ASSIGNMENT-9.4

SYED NAZIYA PARVEEN

2303A52394

Batch 42

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 operations such as add
def calculator():
    print("Simple Calculator")
    print("Select operation:")
    print("1. Addition (+)")
    print("2. Subtraction (-)")
    print("3. Multiplication (*)")
    print("4. Division (/)")
    choice = input("Enter choice (1/2/3/4): ")
    if choice not in ['1', '2', '3', '4']:
        print("Invalid choice")
        return
    num1 = float(input("Enter first number: "))
    num2 = float(input("Enter second number: "))
    if choice == '1':
        result = num1 + num2
        print(f"{num1} + {num2} = {result}")
    elif choice == '2':
        result = num1 - num2
        print(f"{num1} - {num2} = {result}")
    elif choice == '3':
        result = num1 * num2
        print(f"{num1} * {num2} = {result}")
    elif choice == '4':
        if num2 == 0:
            print("Error: Division by zero is not allowed.")
        else:
            result = num1 / num2
            print(f"{num1} / {num2} = {result}")
# Run the calculator
calculator()
```

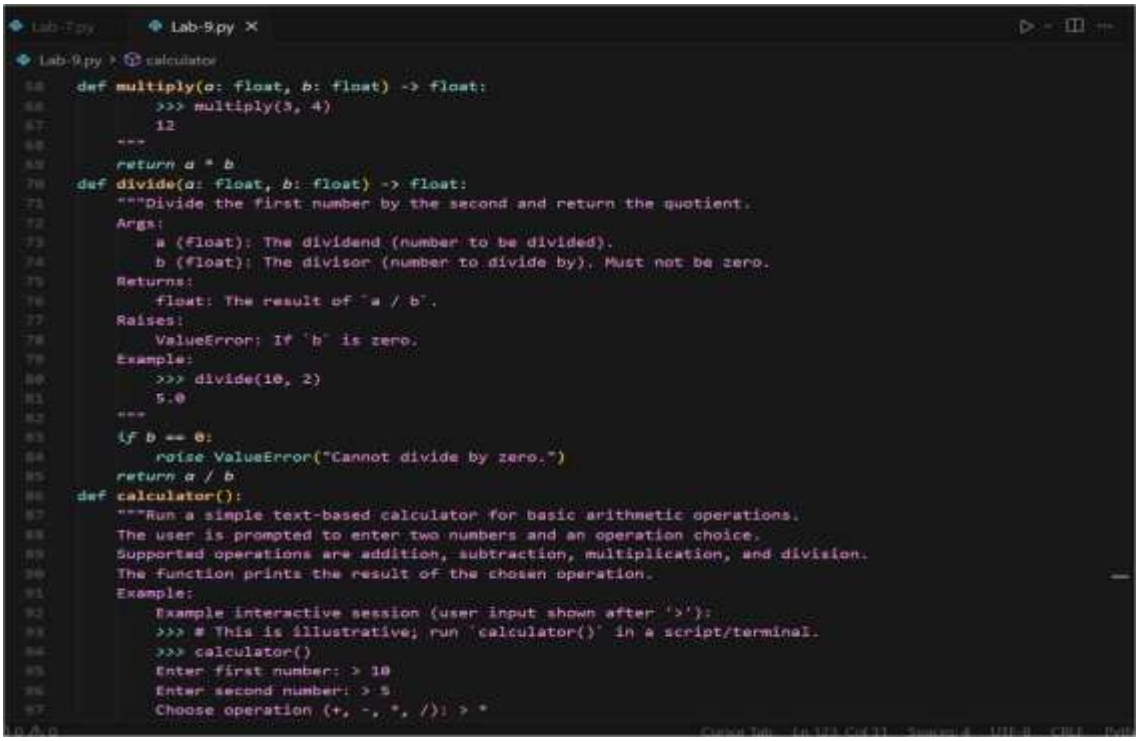
OUTPUT:

```
PS C:\Users\Pranitha Rao\OneDrive\Desktop\AI ASSISTED CODING> & "C:\Users\Pranitha Rao\AppData\Local\
Microsoft\WindowsApps\python3.13.exe" "c:/Users/Pranitha Rao/OneDrive/Desktop/AI ASSISTED CODING/Lab-
9.py"
Simple Calculator
Select operation:
1. Addition (+)
2. Subtraction (-)
3. Multiplication (*)
4. Division (/)
Enter choice (1/2/3/4): 1
Enter first number: 5
Enter second number: 7
5.0 + 7.0 = 12.0
PS C:\Users\Pranitha Rao\OneDrive\Desktop\AI ASSISTED CODING>
```

DOCSTRING 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 style doctstring for each
def add(a: float, b: float) -> float:
    """Add two numbers and return their sum.
    Args:
        a (float): The first addend.
        b (float): The second addend.
    Returns:
        float: The sum of `a` and `b`.
    Example:
        >>> add(3, 5)
        8
    """
    return a + b
def subtract(a: float, b: float) -> float:
    """Subtract the second number from the first and return the result.
    Args:
        a (float): The number to subtract from (minuend).
        b (float): The number to subtract (subtrahend).
    Returns:
        float: The result of `a - b`.
    Example:
        >>> subtract(10, 4)
        6
    """
    return a - b
def multiply(a: float, b: float) -> float:
    """Multiply two numbers and return their product.
    Args:
        a (float): The first factor.
        b (float): The second factor.
    Returns:
        float: The product of `a` and `b`.
    Example:
```



```python
def multiply(a: float, b: float) -> float:
        >>> multiply(3, 4)
        12
    """
    return a * b
def divide(a: float, b: float) -> float:
    """Divide the first number by the second and return the quotient.
    Args:
        a (float): The dividend (number to be divided).
        b (float): The divisor (number to divide by). Must not be zero.
    Returns:
        float: The result of `a / b`.
    Raises:
        ValueError: If `b` is zero.
    Example:
        >>> divide(10, 2)
        5.0
    """
    if b == 0:
        raise ValueError("Cannot divide by zero.")
    return a / b
def calculator():
    """Run a simple text-based calculator for basic arithmetic operations.
    The user is prompted to enter two numbers and an operation choice.
    Supported operations are addition, subtraction, multiplication, and division.
    The function prints the result of the chosen operation.
    Example:
        Example interactive session (user input shown after '>'):
        >>> # This is illustrative; run `calculator()` in a script/terminal.
        >>> calculator()
        Enter first number: > 10
        Enter second number: > 5
        Choose operation (+, -, *, /): > *
```

```python
def calculator():
        Enter second number: > 5
        Choose operation (+, -, *, /): > *
        Result: 50
    """
    print("Simple Calculator")
    try:
        a = float(input("Enter first number: "))
        b = float(input("Enter second number: "))
    except ValueError:
        print("Invalid input. Please enter numeric values.")
        return
    op = input("Choose operation (+, -, *, /): ")
    if op == "+":
        result = add(a, b)
    elif op == "-":
        result = subtract(a, b)
    elif op == "*":
        result = multiply(a, b)
    elif op == "/":
        try:
            result = divide(a, b)
        except ValueError as e:
            print(e)
            return
    else:
        print("Invalid operation.")
        return
    print(f"Result: {result}")
if __name__ == "__main__":
    calculator()
```

OUTPUT:

```
PS C:\Users\Pranitha Rao\OneDrive\Desktop\AI ASSISTED CODING> & "C:\Users\Pranitha Rao\AppData\Local\
Microsoft\WindowsApps\python3.13.exe" "c:/Users/Pranitha Rao/OneDrive/Desktop/AI ASSISTED CODING/Lab-
9.py"
Simple Calculator
Enter first number: 2
Enter second number: 4
Choose operation (+, -, *, /): *
Result: 8.0
PS C:\Users\Pranitha Rao\OneDrive\Desktop\AI ASSISTED CODING>
```

JUSTIFICATION: The shared codebase contained several utility functions without proper documentation, making it di icult for new team members to understand their purpose and usage. To solve this problem, an AI-assisted coding tool was used to automatically generate Google-style docstrings for each function. These docstrings clearly describe the function behavior, parameters with data types, return values, and example usage. This approach improved code readability, reduced the learning curve for new developers, and enhanced the overall maintainability and usability of the codebase.

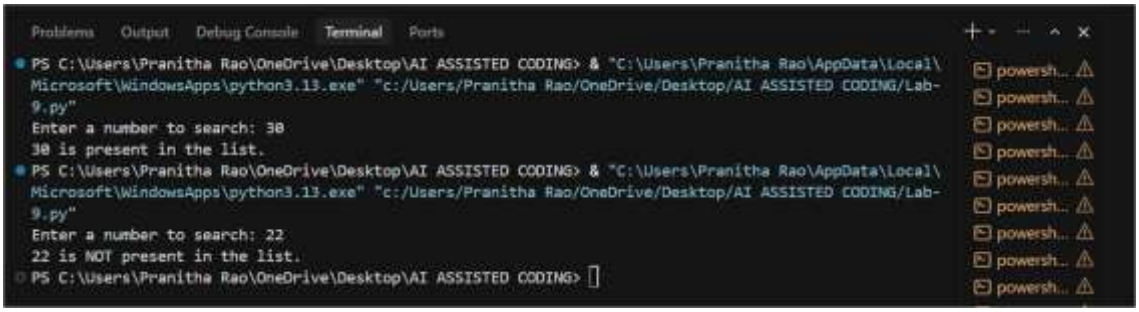
Task 2: Enhancing Readability Through AI-Generated Inline Comments

PROMPT: Write a Python program to search for a number in a list using loops and conditional statements.

CODE:

```python
#Write a Python program to search for a number in a list using loops and conditional statements.
# Sample list
numbers = [10, 20, 30, 40, 50]
# Take input from user
target = int(input("Enter a number to search: "))
found = False  # Flag to check if number is found
# Loop through the list
for num in numbers:
    if num == target:
        found = True
        break
# Print result
if found:
    print(target, "is present in the list.")
else:
    print(target, "is NOT present in the list.")
```

OUTPUT:



DOCSTRING PROMPT: Write a Python program that searches for a value using loops and if conditions.Add a Google-style docstring to explain what the program does, its inputs, and its output.

CODE:

```python
#Write a Python program that searches for a value using loops and if conditions.Add a Google-style docstring
def search_value(numbers, target):
    #Search for a target value in a list of numbers using a loop.
    #This function goes through each number in the list and checks
    #(using an if condition) whether it is equal to the target value.
    #If it finds the target, it returns the index at which it was found.
    #If it does not find the target, it returns -1.
    #Args:
        #numbers (list[int]): List of integers to search through.
        #target (int): The value to search for in the list.
    #Returns:
        #int: The index of the target value if found; -1 otherwise.
    for index in range(len(numbers)):
        if numbers[index] == target:
            return index
    return -1
# Example usage
if __name__ == "__main__":
    nums = [10, 20, 30, 40, 50]
    value_to_find = int(input("Enter a value to search for: "))
    result_index = search_value(nums, value_to_find)
    if result_index != -1:
        print(f"Value {value_to_find} found at index {result_index}.")
    else:
        print(f"Value {value_to_find} not found in the list.")
```

OUTPUT:

```
PS C:\Users\Pranitha Rao\OneDrive\Desktop\AI ASSISTED CODING> & "C:\Users\Pranitha Rao\AppData\Local\
Microsoft\WindowsApps\python3.13.exe" "c:/Users/Pranitha Rao/OneDrive/Desktop/AI ASSISTED CODING/Lab-
9.py"
Enter a value to search for: 10
Value 10 found at index 0.
PS C:\Users\Pranitha Rao\OneDrive\Desktop\AI ASSISTED CODING> & "C:\Users\Pranitha Rao\AppData\Local\
Microsoft\WindowsApps\python3.13.exe" "c:/Users/Pranitha Rao/OneDrive/Desktop/AI ASSISTED CODING/Lab-
9.py"
Enter a value to search for: 11
Value 11 not found in the list.
PS C:\Users\Pranitha Rao\OneDrive\Desktop\AI ASSISTED CODING>
```

JUSTIFICATION: This program demonstrates how AI can improve code readability by adding meaningful inline comments and a clear docstring without cluttering the code. In the first part, inline comments explain only the important logic, such as using a flag and breaking the loop once the target is found, making the searching process easier to understand for future maintainers. In the second part, a Google-style docstring is used to clearly describe the purpose of the function, its inputs, and its return value. Overall, the program uses simple loops and conditional statements to search for a number in a list, while AI-generated comments focus on why the logic is used, helping others understand, debug, and extend the code more easily.

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

PROMPT: Write a simple Python program for a banking system.The program should allow the user to create an account, deposit money, withdraw money, check the balance, and view transaction history.Take input from the user and keep the code minimal and easy to understand.

CODE:

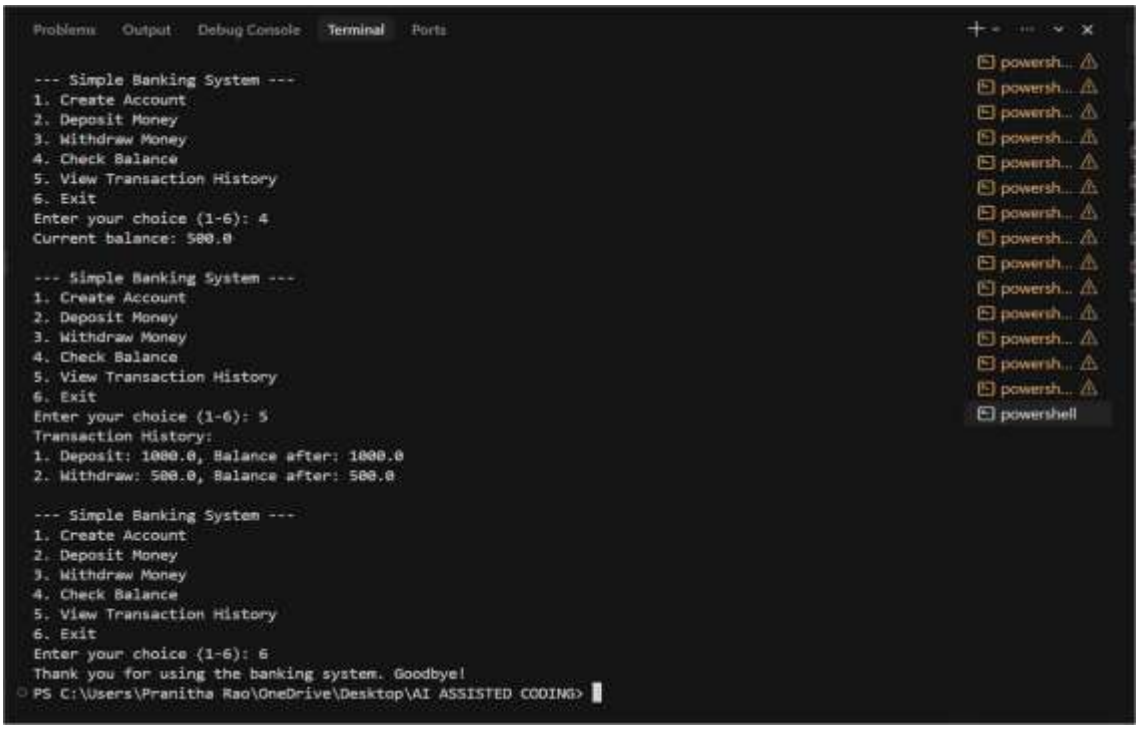
```
Lab-9  C:\Users\Pranitha Rao\OneDrive\Desktop\AI ASSISTED CODING\Lab-7.py
152  # Write a simple Python program for a banking system.The program should allow the user to create an account,
153  # Simple Banking System
154  balance = 0.0
155  transactions = []  # Each item: (type, amount, new_balance)
156  account_created = False
157  def create_account():
158      global account_created, balance, transactions
159      if account_created:
160          print("Account already exists.")
161          return
162      name = input("Enter your name: ")
163      print(f"Account created for {name}.")
164      balance = 0.0
165      transactions = []
166      account_created = True
167  def deposit():
168      global balance
169      if not account_created:
170          print("Create an account first.")
171          return
172      try:
173          amount = float(input("Enter amount to deposit: "))
174          if amount <= 0:
175              print("Amount must be positive.")
176              return
177          balance += amount
178          transactions.append(("Deposit", amount, balance))
179          print(f"Deposited {amount}. New balance: {balance}")
180      except ValueError:
181          print("Invalid amount.")
182  def withdraw():
183      global balance
184      if not account_created:
```

```python
def withdraw():
        print("Create an account first.")
        return
    try:
        amount = float(input("Enter amount to withdraw: "))
        if amount <= 0:
            print("Amount must be positive.")
            return
        if amount > balance:
            print("Insufficient funds.")
            return
        balance -= amount
        transactions.append(("Withdraw", amount, balance))
        print(f"Withdrew {amount}. New balance: {balance}")
    except ValueError:
        print("Invalid amount.")
def check_balance():
    if not account_created:
        print("Create an account first.")
        return
    print(f"Current balance: {balance}")
def view_transactions():
    if not account_created:
        print("Create an account first.")
        return
    if not transactions:
        print("No transactions yet.")
        return
    print("Transaction History:")
    for i, (t_type, amount, bal_after) in enumerate(transactions, start=1):
        print(f"{i}. {t_type}: {amount}, Balance after: {bal_after}")
def show_menu():
    print("\n--- Simple Banking System ---")
```

```python
def show_menu():
    print("\n--- Simple Banking System ---")
    print("1. Create Account")
    print("2. Deposit Money")
    print("3. Withdraw Money")
    print("4. Check Balance")
    print("5. View Transaction History")
    print("6. Exit")
while True:
    show_menu()
    choice = input("Enter your choice (1-6): ")
    if choice == "1":
        create_account()
    elif choice == "2":
        deposit()
    elif choice == "3":
        withdraw()
    elif choice == "4":
        check_balance()
    elif choice == "5":
        view_transactions()
    elif choice == "6":
        print("Thank you for using the banking system. Goodbye!")
        break
    else:
        print("Invalid choice. Please enter a number from 1 to 6.")
```

OUTPUT:

```
PS C:\Users\Pranitha Rao\OneDrive\Desktop\AI ASSISTED CODING> & "C:\Users\Pranitha Rao\AppData\Local\Microsoft\WindowsApps\python3.13.exe" "c:/Users/Pranitha Rao/OneDrive/Desktop/AI ASSISTED CODING/Lab-9.py"

--- Simple Banking System ---
1. Create Account
2. Deposit Money
3. Withdraw Money
4. Check Balance
5. View Transaction History
6. Exit
Enter your choice (1-6): 1
Enter your name: Pranitha
Account created for Pranitha.

--- Simple Banking System ---
1. Create Account
2. Deposit Money
3. Withdraw Money
4. Check Balance
5. View Transaction History
6. Exit
Enter your choice (1-6): 2
Enter amount to deposit: 1000
Deposited 1000.0. New balance: 1000.0

--- Simple Banking System ---
1. Create Account
2. Deposit Money
3. Withdraw Money
4. Check Balance
5. View Transaction History
6. Exit
Enter your choice (1-6): 3
Enter amount to withdraw: 500
Withdrew 500.0. New balance: 500.0
```

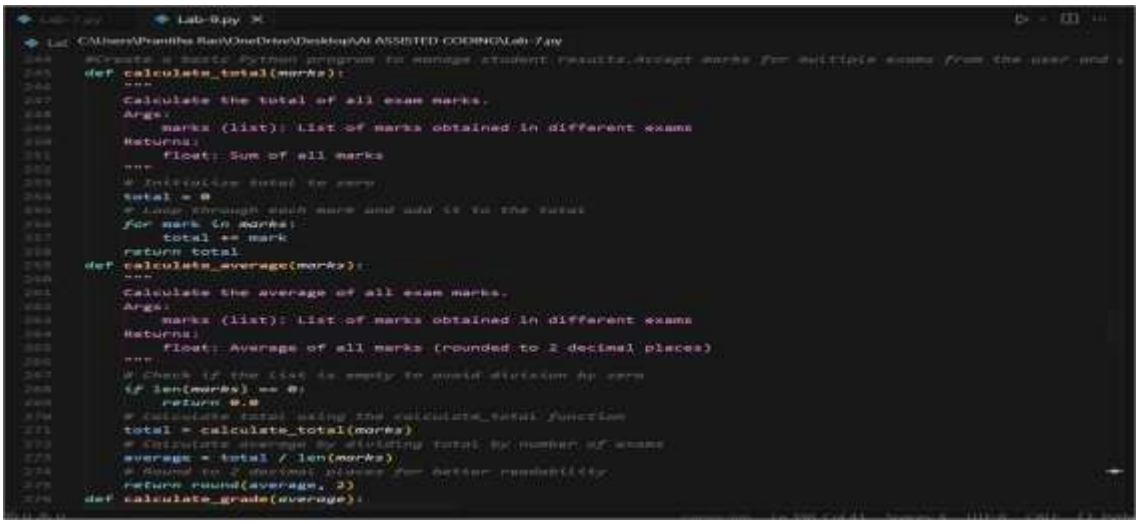


JUSTIFICATION: This program implements a simple banking system using Python functions and user input. It allows a user to create an account and then perform basic banking operations such as depositing money, withdrawing money, checking the current balance, and viewing transaction history. The program uses global variables to store the account balance and a list of transactions, and each function first checks whether an account has been created before performing any operation. A menu-driven loop lets the user choose actions repeatedly until they decide to exit. Overall, the code is minimal, easy to understand, and clearly shows how functions, conditionals, and loops work together in a real-world example.

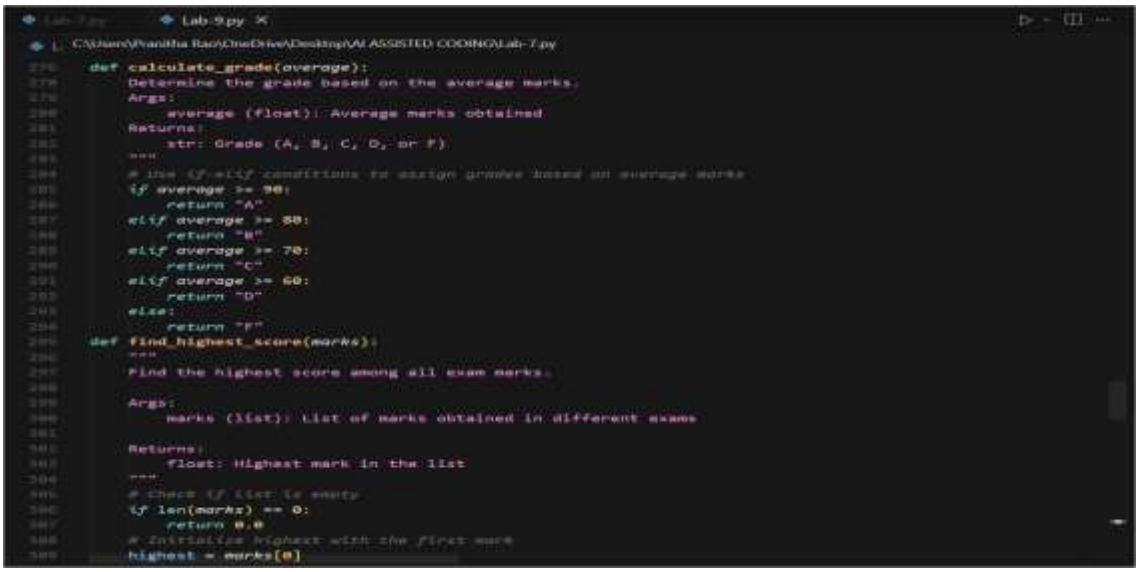
Task 4: Converting Developer Comments into Structured Docstrings

PROMPT: Create a basic Python program to manage student results.Accept marks for multiple exams from the user and use separate functions to find the total, average, grade, highest score, lowest score, and show the final report.Add detailed comments to explain the logic and keep the program simple.

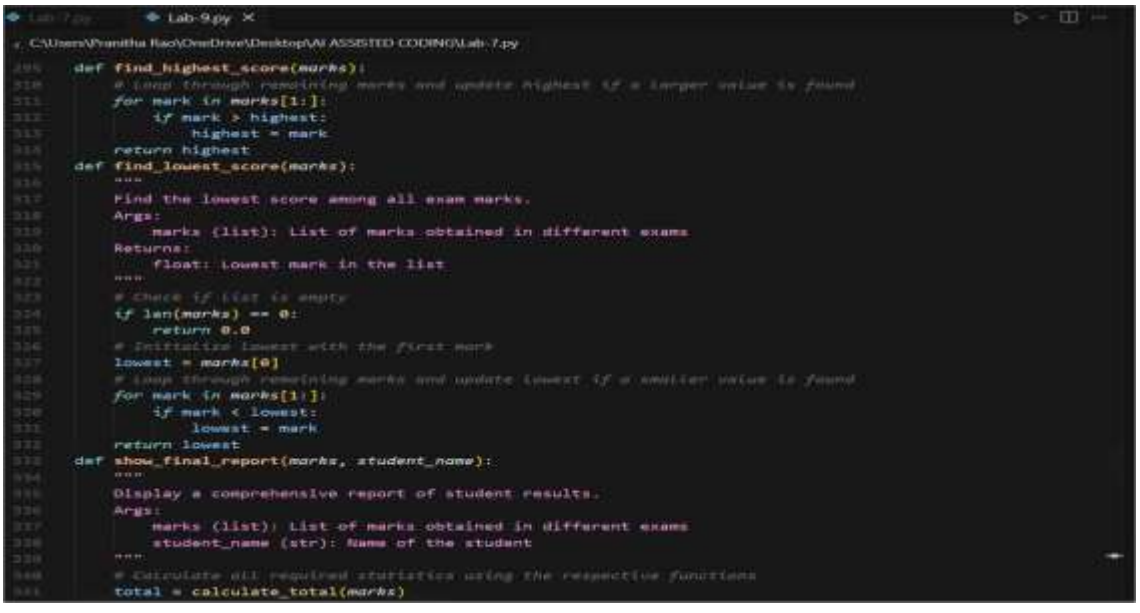
CODE:



```python
# Create a basic Python program to manage student results.Accept marks for multiple exams from the user and c
def calculate_total(marks):
    """
    Calculate the total of all exam marks.
    Args:
        marks (list): List of marks obtained in different exams
    Returns:
        float: Sum of all marks
    """
    # Initialize total to zero
    total = 0
    # Loop through each mark and add it to the total
    for mark in marks:
        total += mark
    return total
def calculate_average(marks):
    """
    Calculate the average of all exam marks.
    Args:
        marks (list): List of marks obtained in different exams
    Returns:
        float: Average of all marks (rounded to 2 decimal places)
    """
    # Check if the list is empty to avoid division by zero
    if len(marks) == 0:
        return 0.0
    # Calculate total using the calculate_total function
    total = calculate_total(marks)
    # Calculate average by dividing total by number of exams
    average = total / len(marks)
    # Round to 2 decimal places for better readability
    return round(average, 2)
def calculate_grade(average):
```

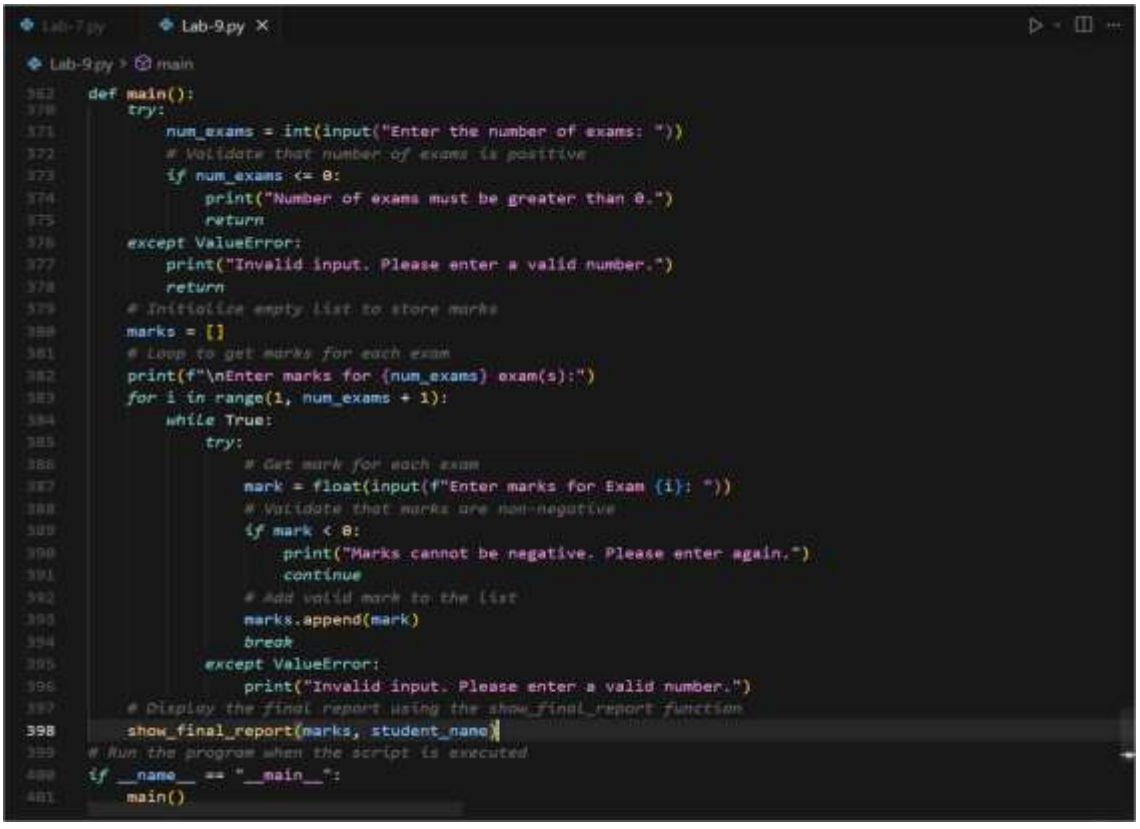


```python
def calculate_grade(average):
    Determine the grade based on the average marks.
    Args:
        average (float): Average marks obtained
    Returns:
        str: Grade (A, B, C, D, or F)
    """
    # Use if-elif conditions to assign grades based on average marks
    if average >= 90:
        return "A"
    elif average >= 80:
        return "B"
    elif average >= 70:
        return "C"
    elif average >= 60:
        return "D"
    else:
        return "F"
def find_highest_score(marks):
    """
    Find the highest score among all exam marks.

    Args:
        marks (list): List of marks obtained in different exams

    Returns:
        float: Highest mark in the list
    """
    # Check if list is empty
    if len(marks) == 0:
        return 0.0
    # Initialize highest with the first mark
    highest = marks[0]
```



```python
def find_highest_score(marks):
    # Loop through remaining marks and update highest if a larger value is found
    for mark in marks[1:]:
        if mark > highest:
            highest = mark
    return highest
def find_lowest_score(marks):
    """
    Find the lowest score among all exam marks.
    Args:
        marks (list): List of marks obtained in different exams
    Returns:
        float: Lowest mark in the list
    """
    # Check if list is empty
    if len(marks) == 0:
        return 0.0
    # Initialize lowest with the first mark
    lowest = marks[0]
    # Loop through remaining marks and update lowest if a smaller value is found
    for mark in marks[1:]:
        if mark < lowest:
            lowest = mark
    return lowest
def show_final_report(marks, student_name):
    """
    Display a comprehensive report of student results.
    Args:
        marks (list): List of marks obtained in different exams
        student_name (str): Name of the student
    """
    # Calculate all required statistics using the respective functions
    total = calculate_total(marks)
```

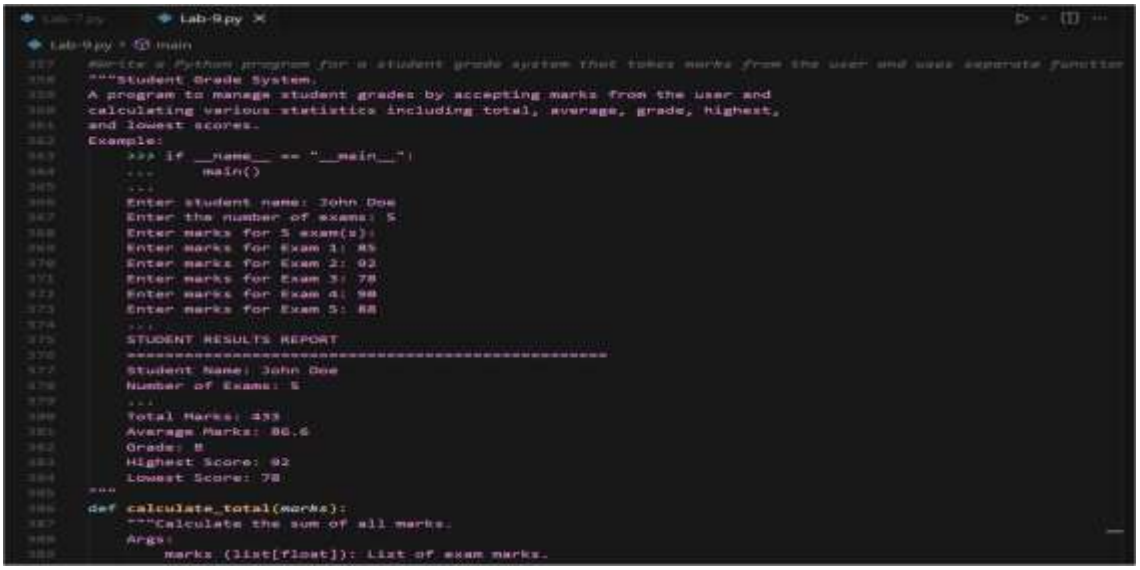
```
def show_final_report(marks, student_name):
    total = calculate_total(marks)
    average = calculate_average(marks)
    grade = calculate_grade(average)
    highest = find_highest_score(marks)
    lowest = find_lowest_score(marks)
    # Display the formatted report
    print("\n" + "="*50)
    print("STUDENT RESULTS REPORT")
    print("="*50)
    print(f"Student Name: {student_name}")
    print(f"Number of Exams: {len(marks)}")
    print(f"\nExam Marks:")
    # Display each exam mark with its number
    for i, mark in enumerate(marks, start=1):
        print(f"  Exam {i}: {mark}")
    print(f"\nTotal Marks: {total}")
    print(f"Average Marks: {average}")
    print(f"Grade: {grade}")
    print(f"Highest Score: {highest}")
    print(f"Lowest Score: {lowest}")
    print("="*50 + "\n")
def main():
    """
    Main function to run the student results management program.
    Handles user input and coordinates all functions.
    """
    # Get student name from user
    student_name = input("Enter student name: ")
    # Get number of exams from user
    try:
        num_exams = int(input("Enter the number of exams: "))
        # Validate that number of exams is positive
```



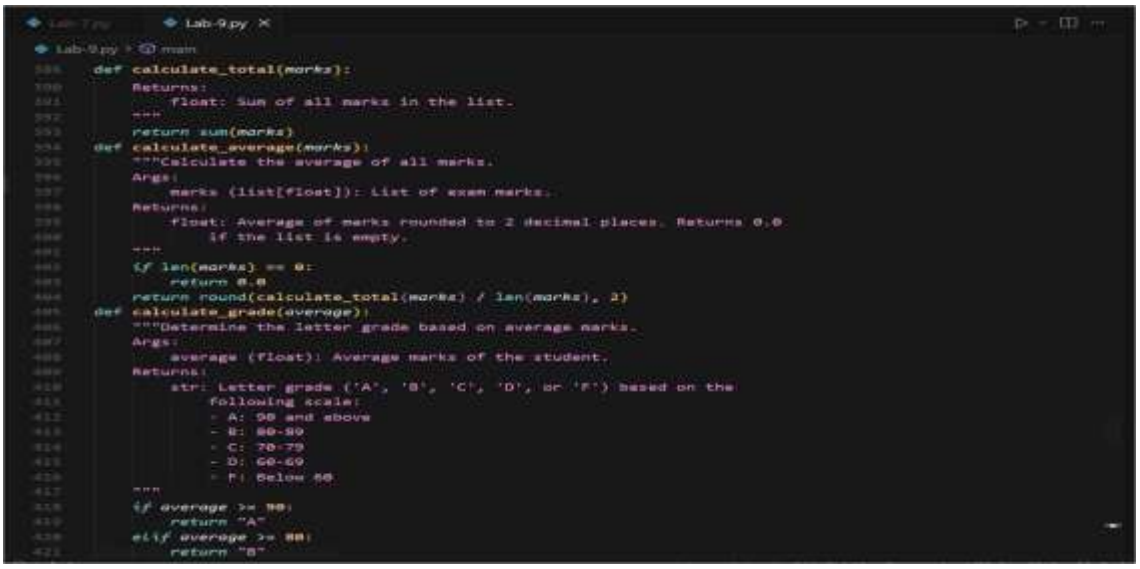
```
def main():
    try:
        num_exams = int(input("Enter the number of exams: "))
        # Validate that number of exams is positive
        if num_exams <= 0:
            print("Number of exams must be greater than 0.")
            return
    except ValueError:
        print("Invalid input. Please enter a valid number.")
        return
    # Initialize empty list to store marks
    marks = []
    # Loop to get marks for each exam
    print(f"\nEnter marks for {num_exams} exam(s):")
    for i in range(1, num_exams + 1):
        while True:
            try:
                # Get mark for each exam
                mark = float(input(f"Enter marks for Exam {i}: "))
                # Validate that marks are non-negative
                if mark < 0:
                    print("Marks cannot be negative. Please enter again.")
                    continue
                # Add valid mark to the list
                marks.append(mark)
                break
            except ValueError:
                print("Invalid input. Please enter a valid number.")
    # Display the final report using the show_final_report function
    show_final_report(marks, student_name)
# Run the program when the script is executed
if __name__ == "__main__":
    main()
```

DOCSTRING PROMPT: Write a Python program for a student grade system that takes marks from the user and uses separate functions to calculate total marks, average, grade, highest and lowest scores, and display a final report. Convert all explanatory comments into clear Google-style or NumPy-style docstrings, remove unnecessary inline comments, include a proper module-level docstring with an example, and keep the code minimal with standardized documentation.

CODE:



```
Lab-9.py
Lab-9.py > main
# Write a Python program for a student grade system that takes marks from the user and uses separate functio
"""Student Grade System.
A program to manage student grades by accepting marks from the user and
calculating various statistics including total, average, grade, highest,
and lowest scores.
Example:
    >>> if __name__ == "__main__":
    ...     main()
    ...
    Enter student name: John Doe
    Enter the number of exams: 5
    Enter marks for 5 exam(s):
    Enter marks for Exam 1: 85
    Enter marks for Exam 2: 92
    Enter marks for Exam 3: 78
    Enter marks for Exam 4: 90
    Enter marks for Exam 5: 88
    ...
    STUDENT RESULTS REPORT
    ==================================================
    Student Name: John Doe
    Number of Exams: 5
    ...
    Total Marks: 433
    Average Marks: 86.6
    Grade: B
    Highest Score: 92
    Lowest Score: 78
"""
def calculate_total(marks):
    """Calculate the sum of all marks.
    Args:
        marks (list[float]): List of exam marks.
```



```
Lab-9.py
Lab-9.py > main
def calculate_total(marks):
    Returns:
        float: Sum of all marks in the list.
    """
    return sum(marks)
def calculate_average(marks):
    """Calculate the average of all marks.
    Args:
        marks (list[float]): List of exam marks.
    Returns:
        float: Average of marks rounded to 2 decimal places. Returns 0.0
            if the list is empty.
    """
    if len(marks) == 0:
        return 0.0
    return round(calculate_total(marks) / len(marks), 2)
def calculate_grade(average):
    """Determine the letter grade based on average marks.
    Args:
        average (float): Average marks of the student.
    Returns:
        str: Letter grade ('A', 'B', 'C', 'D', or 'F') based on the
            following scale:
            - A: 90 and above
            - B: 80-89
            - C: 70-79
            - D: 60-69
            - F: Below 60
    """
    if average >= 90:
        return "A"
    elif average >= 80:
        return "B"
```

```
Lab-9.py
Lab-9.py > main
def calculate_grade(average):
    elif average >= 70:
        return "C"
    elif average >= 60:
        return "D"
    else:
        return "F"
def find_highest_score(marks):
    """Find the highest score from the list of marks.
    Args:
        marks (list[float]): List of exam marks.
    Returns:
        float: Highest mark in the list. Returns 0.0 if the list is empty.
    """
    if len(marks) == 0:
        return 0.0
    return max(marks)
def find_lowest_score(marks):
    """Find the lowest score from the list of marks.
    Args:
        marks (list[float]): List of exam marks.
    Returns:
        float: Lowest mark in the list. Returns 0.0 if the list is empty.
    """
    if len(marks) == 0:
        return 0.0
    return min(marks)
def display_final_report(marks, student_name):
    """Display a formatted report with all student statistics.
    Args:
        marks (list[float]): List of exam marks.
        student_name (str): Name of the student.
    Returns:
```

```python
def display_final_report(marks, student_name):
        None: This function prints the report to stdout.
    """
    total = calculate_total(marks)
    average = calculate_average(marks)
    grade = calculate_grade(average)
    highest = find_highest_score(marks)
    lowest = find_lowest_score(marks)
    print("\n" + "=" * 50)
    print("STUDENT RESULTS REPORT")
    print("=" * 50)
    print(f"Student Name: {student_name}")
    print(f"Number of Exams: {len(marks)}")
    print(f"\nExam Marks:")
    for i, mark in enumerate(marks, start=1):
        print(f"  Exam {i}: {mark}")
    print(f"\nTotal Marks: {total}")
    print(f"Average Marks: {average}")
    print(f"Grade: {grade}")
    print(f"Highest Score: {highest}")
    print(f"Lowest Score: {lowest}")
    print("=" * 50 + "\n")
def get_marks_from_user(num_exams):
    """Collect exam marks from the user via input.
    Args:
        num_exams (int): Number of exams to collect marks for.
    Returns:
        list[float]: List of valid exam marks entered by the user.
    """
    marks = []
    print(f"\nEnter marks for {num_exams} exam(s):")
    for i in range(1, num_exams + 1):
        while True:
```

```python
def get_marks_from_user(num_exams):
            try:
                mark = float(input(f"Enter marks for Exam {i}: "))
                if mark < 0:
                    print("Marks cannot be negative. Please enter again.")
                    continue
                marks.append(mark)
                break
            except ValueError:
                print("Invalid input. Please enter a valid number.")
    return marks
def main():
    """Main function to run the student grade system.
    Prompts user for student name and number of exams, collects marks,
    and displays the final report.
    """
    student_name = input("Enter student name: ")
    try:
        num_exams = int(input("Enter the number of exams: "))
        if num_exams <= 0:
            print("Number of exams must be greater than 0.")
            return
    except ValueError:
        print("Invalid input. Please enter a valid number.")
        return
    marks = get_marks_from_user(num_exams)
    display_final_report(marks, student_name)
if __name__ == "__main__":
    main()
```

OUTPUT:

```
PS C:\Users\Pranitha Rao\OneDrive\Desktop\AI ASSISTED CODING> & "C:\Users\Pranitha Rao\AppData\Local\
Microsoft\WindowsApps\python3.13.exe" "c:/Users/Pranitha Rao/OneDrive/Desktop/AI ASSISTED CODING/Lab-
9.py"
Enter student name: Pranitha
Enter the number of exams: 5

Enter marks for 5 exam(s):
Enter marks for Exam 1: 88
Enter marks for Exam 2: 65
Enter marks for Exam 3: 77
Enter marks for Exam 4: 98
Enter marks for Exam 5: 36

==================================================
STUDENT RESULTS REPORT
==================================================
Student Name: Pranitha
Number of Exams: 5

Exam Marks:
  Exam 1: 88.0
  Exam 2: 65.0
  Exam 3: 77.0
  Exam 4: 98.0
STUDENT RESULTS REPORT
==================================================
Student Name: Pranitha
Number of Exams: 5

Exam Marks:
  Exam 1: 88.0
  Exam 2: 65.0
  Exam 3: 77.0
  Exam 4: 98.0
  Exam 5: 36.0
```

```
  Exam 5: 36.0

Total Marks: 356.0
Average Marks: 71.2
Grade: C
Highest Score: 98.0
Lowest Score: 36.0
==================================================

PS C:\Users\Pranitha Rao\OneDrive\Desktop\AI ASSISTED CODING>
```

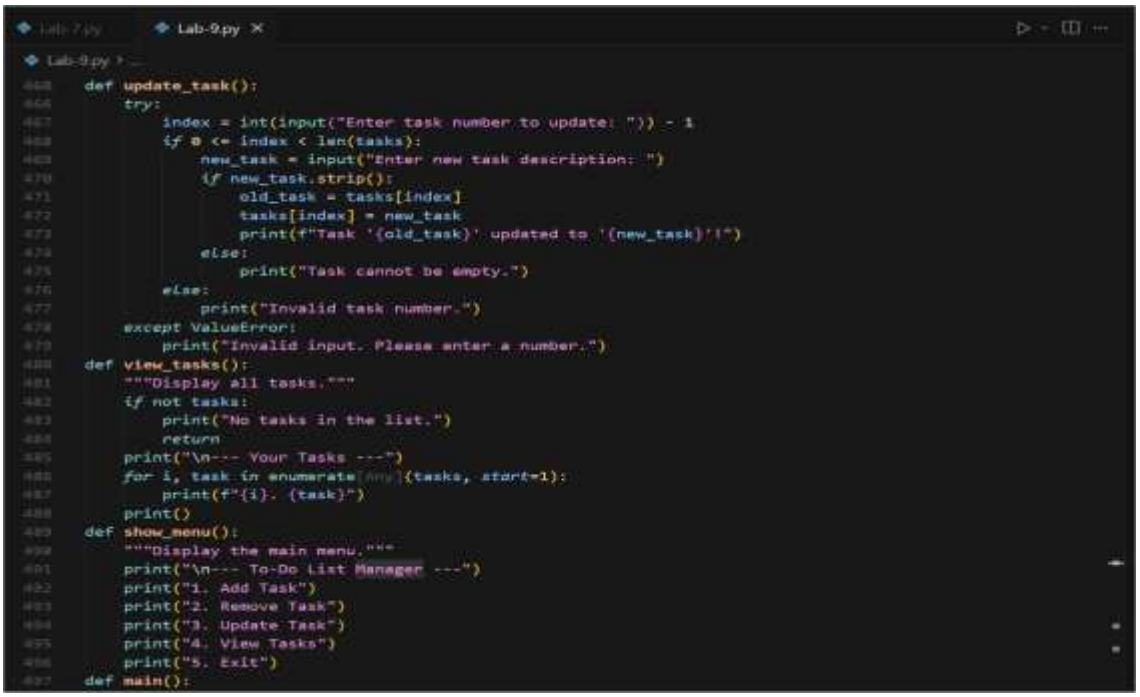
JUSTIFICATION: This code shows how long inline developer comments can be converted into clean, standardized Google-style docstrings to improve documentation quality. The student grade system accepts exam marks from the user and uses well-defined functions to calculate total marks, average, grade, highest score, and lowest score, then displays a final report. By moving explanations into structured docstrings, the function bodies become cleaner and easier to read, while the documentation clearly explains each function's purpose, inputs, and outputs. Overall, the code is more consistent, maintainable, and suitable for realworld or shared projects.

Task 5: Building a Mini Automatic Documentation Generator

PROMPT: Generate a python code in a .py file for a To-Do List Manager which includes multiple utility functions like add_task, remove_task, update_task. Take user inputs for the code. Generate a minimal code.

CODE:

```python
# To-Do List Manager
tasks = []
def add_task():
    """Add a new task to the list."""
    task = input("Enter task to add: ")
    if task.strip():
        tasks.append(task)
        print(f"Task '{task}' added successfully!")
    else:
        print("Task cannot be empty.")
def remove_task():
    """Remove a task from the list."""
    if not tasks:
        print("No tasks to remove.")
        return
    view_tasks()
    try:
        index = int(input("Enter task number to remove: ")) - 1
        if 0 <= index < len(tasks):
            removed = tasks.pop(index)
            print(f"Task '{removed}' removed successfully!")
        else:
            print("Invalid task number.")
    except ValueError:
        print("Invalid input. Please enter a number.")
def update_task():
    """Update an existing task."""
    if not tasks:
        print("No tasks to update.")
        return
    view_tasks()
```

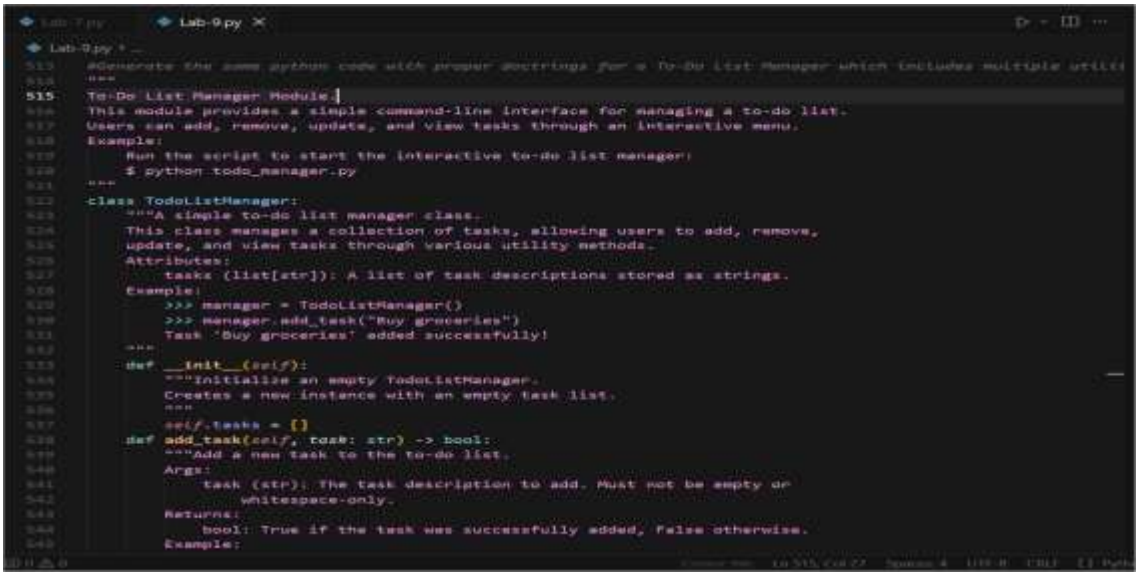


```
def update_task():
    try:
        index = int(input("Enter task number to update: ")) - 1
        if 0 <= index < len(tasks):
            new_task = input("Enter new task description: ")
            if new_task.strip():
                old_task = tasks[index]
                tasks[index] = new_task
                print(f"Task '{old_task}' updated to '{new_task}'!")
            else:
                print("Task cannot be empty.")
        else:
            print("Invalid task number.")
    except ValueError:
        print("Invalid input. Please enter a number.")
def view_tasks():
    """Display all tasks."""
    if not tasks:
        print("No tasks in the list.")
        return
    print("\n--- Your Tasks ---")
    for i, task in enumerate(tasks, start=1):
        print(f"{i}. {task}")
    print()
def show_menu():
    """Display the main menu."""
    print("\n--- To-Do List Manager ---")
    print("1. Add Task")
    print("2. Remove Task")
    print("3. Update Task")
    print("4. View Tasks")
    print("5. Exit")
def main():
```

```
def main():
    """Main program loop."""
    while True:
        show_menu()
        choice = input("Enter your choice (1-5): ")
        if choice == "1":
            add_task()
        elif choice == "2":
            remove_task()
        elif choice == "3":
            update_task()
        elif choice == "4":
            view_tasks()
        elif choice == "5":
            print("Thank you for using To-Do List Manager. Goodbye!")
            break
        else:
            print("Invalid choice. Please enter a number from 1 to 5.")
if __name__ == "__main__":
    main()
```

DOCSTRING PROMPT: Generate the same python code with proper doctrings for a To-Do List Manager which includes multiple utility functions like add_task, remove_task, update_task. Take user inputs for the code. The task is to detact functions and classes and insert placeholder google style docstrings fo reach detected function or class. The goal is documentation scaffolding. Generate a minimal and efficient code.

CODE:

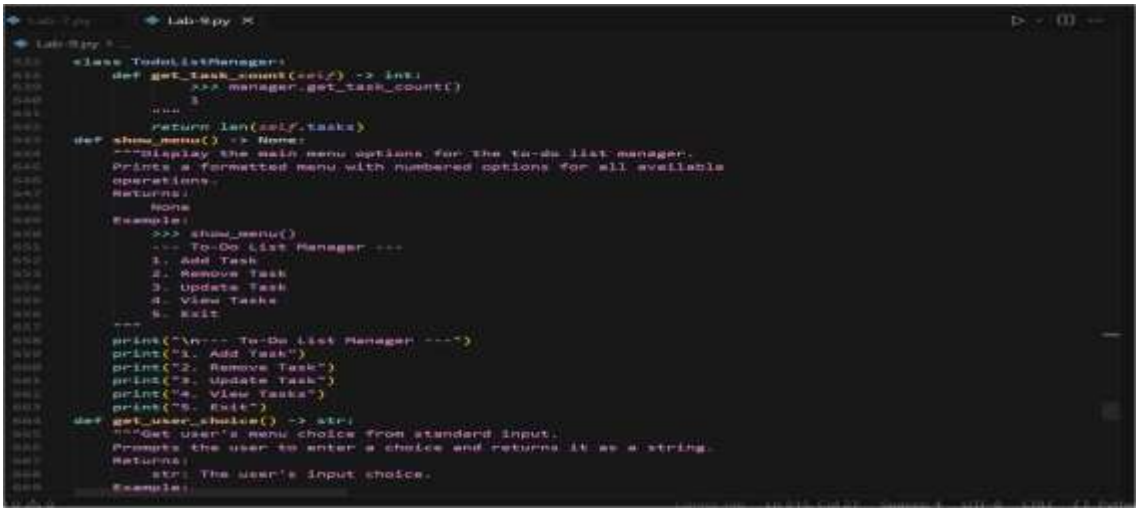


```
#Generate the same python code with proper docstrings for a To-Do List Manager which includes multiple utili
"""
To-Do List Manager Module
This module provides a simple command-line interface for managing a to-do list.
Users can add, remove, update, and view tasks through an interactive menu.
Example:
    Run the script to start the interactive to-do list manager:
    $ python todo_manager.py
"""
class TodoListManager:
    """A simple to-do list manager class.
    This class manages a collection of tasks, allowing users to add, remove,
    update, and view tasks through various utility methods.
    Attributes:
        tasks (list[str]): A list of task descriptions stored as strings.
    Example:
        >>> manager = TodoListManager()
        >>> manager.add_task("Buy groceries")
        Task 'Buy groceries' added successfully!
    """
    def __init__(self):
        """Initialize an empty TodoListManager.
        Creates a new instance with an empty task list.
        """
        self.tasks = []
    def add_task(self, task: str) -> bool:
        """Add a new task to the to-do list.
        Args:
            task (str): The task description to add. Must not be empty or
                whitespace-only.
        Returns:
            bool: True if the task was successfully added, False otherwise.
        Example:
```

```
class TodoListManager:
    def add_task(self, task: str) -> bool:
            >>> manager = TodoListManager()
            >>> manager.add_task("Complete assignment")
            True
        """
        if task.strip():
            self.tasks.append(task)
            print(f"Task '{task}' added successfully!")
            return True
        else:
            print("Task cannot be empty.")
            return False
    def remove_task(self, index: int) -> bool:
        """Remove a task from the to-do list by its index.
        Args:
            index (int): The 1-based index of the task to remove. The index
                should be between 1 and the number of tasks.
        Returns:
            bool: True if the task was successfully removed, False otherwise.
        Example:
            >>> manager = TodoListManager()
            >>> manager.add_task("Task 1")
            >>> manager.remove_task(1)
            True
        """
        if not self.tasks:
            print("No tasks to remove.")
            return False
        if 1 <= index <= len(self.tasks):
            removed = self.tasks.pop(index - 1)
            print(f"Task '{removed}' removed successfully!")
            return True
```

```
class TodoListManager:
    def remove_task(self, index: int) -> bool:
        else:
            print("Invalid task number.")
            return False
    def update_task(self, index: int, new_task: str) -> bool:
        """Update an existing task with a new description.
        Args:
            index (int): The 1-based index of the task to update. The index
                should be between 1 and the number of tasks.
            new_task (str): The new task description. Must not be empty or
                whitespace-only.
        Returns:
            bool: True if the task was successfully updated, False otherwise.
        Example:
            >>> manager = TodoListManager()
            >>> manager.add_task("Old task")
            >>> manager.update_task(1, "New task")
            True
        """
        if not self.tasks:
            print("No tasks to update.")
            return False
        if 1 <= index <= len(self.tasks):
            if new_task.strip():
                old_task = self.tasks[index - 1]
                self.tasks[index - 1] = new_task
                print(f"Task '{old_task}' updated to '{new_task}'!")
                return True
            else:
                print("Task cannot be empty.")
                return False
        else:
```

```
Lab-9.py
class TodoListManager:
    def update_task(self, index: int, new_task: str) -> bool:
            print("Invalid task number.")
            return False
    def view_tasks(self) -> None:
        """Display all tasks in the to-do list.
        Prints a numbered list of all tasks. If the list is empty, displays
        an appropriate message.
        Returns:
            None
        Example:
            >>> manager = TodoListManager()
            >>> manager.add_task("Task 1")
            >>> manager.add_task("Task 2")
            >>> manager.view_tasks()
            --- Your Tasks ---
            1. Task 1
            2. Task 2
        """
        if not self.tasks:
            print("No tasks in the list.")
            return
        print("\n--- Your Tasks ---")
        for i, task in enumerate(self.tasks, start=1):
            print(f"{i}. {task}")
        print()
    def get_task_count(self) -> int:
        """Get the total number of tasks in the list.
        Returns:
            int: The number of tasks currently in the list.
        Example:
            >>> manager = TodoListManager()
            >>> manager.add_task("Task 1")
```



```
Lab-9.py
class TodoListManager:
    def get_task_count(self) -> int:
            >>> manager.get_task_count()
            1
        """
        return len(self.tasks)
def show_menu() -> None:
    """Display the main menu options for the to-do list manager.
    Prints a formatted menu with numbered options for all available
    operations.
    Returns:
        None
    Example:
        >>> show_menu()
        --- To-Do List Manager ---
        1. Add Task
        2. Remove Task
        3. Update Task
        4. View Tasks
        5. Exit
    """
    print("\n--- To-Do List Manager ---")
    print("1. Add Task")
    print("2. Remove Task")
    print("3. Update Task")
    print("4. View Tasks")
    print("5. Exit")
def get_user_choice() -> str:
    """Get user's menu choice from standard input.
    Prompts the user to enter a choice and returns it as a string.
    Returns:
        str: The user's input choice.
    Example:
```

```
Lab-9.py
def get_user_choice() -> str:
        >>> # User enters "1"
        >>> choice = get_user_choice()
        Enter your choice (1-5): 1
        >>> choice
        '1'
    """
    return input("Enter your choice (1-5): ")
def get_task_input() -> str:
    """Get task description from user input.
    Prompts the user to enter a task description.
    Returns:
        str: The task description entered by the user.
    Example:
        >>> # User enters "Buy groceries"
        >>> task = get_task_input()
        Enter task to add: Buy groceries
        >>> task
        'Buy groceries'
    """
    return input("Enter task to add: ")
def get_task_index() -> int:
    """Get task index from user input.
    Prompts the user to enter a task number and converts it to an integer.
    Handles invalid input by returning -1.
    Returns:
        int: The task index (1-based) if valid input, -1 otherwise.
    Example:
        >>> # User enters "2"
        >>> index = get_task_index()
        Enter task number to remove: 2
        >>> index
        2
```

```python
def main() -> None:
    """
    $ python todo_manager.py
    """
    manager = TodoListManager()
    while True:
        show_menu()
        choice = get_user_choice()
        if choice == "1":
            task = get_task_input()
            manager.add_task(task)
        elif choice == "2":
            if manager.tasks:
                manager.view_tasks()
            index = get_task_index()
            if index != -1:
                manager.remove_task(index)
        elif choice == "3":
            if manager.tasks:
                manager.view_tasks()
            index = get_update_index()
            if index != -1:
                new_task = get_new_task_description()
                manager.update_task(index, new_task)
        elif choice == "4":
            manager.view_tasks()
        elif choice == "5":
            print("Thank you for using To-Do List Manager. Goodbye!")
            break
        else:
            print("Invalid choice. Please enter a number from 1 to 5.")
if __name__ == "__main__":
    main()
```

OUTPUT:

```
PS C:\Users\Pranitha Rao\OneDrive\Desktop\AI ASSISTED CODING> & "C:\Users\Pranitha Rao\AppData\Local\Microsoft\WindowsApps\python3.13.exe" "c:/Users/Pranitha Rao/OneDrive/Desktop/AI ASSISTED CODING/Lab-9.py"

--- To-Do List Manager ---
1. Add Task
2. Remove Task
3. Update Task
4. View Tasks
5. Exit
Enter your choice (1-5): 1
Enter task to add: Dance
Task 'Dance' added successfully!

--- To-Do List Manager ---
1. Add Task
2. Remove Task
3. Update Task
4. View Tasks
5. Exit
Enter your choice (1-5): 2

--- Your Tasks ---
1. Dance

Enter task number to remove: 3
Invalid task number.

--- To-Do List Manager ---
1. Add Task
2. Remove Task
3. Update Task
```

```
--- To-Do List Manager ---
1. Add Task
2. Remove Task
3. Update Task
4. View Tasks
5. Exit
Enter your choice (1-5): 4


--- Your Tasks ---
1. Dance


--- To-Do List Manager ---
1. Add Task
2. Remove Task
3. Update Task
4. View Tasks
5. Exit
Enter your choice (1-5): 5
Thank you for using To-Do List Manager. Goodbye!
PS C:\Users\Pranitha Rao\OneDrive\Desktop\AI ASSISTED CODING>
```

JUSTIFICATION: This example demonstrates how a simple to-do list program can be enhanced by automatically adding structured Google-style docstrings to functions and classes. The script manages tasks using a TodoListManager class and provides features like adding, removing, updating, and viewing tasks through user input. By replacing informal comments with clear docstrings, the code becomes easier to understand, more consistent, and suitable for shared or internal projects. Overall, it shows how AI-assisted documentation scaffolding can quickly improve code readability and maintainability without changing the program's core logic.