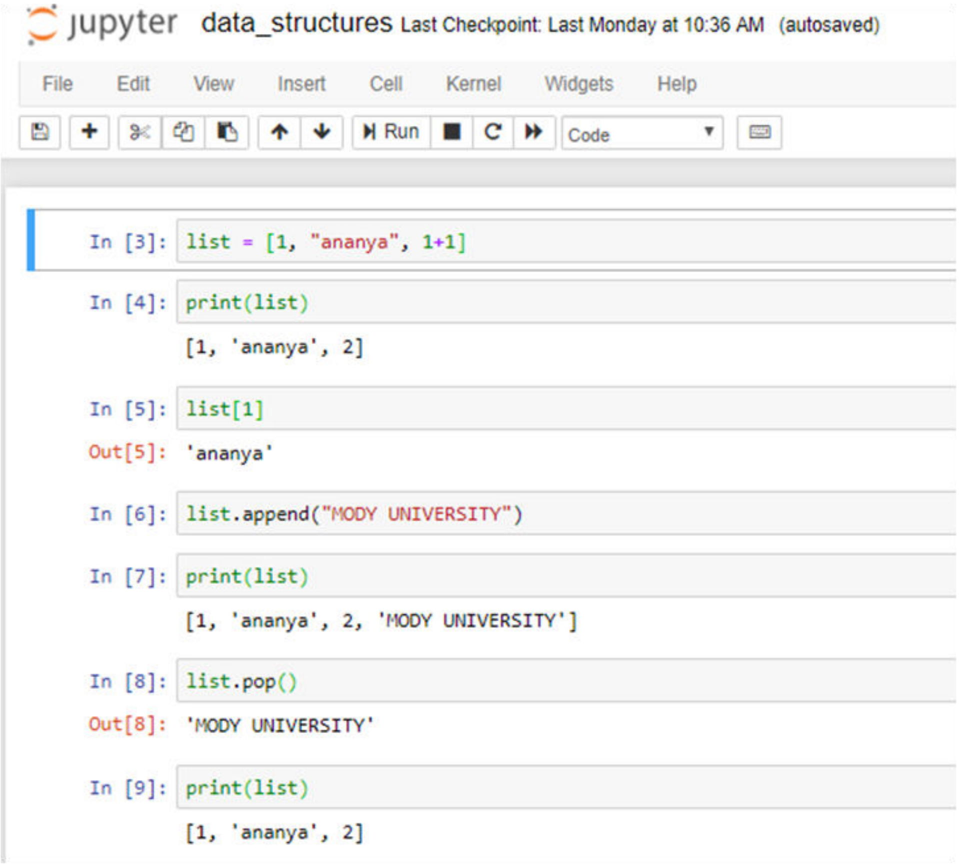
PROGRAM: 4

AIM: Write a program to implement various type of data structures available in python and their operations.

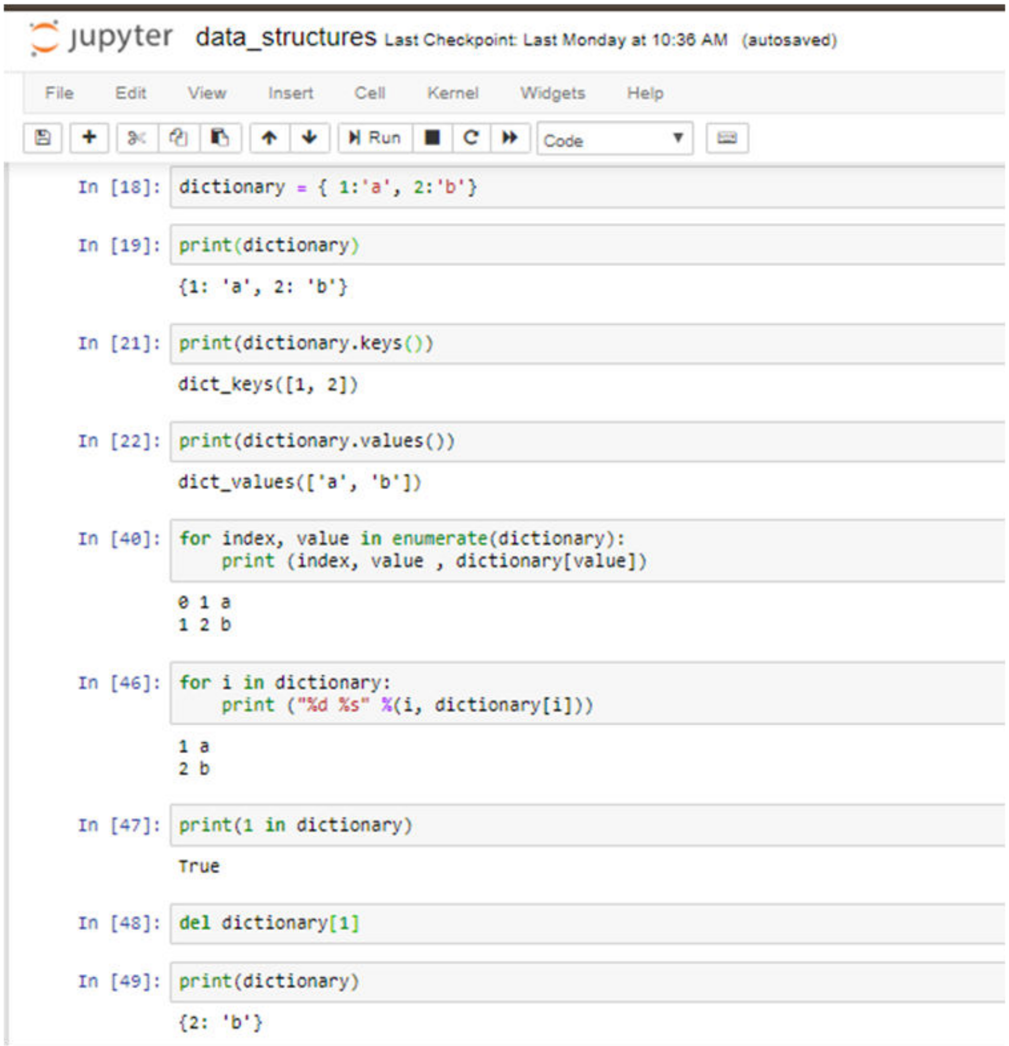
CODE:

```
list = [1, "ananya", 1+1]
print(list)
list[1]
list.append("MODY UNIVERSITY")
print(list)
list.pop()
print(list)


dictionary = { 1:'a', 2:'b'}
print(dictionary)
print(dictionary.keys())
print(dictionary.values())
for index, value in enumerate(dictionary):
        print (index, value , dictionary[value])
for i in dictionary:
        print ("%d %s" %(i, dictionary[i]))
print(1 in dictionary)
del dictionary[1]
print(dictionary)
print(1 in dictionary)


tuple = (1,"ananya", 1+2)
print(tuple)
print(tuple[1])
tuple[2]


set = set()
for i in range(1,10):
        set.add(i)
print(set)
print("set =",set)
```

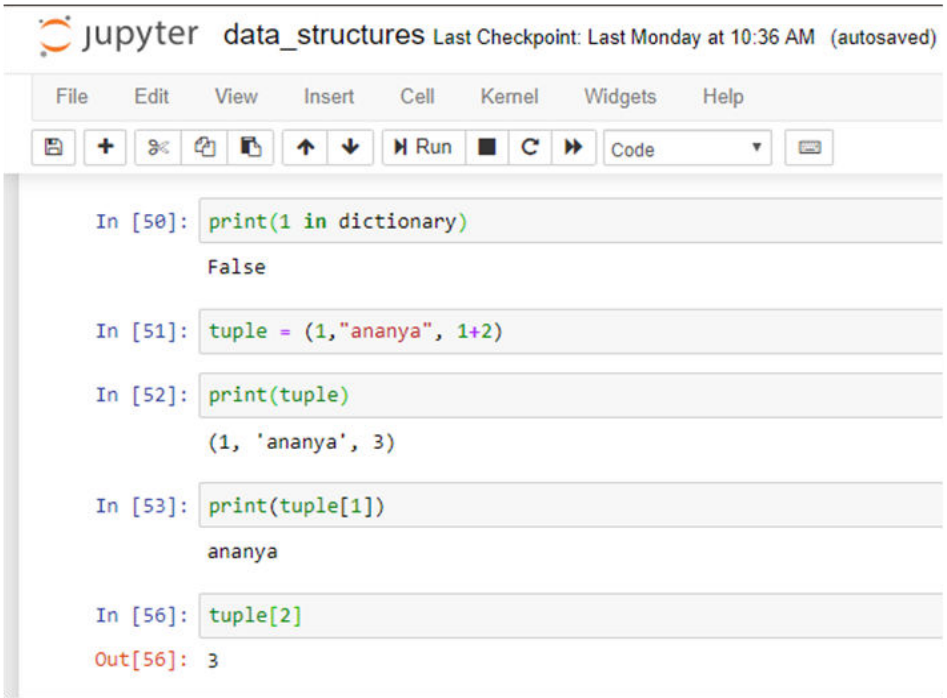
OUTPUT:


jupyter data_structures Last Checkpoint: Last Monday at 10:36 AM (autosaved)
File Edit View Insert Cell Kernel Widgets Help
Run
Code

```
In [3]: list = [1, "ananya", 1+1]

In [4]: print(list)

        [1, 'ananya', 2]

In [5]: list[1]

Out[5]: 'ananya'

In [6]: list.append("MODY UNIVERSITY")

In [7]: print(list)

        [1, 'ananya', 2, 'MODY UNIVERSITY']

In [8]: list.pop()

Out[8]: 'MODY UNIVERSITY'

In [9]: print(list)

        [1, 'ananya', 2]
```


jupyter
data_structures Last Checkpoint: Last Monday at 10:36 AM (autosaved)
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Run
Code

```
In [18]: dictionary = { 1:'a', 2:'b'}

In [19]: print(dictionary)

         {1: 'a', 2: 'b'}

In [21]: print(dictionary.keys())

         dict_keys([1, 2])

In [22]: print(dictionary.values())

         dict_values(['a', 'b'])

In [40]: for index, value in enumerate(dictionary):
             print (index, value , dictionary[value])

         0 1 a
         1 2 b

In [46]: for i in dictionary:
             print ("%d %s" %(i, dictionary[i]))

         1 a
         2 b

In [47]: print(1 in dictionary)

         True

In [48]: del dictionary[1]

In [49]: print(dictionary)

         {2: 'b'}
```



jupyter data_structures Last Checkpoint: Last Monday at 10:36 AM (autosaved)
File Edit View Insert Cell Kernel Widgets Help
Run
Code

```
In [50]: print(1 in dictionary)

False

In [51]: tuple = (1,"ananya", 1+2)

In [52]: print(tuple)

(1, 'ananya', 3)

In [53]: print(tuple[1])

ananya

In [56]: tuple[2]

Out[56]: 3
```

PROGRAM: 5

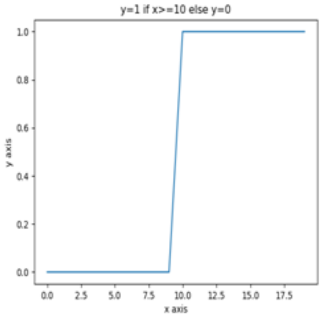
AIM: Plot the graph for following equations

1. y= {1 x<10
0 else }

CODE:

```
import matplotlib.pyplot as plt
x=[num for num in range(0,20)]
y=[]
for i in x:
        if i>=10:
y.append(1)
        else:
y.append(0)
plt.plot(x,y)
plt.xlabel("x axis")
plt.ylabel("y axis")
plt.title("y=1 if x>=10 else y=0")
plt.show()
```

OUTPUT:

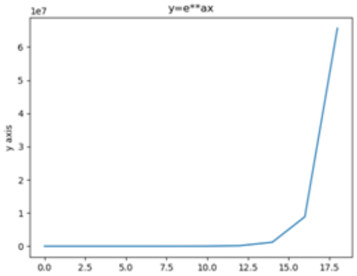


2. y=eaxfor different values of a.

CODE:

```
import numpy as np
import matplotlib.pyplot as plt
num=[num for num in range(0,10)]
a=2 #constant value
x=np.array(num)*2
y=np.exp(x)
plt.plot(x,y)
plt.xlabel("x axis")
plt.ylabel("y axis")
plt.title("y=e**ax")
plt.show()
```

OUTPUT:

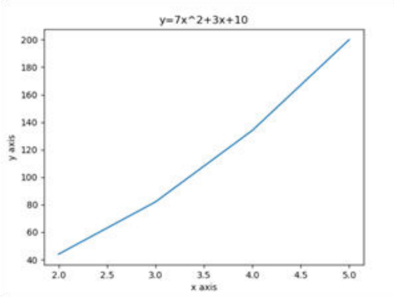


3.$y = 7x^2 + 3x + 10\ for\ 2 \le x \le 5$

CODE:

```
import matplotlib.pyplot as plt
x=[x for x in range(2,6)]
y=[]
for i in x:
y.append(7*pow(i,2)+3*i+10)
plt.plot(x,y)
plt.xlabel('x axis')
plt.ylabel('y axis')
plt.title('y=7x^2+3x+10')
plt.show()
```

OUTPUT:



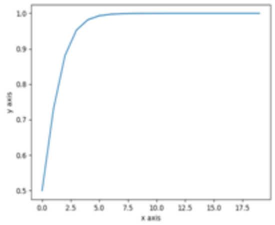
4. $y = \frac{1}{1 + e^{-x}}$

CODE:

```
import matplotlib.pyplot as plt
import numpy as np
x=[x for x in range(0,20)]
```

```
p=np.array(x)
p=1+np.exp(-p)
y=1/p
plt.plot(x,y)
plt.xlabel('x axis')
plt.ylabel('y axis')
plt.show()
```

OUTPUT:

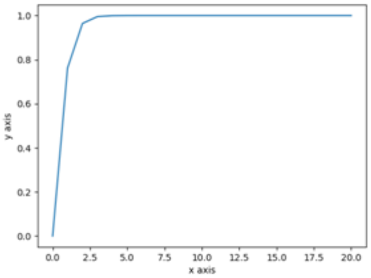


5. $y = \frac{1-e^{-ax}}{1+e^{-ax}}$ **for different values of a.**

CODE:

```
import matplotlib.pyplot as plt
import numpy as np
num=[num for num in range(0,21)]
num=np.array(num)
a=2
p=1-np.exp(-num*a)
q=1+np.exp(-num*a)
y=p/q
plt.plot(num,y)
plt.xlabel('x axis')
plt.ylabel('y axis')
plt.show()
```

OUTPUT:

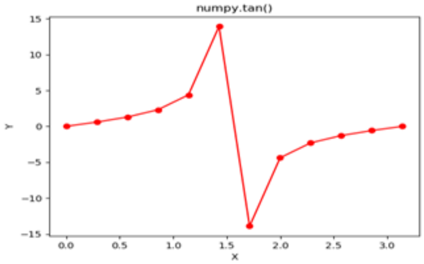


6. $y = \tan hx$

CODE:

```
import numpy as np
import matplotlib.pyplot as plt
in_array = np.linspace(0, np.pi, 12)
h=2
out_array =h*np.tan(in_array)
print("in_array : ", in_array)
print("\nout_array : ",out_array)
# red for numpy.tan()
plt.plot(in_array, out_array, color='red', marker="o")
plt.title("numpy.tan()")
plt.xlabel("X")
plt.ylabel("Y")
plt.show()
```

OUTPUT:



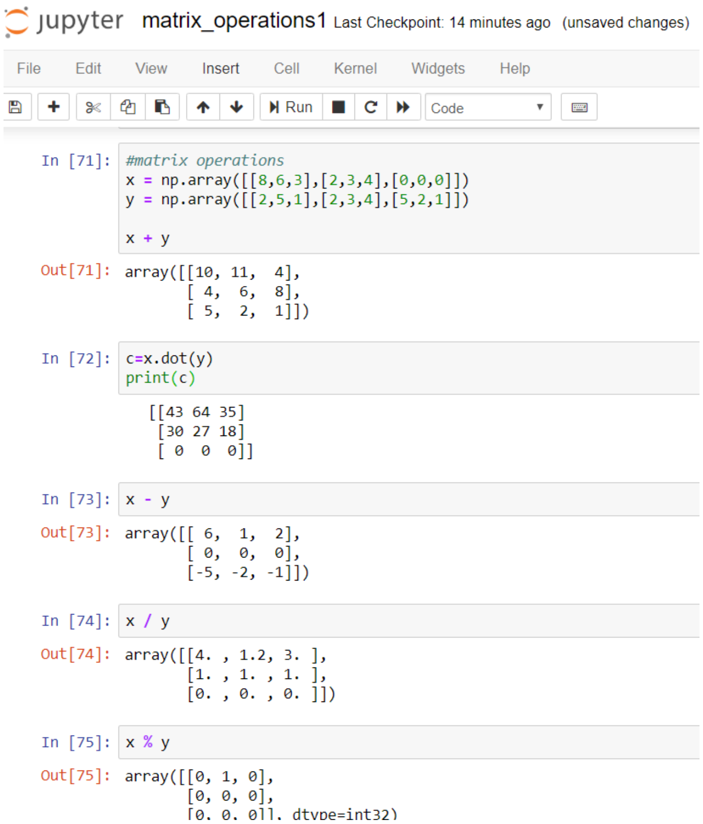
PROGRAM: 6

AIM: Demonstrate the use of NumPy for matrix operations.

CODE:

```
#matrix operations
x = np.array([[8,6,3],[2,3,4],[0,0,0]])
y = np.array([[2,5,1],[2,3,4],[5,2,1]])
x + y
c=x.dot(y)
print(c)
x – y
x / y
x % y
```

OUTPUT:



jupyter
matrix_operations1 Last Checkpoint: 14 minutes ago (unsaved changes)
File Edit View Insert Cell Kernel Widgets Help
Run
Code

```
In [71]: #matrix operations
         x = np.array([[8,6,3],[2,3,4],[0,0,0]])
         y = np.array([[2,5,1],[2,3,4],[5,2,1]])

         x + y

Out[71]: array([[10, 11,  4],
                [ 4,  6,  8],
                [ 5,  2,  1]])

In [72]: c=x.dot(y)
         print(c)

            [[43 64 35]
             [30 27 18]
             [ 0  0  0]]

In [73]: x - y

Out[73]: array([[ 6,  1,  2],
                [ 0,  0,  0],
                [-5, -2, -1]])

In [74]: x / y

Out[74]: array([[4. , 1.2, 3. ],
                [1. , 1. , 1. ],
                [0. , 0. , 0. ]])

In [75]: x % y

Out[75]: array([[0, 1, 0],
                [0, 0, 0],
                [0, 0, 0]], dtype=int32)
```

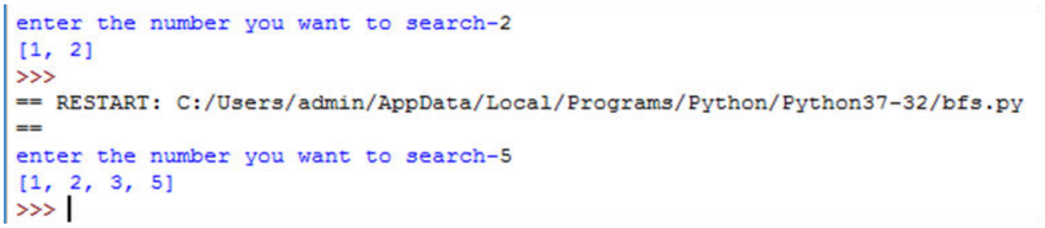
PROGRAM:7

AIM: Write a program to implement bfs and dfs with the input of graph and the goal node to be searched, your output will show the path from the root node to goal node only

CODE:

```
def bfs(graph,start,search):
    explored = []
    queue = [start]
    found = 1
    while found:
        node = queue.pop(0)
        if(node == search):
            found = 0
        if node not in explored:
            explored.append(node)
            neighbours = graph[node]
        for neighbour in neighbours:
            queue.append(neighbour)
    print(explored)

search = int(input("enter the number you want to search-"))
graph = {1: [2, 3, 5],
         2: [1,4, 5],
         3: [1, 6, 7],
         4: [2],
         5: [1, 2,4],
         6: [3],
         7: [3]}
bfs(graph,1,search)
```

OUTPUT:



```
enter the number you want to search-2
[1, 2]
>>>
== RESTART: C:/Users/admin/AppData/Local/Programs/Python/Python37-32/bfs.py
==
enter the number you want to search-5
[1, 2, 3, 5]
>>> |
```

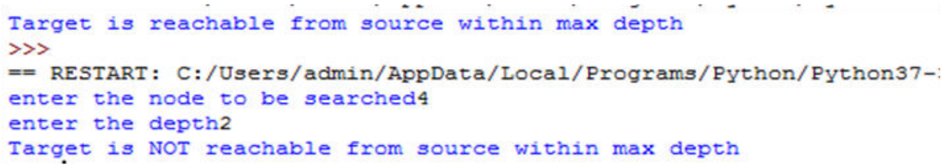
PROGRAM: 8

AIM: Write a program to implement dfs with using a fix limit and return the path to traverse till input node.

CODE:

```
from collections import defaultdict
class Graph:
      def __init__(self,vertices):
        self.V = vertices
        self.graph = defaultdict(list)
    def addEdge(self,u,v):
        self.graph[u].append(v)
    def DLS(self,src,target,maxDepth):
        if src == target : return True
        if maxDepth <= 0 : return False
        for i in self.graph[src]:
                if(self.DLS(i,target,maxDepth-1)):
                    return True
        return False
    def IDDFS(self,src, target, maxDepth):
        for i in range(maxDepth):
            if (self.DLS(src, target, i)):
                return True
        return False
g = Graph (7);
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 3)
g.addEdge(1, 4)
g.addEdge(2, 5)
g.addEdge(2, 6)
target = int(input("enter the node to be searched"));
maxDepth = int(input("enter the depth"));
src = 0
if g.IDDFS(src, target, maxDepth) == True:
    print ("Target is reachable from source " +
        "within max depth")
else :
    print ("Target is NOT reachable from source " +
        "within max depth")
```

OUTPUT:



```
Target is reachable from source within max depth
>>>
== RESTART: C:/Users/admin/AppData/Local/Programs/Python/Python37-
enter the node to be searched4
enter the depth2
Target is NOT reachable from source within max depth
```

PROGRAM: 9

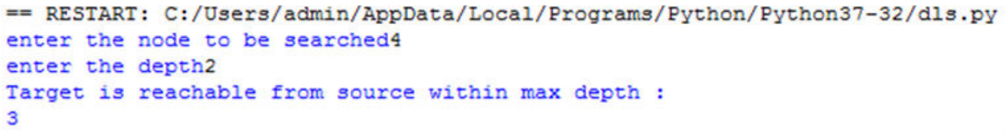
AIM: Write a program to implement iterative deepening dfs with using variable limit and return the path to traverse till input node.

CODE:

```
from collections import defaultdict

class Graph:

        def __init__(self,vertices):

                self.V = vertices
                self.graph = defaultdict(list)
        def addEdge(self,u,v):
                self.graph[u].append(v)

        def DLS(self,src,target,maxDepth):
                if src == target : return True
                if maxDepth <= 0 : return False

                for i in self.graph[src]:
                                if(self.DLS(i,target,maxDepth-1)):
                                        return True
                return False
        def IDDFS(self,src, target, maxDepth):
                for i in range(maxDepth):
                        if (self.DLS(src, target, i)):
                                return True
                return False


g = Graph (7);
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 3)
g.addEdge(1, 4)
g.addEdge(2, 5)
g.addEdge(2, 6)

target = int(input("enter the node to be searched"));
maxDepth = int(input("enter the depth"));
src = 0
found = 1
while(found):
        if g.IDDFS(src, target, maxDepth) == True:
```

```
        print ("Target is reachable from source " +
        "within max depth : ")
        print(maxDepth)
        found = 0
    else :
        maxDepth = maxDepth +1
```

OUTPUT:



```
== RESTART: C:/Users/admin/AppData/Local/Programs/Python/Python37-32/dls.py
enter the node to be searched4
enter the depth2
Target is reachable from source within max depth :
3
```

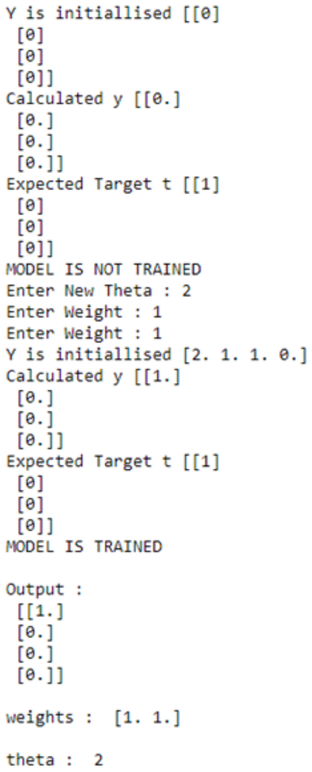
PROGRAM: 10

AIM: Write a program to implement AND logic functions using numpy neuron.

CODE:

```
importnumpy as np
x=np.array([[1,1],[1,0],[0,1],[0,0]])
t=np.array([[1],[0],[0],[0]])
w=np.array([[0],[0]])
theta=1
yin=np.zeros(shape=(4,1))
y=np.zeros(shape=(4,1))
yin=np.dot(x,w)
i=0
found=0
while(found==0):
i=0
yin=np.dot(x,w)
print("Y is initiallised",yin)
while(i<4):
if yin[i]>=theta:
y[i]=1
i=i+1
else:
y[i]=0
i=i+1
print("Calculated y",y)
print("Expected Target t",t)
if (y==t).all():
print("MODEL IS TRAINED ")
print("\nOutput : \n",y)
print("\nweights : ",w,"\n")
print("theta : ",theta)
found=1
else:
print("MODEL IS NOT TRAINED")
           w=np.zeros(shape=(0,0))
theta=int(input("Enter New Theta : "))
for k in range(int(2)):
                w1=int(input("Enter Weight : "))
                w=np.append(w,w1)
```

OUTPUT:



```
Y is initiallised [[0]
 [0]
 [0]
 [0]]
Calculated y [[0.]
 [0.]
 [0.]
 [0.]]
Expected Target t [[1]
 [0]
 [0]
 [0]]
MODEL IS NOT TRAINED
Enter New Theta : 2
Enter Weight : 1
Enter Weight : 1
Y is initiallised [2. 1. 1. 0.]
Calculated y [[1.]
 [0.]
 [0.]
 [0.]]
Expected Target t [[1]
 [0]
 [0]
 [0]]
MODEL IS TRAINED

Output :
 [[1.]
 [0.]
 [0.]
 [0.]]

weights :  [1. 1.]

theta :  2
```

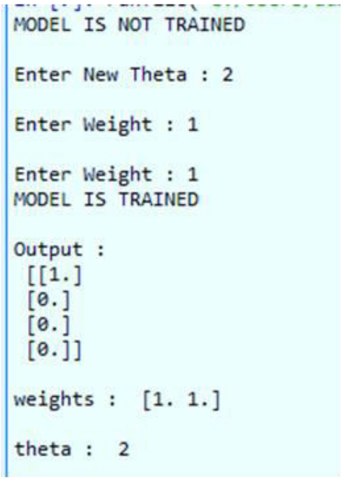
PROGRAM: 11

AIM: Write a program to implement AND logic functions.

CODE:

```
import numpy as np
x=np.array([[1,1],[1,0],[0,1],[0,0]])
t=np.array([[1],[0],[0],[0]])
w=np.array([[0],[0]])
theta=1
yin=np.zeros(shape=(4,1))
y=np.zeros(shape=(4,1))
yin=np.dot(x,w)
i=0
found=0
while(found==0):
        i=0
        yin=np.dot(x,w)
        #print(yin)
        while(i<4):
                if yin[i]>=theta:
                y[i]=1
                i=i+1
                 else:
                y[i]=0
                i=i+1
     #print("y",y)
     #print("t",t)
        if (y==t).all():
                print("MODEL IS TRAINED ")
                print("\nOutput : \n",y)
                print("\nweights : ",w,"\n")
                print("theta : ",theta)
                found=1
         else:
                 print("MODEL IS NOT TRAINED")
                w=np.zeros(shape=(0,0))
                theta=int(input("Enter New Theta : "))
                for k in range(int(2)):
                w1=int(input("Enter Weight : "))
                w=np.append(w,w1)
```

OUTPUT:



```
MODEL IS NOT TRAINED

Enter New Theta : 2

Enter Weight : 1

Enter Weight : 1
MODEL IS TRAINED

Output :
 [[1.]
 [0.]
 [0.]
 [0.]]

weights :  [1. 1.]

theta :  2
```

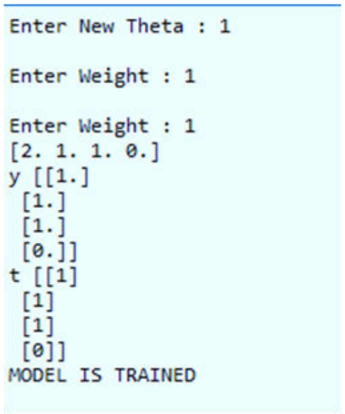
PROGRAM: 12

AIM: Write a program to implement OR logic functions.

CODE:

```
import numpy as np
x=np.array([[1,1],[1,0],[0,1],[0,0]])
t=np.array([[1],[1],[1],[0]])
w=np.array([[0],[0]])
theta=1
yin=np.zeros(shape=(4,1))
y=np.zeros(shape=(4,1))
yin=np.dot(x,w)
i=0
found=0
while(found==0):
    i=0
    yin=np.dot(x,w)
    #print(yin)
    while(i<4):
        if yin[i]>=theta:
        y[i]=1
        i=i+1
         else:
        y[i]=0
        i=i+1
   #print("y",y)
   #print("t",t)
    if (y==t).all():
        print("MODEL IS TRAINED ")
        print("\nOutput : \n",y)
        print("\nweights : ",w,"\n")
        print("theta : ",theta)
        found=1
     else:
         print("MODEL IS NOT TRAINED")
        w=np.zeros(shape=(0,0))
        theta=int(input("Enter New Theta : "))
        for k in range(int(2)):
        w1=int(input("Enter Weight : "))
        w=np.append(w,w1)
```

OUTPUT:



```
Enter New Theta : 1

Enter Weight : 1

Enter Weight : 1
[2. 1. 1. 0.]
y [[1.]
 [1.]
 [1.]
 [0.]]
t [[1]
 [1]
 [1]
 [0]]
MODEL IS TRAINED
```

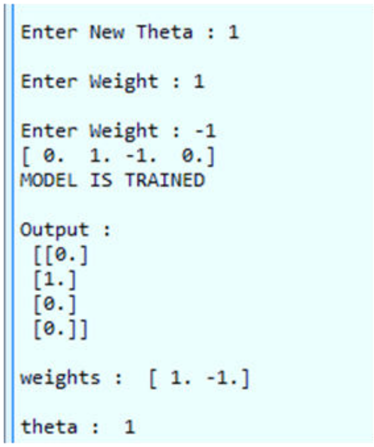
PROGRAM: 13

AIM: Write a program to implement AND-NOT logic functions.

CODE:

```
import numpy as np
x=np.array([[1,1],[1,0],[0,1],[0,0]])
t=np.array([[0],[1],[0],[0]])
w=np.array([[0],[0]])
theta=1
yin=np.zeros(shape=(4,1))
y=np.zeros(shape=(4,1))
yin=np.dot(x,w)
i=0
found=0
while(found==0):
    i=0
    yin=np.dot(x,w)
    #print(yin)
    while(i<4):
        if yin[i]>=theta:
        y[i]=1
        i=i+1
         else:
        y[i]=0
        i=i+1
  #print("y",y)
  #print("t",t)
    if (y==t).all():
        print("MODEL IS TRAINED ")
        print("\nOutput : \n",y)
        print("\nweights : ",w,"\n")
        print("theta : ",theta)
        found=1
     else:
         print("MODEL IS NOT TRAINED")
        w=np.zeros(shape=(0,0))
        theta=int(input("Enter New Theta : "))
        for k in range(int(2)):
        w1=int(input("Enter Weight : "))
        w=np.append(w,w1)
```

OUTPUT:



```
Enter New Theta : 1

Enter Weight : 1

Enter Weight : -1
[ 0.  1. -1.  0.]
MODEL IS TRAINED

Output :
 [[0.]
 [1.]
 [0.]
 [0.]]

weights :  [ 1. -1.]

theta :  1
```

PROGRAM: 14

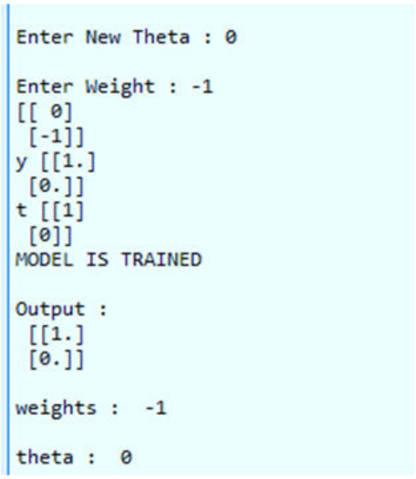
AIM: Write a program to implement NOT logic functions.

CODE:

```
import numpy as np
x=np.array([[0],[1]])
t=np.array([[1],[0]])
w=np.array([0])
theta=1
yin=np.zeros(shape=(2,1))
y=np.zeros(shape=(2,1))
yin=np.dot(x,w)
i=0
found=0
while(found==0):
    i=0
    yin=np.dot(x,w)
    print(yin)
    while(i<2):
        if yin[i]>=theta:
        y[i]=1
        i=i+1

        #if(i==4):
            #break
        else:
        y[i]=0
        i=i+1
    print("y",y)
    print("t",t)
    if (y==t).all():
        print("MODEL IS TRAINED ")
        print("\nOutput : \n",y)
        print("\nweights : ",w,"\n")
        print("theta : ",theta)
        found=1
    else:
        print("MODEL IS NOT TRAINED")
        w=np.zeros(shape=(0,0))
        theta=int(input("Enter New Theta : "))
```

```
for k in range(int(1)):
w=int(input("Enter Weight : "))
```

OUTPUT:



```
Enter New Theta : 0

Enter Weight : -1
[[ 0]
 [-1]]
y [[1.]
 [0.]]
t [[1]
 [0]]
MODEL IS TRAINED

Output :
 [[1.]
 [0.]]

weights :  -1

theta :  0
```

PROGRAM: 15

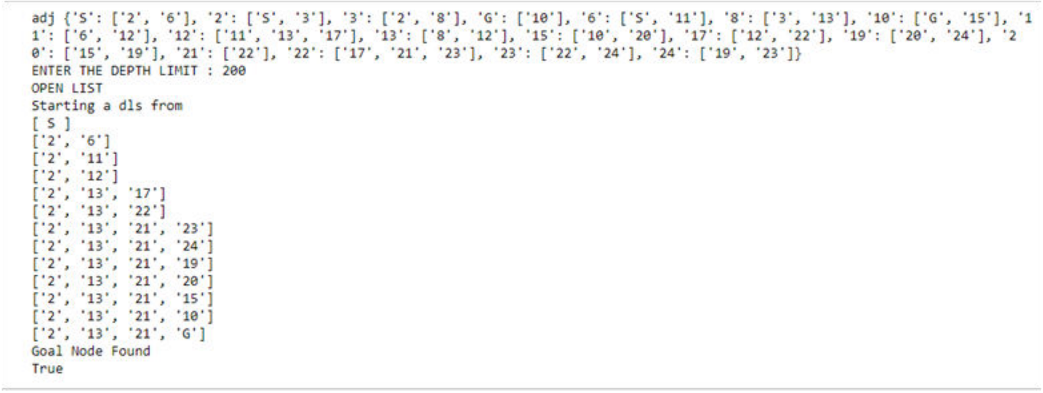
AIM: Write a program to implement depth limited search.

CODE:

```
ADJ = {}
""""
SRRXG
RXRXR
RRRXR
XRXRR
RRRRX
"""
ADJ['S'] = ['2', '6']
ADJ['2'] = ['S', '3']
ADJ['3'] = ['2','8']
ADJ['G'] = ['10']
ADJ['6'] = ['S', '11']
ADJ['8'] = ['3', '13']
ADJ['10'] = ['G', '15']
ADJ['11'] = ['6', '12']
ADJ['12'] = ['11', '13', '17']
ADJ['13'] = ['8', '12']
ADJ['15'] = ['10', '20']
ADJ['17'] = ['12','22']
ADJ['19'] = ['20', '24']
ADJ['20'] = ['15','19']
ADJ['21'] = ['22']
ADJ['22'] = ['17','21','23']
ADJ['23'] = ['22', '24']
ADJ['24'] = ['19','23']
print ("adj",ADJ)
# keep track of visited nodes
visited = {str(i) : False for i in range(1,26)}
visited['S'] = False
visited['G'] = False

def dls(start, goal,limit):
        depth = 0

        OPEN=[]
        CLOSED=[]
        OPEN.append(start)
        visited["S"] = True
        while OPEN != []: # Step 2
                if depth<=limit:
```

```
            current = OPEN.pop()

            if current == goal:
                print("Goal Node Found")
                return True
            else:
                lst = successors(current)
                for i in lst:
                    # try to visit a node in future, if not already been to it
                    if(not(visited[i])):
                        OPEN.append(i)

                        visited[i] = True
                depth +=1

        else:
            print("Not found within depth limit")
            return False
        print(OPEN)
        #print("node visited",i,sep='>',end='\n')
    return False

def successors(city):
    return ADJ[city]

def test():
    start = 'S'
    goal = 'G'
    limit=int(input("ENTER THE DEPTH LIMIT"))
    print("Starting a dls from \n[ " + start+" ]")
    print(dls(start, goal,limit))


if __name__ == "__main__":
    test()
```

OUTPUT:



```
adj {'S': ['2', '6'], '2': ['S', '3'], '3': ['2', '8'], 'G': ['10'], '6': ['S', '11'], '8': ['3', '13'], '10': ['G', '15'], '1
1': ['6', '12'], '12': ['11', '13', '17'], '13': ['8', '12'], '15': ['10', '20'], '17': ['12', '22'], '19': ['20', '24'], '2
0': ['15', '19'], '21': ['22'], '22': ['17', '21', '23'], '23': ['22', '24'], '24': ['19', '23']}
ENTER THE DEPTH LIMIT : 200
OPEN LIST
Starting a dls from
[ S ]
['2', '6']
['2', '11']
['2', '12']
['2', '13', '17']
['2', '13', '22']
['2', '13', '21', '23']
['2', '13', '21', '24']
['2', '13', '21', '19']
['2', '13', '21', '20']
['2', '13', '21', '15']
['2', '13', '21', '10']
['2', '13', '21', 'G']
Goal Node Found
True
```

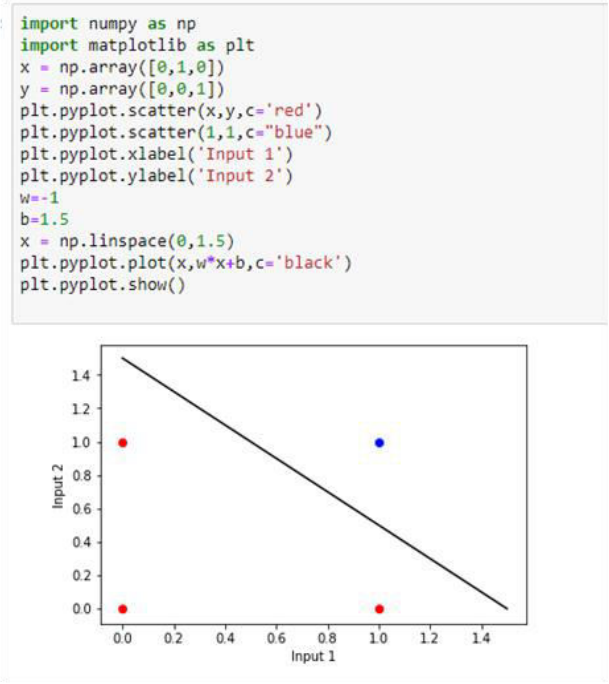
PROGRAM: 16

AIM: Write a program to implement linear separability for AND function.

CODE:

```
import numpy as np
import matplotlib as plt
x = np.array([0,1,0])
y = np.array([0,0,1])
plt.pyplot.scatter(x,y,c='red')
plt.pyplot.scatter(1,1,c="blue")
plt.pyplot.xlabel('Input 1')
plt.pyplot.ylabel('Input 2')
w=-1
b=1.5
x = np.linspace(0,1.5)
plt.pyplot.plot(x,w*x+b,c='black')
plt.pyplot.show()
```

OUTPUT:



```
import numpy as np
import matplotlib as plt
x = np.array([0,1,0])
y = np.array([0,0,1])
plt.pyplot.scatter(x,y,c='red')
plt.pyplot.scatter(1,1,c="blue")
plt.pyplot.xlabel('Input 1')
plt.pyplot.ylabel('Input 2')
w=-1
b=1.5
x = np.linspace(0,1.5)
plt.pyplot.plot(x,w*x+b,c='black')
plt.pyplot.show()
```

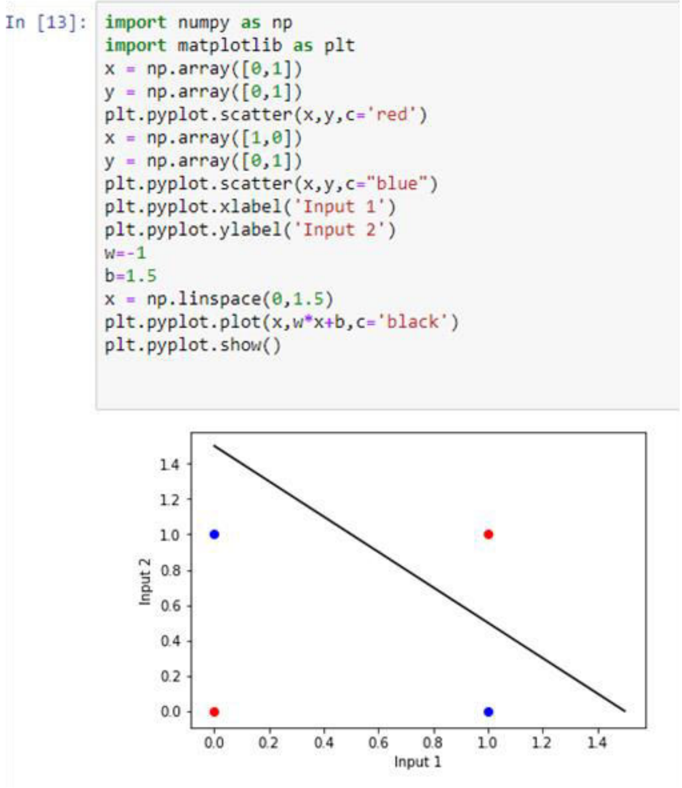
PROGRAM: 17

AIM: Write a program to implement linear separability for OR function.

CODE :

```
import numpy as np
import matplotlib as plt
x = np.array([0,1])
y = np.array([0,1])
plt.pyplot.scatter(x,y,c='red')
x = np.array([1,0])
y = np.array([0,1])
plt.pyplot.scatter(x,y,c="blue")
plt.pyplot.xlabel('Input 1')
plt.pyplot.ylabel('Input 2')
w=-1
b=1.5
x = np.linspace(0,1.5)
plt.pyplot.plot(x,w*x+b,c='black')
plt.pyplot.show()
```

OUTPUT:



```
In [13]: import numpy as np
         import matplotlib as plt
         x = np.array([0,1])
         y = np.array([0,1])
         plt.pyplot.scatter(x,y,c='red')
         x = np.array([1,0])
         y = np.array([0,1])
         plt.pyplot.scatter(x,y,c="blue")
         plt.pyplot.xlabel('Input 1')
         plt.pyplot.ylabel('Input 2')
         w=-1
         b=1.5
         x = np.linspace(0,1.5)
         plt.pyplot.plot(x,w*x+b,c='black')
         plt.pyplot.show()
```

PROGRAM: 18

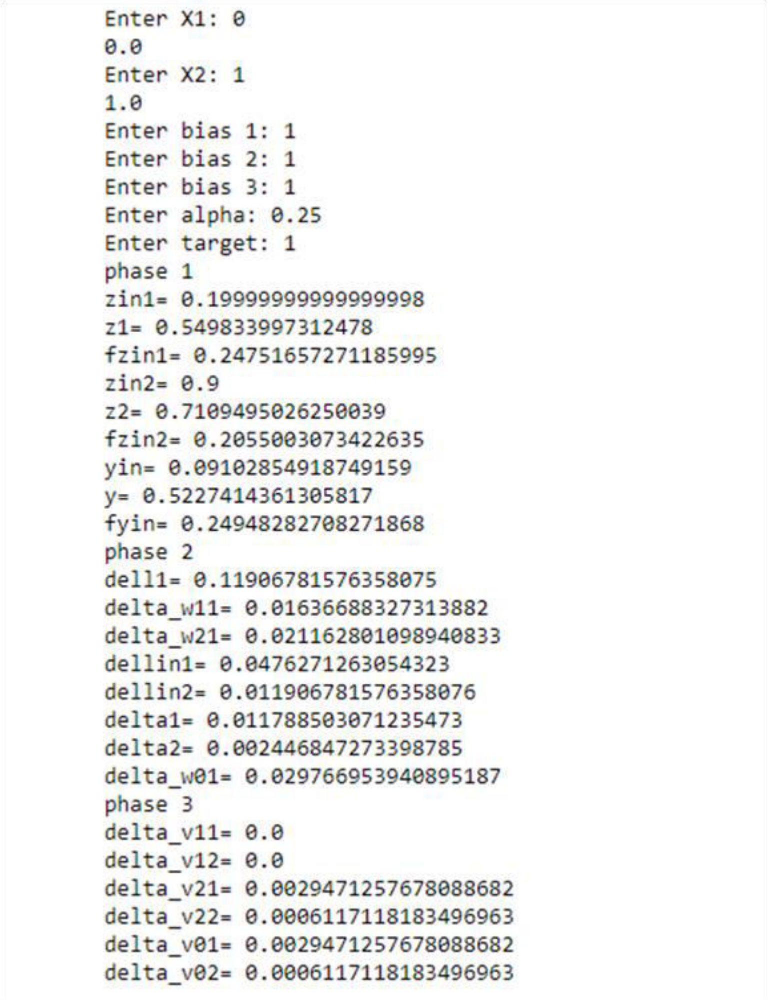
AIM: Write a program to implement Back Propagation Network.

CODE:

```
importnumpy as np
x1 = float(input("Enter X1: "))
print(x1)
x2 = float(input("Enter X2: "))
print(x2)
b1 = float(input("Enter bias 1: "))
b2 = float(input("Enter bias 2: "))
b3 = float(input("Enter bias 3: "))
alpha = float(input("Enter alpha: "))
t = float(input("Enter target: "))

a = [0.6,0.3,-0.1,-0.3,0.4,0.5,0.4,0.1,-0.2]
print('phase 1')
zin1 = float(b1*a[1]+x1*a[0]+x2*a[2])
print('zin1=',zin1)
zp1 = 1/(1+np.exp(-zin1))
print('z1=',zp1)
fzin1= zp1*(1-zp1)
print('fzin1=',fzin1)

zin2= float(a[3]*x1+a[4]*x2+a[5]*b2)
print('zin2=',zin2)
zp2 = 1/(1+np.exp(-zin2))
print('z2=',zp2)
fzin2= zp2*(1-zp2)
print('fzin2=',fzin2)

yin=float(zp1*a[6]+zp2*a[7]+b3*a[8])
print('yin=',yin)
y = 1/(1+np.exp(-yin))
print('y=',y)
fyin= y*(1-y)
print('fyin=',fyin)

print('phase 2')
dell1=(t-y)*fyin
print('dell1=',dell1)
delta_w11=alpha*dell1*zp1
print('delta_w11=',delta_w11)
delta_w21=alpha*dell1*zp2
print('delta_w21=',delta_w21)
```

```
dellin1=dell1*a[6]
print('dellin1=',dellin1)
dellin2 = dell1*a[7]
print('dellin2=',dellin2)

delta1=dellin1*fzin1
print('delta1=',delta1)
delta2=dellin2*fzin2
print('delta2=',delta2)
delta_w01=alpha*dell1
print('delta_w01=',delta_w01)

print('phase 3')
delta_v11=alpha*delta1*x1
print('delta_v11=',delta_v11)
delta_v12=alpha*delta2*x1
print('delta_v12=',delta_v12)
delta_v21=alpha*delta1*x2
print('delta_v21=',delta_v21)
delta_v22=alpha*delta2*x2
print('delta_v22=',delta_v22)

delta_v01 = alpha*delta1
print('delta_v01=',delta_v01)
delta_v02 = alpha*delta2
print('delta_v02=',delta_v02)
```

OUPUT:



```
Enter X1: 0
0.0
Enter X2: 1
1.0
Enter bias 1: 1
Enter bias 2: 1
Enter bias 3: 1
Enter alpha: 0.25
Enter target: 1
phase 1
zin1= 0.19999999999999998
z1= 0.549833997312478
fzin1= 0.24751657271185995
zin2= 0.9
z2= 0.7109495026250039
fzin2= 0.2055003073422635
yin= 0.09102854918749159
y= 0.5227414361305817
fyin= 0.24948282708271868
phase 2
dell1= 0.11906781576358075
delta_w11= 0.01636688327313882
delta_w21= 0.021162801098940833
dellin1= 0.0476271263054323
dellin2= 0.011906781576358076
delta1= 0.011788503071235473
delta2= 0.002446847273398785
delta_w01= 0.029766953940895187
phase 3
delta_v11= 0.0
delta_v12= 0.0
delta_v21= 0.0029471257678088682
delta_v22= 0.0006117118183496963
delta_v01= 0.0029471257678088682
delta_v02= 0.0006117118183496963
```

PROGRAM: 19

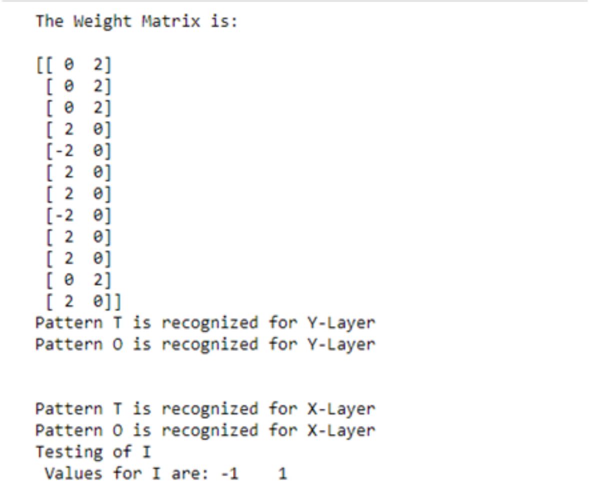
AIM: Write a program in Python to implement Bidirectional Associative Memory (BAM) network to store and test the given patterns.

CODE:

```
importnumpy as np
x1=np.array([[1,1,1,-1,1,-1,-1,1,-1,-1,1,-1]])
x2=np.array([[1,1,1,1,-1,1,1,-1,1,1,1,1]])
x3=np.array([[1,1,1,-1,1,-1,-1,1,-1,1,1,1]])
t1 = np.array([[-1],[1]])
t2 = np.array([[1],[1]])
w1=np.zeros((12,2),dtype=int)
w2=np.zeros((12,2),dtype=int)
w=np.zeros((12,2),dtype=int)
i=0
while(i!=12):
w1[i][0]=x1[0][i]*t1[0][0]
w1[i][1]=x1[0][i]*t1[1][0]
w2[i][0]=x2[0][i]*t2[0][0]
w2[i][1]=x2[0][i]*t2[1][0]
i=i+1
w=w1+w2
print('The Weight Matrix is:\n')
print(w)
Yin11=Yin12=Yin21=Yin22=Yin31=Yin32=0
y1=0
y2=0
i=0
while(i!=12):
        Yin11=Yin11+(x1[0][i]*w[i][0])
        Yin12=Yin12+(x1[0][i]*w[i][1])
        Yin21=Yin21+(x2[0][i]*w[i][0])
        Yin22=Yin22+(x2[0][i]*w[i][1])
        Yin31=Yin31+(x3[0][i]*w[i][0])
        Yin32=Yin32+(x3[0][i]*w[i][1])
i=i+1
if(Yin11>0):
        Yin11=1
else:
        Yin11=-1
if(Yin12>0):
        Yin12=1
else:
        Yin12=-1
if(Yin21>0):
        Yin21=1
else:
        Yin21=-1
if(Yin22>0):
        Yin22=1
else:
        Yin22=-1
if(Yin31>0):
        Yin31=1
else:
        Yin31=-1
if(Yin32>0):
        Yin32=1
else:
        Yin32=-1

if((Yin11==-1) and (Yin12==1)):
print('Pattern T is recognized for Y-Layer')
else:
print('Pattern T is not recognized for Y-Layer')
if((Yin21==1) and (Yin22==1)):
print('Pattern O is recognized for Y-Layer')
else:
print('Pattern O is not recognized for Y-Layer')

i=0
Xin1=np.zeros((12,1),dtype=int)
Xin2=np.zeros((12,1),dtype=int)
while(i!=12):
        Xin1[i][0]=Xin1[i][0]+((Yin11*w[i][0])+(Yin12*w[i][1]))
if(Xin1[i][0]>0):
Xin1[i][0]=1
else:
Xin1[i][0]=-1
        Xin2[i][0]=Xin2[i][0]+((Yin21*w[i][0])+(Yin22*w[i][1]))
if(Xin2[i][0]>0):
Xin2[i][0]=1
else:
```

```
Xin2[i][0]=-1
i=i+1
Xin1=Xin1.T
Xin2=Xin2.T
print('\n')
if((Xin1==x1).all()):
print('Pattern T is recognized for X-Layer')
else:
print('Pattern T is not recognized for X-Layer')
if((Xin2==x2).all()):
print('Pattern O is recognized for X-Layer')
else:
print('Pattern O is not recognized for X-Layer')
print('Testing of I \n Values for I are:', Yin31 ,'\t',Yin32)
```

OUTPUT:



```
The Weight Matrix is:

[[ 0  2]
 [ 0  2]
 [ 0  2]
 [ 2  0]
 [-2  0]
 [ 2  0]
 [ 2  0]
 [-2  0]
 [ 2  0]
 [ 2  0]
 [ 0  2]
 [ 2  0]]
Pattern T is recognized for Y-Layer
Pattern O is recognized for Y-Layer


Pattern T is recognized for X-Layer
Pattern O is recognized for X-Layer
Testing of I
 Values for I are: -1    1
```

PROGRAM: 20

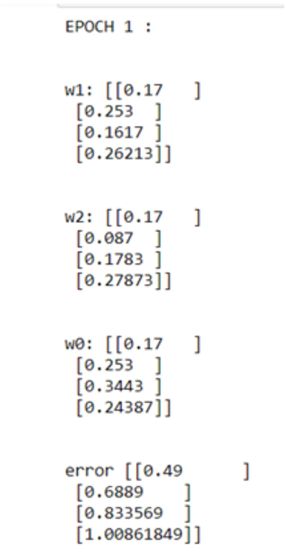
AIM:Write a program in Python to implement Adaline Neural Network.

CODE:

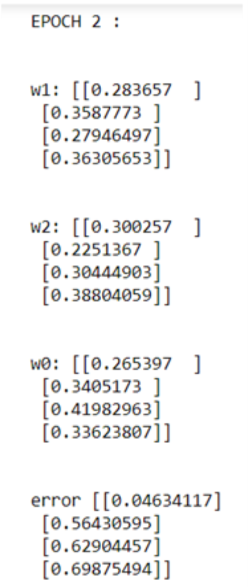
```
import numpy as np
x1=np.array([[1,1,-1,-1]])
x2=np.array([[1,-1,1,-1]])
t=np.array([[1],[1],[1],[-1]])
w11=0.1
w21=0.1
w01=0.1
alpha=0.1
i=0
bias=1
w1=np.zeros((4,1))
w2=np.zeros((4,1))
w0=np.zeros((4,1))
Yin=np.zeros((4,1))
y=np.zeros((4,1))
error=np.zeros((4,1))
count=0
while(count!=3):
    i=0
if(count!=0):
        w11=w1[3]
        w21=w2[3]
```

```
          w01=w0[3]
while(i!=4):
if(i==0):
               Yin[i]= (x1[0][i]*w11)+(x2[0][i]*w21)+(bias*w01)
y[i]=t[i][0]-Yin[i]
w1[i]=w11+(alpha*y[i]*x1[0][i])
w2[i]=w21+(alpha*y[i]*x2[0][i])
w0[i]=w01+(alpha*y[i]*bias)
else:
if(i>0 & i<=4):
                    Yin[i]= (x1[0][i]*w1[i-1])+(x2[0][i]*w2[i-1])+(bias*w0[i-1])
y[i]=t[i][0]-Yin[i]
w1[i]=w1[i-1]+(alpha*y[i]*x1[0][i])
w2[i]=w2[i-1]+(alpha*y[i]*x2[0][i])
w0[i]=w0[i-1]+(alpha*y[i]*bias)
error[i]=(y[i])**2
          i=i+1
print('EPOCH',(count+1),':')
print('\n')
print('w1:',w1)
print('\n')
print('w2:',w2)
print('\n')
print('w0:',w0)
print('\n')
print('error',error)

print('\n\n')

count=count+1
```

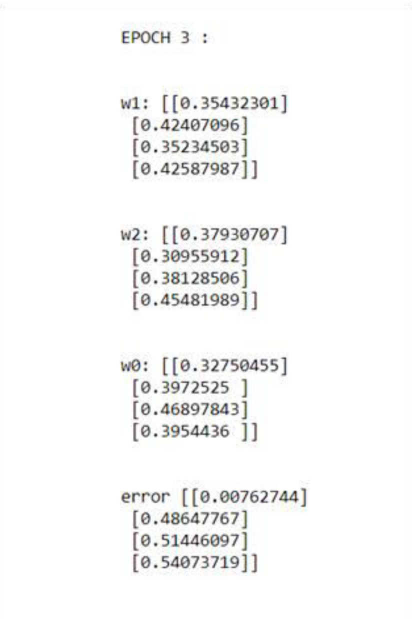
OUTPUT:



```
EPOCH 1 :


w1: [[0.17    ]
 [0.253  ]
 [0.1617 ]
 [0.26213]]


w2: [[0.17    ]
 [0.087  ]
 [0.1783 ]
 [0.27873]]


w0: [[0.17    ]
 [0.253  ]
 [0.3443 ]
 [0.24387]]


error [[0.49        ]
 [0.6889     ]
 [0.833569   ]
 [1.00861849]]
```



```
EPOCH 2 :


w1: [[0.283657   ]
 [0.3587773 ]
 [0.27946497]
 [0.36305653]]


w2: [[0.300257   ]
 [0.2251367 ]
 [0.30444903]
 [0.38804059]]


w0: [[0.265397   ]
 [0.3405173 ]
 [0.41982963]
 [0.33623807]]


error [[0.04634117]
 [0.56430595]
 [0.62904457]
 [0.69875494]]
```



```
EPOCH 3 :


w1: [[0.35432301]
 [0.42407096]
 [0.35234503]
 [0.42587987]]


w2: [[0.37930707]
 [0.30955912]
 [0.38128506]
 [0.45481989]]


w0: [[0.32750455]
 [0.3972525 ]
 [0.46897843]
 [0.3954436 ]]


error [[0.00762744]
 [0.48647767]
 [0.51446097]
 [0.54073719]]
```

PROGRAM: 21

AIM: Write a program in Python to implement Back-Proagation Neural Network.

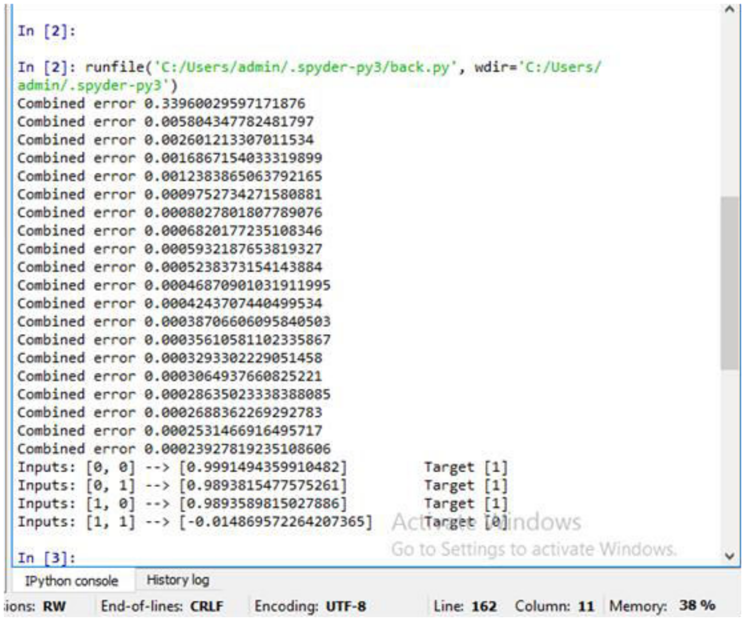
CODE:

```
import math
import random
import string
class NN:
def __init__(self, NI, NH, NO):
        # number of nodes in layers
        self.ni = NI + 1 # +1 for bias
self.nh = NH
        self.no = NO
        self.ai, self.ah, self.ao = [],[], []
        self.ai = [1.0]*self.ni
self.ah = [1.0]*self.nh
        self.ao = [1.0]*self.no
self.wi = makeMatrix (self.ni, self.nh)
self.wo = makeMatrix (self.nh, self.no)
        # initialize node weights to random vals
randomizeMatrix ( self.wi, -0.2, 0.2 )
randomizeMatrix ( self.wo, -2.0, 2.0 )
        self.ci = makeMatrix (self.ni, self.nh)
        self.co = makeMatrix (self.nh, self.no)

defrunNN (self, inputs):
iflen(inputs) != self.ni-1:
print('incorrect number of inputs')
for i in range(self.ni-1):
            self.ai[i] = inputs[i]
for j in range(self.nh):
            sum = 0.0
for i in range(self.ni):
            sum +=( self.ai[i] * self.wi[i][j] )
            self.ah[j] = sigmoid (sum)
for k in range(self.no):
            sum = 0.0
for j in range(self.nh):
            sum +=( self.ah[j] * self.wo[j][k] )
            self.ao[k] = sigmoid (sum)
return self.ao
```

```
defbackPropagate (self, targets, N, M):
output_deltas = [0.0] * self.no
for k in range(self.no):
        error = targets[k] - self.ao[k]
        output_deltas[k] =    error * dsigmoid(self.ao[k])
for j in range(self.nh):
for k in range(self.no):
        change = output_deltas[k] * self.ah[j]
        self.wo[j][k] += N*change + M*self.co[j][k]
                self.co[j][k] = change
        hidden_deltas = [0.0] * self.nh
for j in range(self.nh):
        error = 0.0
for k in range(self.no):
        error += output_deltas[k] * self.wo[j][k]
        hidden_deltas[j] = error * dsigmoid(self.ah[j])
for i in range (self.ni):
        for j in range (self.nh):
                change= hidden_deltas[j] * self.ai[i]
                self.wi[i][j] += N*change + M*self.ci[i][j]
                self.ci[i][j] = change
error = 0.0
for k in range(len(targets)):
error = 0.5 * (targets[k]-self.ao[k])**2
return error
def weights(self):
print('Input weights:')
for i in range(self.ni):
print (self.wi[i])
print()
print('Output weights:')
for j in range(self.nh):
print (self.wo[j])
print ('')
def test(self, patterns):
for p in patterns:
inputs = p[0]
print('Inputs:', p[0], '-->', self.runNN(inputs), '\tTarget', p[1])
def train (self, patterns, max_iterations = 1000, N=0.5, M=0.1):
for i in range(max_iterations):
for p in patterns:
inputs = p[0]
```

```
targets = p[1]
self.runNN(inputs)
error = self.backPropagate(targets, N, M)
if i % 50 == 0:
print('Combined error', error)
self.test(patterns)
def sigmoid (x):
returnmath.tanh(x)
defdsigmoid (y):
return 1 - y**2
defmakeMatrix ( I, J, fill=0.0):
    m = []
for i in range(I):
m.append([fill]*J)
return m
defrandomizeMatrix ( matrix, a, b):
for i in range ( len (matrix) ):
for j in range ( len (matrix[0]) ):
matrix[i][j] = random.uniform(a,b)
def main ():
pat = [
            [[0,0], [1]],
            [[0,1], [1]],
            [[1,0], [1]],
            [[1,1], [0]]
    ]
myNN = NN ( 2, 2, 1)
myNN.train(pat)
if __name__ == "__main__":
main()
```

OUTPUT:



```
In [2]:

In [2]: runfile('C:/Users/admin/.spyder-py3/back.py', wdir='C:/Users/
admin/.spyder-py3')
Combined error 0.33960029597171876
Combined error 0.005804347782481797
Combined error 0.002601213307011534
Combined error 0.0016867154033319899
Combined error 0.0012383865063792165
Combined error 0.0009752734271580881
Combined error 0.0008027801807789076
Combined error 0.0006820177235108346
Combined error 0.0005932187653819327
Combined error 0.0005238373154143884
Combined error 0.00046870901031911995
Combined error 0.0004243707440499534
Combined error 0.00038706606095840503
Combined error 0.00035610581102335867
Combined error 0.0003293302229051458
Combined error 0.0003064937660825221
Combined error 0.00028635023338388085
Combined error 0.0002688362269292783
Combined error 0.0002531466916495717
Combined error 0.00023927819235108606
Inputs: [0, 0] --> [0.9991494359910482]         Target [1]
Inputs: [0, 1] --> [0.9893815477575261]         Target [1]
Inputs: [1, 0] --> [0.9893589815027886]         Target [1]
Inputs: [1, 1] --> [-0.014869572264207365]      Target [0]

In [3]:
```

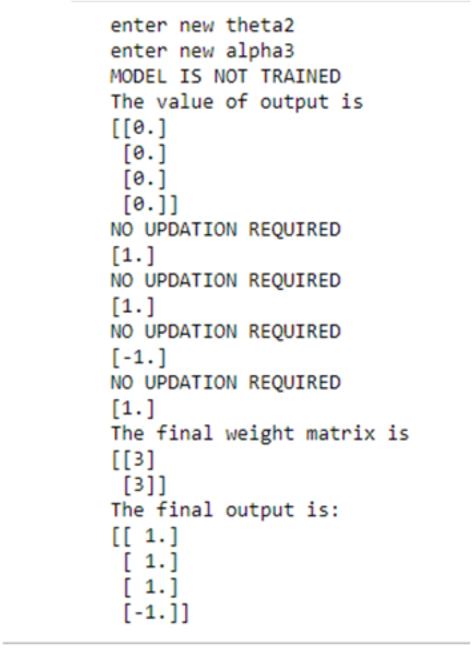
PROGRAM: 22

AIM: Write a program in Python to implement Madaline Neural Network.

CODE:

```
import numpy as np

x=np.array([[1,1],[1,-1],[-1,1],[-1,-1]])

t=np.array([[1],[1],[1],[-1]])

w=np.array([[0],[0]])

b=0

theta=float(input("enter new theta"))

alpha=float(input("enter new alpha"))

yin=np.zeros(shape=(4,1))

y=np.zeros(shape=(4,1))

i=0

found=0

while(found==0):

    yin=x[i][0]*w[0]+x[i][1]*w[1]

    yin = yin+b

    if(yin>theta):

        y[i] = 1

    elif(yin<=theta and yin>=-theta):

        y[i]=0

    else:

        y[i]=-1

    if (y[i]==t[i]):

        print("NO UPDATION REQUIRED")

        print(y[i])

        if(i<3):
            i=i+1
        else:
            i=0
    else:
        print("MODEL IS NOT TRAINED")
        print("The value of output is")
        print(y)

        w[0]=w[0]+alpha*x[i][0]*t[i]
        w[1]=w[1]+alpha*x[i][1]*t[i]
        b = b+alpha*t[i]
        if(i<3):
            i=i+1
        else:
            i=0
    if(y==t).all():
        found=1
print("The final weight matrix is ")
print(w)
print("The final output is:")
print(y)
```

OUTPUT:



```
enter new theta2
enter new alpha3
MODEL IS NOT TRAINED
The value of output is
[[0.]
 [0.]
 [0.]
 [0.]]
NO UPDATION REQUIRED
[1.]
NO UPDATION REQUIRED
[1.]
NO UPDATION REQUIRED
[-1.]
NO UPDATION REQUIRED
[1.]
The final weight matrix is
[[3]
 [3]]
The final output is:
[[ 1.]
 [ 1.]
 [ 1.]
 [-1.]]
```

PROGRAM: 23

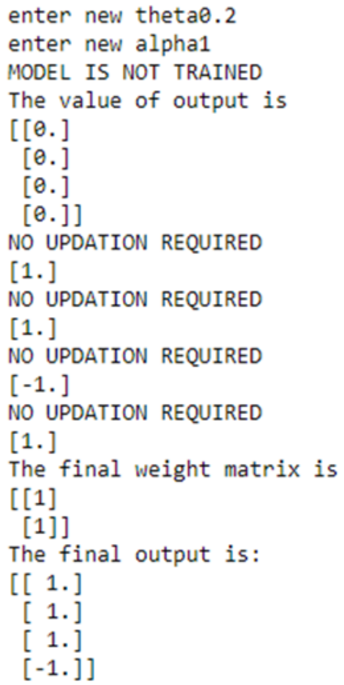
AIM: Write a program to implement the following logic functions using single layer Perceptron OR logic functions.

CODE:

```
#OR
importnumpy as np
x=np.array([[1,1],[1,-1],[-1,1],[-1,-1]])
t=np.array([[1],[1],[1],[-1]])
w=np.array([[0],[0]])
b=0
theta=float(input("enter new theta"))
alpha=float(input("enter new alpha"))
yin=np.zeros(shape=(4,1))
y=np.zeros(shape=(4,1))
i=0
found=0
while(found==0):
yin=x[i][0]*w[0]+x[i][1]*w[1]
yin = yin+b
if(yin>theta):
y[i] = 1
elif(yin<=theta and yin>=-theta):
y[i]=0
else:
y[i]=-1
if (y[i]==t[i]):
print("NO UPDATION REQUIRED")
print(y[i])
```

```
if(i<3):
i=i+1
else:
i=0
else:
print("MODEL IS NOT TRAINED")
print("The value of output is")
print(y)
w[0]=w[0]+alpha*x[i][0]*t[i]
w[1]=w[1]+alpha*x[i][1]*t[i]
                b = b+alpha*t[i]
if(i<3):
i=i+1
else:
i=0
if(y==t).all():
found=1
print("The final weight matrix is ")
print(w)
print("The final output is:")
print(y)
```

OUTPUT:



```
enter new theta0.2
enter new alpha1
MODEL IS NOT TRAINED
The value of output is
[[0.]
 [0.]
 [0.]
 [0.]]
NO UPDATION REQUIRED
[1.]
NO UPDATION REQUIRED
[1.]
NO UPDATION REQUIRED
[-1.]
NO UPDATION REQUIRED
[1.]
The final weight matrix is
[[1]
 [1]]
The final output is:
[[ 1.]
 [ 1.]
 [ 1.]
 [-1.]]
```

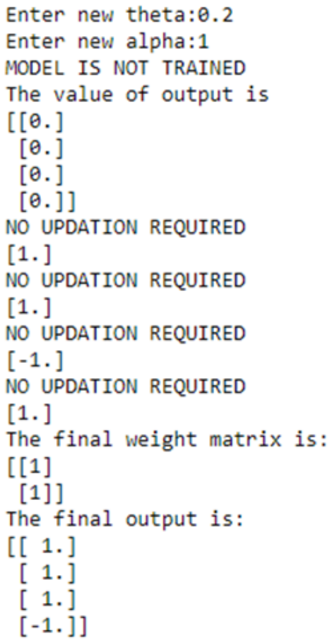
PROGRAM: 24

AIM: Write a program in Python to implement single layer perceptron for AND function.

CODE:

```
import numpy as np
x=np.array([[1,1],[1,-1],[-1,1],[-1,-1]])
t=np.array([[1],[1],[1],[-1]])
w=np.array([[0],[0]])
b=0
theta=float(input("Enter new theta:"))
alpha=float(input("Enter new alpha:"))
yin=np.zeros(shape=(4,1))
y=np.zeros(shape=(4,1))
i=0
found=0
while(found==0):
    yin=x[i][0]*w[0]+x[i][1]*w[1]
    yin = yin+b
    if(yin>theta):
        y[i] = 1
    elif(yin<=theta and yin>=-theta):
        y[i]=0
    else:
        y[i]=-1
    if (y[i]==t[i]):
        print("NO UPDATION REQUIRED")
        print(y[i])
        if(i<3):
            i=i+1
        else:
            i=0
    else:
        print("MODEL IS NOT TRAINED")
        print("The value of output is")
        print(y)
        w[0]=w[0]+alpha*x[i][0]*t[i]
        w[1]=w[1]+alpha*x[i][1]*t[i]
        b = b+alpha*t[i]
        if(i<3):
            i=i+1
        else:
            i=0
    if(y==t).all():
        found=1
print("The final weight matrix is:")
print(w)
print("The final output is:")
print(y)
```

OUTPUT:



```
Enter new theta:0.2
Enter new alpha:1
MODEL IS NOT TRAINED
The value of output is
[[0.]
 [0.]
 [0.]
 [0.]]
NO UPDATION REQUIRED
[1.]
NO UPDATION REQUIRED
[1.]
NO UPDATION REQUIRED
[-1.]
NO UPDATION REQUIRED
[1.]
The final weight matrix is:
[[1]
 [1]]
The final output is:
[[ 1.]
 [ 1.]
 [ 1.]
 [-1.]]
```

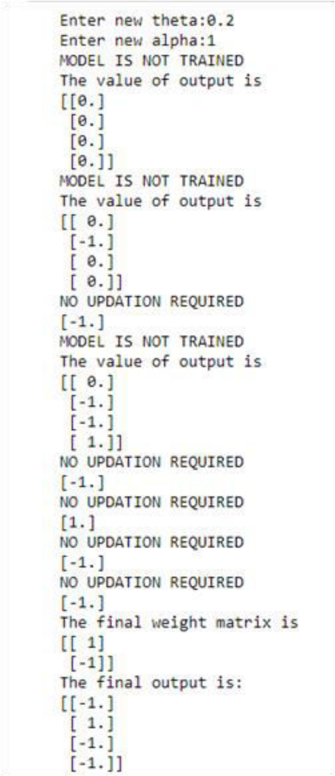
PROGRAM: 25

AIM: Write a program in Python to implement single layer perceptron for ANDNOT function.

CODE:

```
import numpy as np
x=np.array([[1,1],[1,-1],[-1,1],[-1,-1]])
t=np.array([[-1],[1],[-1],[-1]])
w=np.array([[0],[0]])
b=0
theta=float(input("Enter new theta:"))
alpha=float(input("Enter new alpha:"))
yin=np.zeros(shape=(4,1))
y=np.zeros(shape=(4,1))
i=0
found=0
while(found==0):
    yin=x[i][0]*w[0]+x[i][1]*w[1]
    yin = yin+b
    if(yin>theta):
        y[i] = 1
    elif(yin<=theta and yin>=-theta):
        y[i]=0
    else:
        y[i]=-1
    if (y[i]==t[i]):
        print("NO UPDATION REQUIRED")
        print(y[i])
        if(i<3):
            i=i+1
        else:
            i=0
    else:
        print("MODEL IS NOT TRAINED")
        print("The value of output is")
        print(y)
        w[0]=w[0]+alpha*x[i][0]*t[i]
        w[1]=w[1]+alpha*x[i][1]*t[i]
        b = b+alpha*t[i]
        if(i<3):
            i=i+1
        else:
            i=0
    if(y==t).all():
        found=1
print("The final weight matrix is ")
print(w)
print("The final output is:")
print(y)
```

OUTPUT:



```
Enter new theta:0.2
Enter new alpha:1
MODEL IS NOT TRAINED
The value of output is
[[0.]
 [0.]
 [0.]
 [0.]]
MODEL IS NOT TRAINED
The value of output is
[[ 0.]
 [-1.]
 [ 0.]
 [ 0.]]
NO UPDATION REQUIRED
[-1.]
MODEL IS NOT TRAINED
The value of output is
[[ 0.]
 [-1.]
 [-1.]
 [ 1.]]
NO UPDATION REQUIRED
[-1.]
NO UPDATION REQUIRED
[1.]
NO UPDATION REQUIRED
[-1.]
NO UPDATION REQUIRED
[-1.]
The final weight matrix is
[[ 1]
 [-1]]
The final output is:
[[-1.]
 [ 1.]
 [-1.]
 [-1.]]
```

PROGRAM: 26

AIM: Write a program in Python to implement Hopfield neural network

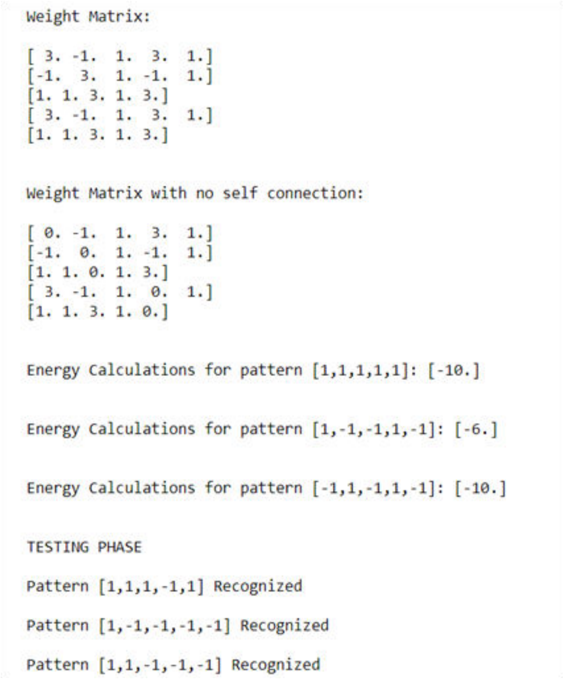
CODE:

```
importnumpy as np
x=np.array([[1,1,1,1,1],[1,-1,-1,1,-1],[-1,1,-1,-1,-1]])
x1=np.transpose(x)
t1=np.array([[1,1,1,-1,1]])
t2=np.array([[1,-1,-1,-1,-1]])
t3=np.array([[1,1,-1,-1,-1]])
w=np.zeros((5,5))
i=0
j=0
k=0
for i in range(len(x1)):
for j in range(len(x[0])):
for k in range(len(x)):
w[i][j] += x1[i][k] * x[k][j]
print('Weight Matrix:\n')
for r in w:
print(r)
print('\n\nWeight Matrix with no self connection:\n')
i=0
j=0
for i in range(int(5)):
for j in range(int(5)):
if(i==j):
w[i][j]=0
```

```
for r in w:
print(r)
E1=0
E2=0
E3=0
x11= x[0].reshape(5,1)
x12=x[1].reshape(5,1)
x13=x[2].reshape(5,1)
E1= -0.5 * np.matmul(x[0],np.matmul(w,x11))
print('\n\nEnergy Calculations for pattern [1,1,1,1,1]:',E1)

E2= -0.5 * np.matmul(x[1],np.matmul(w,x12))
print('\n\nEnergy Calculations for pattern [1,-1,-1,1,-1]:',E2)

E3= -0.5 * np.matmul(x[2],np.matmul(w,x13))
print('\n\nEnergy Calculations for pattern [-1,1,-1,1,-1]:',E3)

print('\n\nTESTING PHASE')
w_dash=np.transpose(w)
Yin1=t1[0][3]+ np.matmul(x[0],w_dash[3])
if(Yin1>0):
t1[0][3]=1
else:
t1[0][3]=-1
if((t1==x).any()):
print('\nPattern [1,1,1,-1,1] Recognized ')
else:
print('\nPattern [1,1,1,-1,1] not Recognized ')
```

```
Yin2=t2[0][3]+ np.matmul(x[1],w_dash[3])

if(Yin2>0):

t2[0][3]=1

else:

t2[0][3]=-1

if((t2==x).any()):

print('\nPattern [1,-1,-1,-1,-1] Recognized ')

else:

print('\nPattern [1,-1,-1,-1,-1] not Recognized ')

Yin3=t3[0][0]+ np.matmul(x[2],w_dash[0])

if(Yin3>0):

t3[0][0]=1

else:

t3[0][0]=-1

if((t3==x).any()):

print('\nPattern [1,1,-1,-1,-1] Recognized ')

else:

print('\nPattern [1,1,-1,-1,-1] not Recognized ')
```

OUTPUT:



```
Weight Matrix:

[ 3. -1.  1.  3.  1.]
[-1.  3.  1. -1.  1.]
[1. 1. 3. 1. 3.]
[ 3. -1.  1.  3.  1.]
[1. 1. 3. 1. 3.]


Weight Matrix with no self connection:

[ 0. -1.  1.  3.  1.]
[-1.  0.  1. -1.  1.]
[1. 1. 0. 1. 3.]
[ 3. -1.  1.  0.  1.]
[1. 1. 3. 1. 0.]


Energy Calculations for pattern [1,1,1,1,1]: [-10.]


Energy Calculations for pattern [1,-1,-1,1,-1]: [-6.]


Energy Calculations for pattern [-1,1,-1,1,-1]: [-10.]


TESTING PHASE

Pattern [1,1,1,-1,1] Recognized

Pattern [1,-1,-1,-1,-1] Recognized

Pattern [1,1,-1,-1,-1] Recognized
```

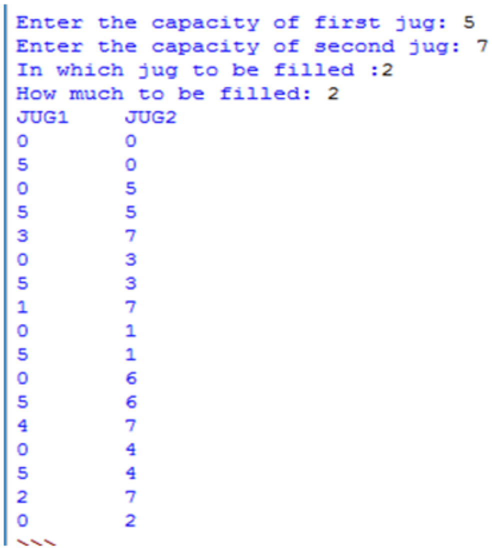
PROGRAM: 27

AIM: Write a program to implement water jug problem with two jugs the capacity of both the jugs should be entered by user. The quantity of the water to be stored should also be dynamic. The output will show all the steps to get the final state.

CODE:

```
n1=int(input("Enter the capacity of first jug: "))

n2=int(input("Enter the capacity of second jug: "))

n3=int(input("In which jug to be filled :"))

n4=int(input("How much to be filled: "))

class Waterjug:

    def __init__(self,am,bm,a,b,g):

        self.a_max = am;

        self.b_max = bm;

        self.a = a;

        self.b = b;

        self.goal = g;

    def fillA(self):

        self.a = self.a_max;

        print ('(', self.a, ',',self.b, ')')

    def fillB(self):

        self.b = self.b_max;

        print ('(', self.a, ',', self.b, ')')

    def emptyA(self):

        self.a = 0;

        print ('(', self.a, ',', self.b, ')')

    def emptyB(self):

        self.b = 0;

        print ('(', self.a, ',', self.b, ')')
    def transferAtoB(self):
        while (True):
            self.a = self.a - 1
            self.b = self.b + 1
            if (self.a == 0 or self.b == self.b_max):
                break
        print ('(', self.a, ',', self.b, ')')
    def main(self):
        while (True):
            if (self.a == self.goal or self.b == self.goal):
                break
            if (self.a == 0):
                self.fillA()
            elif (self.a > 0 and self.b != self.b_max):
                self.transferAtoB()
            elif (self.a > 0 and self.b == self.b_max):
                self.emptyB()
def pour(jug1, jug2):
       max1, max2, fill = n1, n2, n4
       print("%d\t%d" % (jug1, jug2))
       if jug2 is fill:
         return       elif jug2 is max2:

          pour(0, jug1)
       elif jug1 != 0 and jug2 is 0:
          pour(0, jug1)
       elif jug1 is fill:
         pour(jug1, 0)
        elif jug1 < max1:
          pour(max1, jug2)
        elif jug1 < (max2-jug2):
          pour(0, (jug1+jug2))
        else:
          pour(jug1-(max2-jug2), (max2-jug2)+jug2)
print("JUG1\tJUG2")
if(n3==2):
    pour(0, 0)
elif(n3==1):
  print ('(', '0',',', '0', ')')
  waterjug=Waterjug(n1,n2,0,0,n4);
  waterjug.main();
```

OUTPUT:



```
Enter the capacity of first jug: 5
Enter the capacity of second jug: 7
In which jug to be filled :2
How much to be filled: 2
JUG1    JUG2
0       0
5       0
0       5
5       5
3       7
0       3
5       3
1       7
0       1
5       1
0       6
5       6
4       7
0       4
5       4
2       7
0       2
```