LAB ASSIGNMENT-9.4

Name: Sindhu Katakam

Enrollment number: 2303A52481

Batch-41

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

Prompt:

Create a student grade manager class that contains methods total_marks, average_marks, grade . give python code

Code:

```
class StudentGradeManager:
    def __init__(self, marks):
        self.marks = marks

    def total_marks(self):
        return sum(self.marks)

    def average_marks(self):
        return self.total_marks() / len(self.marks)

    def grade(self):
        avg = self.average_marks()
        if avg >= 90:
            return 'A'
        elif avg >= 80:
            return 'B'
        elif avg >= 70:
            return 'C'
        elif avg >= 60:
            return 'D'
        else:
            return 'F'
```

DocString Prompt:

Generate a student grade manager class that contains methods total_marks, average_marks, grade 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:

```
class StudentGradeManager:
    def __init__(self, marks):
        self.marks = marks
    def total_marks(self):
        """
        Calculate the total marks obtained by the student.
        Returns:
            int: The total marks obtained by the student.

        Example:
            >>> manager = StudentGradeManager([85, 90, 78])
            >>> manager.total_marks()
            253
        """
        return sum(self.marks)
    def average_marks(self):
        """
        Calculate the average marks obtained by the student.
        Returns:
            float: The average marks obtained by the student.
        Example:
            >>> manager = StudentGradeManager([85, 90, 78])
            >>> manager.average_marks()
            84.33
        """
```

```
        return self.total_marks() / len(self.marks)

    def grade(self):
        """
        Determine the grade of the student based on the average marks.

        Returns:
            str: The grade of the student (A, B, C, D, F).

        Example:
            >>> manager = StudentGradeManager([85, 90, 78])
            >>> manager.grade()
            'A'
        """
        avg = self.average_marks()
        if avg >= 90:
            return 'A'
        elif avg >= 80:
            return 'B'
        elif avg >= 70:
            return 'C'
        elif avg >= 60:
            return 'D'
# Example usage
if __name__ == "__main__":
    n1=int(input("Enter marks for subject 1: "))
    n2=int(input("Enter marks for subject 2: "))
    n3=int(input("Enter marks for subject 3: "))
    manager = StudentGradeManager([n1, n2, n3])
    print("Total Marks:", manager.total_marks())
    print("Average Marks:", manager.average_marks())
    print("Grade:", manager.grade())
```

Output:

```
PS D:\AI_ASSIT_CODING> & C:/Users/MYSELF,
Enter marks for subject 1: 98
Enter marks for subject 2: 90
Enter marks for subject 3: 78
Total Marks: 266
Average Marks: 88.66666666666667
Grade: B
```

Explanation:

This program uses a **StudentGradeManager** class to store a student's marks, calculate the total and average, and assign a grade based on the average. Each function has **docstrings**, which briefly explain what the function does, what it returns, and how to use it. The user enters marks, and the program displays the total, average, and grade

Task 2: Enhancing Readability Through AI-Generated Inline Comments

Prompt:
give insertion sort code.

Code:

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr
# Example usage
if __name__ == "__main__":
    arr =list(map(int, input("Enter numbers to sort (separated by space): ").split()))
    sorted_arr = insertion_sort(arr)
    print("Sorted array:", sorted_arr)
```

DocString Prompt:

give insertion sort code with concise and meaningful inline comments.comments that explain why the logic exists,not what python syntax does. and code should be readable

Code:

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        # Move elements of arr[0..i-1], that are greater than key, to one position ahead of their current position
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1# Decrement j to continue checking the previous elements
        # Place the key in its correct position in the sorted subarray
        arr[j + 1] = key
    return arr
if __name__ == "__main__":
    arr =list(map(int, input("Enter numbers to sort (separated by space): ").split()))
    sorted_arr = insertion_sort(arr)
    print("Sorted array:", sorted_arr)
```

Output:

```
Enter numbers to sort (separated by space): 2 45 78 24 35
Sorted array: [2, 24, 35, 45, 78]
```

Explanation:

This program sorts numbers using the **insertion sort** method. The function **insertion_sort(arr)** checks each element and places it in the correct position so the list becomes sorted step by step. The code also uses **inline comments** to briefly explain important steps, such as shifting elements and updating the index, which makes the code easier to understand. In the main part, the user enters numbers separated by spaces, and the program sorts and prints the sorted array.

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

Code:

```python
class ToDoList:
    def __init__(self):
        self.tasks = []

    def add_task(self, task):
        self.tasks.append(task)

    def remove_task(self, task):
        if task in self.tasks:
            self.tasks.remove(task)

    def view_tasks(self):
        return self.tasks
if __name__ == "__main__":
    if __name__ == "__main__":
        todo = ToDoList()
        todo.add_task("Buy groceries")
        todo.add_task("Finish homework")
        print(todo.view_tasks())
        todo.remove_task("Buy groceries")
        print(todo.view_tasks())
```

DocString Prompt:

generate a python module for to do list .generate a module level docstring at the top of the file include the purpose of the module ,required libraries and dependies and brief description of functions ans classes and short example of how module can be focus on clarity. the output has to include well written multiline module level docstring and clear overview of what module does ans how to use it . document suitable for real word project or repositories.

Code:

```
class ToDoList:
    """
    ToDoList module provides a simple implementation of a to-do list manager. It allows users to add, remove,
    and view tasks in their to-do list.

    Required Libraries:
        None
    Dependencies:
        None
    Functions:
        add_task(task): Adds a new task to the to-do list.
        remove_task(task): Removes a task from the to-do list.
        view_tasks(): Returns a list of all tasks in the to-do list.
    Example Usage:
        >>> todo = ToDoList()
        >>> todo.add_task("Buy groceries")
        >>> todo.add_task("Finish homework")
        >>> todo.view_tasks()
        ['Buy groceries', 'Finish homework']
        >>> todo.remove_task("Buy groceries")
        >>> todo.view_tasks()
        ['Finish homework']
    """
    def __init__(self):
        self.tasks = []
```

```
    def add_task(self, task):
        """Adds a new task to the to-do list."""
        self.tasks.append(task)

    def remove_task(self, task):
        """Removes a task from the to-do list."""
        if task in self.tasks:
            self.tasks.remove(task)

    def view_tasks(self):
        """Returns a list of all tasks in the to-do list."""
        return self.tasks
# Example usage
if __name__ == "__main__":
    todo = ToDoList()
    todo.add_task("Buy groceries")
    todo.add_task("Finish homework")
    print(todo.view_tasks())
    todo.remove_task("Buy groceries")
    print(todo.view_tasks())
```

Output:

```
['Buy groceries', 'Finish homework']
['Finish homework']
['Buy groceries', 'Finish homework']
['Finish homework']
```

Explanation:

This program defines a **ToDoList** class to manage daily tasks. It allows the user to **add tasks**, **remove tasks**, and **view all tasks** stored in a list. The code includes **docstrings**, which are short descriptions written inside the class and methods to explain what the program and each function do, how they work, and how to use them. These docstrings make the code easier to understand and maintain. In the example usage, tasks are added, displayed, one task is removed, and the updated list is printed.

Task 4: Converting Developer Comments into Structured Docstrings

Prompt:

generate a python code for personal study time monitoring application that tracks study sessions ,breaks and total study time.

Code:

```
import time
class StudyTimeMonitor:
    def __init__(self):
        self.total_study_time = 0
        self.session_start_time = None

    def start_study_session(self):
        self.session_start_time = time.time()

    def end_study_session(self):
        if self.session_start_time is None:
            raise ValueError("No active study session to end.")
        session_end_time = time.time()
        self.total_study_time += session_end_time - self.session_start_time
        self.session_start_time = None

    def take_break(self, duration):
        time.sleep(duration)

    def get_total_study_time(self):
        return self.total_study_time
```

DocString Prompt:

generate a python code for personal study time monitoring application that tracks study sessions,breaks and total

study time.converts the comments into structured google styple or numpy style docstring.remove redundant inline
comments after conversion.output should consists of standardized docstrings.reduced clutter inside function bodies

```python
import time
class StudyTimeMonitor:
    """
    A class to monitor personal study time, including study sessions, breaks, and total study time.

    Methods:
        start_study_session(): Starts a new study session.
        end_study_session(): Ends the current study session and updates total study time.
        take_break(duration): Takes a break for the specified duration in seconds.
        get_total_study_time(): Returns the total study time in seconds.
    """

    def __init__(self):
        self.total_study_time = 0
        self.session_start_time = None

    def start_study_session(self):
        """
        Starts a new study session by recording the current time.
        """
        self.session_start_time = time.time()

    def end_study_session(self):
        """
        Ends the current study session and updates the total study time.
        Raises:
            ValueError: If no study session is currently active.
        """
        if self.session_start_time is None:
            raise ValueError("No active study session to end.")
        session_end_time = time.time()
        self.total_study_time += session_end_time - self.session_start_time
        self.session_start_time = None

    def take_break(self, duration):
        """
        Takes a break for the specified duration in seconds.

        Parameters:
            duration (int): The duration of the break in seconds.
        """
        time.sleep(duration)

    def get_total_study_time(self):
        """
        Returns the total study time in seconds.

        Returns:
            float: The total study time in seconds.
        """
```

```
        return self.total_study_time
# Example usage
if __name__ == "__main__":
    monitor = StudyTimeMonitor()
    monitor.start_study_session()
    time.sleep(2)  # Simulate a study session of 2 seconds
    monitor.end_study_session()
    print("Total Study Time:", monitor.get_total_study_time(), "seconds")
    monitor.take_break(1)  # Take a break of 1 second
    monitor.start_study_session()
    time.sleep(3)  # Simulate another study session of 3 seconds
    monitor.end_study_session()
    print("Total Study Time:", monitor.get_total_study_time(), "seconds")
```

Output:

```
Total Study Time: 2.0006906986236572 seconds
Total Study Time: 5.001116514205933 seconds
```

Explanation:

This program defines a **StudyTimeMonitor** class to track how long a person studies. It allows starting a study session, ending it to calculate the time spent, taking a break, and checking the total study time. The code includes **docstrings**, which are short explanations written inside the class and methods. These docstrings describe what each function does, the parameters it uses, the values it returns, and possible errors. This helps users and programmers understand how to use the program easily. In the example, study sessions and breaks are simulated, and the total study time is displayed.

Task 5: Building a Mini Automatic Documentation Generator

Prompt:

Generate a python code for Banking system which inlcudes multiple functions like creating account, deposit, withdraw, check balance, transaction history. Take user inputs for the system. generate a minimal code.

Code:

```python
class BankAccount:
    def __init__(self, account_holder):
        self.account_holder = account_holder
        self.balance = 0
        self.transaction_history = []

    def deposit(self, amount):
        self.balance += amount
        self.transaction_history.append(f"Deposited: {amount}")

    def withdraw(self, amount):
        if amount > self.balance:
            print("Insufficient funds")
        else:
            self.balance -= amount
            self.transaction_history.append(f"Withdrew: {amount}")

    def check_balance(self):
        return self.balance

    def get_transaction_history(self):
        return self.transaction_history
```

Docstring Prompt:

Generate the same python code with proper doctrings for a banking system which includes multiple utility functions for creating account, deposit, withdraw, check balance,
transaction history. 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:

```python
class BankAccount:
    """
    A class to represent a bank account with functionalities to deposit, withdraw, check balance, and view transaction
    """
    def __init__(self, account_holder):
        """
        Initializes a new bank account for the account holder.

        Parameters:
            account_holder (str): The name of the account holder.
        """
        self.account_holder = account_holder
        self.balance = 0
        self.transaction_history = []
    def deposit(self, amount):
        """
        Deposits a specified amount into the bank account.
        Parameters:
            amount (float): The amount to be deposited.
        """
        self.balance += amount
        self.transaction_history.append(f"Deposited: {amount}")
    def withdraw(self, amount):
        """
        Withdraws a specified amount from the bank account if sufficient funds are available.
        Parameters:
            amount (float): The amount to be withdrawn.
        """
        if amount > self.balance:
            print("Insufficient funds")
        else:
            self.balance -= amount
            self.transaction_history.append(f"Withdrew: {amount}")
    def check_balance(self):
        """
        Checks the current balance of the bank account.

        Returns:
            float: The current balance of the account.
        """
        return self.balance
    def get_transaction_history(self):
        """
        Retrieves the transaction history of the bank account.
        Returns:
            list: A list of transaction history entries.
        """
        return self.transaction_history
if __name__ == "__main__":
```

Output:

```
PS D:\AI_ASSIT_CODING> & C:/Users/MYSELF/A
Enter account holder name: sindhu Katakam

1. Deposit
2. Withdraw
3. Check Balance
4. Transaction History
5. Exit
Choose an option: 1
Enter amount to deposit: 2400
```

Explanation:

This program uses a **BankAccount** class to handle deposit, withdrawal, balance checking, and transaction history in an organized way. Using a class keeps related data and functions together. The **docstrings** briefly explain what the class and each method do, making the code easy to understand and use.