Coding Assignment 1

Question1:

```python
import matplotlib.pyplot as plt
import numpy as np
import math
from scipy.special import gamma
import matplotlib.ticker as mticker

x=[i for i in range(1,10**6)]
# for computational ease we use gamma function to plot the value of the factorial
y=[math.log(gamma(i)) for i in x]
z= [math.log(math.sqrt(2*np.pi*i))+i* (math.log(i)-math.log(np.e))for i in x]
fig,ax=plt.subplots(1)
ax.plot(x,y,color='r',label="Actual value of factorial")
ax.plot(x,z,color='b',label="Stirling's Approximation")
# Rotating the x labels at an angle 45degree to prevent overlapping
for tick in ax.get_xticklabels():
    tick.set_rotation(45)
f = mticker.ScalarFormatter(useOffset=False, useMathText=True)
g = lambda x, pos: "${}$".format(f._formatSciNotation('%0.5e' % x))
# for formatting the x labels and y labels correctly
plt.gca().xaxis.set_major_formatter(mticker.FuncFormatter(g))
plt.gca().yaxis.set_major_formatter(mticker.FuncFormatter(g))
ax.legend()
plt.show()
```

- We use the formula Γ(n)=n! to plot the values of the factorial in the plot.
- We have used python's SciPy module to simulate that.
- Also to further ease computation we take log on both sides of the formula

  n! ≈ $\sqrt{2n\pi}(\frac{n}{e})^n$

Question 2:

```
import random
import matplotlib.pyplot as plt
import numpy as np


class Dice:
    def __init__(self, num_sides=6): # the deafult value of the dice is 6
        self.num_sides = num_sides
        self.dist_values = [] # variable to print the distributin
        self.prob_dist = [] # list for holdng the discrete distribution
        if isinstance(self.num_sides, str) == True:
            try:
                raise Exception('Cannot construct dice')
            except Exception as inst:
                print(type(inst))
                print(inst)
            exit()

        elif self.num_sides <= 3 or isinstance(self.num_sides, int) == False:
            try:
                raise Exception('Cannot construct dice')
            except Exception as inst:
                print(type(inst))
                print(inst)
            exit()
```

t

- The class **Dice** is created which is specified by the number of sides and by default it has 6 sides.
- Few checks are made and exceptions are raised if the no of faces are not correctly specified

```
for i in range(self.num_sides):
    self.prob_dist.append(1 / self.num_sides)
self.dist_values = "{" + ", ".join([str(x) for x in self.prob_dist]) + "}"
```

- The above snapshot captures the code which generates a uniform distribution for the Dice object if no distribution is specified.

```
def setProb(self, prob_values):
    self.prob_values = prob_values
    self.prob_dist.clear()
    # Assigning user provided distribution
    for i in range(self.num_sides):
        self.prob_dist.append(prob_values[i])
    # checking the Assigned is a valid probability distribution
    for i in self.prob_dist:
        if i < 0 or round(sum(self.prob_dist),10)!=1.0:
            print(sum(self.prob_dist))
```

- The **setProb** method sets a valid probability distribution to the dice object

```
labels = [x + 1 for x in range(self.num_sides)]
# A dictionary is created to sample the random disitribution
prob_dict = {i : 0 for i in labels}

CDF_list=[0]
for i in range(len(self.prob_dist)):
    CDF_list.append(CDF_list[i]+self.prob_dist[i])
# Sampling the random distribution
for i in range(self.iter):
    r=random.random()
    for i in range(len(CDF_list)):
        if (CDF_list[i]<r<CDF_list[i+1]):
            prob_dict[i+1]=prob_dict[i+1]+1 # suming up the no of
# Generating the final distribution
random_dist=[prob_dict[i]/iter for i in prob_dict.keys()]
```

- The above code snippet in the **Roll** method simulates n throws of a dice
- **Prob_dict** is a dictionary whose keys are the faces of the dice and the values correspond to the no of time different faces appear in n throws

```
fig, ax = plt.subplots()
x = np.arange(len(labels))
width = 0.15
ax.set_xticks(x)
ax.set_xticklabels(labels)
# random disribution
ax.bar(x - width / 2, [self.iter * i for i in random_dist], width, label='Expected', color='r')
# actual distribution
ax.bar(x + width / 2, [self.iter * i for i in self.prob_dist], width, label='Actual', color='b')
ax.set_title('Outcome of {0} throws of a {1}-faced dice'.format(self.iter, self.num_sides), fontsize=12)
ax.set_ylabel('Sides')
ax.set_xlabel("Outcomes")
ax.legend(bbox_to_anchor=(1.05, 1.0), loc='upper left')
plt.tight_layout()
plt.show()
```

- Finally, we plot the values using matplotlib with the actual expected value vs the expected value of corresponding to the random distribution
- Along x axis we plot the faces of the dice
- Along y axis we plot the no of time different faces occur.

Question 3:

```python
for i in range(INTERVAL):

    rand_x = random.uniform(-1, 1) # uniformly choosing the x-cordinate

    rand_y = random.uniform(-1, 1) # uniformly choosing the y-coordinate

    # checking whether it lies inside the circle or the square
    origin_dist = rand_x ** 2 + rand_y ** 2
    if origin_dist <= 1:
        circle_points += 1
    square_points += 1
    pi.append(4 * circle_points / square_points)
```

- The above code in the function **estimatePi** simulates the method to estimate pi
- We generate points in the Euclidean plane and then check whether they are inside the unit circle and not and accordingly label the points
- In the end we use the formula to estimate the value of π.

Question 4:

```python
    def assimilateText(self,file_name):
        self.file_name=file_name
        my_file = open(self.file_name,"r+")
        content=my_file.read()
        content_split = content.split() # split() is used to split the text into indivdual words

        # A dictionary is created with each two tuples as keys and the corresponding values
        # are intialized as an empty list
        self.prefix_dict={(content_split[i-1],content_split[i]):[] for i in range(1,len(content_split))}

        # The values are poulated in the dictionary according to the keys
        for i in range(1, len(content_split) - 1):
            self.prefix_dict[(content_split[i - 1], content_split[i])].append(content_split[i + 1])
```

- For the function **assimilateText** we simply read the file and then split the original text into individual words and create a prefix dictionary whose keys are all possible adjacent two tuples and the corresponding values contain all possible words followed by that tuple.

```python
def generateText(self,no_of_words,word=None):
    self.no_of_words=no_of_words
    self.word= word # the optional argument for the method

    word_list=[] # The output text is stored in this list

    s=[] # A dummy list of two entries to act as key for the dictionary

    if isinstance(word,str)==False:
      word_list=list(random.choice(list(self.prefix_dict.keys())))
```

- In **generateText** we use **word_list** to generate the desired text and **S** work s as an iterative key to the prefix dictionary to generate the random words using random.choice().
- If no word is specified word_list is populated with a random key from the prefix dictionary.

```python
           for key in self.prefix_dict.keys():
               if (key[0]==self.word):
                   s.append(key)
           word_list=list(random.choice(s))

    if len(self.prefix_dict[tuple(word_list)]) >0:
      s=word_list

    while len(word_list) <=self.no_of_words:
        s=[word_list[len(word_list)-2],word_list[len(word_list)-1]]
        if (len(self.prefix_dict[tuple(s)]))==0:
            break
        word_list.append(random.choice(self.prefix_dict[tuple(s)]))
        s.clear()
    print(" ".join(word_list))
```

- If a word is specified, we use the key in the prefix dictionary to find a match and then begin our process to generate words randomly.
- We use a while loop to check whether the word limit is reached. And inside the while loop **S** is cleared after each step to accommodate a new key which is obtained by taking the last two elements of the **word_list** and then a random word is appended to the word_list by choosing a word randomly which comes after the tuple.