Lab 1 : Probability Theory

1. Sampling from uniform distribution
2. Sampling from Gaussian distribution
3. Sampling from categorical distribution through uniform distribution
4. Central limit theoram
5. Law of large number
6. Area and circumference of a circle using sampling
7. Fun Problem

There are missing fields in the code that you need to fill to get the results but note that you can write you own code to obtain the results

1.Sampling from uniform distribution

a) Generate N points from a uniform distribution range from [0 1]

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

N =  10#  Number of points (Example = 10)
X =  np.random.uniform(size=10)#  Generate N points from a uniform
distribution range from [0 1] # Ref :
https://numpy.org/doc/stable/reference/random/generated/numpy.random.u
niform.html
print("Points from uniform distribution : ", X)
```

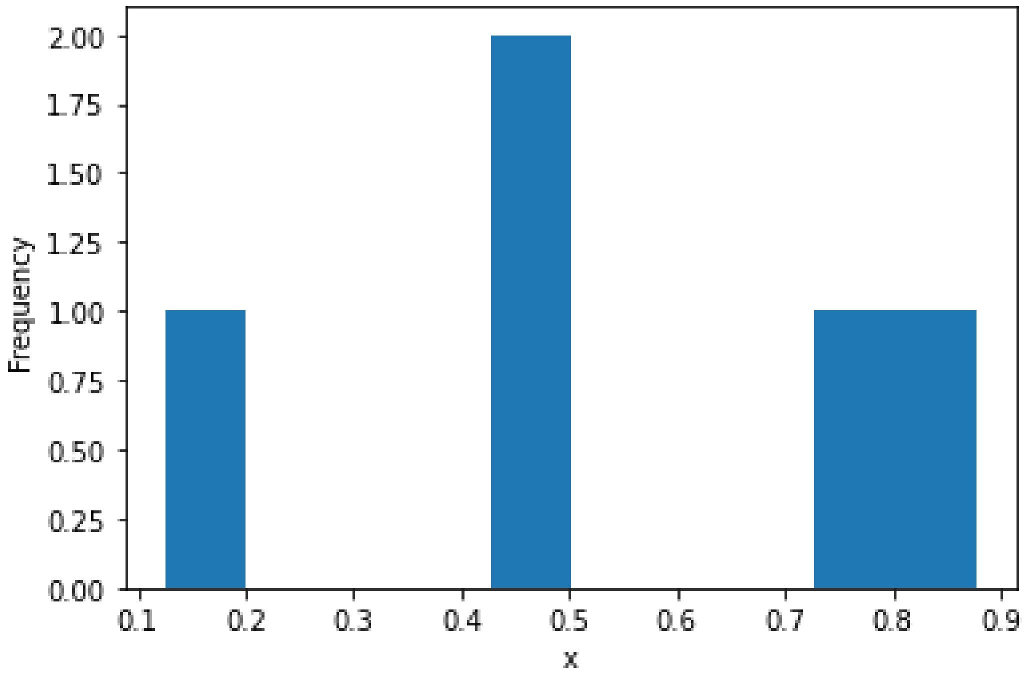
```
Points from uniform distribution :  [0.29574436 0.2771389  0.75322191
0.56023364 0.0472284  0.27030269
 0.17381453 0.60796437 0.89323234 0.03606803]
```

b) Show with respect to no. of sample, how the sampled distribution converges to parent distribution.

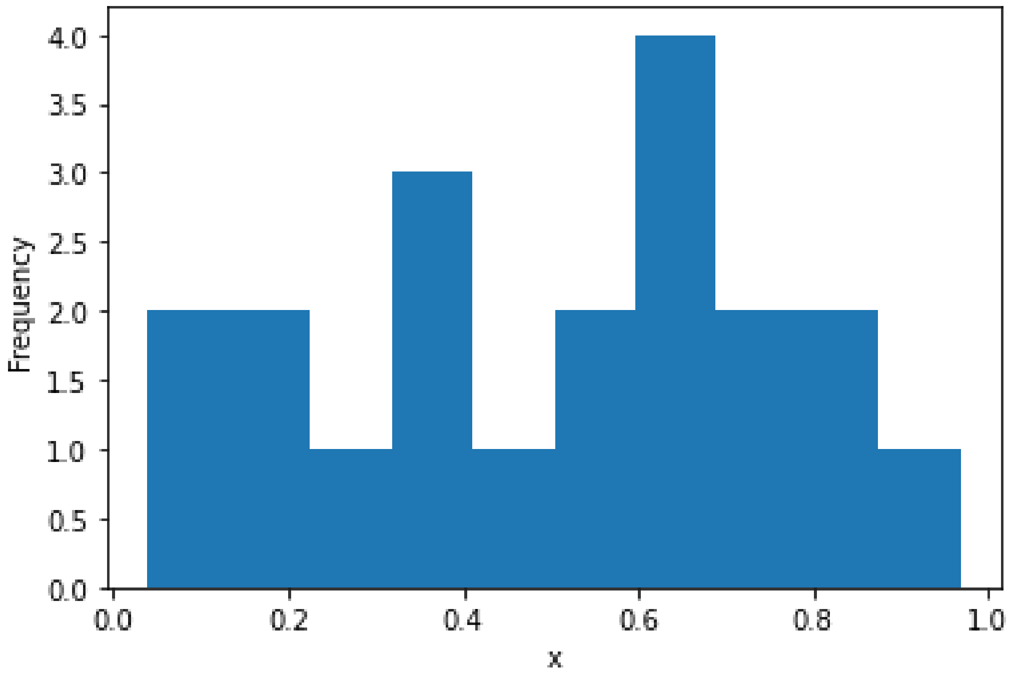
```
arr =  np.array([5, 20, 100, 500, 1000, 2000])# Create a numpy array
of different values of no. of samples # Ref :
https://numpy.org/doc/stable/reference/generated/numpy.array.html

for i in arr:
  x =  np.random.uniform(size=i)# Generate i points from a uniform
distribution range from [0 1]
  print("Number of elements in array :", i)
  plt.hist(x)
  plt.xlabel("x")
  plt.ylabel("Frequency")
  plt.show()
  # write the code to plot the histogram of the samples for all values
```

```
in arr # Ref :
https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.hist.html
```

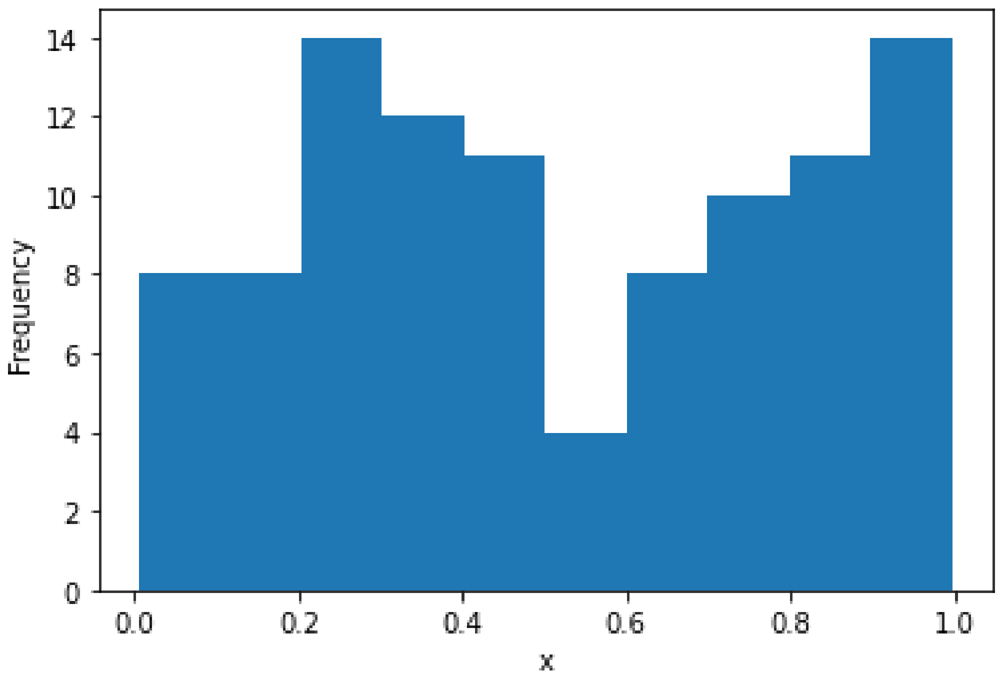
```
Number of elements in array : 5
```



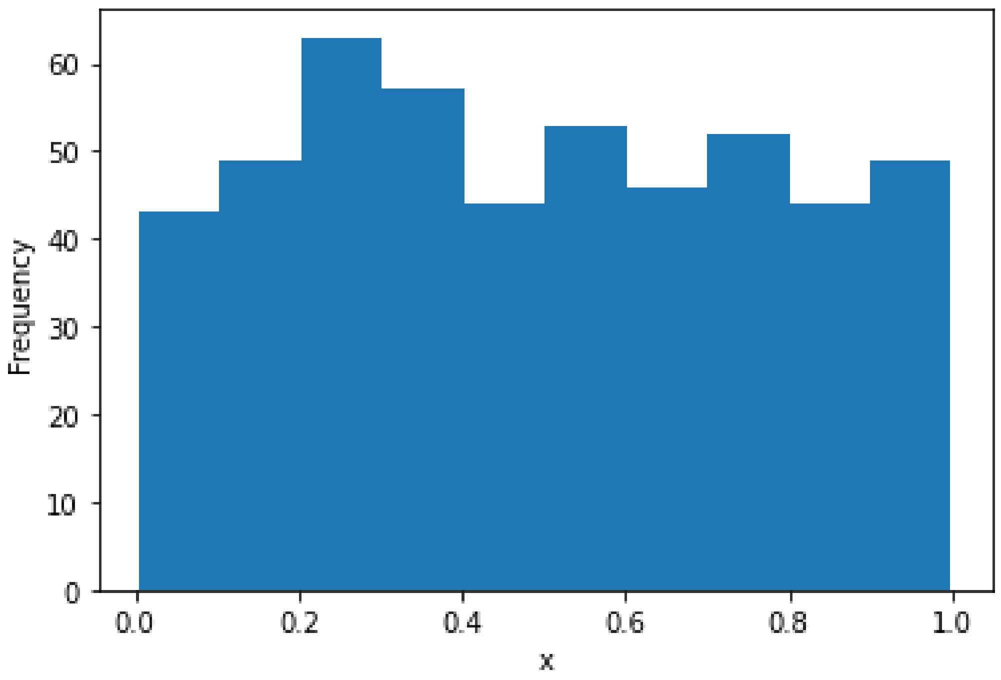
```
Number of elements in array : 20
```



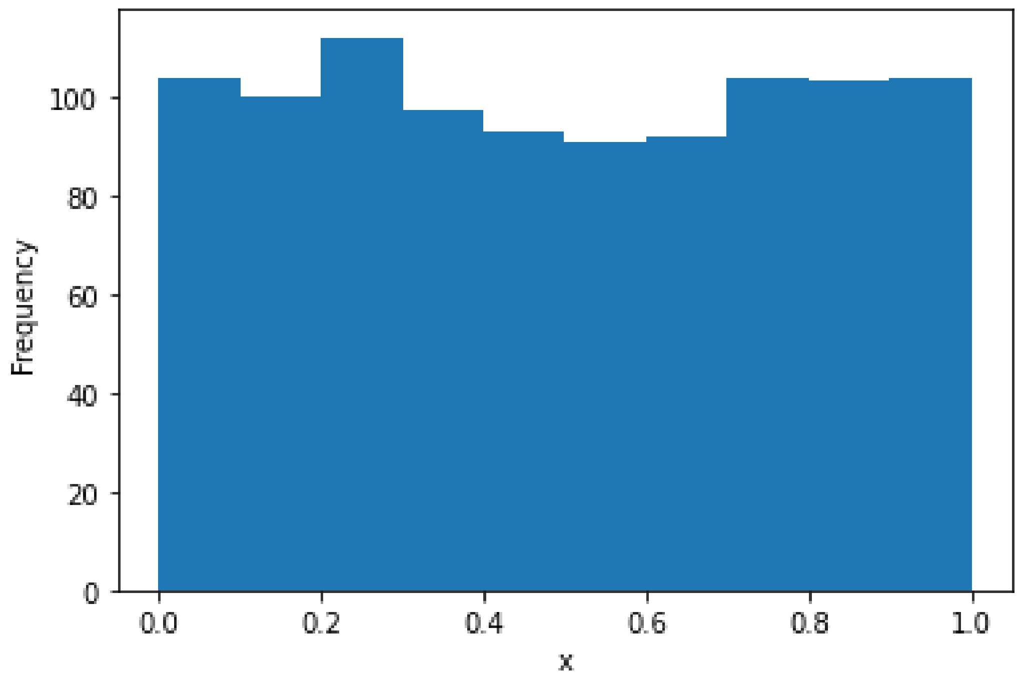
```
Number of elements in array : 100
```



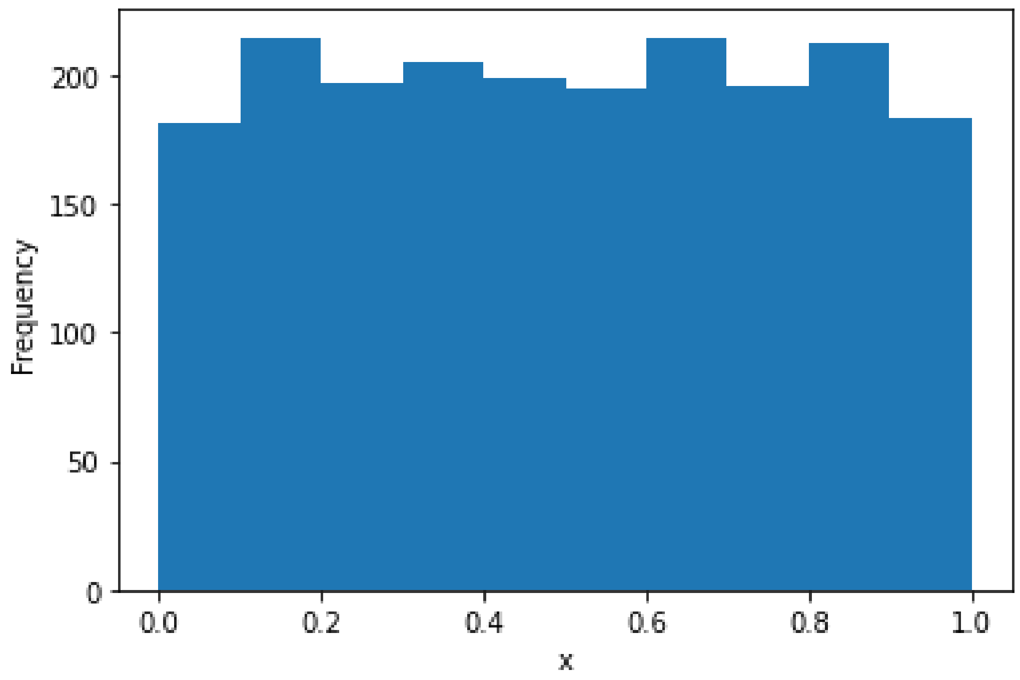
```
Number of elements in array : 500
```



```
Number of elements in array : 1000
```

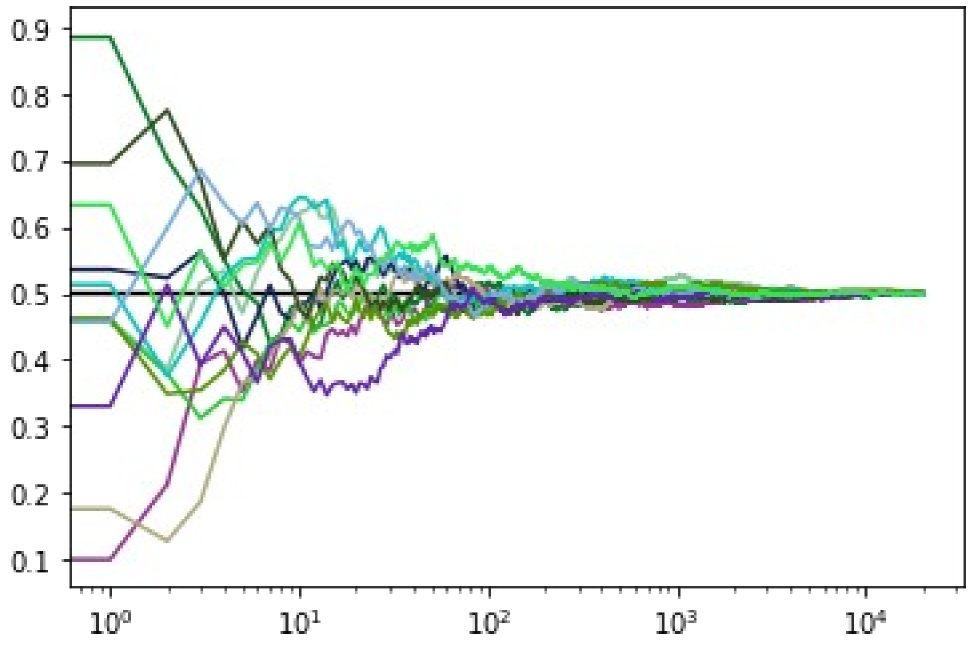


```
Number of elements in array : 2000
```



c) Law of large numbers: $average(x_{sampled})=\acute{x}$, where x is a uniform random variable of range [0,1], thus $\acute{x}=\int_0^1 x f(x)\,dx=0.5$

```
N =  20000# Number of points (>10000)
k =  12# set a value for number of runs

## Below code plots the semilog scaled on x-axis where all the samples
are equal to the mean of distribution
m = 0.5 # mean of uniform distribution
m = np.tile(m,x.shape)
m

plt.semilogx(m,color='k') # Ref :
https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.semilogx.h
tml

for j in range(k):

  i =  np.arange(1, N+1) # Generate a list of numbers from (1,N) # Ref
: https://numpy.org/doc/stable/reference/generated/numpy.arange.html
  x =  np.random.uniform(size=N)#  Generate N points from a uniform
distribution range from [0 1]
  mean_sampled = np.cumsum(x)/(i) # Ref :
https://numpy.org/doc/stable/reference/generated/numpy.cumsum.html

  ## Write code to plot semilog scaled on x-axis of mean_sampled,
follow the above code of semilog for reference
  plt.semilogx(mean_sampled, color=np.random.rand(3,))
```

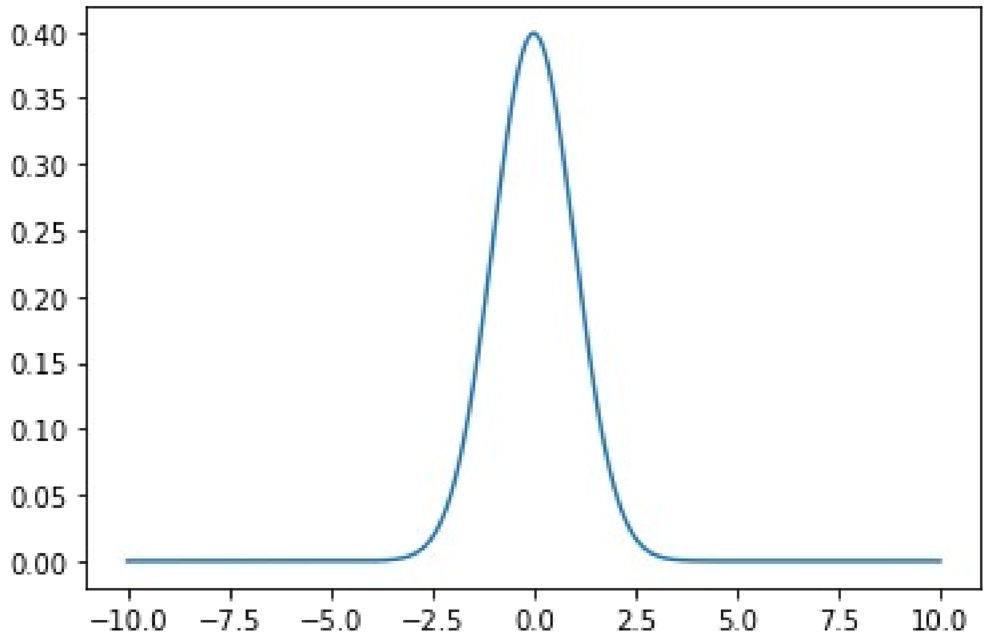


2. Sampling from Gaussian Distribution

a) Draw univariate Gaussian distribution (mean 0 and unit variance)

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

X =  np.linspace(start=-10, stop=10, num=1000)# Generate 1000 points
from -10 to 10 # Ref :
https://numpy.org/doc/stable/reference/generated/numpy.linspace.html

# Define mean and variance
mean = 0
variance = 1

exponent = 0.5 * (np.power(X-mean, 2))/variance
constant = 1/np.sqrt(2*np.pi)
gauss_distribution =  constant * np.exp(-exponent)# Define univariate
gaussian distribution (Hint : Probabilty Distribution Function of
normal distribution)

## Write code to plot the above distribution # Ref :
https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html
plt.plot(X, gauss_distribution)
```

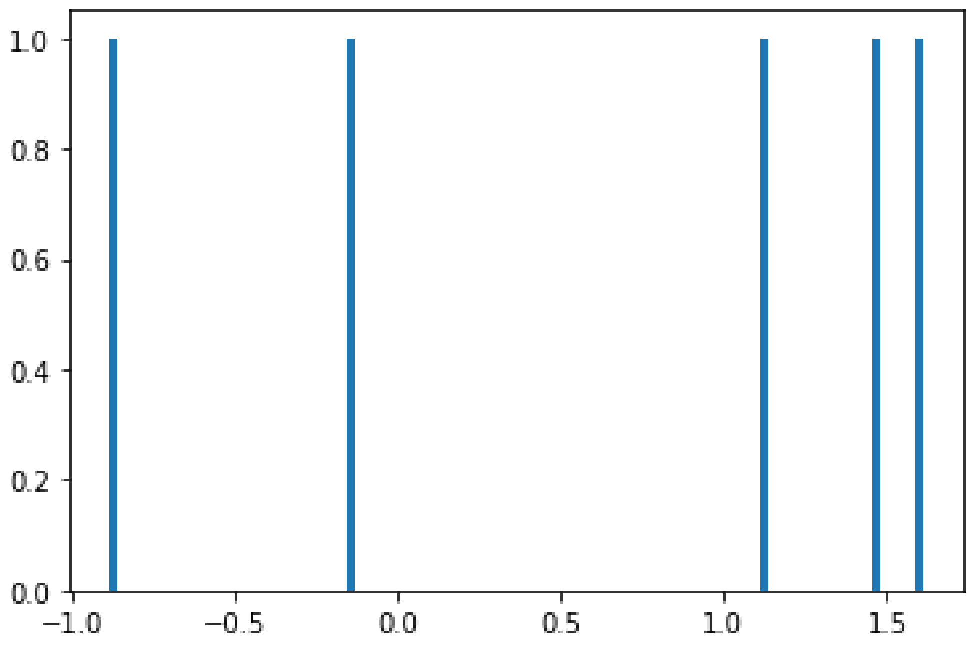
```
[<matplotlib.lines.Line2D at 0x26b9bec7e80>]
```



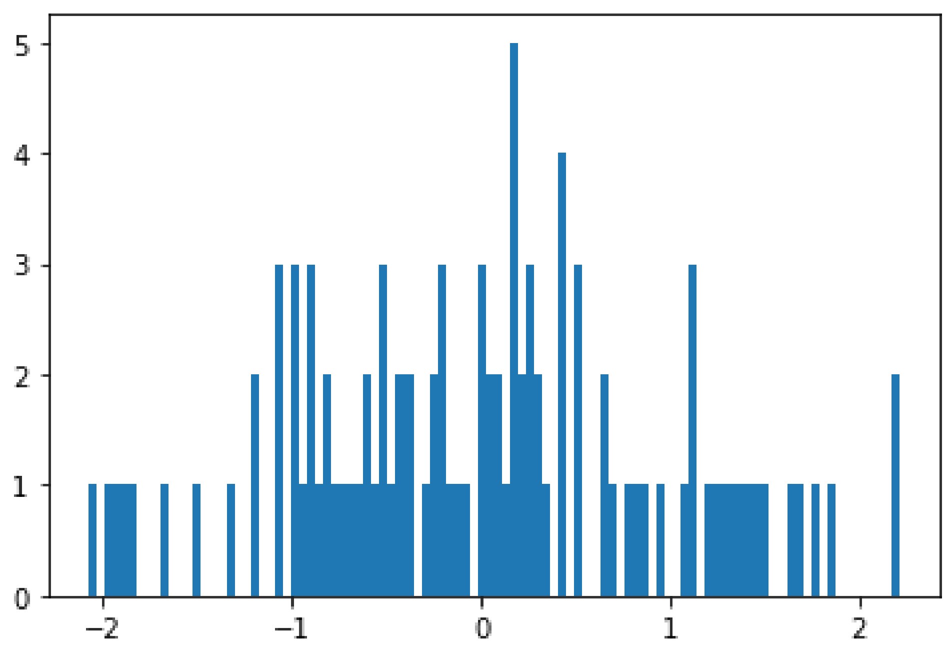
b) Sample from a univariate Gaussian distribution, observe the shape by changing the no. of sample drawn.

```
arr =  np.array([5, 100, 1000, 10000, 100000])# Create a numpy array
of differnt values of no. of samples and plot the histogram to show
the above

for i in arr:
  x_sampled =  np.random.normal(0, 1, size=i)# Generate i samples from
univariate gaussian distribution
  print(x_sampled.shape)
  plt.hist(x_sampled, bins=100)# write the code to plot the histogram
of the samples for all values in arr
  plt.show()
```

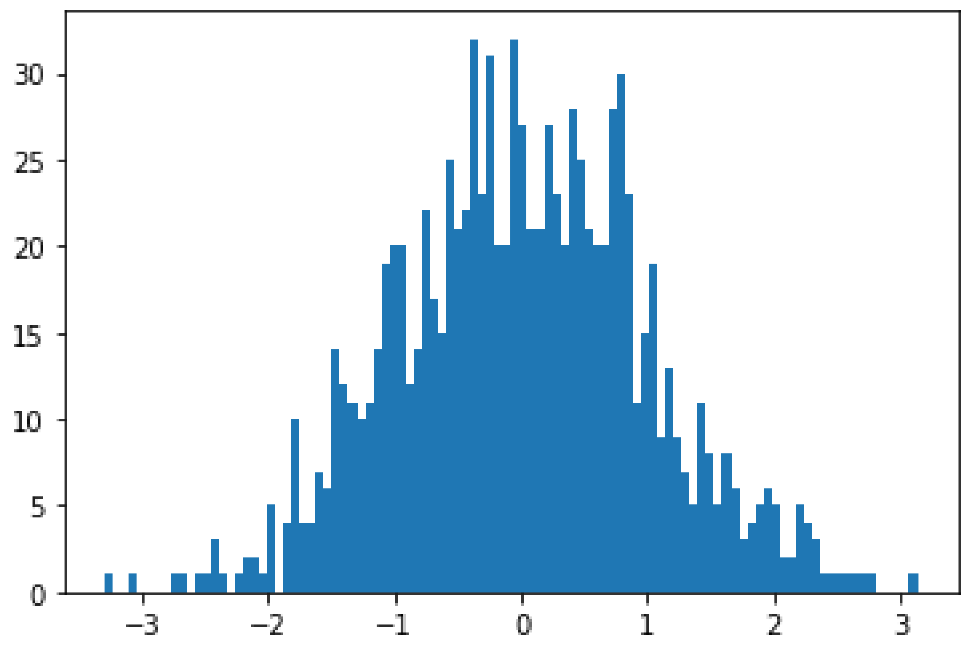
```
(5,)
```



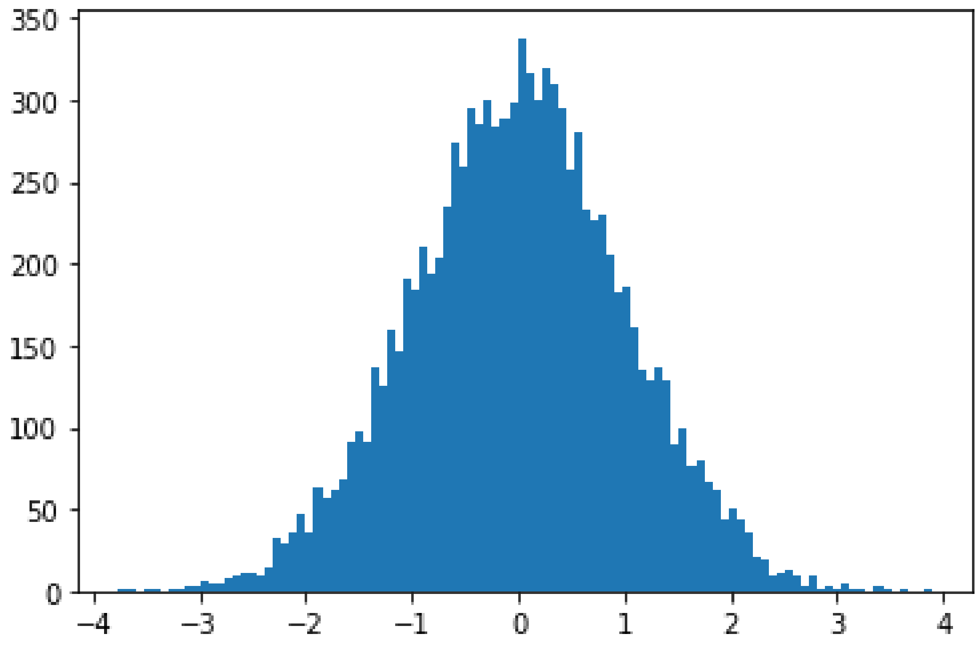
```
(100,)
```



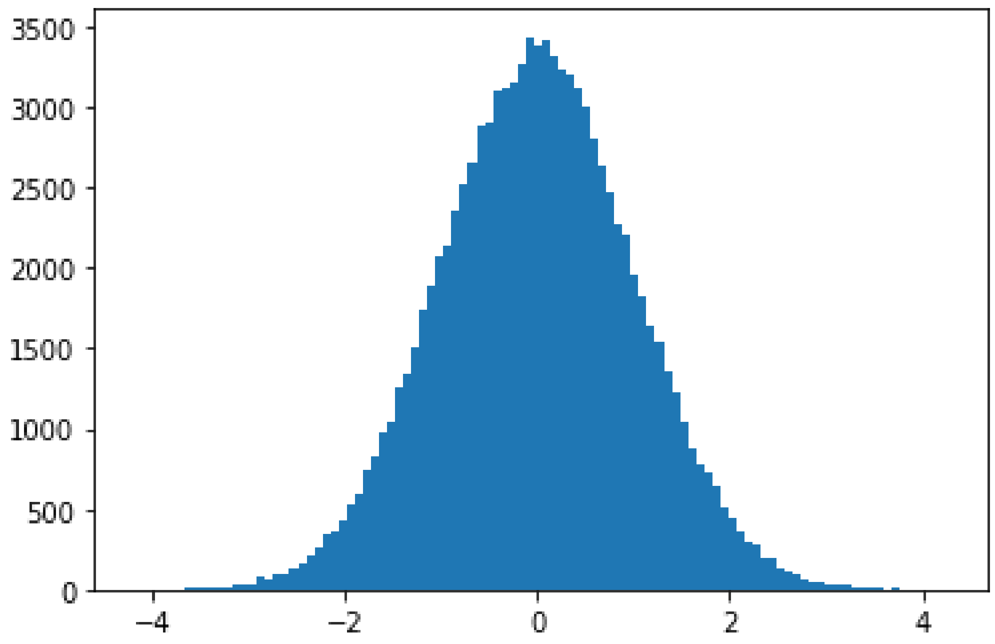
```
(1000,)
```



```
(10000,)
```



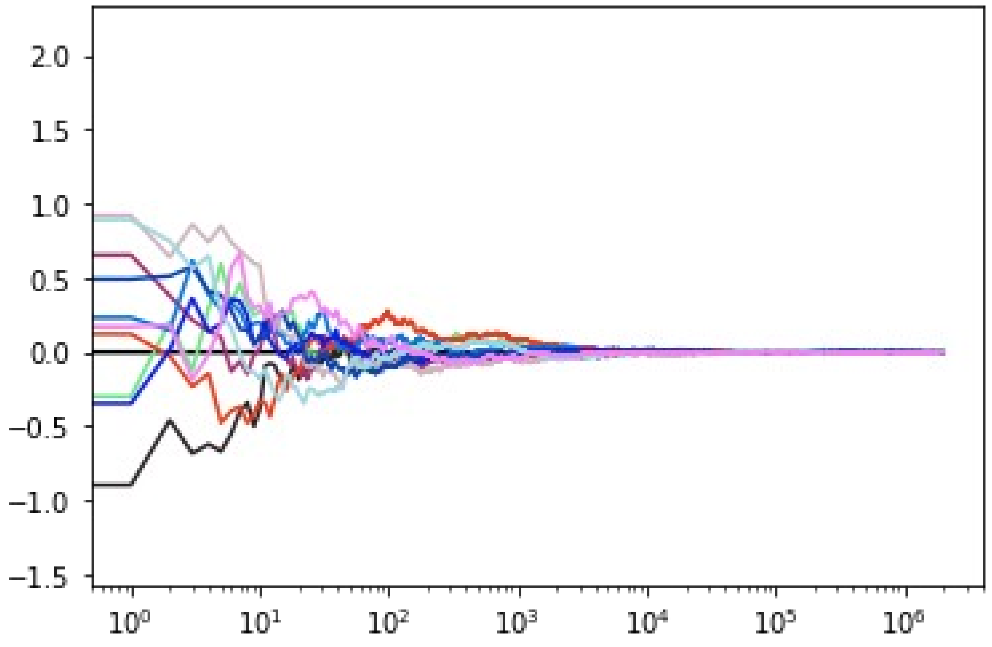
```
(100000,)
```



c) Law of large number

```
N = 2000000 #  Number of points (>1000000)
k = 10 # set a value for number of distributions

## Below code plots the semilog when all the samples are equal to the
mean of distribution
```

```
m = np.tile(mean,x.shape)
plt.semilogx(m,color='k')

for j in range(k):

  i = np.arange(1, N+1) # Generate a list of numbers from (1,N)
  x = np.random.normal(0, 1, N) # Generate N samples from univariate
gaussian distribution # Ref :
https://numpy.org/doc/stable/reference/random/generated/numpy.random.n
ormal.html
  mean_sampled = np.cumsum(x)/i # insert your code here (Hint : Repeat
the same steps as in the uniform distribution case)

  ## Write code to plot semilog scaled on x axis of mean_sampled,
follow the above code of semilog for reference
  plt.semilogx(mean_sampled, color=np.random.rand(3,))
```



3.Sampling of categorical from uniform

i) Generate n points from uniforms distribution range from [0 1] (Take large n)

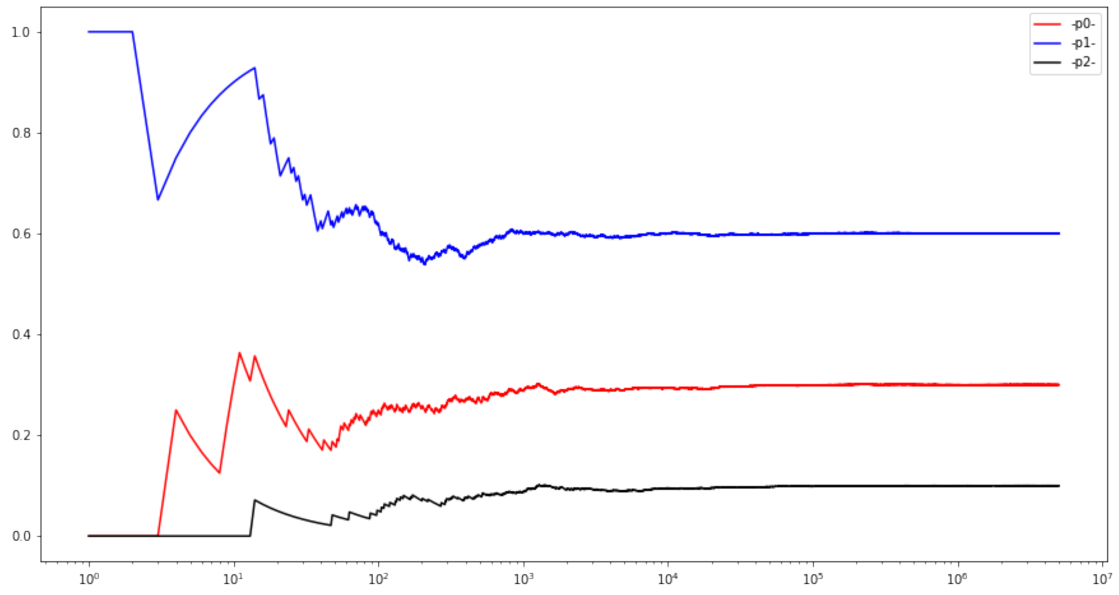
ii) Let $prob_0=0.3$, $prob_1=0.6$ and $prob_2=0.1$

iii) Count the number of occurences and divide by the number of total draws for 3 scenarios :

1. p_0 : ¿ $prob_0$
2. p_1 : ¿ $prob_1$
3. p_2 : ¿ $prob_2$

```
n = 5000000 # Number of points (>1000000)
y = np.random.uniform(size=n) # Generate n points from uniform
distribution range from [0 1]
x = np.arange(1, n+1)
prob0 = 0.3
prob1 = 0.6
prob2 = 0.1

# count number of occurrences and divide by the number of total draws

p0 = np.cumsum(y<prob0)/x # insert your code here
p1 = np.cumsum(y<prob1)/x # insert your code here
p2 = np.cumsum(y<prob2)/x # insert your code here


plt.figure(figsize=(15, 8))
plt.semilogx(x, p0,color='r')
plt.semilogx(x, p1,color='b')
plt.semilogx(x,p2,color='k')
plt.legend(['-p0-','-p1-','-p2-'])
```

```
<matplotlib.legend.Legend at 0x26b9d044e50>
```



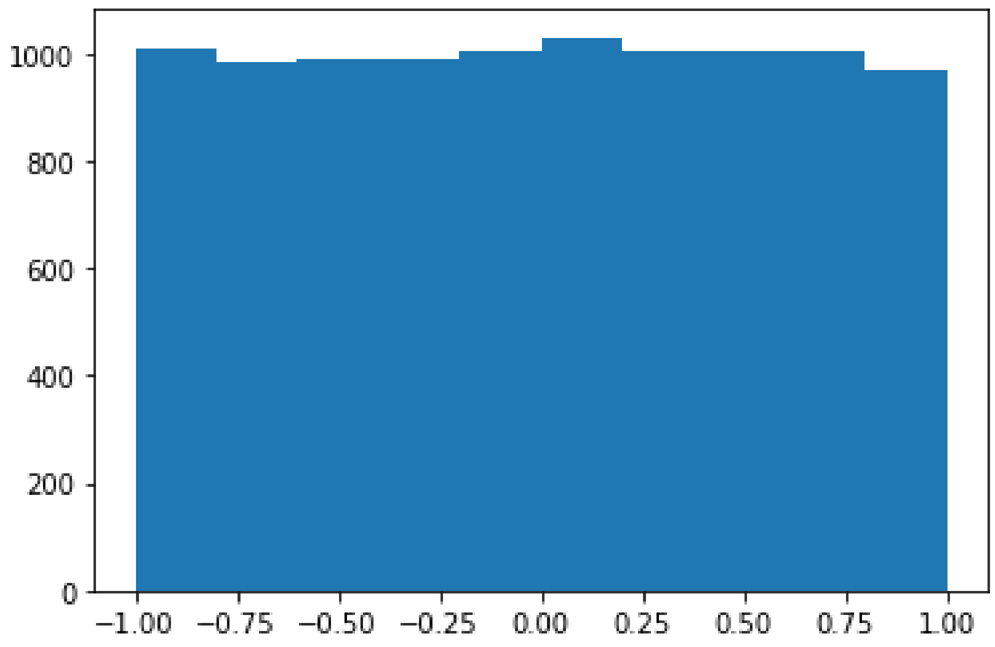
4. Central limit theorem

a) Sample from a uniform distribution (-1,1), some 10000 no. of samples 1000 times (u1,u2,....,u1000). show addition of iid rendom variables converges to a Gaussian distribution as number of variables tends to infinity.

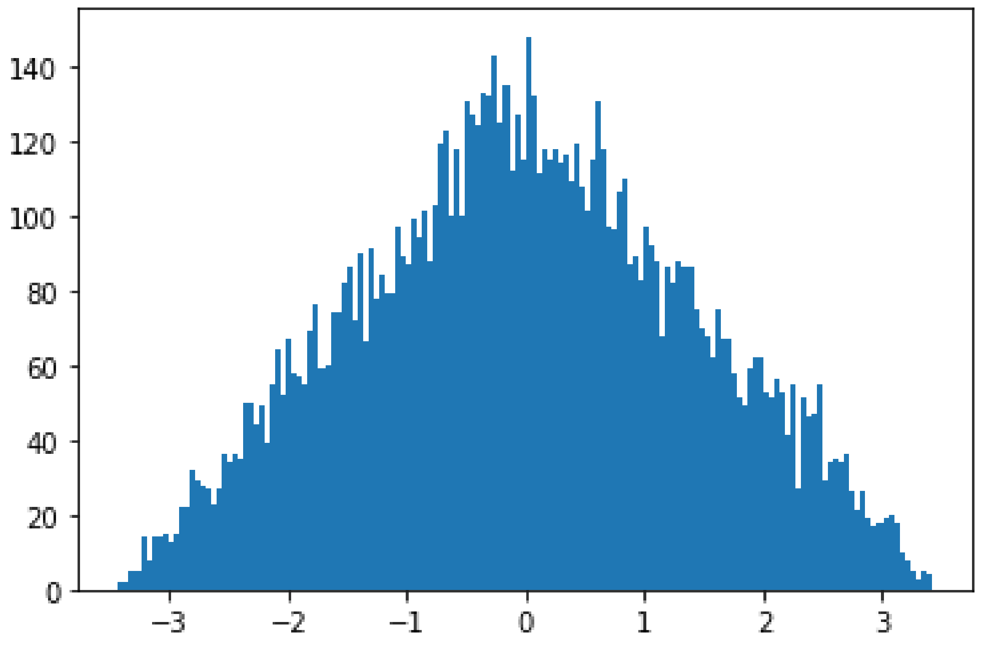
```
x = np.random.uniform(low=-1, high=1, size=(10000, 100)) # Generate
1000 diferent uniform distributions of 10000 samples each in range
from [-1 1]
```

```
plt.figure()
plt.hist(x[:,0])

# addition of 2 random variables
tmp2=np.sum(x[:,0:2],axis=1)/(np.std(x[:,0:2]))
plt.figure()
plt.hist(tmp2,150)

# Repeat the same for 100 and 1000 random variables

# addition of 100 random variables
# start code here
tmp100=np.sum(x[:,0:100],axis=1)/(np.std(x[:,0:100]))
plt.figure()
plt.hist(tmp100,150)

# addition of 1000 random variables
# start code here
tmp1000=np.sum(x,axis=1)/(np.std(x))
plt.figure()
plt.hist(tmp1000,150)
```

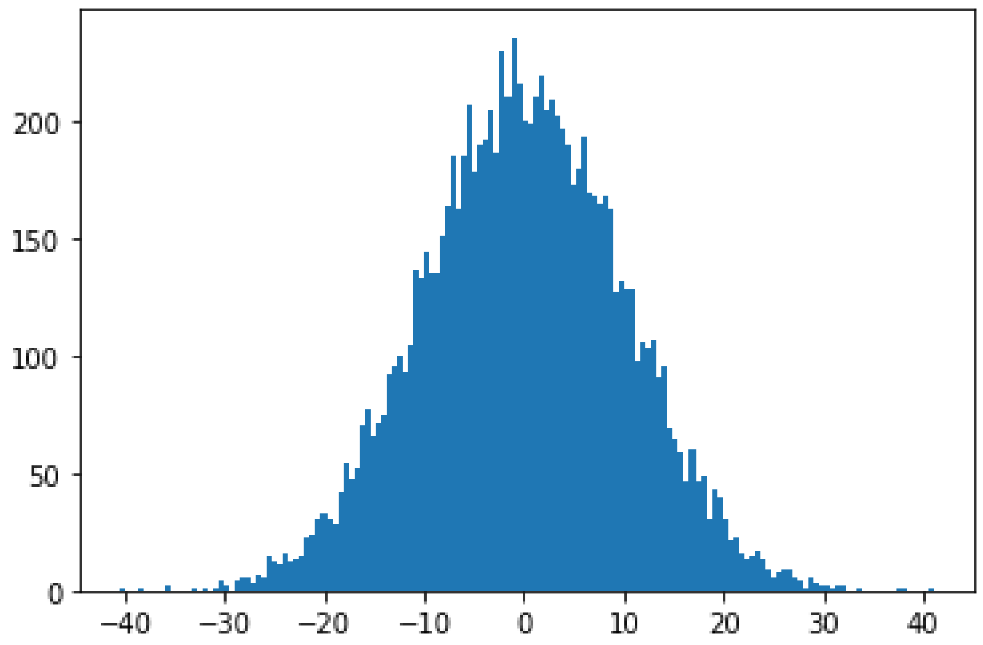
```
(array([  1.,   0.,   0.,   1.,   0.,   0.,   0.,   0.,   2.,   0.,
0.,
          0.,   0.,   1.,   0.,   1.,   0.,   1.,   4.,   2.,   0.,
4.,
          5.,   5.,   3.,   7.,   6.,  15.,  12.,  11.,  16.,  12.,
14.,
         15.,  23.,  24.,  30.,  33.,  30.,  28.,  42.,  54.,  48.,
52.,
         70.,  77.,  66.,  72.,  75.,  92.,  96., 100.,  93., 105.,
137.,
        133., 145., 135., 135., 151., 164., 185., 163., 186., 207.,
179.,
        190., 192., 205., 187., 230., 211., 236., 216., 200., 199.,
211.,
        220., 205., 210., 203., 197., 190., 173., 180., 194., 170.,
168.,
        165., 168., 163., 127., 132., 129., 129.,  98., 106., 103.,
107.,
         91.,  96.,  69.,  65.,  59.,  46.,  60.,  47.,  49.,  31.,
43.,
         40.,  30.,  21.,  23.,  16.,  14.,  15.,  17.,  14.,   9.,
5.,
          8.,   9.,   9.,   6.,   4.,   1.,   5.,   3.,   2.,   2.,
1.,
          2.,   2.,   0.,   0.,   1.,   0.,   0.,   0.,   0.,   0.,
0.,
         1.,    1.,    0.,    0.,    0.,    0.,    1.]),
array([-40.38081852, -39.83835961, -39.29590071, -38.75344181,
       -38.21098291, -37.66852401, -37.12606511, -36.5836062 ,
       -36.0411473 , -35.4986884 , -34.9562295 , -34.4137706 ,
       -33.8713117 , -33.32885279, -32.78639389, -32.24393499,
       -31.70147609, -31.15901719, -30.61655829, -30.07409938,
       -29.53164048, -28.98918158, -28.44672268, -27.90426378,
       -27.36180488, -26.81934597, -26.27688707, -25.73442817,
       -25.19196927, -24.64951037, -24.10705147, -23.56459256,
       -23.02213366, -22.47967476, -21.93721586, -21.39475696,
       -20.85229806, -20.30983915, -19.76738025, -19.22492135,
       -18.68246245, -18.14000355, -17.59754465, -17.05508574,
       -16.51262684, -15.97016794, -15.42770904, -14.88525014,
       -14.34279124, -13.80033233, -13.25787343, -12.71541453,
       -12.17295563, -11.63049673, -11.08803783, -10.54557892,
       -10.00312002,  -9.46066112,  -8.91820222,  -8.37574332,
        -7.83328442,  -7.29082551,  -6.74836661,  -6.20590771,
        -5.66344881,  -5.12098991,  -4.57853101,  -4.0360721 ,
        -3.4936132 ,  -2.9511543 ,  -2.4086954 ,  -1.8662365 ,
        -1.3237776 ,  -0.78131869,  -0.23885979,   0.30359911,
         0.84605801,   1.38851691,   1.93097581,   2.47343472,
         3.01589362,   3.55835252,   4.10081142,   4.64327032,
         5.18572922,   5.72818813,   6.27064703,   6.81310593,
         7.35556483,   7.89802373,   8.44048264,   8.98294154,
         9.52540044,  10.06785934,  10.61031824,  11.15277714,
        11.69523605,  12.23769495,  12.78015385,  13.32261275,
        13.86507165,  14.40753055,  14.94998946,  15.49244836,
        16.03490726,  16.57736616,  17.11982506,  17.66228396,
        18.20474287,  18.74720177,  19.28966067,  19.83211957,
        20.37457847,  20.91703737,  21.45949628,  22.00195518,
        22.54441408,  23.08687298,  23.62933188,  24.17179078,
        24.71424969,  25.25670859,  25.79916749,  26.34162639,
        26.88408529,  27.42654419,  27.9690031 ,  28.511462  ,
        29.0539209 ,  29.5963798 ,  30.1388387 ,  30.6812976 ,
        31.22375651,  31.76621541,  32.30867431,  32.85113321,
        33.39359211,  33.93605101,  34.47850992,  35.02096882,
        35.56342772,  36.10588662,  36.64834552,  37.19080442,
        37.73326333,  38.27572223,  38.81818113,  39.36064003,
        39.90309893,  40.44555783,  40.98801674]),
<BarContainer object of 150 artists>)
```

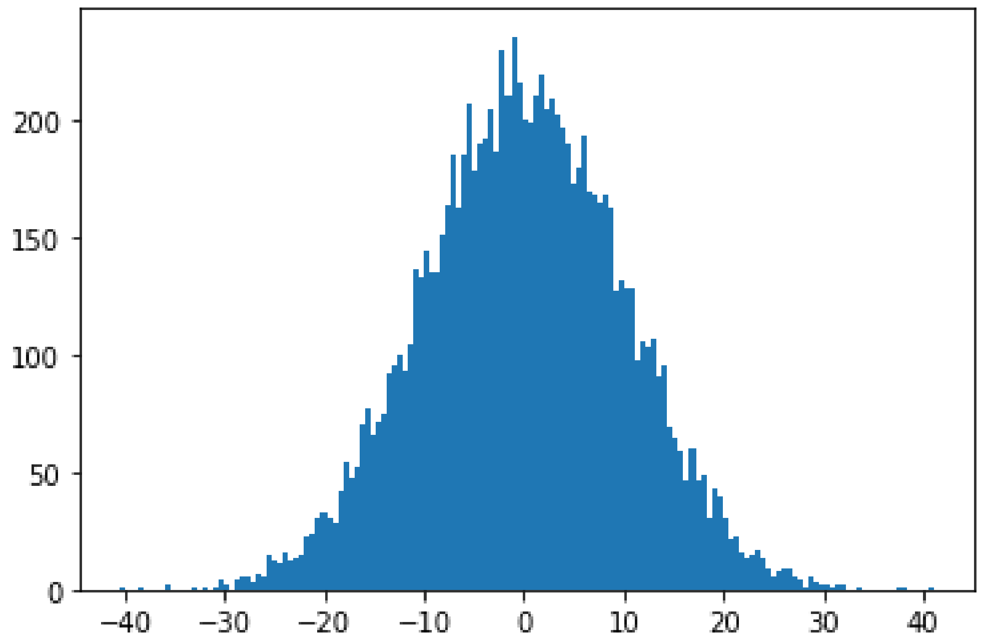


1000
800
600
400
200
0
−1.00
−0.75
−0.50
−0.25
0.00
0.25
0.50
0.75
1.00



140
120
100
80
60
40
20
0
−3
−2
−1
0
1
2
3





5. Computing π using sampling

a) Generate 2D data from uniform distribution of range -1 to 1 and compute the value of π.

b) Equation of circle

$$x^2 + y^2 = 1$$

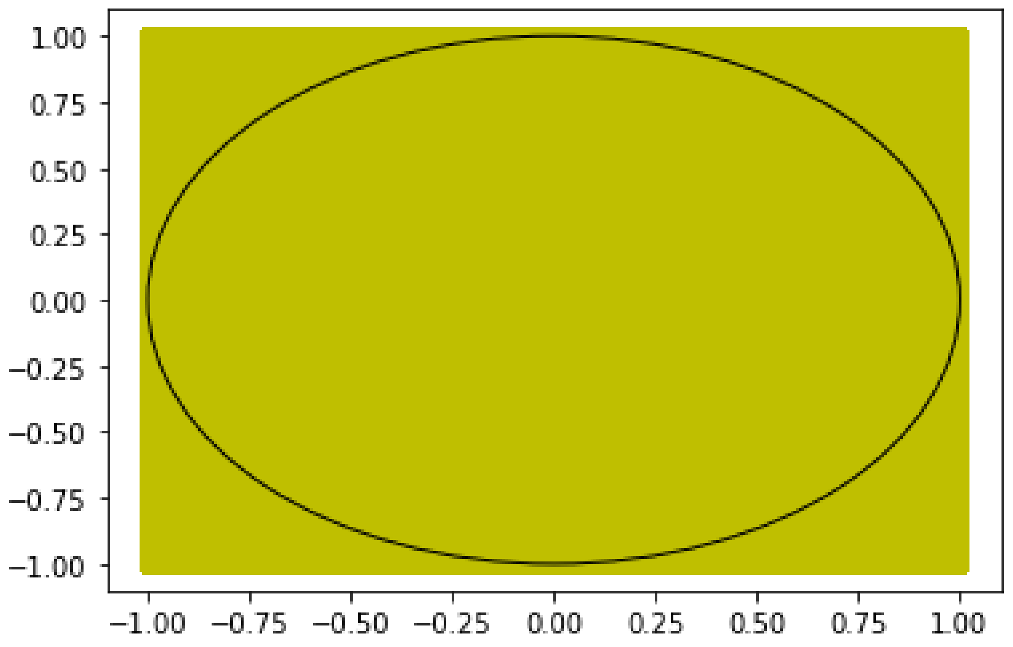
c) Area of a circle can be written as:

$$\frac{No of points\left(x^2+y^2<¿1\right)}{Total no. generated points}=\frac{\pi r^2}{\left(2r\right)^2}$$

where r is the radius of the circle and $2r$ is the length of the vertices of square.

```
import numpy as np
import matplotlib.pyplot as plt
fig = plt.gcf()
ax = fig.gca()

radius = 1

n = int(1e7) # set the value of n (select large n for better results)
x = np.random.uniform(low=-1, high=1, size=(n, 2)) # Generate n 
samples of 2D data from uniform distribution from range -1 to 1 
(output will be a (n X 2) matrix) (Ref = 
https://numpy.org/doc/stable/reference/random/generated/numpy.random.u
niform.html )

ax.scatter(x[:,0],x[:,1],color='y') # Scatter plot of x

# find the number points present inside the circle

x_cr = np.sum(np.linalg.norm(x, axis=1)<=1) # insert your code here

circle1 = plt.Circle((0, 0), 1, fc='None',ec='k')
ax.add_artist(circle1) # plotting circle of radius 1 with centre at 
(0,0)

pi = 4*x_cr/n # calculate pi value using x_cr and radius

print('computed value of pi=',pi)
```

```
computed value of pi= 3.141746
```



6. Monty Hall problem

Here's a fun and perhaps surprising statistical riddle, and a good way to get some practice writing python functions

In a gameshow, contestants try to guess which of 3 closed doors contain a cash prize (goats are behind the other two doors). Of course, the odds of choosing the correct door are 1 in 3. As a twist, the host of the show occasionally opens a door after a contestant makes his or her choice. This door is always one of the two the contestant did not pick, and is also always one of the goat doors (note that it is always possible to do this, since there are two goat doors). At this point, the contestant has the option of keeping his or her original choice, or swtiching to the other unopened door. The question is: is there any benefit to switching doors? The answer surprises many people who haven't heard the question before.

Follow the function descriptions given below and put all the functions together at the end to calculate the percentage of winning cash prize in both the cases (keeping the original door and switching doors)

Note : You can write your own functions, the below ones are given for reference, the goal is to calculate the win percentage

Try this fun problem and if you find it hard, you can refer to the solution here

```
"""
Function
--------
simulate_prizedoor

Generate a random array of 0s, 1s, and 2s, representing
```

```
hiding a prize between door 0, door 1, and door 2

Parameters
----------
nsim : int
    The number of simulations to run

Returns
-------
sims : array
    Random array of 0s, 1s, and 2s

Example
-------
>>> print simulate_prizedoor(3)
array([0, 0, 2])
"""
def simulate_prizedoor(nsim):

  sims = np.random.randint(low=0, high=3, size=nsim) # write your code
here

  return sims
```

```
simulate_prizedoor(3)
```

```
array([2, 0, 2])
```

```
"""
Function
--------
simulate_guess

Return any strategy for guessing which door a prize is behind. This
could be a random strategy, one that always guesses 2, whatever.

Parameters
----------
nsim : int
    The number of simulations to generate guesses for

Returns
-------
guesses : array
    An array of guesses. Each guess is a 0, 1, or 2

Example
-------
>>> print simulate_guess(5)
array([0, 0, 0, 0, 0])
```

```
"""
#your code here

def simulate_guess(nsim):

  guesses = np.random.randint(0, 3, nsim) # write your code here

  return guesses

simulate_guess(5)

array([2, 1, 0, 1, 2])

import random
"""
Function
--------
goat_door

Simulate the opening of a "goat door" that doesn't contain the prize,
and is different from the contestants guess

Parameters
----------
prizedoors : array
    The door that the prize is behind in each simulation
guesses : array
    THe door that the contestant guessed in each simulation

Returns
-------
goats : array
    The goat door that is opened for each simulation. Each item is 0,
1, or 2, and is different
    from both prizedoors and guesses

Examples
--------
>>> print goat_door(np.array([0, 1, 2]), np.array([1, 1, 1]))
>>> array([2, 2, 0])
"""
# write your code here # Define a function and return the required
array
def goat_door(prizedoors, guesses):
    nsim = len(prizedoors)
    goats = []
    for i in range(nsim):
        guess = [0, 1, 2]
        if prizedoors[i] in guess: guess.remove(prizedoors[i])
        if guesses[i] in guess: guess.remove(guesses[i])
        goat = random.choice(guess)
        goats.append(goat)
    return np.array(goats)

goat_door(np.array([0, 1, 2]), np.array([1, 1, 1]))
```

```
array([2, 2, 0])
```

```
"""
Function
--------
switch_guess

The strategy that always switches a guess after the goat door is
opened

Parameters
----------
guesses : array
     Array of original guesses, for each simulation
goatdoors : array
     Array of revealed goat doors for each simulation

Returns
-------
The new door after switching. Should be different from both guesses
and goatdoors

Examples
--------
>>> print switch_guess(np.array([0, 1, 2]), np.array([1, 2, 1]))
>>> array([2, 0, 0])
"""
# write your code here # Define a function and return the required
array
def switch_guess(guesses, goatdoors):
    nsim = len(goatdoors)
    doors = []
    for i in range(nsim):
        guess = [0, 1, 2]
        guess.remove(goatdoors[i])
        guess.remove(guesses[i])
        door = guess[0]
        doors.append(door)
    return np.array(doors)

goat_door(np.array([0, 1, 2]), np.array([1, 2, 1]))
```

```
array([2, 0, 0])
```

```python
"""
Function
--------
win_percentage

Calculate the percent of times that a simulation of guesses is correct

Parameters
-----------
guesses : array
    Guesses for each simulation
prizedoors : array
    Location of prize for each simulation

Returns
--------
percentage : number between 0 and 100
    The win percentage

Examples
---------
>>> print win_percentage(np.array([0, 1, 2]), np.array([0, 0, 0]))
33.333
"""

def win_percentage(guesses, prizedoors):

  answer = 100 * (guesses == prizedoors).mean()

  return answer

win_percentage(np.array([0, 1, 2]), np.array([0, 0, 0]))
```

```
33.33333333333333
```

```python
## Put all the functions together here

nsim = int(1e5) # Number of simulations

## case 1 : Keep guesses
prizedoors = simulate_prizedoor(nsim)
guesses = simulate_guess(nsim)
print("Win percentage without switching: ",
round(win_percentage(guesses, prizedoors), 3), "%")
# write your code here (print the win percentage when keeping original
door)

## case 2 : switch
prizedoors = simulate_prizedoor(nsim)
guesses = simulate_guess(nsim)
```

```
goatdoors = goat_door(prizedoors, guesses)
switchguess = switch_guess(guesses, goatdoors)
print("Win percentage with switching: ",
round(win_percentage(switchguess, prizedoors), 3), "%")
# write your code here (print the win percentage when switching doors)
```

```
Win percentage without switching:  33.29 %
Win percentage with switching:  66.602 %
```

6. Prisoner problem

There is a prison with 100 prisoners numbered 1-100. One day the prison gaurd comes up with a game. There are 100 boxes numbered 1-100 and 100 slips numbered 1-100.

These slips are then placed randomly into the 100 boxes such that each box has exactly one slip. The rules state that every prisoner is allowed to open 50 boxes of his/her choice.

A prisoner wins if he/she selects a box in which the slip has the same number as their own serial number. If all the prisoners win, all the prisoners are let go, but if atleast one of them fails, all of them are put in prison for the rest of their lives.

The prisoners can discuss the strategy before the game starts. Once the game starts none of them can communicate with each other. What is their best strategy?

Environment Creation

```
import random
import tqdm.notebook as nb

# Creating the Box class having box number and slip number
class Box:
    def __init__(self, bn, sn):
        self.box_no = bn
        self.slip_no = sn

# Creating Environment playground for 100 boxes
class Environment:
    def __init__(self, N):
        boxes = list(range(N))
        slips = boxes[:]
        random.shuffle(slips)

        self.playground = []
        for bn, sn in zip(boxes, slips):
            box = Box(bn, sn)
            self.playground.append(box)

    def open(self, n):
        return self.playground[n].slip_no

# Function to save the result into a file
import json
```

```
def save(name, data):
    with open(f'{name}.json', 'w+') as fp:
        json.dump(data, fp, indent=4)
```

Random Strategy: Each prisoner picks 50 random boxes

```
# Function to generate 'n' random picks out of 100
def generate_picks(n):
    picks = list(range(100))
    random.shuffle(picks)
    picks = picks[:n]
    return picks
```

Solution Approach 1: Each of them randomly picks 50 boxes

```
# Number of simulations
epochs = 1000
final_result = []

# Iterate through each simulation
for i in nb.tqdm(range(epochs)):
    result = {}
    outcomes = [0 for j in range(100)] # Win or Loss for each prisoner
    env = Environment(100)  # Create a new environment of size 100
    for prisoner_no in range(100):  # Iterate through each prisoner
        picks = generate_picks(50)  # Generate 50 random picks
        for b, pick in enumerate(picks):
            if env.playground[pick].slip_no==prisoner_no:  # For each
check whether picked box has slip number same as prisoner
                outcomes[prisoner_no]=b+1
                break
    for b, no_of_tries in enumerate(outcomes): # Store number of tries
to get to same number for each prisoner in the current epoch
        result[f'prisoner{b}'] = no_of_tries
    result["outcome"] = 1 if outcomes.count(0)==0 else 0  # Check
outcome that not all are zeros.
    final_result.append(result)  # Append epoch result to final result

save("approach1", final_result)  # Save to approach1.json
```

{"version_major":2,"version_minor":0,"model_id":"81153a87b4824a44a1630
a4c669a9d48"}

Solution Approach 2: Pick Chaining

Each prisoner picks the box with the number equal to the slip number in the previous box. The chain starts with the box with the same number as that of the prisoner.

```
epochs = 100000  # Define number of epochs of the simulation
final_result = []

for i in nb.tqdm(range(epochs)):
    result = {}
    outcomes = [0 for j in range(100)] # Win or Loss for each prisoner
    env = Environment(100) # Create a new environment of size 100
    for prisoner_no in range(100): # Iterate through each prisoner
        pick = prisoner_no   # first pick is the same as prisoner
number
        for b in range(50):
            if env.playground[pick].slip_no==prisoner_no:  # If
prisoner gets the number update outcomes and break
                outcomes[prisoner_no]=b+1
                break
            pick = env.playground[pick].slip_no   # Update next pick
equal to slip in the previous box.
    for b, no_of_tries in enumerate(outcomes): # Store number of tries
to get to same number for each prisoner in the current epoch
        result[f'prisoner{b}'] = no_of_tries
    result["outcome"] = 1 if outcomes.count(0)==0 else 0 # Check
outcome that not all are zeros.
    final_result.append(result)  # Append epoch result to final result

save("approach2", final_result) # Save to approach2.json
```

{"version_major":2,"version_minor":0,"model_id":"2ffeea05e2764649b5c8f
f97f273a8ca"}

Testing the results for the two cases

```
import json
# Approach 1: Random
with open("approach1.json", "r+") as f:
    result_random = json.load(f)

# Approach 2: Chaining
with open("approach2.json", "r+") as f:
    result_chain = json.load(f)

total = len(result_random)

chance_random = []
chance_chain = []
for i in range(total):
    chance_random.append(result_random[i]["outcome"])
    chance_chain.append(result_chain[i]["outcome"])

print("Chance of Freedom Random:", 100*sum(chance_random)/total, "%")
print("Chance of Freedom Chain:", 100*sum(chance_chain)/total, "%")
```

```
Chance of Freedom Random: 0.0 %
Chance of Freedom Chain: 33.8 %
```