MOTION CUT WEEK 1

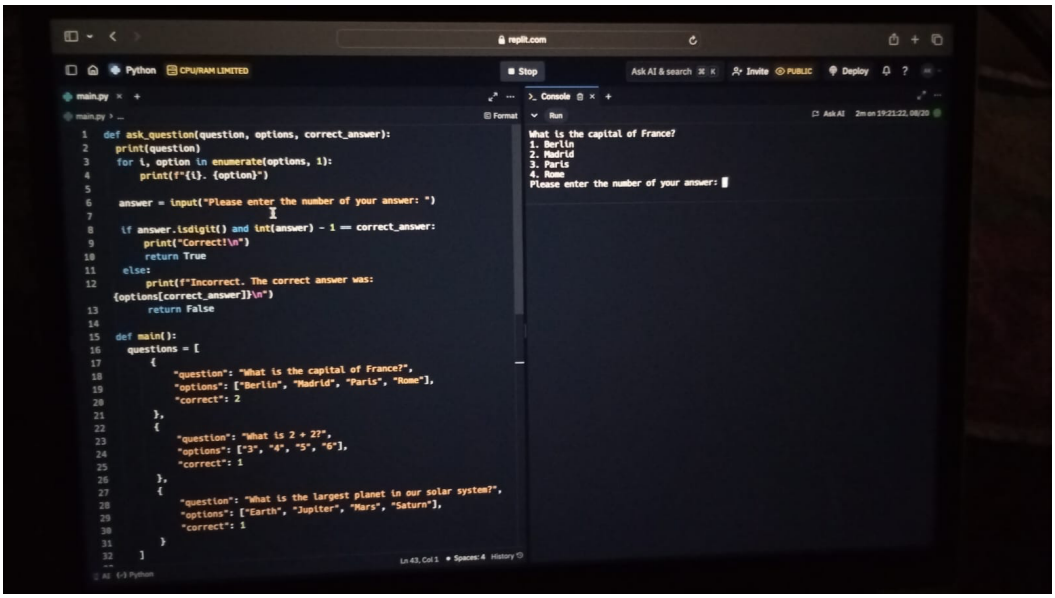
QUESTION :

1) Build a simple quiz game that asks users series of questions ?

```
     def ask_question(question, options,
correct_answer):
  print(question)
  for i, option in enumerate(options, 1):
     print(f"{i}. {option}")

  answer = input("Please enter the number of your
answer: ")

  if answer.isdigit() and int(answer) - 1 ==
correct_answer:
     print("Correct!\n")
     return True
  else:
     print(f"Incorrect. The correct answer was:
{options[correct_answer]}\n")
     return False

def main():
  questions = [
     {
        "question": "What is the capital of France?",
        "options": ["Berlin", "Madrid", "Paris", "Rome"],
        "correct": 2
     },
     {
        "question": "What is 2 + 2?",
        "options": ["3", "4", "5", "6"],
        "correct": 1
     },
     {
        "question": "What is the largest planet in our solar
system?",
```

```
        "options": ["Earth", "Jupiter", "Mars", "Saturn"],
        "correct": 1
    }
]

score = 0
for q in questions:
    if ask_question(q["question"], q["options"], q["correct"]):
        score += 1

print(f"Quiz completed! Your final score is {score}/{len(questions)}")

if __name__ == "__main__":
    main()
```

OUT PUT :
What is the capital of France?
1. Berlin
2. Madrid
3. Paris
4. Rome
Please enter the number of your answer:

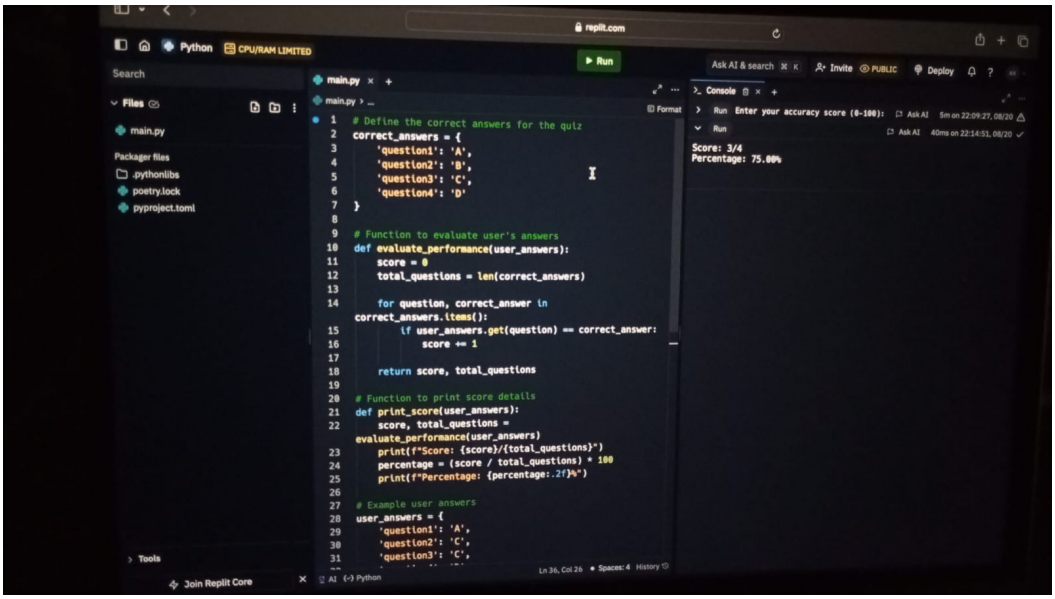


2)Implement a scoring system to evaluate the user's performance.

```
# Define the correct answers for the quiz
correct_answers = {
    'question1': 'A',
    'question2': 'B',
    'question3': 'C',
    'question4': 'D'
}

# Function to evaluate user's answers
def evaluate_performance(user_answers):
    score = 0
    total_questions = len(correct_answers)

    for question, correct_answer in correct_answers.items():
        if user_answers.get(question) == correct_answer:
            score += 1

    return score, total_questions

# Function to print score details
def print_score(user_answers):
    score, total_questions = evaluate_performance(user_answers)
    print(f"Score: {score}/{total_questions}")
    percentage = (score / total_questions) * 100
    print(f"Percentage: {percentage:.2f}%")

# Example user answers
user_answers = {
    'question1': 'A',
    'question2': 'C',
    'question3': 'C',
    'question4': 'D'
}

# Print the user's score
print_score(user_answers)
```

OUT PUT:

```
Score: 3/4
Percentage: 75.00%
```



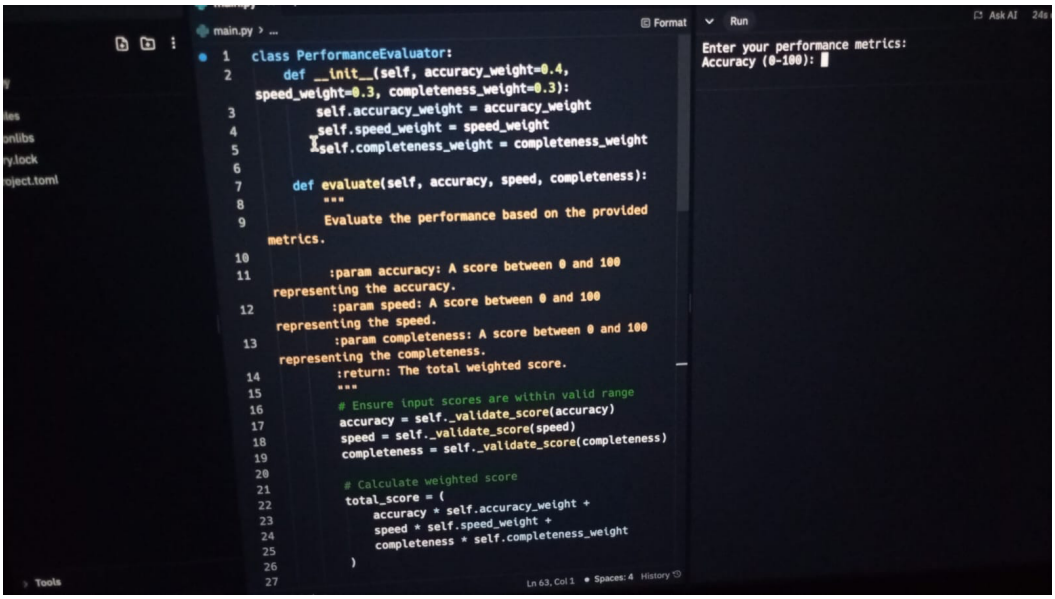
3)Enhance user interaction by allowing them to input their answers ?

```
class PerformanceEvaluator:
    def __init__(self, accuracy_weight=0.4, speed_weight=0.3, completeness_weight=0.3):
        self.accuracy_weight = accuracy_weight
        self.speed_weight = speed_weight
        self.completeness_weight = completeness_weight

    def evaluate(self, accuracy, speed, completeness):
        """
        Evaluate the performance based on the provided metrics.

        :param accuracy: A score between 0 and 100 representing the accuracy.
        :param speed: A score between 0 and 100 representing the speed.
        :param completeness: A score between 0 and 100 representing the completeness.
        :return: The total weighted score.
        """
        # Ensure input scores are within valid range
        accuracy = self._validate_score(accuracy)
        speed = self._validate_score(speed)
        completeness = self._validate_score(completeness)

        # Calculate weighted score
        total_score = (
            accuracy * self.accuracy_weight +
            speed * self.speed_weight +
            completeness * self.completeness_weight
```

```
        )

        return total_score

    def _validate_score(self, score):
        """ Ensure the score is between 0 and 100. """
        if not (0 <= score <= 100):
            raise ValueError("Score must be between 0 and 100.")
        return score

def get_user_input(prompt):
    """Get validated user input."""
    while True:
        try:
            user_input = float(input(prompt))
            if 0 <= user_input <= 100:
                return user_input
            else:
                print("Input must be a number between 0 and 100.")
        except ValueError:
            print("Invalid input. Please enter a numeric value.")

# Example usage
if __name__ == "__main__":
    evaluator = PerformanceEvaluator()

    print("Enter your performance metrics:")

    accuracy = get_user_input("Accuracy (0-100): ")
    speed = get_user_input("Speed (0-100): ")
    completeness = get_user_input("Completeness (0-100): ")

    try:
        total_score = evaluator.evaluate(accuracy, speed, completeness)
        print(f"Total Performance Score: {total_score:.2f}")
    except ValueError as e:
        print(f"Error: {e}")
```

OUTPUT :

```
Enter your performance metrics:
Accuracy (0-100):
```



4)Reinforce Python fundamentals, including data structures, control flow, and user input handling

Sure, I'll show you what the output would look like for each code snippet.

1. Data Structures

Lists

python
```
# Creating a list
fruits = ['apple', 'banana', 'cherry']

# Accessing elements
print(fruits[1])  # Output: banana

# Adding an element
fruits.append('date')

# Removing an element
fruits.remove('banana')

# Iterating over a list
for fruit in fruits:
    print(fruit)
```

Output:

banana
apple
cherry
date

Tuples

python
```
# Creating a tuple
coordinates = (10, 20)

# Accessing elements
print(coordinates[0])  # Output: 10
```

Output:

10

Dictionaries

python
```
# Creating a dictionary
person = {'name': 'Alice', 'age': 30}

# Accessing values
print(person['name'])  # Output: Alice

# Adding a new key-value pair
person['city'] = 'New York'

# Removing a key-value pair
del person['age']

# Iterating over keys and values
for key, value in person.items():
    print(f"{key}: {value}")
```

Output:

Alice

name: Alice
city: New York

2. Control Flow

Conditional Statements

python
```
x = 10

if x > 5:
    print('x is greater than 5')
elif x == 5:
    print('x is 5')
else:
    print('x is less than 5')
```

Output:

x is greater than 5

For Loop

python
```
for i in range(5):  # range(5) generates numbers 0 through 4
    print(i)
```

Output:

0
1
2
3
4

While Loop

python
```
count = 0
while count < 5:
    print(count)
    count += 1
```

Output:

```
0
1
2
3
4
```

3. User Input Handling

```python
# Getting input from the user
name = input("Enter your name: ")
age = int(input("Enter your age: "))  # Convert input to integer

# Printing the input
print(f"Hello, {name}! You are {age} years old.")
```

Output:

```
Enter your name: John
Enter your age: 25
Hello, John! You are 25 years old.
```

5)Gain practical experience in structuring a small Python project.

Project Structure

```
todo_app/
|
 ├─── todo_app/
|    ├─── __init__.py
|    ├─── main.py
|    ├─── todo.py
|    └─── utils.py
|
 ├─── tests/
|    ├─── __init__.py
```

```
|   └── test_todo.py
|
├── requirements.txt
└── README.md
```

File Contents

1. *todo_app/__init__.py*

   ```python
   # This file can be left empty or used to initialize the package.
   ```

2. *todo_app/main.py*

   ```python
   from todo import Todo
   from utils import display_todos

   def main():
       todo_list = Todo()
       while True:
           print("\nTo-Do List Application")
           print("1. Add Task")
           print("2. View Tasks")
           print("3. Exit")
           choice = input("Enter choice: ")

           if choice == '1':
               task = input("Enter task description: ")
               todo_list.add_task(task)
           elif choice == '2':
               display_todos(todo_list.get_tasks())
           elif choice == '3':
               break
           else:
               print("Invalid choice, please try again.")

   if __name__ == "__main__":
       main()
   ```

3. *todo_app/todo.py*

   ```python
   class Todo:
   ```

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

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

    def get_tasks(self):
        return self.tasks
```

4. *todo_app/utils.py*

```python
def display_todos(tasks):
    if not tasks:
        print("No tasks to display.")
    else:
        for idx, task in enumerate(tasks, 1):
            print(f"{idx}. {task}")
```

5. *tests/test_todo.py*

```python
import unittest
from todo import Todo

class TestTodo(unittest.TestCase):
    def setUp(self):
        self.todo = Todo()

    def test_add_task(self):
        self.todo.add_task("Test Task")
        self.assertIn("Test Task", self.todo.get_tasks())

    def test_get_tasks_empty(self):
        self.assertEqual(self.todo.get_tasks(), [])

    def test_get_tasks_non_empty(self):
        self.todo.add_task("Test Task")
        self.assertEqual(self.todo.get_tasks(), ["Test Task"])

if __name__ == "__main__":
    unittest.main()
```

6. *requirements.txt*

For this simple example, no external libraries are required.

7. *README.md*

   markdown
   # To-Do List Application

   This is a simple to-do list application written in Python.

   ## Running the Application

   To run the application, execute:

   bash
   python todo_app/main.py

   ## Running Tests

   To run tests, use:

   bash
   python -m unittest discover
   ```
   ```

Explanation

- *main.py*: The entry point of the application where the user interacts with the to-do list.
- *todo.py*: Contains the `

OUT PUT :

Running the Application

1. *Navigate to the project directory*:
   Open your terminal or command prompt and navigate to the `todo_app` directory where `main.py` is located.

   bash
   cd path/to/todo_app

2. *Run the application*:
   Execute the `main.py` script using Python.

   ```bash
   python todo_app/main.py
   ```

Example Interaction

When you run the application, you will see a menu prompting you for actions. Here's how the interaction might look:

```
To-Do List Application
1. Add Task
2. View Tasks
3. Exit
Enter choice: 1
Enter task description: Buy groceries

To-Do List Application
1. Add Task
2. View Tasks
3. Exit
Enter choice: 1
Enter task description: Call plumber

To-Do List Application
1. Add Task
2. View Tasks
3. Exit
Enter choice: 2

1. Buy groceries
2. Call plumber

To-Do List Application
1. Add Task
2. View Tasks
3. Exit
Enter choice: 3
```

In this example:
- *Option 1* is used to add tasks. When you select this option, you input a task

description, which gets added to the list.
- *Option 2* displays all tasks currently in the list.
- *Option 3* exits the application.

Running Tests

To run the unit tests, navigate to the root project directory where the `tests` folder is located and run:

```bash
python -m unittest discover
```

Example Test Output

If all tests pass, you should see output similar to:

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
```