1. Implement the Propositional basic logic gates along with implies and biconditional

Code to implement AND gate

```
def AND (a, b):
  if a == 1 and b == 1:
    return True
  else:
    return False
# main function
if __name__=='__main__':
  print(AND(0,0))
  print(AND(1,0))
  print(AND(0,1))
  print(AND(1,1))
```

OUTPUT:

```
In [1]: def AND (a, b):
           if a == 1 and b == 1:
              return True
           else:
              return False

        # main function
        if __name__=='__main__':
           print(AND(0,0))
           print(AND(1,0))
           print(AND(0,1))
           print(AND(1,1))

False
False
False
True

In [ ]:
```

Code to implement OR gate

```
def OR(a, b):
  if a == 1:
    return True
```

```
  elif b == 1:
    return True
  else:
    return False
# main function
if __name__=='__main__':
  print(OR(0,0))
  print(OR(1,0))
  print(OR(0,1))
  print(OR(1,1))
```

OUTPUT:

```
In [2]: def OR(a, b):
            if a == 1:
                return True
            elif b == 1:
                return True
            else:
                return False
        # main function
        if __name__=='__main__':
            print(OR(0,0))
            print(OR(1,0))
            print(OR(0,1))
            print(OR(1,1))

        False
        True
        True
        True
```

Code to implement NOT gate

```
def NOT(a):
  if(a == 0):
    return 1
  elif(a == 1):
    return 0
```

```
# main function
if __name__=='__main__':
  print(NOT(0))
  print(NOT(1))
```

OUTPUT:

```
In [4]: def NOT(a):
          if(a == 0):
            return 1
          elif(a == 1):
            return 0

        # main function
        if __name__=='__main__':
          print(NOT(0))
          print(NOT(1))

        1
        0
```

2. Design the simplex reflex vacuum cleaner

```
flag=True
count=1
while flag:
   perc=input("enter the percept\n")
   loc=input("enter the location\n")
   if loc=="A":
      if perc=="dirty":
         print("action: suck...turn right")

      else:
         print("action: turn right")
```

```
else:
   if perc=="dirty":
      print("action: suck.....turn left")

   else:
      print("action: turn left")

print("Do you want to continue?")
Cont=input("Enter Y or N")
if Cont == 'Y':
    flag= True
else :
    flag = False
```

OUTPUT:

```
print("Do you want to continue?")
Cont=input("Enter Y or N")
if Cont == 'Y':
    flag= True
else :
    flag = False
```

```
enter the percept
dirty
enter the location
A
action: suck...turn right
Do you want to continue?
Enter Y or NY
enter the percept
clean
enter the location
B
action: turn left
Do you want to continue?
Enter Y or NN
```

3. Assume that there are 3 floors and 4 rooms in each floor. Design the vacuum cleaner to ensure the rooms are clean. You may make suitable assumption for initial state.

```
# Given M x N grid(floor) create an agent that moves around the grid
until the entire grid is clean

floor = [[1, 0, 0, 0], # '1' represents dirty and '0' represents
clean
         [0, 1, 0, 1],
         [1, 0, 1, 1]]


def clean(floor):
    m = len(floor[0]) # no of cols
    n = len(floor)    # no of rows
    no_of_tiles = m * n

    tiles_checked = 0

    row = 0
    col = 0
    while tiles_checked < no_of_tiles:
        # Current position
        print_floor(floor, row, col)

        # Suck if dirty
        if floor[row][col] == 1:
            floor[row][col] = 0
            print('Sucked the dirt')
        else:
            print('Already Clean')


        # Next tile
        if row % 2 == 0:          # Even rows the bot moves right to
the next tile
            if col < m-1:
                col += 1
            else:
                row += 1  # Move to next row if we reached the last
col

        elif row % 2 == 1:        # Odd rows the bot moves left to
the next tile
            if 0 < col:
                col -= 1
            else:
                row += 1  # Move to next row if we reached the last
col
```

```
        tiles_checked += 1
        print('---------------')

    print('Cleaned!!!')



def print_floor(floor, row, col):
    temp = floor[row][col]
    floor[row][col] = 'VC'
    for x in floor:
        print(x)

    floor[row][col] = temp

# Call the function
clean(floor)
```

OUTPUT:

```
--------------
[0, 0, 0, 0]
['VC', 0, 0, 0]
[1, 0, 1, 1]
Already Clean
--------------
[0, 0, 0, 0]
[0, 0, 0, 0]
['VC', 0, 1, 1]
Sucked the dirt
--------------
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 'VC', 1, 1]
Already Clean
--------------
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 0, 'VC', 1]
Sucked the dirt
--------------
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 0, 0, 'VC']
Sucked the dirt
--------------
Cleaned!!!
```

```
--------------
[0, 0, 0, 0]
['VC', 0, 0, 0]
[1, 0, 1, 1]
Already Clean
--------------
[0, 0, 0, 0]
[0, 0, 0, 0]
['VC', 0, 1, 1]
Sucked the dirt
--------------
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 'VC', 1, 1]
Already Clean
--------------
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 0, 'VC', 1]
Sucked the dirt
--------------
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 0, 0, 'VC']
Sucked the dirt
--------------
Cleaned!!!
```

4. Consider S and T as variables and the following relation representing the relationships:

(i) a: ¬(SVT)

(ii) b: (S&T)

(iii) c: TV¬T

(iv) d: ¬(S⍰S)

(v) e: ¬S ¬T

Analyze the following for PL-TT entailment and show whether

(i). ‘a’ entails ‘b’,

(ii). ‘a’ entails ’c’,

(iii). ‘a’ entails ’d’ and

(iv). ‘a’ entails ’e’

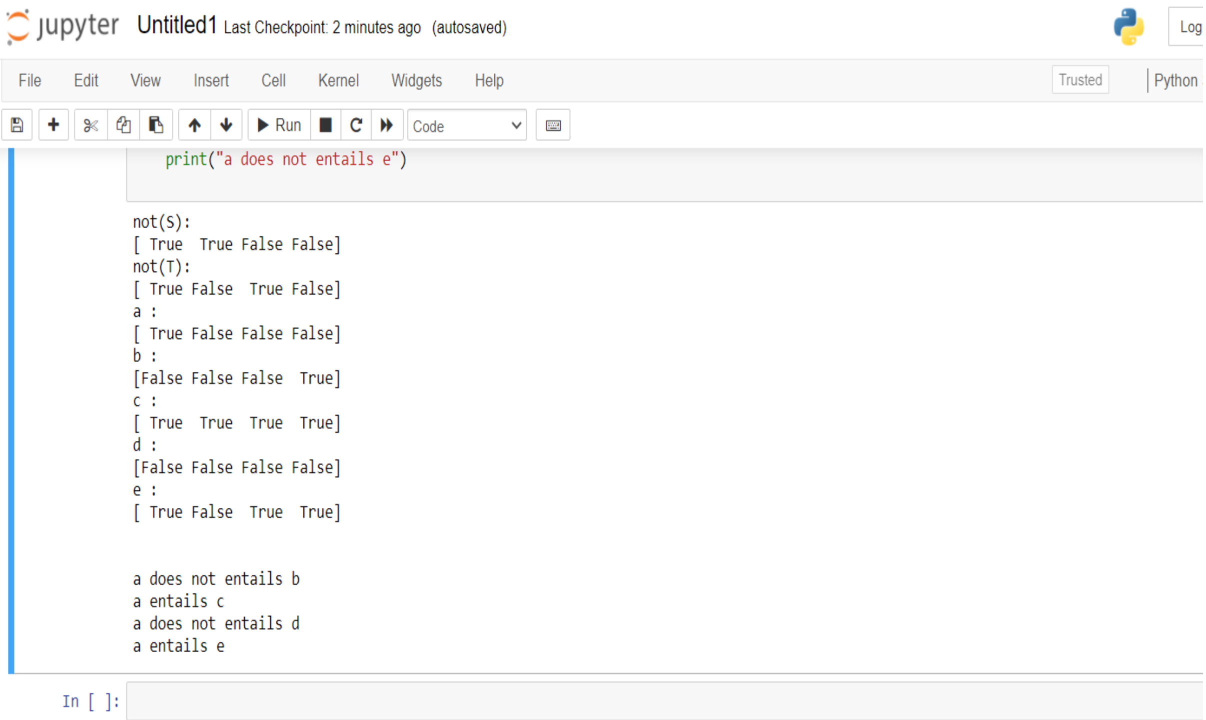
Code

```
import numpy as np



S=np.array([0,0,1,1])
T=np.array([0,1,0,1])

# not(s) , not(t)
X=np.logical_not(S)
Y=np.logical_not(T)

#not(s or t)
OR=np.logical_or(S,T)
a=np.logical_not(OR)

#(s and t)
b=np.logical_and(S,T)

#(t or not(t))
c=np.logical_or(T,Y)

#not(s<->s)
#(not(s) or s) and (s or not(s))

or1=np.logical_or(X,S)
or2=np.logical_or(S,X)
and1=np.logical_and(or1,or2)
d=np.logical_not(and1)
```

```
#not(s)->not(t)  here not(s) is x and not(t) is y
# so x->y can be reduced to not(x) or y
# not(x) is same as s
e=np.logical_or(S,Y)


print("not(S): ")
print(X)
print("not(T):")
print(Y)
print("a : ")
print(a)
print("b : ")
print(b)
print("c : ")
print(c)
print("d : ")
print(d)
print("e : ")
print(e)
print("\n")

if any(np.logical_and(a,b)):
    print("a entails b")
else:
    print("a does not entails b")


if any(np.logical_and(a,c)):
```

```
    print("a entails c")
else:
    print("a does not entails c")


if any(np.logical_and(a,d)):
    print("a entails d")
else:
    print("a does not entails d")


if any(np.logical_and(a,e)):
    print("a entails e")
else:
    print("a does not entails e")
```

OUTPUT



5. Implement the 8-puzzle problem using A* algorithm, using Heuristic function as Manhattan distance with depth not more the 3. If goal state is not reached within this limit, agent must report "NOSOLUTION".

8 2 3

4 6

7 5 1

Start state

1 2 3

4 5 6

7 8

Goal State

Code:

```
GoalNode=[[1,2,3],[4,5,6],[7,8,0]]

StartNode=[[8,2,3],[0,4,6],[7,5,1]]

temp = []

h1 = -1

h2 = 0


print("Given StartNode is: ",StartNode)


print("\n\n\t Given GoalNode is: ",GoalNode)


print("\n\n######################################")


for i in range(len(StartNode)):
```

```
    for j in range (len(StartNode)):
        if StartNode[i][j] != GoalNode[i][j]:
            h1+=1
print("\n\n\t h1 : Number of misplaced tiles =>",h1)




'''
for i in StartNode:
    for j in i:
        print("StartNode",j)

print("########################################")
for i in GoalNode:
    for j in i:
        print("GoalNode",j)
print("########################################")
for i in range(len(StartNode)):
    for j in range (len(StartNode)):
        print("i is ",i,"j is :",j)'''

print("\n\n########################################")

print("\n\nDistances of the tiles from their goal positions are: \n")

for i in range(len(StartNode)):
    for j in range (len(StartNode)):
        if (StartNode[i][j]==0):
            pass
```

```
else:
    if (GoalNode[0][0] == StartNode[i][j]):
        temp.append(abs(i-0) + abs(j-0))
        print("\t",temp)

    elif (GoalNode[0][1] == StartNode[i][j]):
        temp.append(abs(i-0) + abs(j-1))
        print("\t",temp)
    elif (GoalNode[0][2] == StartNode[i][j]):
        temp.append(abs(i-0) + abs(j-2))
        print("\t",temp)
    elif (GoalNode[1][0] == StartNode[i][j]):
        temp.append(abs(i-1) + abs(j-0))
        print("\t",temp)
    elif (GoalNode[1][1] == StartNode[i][j]):
        temp.append(abs(i-1) + abs(j-1))
        print("\t",temp)
    elif (GoalNode[1][2] == StartNode[i][j]):
        temp.append(abs(i-1) + abs(j-2))
        print("\t",temp)
    elif (GoalNode[2][0] == StartNode[i][j]):
        temp.append(abs(i-2) + abs(j-0))
        print("\t",temp)
    elif (GoalNode[2][1] == StartNode[i][j]):
        temp.append(abs(i-2) + abs(j-1))
        print("\t",temp)
    elif (GoalNode[2][2] == StartNode[i][j]):
        temp.append(abs(i-2) + abs(j-2))
        print("\t",temp)
```

```python
        else:
            print("Warning!!! This is for 8-puzzle program.So, don't cross the array limit.")



print("\n\n######################################")

for i in range(len(temp)):
   h2+=temp[i]
print("\nh2 :  The sum of the distances of the tiles from their goal positions =>",h2)

h=h1+h2

print("\n\n\tSo, the instance of given 8-puzzle solution is",h,"steps long.")
```

OUTPUT

```
Given StartNode is:  [[8, 2, 3], [0, 4, 6], [7, 5, 1]]

        Given GoalNode is:  [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

######################################

        h1 : Number of misplaced tiles => 4

######################################

Distances of the tiles from their goal positions are:

        [3]
        [3, 0]
        [3, 0, 0]
        [3, 0, 0, 1]
        [3, 0, 0, 1, 0]
        [3, 0, 0, 1, 0, 0]
        [3, 0, 0, 1, 0, 0, 1]
        [3, 0, 0, 1, 0, 0, 1, 4]

######################################

h2 :  The sum of the distances of the tiles from their goal positions => 9
```

6. You are given two jugs, a 4-litre one and a 3-litre one. Neither has any measuring markers on it. There is a pump that can be used to fill the jugs with water. How can you get exactly 2 litres of water into 4-litre jug? Implement this using Depth First Search.

CODE

```
def pour(jug1, jug2):
    max1, max2, fill = 3, 4, 2  #Change maximum capacity and final capacity
    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")
pour(0, 0)
```

OUTPUT

jupyter Untitled6 Last Checkpoint: a few seconds ago (unsaved changes)

File Edit View Insert Cell Kernel Widgets Help Trusted Python 3

```python
    max1, max2, fill = 3, 4, 2 #change maximum capacity and final capacity
    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")
pour(0, 0)
```

```
JUG1	JUG2
0	0
3	0
0	3
3	3
2	4
0	2
```