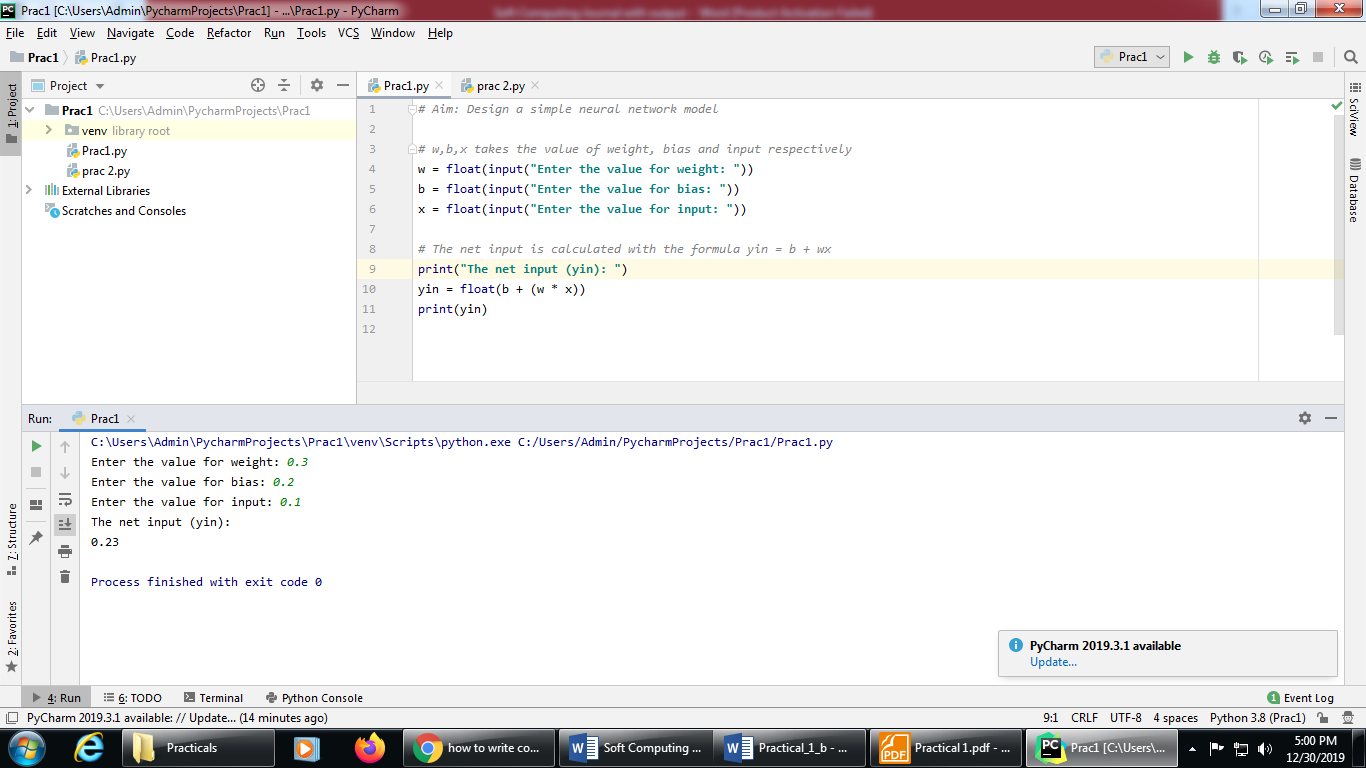
**PRACTICAL NO: 01 (A)**

**Title:** Design a simple neural network model

**Source Code:**

*# Aim: Design a simple neural network model  
  
# w, b, x takes the value of weight, bias and input respectively*w = float(input("Enter the value for weight: "))  
b = float(input("Enter the value for bias: "))  
x = float(input("Enter the value for input: "))  
  
*# The net input is calculated with the formula yin = b + wx*print("The net input (yin): ")  
yin = float(b + (w \* x))  
print(yin)

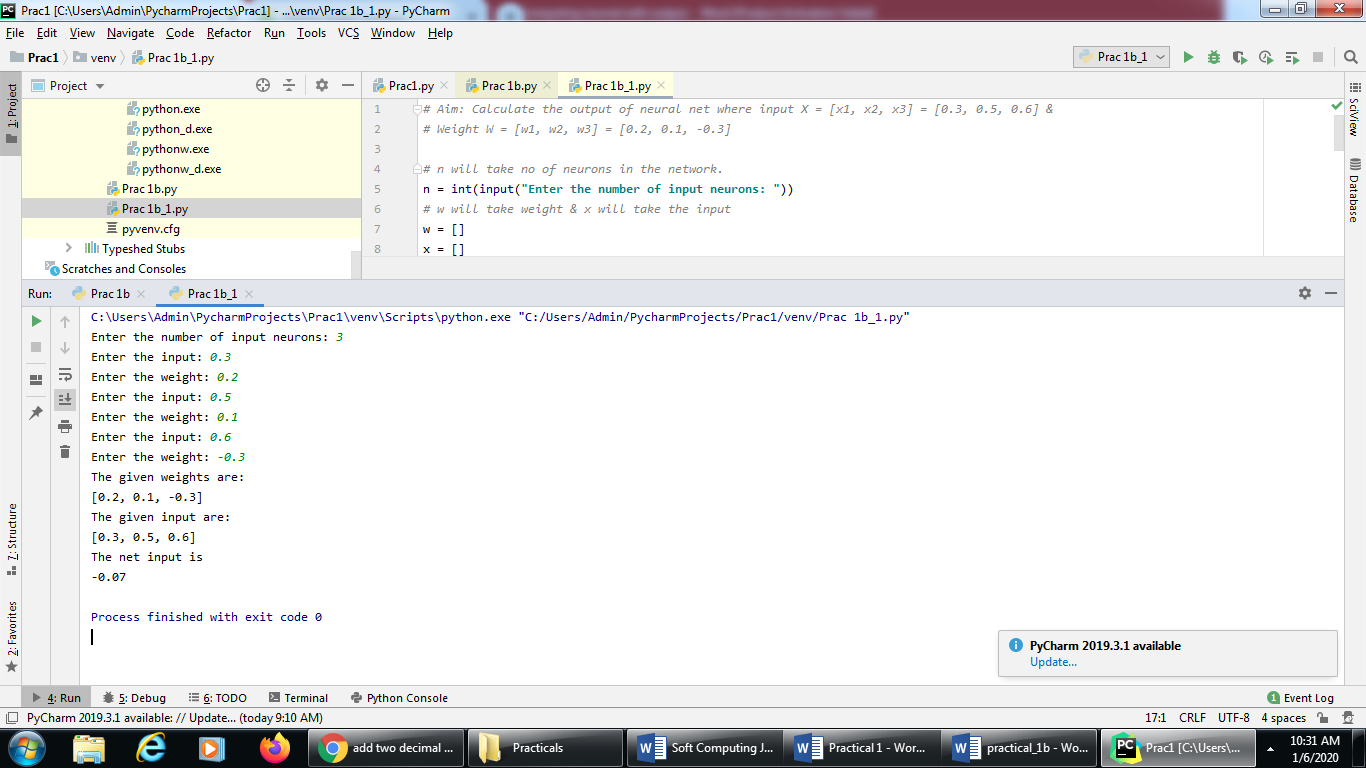
**Output:**

**PRACTICAL NO: 01 (B)**

**Title:** Calculate the output of neural net.

**Source Code for Problem statement A:**

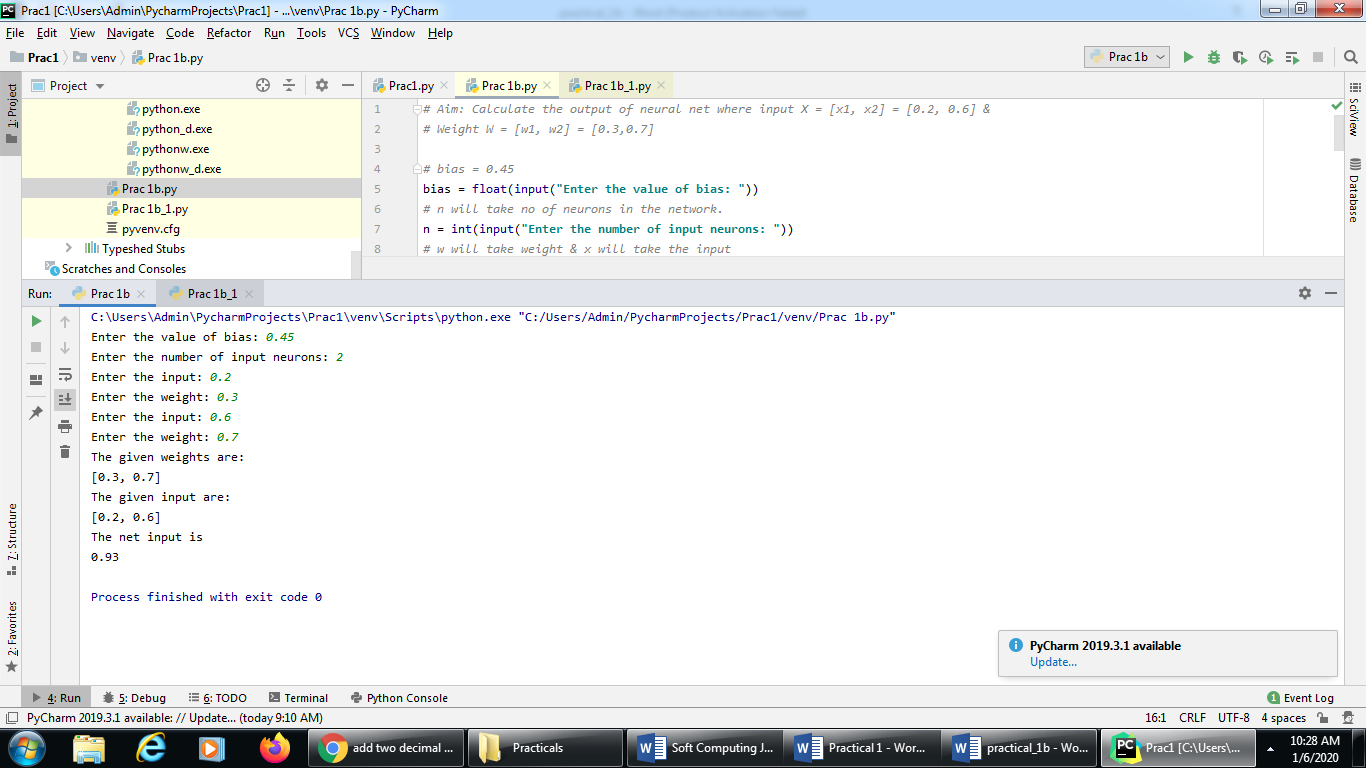
*# Aim: Calculate the output of neural net where input X = [x1, x2, x3] = [0.3, 0.5, 0.6] &  
# Weight W = [w1, w2, w3] = [0.2, 0.1, -0.3]  
  
# n will take no of neurons in the network.*n = int(input("Enter the number of input neurons: "))  
*# w will take weight & x will take the input*w = [ ]  
x = [ ]  
  
*# taking the value of input and their weight*for i inrange(0,n):  
 a = float(input("Enter the input: "))  
 x.append(a)  
 b = float(input("Enter the weight: "))  
 w.append(b)  
  
print("The given weights are: ")  
print(w)  
print("The given input are: ")  
print(x)  
  
y = 0.0  
for i in range(0,n):  
 y = y + (w[i]\*x[i])  
  
print(**"**The net input is **"**)  
print (round(y,3))

**Output:**

**Source Code for Problem statement B:**

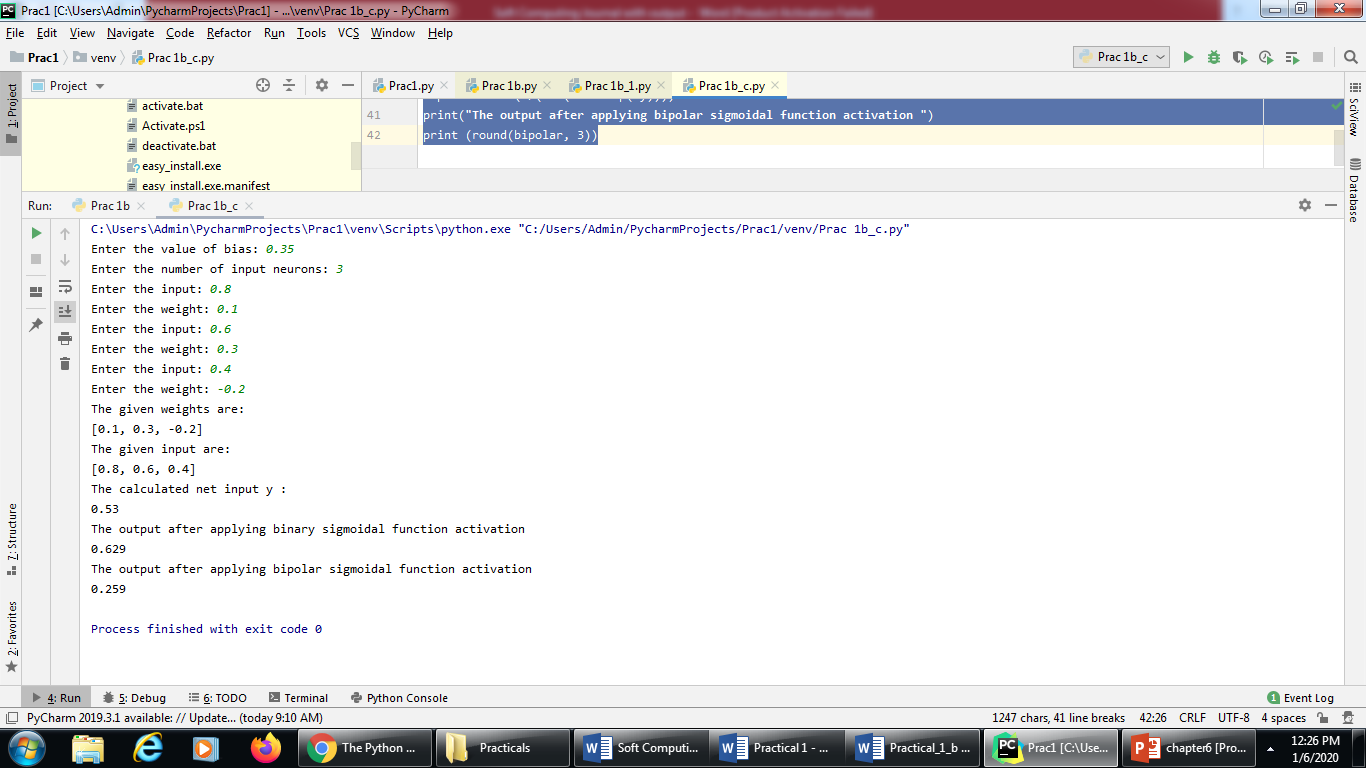
*# Aim: Calculate the output of neural net where input X = [x1, x2] = [0.2, 0.6] &  
# Weight W = [w1, w2] = [0.3,0.7] & bias = 0.45  
  
# bias = 0.45*bias = float(input("Enter the value of bias: "))  
*# n will take no of neurons in the network.*n = int(input("Enter the number of input neurons: "))  
*# w will take weight & x will take the input*w = []  
x = []  
  
for i in range(0,n):  
 a = float(input("Enter the input: "))  
 x.append(a)  
 b = float(input("Enter the weight: "))  
 w.append(b)  
  
print("The given weights are: ")  
print(w)  
print("The given input are: " )  
print(x)  
  
y = bias  
for i in range(0,n):  
 y = y + (w[i]\*x[i])

print("The net input is ")  
print (round(y,3))

**Output:**

**Source Code for Problem statement C:**

*# Aim: Calculate the output of neural net where input X = [x1, x2, x3] = [0.8, 0.6, 0.4]] &  
# Weight W = [w1, w2, w3] = [0.1, 0.3, -0.2] & bias =0.35.  
# Apply Binary & Bipolar Sigmoidal activation function  
# import math so that we can use mathematical function exp()*import math  
  
bias = float(input("Enter the value of bias: "))  
*# n will take no of neurons in the network.*n = int(input("Enter the number of input neurons: "))  
*# w will take weight & x will take the input*w = [ ]  
x = [ ]  
  
*# taking the value of input and their weight*for i in range(0,n):  
 a = float(input("Enter the input: "))  
 x.append(a)  
 b = float(input("Enter the weight: "))  
 w.append(b)  
  
print("The given weights are: ")  
print(w)  
print("The given input are: " )  
print(x)  
  
y = bias  
for i in range(0,n):  
 y = y + (w[i]\*x[i])  
  
print("The calculated net input y : ")  
print(y)  
  
*# Applying Binary Sigmoidal function on the net input i.e y*binary = 1/(1+ (math.exp(-y)))  
print("The output after applying binary sigmoidal function activation ")  
print (round(binary, 3))  
  
*# Applying Bipolar Sigmoidal function on the net input i.e y*bipolar = -1+(2/(1+ (math.exp(-y))))  
print("The output after applying bipolar sigmoidal function activation ")  
print (round(bipolar, 3))

**Output:**

**PRACTICAL NO: 02 (A)**

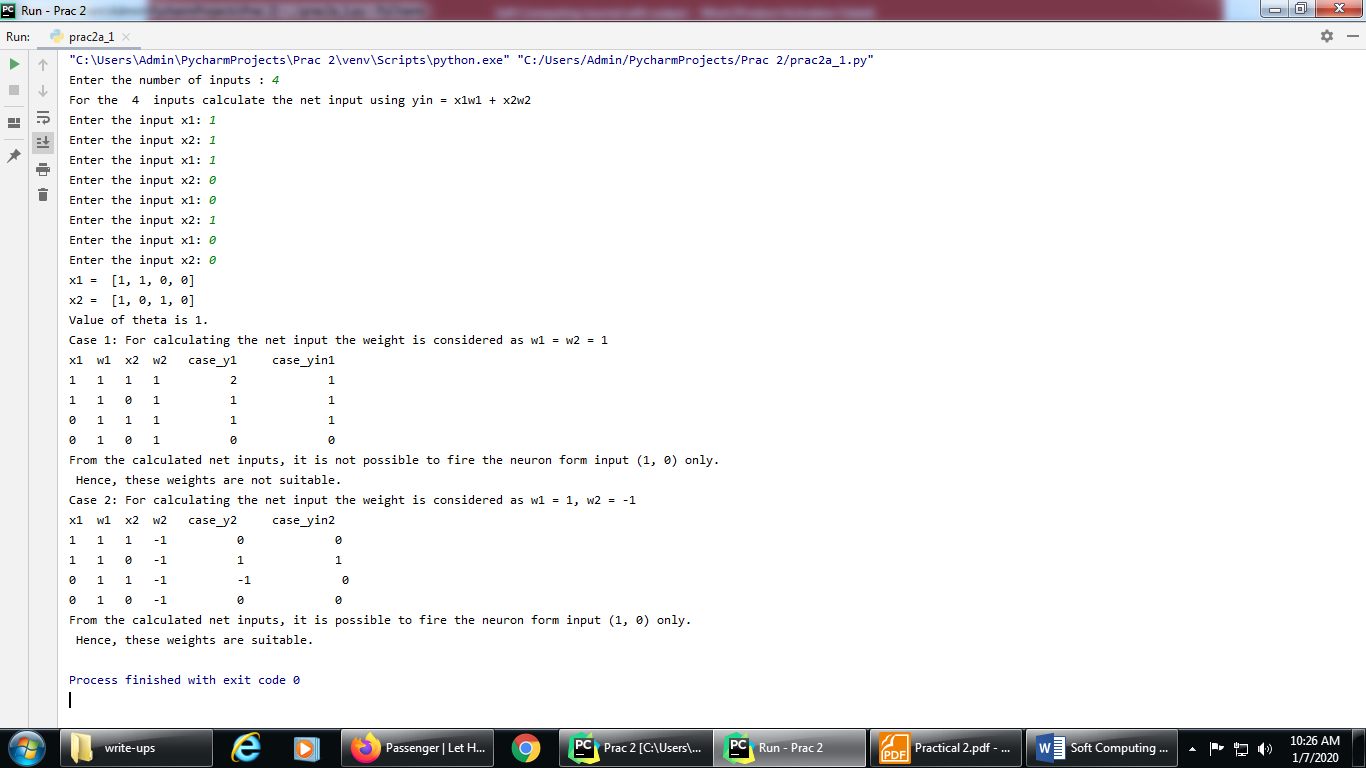
**Title:** Implement AND/NOT function using McCulloch-Pits neuron

**Source Code:**

*#Aim: Generate AND/NOT function using McCulloch-Pitts (M-P) neural net.  
# Logic: In the case of ANDNOT function,  
# the response is true if the first input is true and the second input is false.  
# Threshold for this neural network will be theta > = 1  
# There are two inputs and binary values will be taken as input i.e. either 0 or 1.*

num\_ip = int(input("Enter the number of inputs : "))  
print("For the ", num\_ip , " inputs calculate the net input using yin = x1w1 + x2w2 ")  
theta = 1  
x1 =[]  
x2 =[]  
  
for i in range(0,num\_ip):  
 a = int(input("Enter the input x1: "))  
 x1.append(a)  
 b = int(input("Enter the input x2: "))  
 x2.append(b)  
  
print("x1 = ", x1)  
print("x2 = ", x2)  
print("Value of theta is 1.")  
  
print("Case 1: For calculating the net input the weight is considered as w1 = w2 = 1")  
w1 = w2 = 1  
case\_y1 =[]  
case\_yin1 = []  
print("x1 w1 x2 w2 case\_y1 case\_yin1")  
for i in range(0,num\_ip):  
 case\_y1.append(x1[i]\*w1 + x2[i]\*w2)  
 if (case\_y1[i] >= theta):  
 case\_yin1.append( 1 )  
 else:  
 case\_yin1.append( 0 )  
 print(x1[i]," ", w1," ",x2[i]," ",w2," ", case\_y1[i]," ",case\_yin1[i])  
print("From the calculated net inputs, it is not possible to fire the neuron form input (1, 0) only."

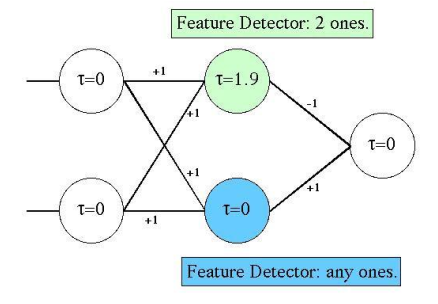
"\n Hence, these weights are not suitable. ")  
  
print("Case 2: For calculating the net input the weight is considered as w1 = 1, w2 = -1")  
w1 = 1  
w2 = -1  
case\_y2 =[]  
case\_yin2 = []  
print("x1 w1 x2 w2 case\_y2 case\_yin2")  
for i in range(0,num\_ip):  
 case\_y2.append(x1[i]\*w1 + x2[i]\*w2)  
 if (case\_y2[i] >= theta):  
 case\_yin2.append( 1 )  
 else:  
 case\_yin2.append( 0 )  
 print(x1[i]," ", w1," ",x2[i]," ",w2," ", case\_y2[i]," ",case\_yin2[i])  
print("From the calculated net inputs, it is possible to fire the neuron form input (1, 0) only."  
"\n Hence, these weights are suitable. ")

**Output:**

**PRACTICAL NO: 02 (B)**

**Title:** Generate XOR function using McCulloch-Pitts neural net.

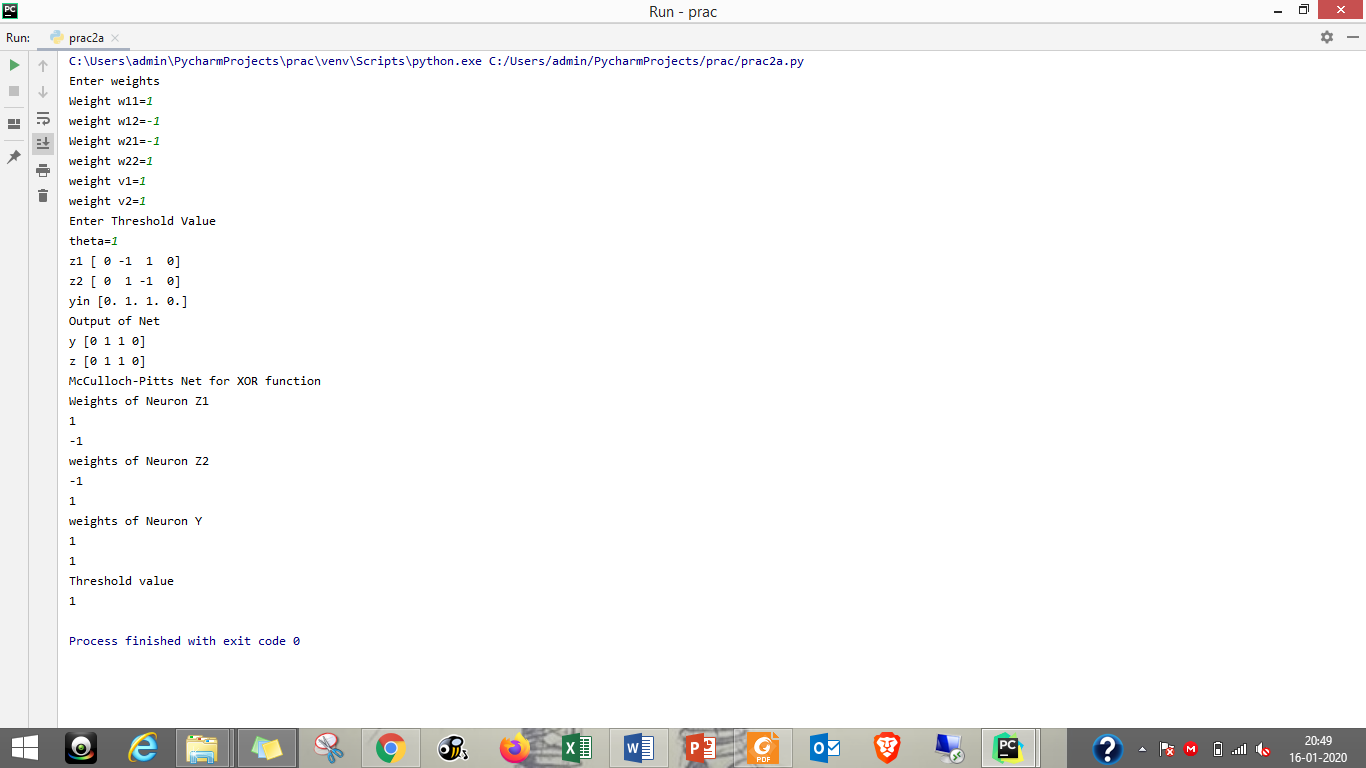
**XOR neural network:**



**Source code:**

*#Aim: Creating XOR network with the two neurons in the hidden layers.  
  
#Getting weights and threshold value*import numpy as np  
print('Enter weights')  
w11=int(input('Weight w11='))  
w12=int(input('weight w12='))  
w21=int(input('Weight w21='))  
w22=int(input('weight w22='))  
v1=int(input('weight v1='))  
v2=int(input('weight v2='))  
print('Enter Threshold Value')  
theta=int(input('theta='))  
x1=np.array([0, 0, 1, 1])  
x2=np.array([0, 1, 0, 1])  
z=np.array([0, 1, 1, 0])  
con=1  
y1=np.zeros((4,))  
y2=np.zeros((4,))  
y=np.zeros((4,))  
while con==1:  
 zin1=np.zeros((4,))  
 zin2=np.zeros((4,))  
 zin1=x1\*w11+x2\*w21  
 zin2=x1\*w21+x2\*w22  
  
 print("z1",zin1)  
 print("z2",zin2)  
  
 for i in range(0,4):  
 if zin1[i]>=theta:  
 y1[i]=1  
 else:  
 y1[i]=0  
  
 if zin2[i]>=theta:  
 y2[i]=1  
 else:  
 y2[i]=0  
  
 yin=np.array([])  
 yin=y1\*v1+y2\*v2  
  
 for i in range(0,4):  
 if yin[i]>=theta:  
 y[i]=1  
 else:  
 y[i]=0  
  
 print("yin",yin)  
 print('Output of Net')  
 y=y.astype(int)  
 print("y",y)  
 print("z",z)  
 if np.array\_equal(y,z):  
 con=0  
 else:  
 print("Net is not learning enter another set of weights and Threshold value")  
 w11=input("Weight w11=")  
 w12=input("weight w12=")  
 w21=input("Weight w21=")  
 w22=input("weight w22=")  
 v1=input("weight v1=")  
 v2=input("weight v2=")  
 theta=input("theta=")  
  
print("McCulloch-Pitts Net for XOR function")  
print("Weights of Neuron Z1")  
print(w11)  
print(w21)  
print("weights of Neuron Z2")  
print(w12)  
print(w22)  
print("weights of Neuron Y")  
print(v1)  
print(v2)  
print("Threshold value")  
print(theta)

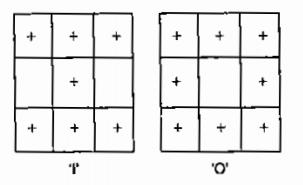
**Ouput:**



**PRACTICAL NO: 03 (A)**

**Title:** Write a program to implement Hebb’s rule.

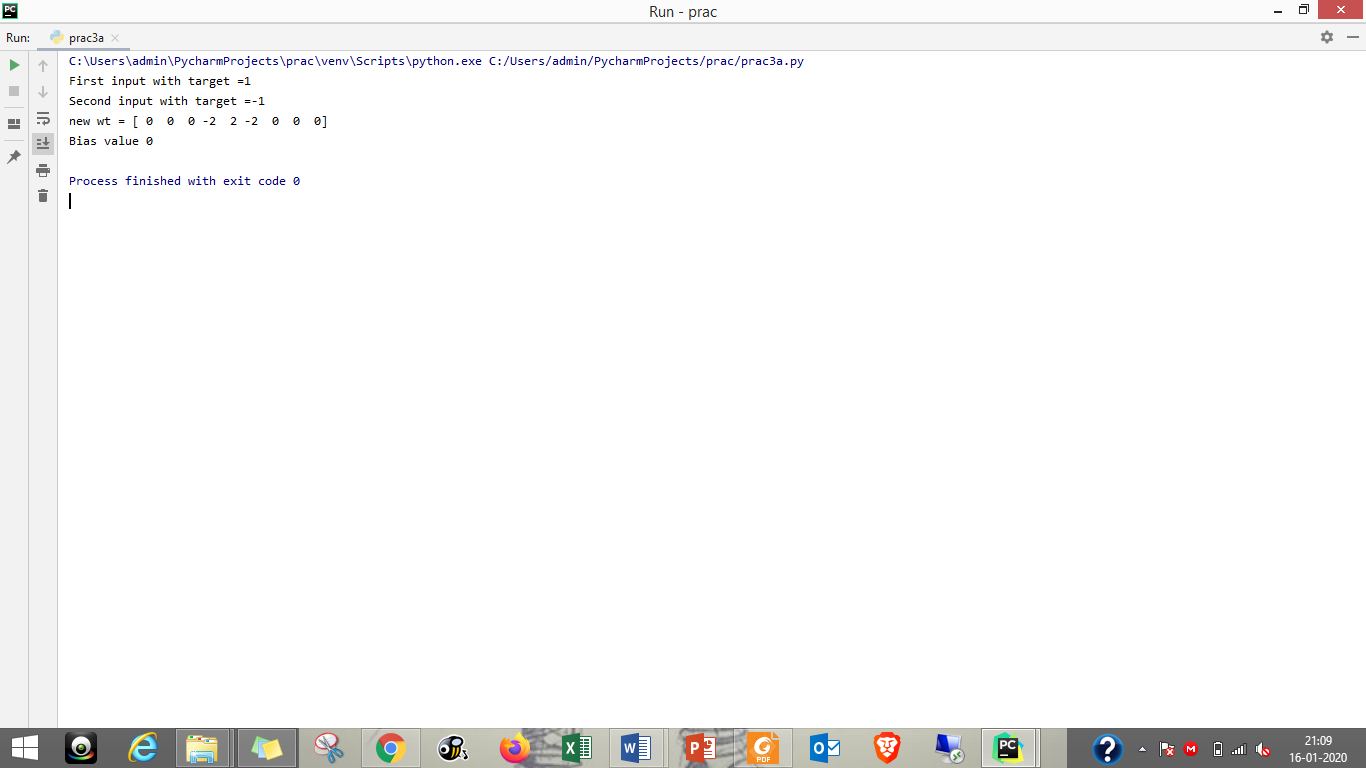
**Data for input pattern:**



**Source Code:**

*#Aim: Using Hebb rule, find the weights required to perform the above  
# classifications of the given input patterns. The patterns show 3x3 matrix  
# form in the square. The '+' symbol represents the value "+1" and the empty  
# square represents "-1". Consider "I" belongs to the member of the class (so  
# has target value 1) and "O" does not belong to the members of the class  
#(so has target value -1)*import numpy as np  
*#first pattern*x1=np.array([1,1,1,-1,1,-1,1,1,1])  
*#second pattern*x2=np.array([1,1,1,1,-1,1,1,1,1])  
*#initialize bais value*b=0  
*#define target*y=np.array([1,-1])  
wtold=np.zeros((9,))  
wtnew=np.zeros((9,))  
wtnew=wtnew.astype(int)  
wtold=wtold.astype(int)  
bais=0  
  
print("First input with target =1")  
for i in range(0,9):  
 wtold[i]=wtold[i]+x1[i]\*y[0]  
wtnew=wtold  
b=b+y[0]  
  
print("Second input with target =-1")  
for i in range(0,9):  
 wtnew[i]=wtold[i]+x2[i]\*y[1]  
b=b+y[1]  
print("new wt =", wtnew)  
print("Bias value",b)

**Output:**



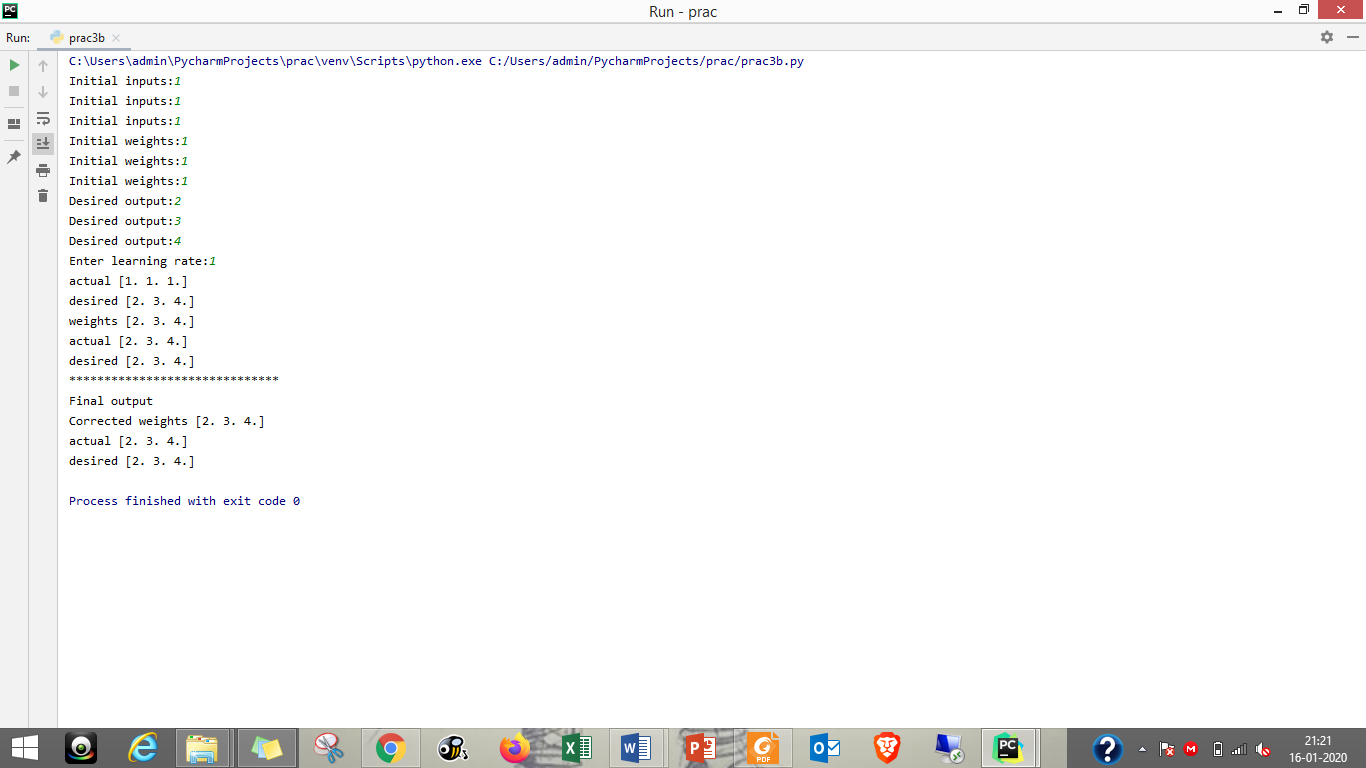
**PRACTICAL NO: 03 (B)**

**Title:** Write a program to implement Delta rule.

**Source Code:**

*#supervised learning*import numpy as np  
import time  
np.set\_printoptions(precision=2)  
x=np.zeros((3,))  
weights=np.zeros((3,))  
desired=np.zeros((3,))  
actual=np.zeros((3,))  
for i in range(0,3):  
 x[i]=float(input("Initial inputs:"))  
  
for i in range(0,3):  
 weights[i]=float(input("Initial weights:"))  
  
for i in range(0,3):  
 desired[i]=float(input("Desired output:"))  
  
a=float(input("Enter learning rate:"))  
actual=x\*weights  
print("actual",actual)  
print("desired",desired)  
  
while True:  
 if np.array\_equal(desired,actual):  
 break *#no change* else:  
 for i in range(0,3):  
 weights[i]=weights[i]+a\*(desired[i]-actual[i])  
  
 actual=x\*weights  
 print("weights",weights)  
 print("actual",actual)  
 print("desired",desired)  
print("\*"\*30)  
print("Final output")  
print("Corrected weights",weights)  
print("actual",actual)  
print("desired",desired)

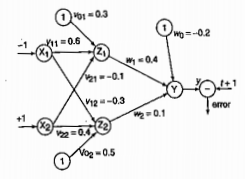
**Output:**



**PRACTICAL NO: 04 (A)**

**Title:** Write a program for Back Propagation Algorithm.

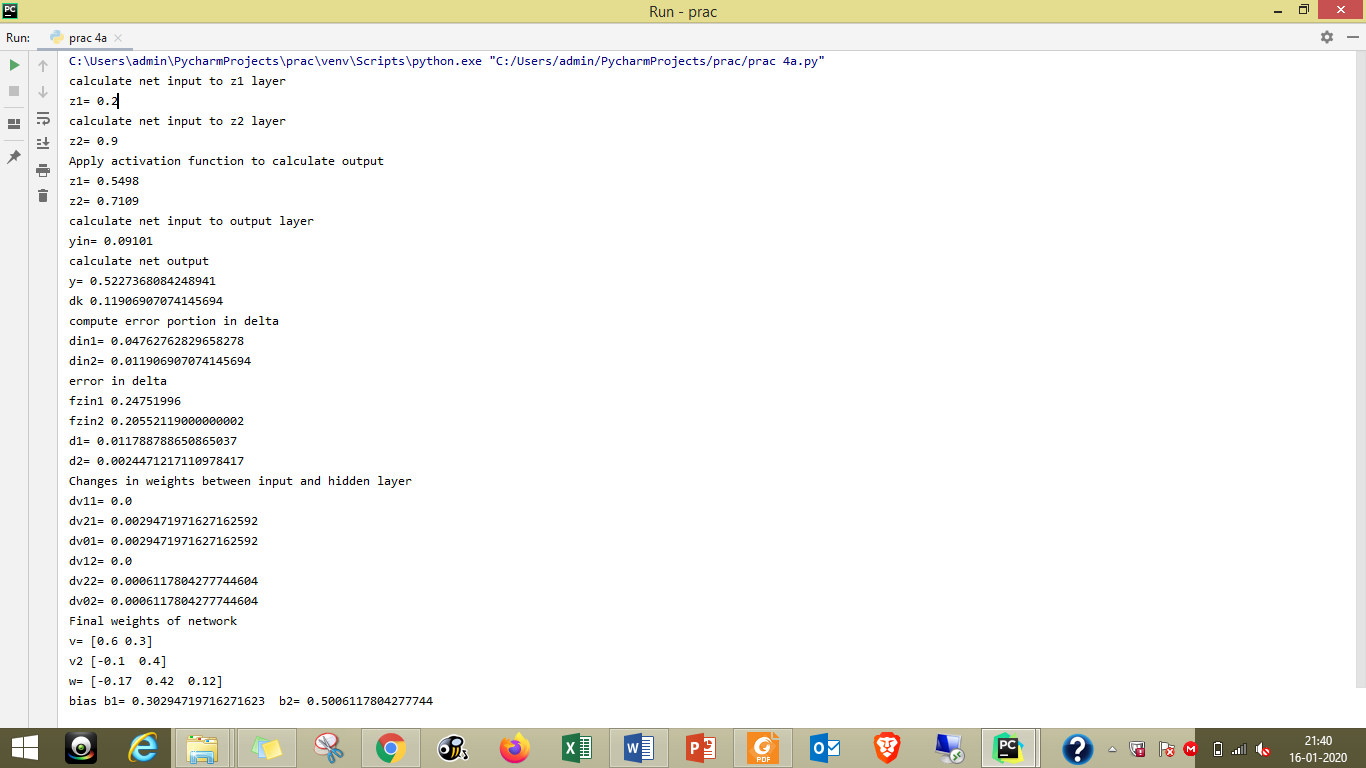
**Neural Network for Back Propagation:**



**Source Code:**

*#Aim: Implement the back propogation algorithm for the above  
# neural network*import numpy as np  
import decimal  
import math  
np.set\_printoptions(precision=2)  
v1=np.array([0.6, 0.3])  
v2=np.array([-0.1, 0.4])  
w=np.array([-0.2,0.4,0.1])  
b1=0.3  
b2=0.5  
x1=0  
x2=1  
alpha=0.25  
print("calculate net input to z1 layer")  
zin1=round(b1+ x1\*v1[0]+x2\*v2[0],4)  
print("z1=",round(zin1,3))  
  
print("calculate net input to z2 layer")  
zin2=round(b2+ x1\*v1[1]+x2\*v2[1],4)  
print("z2=",round(zin2,4))  
print("Apply activation function to calculate output")  
z1=1/(1+math.exp(-zin1))  
z1=round(z1,4)  
z2=1/(1+math.exp(-zin2))  
z2=round(z2,4)  
print("z1=",z1)  
print("z2=",z2)  
  
print("calculate net input to output layer")  
yin=w[0]+z1\*w[1]+z2\*w[2]  
print("yin=",yin)  
  
print("calculate net output")  
y=1/(1+math.exp(-yin))  
print("y=",y)  
  
fyin=y \*(1- y)  
dk=(1-y)\*fyin  
print("dk",dk)  
  
dw1= alpha \* dk \* z1  
dw2= alpha \* dk \* z2  
dw0= alpha \* dk  
  
print("compute error portion in delta")  
din1=dk\* w[1]  
din2=dk\* w[2]  
print("din1=",din1)  
print("din2=",din2)  
  
print("error in delta")  
fzin1= z1 \*(1-z1)  
print("fzin1",fzin1)  
d1=din1\* fzin1  
fzin2= z2 \*(1-z2)  
print("fzin2",fzin2)  
d2=din2\* fzin2  
  
print("d1=",d1)  
print("d2=",d2)  
  
print("Changes in weights between input and hidden layer")  
dv11=alpha \* d1 \* x1  
print("dv11=",dv11)  
dv21=alpha \* d1 \* x2  
print("dv21=",dv21)  
dv01=alpha \* d1  
print("dv01=",dv01)  
dv12=alpha \* d2 \* x1  
print("dv12=",dv12)  
dv22=alpha \* d2 \* x2  
print("dv22=",dv22)  
dv02=alpha \* d2  
print("dv02=",dv02)  
  
print("Final weights of network")  
v1[0]=v1[0]+dv11  
v1[1]=v1[1]+dv12  
print("v=",v1)  
v2[0]=v2[0]+dv21  
v2[1]=v2[1]+dv22  
print("v2",v2)  
w[1]=w[1]+dw1  
w[2]=w[2]+dw2  
b1=b1+dv01  
b2=b2+dv02  
w[0]=w[0]+dw0  
print("w=",w)  
print("bias b1=",b1, " b2=",b2)

**Output:**



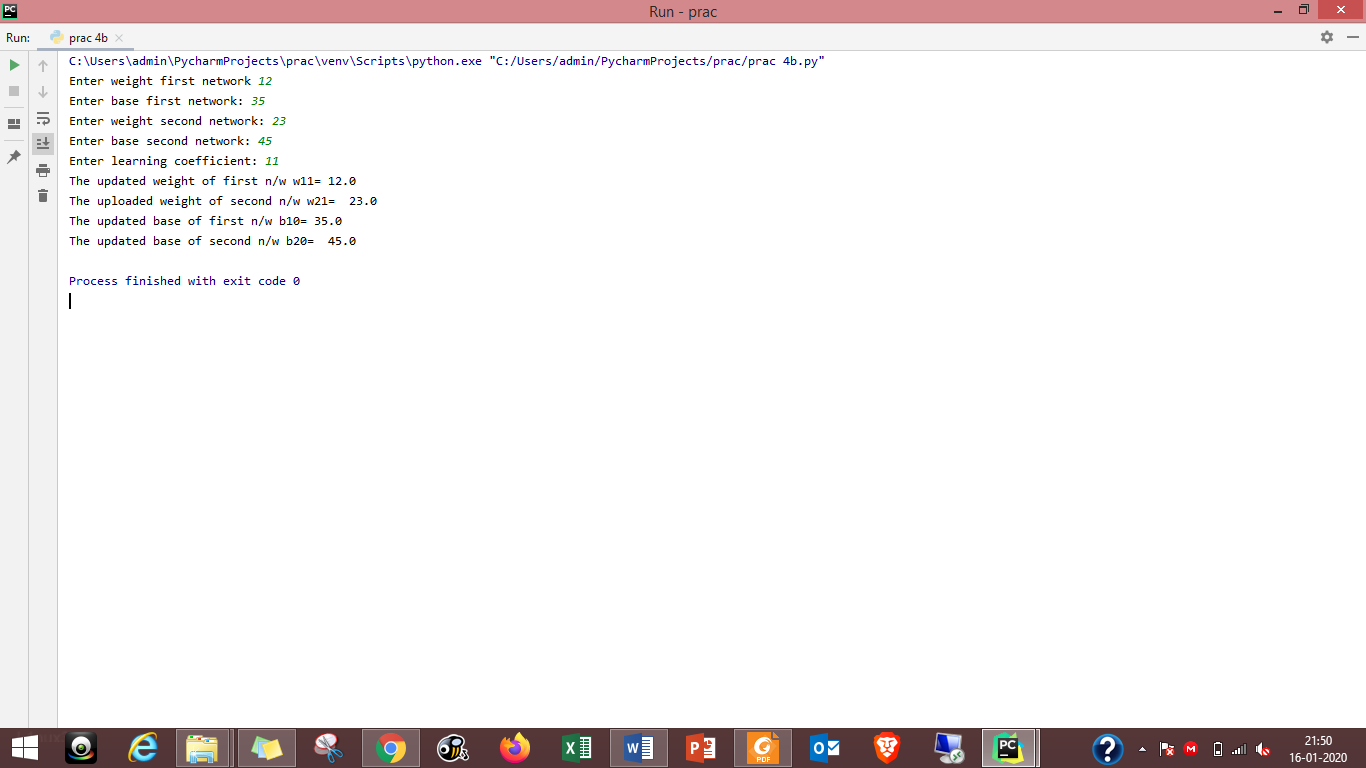
**PRACTICAL NO: 04 (B)**

**Title:** Write a Program For Error Back Propagation Algorithm (Ebpa) Learning

**Source Code:**

import math  
a0=-1  
t=-1  
w10=float(input("Enter weight first network"))  
b10=float(input("Enter base first network:"))  
w20=float(input("Enter weight second network:"))  
b20=float(input("Enter base second network:"))  
c=float(input("Enter learning coefficient:"))  
n1=float(w10\*c+b10)  
a1=math.tanh(n1)  
n2=float(w20\*a1+b20)  
a2=math.tanh(float(n2))  
e=t-a2  
s2=-2\*(1-a2\*a2)\*e  
s1=(1-a1\*a1)\*w20\*s2  
w21=w20-(c\*s2\*a1)  
w11=w10-(c\*s1\*a0)  
b21=b20-(c\*s2)  
b11=b10-(c\*s1)  
print("The updated weight of first n/w w11=",w11)  
print("The uploaded weight of second n/w w21= ",w21)  
print("The updated base of first n/w b10=",b10)  
print("The updated base of second n/w b20= ",b20)

**Output:**



**PRACTICAL NO: 05 (A)**

**Title:** Write a program for Hopfield Network.

**Source Code:**

#include "hop.h"  
neuron::neuron(int \*j)  
{

inti;  
for(i=0;i<4;i++)  
{

weightv[i]= \*(j+i);  
}

}  
int neuron::act(int m, int \*x)  
{

inti;  
int a=0;  
for(i=0;i<m;i++)  
{  
a += x[i]\*weightv[i];  
}  
return a;

}  
int network::threshld(int k)  
{

if(k>=0)  
return (1);  
else  
return (0);

}  
network::network(int a[4],int b[4],int c[4],int d[4])  
{  
 nrn[0] = neuron(a) ;  
 nrn[1] = neuron(b) ;  
 nrn[2] = neuron(c) ;  
 nrn[3] = neuron(d) ;  
}

void network::activation(int \*patrn)  
{  
 int i,j;  
 for(i=0;i<4;i++)  
 {  
 for(j=0;j<4;j++)  
 {  
 cout<<"\n nrn["<<i<<"].weightv["<<j<<"] is "  
 <<nrn[i].weightv[j];  
 }  
 nrn[i].activation = nrn[i].act(4,patrn);  
 cout<<"\nactivation is "<<nrn[i].activation;

output[i]=threshld(nrn[i].activation);  
 cout<<"\noutput value is "<<output[i]<<"\n";  
 }  
}

void main ()  
{  
 int patrn1[]= {1,0,1,0},i;  
 int wt1[]= {0,-3,3,-3};  
 int wt2[]= {-3,0,-3,3};  
 int wt3[]= {3,-3,0,-3};  
 int wt4[]= {-3,3,-3,0};  
 cout<<"\nTHIS PROGRAM IS FOR A HOPFIELD NETWORK WITH A SINGLE LAYER OF";  
 cout<<"\n4 FULLY INTERCONNECTED NEURONS. THE NETWORK SHOULD RECALLTHE";  
 cout<<"\nPATTERNS 1010 AND 0101 CORRECTLY.\n";  
 //create the network by calling its constructor.  
 // the constructor calls neuron constructor as many times as thenumber of  
 // neurons in the network.

network h1(wt1,wt2,wt3,wt4);  
 //present a pattern to the network and get the activations of theneurons  
 h1.activation(patrn1);  
 //check if the pattern given is correctly recalled and give message  
 for(i=0;i<4;i++)  
 {  
 if (h1.output[i] == patrn1[i])  
 cout<<"\n pattern= "<<patrn1[i]<<  
 " output = "<<h1.output[i]<<" component matches";  
 else  
 cout<<"\n pattern= "<<patrn1[i]<<  
 " output = "<<h1.output[i]<<  
 " discrepancy occurred";  
 }  
 cout<<"\n\n";  
 int patrn2[]= {0,1,0,1};  
 h1.activation(patrn2);  
 for(i=0;i<4;i++)  
 {  
 if (h1.output[i] == patrn2[i])  
 cout<<"\n pattern= "<<patrn2[i]<<  
 " output = "<<h1.output[i]<<" component matches";  
 else  
 cout<<"\n pattern= "<<patrn2[i]<<  
 " output = "<<h1.output[i]<<  
 " discrepancy occurred";  
 }  
}  
======== End code of main program=============

//Hop.h  
//Single layer Hopfield Network with 4 neurons  
#include <stdio.h>  
#include <iostream.h>  
#include <math.h>  
class neuron  
{  
 protected:  
 int activation;  
 friend class network;  
 public:  
 int weightv[4];  
 neuron() {};  
 neuron(int \*j) ;  
 int act(int, int\*);  
};  
class network  
{  
 public:  
 neuron nrn[4];  
 int output[4];  
 int threshld(int) ;  
 void activation(int j[4]);  
 network(int\*,int\*,int\*,int\*);  
};

**PRACTICAL NO: 05 (B)**

**Title:** Write a program for Radial Basis function

**Source Code:**

from scipy import \*

from scipy.linalg import norm, pinv

from matplotlib import pyplot as plt

class RBF:

def \_\_init\_\_(self, indim, numCenters, outdim):

self.indim =indim

self.outdim =outdim

self.numCenters =numCenters

self.centers =[random.uniform(-1, 1, indim) for i in range(numCenters)]

self.beta =8

self.W =random.random((self.numCenters, self.outdim))

def \_basisfunc(self, c, d):

assert len(d) ==self.indim

return exp(-self.beta \*norm(c-d)\*\*2)

def \_calcAct(self, X):

# calculate activations of RBFs

G =zeros((X.shape[0], self.numCenters), float)

for ci, c in enumerate(self.centers):

for xi, x in enumerate(X):

G[xi,ci] =self.\_basisfunc(c, x)

return G

def train(self, X, Y):

""" X: matrix of dimensions n x indim

y: column vector of dimension n x 1 """

# choose random center vectors from training set

rnd\_idx =random.permutation(X.shape[0])[:self.numCenters]

self.centers =[X[i,:] for i in rnd\_idx]

print("center", self.centers)

# calculate activations of RBFs

G =self.\_calcAct(X)

print (G)

# calculate output weights (pseudoinverse)

self.W =dot(pinv(G), Y)

def test(self, X):

""" X: matrix of dimensions n x indim """

G =self.\_calcAct(X)

Y =dot(G, self.W)

return Y

if \_\_name\_\_ =='\_\_main\_\_':

# ----- 1D Example ------------------------------------------------

n =100

x =mgrid[-1:1:complex(0,n)].reshape(n, 1)

# set y and add random noise

y =sin(3\*(x+0.5)\*\*3-1)

# y += random.normal(0, 0.1, y.shape)

# rbf regression

rbf = RBF(1, 10, 1)

rbf.train(x, y)

z = rbf.test(x)

# plot original data

plt.figure(figsize=(12, 8))

plt.plot(x, y, 'k-')

# plot learned model

plt.plot(x, z, 'r-', linewidth=2)

# plot rbfs

plt.plot(rbf.centers, zeros(rbf.numCenters), 'gs')

for c in rbf.centers:

# RF prediction lines

cx = arange(c - 0.7, c + 0.7, 0.01)

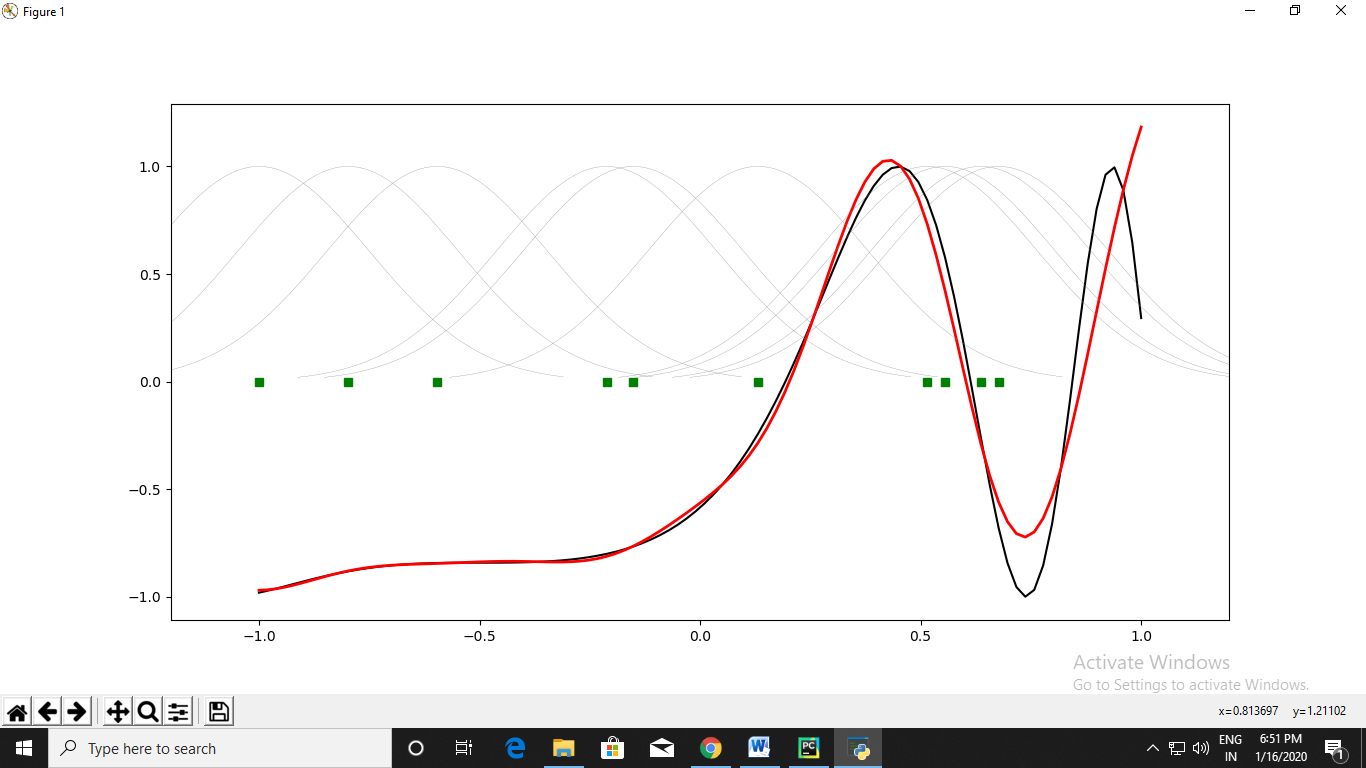
cy = [rbf.\_basisfunc(array([cx\_]), array([c])) for cx\_ in cx]

plt.plot(cx, cy, '-', color='gray', linewidth=0.2)

plt.xlim(-1.2, 1.2)

plt.show()

**Output:**



**PRACTICAL NO: 06 (A)**

**Title:** Write a program for Kohonen Self-Organizing map

**Source Code:**

from mvpa2.suite import \*

colors = np.array(  
 [[0., 0., 0.],  
 [0., 0., 1.],  
 [0., 0., 0.5],  
 [0.125, 0.529, 1.0],  
 [0.33, 0.4, 0.67],  
 [0.6, 0.5, 1.0],  
 [0., 1., 0.],  
 [1., 0., 0.],  
 [0., 1., 1.],  
 [1., 0., 1.],  
 [1., 1., 0.],  
 [1., 1., 1.],  
 [.33, .33, .33],  
 [.5, .5, .5],  
 [.66, .66, .66]])

*# store the names of the colors for visualization later on*

color\_names = \

['black', 'blue', 'darkblue', 'skyblue',

'greyblue', 'lilac', 'green', 'red',

'cyan', 'violet', 'yellow', 'white',

'darkgrey', 'mediumgrey', 'lightgrey']

som = SimpleSOMMapper((20, 30), 400, learning\_rate=0.05)

som.train(colors)

pl.imshow(som.K, origin='lower')

mapped = som(colors)

pl.title('Color SOM')

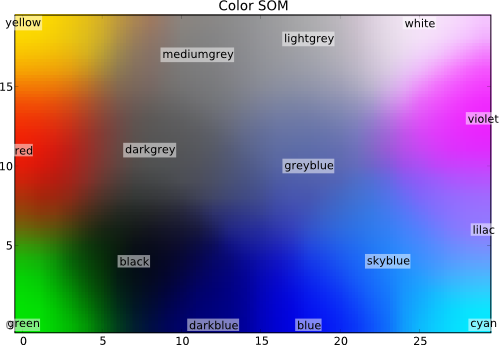
*# SOM's kshape is (rows x columns), while matplotlib wants (X x Y)*

for i, m **in** enumerate(mapped):

pl.text(m[1], m[0], color\_names[i], ha='center', va='center',

bbox=dict(facecolor='white', alpha=0.5, lw=0))

**Output:**



**PRACTICAL NO: 06 (B)**

**Title:** Write a program for Adaptive Resonance Theory

**Source Code:**

from \_\_future\_\_ import division  
  
import numpy as np  
  
from neupy.utils import format\_data  
from neupy.core.properties import (ProperFractionProperty,  
 IntProperty)  
from neupy.algorithms.base import BaseNetwork  
  
\_\_all\_\_ = ('ART1',)  
  
  
class ART1(BaseNetwork):  
 *"""  
 Adaptive Resonance Theory (ART1) Network for binary  
 data clustering.  
  
 Notes  
 -----  
 - Weights are not random, so the result will be  
 always reproduceble.  
  
 Parameters  
 ----------  
 rho : float  
 Control reset action in training process. Value must be  
 between ``0`` and ``1``, defaults to ``0.5``.  
  
 n\_clusters : int  
 Number of clusters, defaults to ``2``. Min value is also ``2``.  
  
 {BaseNetwork.Parameters}  
  
 Methods  
 -------  
 train(X)  
 ART trains until all clusters are found.  
  
 predict(X)  
 Each prediction trains a new network. It's an alias to  
 the ``train`` method.  
  
 {BaseSkeleton.fit}  
  
 Examples  
 --------  
 >>> import numpy as np  
 >>> from neupy import algorithms  
 >>>  
 >>> data = np.array([  
 ... [0, 1, 0],  
 ... [1, 0, 0],  
 ... [1, 1, 0],  
 ... ])  
 >>>>  
 >>> artnet = algorithms.ART1(  
 ... step=2,  
 ... rho=0.7,  
 ... n\_clusters=2,  
 ... verbose=False  
 ... )  
 >>> artnet.predict(data)  
 array([ 0., 1., 1.])  
 """* rho = ProperFractionProperty(default=0.5)  
 n\_clusters = IntProperty(default=2, minval=2)  
  
 def train(self, X):  
 X = format\_data(X)  
  
 if X.ndim != 2:  
 raise ValueError("Input value must be 2 dimensional, got "  
 "{}".format(X.ndim))  
  
 n\_samples, n\_features = X.shape  
 n\_clusters = self.n\_clusters  
 step = self.step  
 rho = self.rho  
  
 if np.any((X != 0) & (X != 1)):  
 raise ValueError("ART1 Network works only with binary matrices")  
  
 if not hasattr(self, 'weight\_21'):  
 self.weight\_21 = np.ones((n\_features, n\_clusters))  
  
 if not hasattr(self, 'weight\_12'):  
 scaler = step / (step + n\_clusters - 1)  
 self.weight\_12 = scaler \* self.weight\_21.T  
  
 weight\_21 = self.weight\_21  
 weight\_12 = self.weight\_12  
  
 if n\_features != weight\_21.shape[0]:  
 raise ValueError("Input data has invalid number of features. "  
 "Got {} instead of {}"  
 "".format(n\_features, weight\_21.shape[0]))  
  
 classes = np.zeros(n\_samples)  
  
 *# Train network* for i, p in enumerate(X):  
 disabled\_neurons = []  
 reseted\_values = []  
 reset = True  
  
 while reset:  
 output1 = p  
 input2 = np.dot(weight\_12, output1.T)  
  
 output2 = np.zeros(input2.size)  
 input2[disabled\_neurons] = -np.inf  
 winner\_index = input2.argmax()  
 output2[winner\_index] = 1  
  
 expectation = np.dot(weight\_21, output2)  
 output1 = np.logical\_and(p, expectation).astype(int)  
  
 reset\_value = np.dot(output1.T, output1) / np.dot(p.T, p)  
 reset = reset\_value < rho  
  
 if reset:  
 disabled\_neurons.append(winner\_index)  
 reseted\_values.append((reset\_value, winner\_index))  
  
 if len(disabled\_neurons) >= n\_clusters:  
 *# Got this case only if we test all possible clusters* reset = False  
 winner\_index = None  
  
 if not reset:  
 if winner\_index is not None:  
 weight\_12[winner\_index, :] = (step \* output1) / (  
 step + np.dot(output1.T, output1) - 1  
 )  
 weight\_21[:, winner\_index] = output1  
 else:  
 *# Get result with the best `rho`* winner\_index = max(reseted\_values)[1]  
  
 classes[i] = winner\_index  
  
 return classes  
  
 def predict(self, X):  
 return self.train(X)

**PRACTICAL NO: 06 (C)**

**Title:** Implementation of cross validation with suitable example

**Source Code in R:**

***#*** *tidyverse for easy data manipulation and visualization*

*# caret for easily computing cross-validation methods*

library(tidyverse)

library(caret)

*# Load the data*

data("swiss")

*# Inspect the data*

sample\_n(swiss, 3)

*# Split the data into training and test set*

set.seed(123)

training.samples <- swiss$Fertility %>%

createDataPartition(p = 0.8, list = FALSE)

train.data <- swiss[training.samples, ]

test.data <- swiss[-training.samples, ]

*# Build the model*

model <- lm(Fertility ~., data = train.data)

*# Make predictions and compute the R2, RMSE and MAE*

predictions <- model %>% predict(test.data)

data.frame( R2 = R2(predictions, test.data$Fertility),

RMSE = RMSE(predictions, test.data$Fertility),

MAE = MAE(predictions, test.data$Fertility))

**Leave one out cross validation – LOOCV**

*# Define training control*

train.control <- trainControl(method = "LOOCV")

*# Train the model*

model <- train(Fertility ~., data = swiss, method = "lm",

trControl = train.control)

*# Summarize the results*

print(model)

### **K-fold cross-validation**

*# Define training control*

set.seed(123)

train.control <- trainControl(method = "cv", number = 10)

*# Train the model*

model <- train(Fertility ~., data = swiss, method = "lm",

trControl = train.control)

*# Summarize the results*

print(model)

### **Repeated K-fold cross-validation**

# Define training control

set.seed(123)

train.control <- trainControl(method = "repeatedcv",

number = 10, repeats = 3)

# Train the model

model <- train(Fertility ~., data = swiss, method = "lm",

trControl = train.control)

# Summarize the results

print(model)

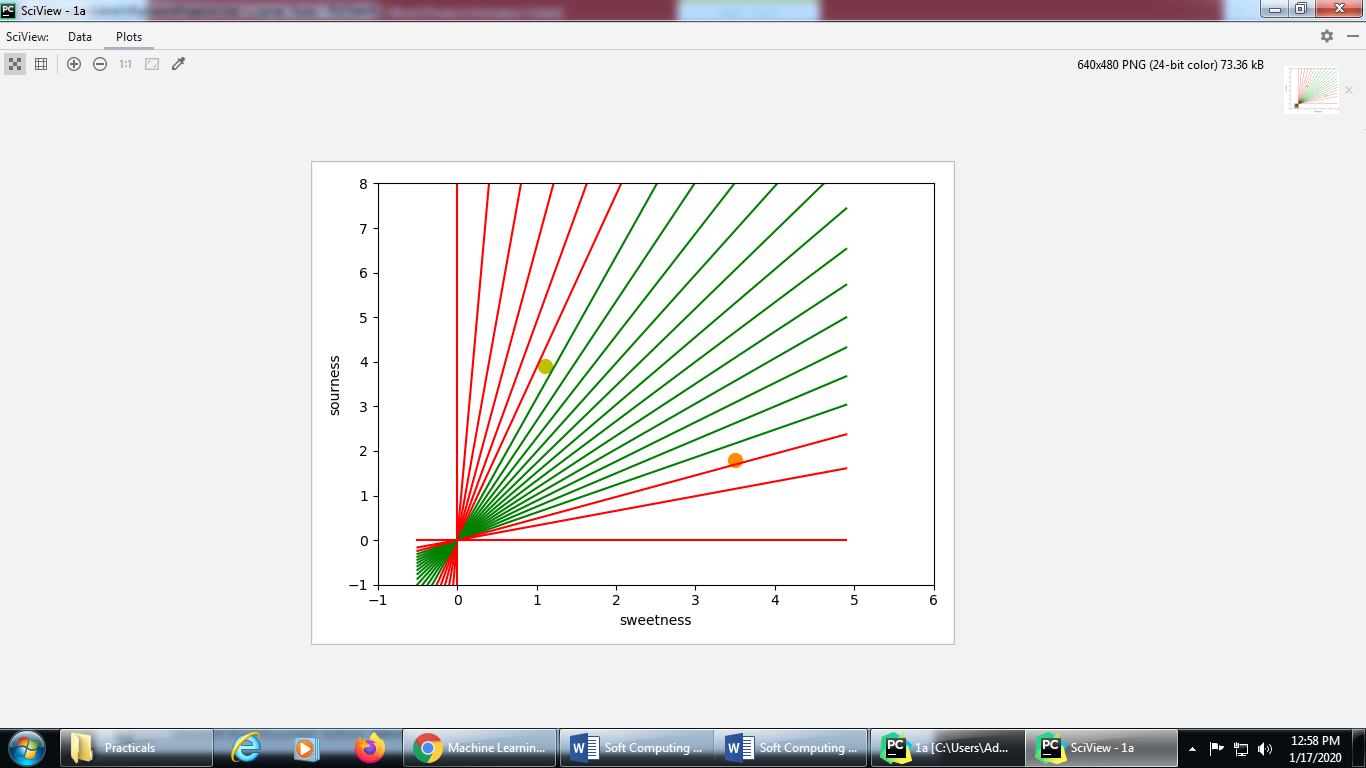
**PRACTICAL NO: 07 (A)**

**Title:** Write a program for line separation

**Source Code:**

import numpy as np  
import matplotlib.pyplot as plt  
  
def create\_distance\_function(a, b, c):  
 *""" 0 = ax + by + c """* def distance(x, y):  
 *""" returns tuple (d, pos)  
 d is the distance  
 If pos == -1 point is below the line,  
 0 on the line and +1 if above the line  
 """* nom = a \* x + b \* y + c  
 if nom == 0:  
 pos = 0  
 elif (nom < 0 and b < 0) or (nom > 0 and b > 0):  
 pos = -1  
 else:  
 pos = 1  
 return (np.absolute(nom) / np.sqrt(a \*\* 2 + b \*\* 2), pos)  
 return distance  
  
points = [(3.5, 1.8), (1.1, 3.9)]  
  
fig, ax = plt.subplots()  
ax.set\_xlabel("sweetness")  
ax.set\_ylabel("sourness")  
ax.set\_xlim([-1, 6])  
ax.set\_ylim([-1, 8])  
X = np.arange(-0.5, 5, 0.1)  
  
colors = ["r", ""] *# for the samples*size = 10  
for (index, (x, y)) in enumerate(points):  
 if index == 0:  
 ax.plot(x, y, "o",  
 color="darkorange",  
 markersize=size)  
 else:  
 ax.plot(x, y, "oy",  
 markersize=size)  
  
step = 0.05  
for x in np.arange(0, 1 + step, step):  
 slope = np.tan(np.arccos(x))  
 dist4line1 = create\_distance\_function(slope, -1, 0)  
 *# print("x: ", x, "slope: ", slope)* Y = slope \* X  
  
 results = []  
 for point in points:  
 results.append(dist4line1(\*point))  
 *# print(slope, results)* if (results[0][1] != results[1][1]):  
 ax.plot(X, Y, "g-")  
 else:  
 ax.plot(X, Y, "r-")  
  
plt.show()

**Output:**



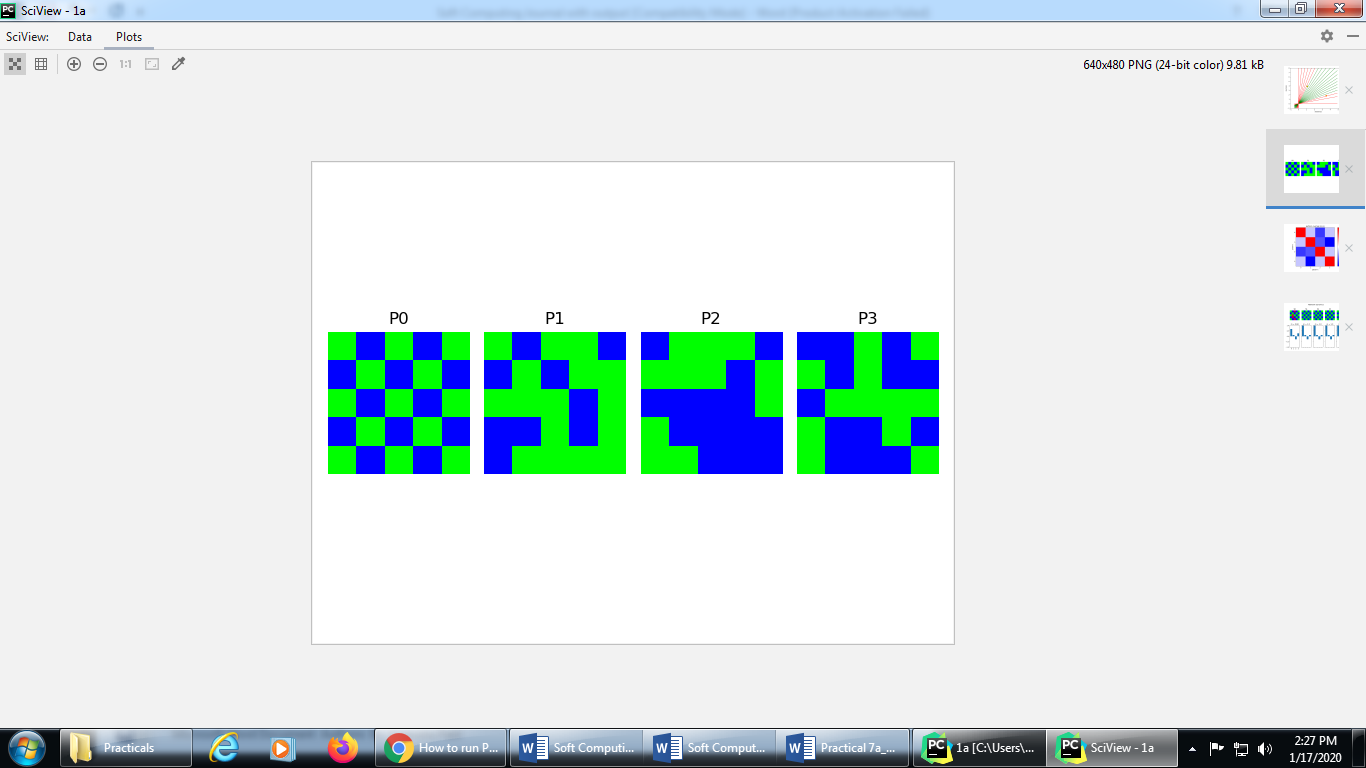
**PRACTICAL NO: 07(B)**

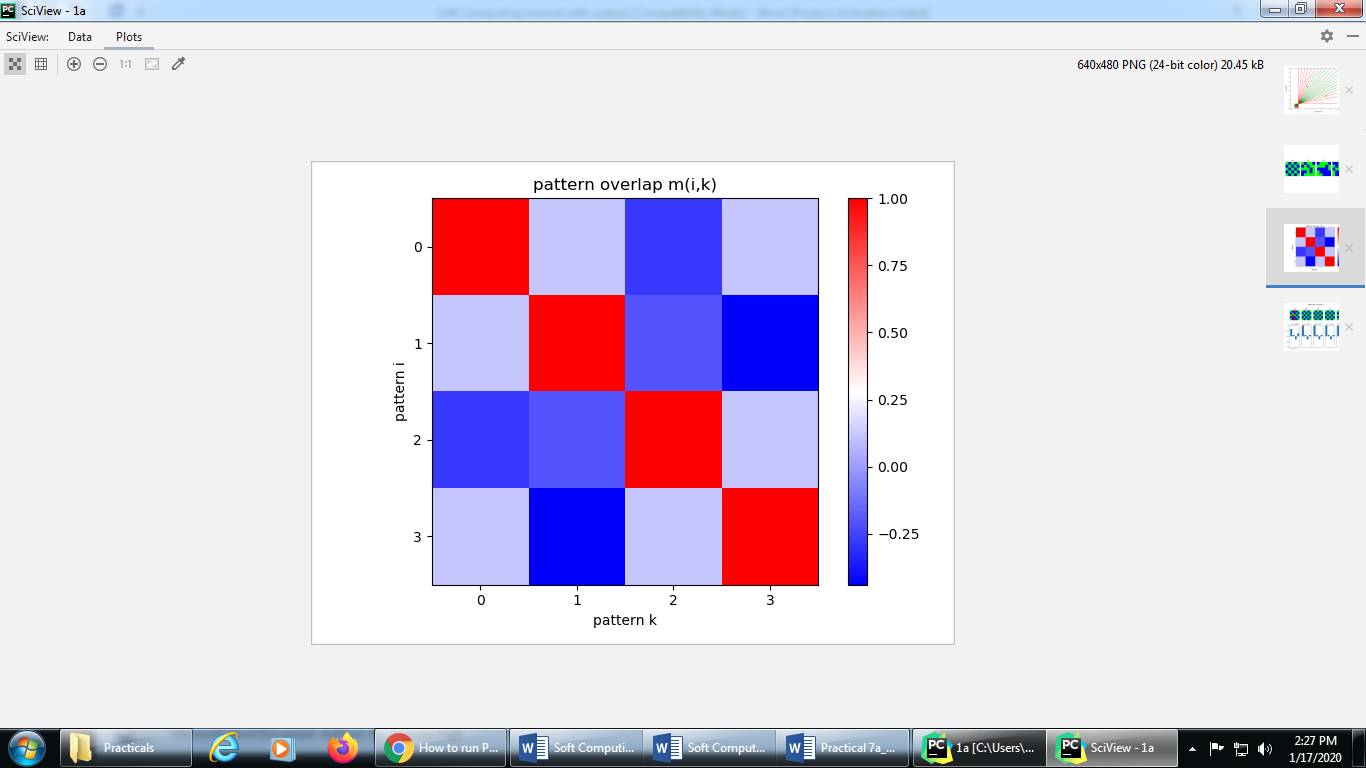
**Title:** Write a program for Hopfield network model of associative memory

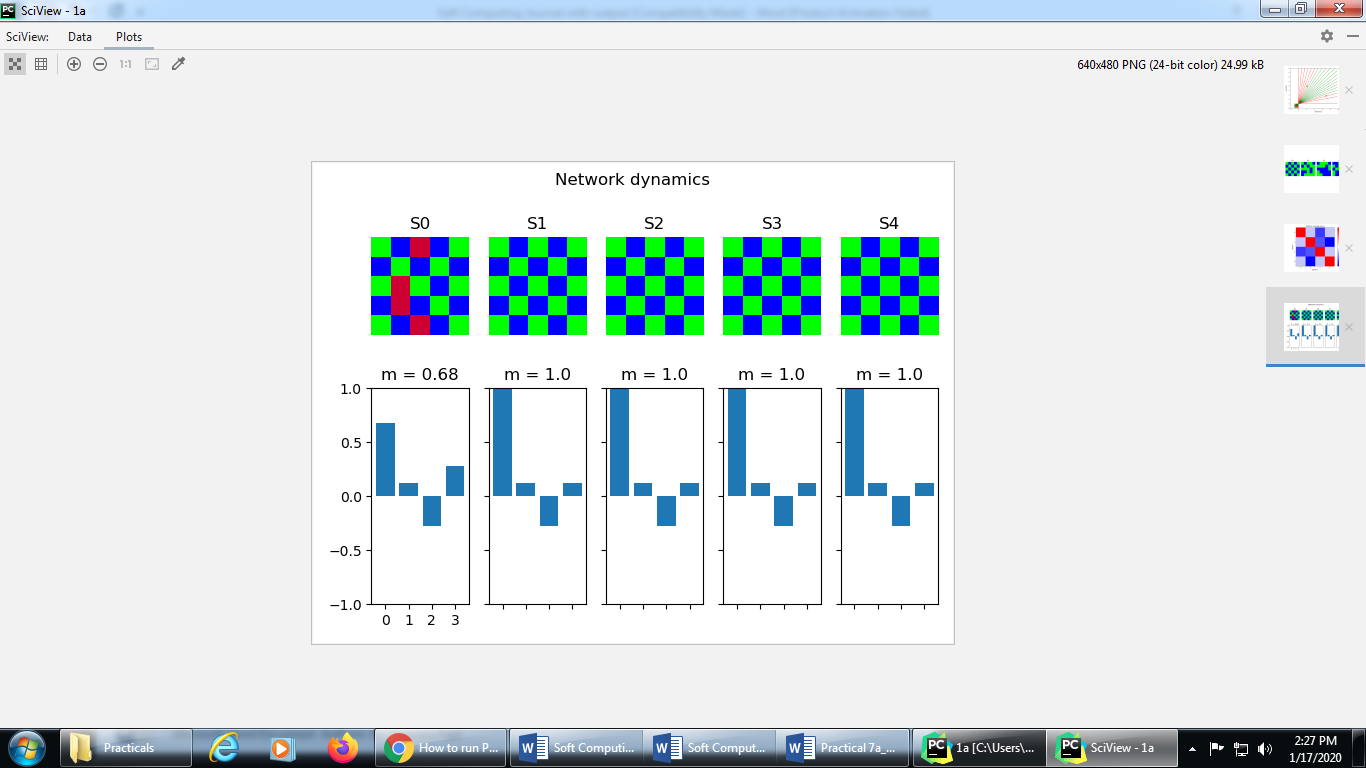
**Source Code:**

import matplotlib  
  
from neurodynex.hopfield\_network import network, pattern\_tools, plot\_tools  
  
pattern\_size = 5  
  
*# create an instance of the class HopfieldNetwork*hopfield\_net = network.HopfieldNetwork(nr\_neurons= pattern\_size\*\*2)  
*# instantiate a pattern factory*factory = pattern\_tools.PatternFactory(pattern\_size, pattern\_size)  
*# create a checkerboard pattern and add it to the pattern list*checkerboard = factory.create\_checkerboard()  
pattern\_list = [checkerboard]  
  
*# add random patterns to the list*pattern\_list.extend(factory.create\_random\_pattern\_list(nr\_patterns=3, on\_probability=0.5))  
plot\_tools.plot\_pattern\_list(pattern\_list)  
*# how similar are the random patterns and the checkerboard? Check the overlaps*overlap\_matrix = pattern\_tools.compute\_overlap\_matrix(pattern\_list)  
plot\_tools.plot\_overlap\_matrix(overlap\_matrix)  
  
*# let the hopfield network "learn" the patterns. Note: they are not stored  
# explicitly but only network weights are updated !*hopfield\_net.store\_patterns(pattern\_list)  
  
*# create a noisy version of a pattern and use that to initialize the network*noisy\_init\_state = pattern\_tools.flip\_n(checkerboard, nr\_of\_flips=4)  
hopfield\_net.set\_state\_from\_pattern(noisy\_init\_state)  
  
*# from this initial state, let the network dynamics evolve.*states = hopfield\_net.run\_with\_monitoring(nr\_steps=4)  
  
*# each network state is a vector. reshape it to the same shape used to create the patterns.*states\_as\_patterns = factory.reshape\_patterns(states)  
*# plot the states of the network*plot\_tools.plot\_state\_sequence\_and\_overlap(states\_as\_patterns, pattern\_list, reference\_idx=0, suptitle="Network dynamics")

**Output:**







**PRACTICAL NO: 08 (A)**

**Title:** Implement the membership operator: in, not in

**Source Code:**

*# Aim: write a program to find the whether there is common member in two list*

*# Define a function() that takes two lists*

def overlapping(list1,list2):  
 c = len( list1 )  
 d = len( list2 )  
 print( "List 1: ", list1 )  
 print( "The length of List 1: ", c )  
 print( "List 2: ", list2 )  
 print( "The length of List 2: ", d )  
 for i in range(0,c):  
 for j in range(0,d):  
 if(list1[i]==list2[j]):  
 return 1  
 return 0  
*#end of the function*

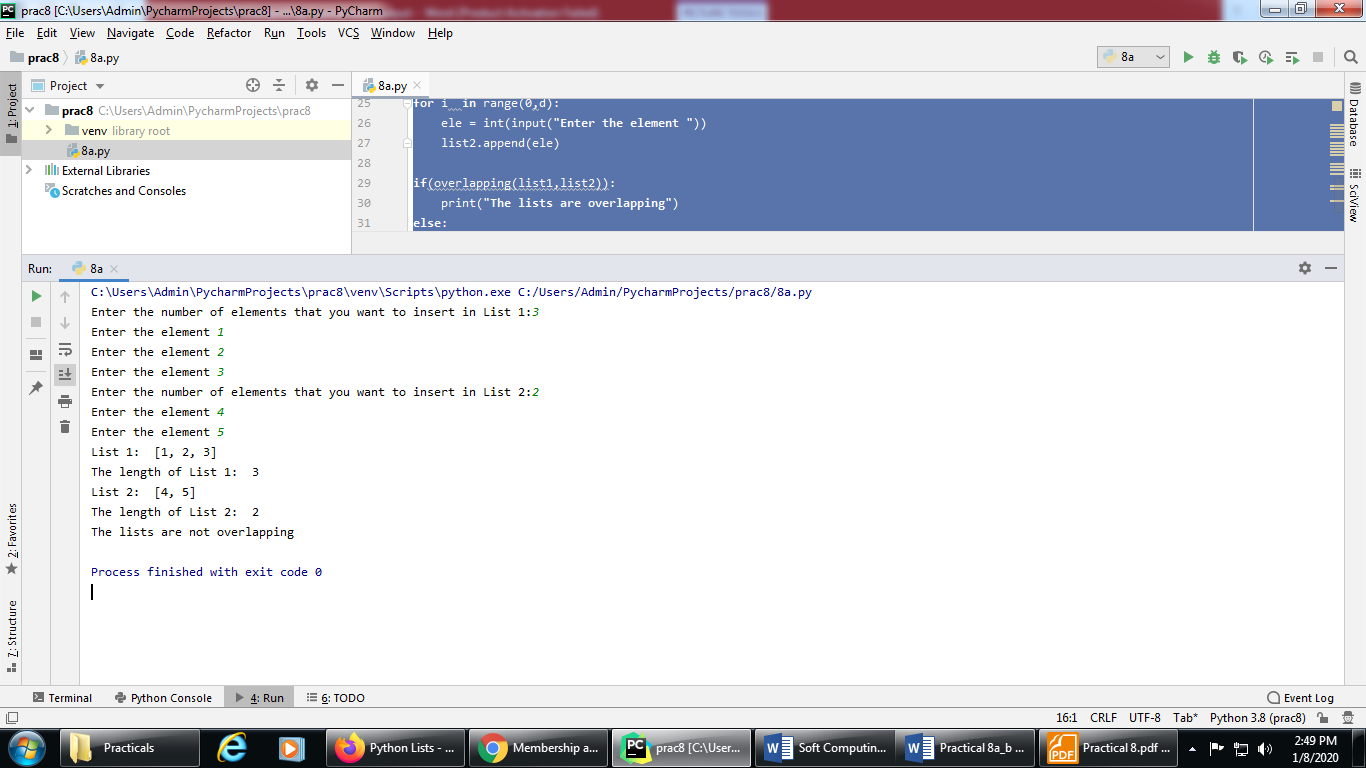
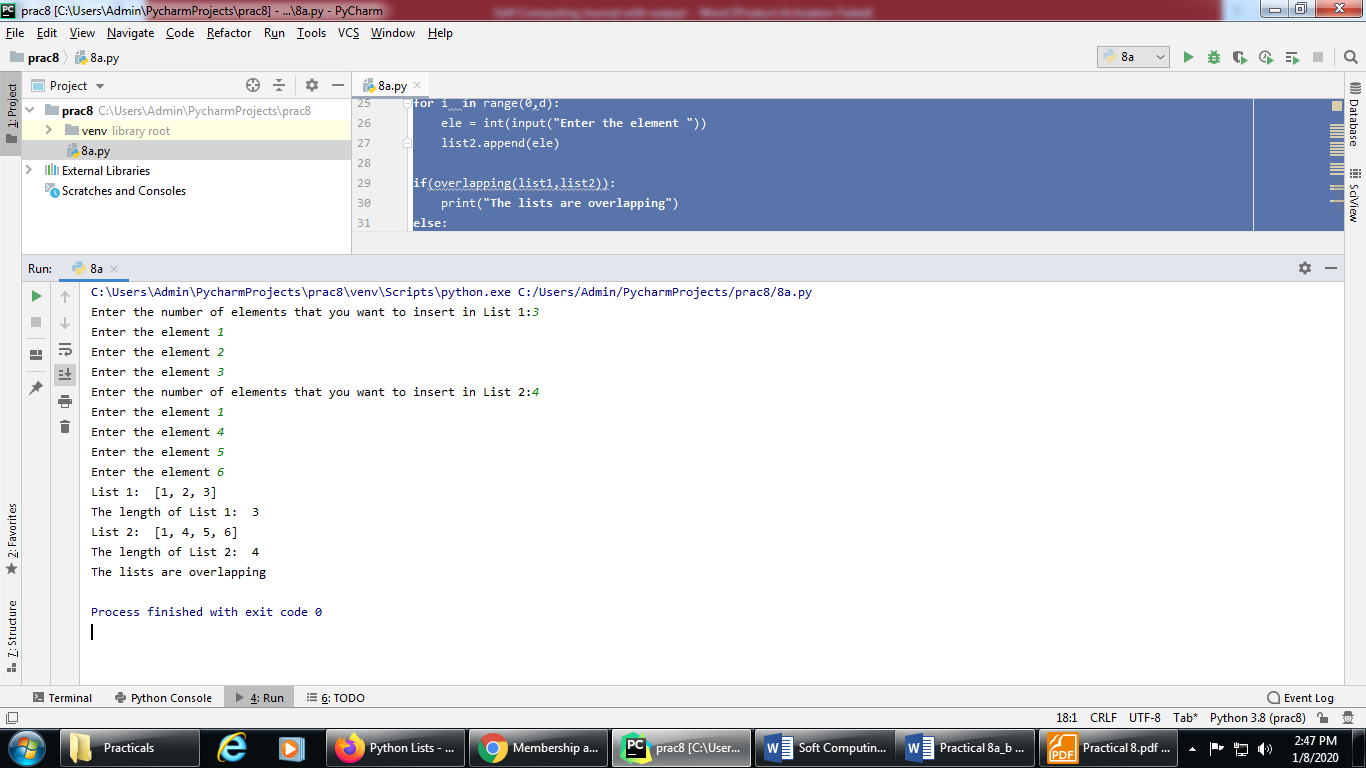
list1=[]  
list2=[]

c=int(input("Enter the number of elements that you want to insert in List 1:"))  
for i in range(0,c):  
 ele = int(input("Enter the element "))  
 list1.append(ele)

d=int(input("Enter the number of elements that you want to insert in List 2:"))  
for i in range(0,d):  
 ele = int(input("Enter the element "))  
 list2.append(ele)

if(overlapping(list1,list2)):  
 print("The lists are overlapping")  
else:  
 print("The lists are not overlapping")

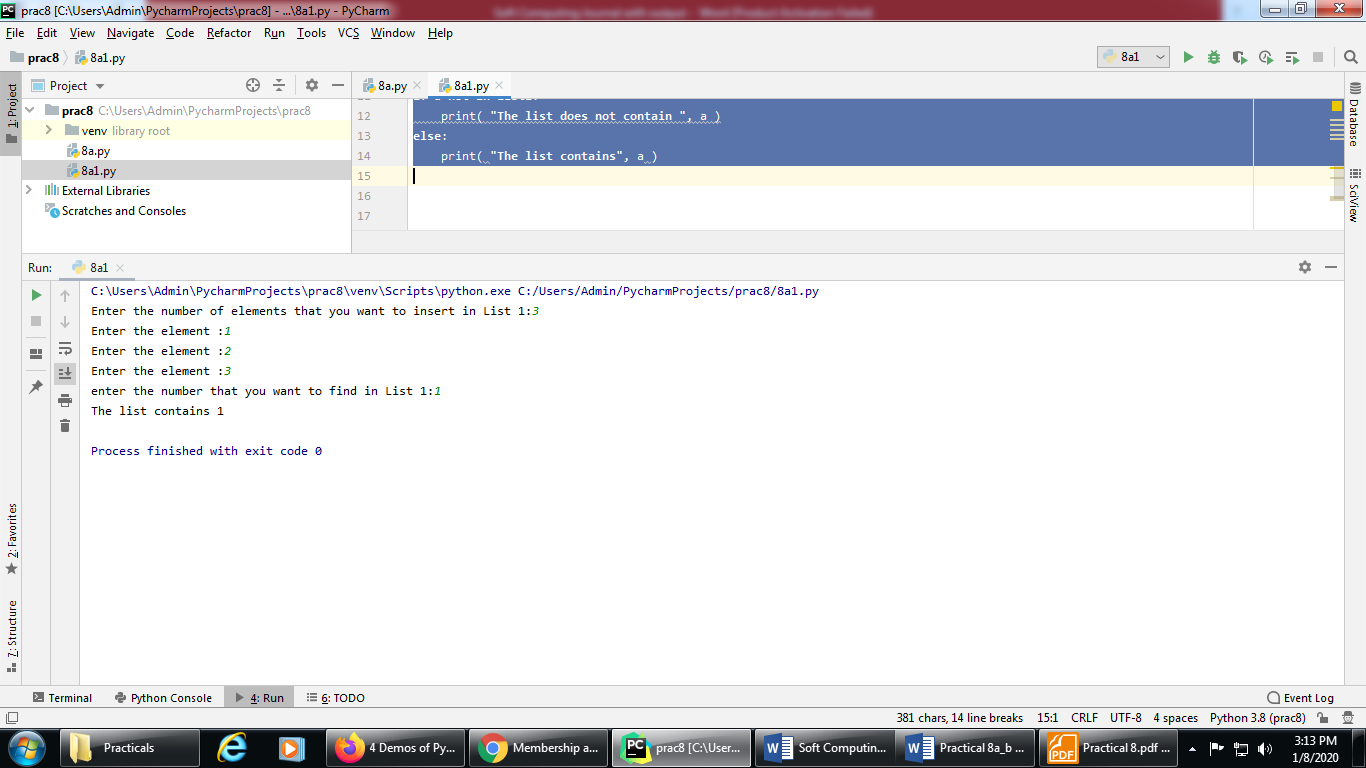
**Output**

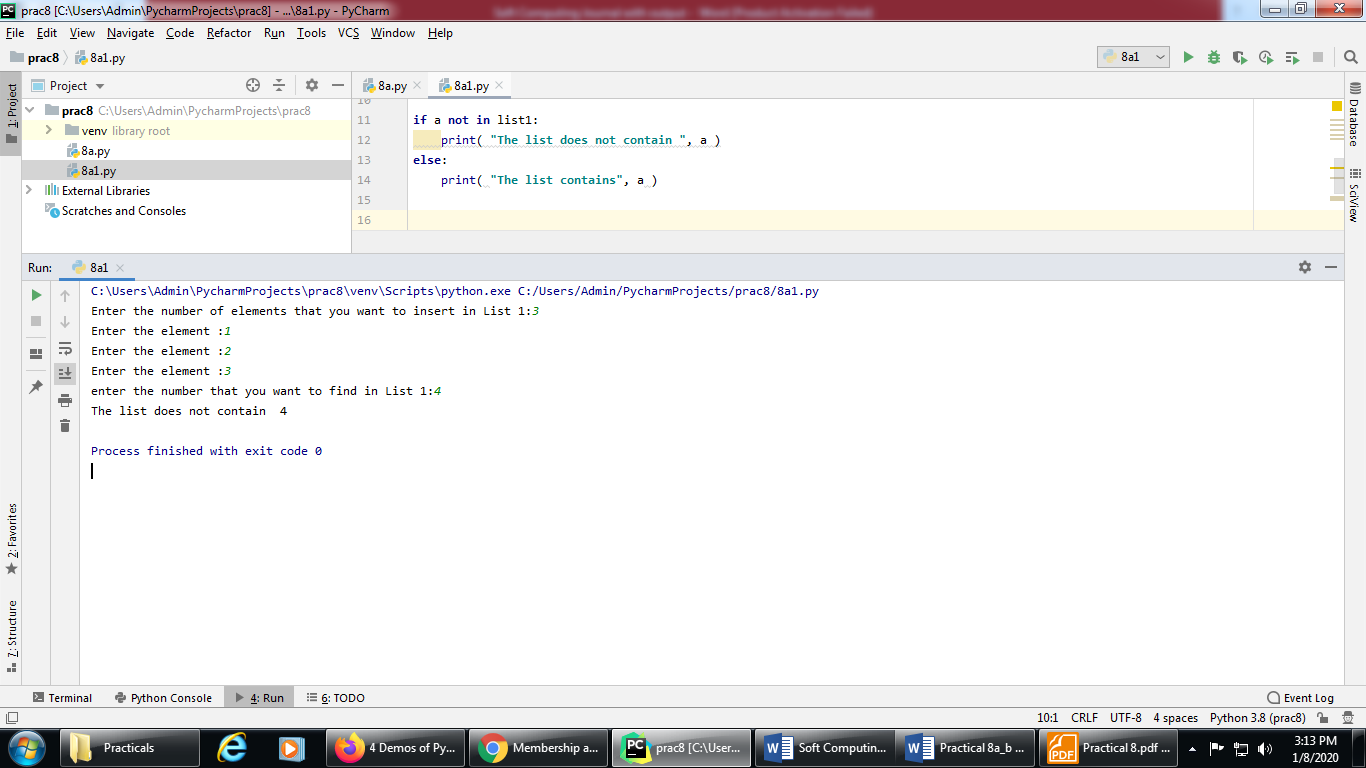


**Source code:**

*# Aim: Not in operator in python*list1=[]  
c=int(input("Enter the number of elements that you want to insert in List 1:"))  
for i in range(0,c):  
 ele = int(input("Enter the element :"))  
 list1.append(ele)  
  
a = int(input("enter the number that you want to find in List 1:"))  
  
  
if a not in list1:  
 print( "The list does not contain ", a )  
else:  
 print( "The list contains", a )

**Output:**





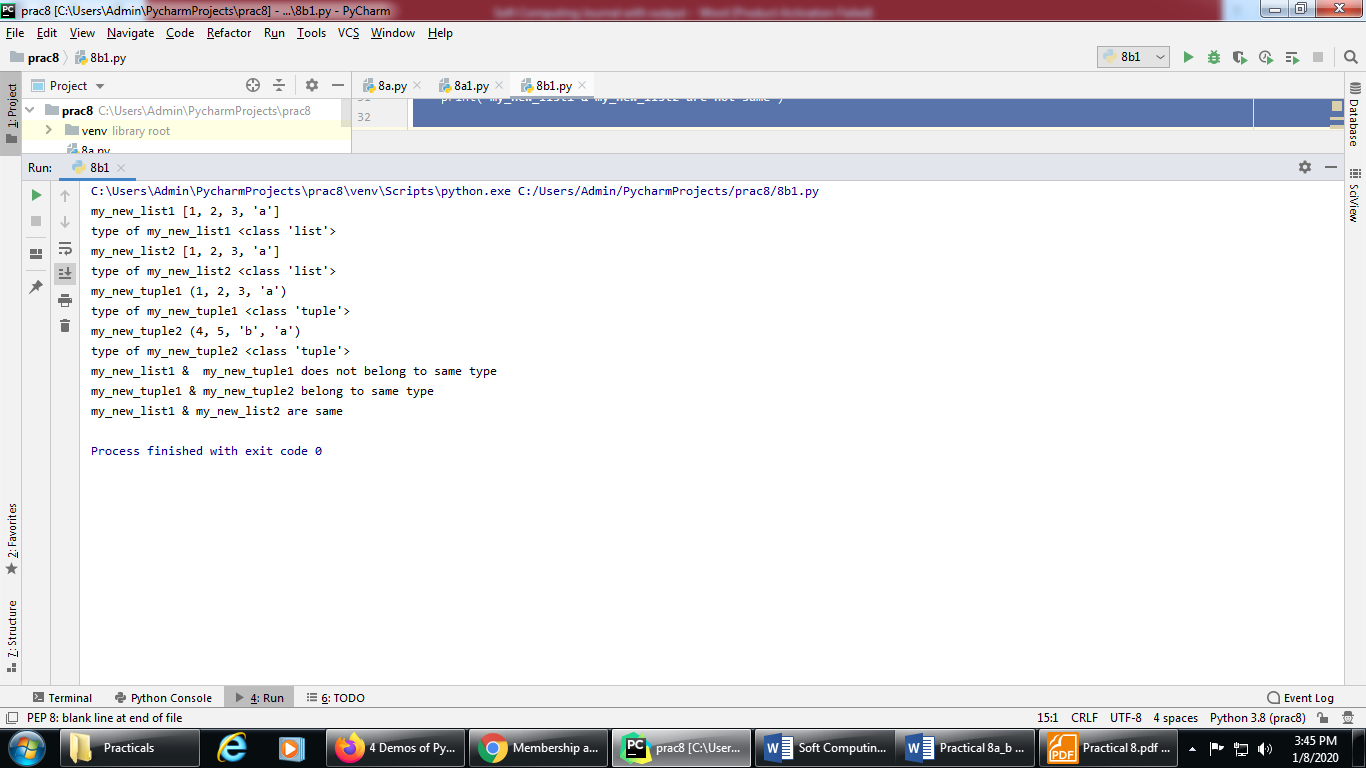
**PRACTICAL NO: 08 (B)**

**Title:** Implement the identity operator: is, is not

**Source Code:**

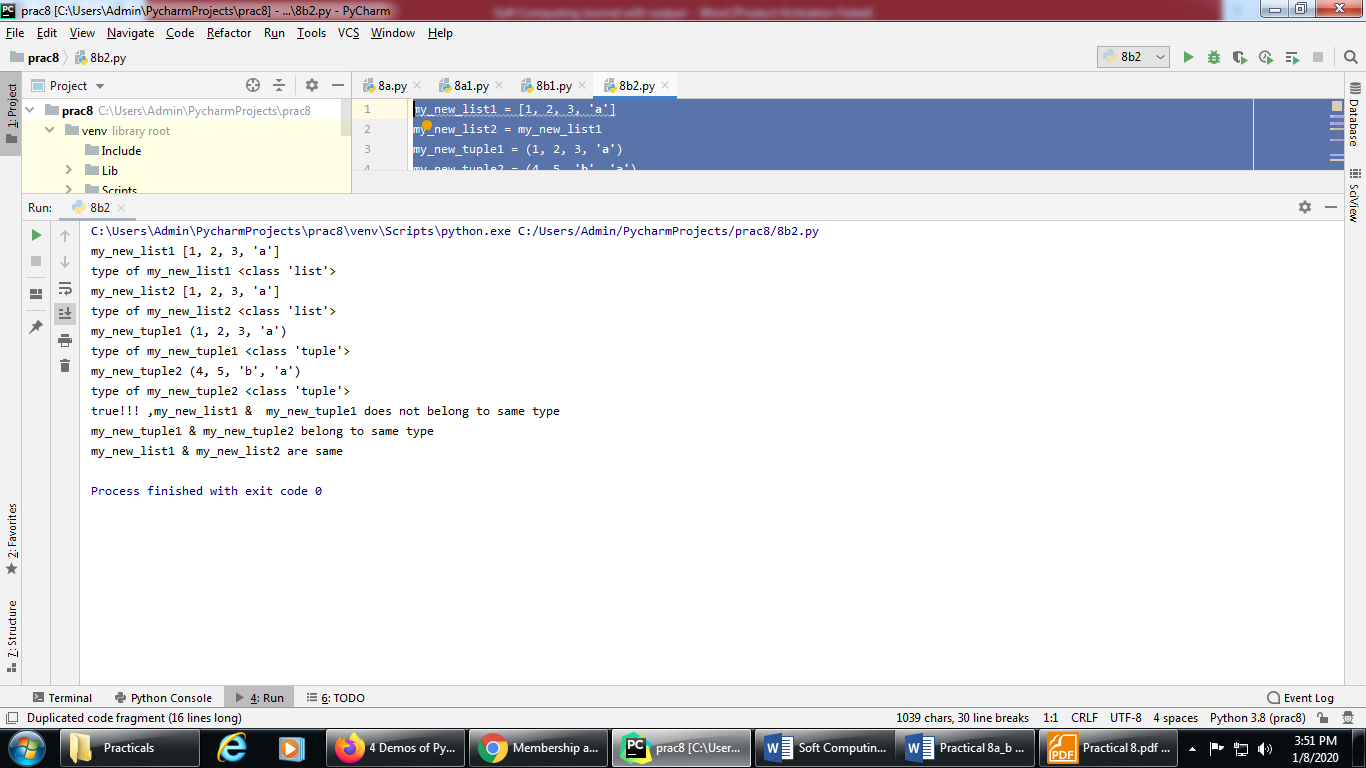
my\_new\_list1 = [1, 2, 3, 'a']  
my\_new\_list2 = my\_new\_list1  
my\_new\_tuple1 = (1, 2, 3, 'a')  
my\_new\_tuple2 = (4, 5, 'b', 'a')  
  
print("my\_new\_list1", my\_new\_list1)  
print("type of my\_new\_list1",type(my\_new\_list1))  
  
print("my\_new\_list2", my\_new\_list2)  
print("type of my\_new\_list2",type(my\_new\_list2))  
  
print("my\_new\_tuple1", my\_new\_tuple1)  
print("type of my\_new\_tuple1", type(my\_new\_tuple1))  
  
print("my\_new\_tuple2", my\_new\_tuple2)  
print("type of my\_new\_tuple2", type(my\_new\_tuple2))  
  
if type(my\_new\_list1) is type(my\_new\_tuple1):  
 print('my\_new\_list1 & my\_new\_tuple1 belong to same type')  
else:  
 print("my\_new\_list1 & my\_new\_tuple1 does not belong to same type")  
  
if type(my\_new\_tuple1) is type(my\_new\_tuple2):  
 print('my\_new\_tuple1 & my\_new\_tuple2 belong to same type')  
else:  
 print("my\_new\_tuple1 & my\_new\_tuple2 does not belong to same type")  
  
if (my\_new\_list1) is (my\_new\_list2):  
 print("my\_new\_list1 & my\_new\_list2 are same")  
else:  
 print("my\_new\_list1 & my\_new\_list2 are not same")

**Output:**



**Source Code:**

my\_new\_list1 = [1, 2, 3, 'a']  
my\_new\_list2 = my\_new\_list1  
my\_new\_tuple1 = (1, 2, 3, 'a')  
my\_new\_tuple2 = (4, 5, 'b', 'a')  
  
print("my\_new\_list1", my\_new\_list1)  
print("type of my\_new\_list1",type(my\_new\_list1))  
  
print("my\_new\_list2", my\_new\_list2)  
print("type of my\_new\_list2",type(my\_new\_list2))  
  
print("my\_new\_tuple1", my\_new\_tuple1)  
print("type of my\_new\_tuple1", type(my\_new\_tuple1))  
  
print("my\_new\_tuple2", my\_new\_tuple2)  
print("type of my\_new\_tuple2", type(my\_new\_tuple2))  
  
if type(my\_new\_list1) is not type(my\_new\_tuple1):  
 print('true!!! ,my\_new\_list1 & my\_new\_tuple1 does not belong to same type')  
else:  
 print("my\_new\_list1 & my\_new\_tuple1 belong to same type")  
  
if type(my\_new\_tuple1) is not type(my\_new\_tuple2):  
 print('true!!!, my\_new\_tuple1 & my\_new\_tuple2 does not belong to same type')  
else:  
 print("my\_new\_tuple1 & my\_new\_tuple2 belong to same type")  
  
if (my\_new\_list1) is not (my\_new\_list2):  
 print("true!!!, my\_new\_list1 & my\_new\_list2 are not same")  
else:  
 print("my\_new\_list1 & my\_new\_list2 are same")

**Output:**

**PRACTICAL NO: 09 (A)**

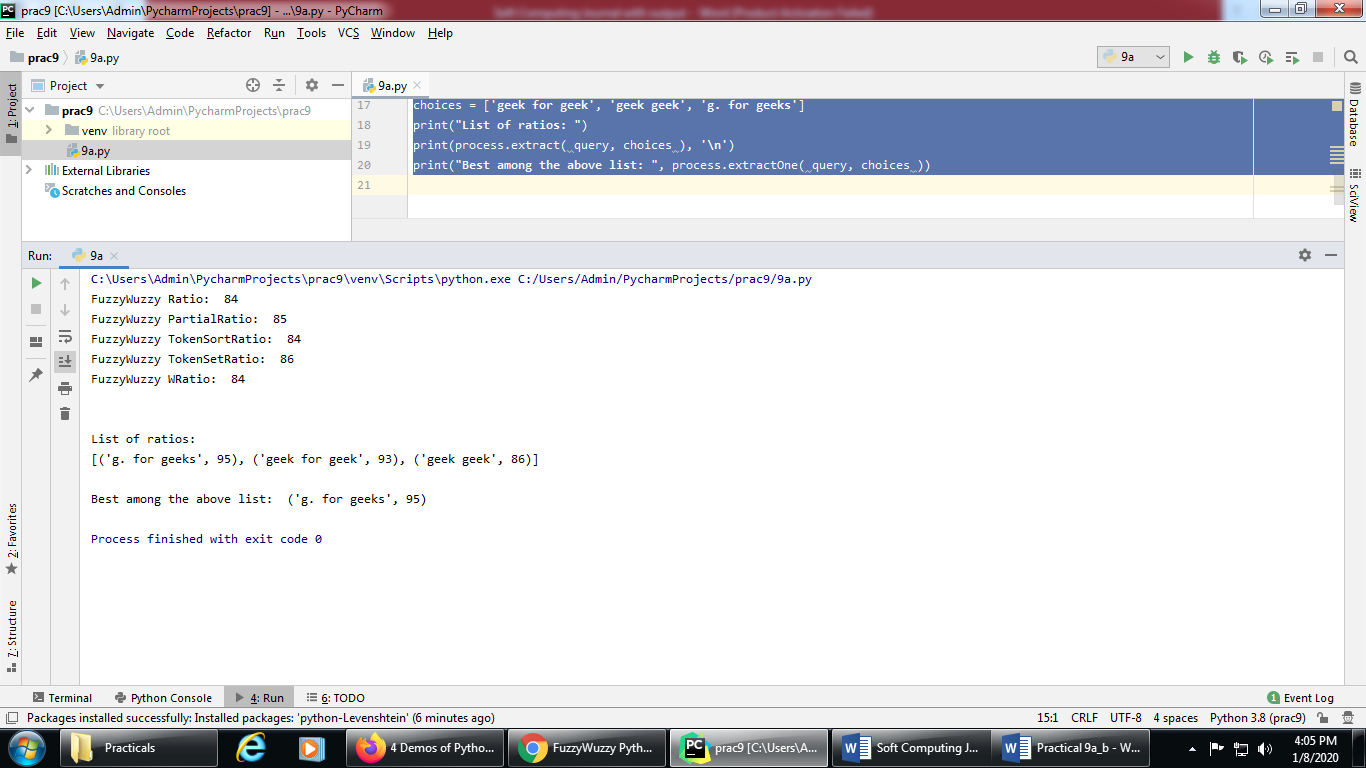
**Title:** Find the ratios using fuzzy logic

**Source Code:**

*pip install fuzzywuzzy*

*pip install python-Levenshtein*

*# Python code showing all the ratios together,  
# make sure you have installed fuzzywuzzy module*from fuzzywuzzy import fuzz  
from fuzzywuzzy import process  
  
s1 = "I love GeeksforGeeks"  
s2 = "I am loving GeeksforGeeks"  
print("FuzzyWuzzy Ratio: ", fuzz.ratio( s1, s2 ))  
print("FuzzyWuzzy PartialRatio: ", fuzz.partial\_ratio( s1, s2 ))  
print("FuzzyWuzzy TokenSortRatio: ", fuzz.token\_sort\_ratio( s1, s2 ))  
print("FuzzyWuzzy TokenSetRatio: ", fuzz.token\_set\_ratio( s1, s2 ))  
print("FuzzyWuzzy WRatio: ", fuzz.WRatio( s1, s2 ), '\n\n')  
  
*# for process library,*query = 'geeks for geeks'  
choices = ['geek for geek', 'geek geek', 'g. for geeks']  
print("List of ratios: ")  
print(process.extract( query, choices ), '\n')  
print("Best among the above list: ", process.extractOne( query, choices ))

**Output:**

**PRACTICAL NO: 09 (B)**

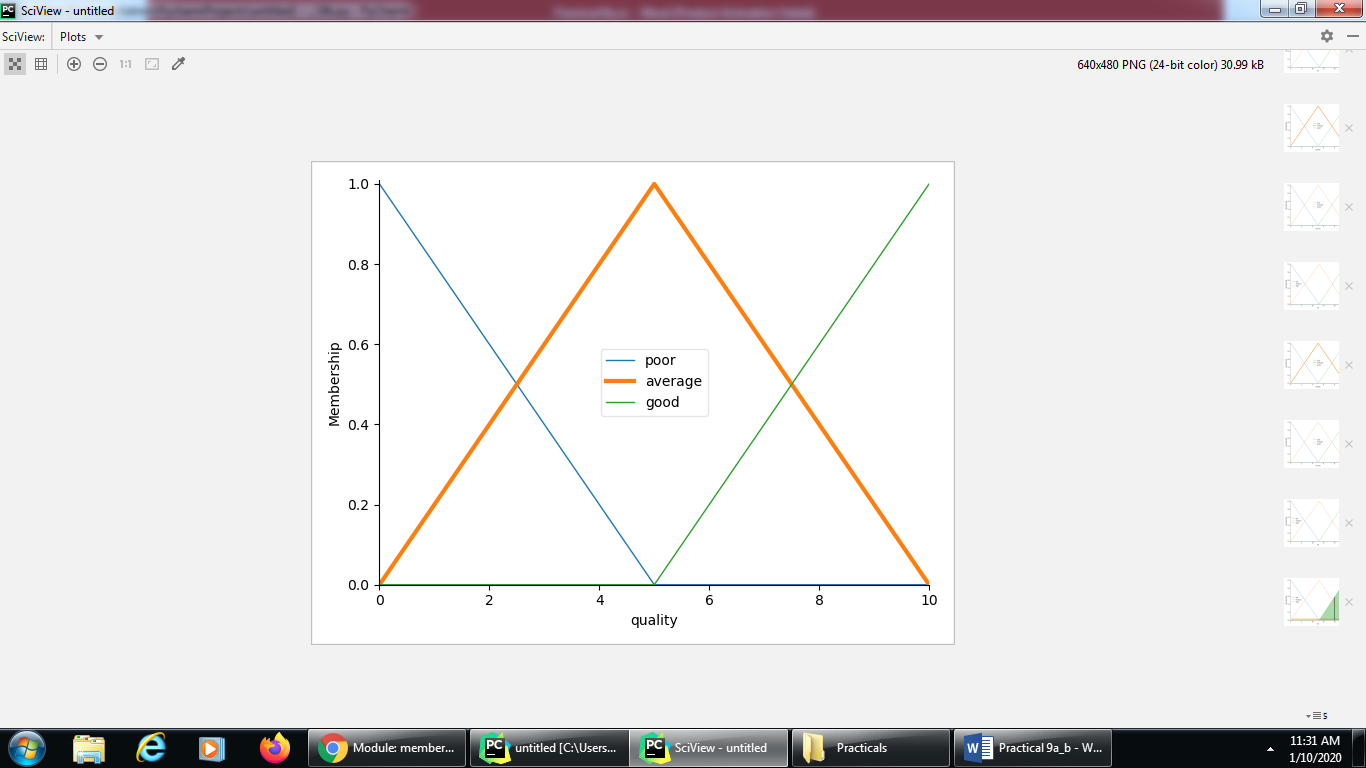
**Title:** Solve Tipping Problem using fuzzy logic

**Source Code:**

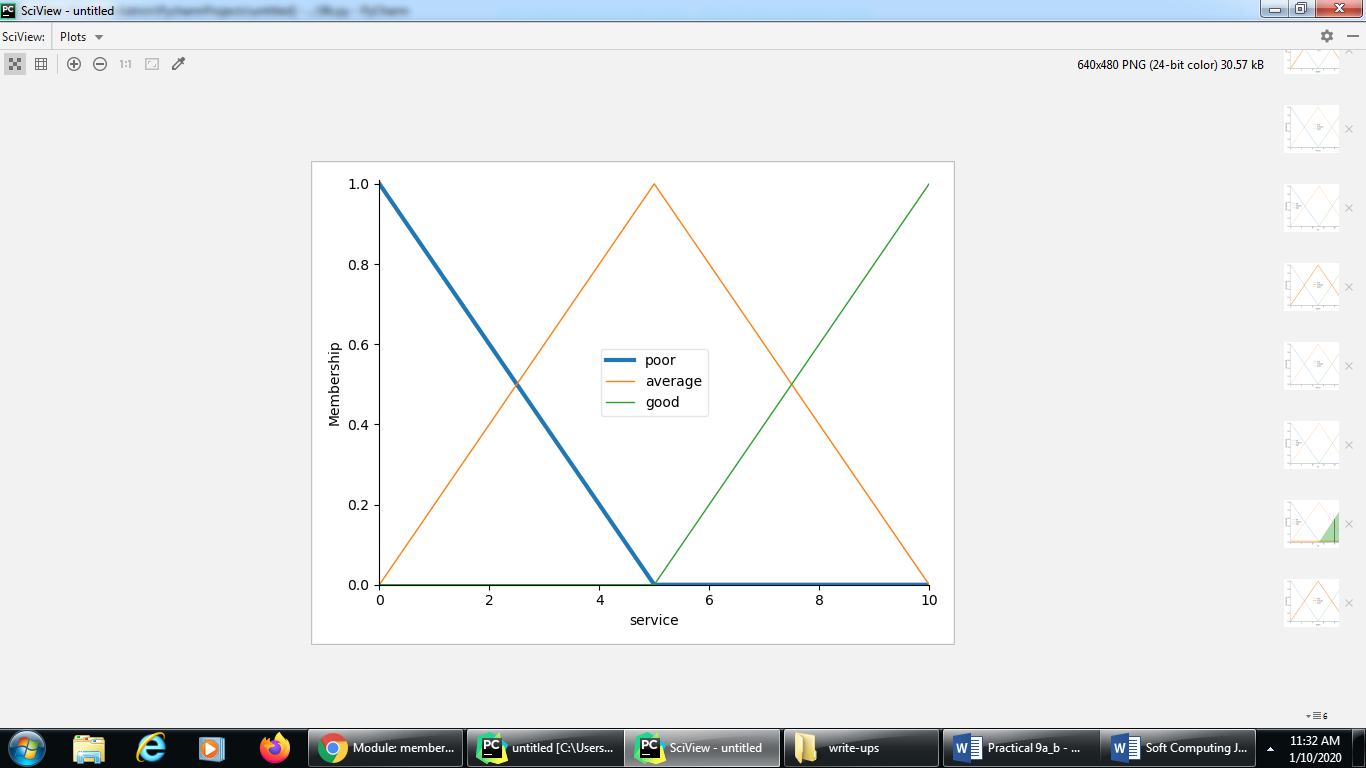
import numpy as np  
import skfuzzy as fuzz  
from skfuzzy import control as ctrl  
  
*# New Antecedent/Consequent objects hold universe variables and membership  
# functions  
#np.arange will give a array starting from zero to 11, with increment of 1*quality = ctrl.Antecedent(np.arange(0, 11, 1), 'quality')  
service = ctrl.Antecedent(np.arange(0, 11, 1), 'service')  
tip = ctrl.Consequent(np.arange(0, 26, 1), 'tip')  
  
*# Auto-membership function population is possible with .automf(3, 5, or 7)*quality.automf(3)  
service.automf(3)  
  
*# Custom membership functions can be built interactively with a familiar,  
# Pythonic API  
#trimf() is triangular membership generator*tip['low'] = fuzz.trimf(tip.universe, [0, 0, 13])  
tip['medium'] = fuzz.trimf(tip.universe, [0, 13, 25])  
tip['high'] = fuzz.trimf(tip.universe, [13, 25, 25])  
  
*# You can see how these look with .view()  
#quality['average'].view()  
#service['poor'].view()  
#tip['high'].view()  
  
#rule to implemented*rule1 = ctrl.Rule(quality['poor'] | service['poor'], tip['low'])  
rule2 = ctrl.Rule(service['average'], tip['medium'])  
rule3 = ctrl.Rule(service['good'] | quality['good'], tip['high'])  
rule1.view() *# rule2.view()  
# rule3.view()*tipping\_ctrl = ctrl.ControlSystem([rule1, rule2, rule3])  
tipping = ctrl.ControlSystemSimulation(tipping\_ctrl)  
  
*# Pass inputs to the ControlSystem using Antecedent labels with Pythonic API  
# Note: if you like passing many inputs all at once, use .inputs(dict\_of\_data)*tipping.input['quality'] = 6.5  
tipping.input['service'] = 9.8  
  
*# Crunch the numbers*tipping.compute()  
print (tipping.output['tip'])  
tip.view(sim=tipping)

**Output:**

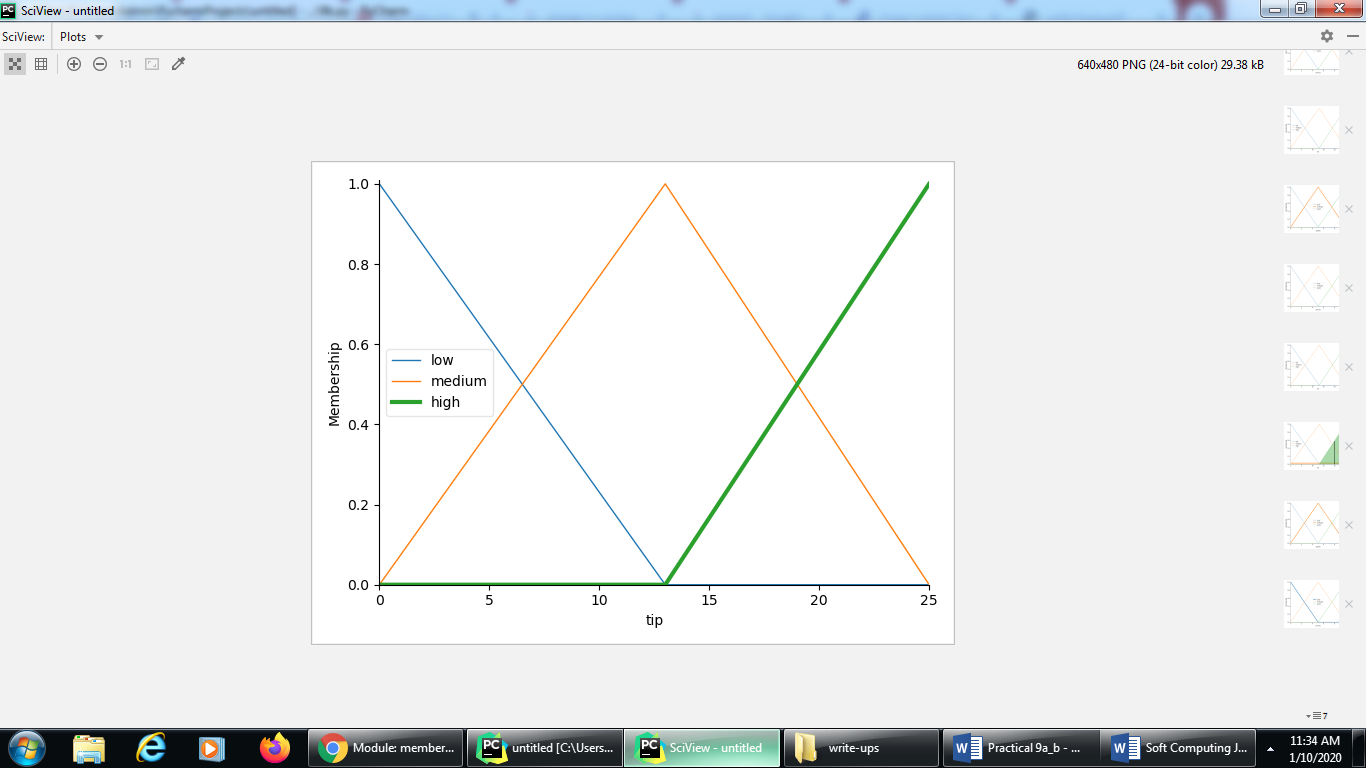
quality['average'].view()



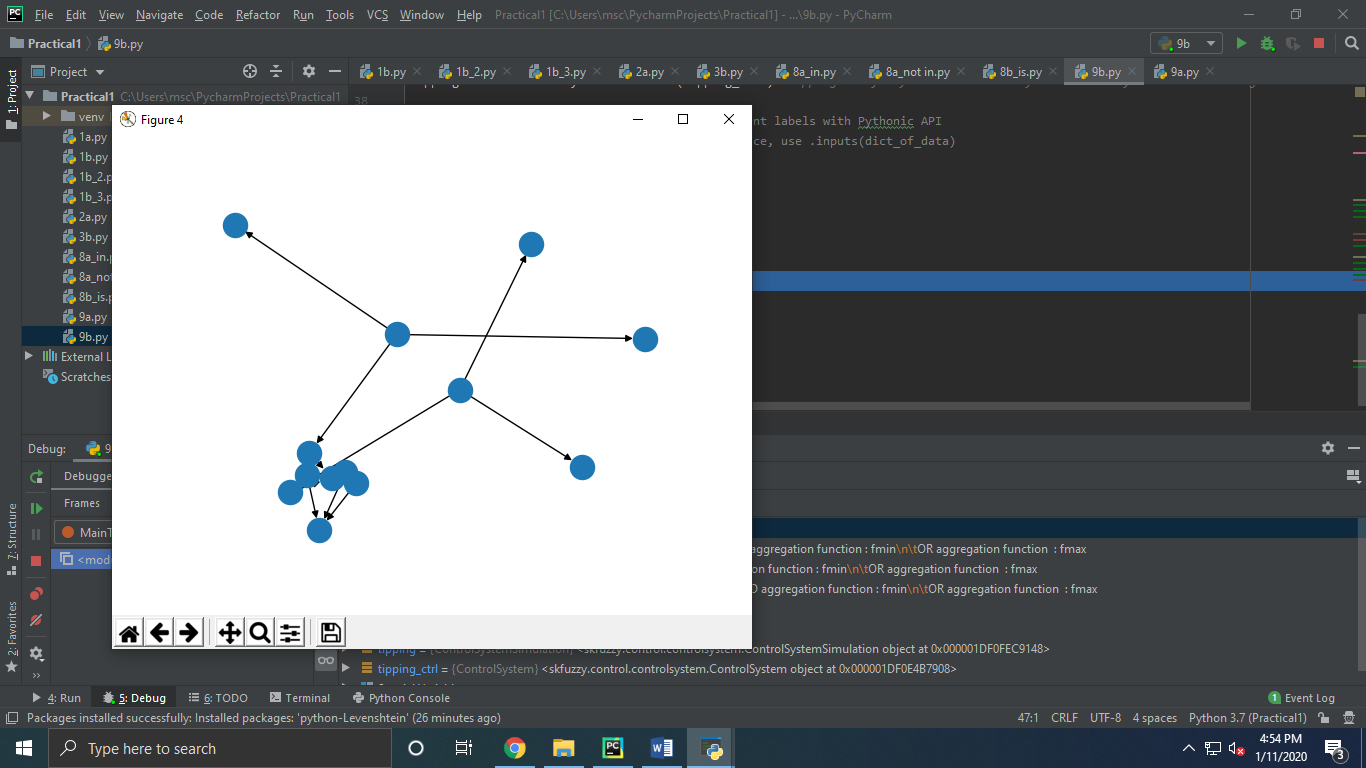
service['poor'].view()



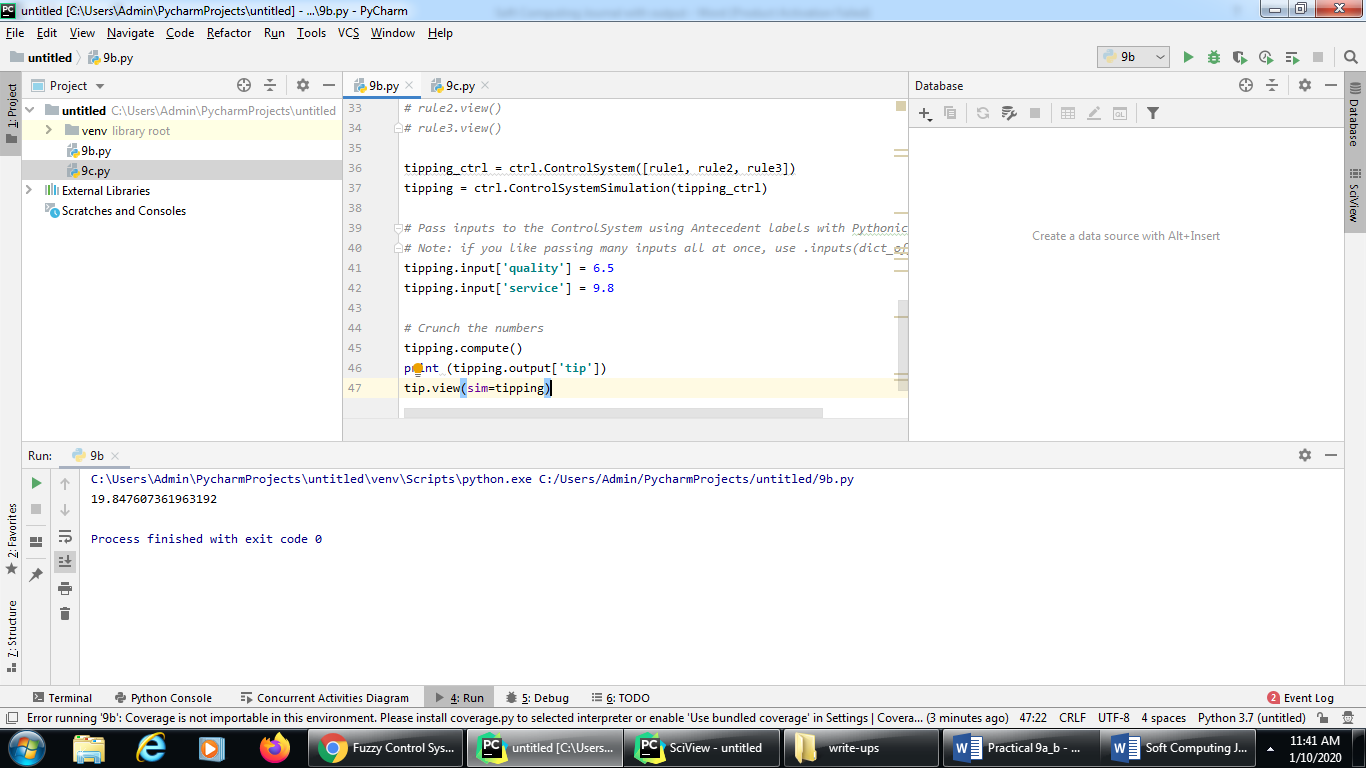
tip['high'].view()

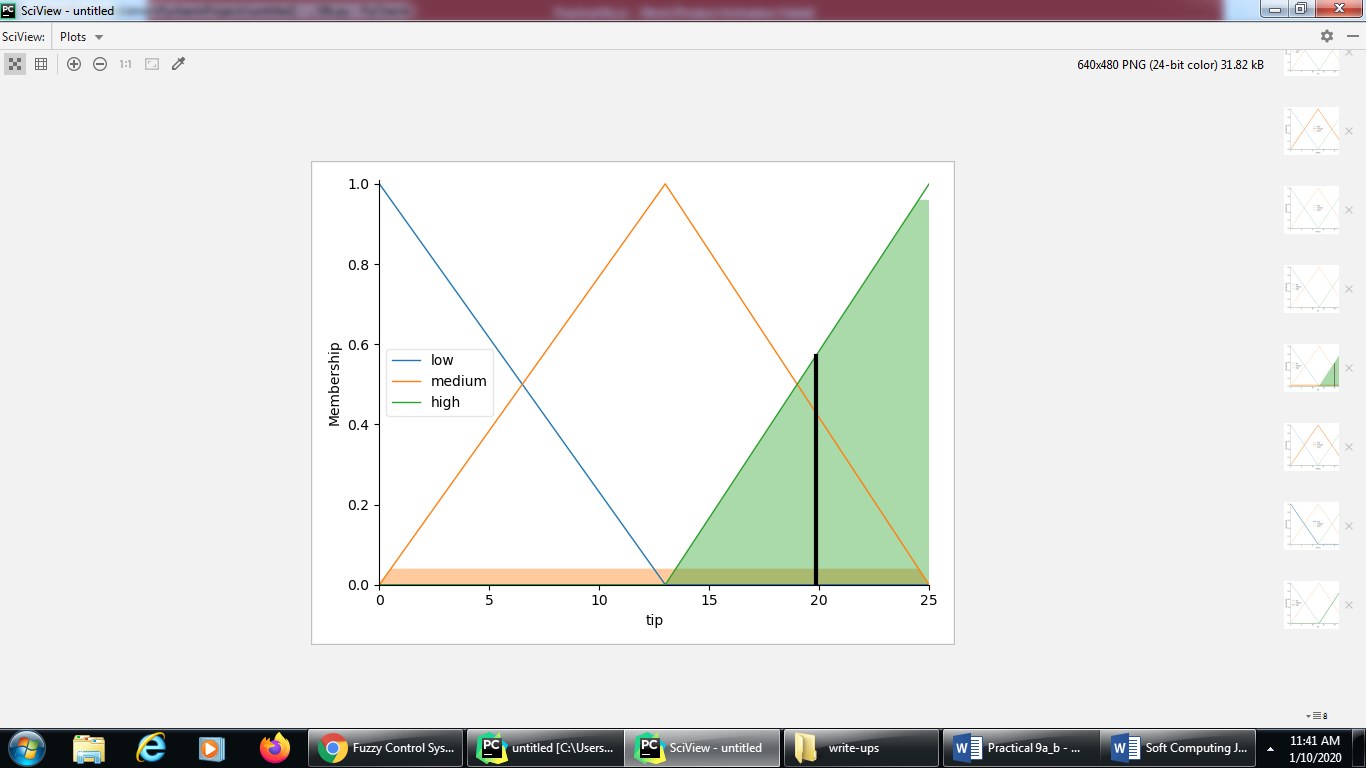


rule1.view()



print (tipping.output['tip'])  
tip.view(sim=tipping)





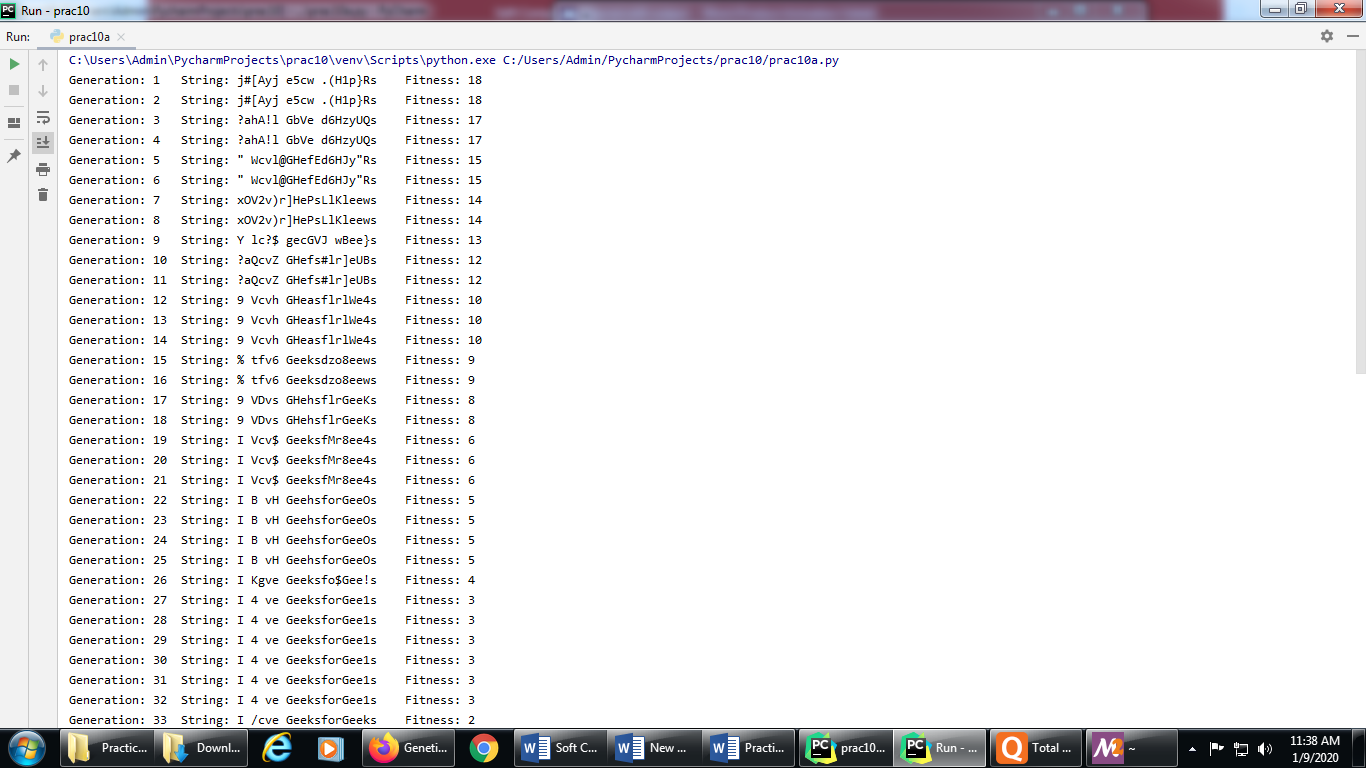
**PRACTICAL NO: 10 (A)**

**Title:** Implementation of simple genetic algorithm

**Source Code:**

import random  
  
*# Number of individuals in each generation*POPULATION\_SIZE = 100  
  
*# Valid genes*GENES = '''abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOP   
QRSTUVWXYZ 1234567890, .-;:\_!"#%&/()=?@${[]}'''  
  
*# Target string to be generated*TARGET = "I love GeeksforGeeks"  
  
  
class Individual( object ):  
 *'''  
 Class representing individual in population  
 '''* def \_\_init\_\_(self, chromosome):  
 self.chromosome = chromosome  
 self.fitness = self.cal\_fitness()  
  
 @classmethod  
 def mutated\_genes(self):  
 *'''  
 create random genes for mutation  
 '''* global GENES  
 gene = random.choice( GENES )  
 return gene  
  
 @classmethod  
 def create\_gnome(self):  
 *'''  
 create chromosome or string of genes  
 '''* global TARGET  
 gnome\_len = len( TARGET )  
 return [self.mutated\_genes() for \_ in range( gnome\_len )]  
  
 def mate(self, par2):  
 *'''  
 Perform mating and produce new offspring  
 '''  
  
 # chromosome for offspring* child\_chromosome = []  
 for gp1, gp2 in zip( self.chromosome, par2.chromosome ):  
  
 *# random probability* prob = random.random()  
  
 *# if prob is less than 0.45, insert gene  
 # from parent 1* if prob < 0.45:  
 child\_chromosome.append( gp1 )  
  
 *# if prob is between 0.45 and 0.90, insert  
 # gene from parent 2* elif prob < 0.90:  
 child\_chromosome.append( gp2 )  
  
 *# otherwise insert random gene(mutate),  
 # for maintaining diversity* else:  
 child\_chromosome.append( self.mutated\_genes() )  
  
 *# create new Individual(offspring) using  
 # generated chromosome for offspring* return Individual( child\_chromosome )  
  
 def cal\_fitness(self):  
 *'''  
 Calculate fittness score, it is the number of  
 characters in string which differ from target  
 string.  
 '''* global TARGET  
 fitness = 0  
 for gs, gt in zip( self.chromosome, TARGET ):  
 if gs != gt: fitness += 1  
 return fitness  
  
 *# Driver code*def main():  
 global POPULATION\_SIZE  
  
 *# current generation* generation = 1  
  
 found = False  
 population = []  
  
 *# create initial population* for \_ in range( POPULATION\_SIZE ):  
 gnome = Individual.create\_gnome()  
 population.append( Individual( gnome ) )  
  
 while not found:  
  
 *# sort the population in increasing order of fitness score* population = sorted( population, key=lambda x: x.fitness )  
  
 *# if the individual having lowest fitness score ie.  
 # 0 then we know that we have reached to the target  
 # and break the loop* if population[0].fitness <= 0:  
 found = True  
 break  
  
 *# Otherwise generate new offsprings for new generation* new\_generation = []  
  
 *# Perform Elitism, that mean 10% of fittest population  
 # goes to the next generation* s = int( (10 \* POPULATION\_SIZE) / 100 )  
 new\_generation.extend( population[:s] )  
  
 *# From 50% of fittest population, Individuals  
 # will mate to produce offspring* s = int( (90 \* POPULATION\_SIZE) / 100 )  
 for \_ in range( s ):  
 parent1 = random.choice( population[:50] )  
 parent2 = random.choice( population[:50] )  
 child = parent1.mate( parent2 )  
 new\_generation.append( child )  
  
 population = new\_generation  
  
 print( "Generation: {}\tString: {}\tFitness: {}". \  
 format( generation,  
 "".join( population[0].chromosome ),  
 population[0].fitness ) )  
  
 generation += 1  
  
 print( "Generation: {}\tString: {}\tFitness: {}". \  
 format( generation,  
 "".join( population[0].chromosome ),  
 population[0].fitness ) )  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

**Output:**



**PRACTICAL NO: 10 (B)**

**Title:** Solve Travelling salesman problem (TSP) using GA

**Source Code:**

*#Aim: Solve Travelling salesman problem (TSP) using GA*import numpy as np, random, operator, pandas as pd, matplotlib.pyplot as plt  
  
*# We first create a City class that will allow us to create and handle our cities.  
# These are simply our (x, y) coordinates. Within the City class,  
# we add a distance calculation (making use of the Pythagorean theorem)  
# and a cleaner way to output the cities as coordinates with \_\_repr\_\_.*class City:  
 def \_\_init\_\_(self, x, y):  
 self.x = x  
 self.y = y  
  
 def distance(self, city):  
 xDis = abs(self.x - city.x)  
 yDis = abs(self.y - city.y)  
 distance = np.sqrt((xDis \*\* 2) + (yDis \*\* 2))  
 return distance  
  
 def \_\_repr\_\_(self):  
 return "(" + str(self.x) + "," + str(self.y) + ")"  
  
*# We’ll also create a Fitness class. In our case, we’ll treat the fitness as the inverse of the #route distance.  
# We want to minimize route distance, so a larger fitness score is better.  
# Based on Rule #2, we need to start and end at the same place,  
# so this extra calculation is accounted for the distance calculation.*class Fitness:  
 def \_\_init\_\_(self, route):  
 self.route = route  
 self.distance = 0  
 self.fitness = 0.0  
  
 def routeDistance(self):  
 if self.distance == 0:  
 pathDistance = 0  
 for i in range(0, len(self.route)):  
 fromCity = self.route[i]  
 toCity = None  
 if i + 1 < len(self.route):  
 toCity = self.route[i + 1]  
 else:  
 toCity = self.route[0]  
 pathDistance += fromCity.distance(toCity)  
 self.distance = pathDistance  
 return self.distance  
  
 def routeFitness(self):  
 if self.fitness == 0:  
 self.fitness = 1 / float(self.routeDistance())  
 return self.fitness  
  
*#Creating the population:  
#We now can make our initial population (aka first generation).  
# To do so, we need a way to create a function that produces routes that satisfy our #conditions  
# To create an individual, we randomly select the order in which we visit each city:*def createRoute(cityList):  
 route = random.sample(cityList, len(cityList))  
 return route  
  
*#This produces one individual, but we want a full population, so let’s do that in our next #function.  
# This is as simple as looping through the createRoute function until  
# we have as many routes as we want for our population.*def initialPopulation(popSize, cityList):  
 population = []  
  
 for i in range(0, popSize):  
 population.append(createRoute(cityList))  
 return population  
  
*#Determine Fitness:  
#To simulate our “survival of the fittest”,  
# we can make use of Fitness to rank each individual in the population.  
# Our output will be an ordered list with the route IDs and each associated fitness score.*def rankRoutes(population):  
 fitnessResults = {}  
 for i in range(0,len(population)):  
 fitnessResults[i] = Fitness(population[i]).routeFitness()  
 return sorted(fitnessResults.items(), key = operator.itemgetter(1), reverse = True)  
  
*#Select the mating Pool:  
#For the purpose of clarity, we’ll create the mating pool in two steps.  
# First, we’ll use the output from rankRoutes to determine which routes to select in our #selection function.  
# We set up the roulette wheel by calculating a relative fitness weight for each individual.  
# We compare a randomly drawn number to these weights to select our mating pool.  
# We’ll also want to hold on to our best routes, so we introduce elitism.  
# Ultimately, the selection function returns a list of route IDs,  
# which we can use to create the mating pool in the matingPool function.*def selection(popRanked, eliteSize):  
 selectionResults = []  
 df = pd.DataFrame(np.array(popRanked), columns=["Index", "Fitness"])  
 df['cum\_sum'] = df.Fitness.cumsum()  
 df['cum\_perc'] = 100 \* df.cum\_sum / df.Fitness.sum()  
  
 for i in range(0, eliteSize):  
 selectionResults.append(popRanked[i][0])  
 for i in range(0, len(popRanked) - eliteSize):  
 pick = 100 \* random.random()  
 for i in range(0, len(popRanked)):  
 if pick <= df.iat[i, 3]:  
 selectionResults.append(popRanked[i][0])  
 break  
 return selectionResults  
  
*#Now that we have the IDs of the routes that will make up our mating pool from the #selection function,  
# we can create the mating pool. We’re simply extracting the selected individuals from our #population.*def matingPool(population, selectionResults):  
 matingpool = []  
 for i in range(0, len(selectionResults)):  
 index = selectionResults[i]  
 matingpool.append(population[index])  
 return matingpool  
  
*#Breed:  
#With our mating pool created, we can create the next generation in a process called crossover (aka “breeding”).  
#the TSP is unique in that we need to include all locations exactly one time.  
# To abide by this rule, we can use a special breeding function called ordered crossover.  
# In ordered crossover, we randomly select a subset of the first parent string (see line 12 in #breed function below)  
# and then fill the remainder of the route with the genes from the second parent in the #order in which they appear,  
# without duplicating any genes in the selected subset from the first parent*def breed(parent1, parent2):  
 child = []  
 childP1 = []  
 childP2 = []  
  
 geneA = int(random.random() \* len(parent1))  
 geneB = int(random.random() \* len(parent1))  
  
 startGene = min(geneA, geneB)  
 endGene = max(geneA, geneB)  
  
 for i in range(startGene, endGene):  
 childP1.append(parent1[i])  
  
 childP2 = [item for item in parent2 if item not in childP1]  
  
 child = childP1 + childP2  
 return child  
  
*#we’ll generalize this to create our offspring population.  
# We use elitism to retain the best routes from the current population.  
# Then, we use the breed function to fill out the rest of the next generation.*def breedPopulation(matingpool, eliteSize):  
 children = []  
 length = len(matingpool) - eliteSize  
 pool = random.sample(matingpool, len(matingpool))  
  
 for i in range(0, eliteSize):  
 children.append(matingpool[i])  
  
 for i in range(0, length):  
 child = breed(pool[i], pool[len(matingpool) - i - 1])  
 children.append(child)  
 return children  
  
*#Mutate:  
#Mutation serves an important function in GA, as it helps to avoid local convergence by #introducing novel routes  
# that will allow us to explore other parts of the solution space.  
# Similar to crossover, the TSP has a special consideration when it comes to mutation.  
#since we need to abide by our rules, we can’t drop cities. Instead, we’ll use swap mutation.  
# This means that, with specified low probability, two cities will swap places in our route.  
# We’ll do this for one individual in our mutate function:*def mutate(individual, mutationRate):  
 for swapped in range(len(individual)):  
 if (random.random() < mutationRate):  
 swapWith = int(random.random() \* len(individual))  
  
 city1 = individual[swapped]  
 city2 = individual[swapWith]  
  
 individual[swapped] = city2  
 individual[swapWith] = city1  
 return individual  
  
*#Next, we can extend the mutate function to run through the new population.*def mutatePopulation(population, mutationRate):  
 mutatedPop = []  
  
 for ind in range(0, len(population)):  
 mutatedInd = mutate(population[ind], mutationRate)  
 mutatedPop.append(mutatedInd)  
 return mutatedPop  
  
*#We’re almost there. Let’s pull these pieces together to create a function that produces a #new generation.  
# First, we rank the routes in the current generation using rankRoutes.  
# We then determine our potential parents by running the selection function,  
# which allows us to create the mating pool using the matingPool function.  
# Finally, we then create our new generation using the breedPopulation function and  
# then applying mutation using the mutatePopulation function.*def nextGeneration(currentGen, eliteSize, mutationRate):  
 popRanked = rankRoutes(currentGen)  
 selectionResults = selection(popRanked, eliteSize)  
 matingpool = matingPool(currentGen, selectionResults)  
 children = breedPopulation(matingpool, eliteSize)  
 nextGeneration = mutatePopulation(children, mutationRate)  
 return nextGeneration  
  
*#We finally have all the pieces in place to create our GA!  
# All we need to do is create the initial population, and then we can loop through as many #generations as we desire.  
# Of course we also want to see the best route and how much we’ve improved, so we #capture the initial distance  
# the final distance and the best route*def geneticAlgorithm(population, popSize, eliteSize, mutationRate, generations):  
 pop = initialPopulation(popSize, population)  
 print("Initial distance: " + str(1 / rankRoutes(pop)[0][1]))  
  
 for i in range(0, generations):  
 pop = nextGeneration(pop, eliteSize, mutationRate)  
  
 print("Final distance: " + str(1 / rankRoutes(pop)[0][1]))  
 bestRouteIndex = rankRoutes(pop)[0][0]  
 bestRoute = pop[bestRouteIndex]  
 return bestRoute  
  
*#Running the genetic algorithm  
#First, we need a list of cities to travel between. For this demonstration, we’ll create a list #of 25 random cities*cityList = []  
  
for i in range(0,25):  
 cityList.append(City(x=int(random.random() \* 200), y=int(random.random() \* 200)))  
  
*#Then, running the genetic algorithm is one simple line of code. This is where art meets #science;  
# you should see which assumptions work best for you. In this example, we have 100 #individuals in each generation,  
# keep 20 elite individuals, use a 1% mutation rate for a given gene, and run through 500 generations:*geneticAlgorithm(population=cityList, popSize=100, eliteSize=20, mutationRate=0.01, generations=500)  
  
*# we can store the shortest distance from each generation in a progress list and then plot #the results.*def geneticAlgorithmPlot(population, popSize, eliteSize, mutationRate, generations):  
 pop = initialPopulation(popSize, population)  
 progress = []  
 progress.append(1 / rankRoutes(pop)[0][1])  
  
 for i in range(0, generations):  
 pop = nextGeneration(pop, eliteSize, mutationRate)  
 progress.append(1 / rankRoutes(pop)[0][1])  
  
 plt.plot(progress)  
 plt.ylabel('Distance')  
 plt.xlabel('Generation')  
 plt.show()  
  
*#Run the GA in the same way as before, but now using the newly created #geneticAlgorithmPlot function*geneticAlgorithmPlot(population=cityList, popSize=100, eliteSize=20, mutationRate=0.01, generations=500)

**Output:**

