AI Assisted Coding-Lab 9.4

Shivani Pambidi

2303A52172

Batch – 41

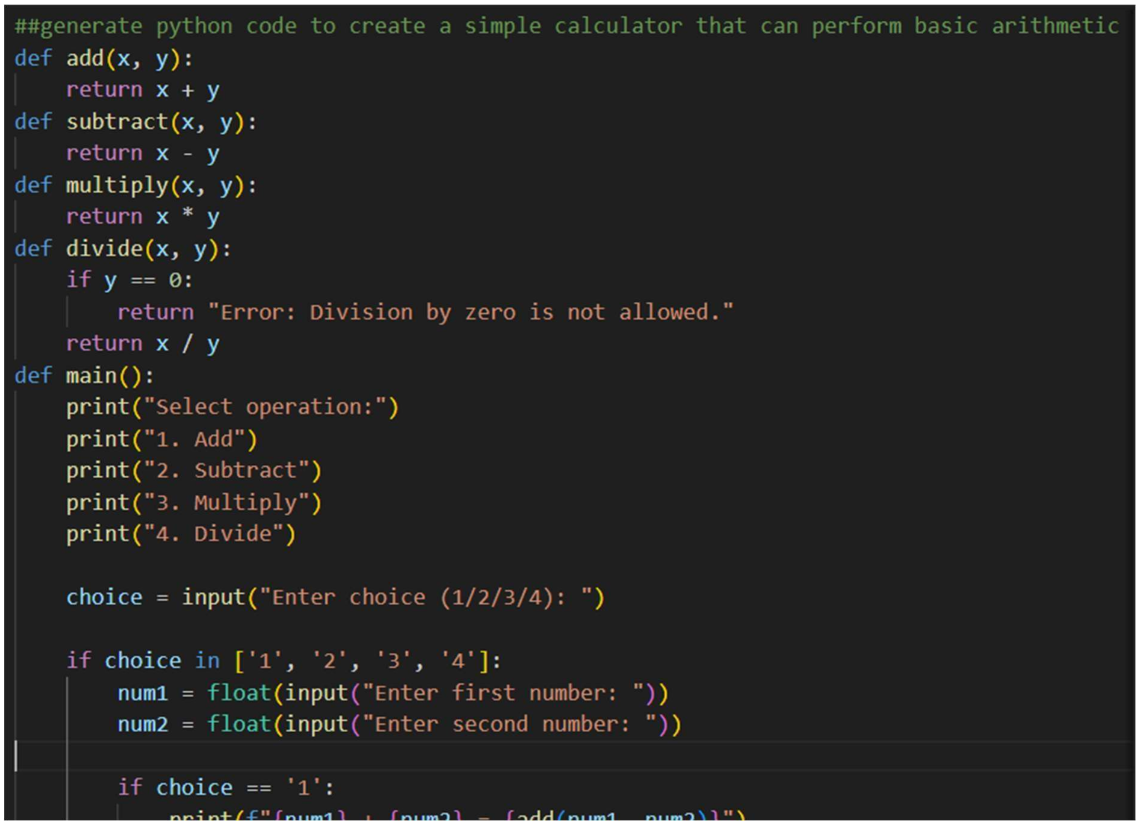
Question 1:

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

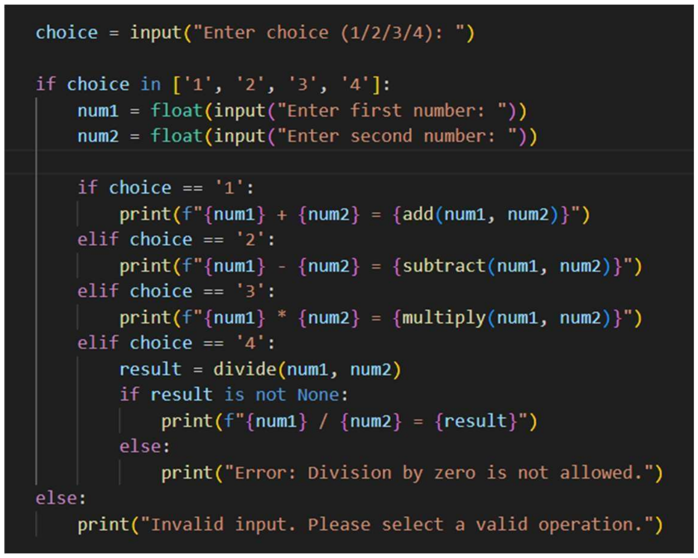
PROMPT:

Generate python code to create a simple calculator that can perform basic arithmetic operations such as addition, subtraction, multiplication, and division. The calculator should take user input for the operation and the numbers to be calculated.

CODE:



```python
##generate python code to create a simple calculator that can perform basic arithmetic
def add(x, y):
    return x + y
def subtract(x, y):
    return x - y
def multiply(x, y):
    return x * y
def divide(x, y):
    if y == 0:
        return "Error: Division by zero is not allowed."
    return x / y
def main():
    print("Select operation:")
    print("1. Add")
    print("2. Subtract")
    print("3. Multiply")
    print("4. Divide")

    choice = input("Enter choice (1/2/3/4): ")

    if choice in ['1', '2', '3', '4']:
        num1 = float(input("Enter first number: "))
        num2 = float(input("Enter second number: "))

        if choice == '1':
```



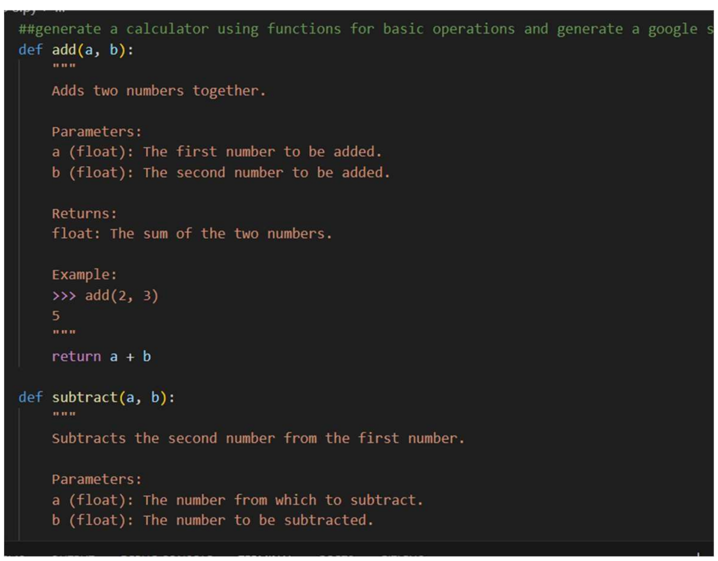
```python
choice = input("Enter choice (1/2/3/4): ")

if choice in ['1', '2', '3', '4']:
    num1 = float(input("Enter first number: "))
    num2 = float(input("Enter second number: "))

    if choice == '1':
        print(f"{num1} + {num2} = {add(num1, num2)}")
    elif choice == '2':
        print(f"{num1} - {num2} = {subtract(num1, num2)}")
    elif choice == '3':
        print(f"{num1} * {num2} = {multiply(num1, num2)}")
    elif choice == '4':
        result = divide(num1, num2)
        if result is not None:
            print(f"{num1} / {num2} = {result}")
        else:
            print("Error: Division by zero is not allowed.")
else:
    print("Invalid input. Please select a valid operation.")
```

Google-style function docstrings

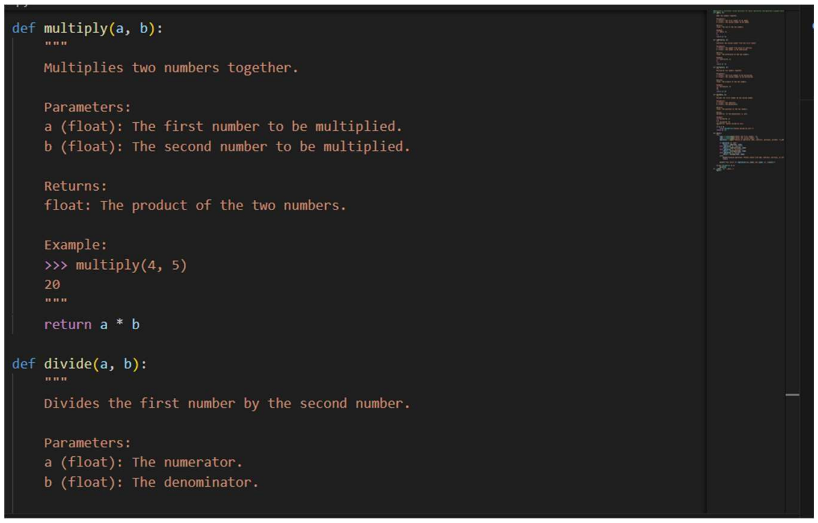
PROMPT:

Generate a calculator using functions for basic operations and generate a google style doctstring for each function.each doctstring should include a brief description about the function,parametres,return value and an example of how to use the function.

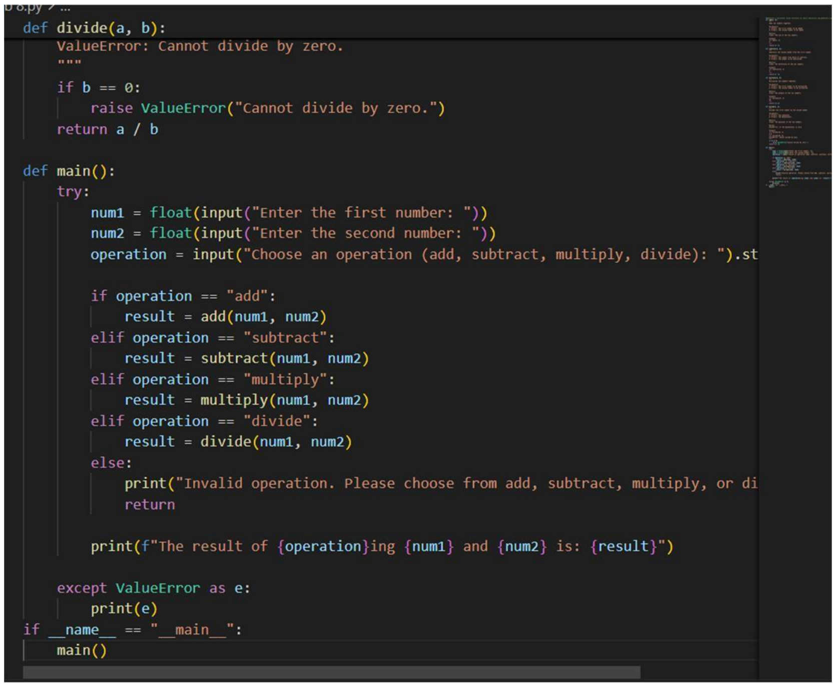
CODE:



```python
##generate a calculator using functions for basic operations and generate a google s
def add(a, b):
    """
    Adds two numbers together.

    Parameters:
    a (float): The first number to be added.
    b (float): The second number to be added.

    Returns:
    float: The sum of the two numbers.

    Example:
    >>> add(2, 3)
    5
    """
    return a + b

def subtract(a, b):
    """
    Subtracts the second number from the first number.

    Parameters:
    a (float): The number from which to subtract.
    b (float): The number to be subtracted.
```

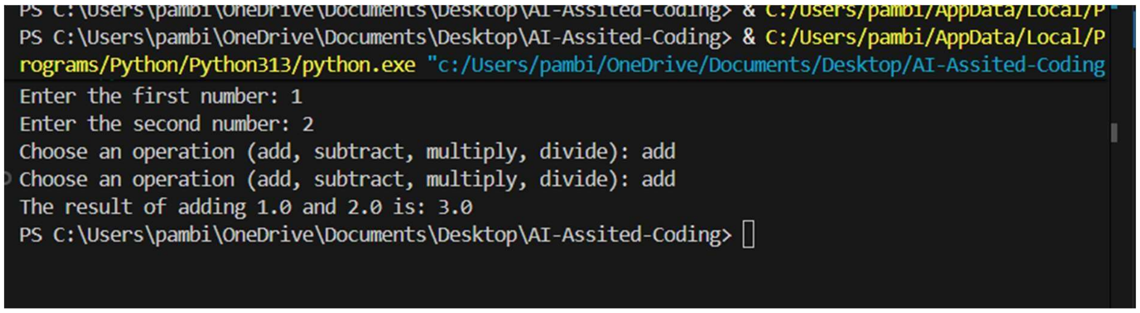


```python
def multiply(a, b):
    """
    Multiplies two numbers together.

    Parameters:
    a (float): The first number to be multiplied.
    b (float): The second number to be multiplied.

    Returns:
    float: The product of the two numbers.

    Example:
    >>> multiply(4, 5)
    20
    """
    return a * b

def divide(a, b):
    """
    Divides the first number by the second number.

    Parameters:
    a (float): The numerator.
    b (float): The denominator.
```



```python
def divide(a, b):
    ValueError: Cannot divide by zero.
    """
    if b == 0:
        raise ValueError("Cannot divide by zero.")
    return a / b

def main():
    try:
        num1 = float(input("Enter the first number: "))
        num2 = float(input("Enter the second number: "))
        operation = input("Choose an operation (add, subtract, multiply, divide): ").st

        if operation == "add":
            result = add(num1, num2)
        elif operation == "subtract":
            result = subtract(num1, num2)
        elif operation == "multiply":
            result = multiply(num1, num2)
        elif operation == "divide":
            result = divide(num1, num2)
        else:
            print("Invalid operation. Please choose from add, subtract, multiply, or di
            return

        print(f"The result of {operation}ing {num1} and {num2} is: {result}")

    except ValueError as e:
        print(e)
if __name__ == "__main__":
    main()
```

OUTPUT:



```
PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assited-Coding> & C:/Users/pambi/AppData/Local/P
PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assited-Coding> & C:/Users/pambi/AppData/Local/P
rograms/Python/Python313/python.exe "c:/Users/pambi/OneDrive/Documents/Desktop/AI-Assited-Coding
Enter the first number: 1
Enter the second number: 2
Choose an operation (add, subtract, multiply, divide): add
Choose an operation (add, subtract, multiply, divide): add
The result of adding 1.0 and 2.0 is: 3.0
PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assited-Coding> 
```

JUSTIFICATION:

Before adding docstrings, the functions worked but were hard to understand for new team members. Without docstrings, users had to read the full code to know inputs, outputs, and usage. After adding Google-style docstrings, each function clearly explains its purpose. Parameters, return values, and examples make the code easy to use and reuse. Overall, docstrings improve readability, reduce confusion, and save development time.

QUESTION 2

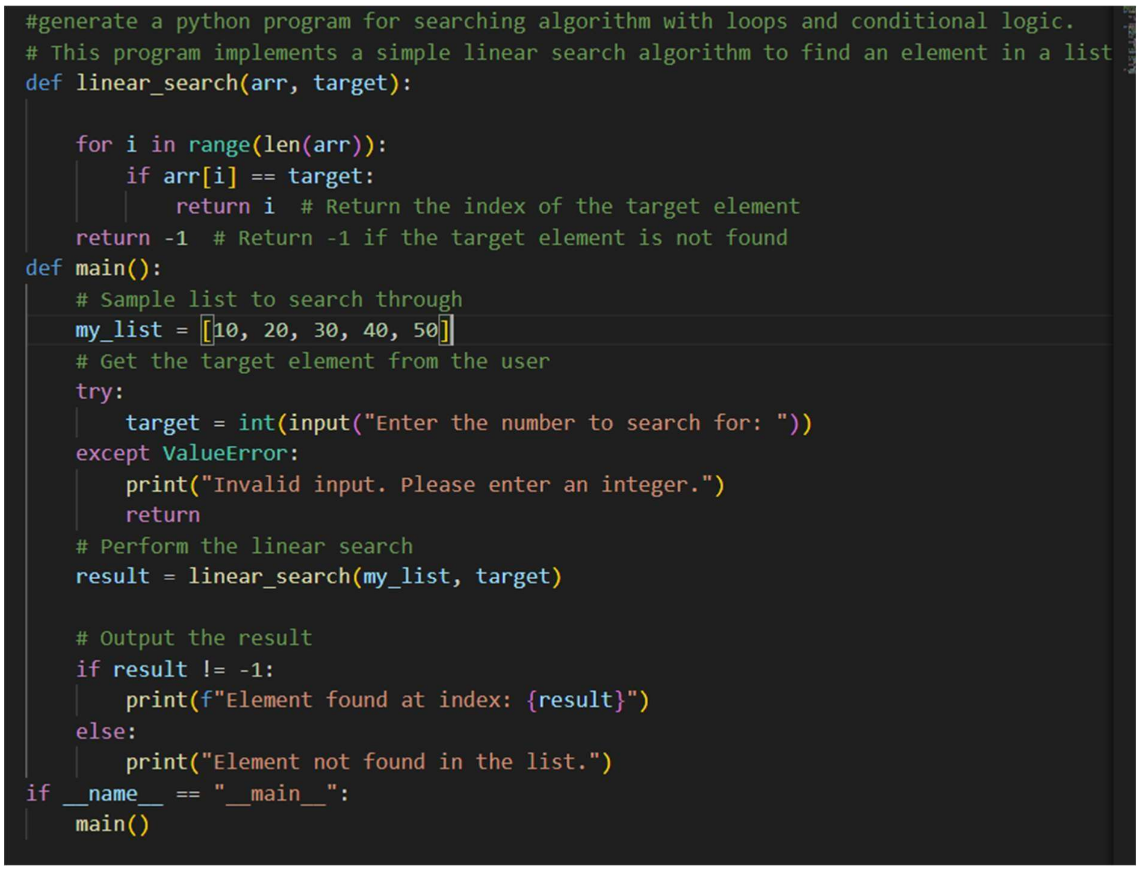
TASK 2: Enhancing Readability Through AI-Generated Inline

Comments

PROMPT:

generate a python program for searching algorithm with loops and conditional logic.

CODE:



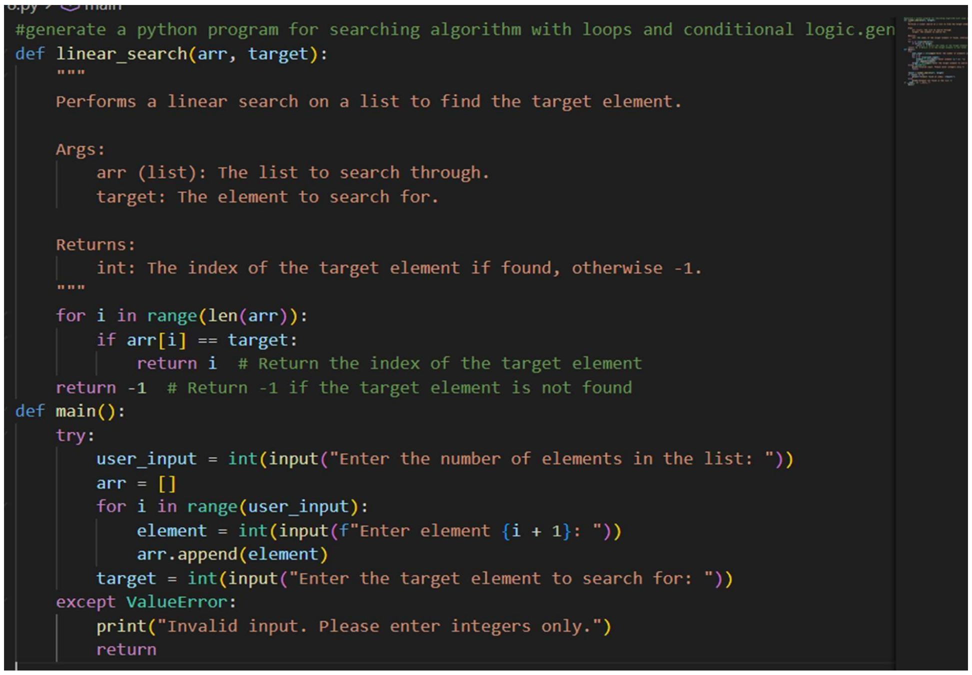
```python
#generate a python program for searching algorithm with loops and conditional logic.
# This program implements a simple linear search algorithm to find an element in a list
def linear_search(arr, target):

    for i in range(len(arr)):
        if arr[i] == target:
            return i  # Return the index of the target element
    return -1  # Return -1 if the target element is not found
def main():
    # Sample list to search through
    my_list = [10, 20, 30, 40, 50]
    # Get the target element from the user
    try:
        target = int(input("Enter the number to search for: "))
    except ValueError:
        print("Invalid input. Please enter an integer.")
        return
    # Perform the linear search
    result = linear_search(my_list, target)

    # Output the result
    if result != -1:
        print(f"Element found at index: {result}")
    else:
        print("Element not found in the list.")
if __name__ == "__main__":
    main()
```

Google-style function docstrings

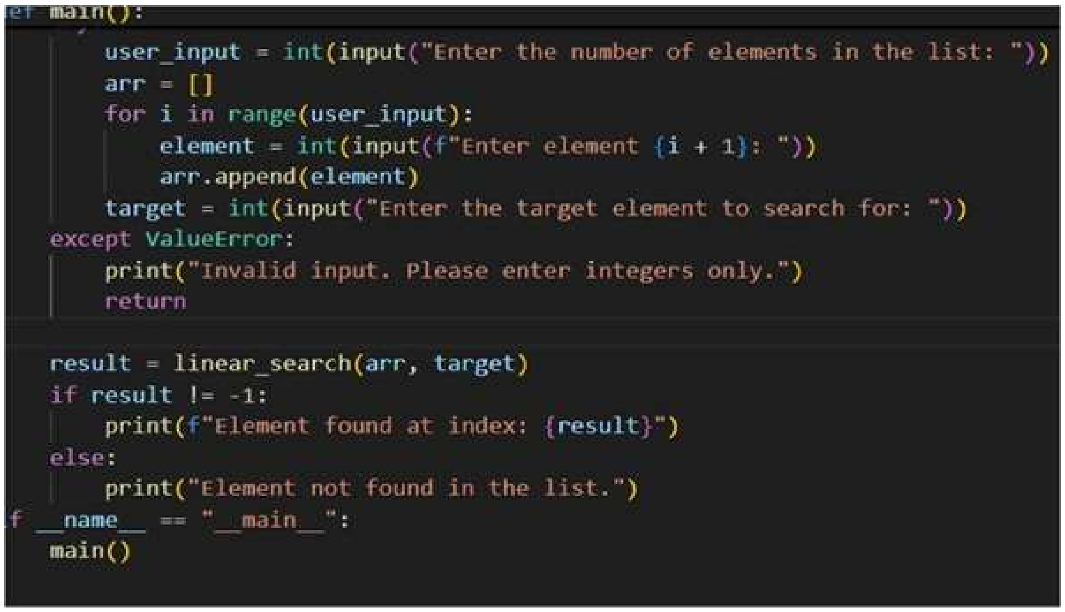
PROMPT:

#generate a python program for searching algorithm with loops and conditional logic.generate a google style doctstring for this program.

CODE:

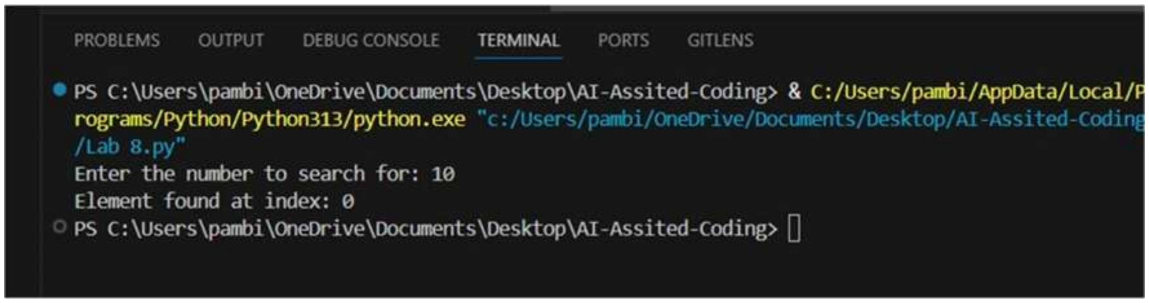


```python
#generate a python program for searching algorithm with loops and conditional logic.gen
def linear_search(arr, target):
    """
    Performs a linear search on a list to find the target element.

    Args:
        arr (list): The list to search through.
        target: The element to search for.

    Returns:
        int: The index of the target element if found, otherwise -1.
    """
    for i in range(len(arr)):
        if arr[i] == target:
            return i  # Return the index of the target element
    return -1  # Return -1 if the target element is not found
def main():
    try:
        user_input = int(input("Enter the number of elements in the list: "))
        arr = []
        for i in range(user_input):
            element = int(input(f"Enter element {i + 1}: "))
            arr.append(element)
        target = int(input("Enter the target element to search for: "))
    except ValueError:
        print("Invalid input. Please enter integers only.")
        return
```



```python
ef main():
        user_input = int(input("Enter the number of elements in the list: "))
        arr = []
        for i in range(user_input):
            element = int(input(f"Enter element {i + 1}: "))
            arr.append(element)
        target = int(input("Enter the target element to search for: "))
    except ValueError:
        print("Invalid input. Please enter integers only.")
        return

    result = linear_search(arr, target)
    if result != -1:
        print(f"Element found at index: {result}")
    else:
        print("Element not found in the list.")
f __name__ == "__main__":
    main()
```

OUTPUT:



```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS    GITLENS

PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assited-Coding> & C:/Users/pambi/AppData/Local/P
rograms/Python/Python313/python.exe "c:/Users/pambi/OneDrive/Documents/Desktop/AI-Assited-Coding
/Lab 8.py"
Enter the number to search for: 10
Element found at index: 0
PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assited-Coding> 
```

JUSTIFICATION:

Using docstrings helps explain the purpose of functions clearly, while inline comments without docstrings focus only on complex logic. Both methods avoid unnecessary comments on simple syntax. This improves readability and makes the code easier to understand. Overall, the program becomes easier to debug and maintain.

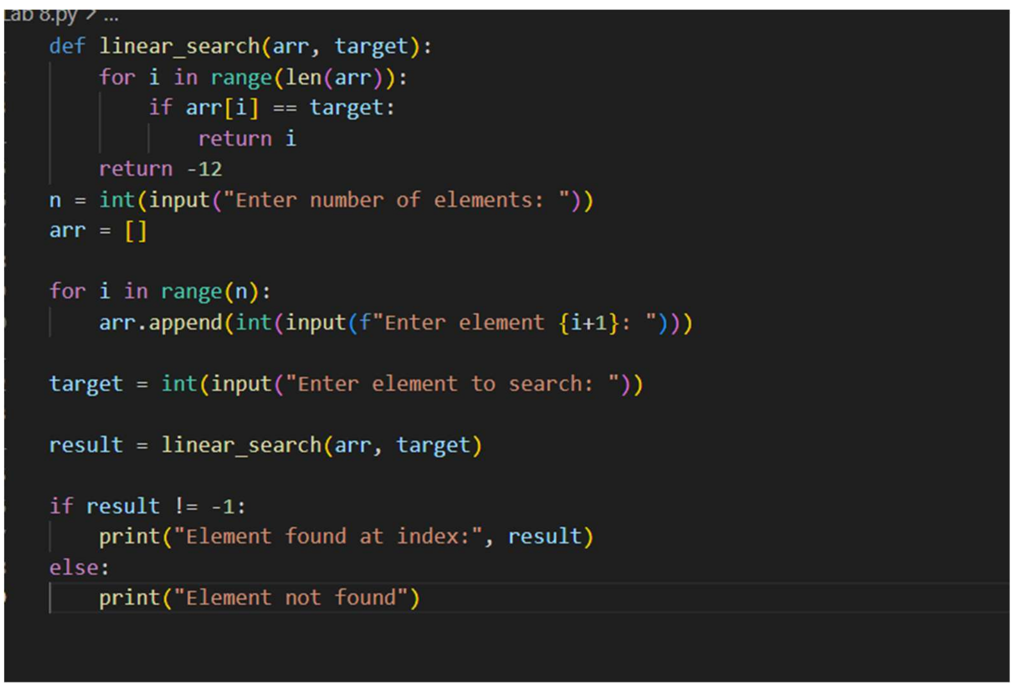
QUESTION 3

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

PROMPT:

Explain the module's purpose, dependencies, main functions or classes, and usage in simple plain text without using a docstring format.

CODE:



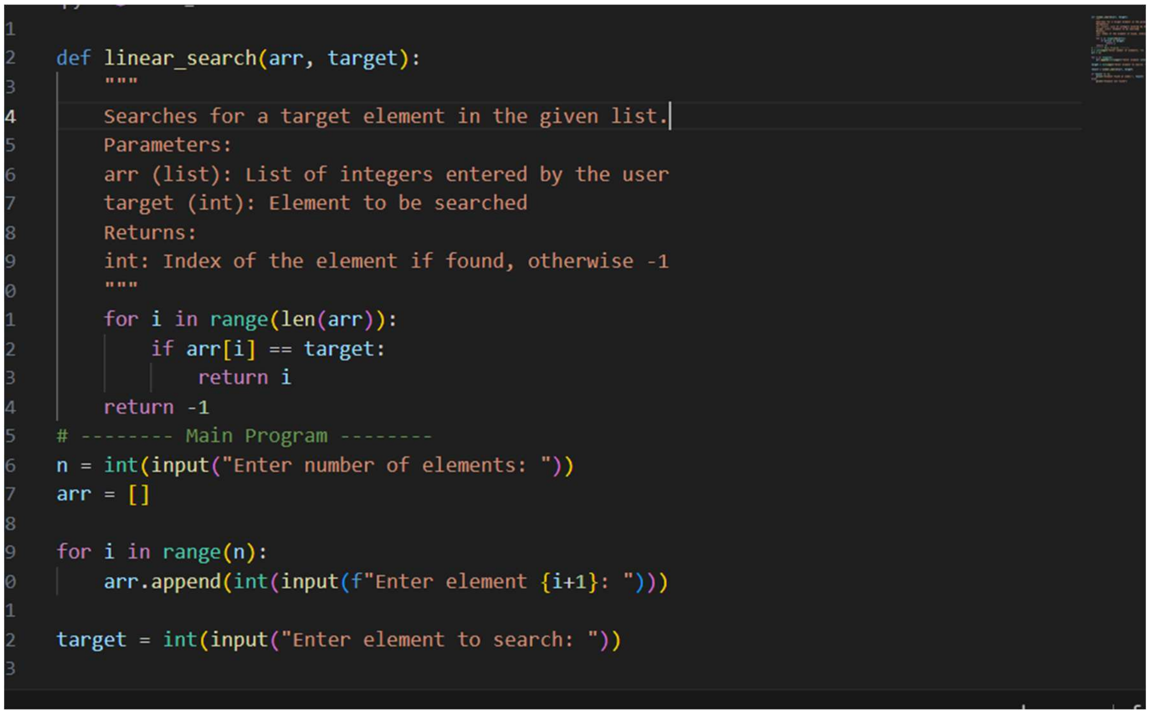
```python
Lab 8.py > ...
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -12
n = int(input("Enter number of elements: "))
arr = []

for i in range(n):
    arr.append(int(input(f"Enter element {i+1}: ")))

target = int(input("Enter element to search: "))

result = linear_search(arr, target)

if result != -1:
    print("Element found at index:", result)
else:
    print("Element not found")
```

Google-style function docstrings

PROMPT:

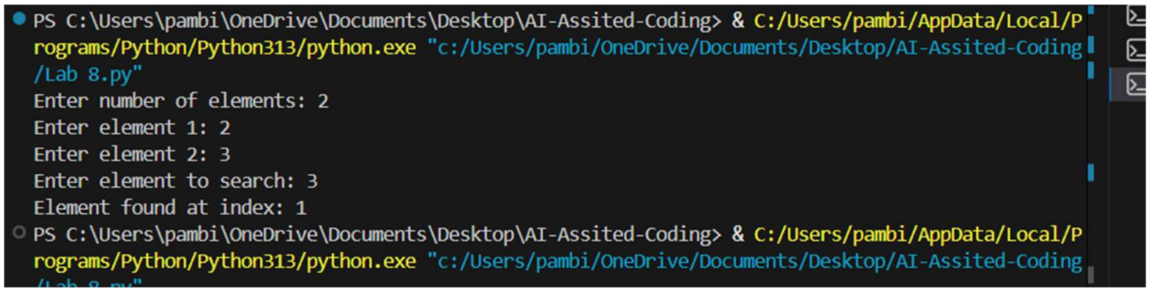
Generate a module-level Python docstring that explains the module’s purpose, required libraries, key functions or classes, and a short usage example.

CODE:



```python
def linear_search(arr, target):
    """
    Searches for a target element in the given list.
    Parameters:
    arr (list): List of integers entered by the user
    target (int): Element to be searched
    Returns:
    int: Index of the element if found, otherwise -1
    """
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1
# -------- Main Program --------
n = int(input("Enter number of elements: "))
arr = []

for i in range(n):
    arr.append(int(input(f"Enter element {i+1}: ")))

target = int(input("Enter element to search: "))
```

OUTPUT:



```
PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assited-Coding> & C:/Users/pambi/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/pambi/OneDrive/Documents/Desktop/AI-Assited-Coding/Lab 8.py"
Enter number of elements: 2
Enter element 1: 2
Enter element 2: 3
Enter element to search: 3
Element found at index: 1
PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assited-Coding> & C:/Users/pambi/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/pambi/OneDrive/Documents/Desktop/AI-Assited-Coding
```

JUSTIFICATION:

Using a module-level docstring tells what the module does and how to use it.It helps new users understand the code without reading everything.Without a docstring, understanding the module takes more time.Docstrings make the code look professional and clear.Overall, they help in easy use and maintenance of the module.

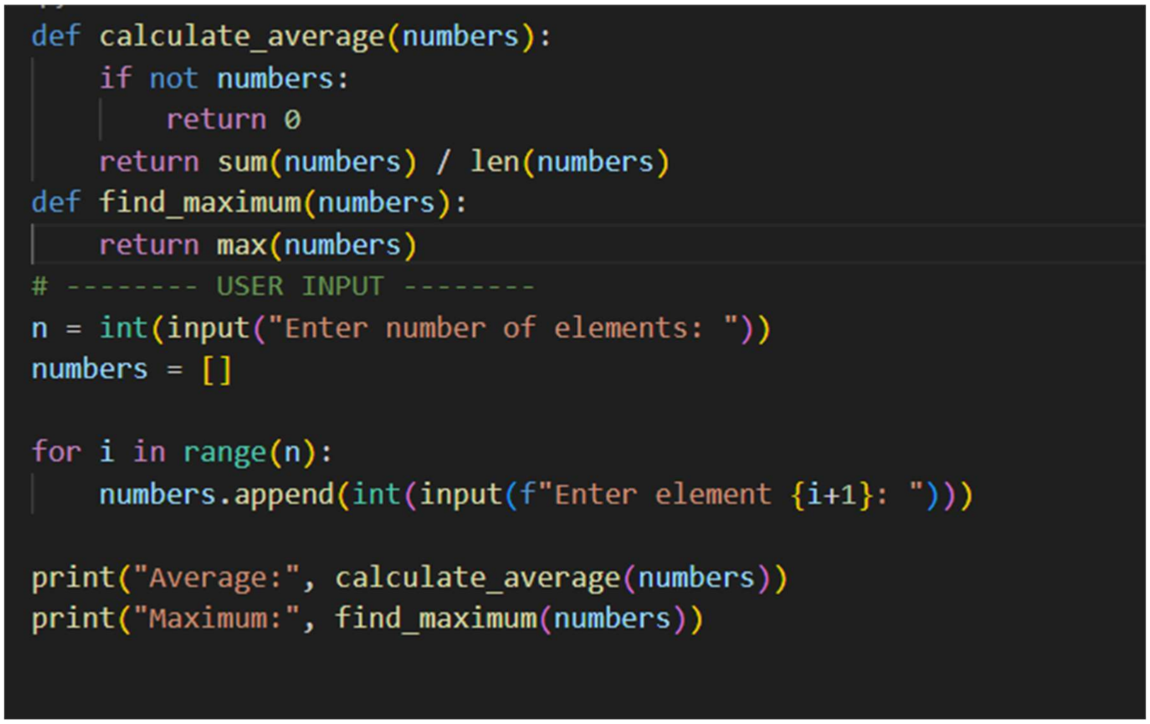
QUESTION 4

TASK 4: Converting Developer Comments into Structured Docstrings

PROMPT:

Given the Python code, rewrite the long inline comments into clear function descriptions in plain text, keep the meaning the same, and remove extra comments from inside the functions.

CODE:



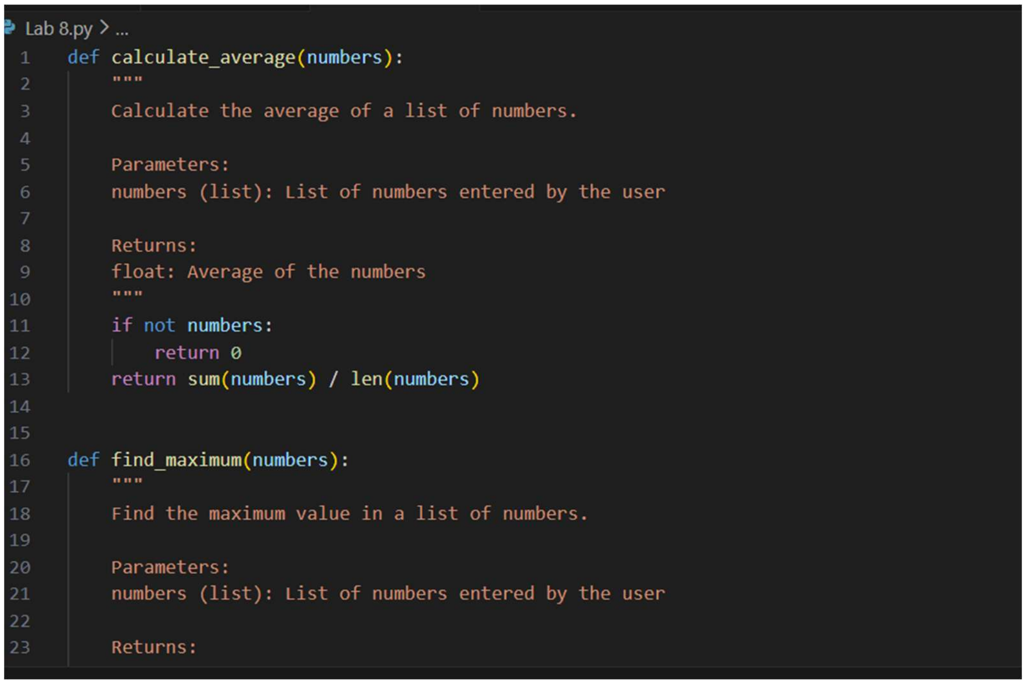
```python
def calculate_average(numbers):
    if not numbers:
        return 0
    return sum(numbers) / len(numbers)
def find_maximum(numbers):
    return max(numbers)
# -------- USER INPUT --------
n = int(input("Enter number of elements: "))
numbers = []

for i in range(n):
    numbers.append(int(input(f"Enter element {i+1}: ")))

print("Average:", calculate_average(numbers))
print("Maximum:", find_maximum(numbers))
```

Google-style function docstrings

PROMPT:

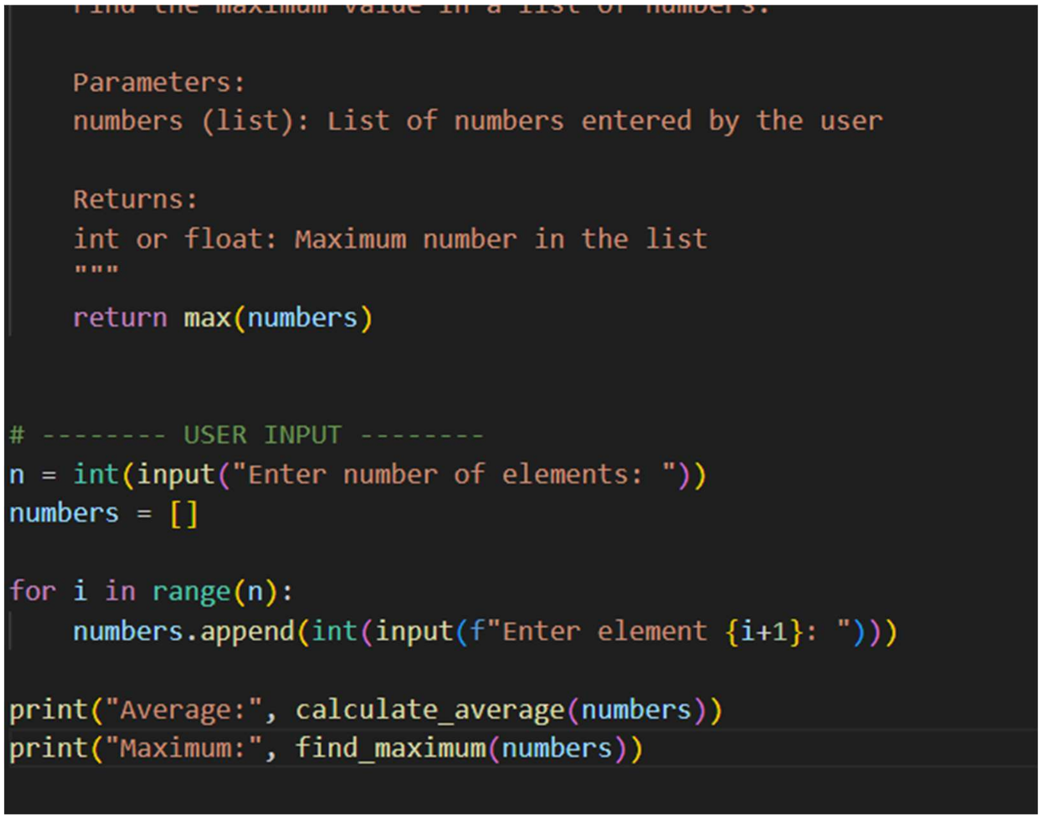
Given the Python code, convert the long inline comments inside functions into clean, structured Google-style or NumPy-style docstrings. Keep the same meaning, remove unnecessary inline comments, and make the code neat and consistent.

CODE:



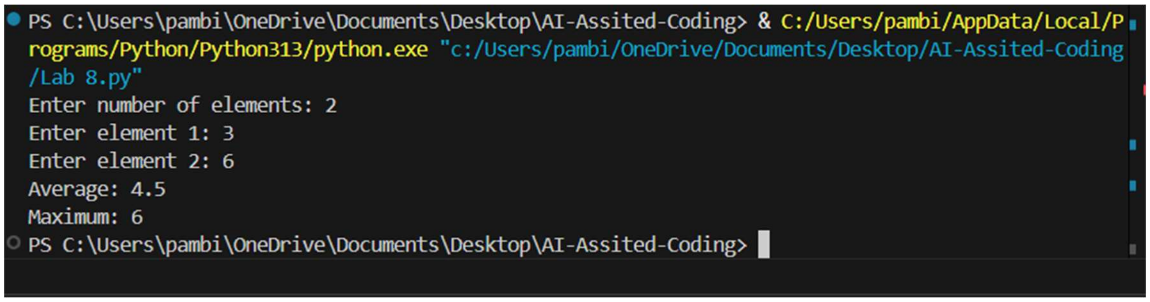
Lab 8.py > ...

```python
def calculate_average(numbers):
    """
    Calculate the average of a list of numbers.

    Parameters:
    numbers (list): List of numbers entered by the user

    Returns:
    float: Average of the numbers
    """
    if not numbers:
        return 0
    return sum(numbers) / len(numbers)


def find_maximum(numbers):
    """
    Find the maximum value in a list of numbers.

    Parameters:
    numbers (list): List of numbers entered by the user

    Returns:
```



```python
    Find the maximum value in a list of numbers.

    Parameters:
    numbers (list): List of numbers entered by the user

    Returns:
    int or float: Maximum number in the list
    """
    return max(numbers)


# -------- USER INPUT --------
n = int(input("Enter number of elements: "))
numbers = []

for i in range(n):
    numbers.append(int(input(f"Enter element {i+1}: ")))

print("Average:", calculate_average(numbers))
print("Maximum:", find_maximum(numbers))
```

OUTPUT:



```
PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assited-Coding> & C:/Users/pambi/AppData/Local/P
rograms/Python/Python313/python.exe "c:/Users/pambi/OneDrive/Documents/Desktop/AI-Assited-Coding
/Lab 8.py"
Enter number of elements: 2
Enter element 1: 3
Enter element 2: 6
Average: 4.5
Maximum: 6
PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assited-Coding>
```

JUSTIFICATION:

Using **docstrings**, long explanations are moved from inside the function into a clear and standard format at the top of each function. This reduces clutter inside the code and makes the purpose, inputs, and outputs easy to understand. It also improves consistency and helps other developers read the code quickly.Without **docstrings**, the code still works but understanding the function logic takes more time because explanations are missing. Overall, using docstrings makes the code cleaner, more readable, and easier to maintain.

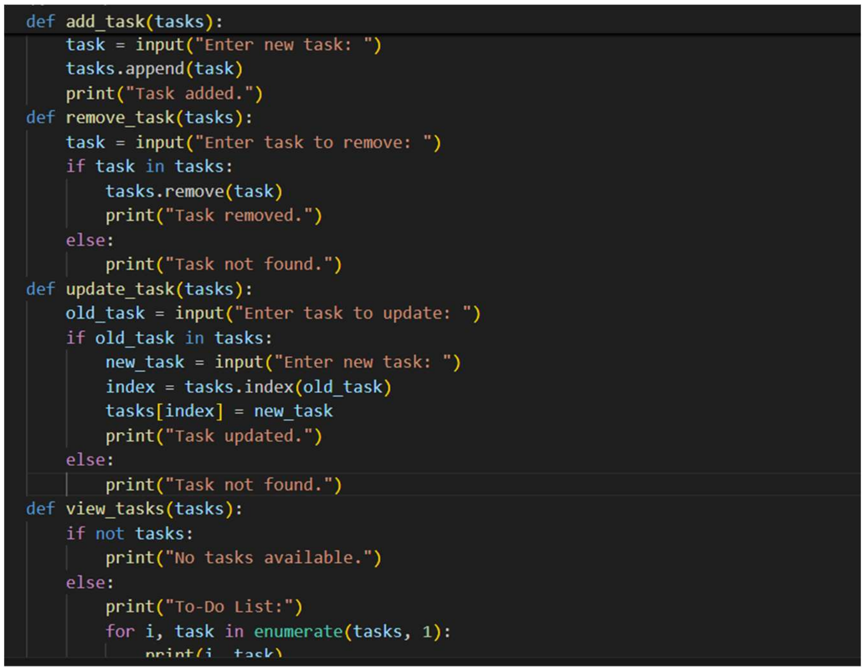
QUESTION 5

TASK 5: Building a Mini Automatic Documentation Generator

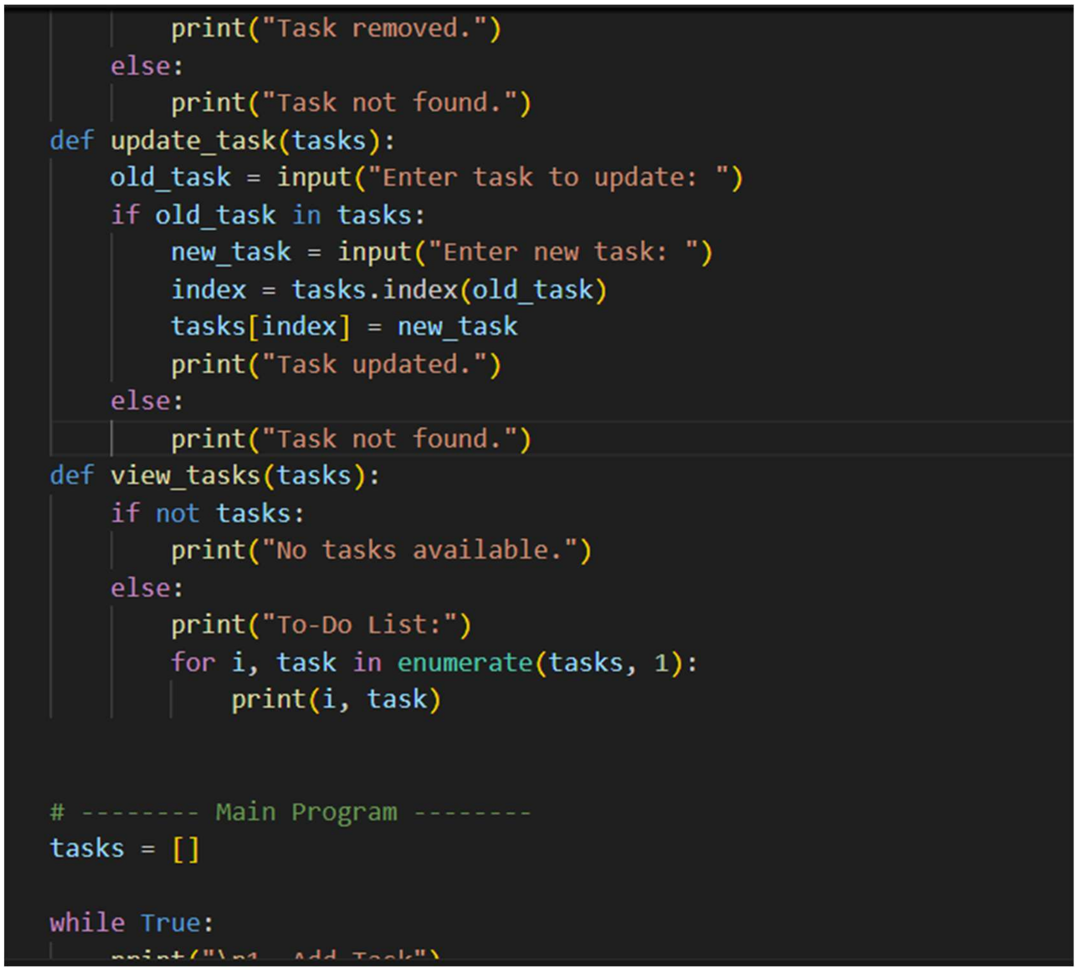
PROMPT:

Create a Python tool that reads a .py file, detects functions and classes, and lists them without adding any docstrings.

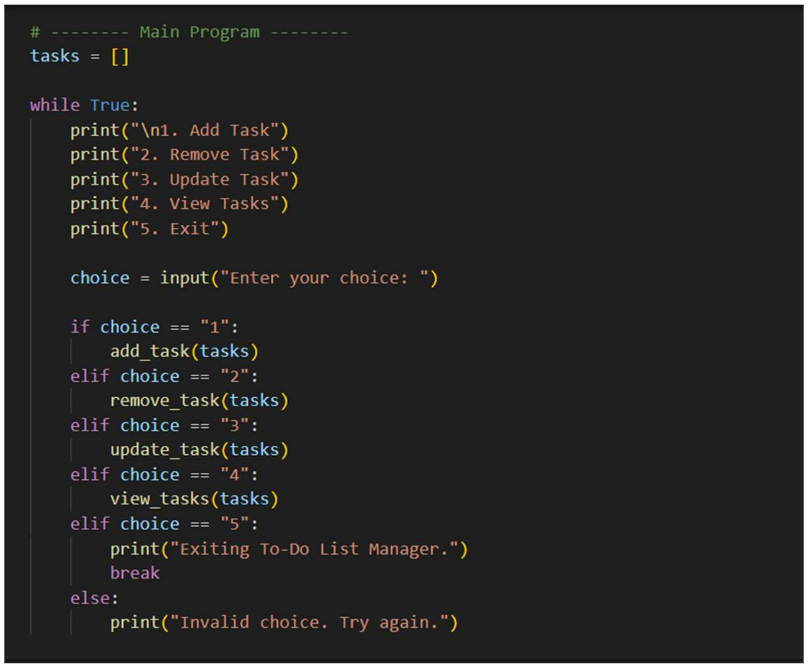
CODE:



```python
def add_task(tasks):
    task = input("Enter new task: ")
    tasks.append(task)
    print("Task added.")
def remove_task(tasks):
    task = input("Enter task to remove: ")
    if task in tasks:
        tasks.remove(task)
        print("Task removed.")
    else:
        print("Task not found.")
def update_task(tasks):
    old_task = input("Enter task to update: ")
    if old_task in tasks:
        new_task = input("Enter new task: ")
        index = tasks.index(old_task)
        tasks[index] = new_task
        print("Task updated.")
    else:
        print("Task not found.")
def view_tasks(tasks):
    if not tasks:
        print("No tasks available.")
    else:
        print("To-Do List:")
        for i, task in enumerate(tasks, 1):
            print(i, task)
```



```python
        print("Task removed.")
    else:
        print("Task not found.")
def update_task(tasks):
    old_task = input("Enter task to update: ")
    if old_task in tasks:
        new_task = input("Enter new task: ")
        index = tasks.index(old_task)
        tasks[index] = new_task
        print("Task updated.")
    else:
        print("Task not found.")
def view_tasks(tasks):
    if not tasks:
        print("No tasks available.")
    else:
        print("To-Do List:")
        for i, task in enumerate(tasks, 1):
            print(i, task)


# -------- Main Program --------
tasks = []

while True:
    print("\n1. Add Task")
```



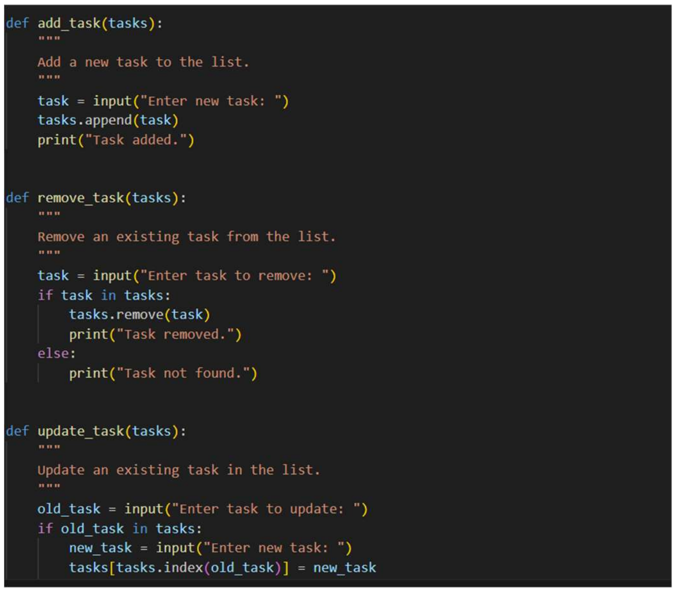
```python
# -------- Main Program --------
tasks = []

while True:
    print("\n1. Add Task")
    print("2. Remove Task")
    print("3. Update Task")
    print("4. View Tasks")
    print("5. Exit")

    choice = input("Enter your choice: ")

    if choice == "1":
        add_task(tasks)
    elif choice == "2":
        remove_task(tasks)
    elif choice == "3":
        update_task(tasks)
    elif choice == "4":
        view_tasks(tasks)
    elif choice == "5":
        print("Exiting To-Do List Manager.")
        break
    else:
        print("Invalid choice. Try again.")
```

Google-style function docstrings

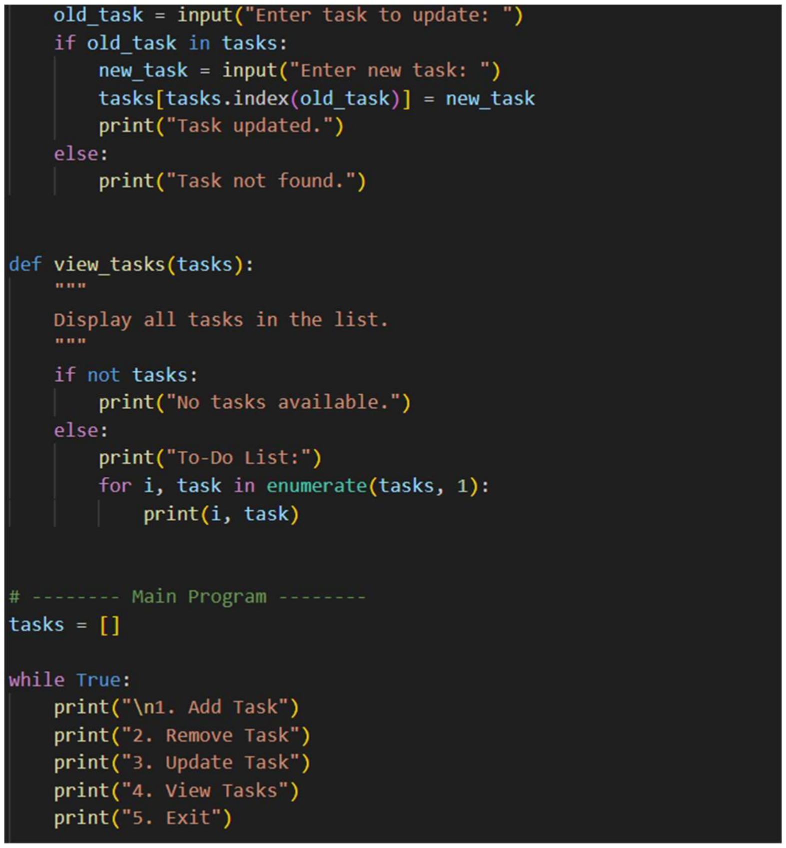
PROMPT:

Create a Python tool that reads a .py file, finds functions and classes, and inserts simple Google-style placeholder docstrings for each one.

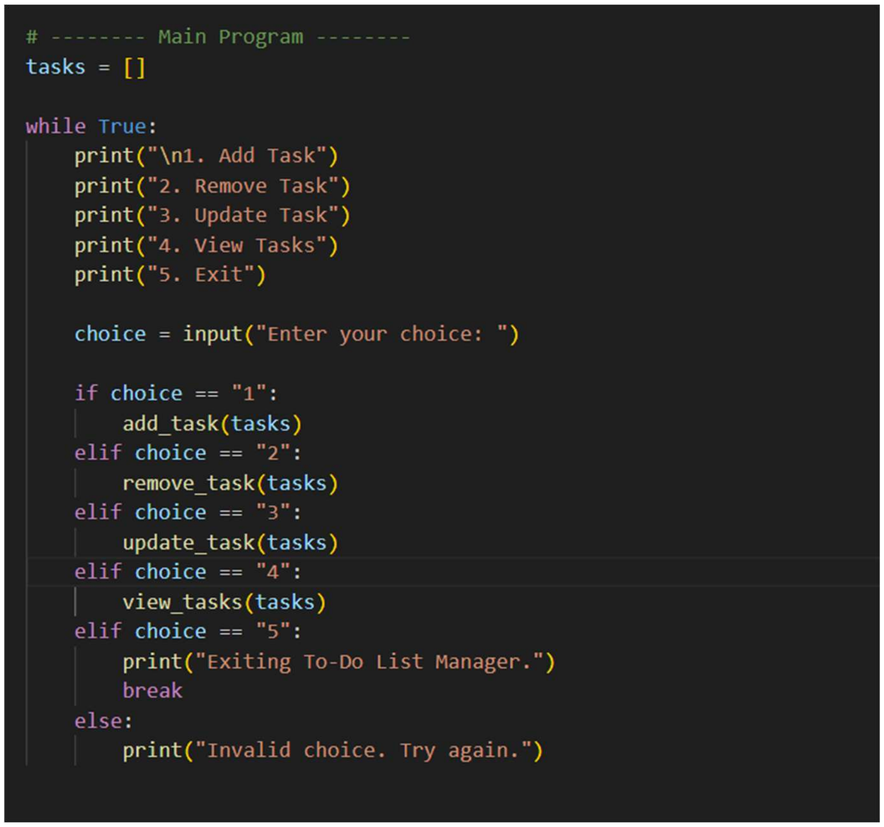
CODE:



```python
def add_task(tasks):
    """
    Add a new task to the list.
    """
    task = input("Enter new task: ")
    tasks.append(task)
    print("Task added.")


def remove_task(tasks):
    """
    Remove an existing task from the list.
    """
    task = input("Enter task to remove: ")
    if task in tasks:
        tasks.remove(task)
        print("Task removed.")
    else:
        print("Task not found.")


def update_task(tasks):
    """
    Update an existing task in the list.
    """
    old_task = input("Enter task to update: ")
    if old_task in tasks:
        new_task = input("Enter new task: ")
        tasks[tasks.index(old_task)] = new_task
```

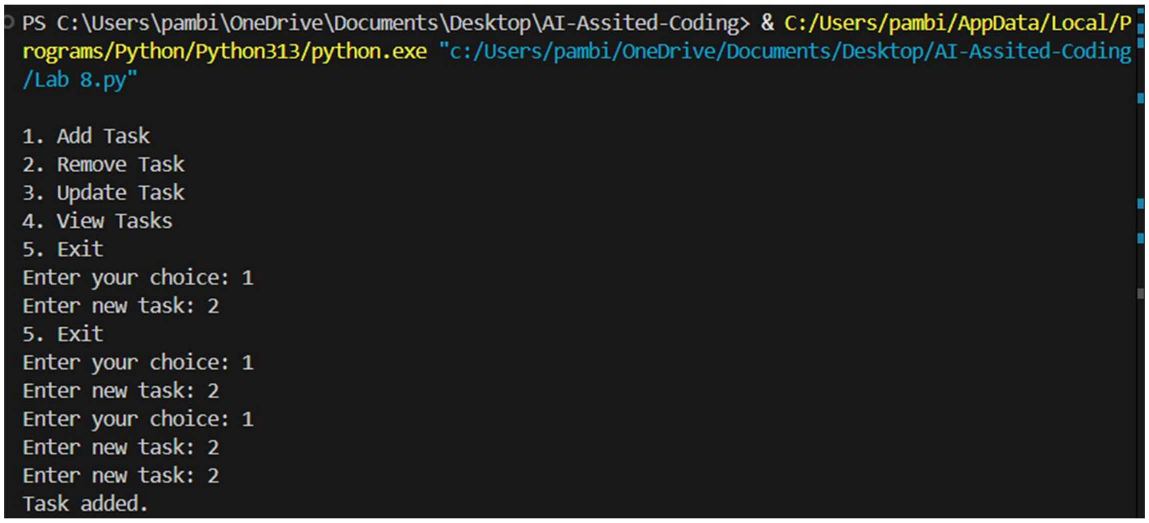


```
    old_task = input("Enter task to update: ")
    if old_task in tasks:
        new_task = input("Enter new task: ")
        tasks[tasks.index(old_task)] = new_task
        print("Task updated.")
    else:
        print("Task not found.")


def view_tasks(tasks):
    """
    Display all tasks in the list.
    """
    if not tasks:
        print("No tasks available.")
    else:
        print("To-Do List:")
        for i, task in enumerate(tasks, 1):
            print(i, task)


# -------- Main Program --------
tasks = []

while True:
    print("\n1. Add Task")
    print("2. Remove Task")
    print("3. Update Task")
    print("4. View Tasks")
    print("5. Exit")
```



```
# -------- Main Program --------
tasks = []

while True:
    print("\n1. Add Task")
    print("2. Remove Task")
    print("3. Update Task")
    print("4. View Tasks")
    print("5. Exit")

    choice = input("Enter your choice: ")

    if choice == "1":
        add_task(tasks)
    elif choice == "2":
        remove_task(tasks)
    elif choice == "3":
        update_task(tasks)
    elif choice == "4":
        view_tasks(tasks)
    elif choice == "5":
        print("Exiting To-Do List Manager.")
        break
    else:
        print("Invalid choice. Try again.")
```

OUTPUT:



```
PS C:\Users\pambi\OneDrive\Documents\Desktop\AI-Assited-Coding> & C:/Users/pambi/AppData/Local/P
rograms/Python/Python313/python.exe "c:/Users/pambi/OneDrive/Documents/Desktop/AI-Assited-Coding
/Lab 8.py"

1. Add Task
2. Remove Task
3. Update Task
4. View Tasks
5. Exit
Enter your choice: 1
Enter new task: 2
5. Exit
Enter your choice: 1
Enter new task: 2
Enter your choice: 1
Enter new task: 2
Enter new task: 2
Task added.
```

JUSTIFICATION:

Using this mini documentation generator **with docstrings**, the tool automatically adds standard Google-style placeholders for functions and classes, saving developers time and ensuring consistent documentation from the start. It helps new files look organized and professional without manual effort.Without this tool (and without docstrings), developers must write documentation manually or skip it, which leads to missing or inconsistent docs. This makes the code harder to understand later. Overall, the utility improves documentation quality, speed, and consistency across the project.