**SECURIN – ASSESSMENT**

* **Submitted by**

**Divya Shakthi R**

**LOGIC:**

[PART – A]

1. Calculate the total number of combinations possible:

- Since each die has 6 sides, there are 6 possibilities for each die, resulting in a total of 6 x 6 = **36 combinations**.

2. Calculate and display the distribution of all possible combinations:

- Generate all possible combinations of rolling Die A and Die B together, resulting in a 6x6 matrix **combination matrix**.

- Each cell in the **sum matrix** represents the sum of the values rolled on Die A and Die B.

3. Calculate the probability of each possible sum occurring:

- Iterate through all possible sums from 2 to 12.

- For each sum, count the number of combinations that result in that sum.

- Calculate the probability of each sum occurring by **dividing the count by the total number of combinations**.

[PART – B]

1. Develop the `undoom\_dice` function:

- Calculate the scaling factor by dividing the sum of spots on Die A by the sum of spots on Die B.

- Redistribute spots on Die A while ensuring that no face has more than 4 spots.

- Scale spots on Die B by multiplying each spot by the scaling factor.

- Round the scaled spots on Die B to the nearest integer.

- Return the new configurations of Die A and Die B.

- I’ve also ensured that all the given constraints have been adhered to.

**SOURCE CODE AND OUTPUT SCREENSHOT:**

Link to GitHub repository:

<https://github.com/divyashakthi/The_Doomed_Dice_Challenge>

**EXPLANATION OF SOLUTION APPROACH:**

[PART – A]

**1. Total Number of Combinations**:

- Recognized that the total number of combinations can be calculated by multiplying the number of sides on each die (6 sides for both Die A and Die B).

- Implemented the calculation in Python using a simple multiplication operation.

**2. Distribution of Combinations**:

- Realized that representing all possible combinations as a 6x6 matrix would provide a clear visual representation of the distribution.

- Utilized nested loops in Python to iterate through each possible combination and populate the matrix with the corresponding sums.

- The code snippet `all\_sums[i - 1][j - 1] = i + j` is responsible for calculating the sum of the values rolled on Die A and Die B and storing it in the appropriate cell of the matrix.

**3. Probability of Each Sum**:

- Developed a method to calculate the probability of each possible sum occurring when rolling both dice.

- Iterated through all possible sums from 2 to 12 using a loop.

- For each sum, counted the number of combinations resulting in that sum by iterating through the combinations and checking if the sum matched the current value.

- Calculated the probability by dividing the count by the total number of combinations.

[PART – B]

**1. `undoom\_dice` Function**:

- Recognized the need to modify the configurations of Die A and Die B while maintaining the same probabilities for each sum.

- Calculated the scaling factor by dividing the sum of spots on Die A by the sum of spots on Die B, ensuring accurate redistribution of spots.

- Implemented logic to redistribute spots on Die A by capping each face at a maximum of 4 spots, adhering to Loki's condition.

- Utilized scaling factors and rounding techniques to adjust the spots on Die B while preserving the overall distribution and probabilities.

- The code snippet `b = [round(spots \* scaling\_factor) for spots in die\_b]` demonstrates the scaling of spots on Die B using a list comprehension.

By adopting this approach, the solutions were implemented effectively in Python, ensuring accurate calculations and compliance with the specified constraints.