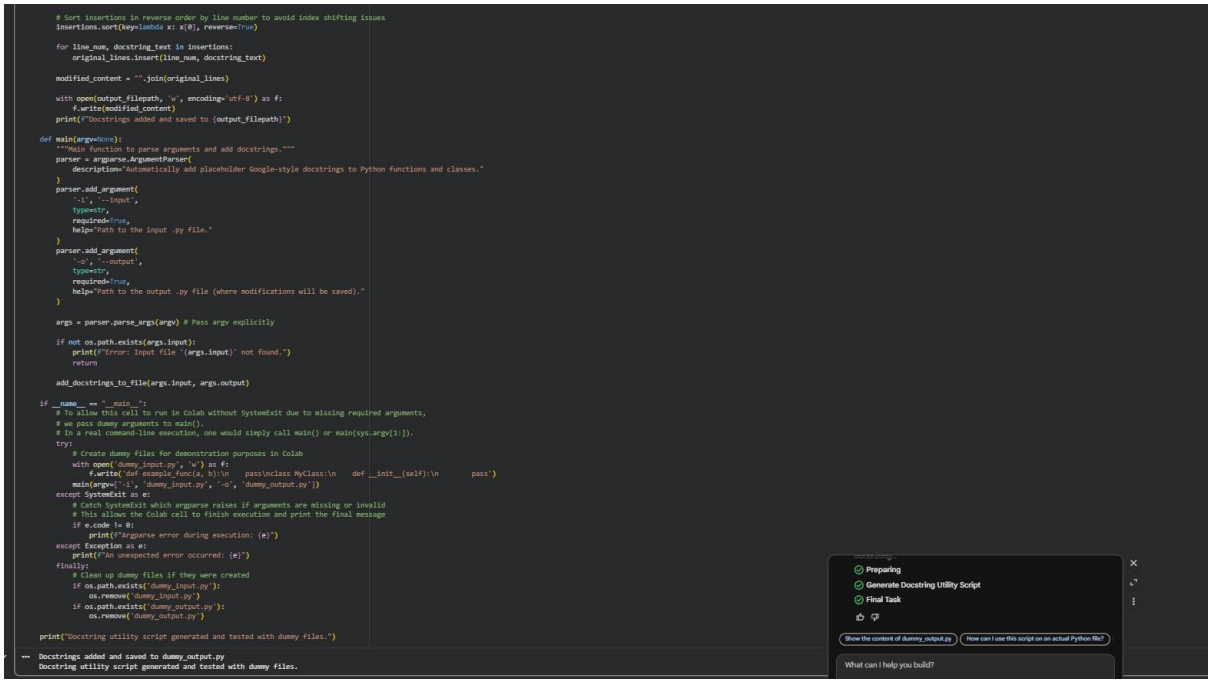
- Do not modify existing docstrings.

Generate the complete working Python script for this utility.



```
import ast
import argparse
import textwrap
import re
import os
import sys # Added for sys.exit handling and potential argv access

def generate_function_docstring(func_node, base_indent):
    """Generates a Google-style placeholder docstring for a function node."""
    docstring_lines = []
    docstring_lines.append("""Brief description of the function.\n\n""")

    # Args section
    args_found = False
    for arg_list in [func_node.args.posonlyargs, func_node.args.args, func_node.args.kwonlyargs]:
        for arg in arg_list:
            if func_node.args.args and arg == func_node.args.args[0] and arg.arg in ('self', 'cls'):
                continue
            if not args_found:
                docstring_lines.append("Args:")
                args_found = True
            arg_name = arg.arg
            type_hint = ast.unparse(arg.annotation).strip() if arg.annotation else "type"
            docstring_lines.append(f"    {arg_name} ({type_hint}): Description of {arg_name}.")

    if func_node.args.vararg:
        if not args_found:
            docstring_lines.append("Args:")
            args_found = True
        docstring_lines.append(f"    *{func_node.args.vararg.arg}: Variable length argument list.")

    if func_node.args.kwarg:
        if not args_found:
            docstring_lines.append("Args:")
            args_found = True
        docstring_lines.append(f"    **{func_node.args.kwarg.arg}: Arbitrary keyword arguments.")

    # Returns section
    if func_node.returns:
        return_type = ast.unparse(func_node.returns).strip() if func_node.returns else "type"
        docstring_lines.append("\nReturns:")
        docstring_lines.append(f"    {return_type}: Description of return value.")
    else:
        # If no return annotation, still add a placeholder for consistency
        docstring_lines.append("\nReturns:")
        docstring_lines.append(f"    None: Description of return value.")

    docstring_content = "\n".join(docstring_lines)
    # Indent the docstring content by four spaces relative to base_indent
    indented_docstring = textwrap.indent(docstring_content, base_indent + '    ')
    return f'{base_indent}"""\n{indented_docstring}\n{base_indent}"""\n{base_indent}'

def generate_class_docstring(class_node, base_indent):
    """Generates a Google-style placeholder docstring for a class node."""
    docstring_lines = []
    docstring_lines.append("""Brief description of the class.\n\n""")

    # Attributes section (placeholder)
    docstring_lines.append("Attributes:")
    docstring_lines.append("    <attribute_name> (<type>): Description of attribute.")

    # Methods section (placeholder)
    docstring_lines.append("\nMethods:")
    docstring_lines.append("    <method_name>(<args>): Description of method.")

    docstring_content = "\n".join(docstring_lines)
    # Indent the docstring content by four spaces relative to base_indent
    indented_docstring = textwrap.indent(docstring_content, base_indent + '    ')
    return f'{base_indent}"""\n{indented_docstring}\n{base_indent}"""\n{base_indent}'

def add_docstrings_to_file(input_filepath, output_filepath):
    """Adds placeholder docstrings to functions and classes in a Python file."""
    with open(input_filepath, 'r', encoding='utf-8') as f:
```

Preparing
Generate Docstring Utility Script
Final Task
Show the content of dummy_output.py
How can I use this script on an actual Python file?
What can I help you build?
Gemini 2.5 Flash
Gemini can make mistakes so double-check it and use code with caution. Learn more



```
    # Sort insertions in reverse order by line number to avoid index shifting issues
    insertions.sort(key=lambda x: x[0], reverse=True)

    for line_num, docstring_text in insertions:
        original_lines.insert(line_num, docstring_text)

    modified_content = "".join(original_lines)

    with open(output_filepath, 'w', encoding='utf-8') as f:
        f.write(modified_content)
    print(f"Docstrings added and saved to {output_filepath}")

def main(argv=None):
    """Main function to parse arguments and add docstrings."""
    parser = argparse.ArgumentParser(
        description="Automatically add placeholder Google-style docstrings to Python functions and classes."
    )
    parser.add_argument(
        '-i', '--input',
        type=str,
        required=True,
        help="Path to the input .py file."
    )
    parser.add_argument(
        '-o', '--output',
        type=str,
        required=True,
        help="Path to the output .py file (where modifications will be saved)."
    )

    args = parser.parse_args(argv) # Pass argv explicitly

    if not os.path.exists(args.input):
        print(f"Error: Input file '{args.input}' not found.")
        return

    add_docstrings_to_file(args.input, args.output)

if __name__ == "__main__":
    # To allow this cell to run in Colab without SystemExit due to missing required arguments,
    # we pass dummy arguments to main().
    # In a real command-line execution, one would simply call main() or main(sys.argv[1:]).
    try:
        # Create dummy files for demonstration purposes in Colab
        with open('dummy_input.py', 'w') as f:
            f.write('def example_func(a, b):\n    pass\nclass MyClass:\n    def __init__(self):\n        pass')
        main(argv=['-i', 'dummy_input.py', '-o', 'dummy_output.py'])
    except SystemExit as e:
        # Catch SystemExit which argparse raises if arguments are missing or invalid
        # This allows the Colab cell to finish execution and print the final message
        if e.code != 0:
            print(f"Argparse error during execution: {e}")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")
    finally:
        # Clean up dummy files if they were created
        if os.path.exists('dummy_input.py'):
            os.remove('dummy_input.py')
        if os.path.exists('dummy_output.py'):
            os.remove('dummy_output.py')

print("Docstring utility script generated and tested with dummy files.")
```

```
Docstrings added and saved to dummy_output.py
Docstring utility script generated and tested with dummy files.
```

Preparing
Generate Docstring Utility Script
Final Task
Show the content of dummy_output.py
How can I use this script on an actual Python file?
What can I help you build?