Extra Credit - RPG Dice Simulator

John Karasinski
karasinski@gmail.com

1 Introduction

> *God's dice always have a lucky roll.*
>
> Sophocles (497/6 — 406/5 BC)
>
> *God does not play dice.*
>
> Albert Einstein (1879 — 1955)
>
> *God not only plays dice, but also sometimes throws them where they cannot be seen.*
>
> Stephen Hawking (1942 —)

While God's use of dice may be disputed, humans have been using dice in games of chance for many thousands of years [1]. Dice insert a certain amount of randomness into games, and can lead to wonderfully unpredictable situations. Perhaps more importantly, and in spite of the randomness associated with them, dice are also excellent for bringing everyone to the same level. A king and a peasant have the same odds of rolling a six, and the die does not care who is more wealthy, talented, or desperate. Despite their use for thousands of years, humans are very bad at predicting the outcomes of dice throws. Luckily, while the result of any individual roll of the die may be random and unknown, a basic application of statistics can lead us to make more informed decisions about the likelihood of events.

The only assumption going forward is that we are considering perfect dice. These perfect dice have the property of having an equal probability of landing on any particular face. According to standard die notation, die rolls are given in the form AdX. A and X are variables, separated by the letter "d", which stands for die or dice [2]. A six-sided die (1d6), for example, would then have a $1/6$ probability of resulting in a 1, 2, 3, 4, 5 or 6. Cheap dice purchased for gaming will not have this property, and will instead slightly favor different outcomes. For the purposes of the simulations here, you can assume that we have purchased *very, very expensive* dice which behave perfectly.

2 Instructions

The project should consist of a brief write-up introducing relevant theory behind each simulation problem, your simulation method, your results, and a summary of your conclusions. I'm looking for short papers ranging 5-8 pages in length. These papers should focus on the simulation methodology and results, with these sections making up the bulk of the paper.

Create a function that will allow me to specify the number of dice I want to use per roll, the number of faces on each die (assume all dice used in a roll have the same number of faces), whether I will sum the scores of each die or will have a minimum value needed to count as a success per die, and whether I will include botches or not as an outcome for rolls.

Example requirements:

1. Roll 1d20 and I just want to know the score.
2. Roll 2d6 and I want to know my summed scores.
3. Roll 4d10 where I need to roll 5s or greater such that each roll of 5 or greater counts as a success.
4. The same scenario as above, but now each 1 rolled detracts from the number of successes.

3 Simulation Method

This simulation will make use of Python 3.5.0. The core idea behind the simulation is to generate pseudo-random numbers to simulate the throwing of many dice. From

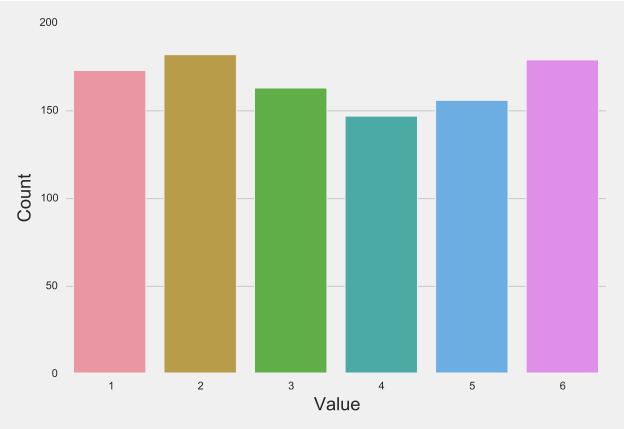

Figure 1: Results from 1,000 1d6 rolls.

Value	Count	Frequency
1	173	17.3
2	182	18.2
3	163	16.3
4	147	14.7
5	156	15.6
6	179	17.9

Table 1: Results of 1,000 1d6 rolls

there, the rules given in the problem prompt can be applied to deduce the result of the throw.

The basic functionality behind the simulation is the `dice()` function. This function accepts three arguments, `A`, `X`, and an optional `n`, where `A` and `X` take on the same meaning as in standard dice notation, and n is the number of times to simulate the throw. Calling `dice(3, 6, n=2)`, for example, would return `array([[2, 2, 6], [5, 3, 4]])`. This array has 2 rows of 3 simulated 6-sided dice.

The `dice()` function is essentially a helper function, which calls Numpy's random integer function. Numpy uses the Mersenne twister sequence to generate their random numbers. As these are psuedo-random numbers and not true random numbers, knowing the seed of the algorithm allows you to reproduce the results, and predict future events with 100% accuracy.

4 Results

a. Roll 1d6 1,000 times, frequency of each outcome?

b. Roll 2d6 1,000 times, frequency of each outcome? Use the summed scores of each die as your outcome.

c. Roll 6d10 1,000 times, frequency of each outcome? Treat a score of 6 or better as a success and scores less than 6 as nothing, the outcome is total number of successes (include the frequency of no successes).

d. Same as (c) above except now scores of 1 count as

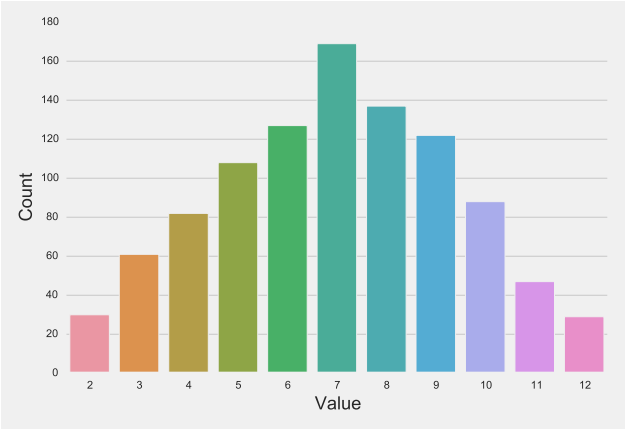

Figure 2: Results of summed values from 1,000 2d6 rolls.

Value	Count	Frequency
2	30	3.0
3	61	6.1
4	82	8.2
5	108	10.8
6	127	12.7
7	169	16.9
8	137	13.7
9	122	12.2
10	88	8.8
11	47	4.7
12	29	2.9

Table 2: Results of summed values from 1,000 2d6 rolls.

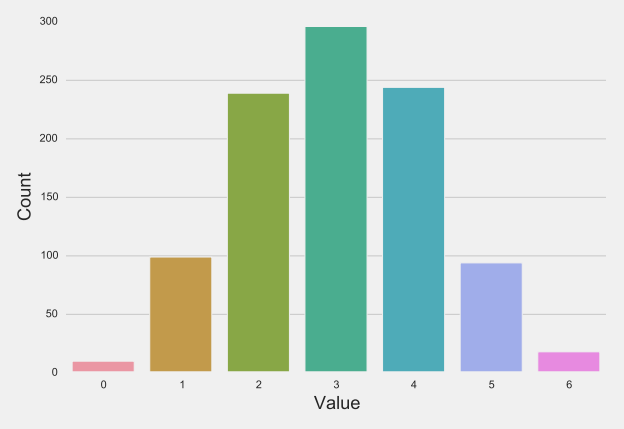

Figure 3: Results of succeses from 1,000 6d10 rolls, where scores of 6 or greater are considered successes.

Value	Count	Frequency
0	10	1.0
1	99	9.9
2	239	23.9
3	296	29.6
4	244	24.4
5	94	9.4
6	18	1.8

Table 3: Results of succeses from 1,000 6d10 rolls, where scores of 6 or greater are considered successes.

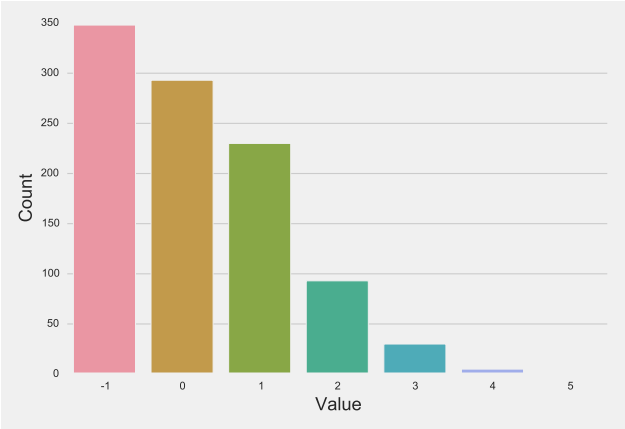

Figure 4: Results of succeses from 1,000 6d10 rolls, where scores of 6 or greater are considered successes, and scores of 1 are considered botches.

a botch. Each botch is subtracted from each success. If there are more botches than success (i.e., -1 successes) then the overall outcome is a botch. Now depict the frequency of botching, and total number of successes (as before, include the frequency of no successes).

Value	Count	Frequency
-1	348	34.8
0	293	29.3
1	230	23.0
2	93	9.3
3	30	3.0
4	5	0.5
5	1	0.1

Table 4: Results of succeses from 1,000 6d10 rolls, where scores of 6 or greater are considered successes, and scores of 1 are considered botches.

5 Summary

References

[1] Carlisle, Rodney P., ed. Encyclopedia of play in today's society. Vol. 1. Sage, 2009.

[2] "Standard Dice Notation". dice-play. 2006-04-06. Archived from the original on 2007-04-26. https://web.archive.org/web/20070426013749/http://homepage.ntlworld.com/dice-play/Notation.htm

A Source Code

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style('white')
plt.style.use('fivethirtyeight')


def dice(A, X, n=1):
    ''' Generate n instances of an AdX dice roll.'''

    return np.random.randint(low=1, high=X+1, size=(n, A))


def throw(AdX, n=1, success_min=None, botching=False):
    '''
    Throw dice with optional repetitions, minimum value for success, and
    botching.
    '''

    # Pull the values for your roll.
    A, X = np.array(AdX.split('d'), dtype=int)

    if n == 1 and success_min is None:
        # Just the results of the dice roll.
        if A == 1:
            return dice(A, X)[0][0]
        else:
            return dice(A, X).sum()
    elif success_min is None:
        # Statistics on the result of many dice rolls.
        scores = dice(A, X, n).sum(axis=1)
    elif success_min and not botching:
        # Statistics of the results of many dice rolls with a minimum value for
        # success.
        scores = (dice(A, X, n) >= success_min).sum(axis=1)

        # Just the number of successes for this roll.
        if n == 1:
            return scores[0]
    elif botching:
        # Statistics of the results of many dice rolls with a minimum value for
        # success and the possibility of botching.
        d = dice(A, X, n)
        d[d == 1] = -1
        d[np.logical_and(d > 0, d < success_min)] = 0
        d[d > 0] = 1
        scores = d.sum(axis=1)
        scores[scores < 0] = -1
        if n == 1:
            return scores[0]
    else:
        raise ValueError('Sorry, these values are not supported.')

    return scores
```

```
56
57
58  def statistics(scores, plot=False, save=False):
59      # Generate the statistics.
60      s = pd.Series(scores).value_counts().reset_index()
61      s.columns = ['Value', 'Count']
62      s['Frequency'] = s['Count']/s['Count'].sum() * 100
63      s = s.sort('Value')
64
65      print(s)
66      if plot:
67          sns.countplot(scores, order=np.unique(scores))
68          plt.xlabel('Value')
69          plt.ylabel('Count')
70          plt.tight_layout()
71          if save:
72              plt.savefig(save + '.pdf')
73          plt.close('all')
74          #plt.show()
75      return s
76
77
78  def main():
79      # %timeit throw('1d20')
80      # 100000 loops, best of 3: 10.4 s per loop
81
82      # %timeit throw('2d6')
83      # 100000 loops, best of 3: 15.4 s per loop
84
85      # %timeit throw('4d10', success_min=5)
86      # 10000 loops, best of 3: 20.4 s per loop
87
88      # %timeit throw('4d10', success_min=5, botching=True)
89      # 10000 loops, best of 3: 32.3 s per loop
90
91      print('a. Roll 1d6 1,000 times, frequency of each outcome?\n')
92      statistics(throw('1d6', n=1000), plot=True, save='1d6')
93
94      print('b. Roll 2d6 1,000 times, frequency of each outcome? Use the summed scores of each die
        as your outcome.\n')
95      statistics(throw('2d6', n=1000), plot=True, save='2d6')
96
97      print('c. Roll 6d10 1,000 times, frequency of each outcome? Treat a score of 6 or better as a
         success and scores less than 6 as nothing, the outcome is total number of successes (include
         the frequency of no successes).\n')
98      statistics(throw('6d10', n=1000, success_min=6), plot=True, save='6d10>6')
99
100     print('d. Same as (c) above except now scores of 1 count as a botch. Each botch is subtracted
         from each success. If there are more botches than success (i.e., -1 successes) then the
        overall outcome is a botch. Now depict the frequency of botching, and total number of
        successes (as before, include the frequency of no successes).\n')
101     statistics(throw('6d6', n=1000, success_min=6, botching=True), plot=True, save='6d10>6-
        botching')
102
103 if __name__ == '__main__':
104     main()
```