```
# Python Part

# Keyan Vakil
from matplotlib import pyplot as plt
import numpy as np
from numpy.random import randint
import random


# Sources used:
# Stack Overflow
# Youtube matplotlib tutorials by Sentdex
# Matplotlib pyPlot documentation
# This website: http://sphinxcontrib-napoleon.read
thedocs.io/en/latest/example_google.html
    # for doctstrings

# Discussed the assignment with Ray Katz and Ellie
...?

# No late days used in general or for this assignme
nt


def ex1():
    """plot the points (1,1),(2,2),(3,3),(4,4)

    No args

    Returns:
        plot: a plot with a dotted red line going
        through each point
        plot: a different plot with 4 points
        as blue triangles going through each point

    """
    # create x and y values
    x = [1,2,3,4]
    y = [1,2,3,4]

    # make sure they can represent points
    assert len(x) == len(y)
```

```
    #iterate through the points
    for i in range(len(y)):
        #switch to the first plot
        plt.figure(0)
        #plot the point on the first plot as a blue
 triangle
        plt.plot(x[i], y[i], 'b^')
        #draw the figure
        plt.draw()

    #switch to the second plot
    plt.figure(1)
    #plot the points on the second plot as a red do
tted line
    plt.plot(x,y, linestyle=':', color='r')

    #draw the figure
    plt.draw()
    #show the plot
    plt.show()


def dieroll(m):
    """ dieroll estimates the PMF of a random varia
ble Y
    where Y represents the number that comes up whe
n
    you roll a fair die

    Args:
        m (int): The number of rolls to simulate

    Returns:
        - a plot of the PMF of the random variable

    """
    # number of rolls of m die
    number_of_rolls = 10000

    #Generate an array of size num_of rolls with in
tegers 1 thorugh 6
```

```
    #create all die rolls necessary
    rolls = randint(1,7,(1,number_of_rolls*m))
    #The previous line returns a multi-dimensional
array,
    #and we only want the first row of the output
    rolls = rolls[0]

    #intantiate y_series data
    y_series = []
    #iterate through each group of die rolls
    for i in range(number_of_rolls):
        #initialize the end value to 0
        val = 0
        #for each group of die rolls
        for j in range(m):
            #get the sum of the group of die rolls
            val += rolls[i*m+j]
        #add the sum of the group of die rolls to t
he y_series
        y_series.append(val)
    #Change the list into a numpy array
    y_series = np.array(y_series)

    #Create a figure
    plt.figure(1)

    #Use the pyPlot.hist() function to plot the dat
a.
    plt.hist(y_series,
    #This sets the number of bars in the histogram
    #assuming the die is 6 sided
    # the range of the bins to be displayed
    # (formatted for visual appeasement)
    bins=range(m-1, 6*m+2),
    #This changes the relative width of the bars
    width=.7,
    #This centers the bars
    align='mid',
    #This reweights the data so we see probabilitie
s on the y-axis
    weights=np.zeros_like(y_series) + 1. / y_series
```

```
.size)

    #Set some feature of the graph, so it looks goo
d
    #X-axis size (makes sure all bars are visible)
    # plt.xlim(m-1,5*m+3)
    plt.title("Rolling a Die")
    plt.xlabel("Value")
    plt.ylabel("Probability")

    #Show the figure
    plt.show()


def pi_Estimate(numPoints):
    """ estimates pi=3.14 using the Monte Carlo Met
hod

    Args:
        numPoints (int): The number of points used
to calculate
        the value of pi

    Returns:
        float: The estimated value of pi.

    """
    # count of the points in the circle
    C = 0
    # count of the points in total
    T = 0

    # for each point
    for _ in range(numPoints):
        # create a random x, y pair to represent co
ordinates
        x = random.uniform(-1, 1)
        y = random.uniform(-1, 1)

        # if the coordinates are contained within t
he
        # unit circle (if the distance from the cen
```

```
ter
        # is less than the radius of the unit circl
e = 1)
        if (x**2 + y**2) < 1.0:
            # add one to the in_circle count
            C += 1
        T += 1

    # 4 times this ratio will equal pi
    # this can be determined by manipulating the fo
rmula for the
    # area of a circle (pi*r^2) and the formula for
 the area of
    # a square (4*r^2)
    return 4.0*C / T


def expt(p):
    """ simulates flipping a p-biased coin until th
e
    coin comes up as heads

    Args:
        p (float): A float value from 0-1 inclusive
 that
        represents the bias of the

    Returns:
        float: The estimated value of pi.

    """
    # function that simulates flipping a p-biased
    # coin until the coin comes up heads, and outpu
ts
    # the numer of flips that were required (up to
and
    # including the final flip)

    #keep track of the count
    count = 0
    # keep iterating until we get a head
    while True:
```

```
        # add one to the count
        count += 1
        # get the random number
        num = np.random.random()
        # if the random number is less than p
        if num < p:
            # that would be considered a heads,
            # so return the count
            return count


def Coin_flip_test(n,p):
    """ runs expt(p) multiple times, each with inde
pendent
    randomness

    Args:
        n (int): The number of times to run expt(p)

        p (float): The float value to be passed int
o the expt
        function

    Returns:
        list: A list of ints containing the counts
for
        each call of expt(p)

    """
    # runs expt(p) n different times, each with
    # independent randomness
    results = []
    while n > 0:
        results.append(expt(p))
        n -= 1
    return results


def plot_coin_flips(n,p):
    """ plots a histogram of the output of
    Coin_flip_test(n,p)
```

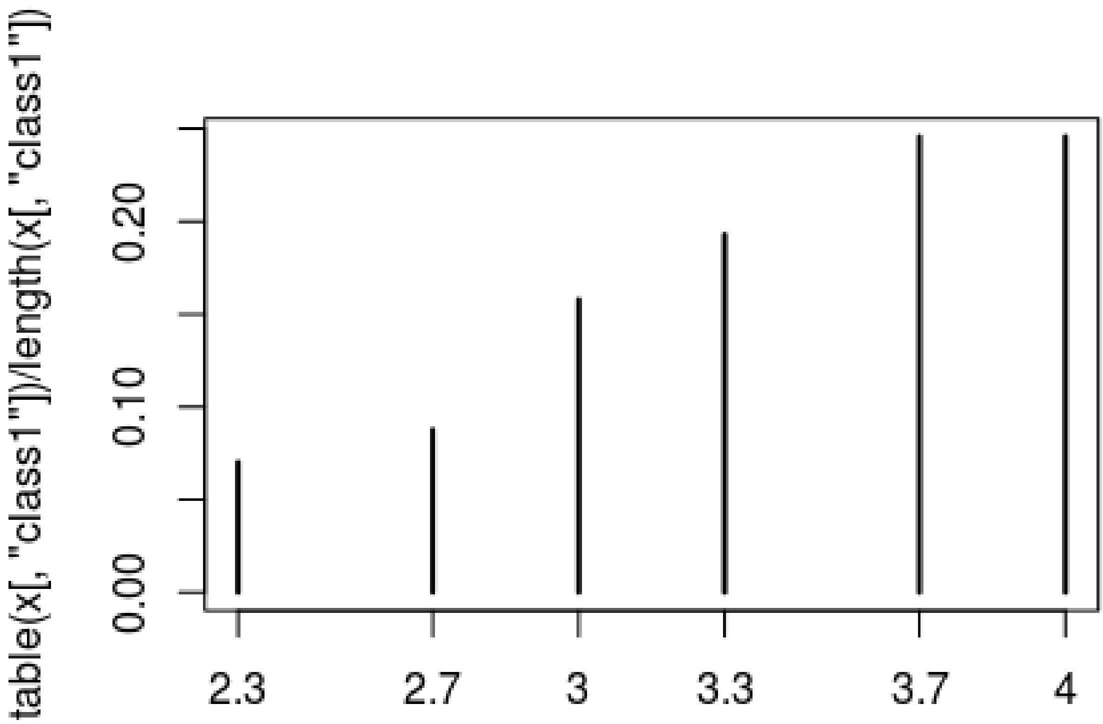
```
    Args:
        n (int): Number of flips to be passed into
        Coin_flip_test
        p (float): Weight of the coin to be passed
into
        Coin_flip_test

    Returns:
        plot: A histogram of the output of the func
tion

    """



    def draw_pmf(results, p):
        """ function that plots the PMF correspondi
ng
        with the values shown in plot_coin_flips

        Args:
            results (list): A list of the results o
f
            Coin_flip_test so that the pmf correspo
nds
            to the same results that were graphed (
we can't
            call Coin_flip_test again because the r
esults
            would be different)
            p (float): Weight of the coin to be use
d for
            calculation of the pmf

        Returns:
            plot: A curved line representing the PM
F of
            the frequencies in question

        """

        #make sure we're on figure 1
```

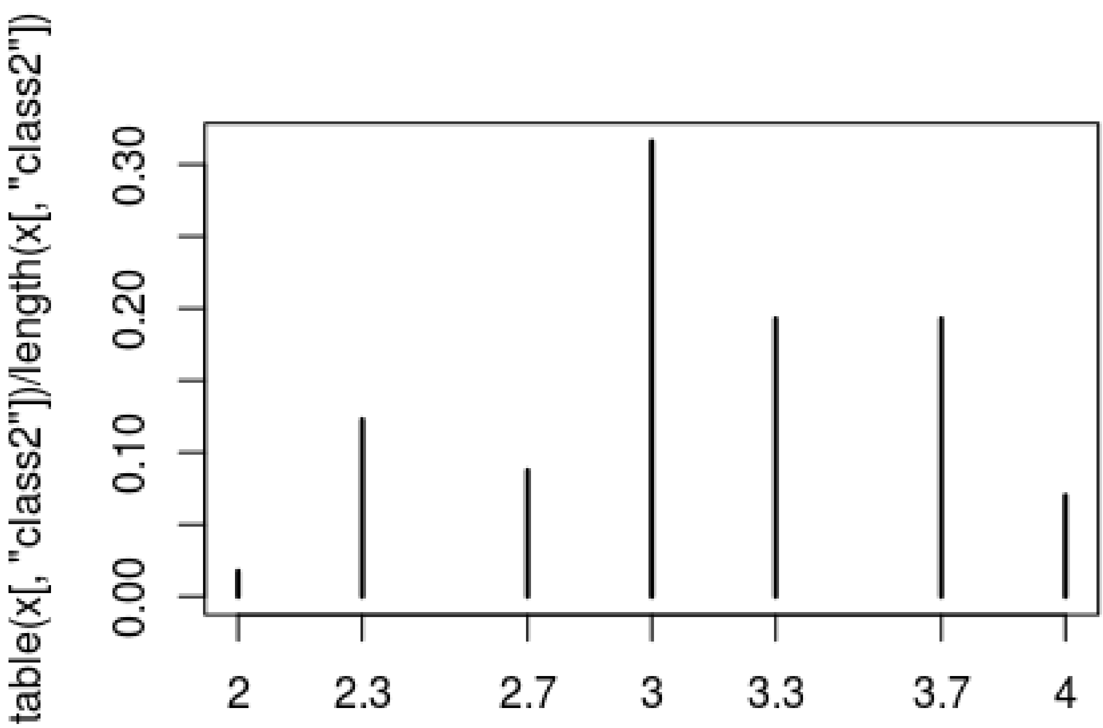
```
        plt.figure(1)
        # get the max value of the results...
        max_val = np.amax(results)
        # ...in order to find the range of the x va
lues
        # in the graph
        x = range(1, max_val+1)
        # find the correspond PMF values
        y = [p*(1-p)**(val-1) for val in x]
        # plot the values
        plt.plot(x, y)
        # draw the values on the graph
        plt.draw()

    # make sure we're on figure 1
    plt.figure(1)

    # get the result of flipping a p-biased coin n
times
    results = Coin_flip_test(n,p)
    # convert to a numpy array
    results = np.array(results)

    # plot a histogram of the frequencies of the re
sults
    plt.hist(results,
        # center align the bins
        align='mid',
        # the number of bins should be equal to the

        # max value within the results
        bins=np.amax(results),
        #This reweights the data so we see probabil
ities on the y-axis
        weights=np.zeros_like(results) + 1. / resul
ts.size)

    # draw the histogram
    plt.draw()
    # draw the pmf function on top of the histogram

    # (for problem 6)
```

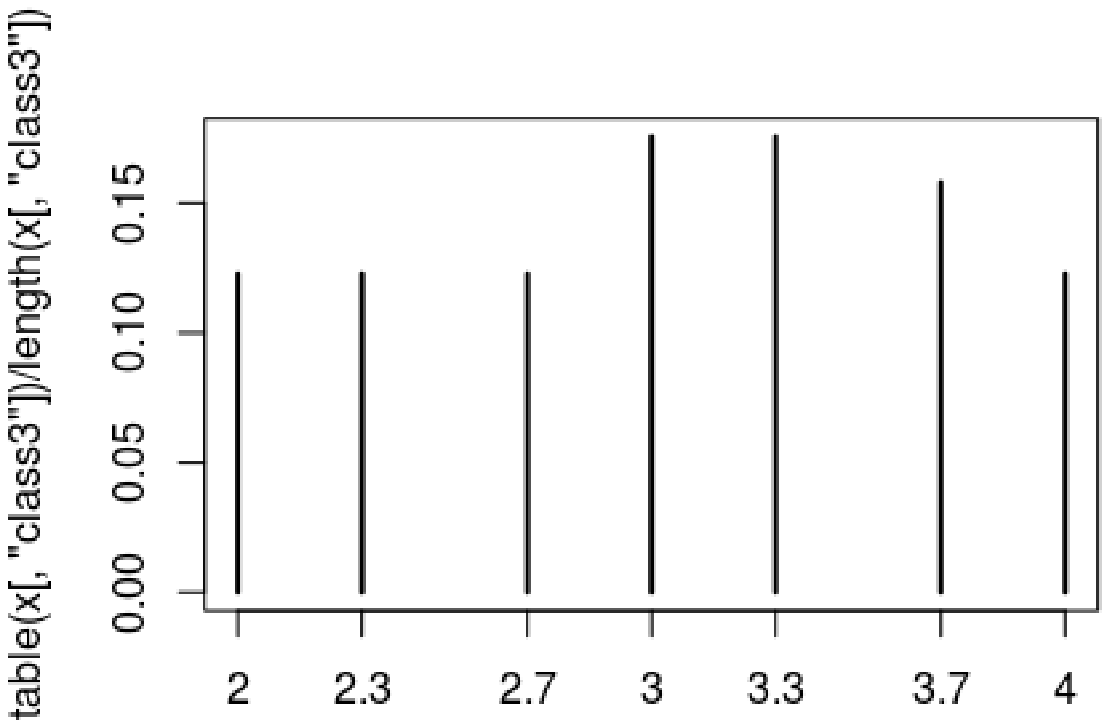
```
    # comment this line out if testing problem 4 by

    # itself
    draw_pmf(results, p)
    # show the graph
    plt.show()


# Problem 1
#ex1()

# Problem 2
#dieroll(1)
#dieroll(5)
#dieroll(10)
#dieroll(100)

# Problem 3
#print pi_Estimate(1000000)
# It requires numPoints on the order of 10^6 to get

# pi=3.14 consistantly

# Problem 4 (must comment out a line in order for t
his
# to work without the PMF values)
#samples = 10000
#plot_coin_flips(samples, .2)
#plot_coin_flips(samples, .5)
#plot_coin_flips(samples, .8)
# These graphs do look like how I expect them to lo
ok
# since the probability of the first coin being hea
ds
# is the same as the weight of the coin and that th
e
# probability that it would take more than 1 coin t
o
# get a heads consistnatly decreases exponentially

# Problem 5
# k is the number of trials and p is probability
```

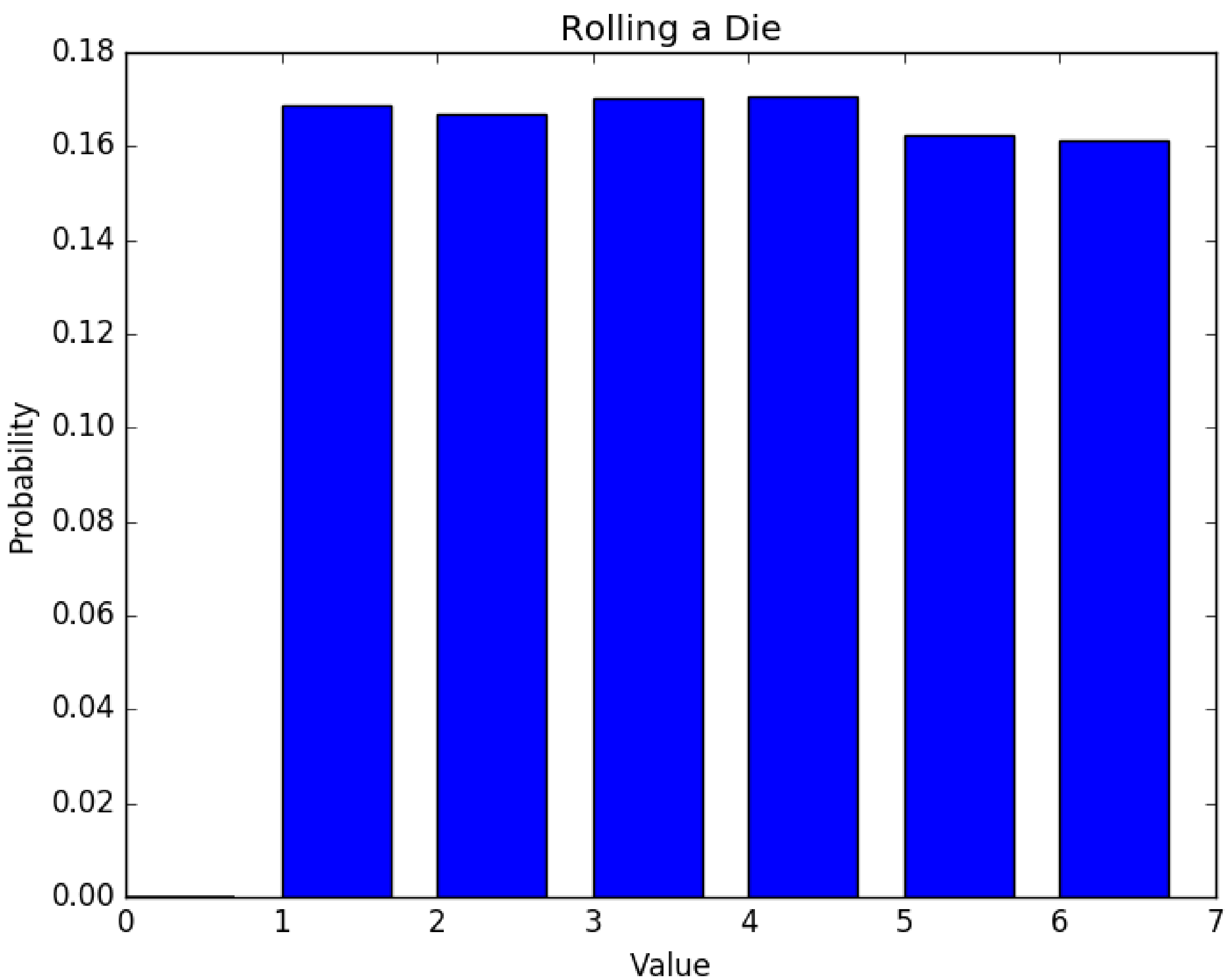
```
# for k=1 --> probability = p (1 heads)
# for k=2 --> probability = p*(1-p) (1 heads and 1
tails)
# for k=3 --> probability = p*(1-p)*(1-p) (1 heads
and two tails)
# for k=4 --> probability = p*(1-p)^3 (1 heads and
three tails)

# General formula based off of these cases:
# Pr(X=k) for all k in Range(X) = p*(p-1)^(k-1)

# Problem 6
#m = 10000
#plot_coin_flips(m, .2)
#plot_coin_flips(m, .5)
#plot_coin_flips(m, .8)



# Data analysis with R
# Keyan Vakil

# Problem 7
grades <- read.csv('/tmp/grades.csv')
View(grades)
# Among the first 10 students in each class,
# Class 1 has 1 student with a 4.0
# Class 2 has 1 student with a 4.0
# Class 3 has 1 student with a 4.0
#
# Problem 8
summary(grades)
# The summary function shows an easier way to get a
 sense of the
# distribution of grades for each class
# Min. is the minimum grade (0th percentile grade)
# 1st Qu. is the 25th percentile grade
# Median is the 50th percentile grade
# Mean is the average grade
# 3rd Qu. is the 75th percentile grade
# Max is the maximum grade (100th percentile grade)
```

```
#
# Problem 9
sd(grades[,'class1'])
sd(grades[,'class2'])
sd(grades[,'class3'])
#
# Problem 10
boxplot(grades)
# The horizontal line above the lower vertical dot
ted line represents the 1st Qu. (the 25th highest
value of all values)
# The bold horizontal line in the middle represent
s the Median (the middle value of all values)
# The horizontal line below the upper vertical dot
ted line represents the 3rd Qu. (the 75th highest
value of all values)
#
# Problem 11
l1 <- length(grades[,'class1'])
l1
l2 <- length(grades[,'class2'])
l2
l3 <- length(grades[,'class3'])
l3

t1 <- table(grades[,'class1'])/l1
t1
t2 <- table(grades[,'class2'])/l2
t2
t3 <- table(grades[,'class3'])/l3
t3

plot(t1)
plot(t2)
plot(t3)
#
```

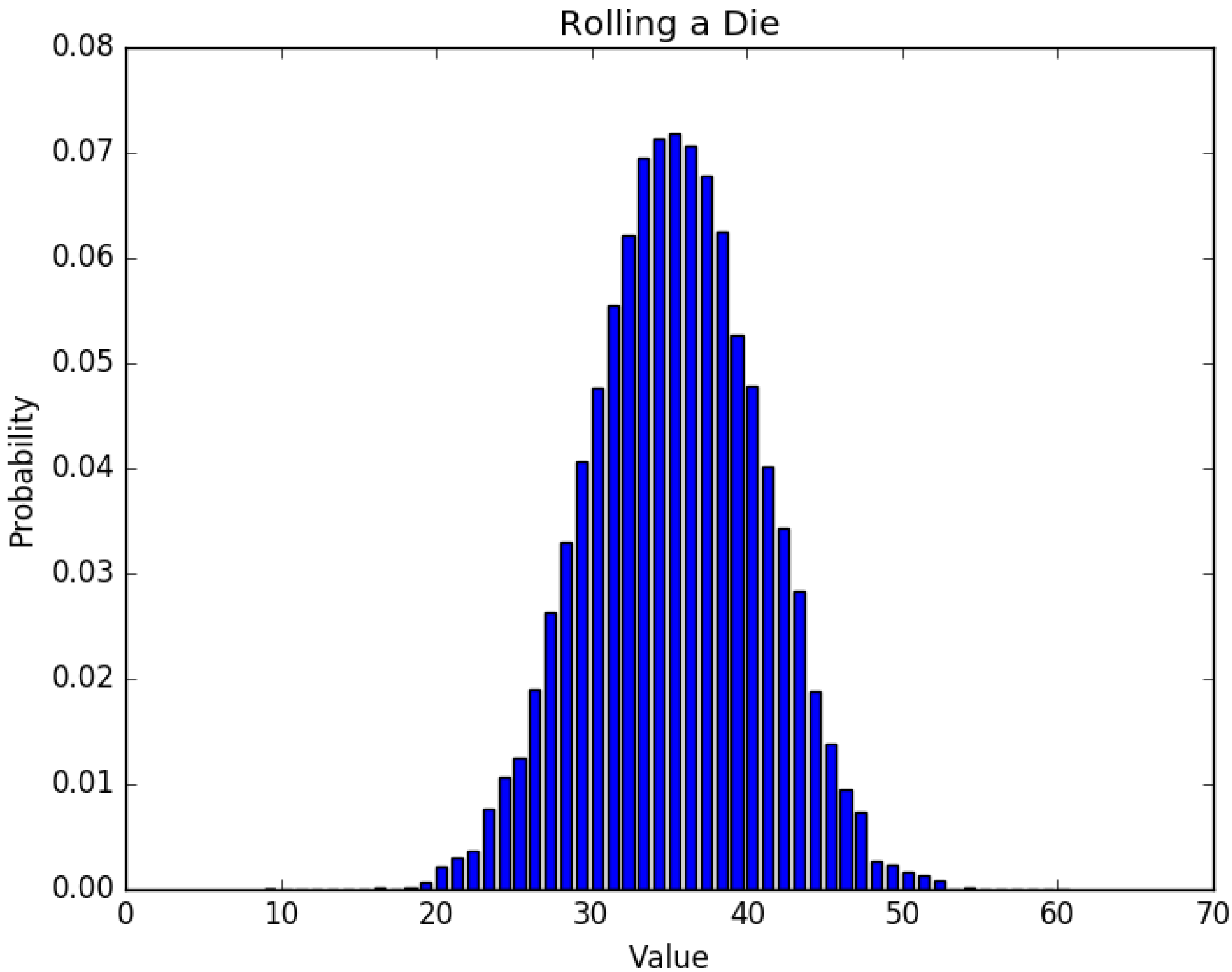

table(x[, "class1"])/length(x[, "class1"])
0.00
0.10
0.20
2.3
2.7
3
3.3
3.7
4

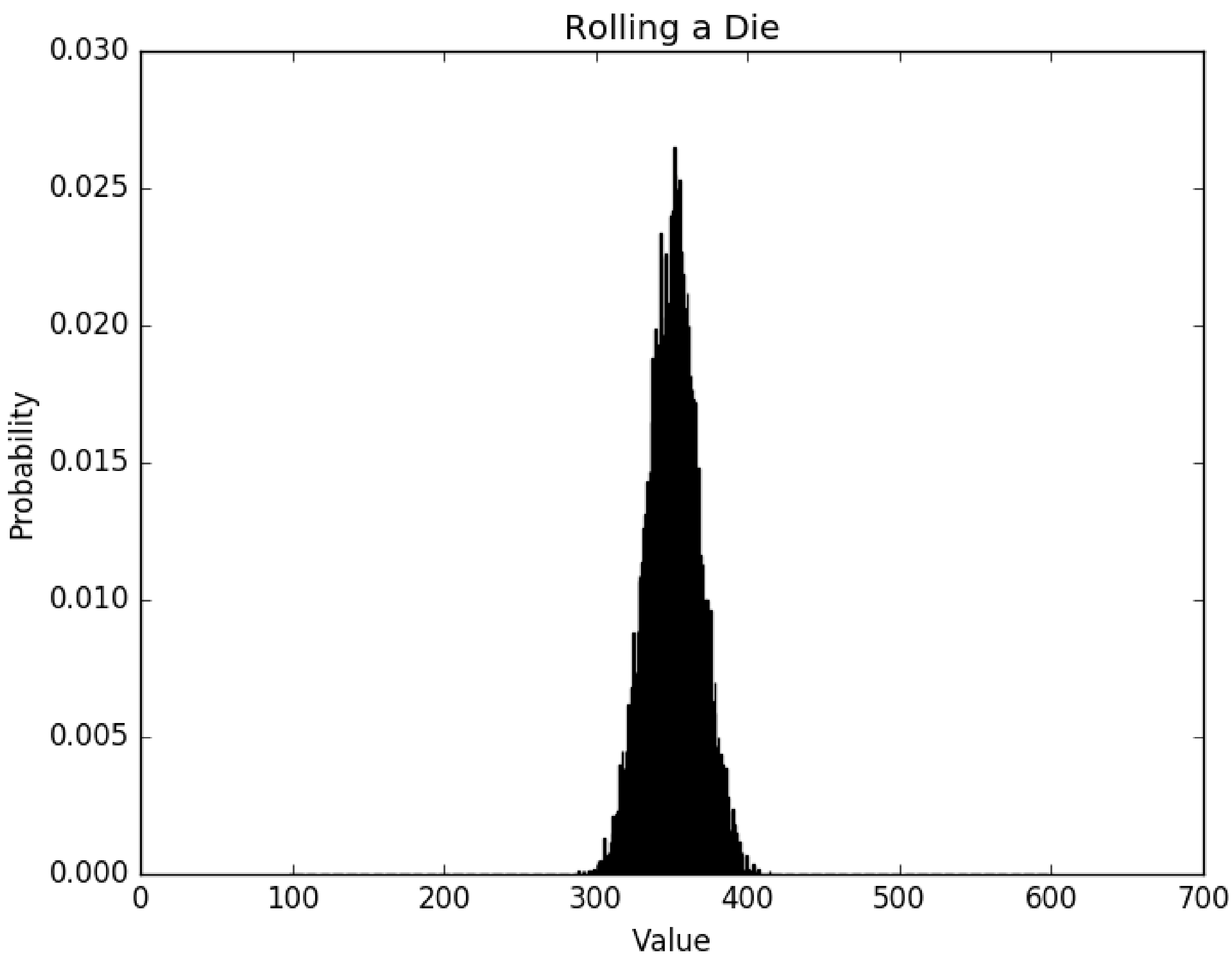

table(x[, "class2"])/length(x[, "class2"])
0.00
0.10
0.20
0.30
2
2.3
2.7
3
3.3
3.7
4


table(x[, "class3"])/length(x[, "class3"])
0.00
0.05
0.10
0.15
2
2.3
2.7
3
3.3
3.7
4

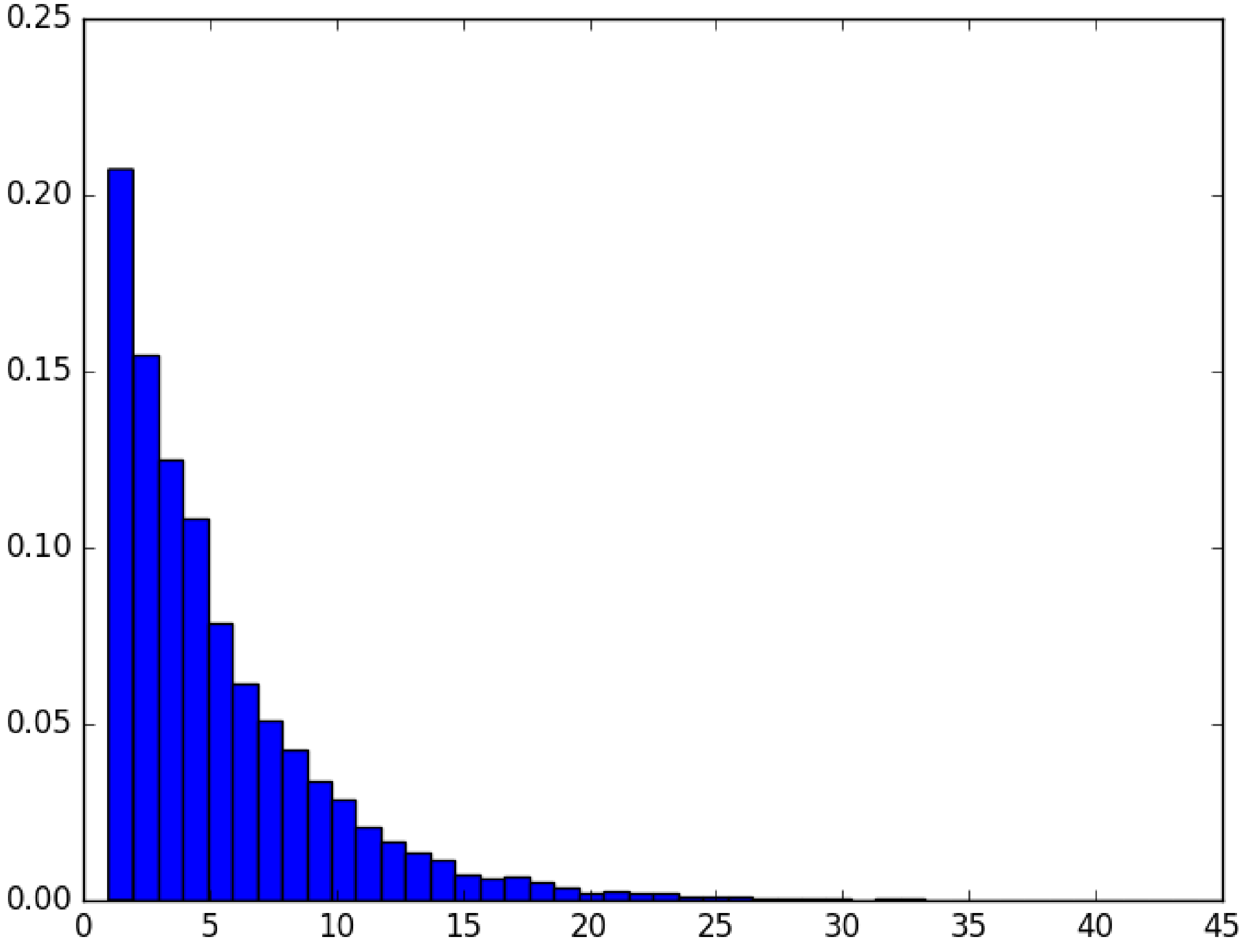


Rolling a Die
Probability
Value
0.00
0.02
0.04
0.06
0.08
0.10
0.12
0.14
0.16
0.18
0
1
2
3
4
5
6
7

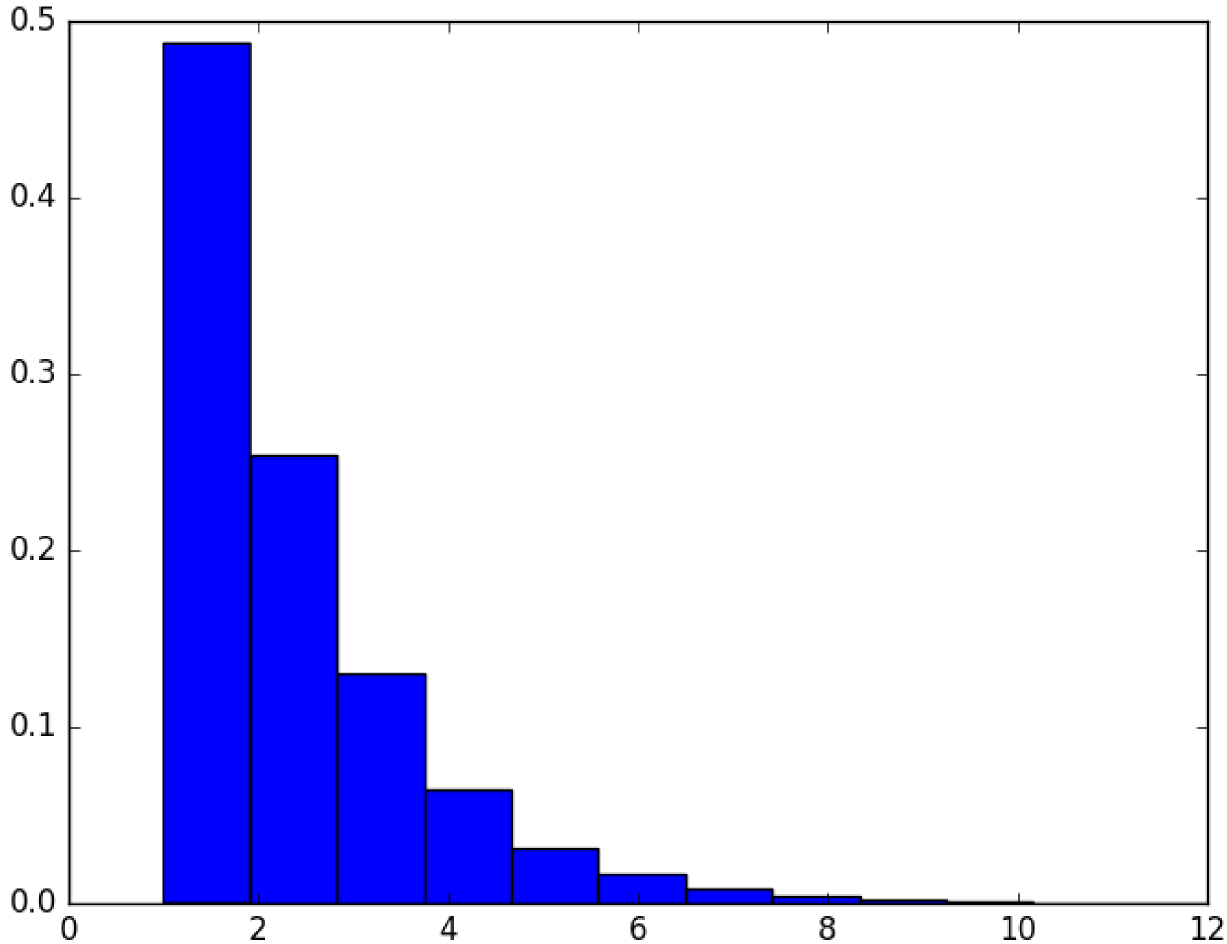

Rolling a Die
Probability
Value
0.00
0.01
0.02
0.03
0.04
0.05
0.06
0.07
0.08
0
10
20
30
40
50
60
70

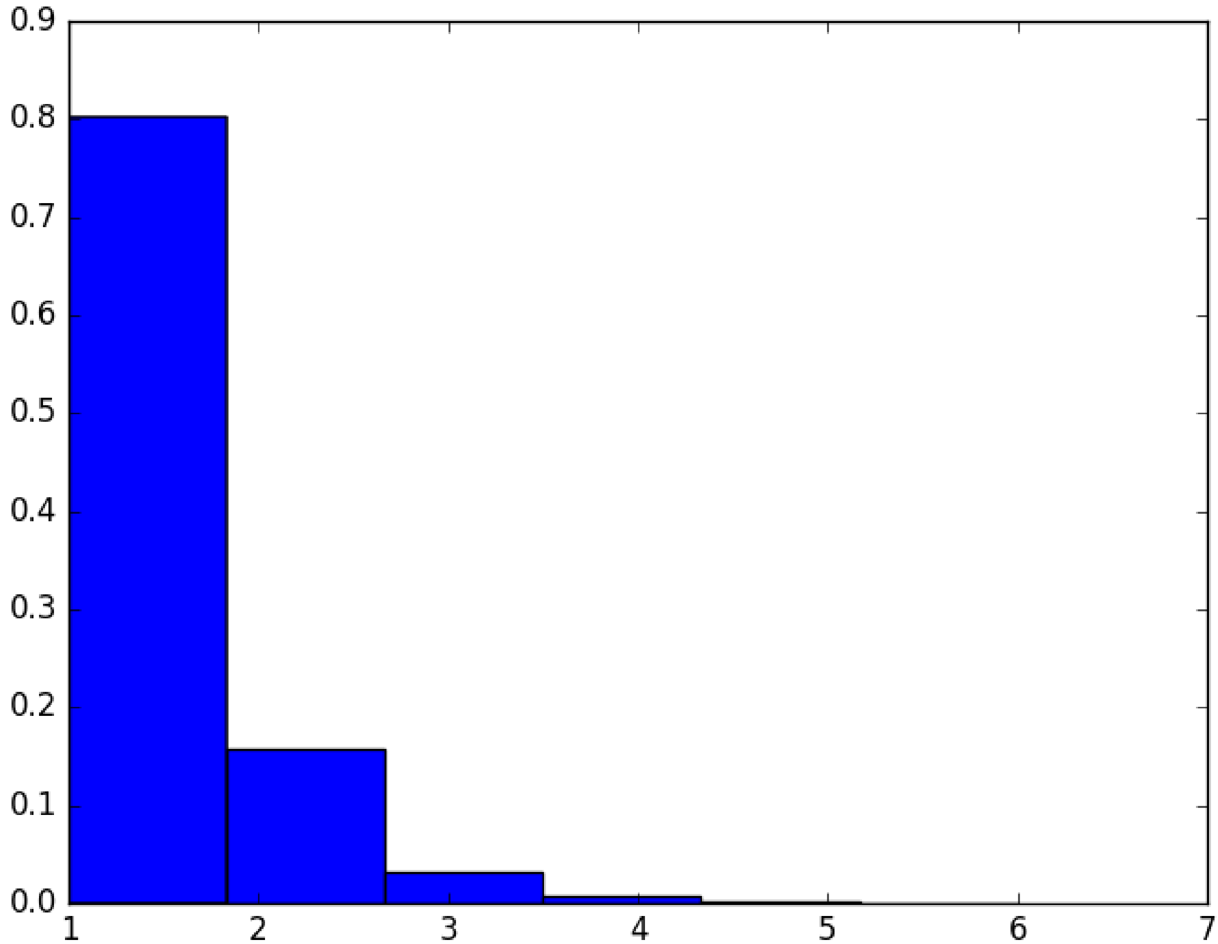


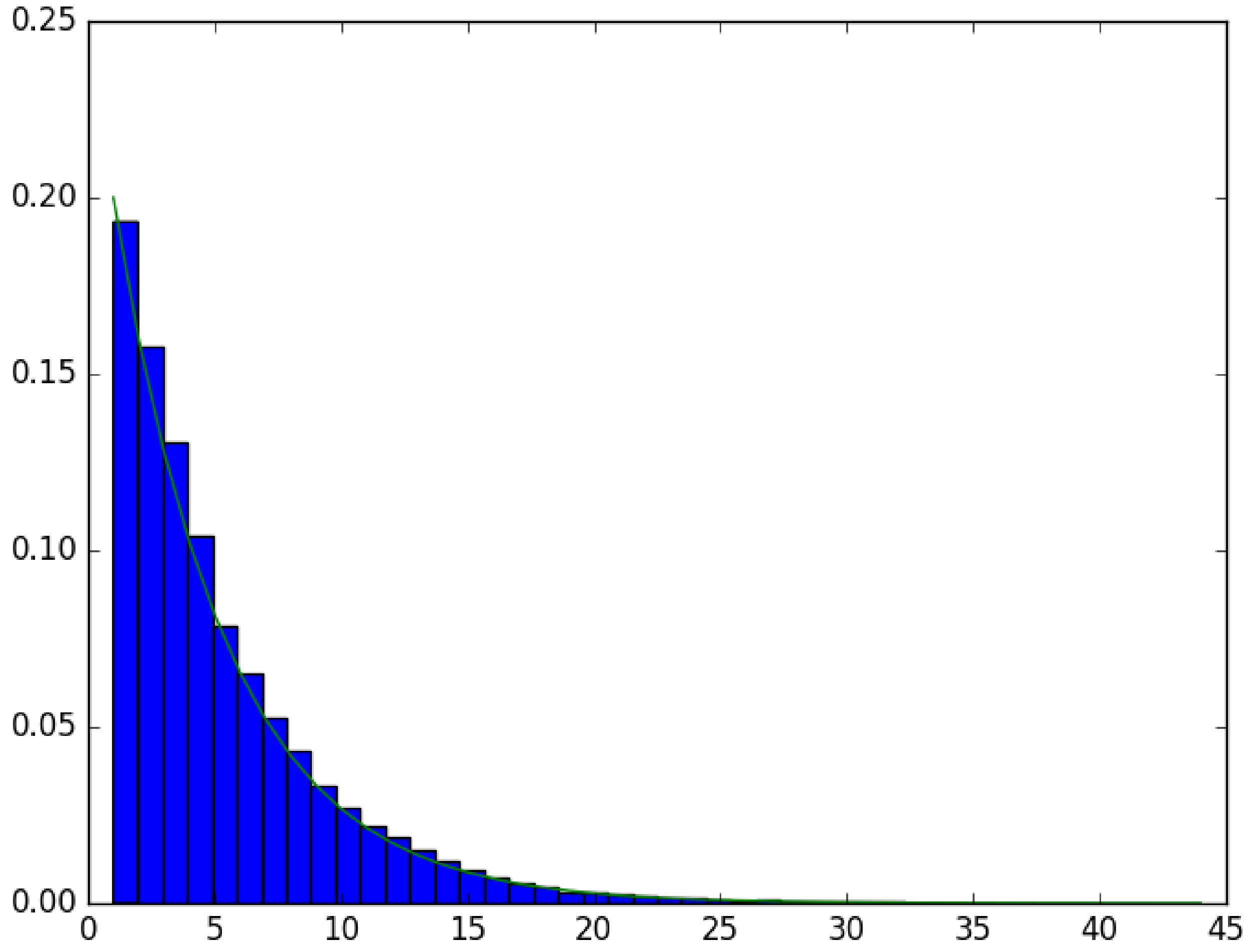
Rolling a Die
Probability
Value
0.000
0.005
0.010
0.015
0.020
0.025
0.030
0
100
200
300
400
500
600
700

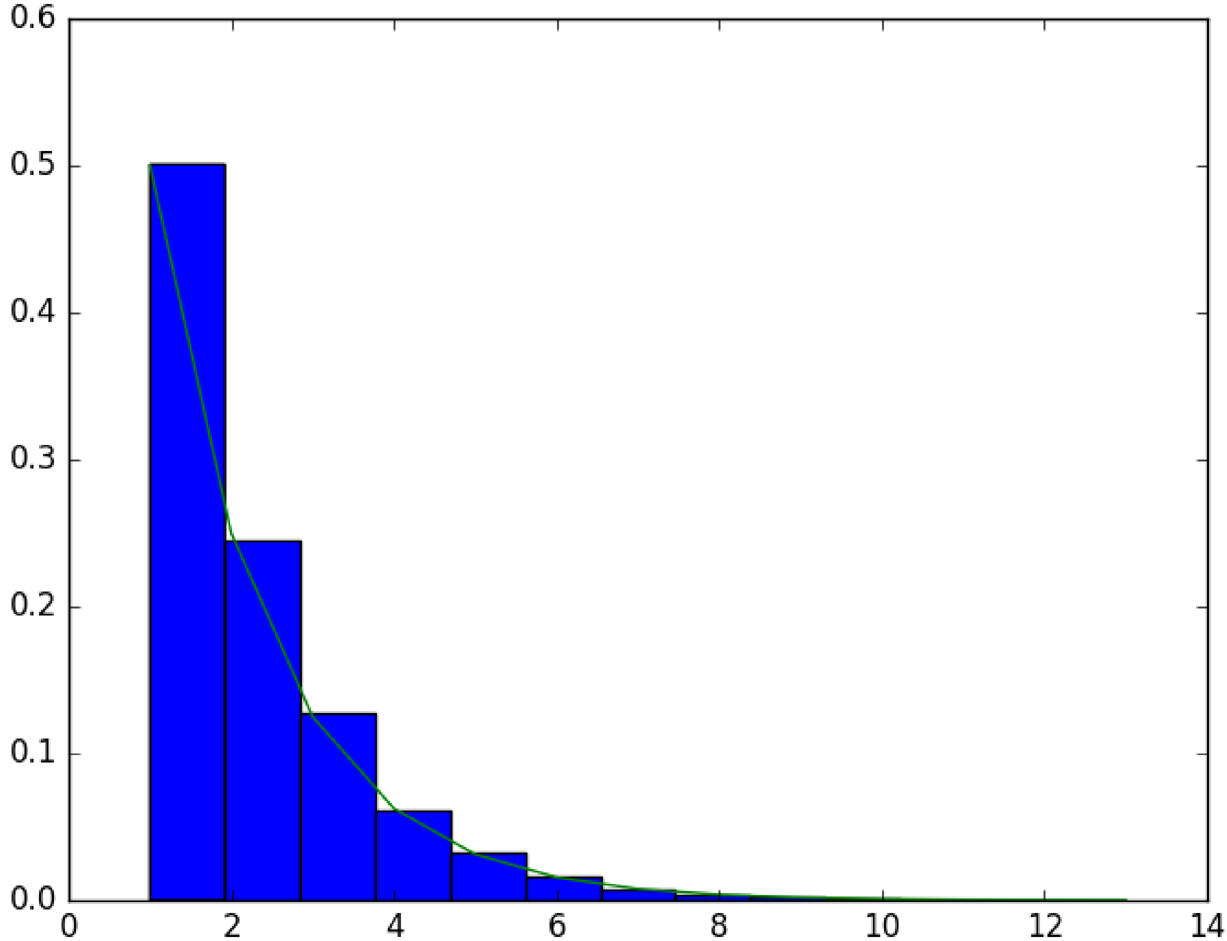


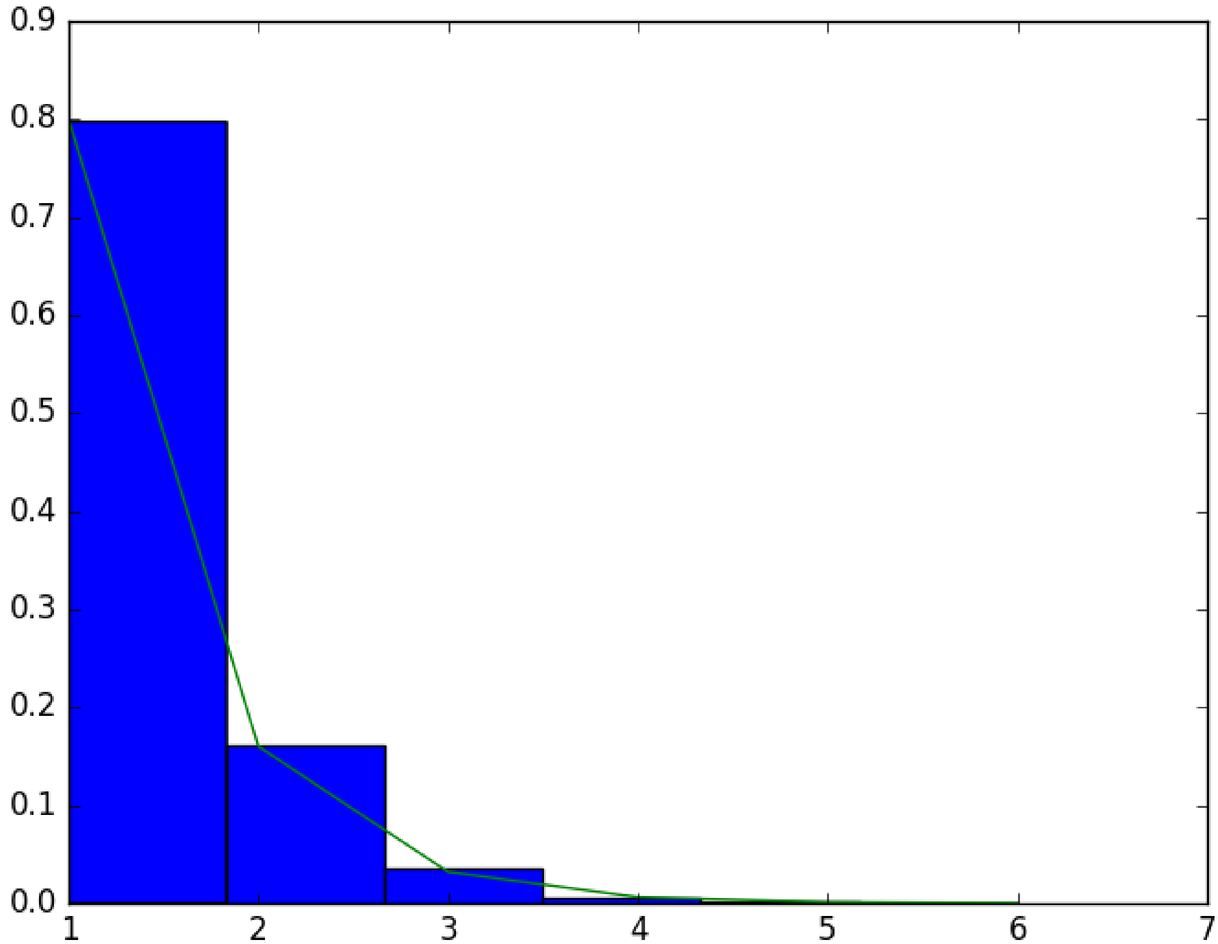
0.25
0.20
0.15
0.10
0.05
0.00
0
5
10
15
20
25
30
35
40
45


0.5
0.4
0.3
0.2
0.1
0.0
0
2
4
6
8
10
12

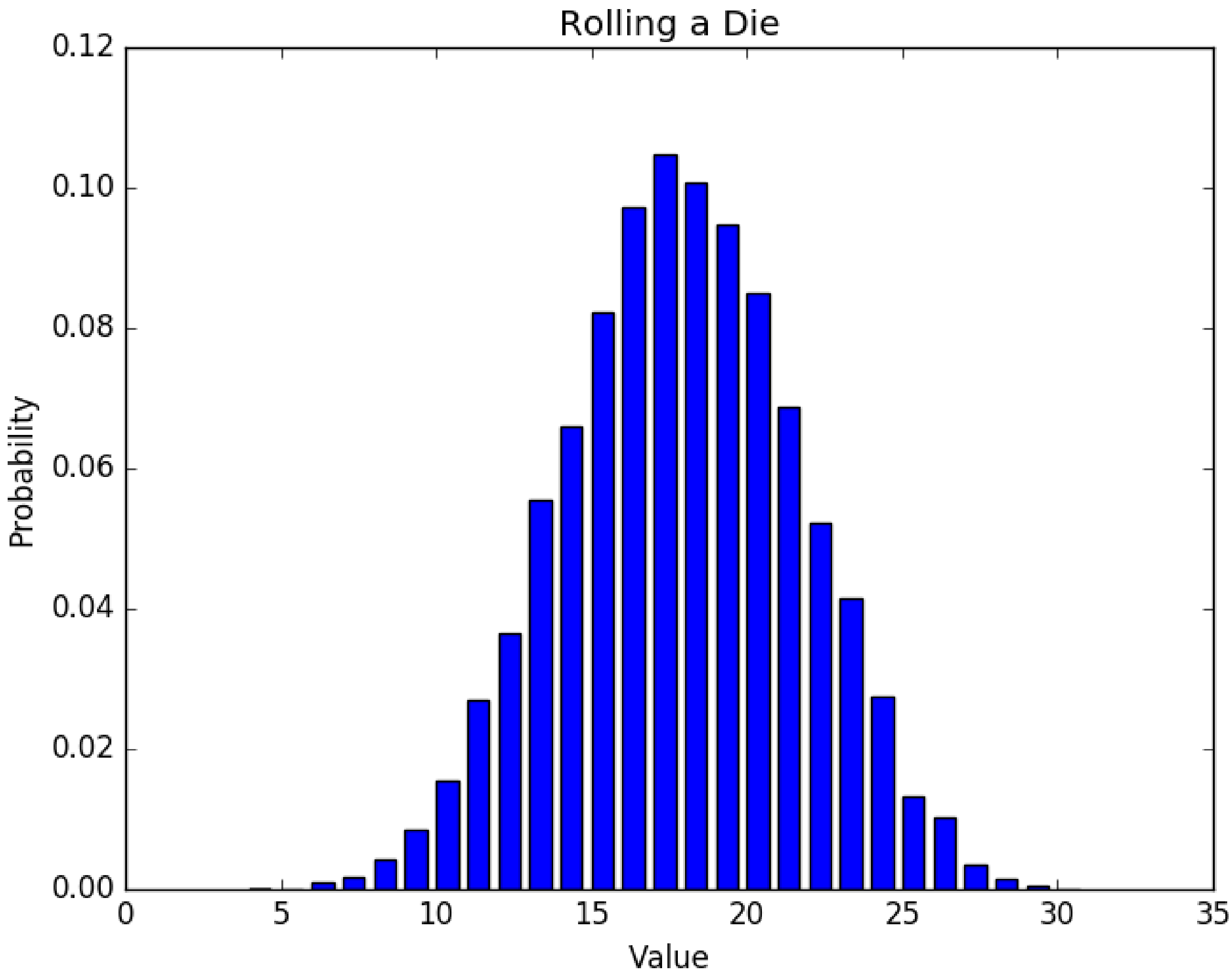

0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
1
2
3
4
5
6
7


0.25
0.20
0.15
0.10
0.05
0.00
0
5
10
15
20
25
30
35
40
45

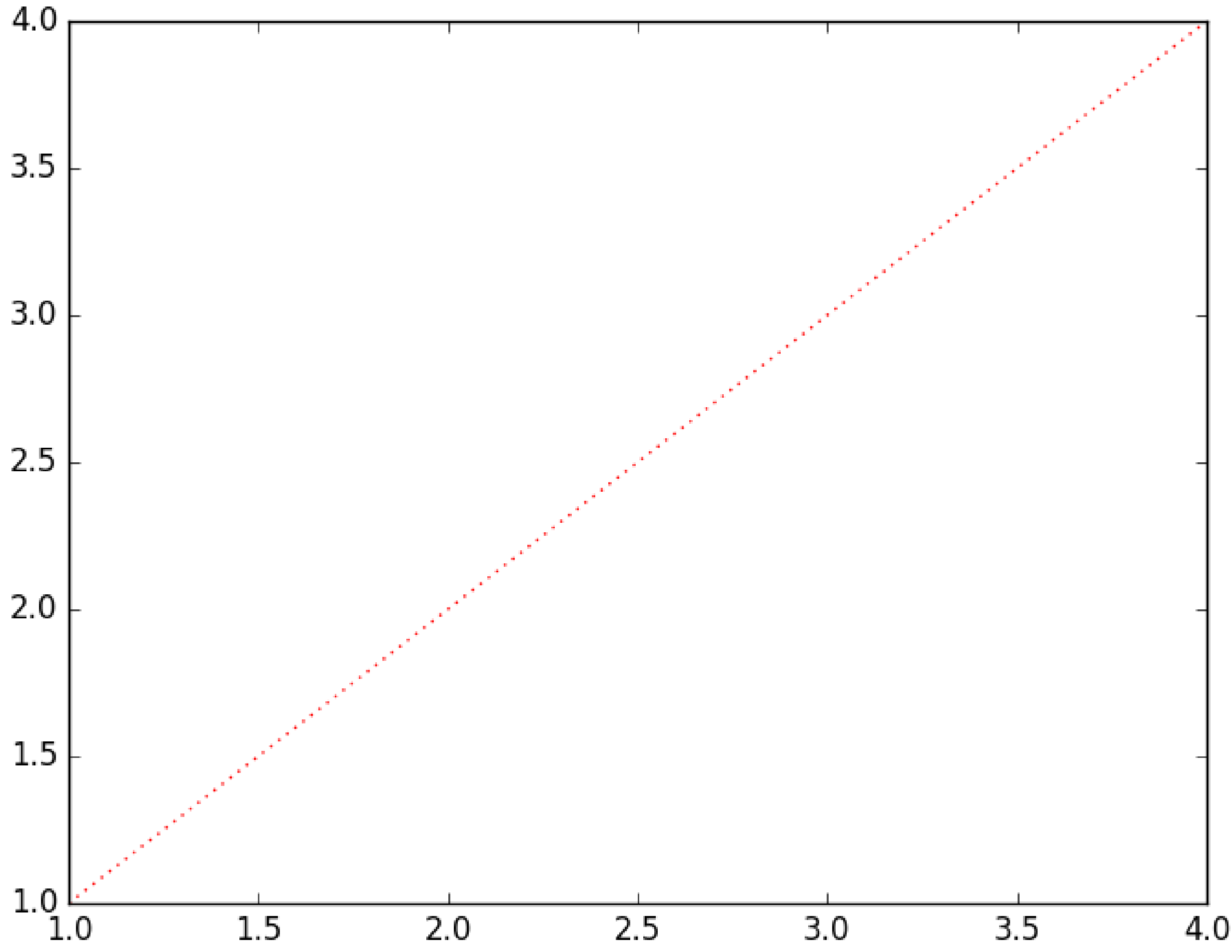

0.6
0.5
0.4
0.3
0.2
0.1
0.0
0
2
4
6
8
10
12
14

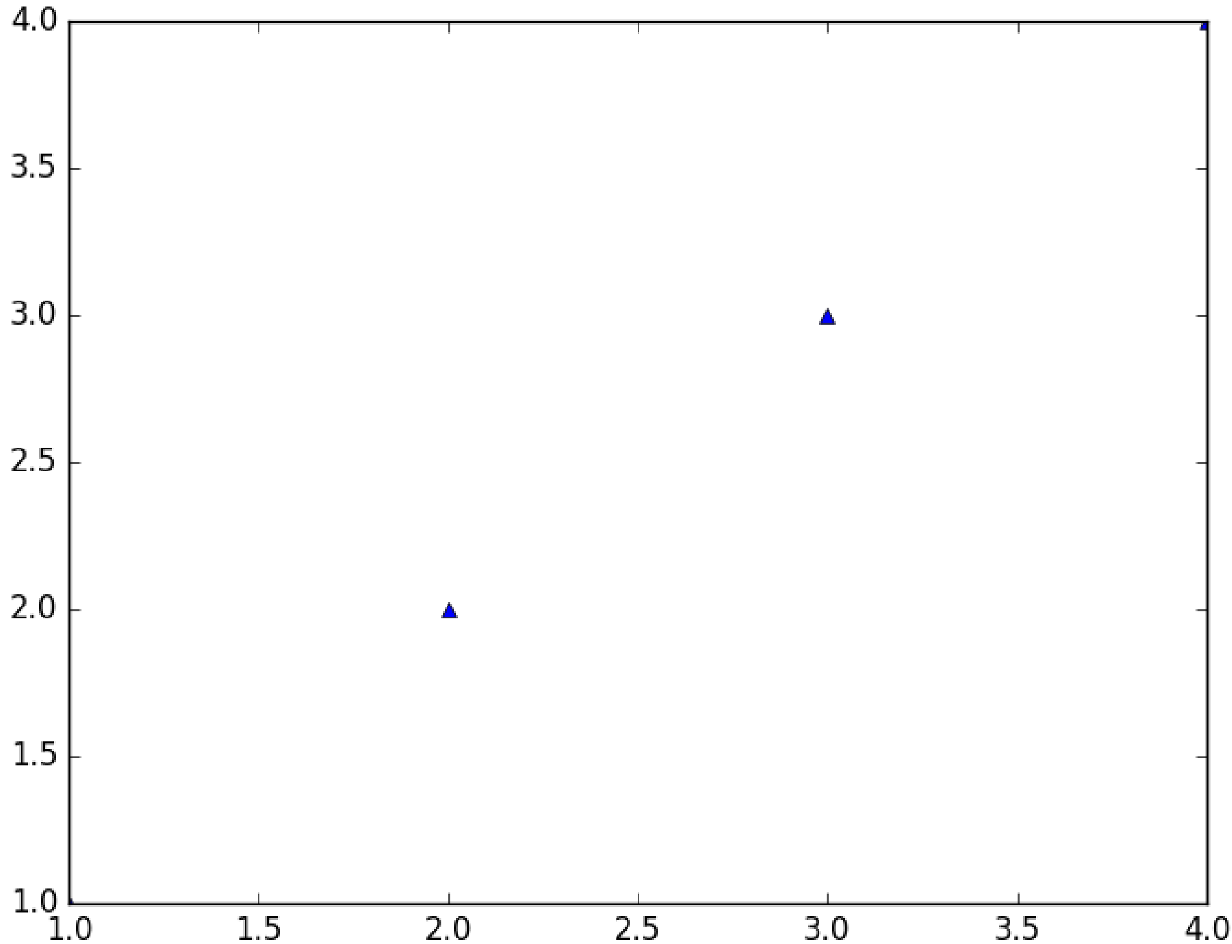


0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
1
2
3
4
5
6
7



Rolling a Die
Probability
Value
0.12
0.10
0.08
0.06
0.04
0.02
0.00
0
5
10
15
20
25
30
35


4.0
3.5
3.0
2.5
2.0
1.5
1.0
1.0
1.5
2.0
2.5
3.0
3.5
4.0



4.0
3.5
3.0
2.5
2.0
1.5
1.0
1.0
1.5
2.0
2.5
3.0
3.5
4.0