THE DOOMED DICE CHALLENGE

PART-A

PROBLEM STATEMENT:

Consider two six-sided faces namely Die a and Die b with faces numbered from 1 to 6. Both the dice are rolled together

1. **HOW MANY TOTAL COMBINATIONS ARE POSSIBLE? SHOW THE MATH ALONG WITH THE CODE?**

LOGIC:

* In the case of two six-sided dice:
* Each die has 6 faces, numbered from 1 to 6.
* When rolling a single die, there are 6 possible outcomes (1, 2, 3, 4, 5, 6).
* To calculate the total combinations possible when rolling two dice, you multiply the number of outcomes of the first die by the number of outcomes of the second die. This is because each outcome of the first die can be paired with each outcome of the second die to create a unique combination.
* Mathematically, it's expressed as:
* Total Combinations = Number of faces on Die A \* Number of faces on Die B
* For six-sided dice, each die has 6 faces, so:
* Number of faces on Die A = 6
* Number of faces on Die B = 6
* Therefore,
* Total Combinations = 6 \* 6 = 36
* Hence, when two six-sided dice are rolled together, there are a total of 36 possible combinations of outcomes that can occur.

**CODE:**

def calculate\_total\_combinations():

number\_of\_faces\_die\_a = 6

number\_of\_faces\_die\_b = 6

total\_combinations = number\_of\_faces\_die\_a \* number\_of\_faces\_die\_b

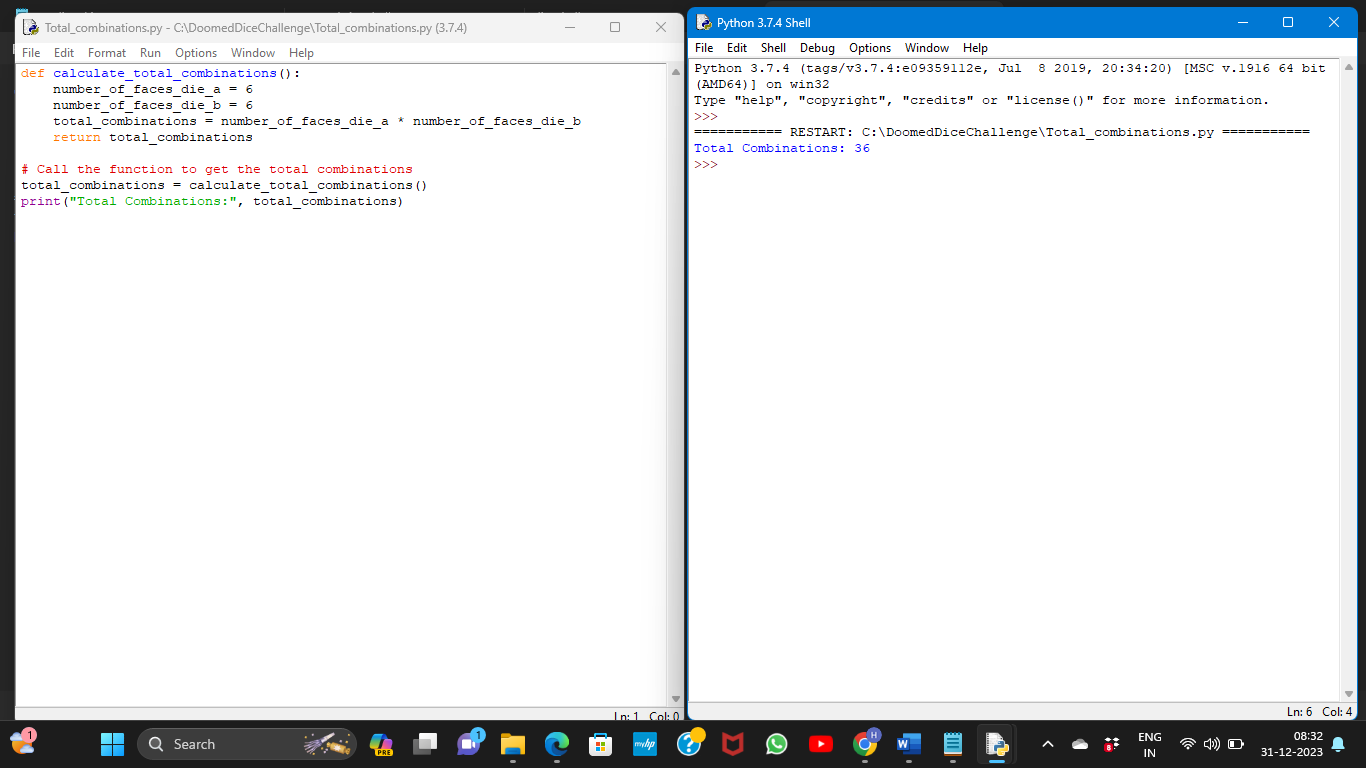
return total\_combinations

# Call the function to get the total combinations

total\_combinations = calculate\_total\_combinations()

print ("Total Combinations:", total\_combinations)

**SCREENSHOT:**

****

**2) CALCULATE AND DISPLAY THE DISTRIBUTION OF ALL POSSIBLE COMBINATIONS THAT CAN BE OBTAINED WHEN ROLLING BOTH DIE A AND DIE B TOGETHER**

**LOGIC:**

* The combination distribution logic generates all possible outcomes when rolling two six-sided dice.
* It uses nested loops or comprehensions to create a structure (like a list of dictionaries or a list of lists) representing each combination.
* For each value of Die A (1 to 6), it pairs it with all values of Die B (also 1 to 6), showing the values of both dice and their sum for every combination.
* This results in a comprehensive display of all potential outcomes when rolling two dice together, listing their individual values and their sums.

**CODE:**

def calculate\_combination\_distribution():

distribution\_list = [

[{'Die A': i, 'Die B': j, 'Sum': i + j} for j in range (1, 7)] for i in range (1, 7)]

return distribution\_list

def main():

# Calculate and display combination distribution

combination\_distribution = calculate\_combination\_distribution()

print("Combination Distribution:")

for row in combination\_distribution:

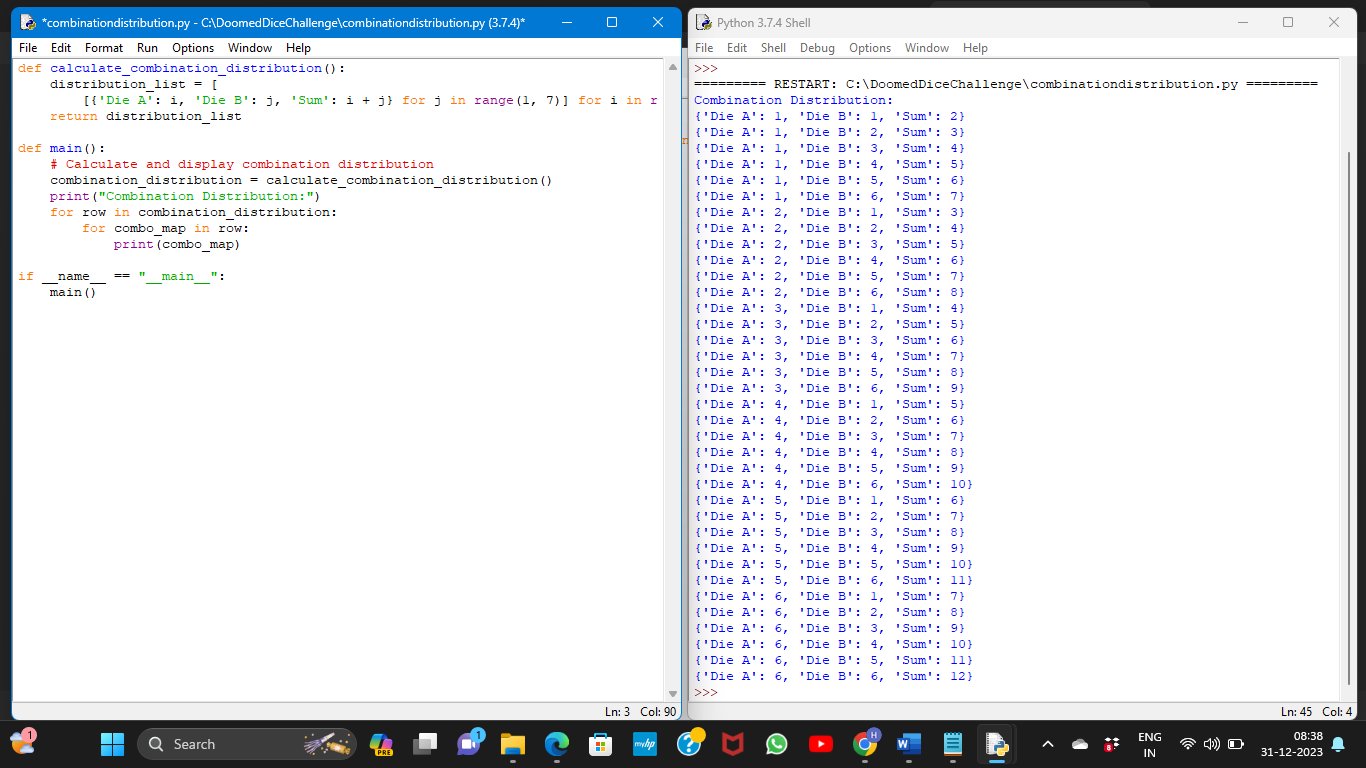
for combo\_map in row:

print(combo\_map)

if \_\_name\_\_ == "\_\_main\_\_":

main()

**SCREENSHOT:**

****

**3)CALCULATE THE PROBABILITY OF ALL POSSIBLE SUMS OCCURRING AMONG THE NUMBER OF COMBINATIONS FROM (2).EXAMPLE: P (SUM = 2) = 1/X AS THERE IS ONLY ONE COMBINATION POSSIBLE TO OBTAIN SUM = 2. DIE A = DIE B = 1.**

**LOGIC:**

**Total Combinations**: Determine the total number of combinations possible when rolling two six-sided dice (total\_ combinations = 36, as each die has 6 sides).

**Counting Valid Combinations for Each Sum**:

Iterate through all possible sums from 2 to 12 (as the minimum sum is 2 and the maximum is 12 when rolling two six-sided dice).

For each sum:

Loop through the potential values of the first die (1 to 6).

Each value of the first die, determine the valid values for the second die that would result in the desired sum.

Count how many valid combinations yield the current sum.

**Calculate Probabilities**: Calculate the probability for each sum by dividing the count of valid combinations for that sum by the total number of combinations possible (36).

**Store Probabilities**: Store the calculated probabilities in a data structure (such as a dictionary) with the sums as keys and their corresponding probabilities as values. This logic computes probabilities by counting valid combinations for each sum and then dividing those counts by the total number of combinations possible when rolling two six-sided dice.

**CODE:**

def calculate\_sum\_probability():

sides = 6

total\_combinations = sides \*\* 2

probability\_map = {}

for i in range(2, 13):

count = 0

for j in range(1, sides + 1):

if i - j <= sides and i - j >= 1:

count += 1

probability\_map[i] = f"{count}/{total\_combinations}"

return probability\_map

def main():

# Calculate and display sum probabilities

sum\_probabilities = calculate\_sum\_probability()

print("Probability of Sums:")

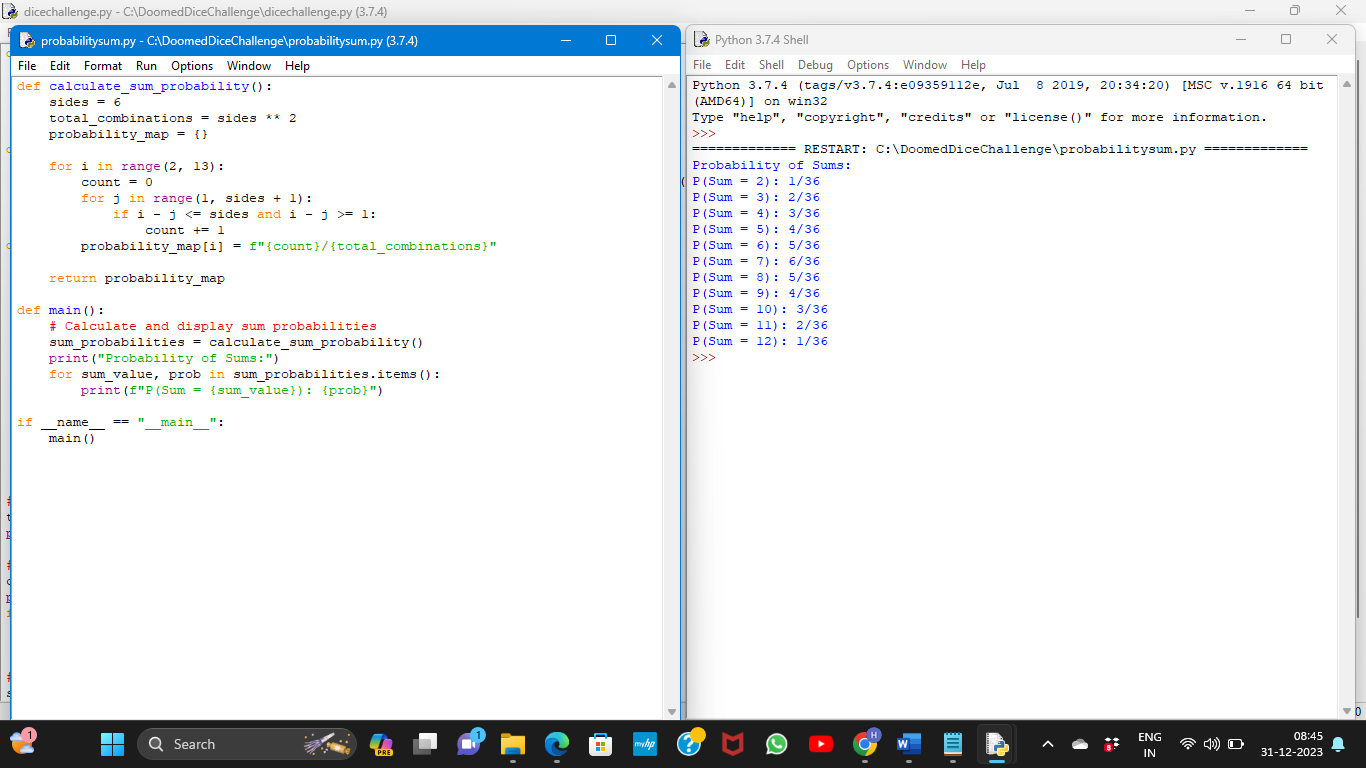
for sum\_value, prob in sum\_probabilities.items():

print(f"P(Sum = {sum\_value}): {prob}")

if \_\_name\_\_ == "\_\_main\_\_":

main()

**SCREENSHOT:**

****

**COMPLETE CODE:**

def calculate\_total\_combinations():

number\_of\_faces\_die\_a = 6

number\_of\_faces\_die\_b = 6

total\_combinations = number\_of\_faces\_die\_a\*number\_of\_faces\_die\_b

return total\_combinations

def calculate\_combination\_distribution():

distribution\_list = [

[{'Die A': i, 'Die B': j, 'Sum': i + j} for j in range(1, 7)] for i in range(1, 7)

]

return distribution\_list

def calculate\_sum\_probability():

sides = 6

total\_combinations = sides \*\* 2

probability\_map = {}

for i in range(2, 13):

count = 0

for j in range(1, sides + 1):

if i - j <= sides and i - j >= 1:

count += 1

probability\_map[i] = f"{count}/{total\_combinations}"

return probability\_map

def main():

# Calculate and display total combinations

total\_combinations = calculate\_total\_combinations()

print("Total Combinations:", total\_combinations)

# Calculate and display combination distribution

combination\_distribution = calculate\_combination\_distribution()

print("\nCombination Distribution:")

for row in combination\_distribution:

for combo\_map in row:

print(combo\_map)

# Calculate and display sum probabilities

sum\_probabilities = calculate\_sum\_probability()

print("\nProbability of Sums:")

for sum\_value, prob in sum\_probabilities.items():

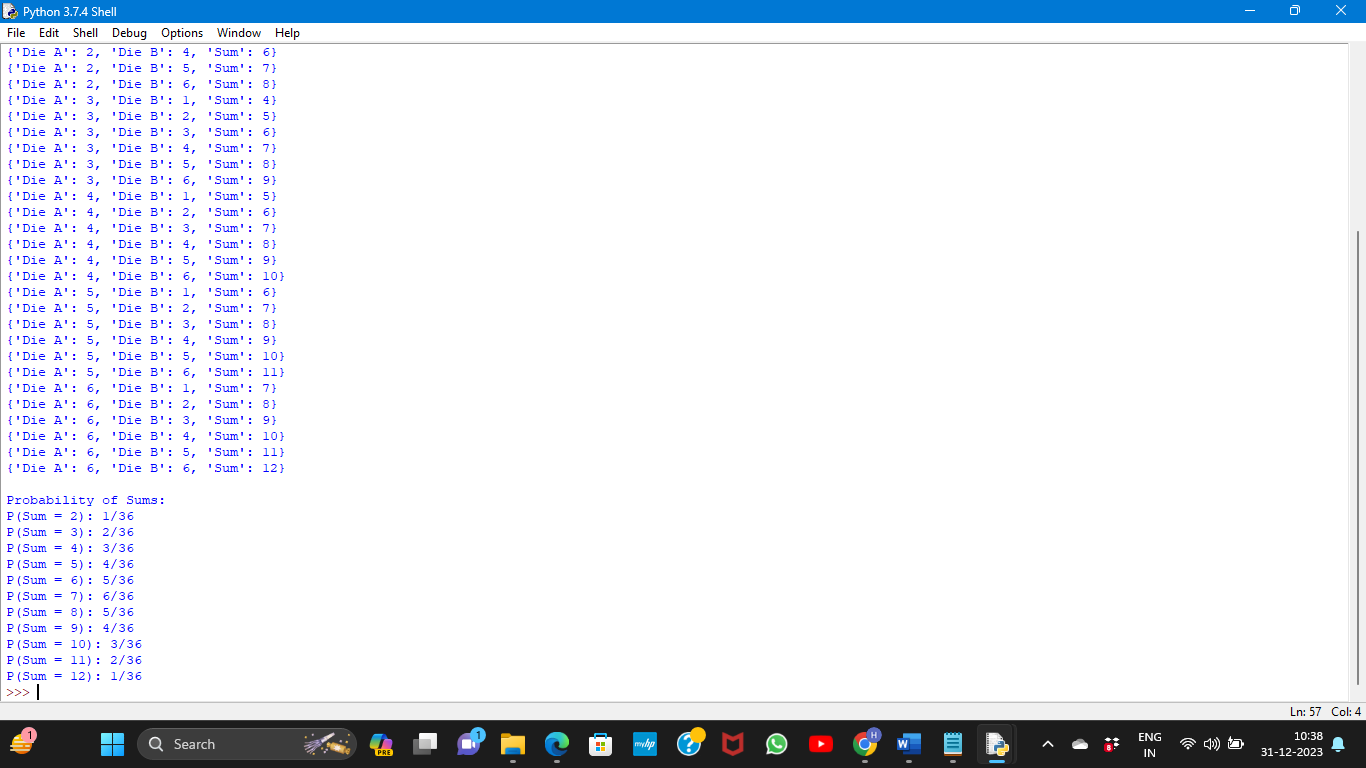
print(f"P(Sum = {sum\_value}): {prob}")

if \_\_name\_\_ == "\_\_main\_\_":

main()

**SCREENSHOT:**

****

****

**Github link:** **https://github.com/harshag782/Doomed-dice-challenge**

PART-B

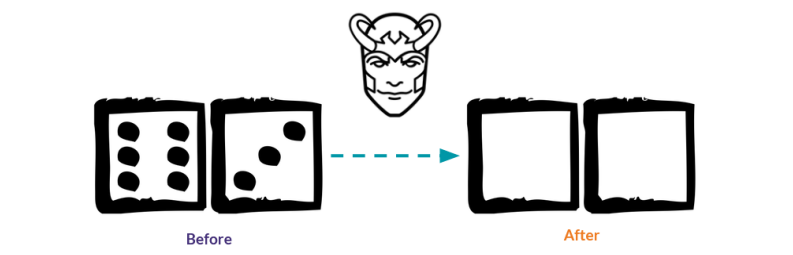
**PROBLEM STATEMENT:**

Now comes the real challenge. You were happily spending a lazy afternoon playing

your board game with your dice when suddenly the mischievous Norse God Loki (You

love Thor too much & Loki didn’t like that much) appeared.

Loki dooms your dice for his fun removing all the “Spots” off the dice.



No problem! You have the tools to re-attach the “Spots” back on the Dice.

However, Loki has doomed your dice with the following conditions:

● Die A cannot have more than 4 Spots on a face.

● Die A may have multiple faces with the same number of spots.

● Die B can have as many spots on a face as necessary i.e. even more than 6.

But in order to play your game, the probability of obtaining the Sums must remain the

same!

So, if you could only roll P (Sum = 2) = 1/X, the new dice must have the spots reattached

such that those probabilities are not changed.

**Input**:

● Die A = [1, 2, 3, 4, 5, 6] & Die B = Die A = [1, 2, 3, 4, 5, 6]

**Output**:

● A Transform Function un doom dice that takes (Die A, Die B) as input &

outputs New Die A = [? , ?, ?, ?, ?, ?],New Die B = [?, ?,?, ?, ?, ?] where,

● No New Die A[x] > 4

**REQUIREMENT SUMMARY:**

The requirement is to reattach the "Spots" back on a pair The requirement is to reattach the "Spots" back on a pair of dice that have been doomed by the mischievous Norse God Loki. The dice have the following conditions:

* Die A cannot have more than 4 spots on a face.
* Die A may have multiple faces with the same number of spots.
* Die B can have as many spots on a face as necessary, even more than 6.
* The probability of obtaining the sums when rolling the dice must remain the same.

**LOGIC:**

* First, we have the initial dice configurations for Die A and Die B, which are both set to [1, 2, 3, 4, 5, 6].
* Next, we create duplicates of the original dice configurations using the slice operator. This is done to avoid modifying the original dice configurations during the transformation process.
* The transformation logic for New \_Die\_ A is implemented using a for loop that iterates over each element in New \_ Die \_A. If an element is greater than 4, it is replaced with the lowest value in New \_ Die \_A. This ensures that Die A does not have more than 4 spots on a face.
* The transformation logic for New \_ Die \_B is also implemented using a for loop. In this case, each element in New \_ Die \_B is replaced with a sequential value from 1 to 6. This allows Die B to have as many spots on a face as necessary, even more than 6.
* Finally, the results are displayed using print statements, showing the original configurations of Die A and Die B, as well as the transformed configurations of New \_Die \_A and New\_Die\_B.

**CODE & OUTPUT:**

# Initial dice configurations

A\_die = [1, 2, 3, 4, 5, 6]

B\_die = [1, 2, 3, 4, 5, 6]

# Create duplicates of dice for transformation

New\_Die\_A = A\_die[:]

New\_Die\_B = B\_die[:]

# Transformation logic for New\_Die\_A

for i in range(len(New\_Die\_A)):

# Replace values greater than 4 with the lowest values in New\_Die\_A

if New\_Die\_A[i] > 4:

min\_value = min(New\_Die\_A)

New\_Die\_A[i] = min\_value

# Transformation logic for New\_Die\_B

for i in range(len(New\_Die\_B)):

# Replace values with sequential values from 1 to 6 in New\_Die\_B

New\_Die\_B[i] = (i % 6) + 1

# Display the results

print("Original Die\_A:", A\_die)

print("Original Die\_B:", B\_die)

print("Modified Die\_A:", New\_Die\_A)

print("Modified Die\_B:", New\_Die\_B)

OUTPUT:

