EE 381 Project 1

William Thinh Luu
California State University of Long Beach

EE 381 Probability and Statistics, with Application on Computing

Behruz Revani, Ph.D.

09/19/24

Part 1

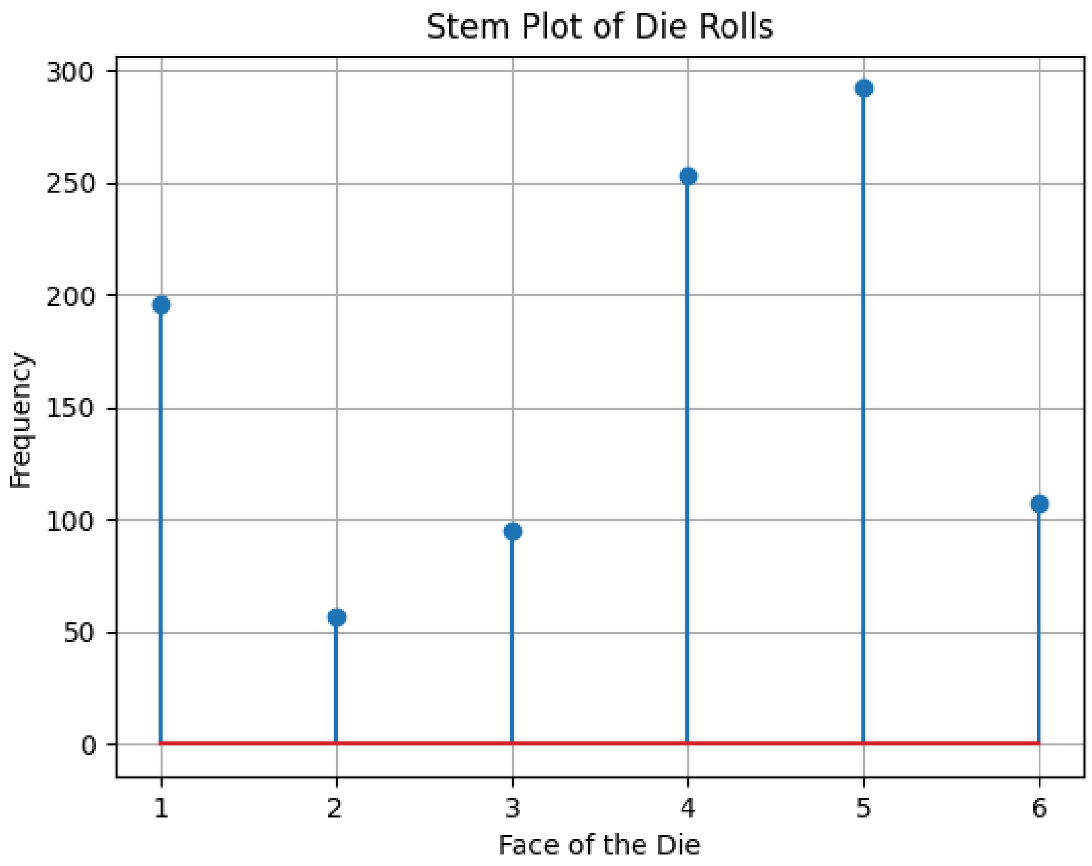
Introduction

The problem given was to simulate a single roll of n-sided die in Python, where the input consists of a vector of probabilities for each side: p = [p1, p2, p3, … , pn]. The output of the function is the number of the face of the die after a single roll from the set of integers {1, 2, …, n}. After setting the probability of a 6-sided die and calling the function 1000 times, the function would generate a stem plot displaying the probability distribution of how often each side got picked relative to its vector probability.

Methodology

First I initialize my probability vector p = [0.2, 0.05, 0.1, 0.25, 0.30, 0.1] for testing purposes. Then I implemented my function nsided_die(p) with the parameter being the vector of probability for each side of the die. Within this function, a list of die faces is created using range(1, len(p) + 1), which generates numbers from 1 to n, where n is the number of sides determined by the length of the probability vector p. Then I initialize a variable N to represent the amount of rolls. Utilizing python's random library, I used the function random.choices() that returns a list of a random selection from a given sequence, with the option of adjusting the probability of each item in the given sequence with the probability vector p. After the list of random selection is returned, the frequency of each side that was chosen by the function is initialized using the count() function. Finally the stem plot is created using python's matplotlib.pyplot library for visual results.

Results

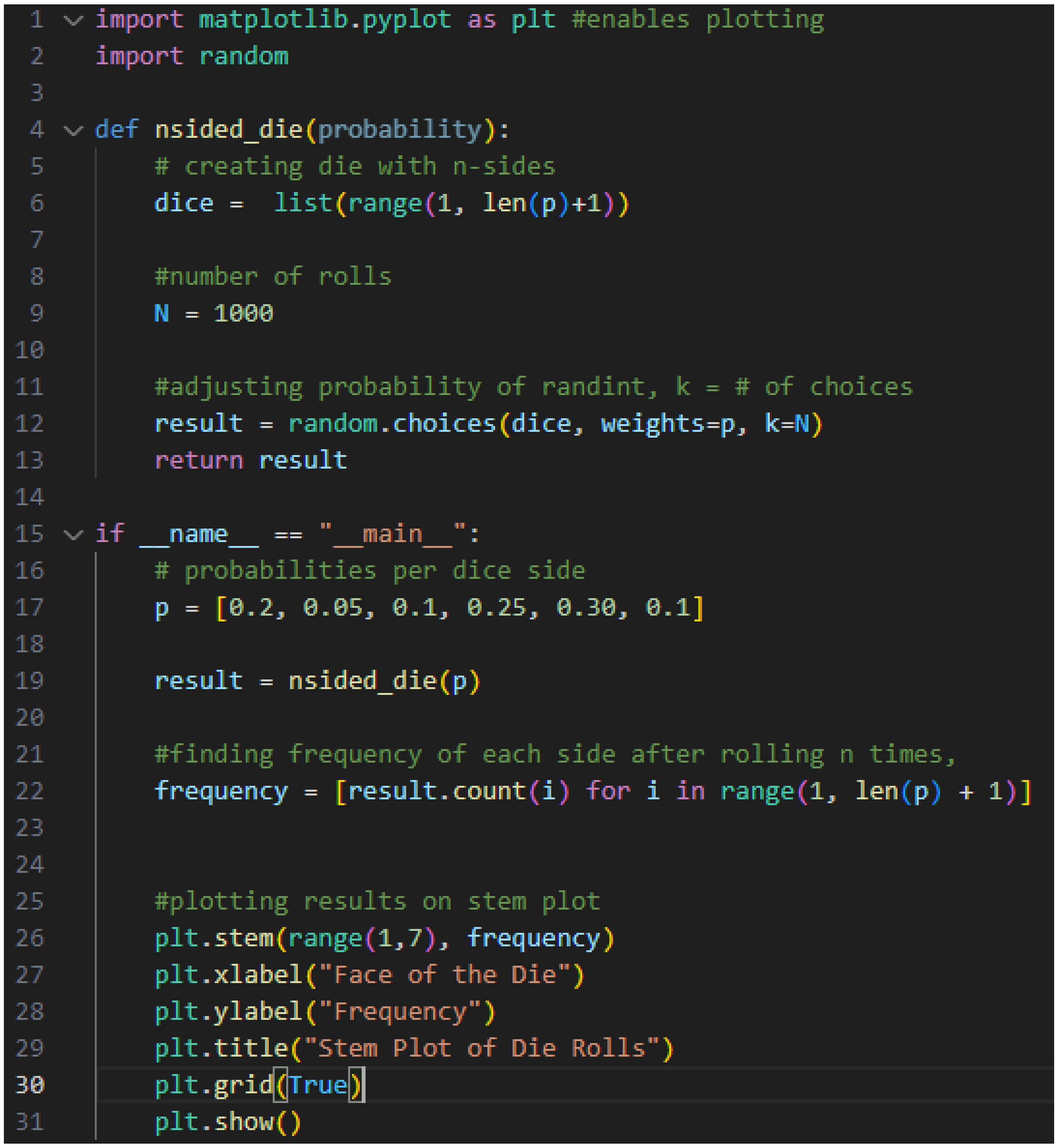


Results after 1000 rolls with p = [0.2, 0.05, 0.1, 0.25, 0.30, 0.1]

Conclusion

When the experiment is completed, the results of the frequency of each side of the die is plotted onto a stem-plot via the matplotlib library in python. When compared to the probability vector of [0.2, 0.05, 0.1, 0.25, 0.30, 0.1], the stem plot holds true with side 1 being chosen exactly 20% of the 1000 rolls, side 2 being 5% of the rolls, side 3 being 10% of the rolls, side 4 being 25% of the rolls, side 5 being the largest chosen side with 30% of the rolls, and side 6 being around 10% of the rolls. With side 3 and side 6 both being 10%, and the function inherently being random, the results of side 3 and side 6 fluctuating every function call is expected.

Appendix 1



```python
import matplotlib.pyplot as plt #enables plotting
import random

def nsided_die(probability):
    # creating die with n-sides
    dice =  list(range(1, len(p)+1))

    #number of rolls
    N = 1000

    #adjusting probability of randint, k = # of choices
    result = random.choices(dice, weights=p, k=N)
    return result

if __name__ == "__main__":
    # probabilities per dice side
    p = [0.2, 0.05, 0.1, 0.25, 0.30, 0.1]

    result = nsided_die(p)

    #finding frequency of each side after rolling n times,
    frequency = [result.count(i) for i in range(1, len(p) + 1)]


    #plotting results on stem plot
    plt.stem(range(1,7), frequency)
    plt.xlabel("Face of the Die")
    plt.ylabel("Frequency")
    plt.title("Stem Plot of Die Rolls")
    plt.grid(True)
    plt.show()
```

Part 2

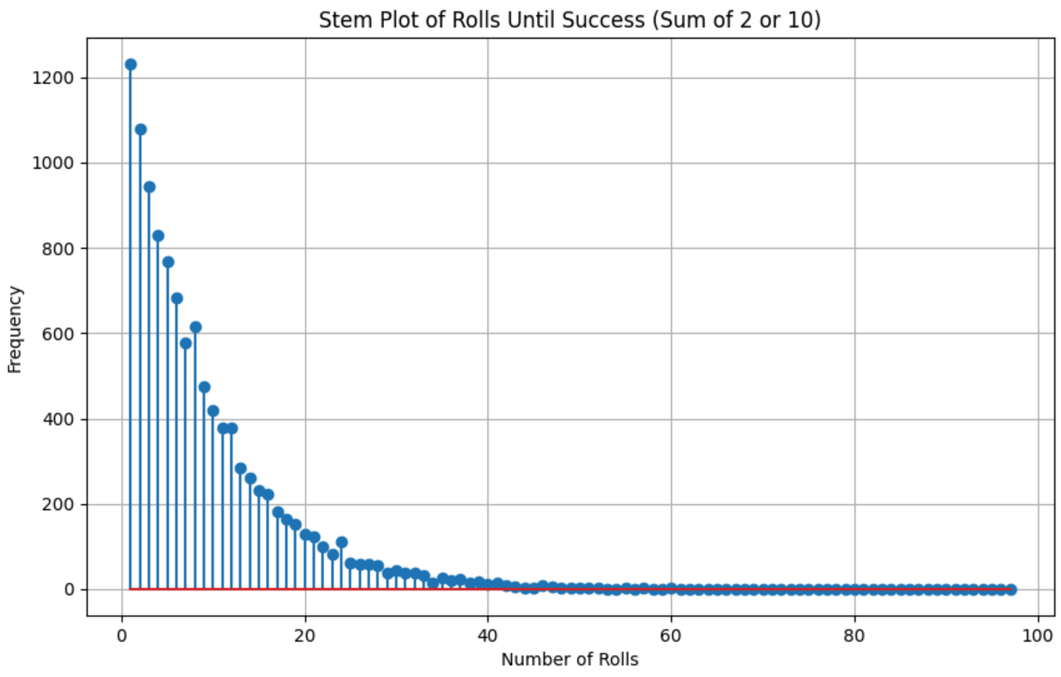
Introduction

Calculate the sum of a pair of fair dice, and determine if the sum is either 2 or 10. When the sum is 2 or 10, then the experiment is considered a “success”. Everytime that the experiment is considered a “success”, keep track of the number of rolls it took to achieve that “success”. Finally create a probability mass function graph of the number of rolls it takes for “success”.

Methodology

My approach to this problem was first to create a function called fairDie() that returns a random integer from 1 to 6 using python’s random library. In main, I initialized an empty set named success that each item will be the number of rolls taken to achieve a sum of “2 or 10”. Integer rolls are also initialized to hold the number of rolls. A for loop is created that loops from 0-100,000 for testing. Every loop, two dice are rolled using the fairDie() function and integer rolls are incremented up to indicate the dice have been rolled. An if statement is used to check the conditions if the pair of dice’s sum is equal to 2 or 10, if so then the number of rolls done to achieve this condition is appended to the set success, and rolls is reseted back to 0. The frequency is then calculated using the count() function and then later plotted onto the stem plot provided by matplotlib.pyplot library.

Results

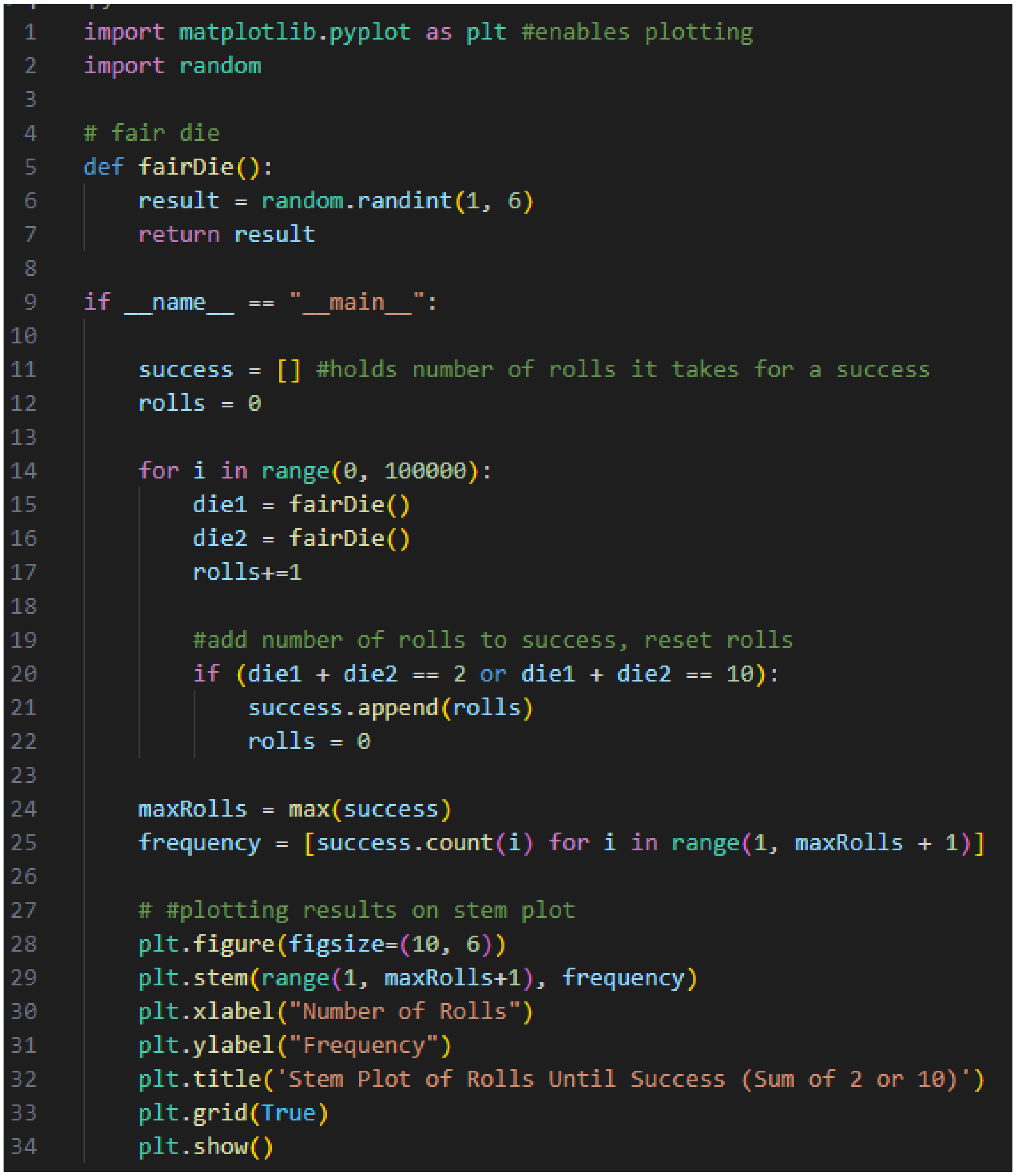


Results after 100,000 rolls that have sum of 2 or 10

Conclusion

In summary, the graph shows that the frequency of rolls required to achieve a sum of 2 or 10 tends to be skewed to the left. This means that "success" is reached within fewer rolls, indicating that the conditions of the experiment are met more frequently in a smaller number of attempts. The data suggests that it is less common to require a larger number of rolls to achieve success. This result aligns with the probability distribution of rolling two fair dice, where the sum of 10 is more likely than 2, which contributes to the overall tendency for faster success.

Appendix 2



```python
import matplotlib.pyplot as plt #enables plotting
import random

# fair die
def fairDie():
    result = random.randint(1, 6)
    return result

if __name__ == "__main__":

    success = [] #holds number of rolls it takes for a success
    rolls = 0

    for i in range(0, 100000):
        die1 = fairDie()
        die2 = fairDie()
        rolls+=1

        #add number of rolls to success, reset rolls
        if (die1 + die2 == 2 or die1 + die2 == 10):
            success.append(rolls)
            rolls = 0

    maxRolls = max(success)
    frequency = [success.count(i) for i in range(1, maxRolls + 1)]

    # #plotting results on stem plot
    plt.figure(figsize=(10, 6))
    plt.stem(range(1, maxRolls+1), frequency)
    plt.xlabel("Number of Rolls")
    plt.ylabel("Frequency")
    plt.title('Stem Plot of Rolls Until Success (Sum of 2 or 10)')
    plt.grid(True)
    plt.show()
```

Part 3

Introduction

The experiment consists of flipping 1000 fair coins and recording the number of “heads” landed. If the experiment gets exactly 500 heads then the experiment is considered a “success”. This experiment of flipping 1000 fair coins is considered a single experiment. Repeat this experiment 100,000 times and calculate the total number of successes. The probability will be calculated as the number of successes divided by 100,000.

Methodology

Utilizing python’s library of numpy, I was able to create a matrix of size N=100,000 and n=1000 where the rows are the experiments of 1000 flips, and the columns are the coin toss result with heads = 1 and tails = 0. Using np.sum(), the total headcount is calculated by looking through the matrix and searching for the total number of heads (1) in each experiment. The number of successes are calculated by using np.sum() with the parameter checking if the head count of each experiment is equal to 500.

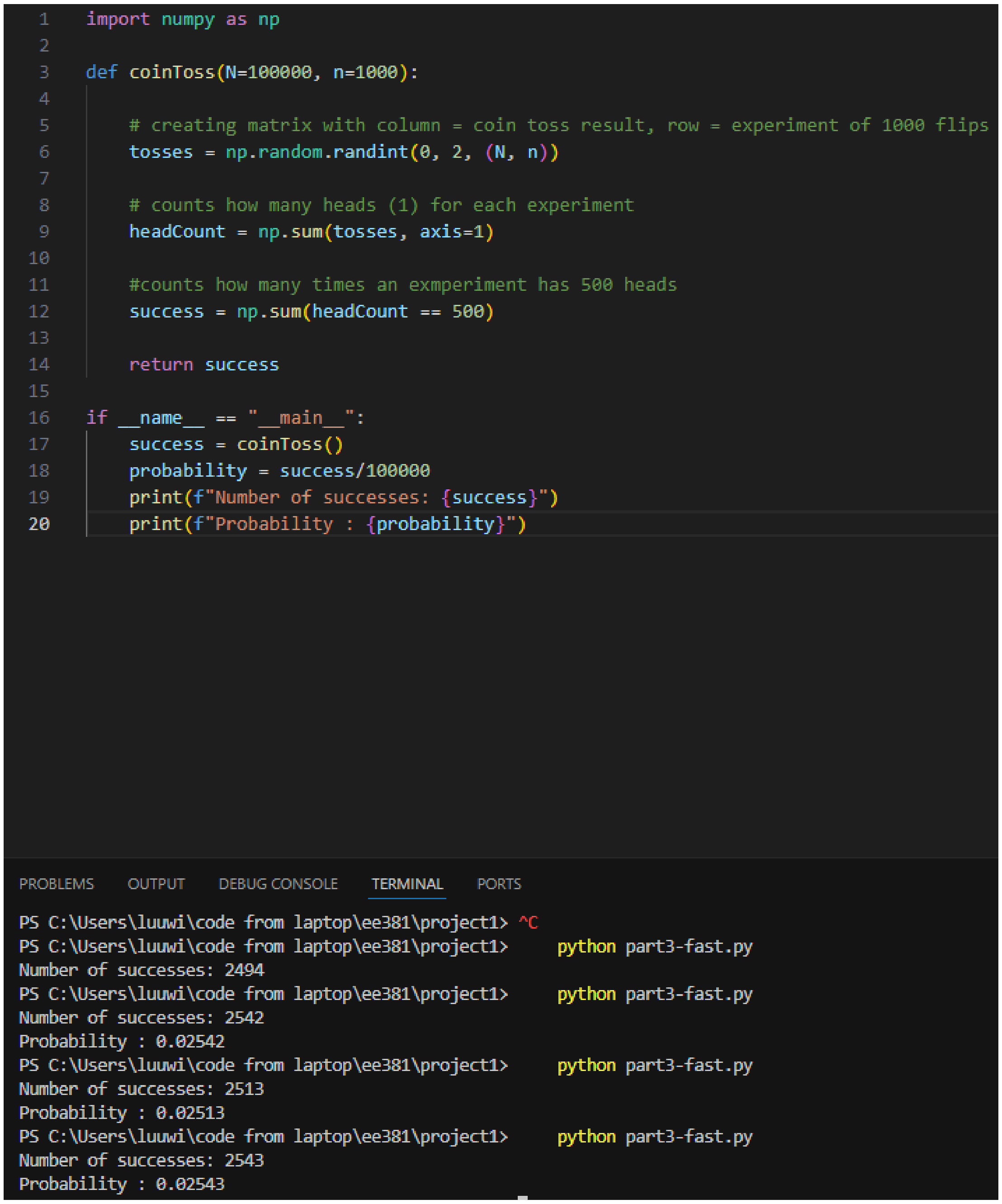
Results

Probability of 500 heads in tossing 1000 fair coins	
Ans.	**p= 0.02 or 20%**

Conclusion

The analysis conducted involved repeating the experiment of flipping 1000 fair coins 100,000 times and recording the number of "heads" in each experiment. By leveraging NumPy for efficient matrix operations, we were able to accurately determine the number of experiments that resulted in exactly 500 heads. Our results showed that the probability of achieving exactly 500 heads in 1000 coin tosses is approximately 0.02. This probability reflects the relatively rare occurrence of this specific outcome in a large number of trials.

Appendix 3



```python
import numpy as np

def coinToss(N=100000, n=1000):

    # creating matrix with column = coin toss result, row = experiment of 1000 flips
    tosses = np.random.randint(0, 2, (N, n))

    # counts how many heads (1) for each experiment
    headCount = np.sum(tosses, axis=1)

    #counts how many times an exmperiment has 500 heads
    success = np.sum(headCount == 500)

    return success

if __name__ == "__main__":
    success = coinToss()
    probability = success/100000
    print(f"Number of successes: {success}")
    print(f"Probability : {probability}")
```

```
PS C:\Users\luuwi\code from laptop\ee381\project1> ^C
PS C:\Users\luuwi\code from laptop\ee381\project1>     python part3-fast.py
Number of successes: 2494
PS C:\Users\luuwi\code from laptop\ee381\project1>     python part3-fast.py
Number of successes: 2542
Probability : 0.02542
PS C:\Users\luuwi\code from laptop\ee381\project1>     python part3-fast.py
Number of successes: 2513
Probability : 0.02513
PS C:\Users\luuwi\code from laptop\ee381\project1>     python part3-fast.py
Number of successes: 2543
Probability : 0.02543
```

Part 4

Introduction

Given a single experiment where you draw 5 cards from a 52 card deck. The experiment is considered a “success” if and only if you get 4 of a kind from that draw of 5 cards. A 4 of a kind refers to when a draw has 4 cards that are the same rank. For example : 7 of hearts, 7 of clubs, 7 of diamonds, 7 of spades. After repeating the experiment 1,000,000 times, determine the probability of getting 4 of a kind by keeping track of the number of successes from the single experiments, then dividing by 1,000,000.

Methodology

I first started with creating a function that will create a list of cards with each card having a suit and a rank. Then I created two boolean helper functions, one to check the hand if there are any 4 of a kind, and the other to help with card drawing. The function drawAndCheck(deck, n=5) randomly chooses 5 indices in the deck that cannot be picked again, and then those 5 indices are drawn from the deck and placed into the hand. isFourOfAKind(hand) is called, and creates a list of all ranks in the hand. With the use of Counter(), the occurrence of each rank is returned as a list of key value pairs, where the key is the rank, and the value is the amount of occurrences. Later this is returned as a bool, checking if there is a rank with an occurrence of 5. A function experiment is created which loops N = 1,000,000 times, using the drawAndCheck(deck) function to see if there was a success, which later probability is calculated.

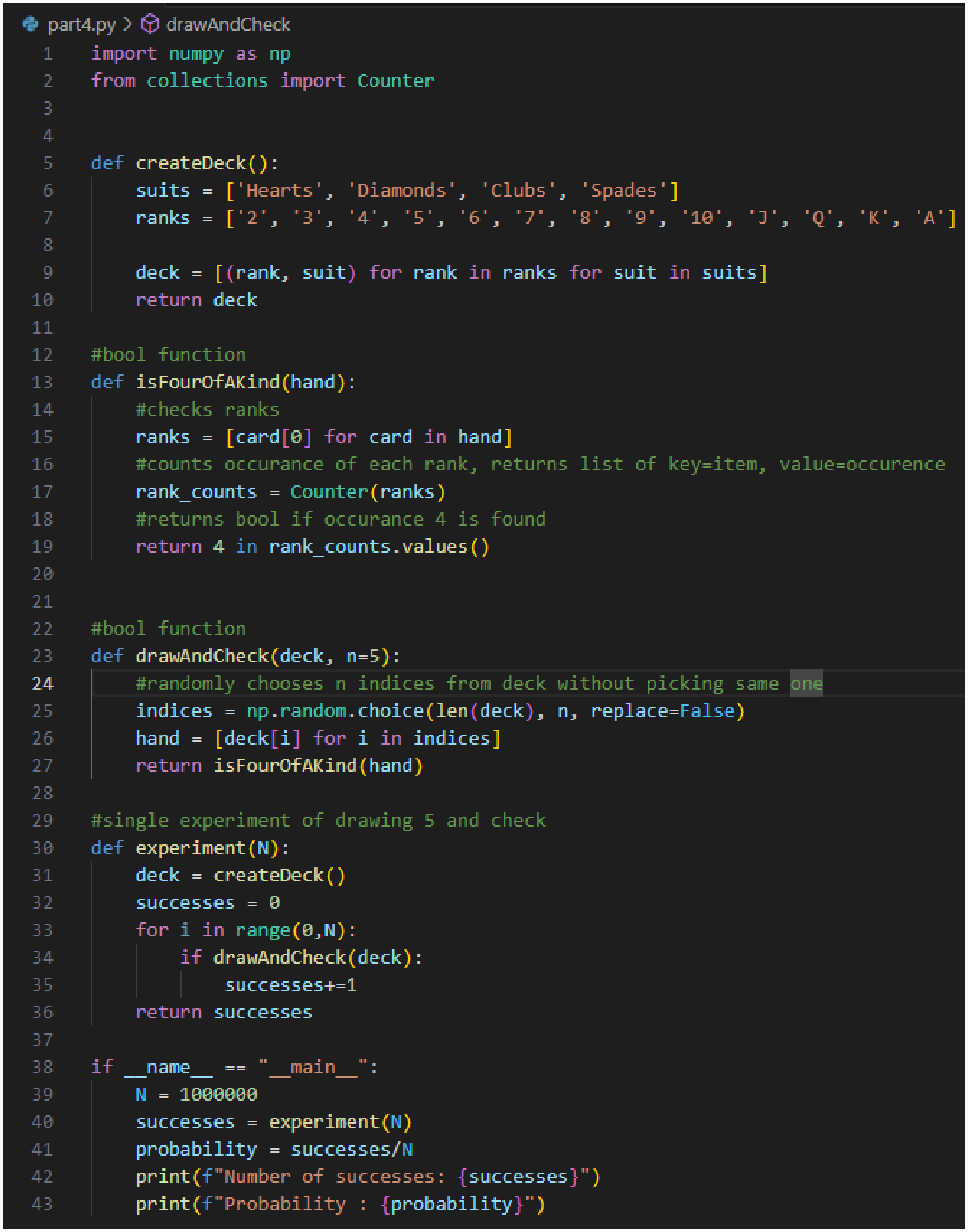
Results

Probability of 4 of a kind	
Ans.	**p= 0.00026 or 0.026%**

Conclusion

In conclusion, the simulation yielded a probability of 0.00026, or 0.026%, for drawing 4 of a kind from a 5-card hand. This result is close to the theoretical probability of approximately 0.0002401, or 0.02401%. The minor discrepancy is due to the inherent variability in random sampling. Overall, the simulation effectively estimates the rare event of drawing 4 of a kind.

Appendix 4


part4.py > drawAndCheck

```python
import numpy as np
from collections import Counter


def createDeck():
    suits = ['Hearts', 'Diamonds', 'Clubs', 'Spades']
    ranks = ['2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K', 'A']

    deck = [(rank, suit) for rank in ranks for suit in suits]
    return deck

#bool function
def isFourOfAKind(hand):
    #checks ranks
    ranks = [card[0] for card in hand]
    #counts occurance of each rank, returns list of key=item, value=occurence
    rank_counts = Counter(ranks)
    #returns bool if occurance 4 is found
    return 4 in rank_counts.values()


#bool function
def drawAndCheck(deck, n=5):
    #randomly chooses n indices from deck without picking same one
    indices = np.random.choice(len(deck), n, replace=False)
    hand = [deck[i] for i in indices]
    return isFourOfAKind(hand)

#single experiment of drawing 5 and check
def experiment(N):
    deck = createDeck()
    successes = 0
    for i in range(0,N):
        if drawAndCheck(deck):
            successes+=1
    return successes

if __name__ == "__main__":
    N = 1000000
    successes = experiment(N)
    probability = successes/N
    print(f"Number of successes: {successes}")
    print(f"Probability : {probability}")
```

Part 5

Introduction

When given a 4-digit passcode, the total number of unique combinations is 10^4. A hacker creates a list of m=10^4 randomly generated 4-digit passcodes with the possibility some passcodes will be duplicates. A single experiment is made when a passcode is checked to see if it is in the list of randomly generated 4-digit passcodes. The experiment is a success if the passcode is in the list. Repeat this N = 1000 times and find its probability of success after 1000 times. Then test to see the probability of success when the hacker's list size is m = 10^6 with randomly generated 4-digit passcodes. Finally, trial and error will be used to determine the approximate list size m required to achieve a 50% probability of finding at least one matching passcode.

Methodology

Since the hacker's list needs to be implemented, I will create a function with a list. By utilizing python's random library, I used the function random.randint() to generate a random number from 0 to 9999. Using f strings in python, I added :04d to the end of the randint() function to ensure the randomly generated digit is a string with 4 digits. For example, 9 would be '0009'. To finish off the hacker's list I added a for loop that loops until it reaches the size of m. Using the same logic of generating a random number with 4-digits, I created another function to generate one for the user. The experiment is run by looping through the created list and checking if the user's password is in the password list. If this is true then a variable called successes is incremented and then returned when the experiment is finished looping. Probability is found using the formula successes/N, where N is the amount of experiment runs.

Results

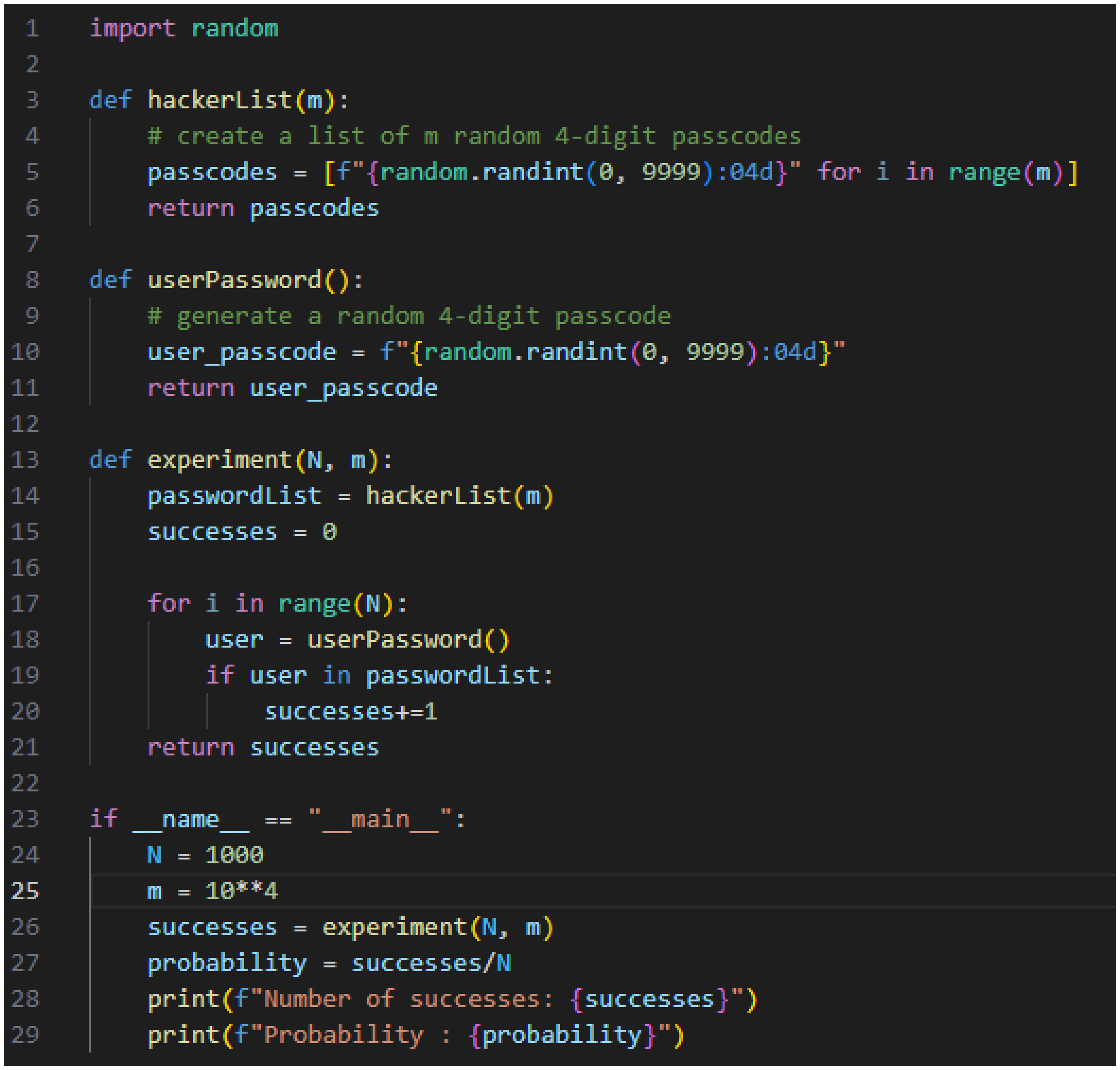
Prob. that at least one of the numbers matches the passcode m=10^4	**p=0.636 or 63%**
Prob. that at least one of the numbers matches the passcode	**p=1.0 or 100%**

m=10^6	
Approximate number of numbers in the list p=0.5	**m= 7000**

Conclusion

After a couple of runs of the program with m=10^4, the percentage varied from 60 to 64 percent, with it averaging out around 63% after 10 runs. In comparison to the experiment running on m=10^4, when running the experiment with m=10^6, it results in a 100% probability of the hacker's list having a matching password with the user. This is due to 4-digit passwords only having 10,000 combinations, therefore when running the experiment where the hacker's list contains 1,000,000 randomly generated 4-digit passcodes, it almost certainly guarantees a matching password. When finding the approximate number (m) that would lead to a probability of 0.5, after trial and error, 7000 gave the most consistent probability of approximately 0.5.

Appendix 5



```python
import random

def hackerList(m):
    # create a list of m random 4-digit passcodes
    passcodes = [f"{random.randint(0, 9999):04d}" for i in range(m)]
    return passcodes

def userPassword():
    # generate a random 4-digit passcode
    user_passcode = f"{random.randint(0, 9999):04d}"
    return user_passcode

def experiment(N, m):
    passwordList = hackerList(m)
    successes = 0

    for i in range(N):
        user = userPassword()
        if user in passwordList:
            successes+=1
    return successes

if __name__ == "__main__":
    N = 1000
    m = 10**4
    successes = experiment(N, m)
    probability = successes/N
    print(f"Number of successes: {successes}")
    print(f"Probability : {probability}")
```