**LAB 1:**

board = [' ' for x in range(10)]

def insertLetter(letter, pos):

board[pos] = letter

def spaceIsFree(pos):

return board[pos] == ' '

def printBoard(board):

print(' | |')

print(' ' + board[1] + ' | ' + board[2] + ' | ' + board[3])

print(' | |')

print('-----------')

print(' | |')

print(' ' + board[4] + ' | ' + board[5] + ' | ' + board[6])

print(' | |')

print('-----------')

print(' | |')

print(' ' + board[7] + ' | ' + board[8] + ' | ' + board[9])

print(' | |')

def isWinner(bo, le):

return (bo[7] == le and bo[8] == le and bo[9] == le) or (bo[4] == le and bo[5] == le and bo[6] == le) or (

bo[1] == le and bo[2] == le and bo[3] == le) or (bo[1] == le and bo[4] == le and bo[7] == le) or (

bo[2] == le and bo[5] == le and bo[8] == le) or (

bo[3] == le and bo[6] == le and bo[9] == le) or (

bo[1] == le and bo[5] == le and bo[9] == le) or (bo[3] == le and bo[5] == le and bo[7] == le)

def playerMove():

run = True

while run:

move = input('Please select a position to place an \'X\' (1-9): ')

try:

move = int(move)

if move > 0 and move < 10:

if spaceIsFree(move):

run = False

insertLetter('X', move)

else:

print('Sorry, this space is occupied!')

else:

print('Please type a number within the range!')

except:

print('Please type a number!')

def compMove():

possibleMoves = [x for x, letter in enumerate(board) if letter == ' ' and x != 0]

move = 0

for let in ['O', 'X']:

for i in possibleMoves:

boardCopy = board[:]

boardCopy[i] = let

if isWinner(boardCopy, let):

move = i

return move

cornersOpen = []

for i in possibleMoves:

if i in [1, 3, 7, 9]:

cornersOpen.append(i)

if len(cornersOpen) > 0:

move = selectRandom(cornersOpen)

return move

if 5 in possibleMoves:

move = 5

return move

edgesOpen = []

for i in possibleMoves:

if i in [2, 4, 6, 8]:

edgesOpen.append(i)

if len(edgesOpen) > 0:

move = selectRandom(edgesOpen)

return move

def selectRandom(li):

import random

ln = len(li)

r = random.randrange(0, ln)

return li[r]

def isBoardFull(board):

if board.count(' ') > 1:

return False

else:

return True

def main():

print('Welcome to Tic Tac Toe!')

printBoard(board)

while not (isBoardFull(board)):

if not (isWinner(board, 'O')):

playerMove()

printBoard(board)

else:

print('Sorry, O\'s won this time!')

break

if not (isWinner(board, 'X')):

move = compMove()

if move == 0:

print('Tie Game!')

else:

insertLetter('O', move)

print('Computer placed an \'O\' in position', move, ':')

printBoard(board)

else:

print('X\'s won this time! Good Job!')

break

if isBoardFull(board):

print('Tie Game!')

while True:

answer = input('Do you want to play again? (Y/N)')

if answer.lower() == 'y' or answer.lower == 'yes':

board = [' ' for x in range(10)]

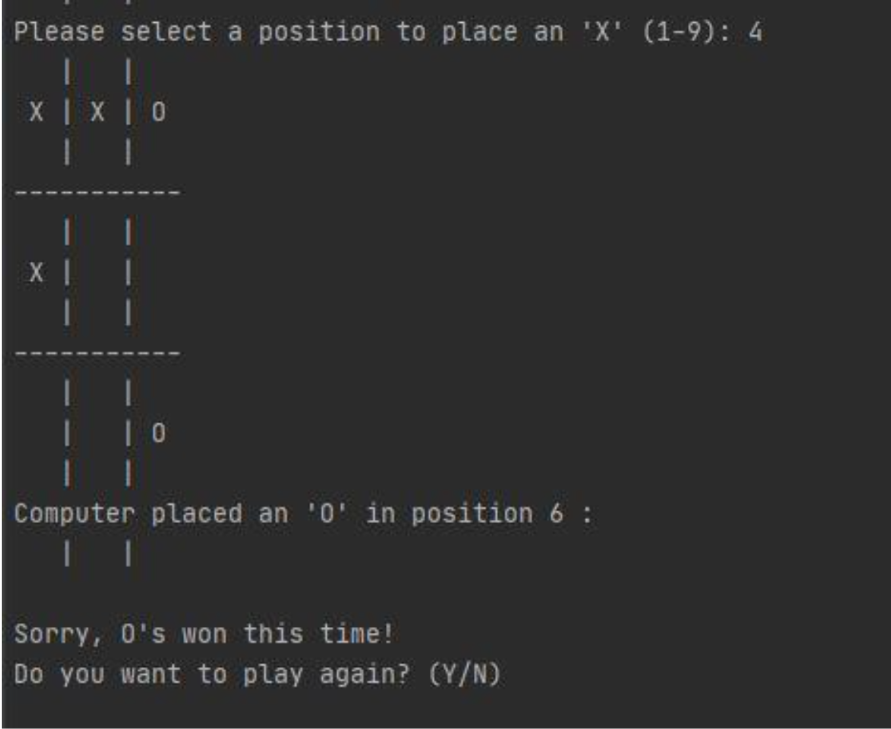
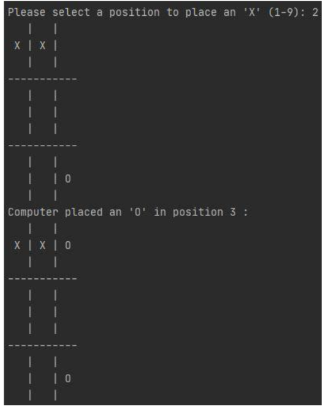
print('-----------------------------------')

main()

else:

break

**OUTPUT:**

****

**LAB 2:**

class Node:

def \_\_init\_\_(self, data, level, fval):

""" Initialize the node with the data, level of the node and the calculated fvalue """

self.data = data

self.level = level

self.fval = fval

def generate\_child(self):

""" Generate child nodes from the given node by moving the blank space

either in the four directions {up,down,left,right} """

x, y = self.find(self.data, '\_')

""" val\_list contains position values for moving the blank space in either of

the 4 directions [up,down,left,right] respectively. """

val\_list = [[x, y - 1], [x, y + 1], [x - 1, y], [x + 1, y]]

children = []

for i in val\_list:

child = self.shuffle(self.data, x, y, i[0], i[1])

if child is not None:

child\_node = Node(child, self.level + 1, 0)

children.append(child\_node)

return children

def shuffle(self, puz, x1, y1, x2, y2):

""" Move the blank space in the given direction and if the position value are out

of limits the return None """

if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):

temp\_puz = []

temp\_puz = self.copy(puz)

temp = temp\_puz[x2][y2]

temp\_puz[x2][y2] = temp\_puz[x1][y1]

temp\_puz[x1][y1] = temp

return temp\_puz

else:

return None

def copy(self, root):

""" Copy function to create a similar matrix of the given node"""

temp = []

for i in root:

t = []

for j in i:

t.append(j)

temp.append(t)

return temp

def find(self, puz, x):

""" Specifically used to find the position of the blank space """

for i in range(0, len(self.data)):

for j in range(0, len(self.data)):

if puz[i][j] == x:

return i, j

class Puzzle:

def \_\_init\_\_(self, size):

""" Initialize the puzzle size by the specified size,open and closed lists to empty """

self.n = size

self.open = []

self.closed = []

def accept(self):

""" Accepts the puzzle from the user """

puz = []

for i in range(0, self.n):

temp = input().split(" ")

puz.append(temp)

return puz

def f(self, start, goal):

""" Heuristic Function to calculate hueristic value f(x) = h(x) + g(x) """

return self.h(start.data, goal) + start.level

def h(self, start, goal):

""" Calculates the different between the given puzzles """

temp = 0

for i in range(0, self.n):

for j in range(0, self.n):

if start[i][j] != goal[i][j] and start[i][j] != '\_':

temp += 1

return temp

def process(self):

""" Accept Start and Goal Puzzle state"""

print("Enter the start state matrix \n")

start = self.accept()

print("Enter the goal state matrix \n")

goal = self.accept()

start = Node(start, 0, 0)

start.fval = self.f(start, goal)

""" Put the start node in the open list"""

self.open.append(start)

print("\n\n")

while True:

cur = self.open[0]

print("")

print(" | ")

print(" | ")

print(" \\\'/ \n")

for i in cur.data:

for j in i:

print(j, end=" ")

print("")

""" If the difference between current and goal node is 0 we have reached the goal node"""

if (self.h(cur.data, goal) == 0):

break

for i in cur.generate\_child():

i.fval = self.f(i, goal)

self.open.append(i)

self.closed.append(cur)

del self.open[0]

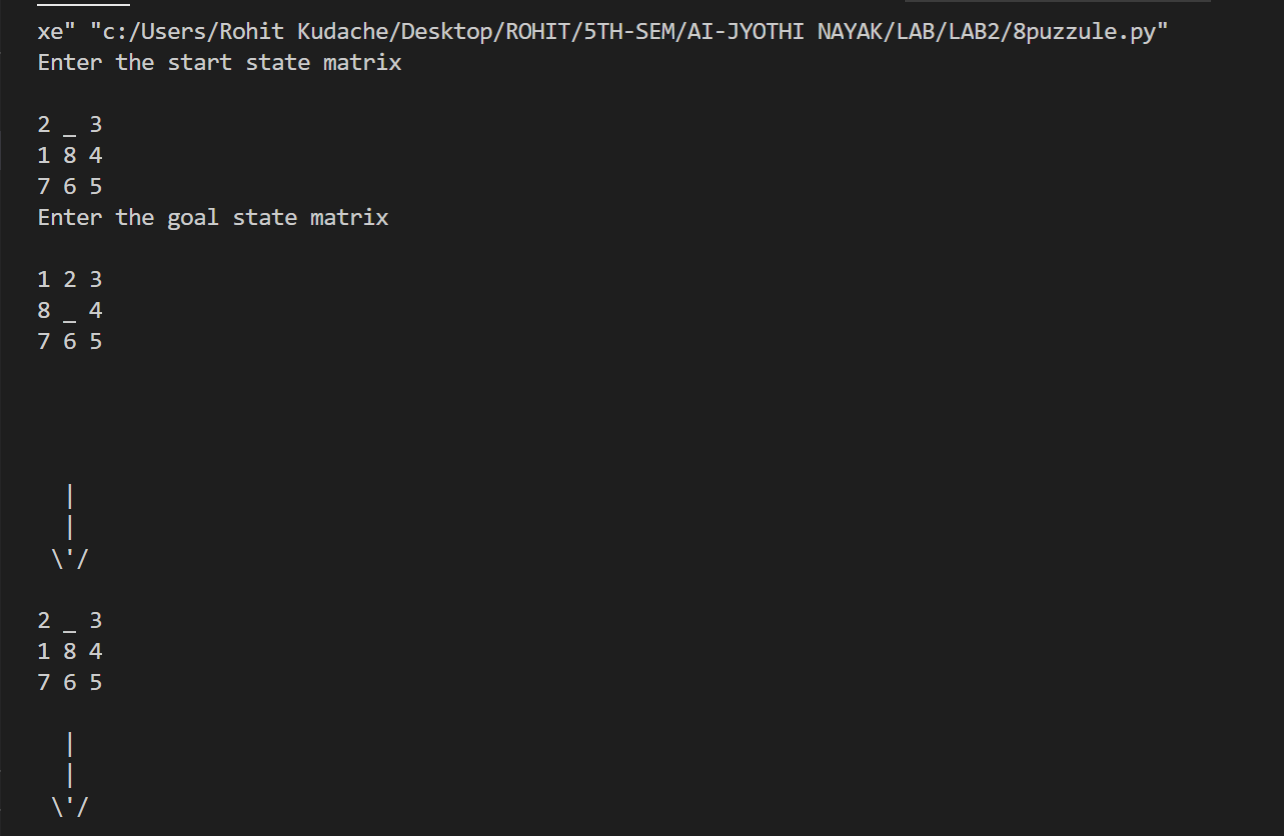
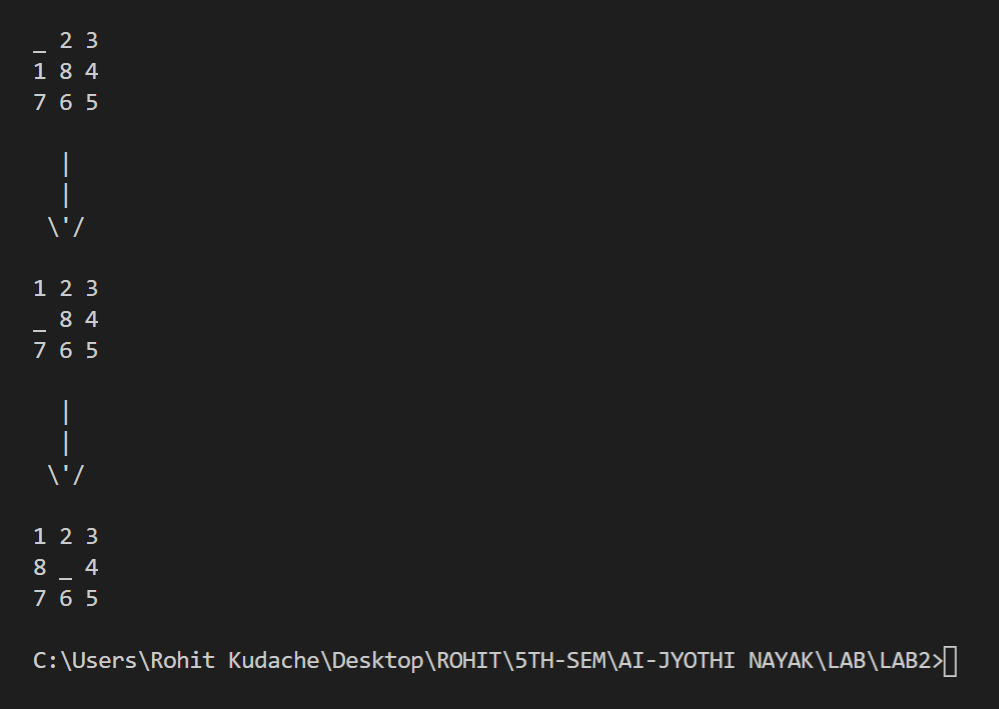
""" sort the open list based on f value """

self.open.sort(key=lambda x: x.fval, reverse=False)

puz = Puzzle(3)

puz.process()

**OUTPUT:**

****

**LAB 3:**

def dfs(src,target,limit,visited\_states):

if src == target:

return True

if limit <= 0:

return False

visited\_states.append(src)

moves = possible\_moves(src,visited\_states)

for move in moves:

if dfs(move, target, limit-1, visited\_states):

return True

return False

def possible\_moves(state,visited\_states):

b = state.index(-1)

d = []

if b not in [0,1,2]:

d += 'u'

if b not in [6,7,8]:

d += 'd'

if b not in [2,5,8]:

d += 'r'

if b not in [0,3,6]:

d += 'l'

pos\_moves = []

for move in d:

pos\_moves.append(gen(state,move,b))

return [move for move in pos\_moves if move not in visited\_states]

def gen(state, move, blank):

temp = state.copy()

if move == 'u':

temp[blank-3], temp[blank] = temp[blank], temp[blank-3]

if move == 'd':

temp[blank+3], temp[blank] = temp[blank], temp[blank+3]

if move == 'r':

temp[blank+1], temp[blank] = temp[blank], temp[blank+1]

if move == 'l':

temp[blank-1], temp[blank] = temp[blank], temp[blank-1]

return temp

def iddfs(src,target,depth):

for i in range(depth):

visited\_states = []

if dfs(src,target,i+1,visited\_states):

return True

return False

#Test 1

src = [1,2,3,-1,4,5,6,7,8]

target = [1,2,3,4,5,-1,6,7,8]

depth = 1

iddfs(src, target, depth)

#Test 2

src = [3,5,2,8,7,6,4,1,-1]

target = [-1,3,7,8,1,5,4,6,2]

depth = 1

iddfs(src, target, depth)

# Test 2

src = [1,2,3,-1,4,5,6,7,8]

target=[1,2,3,6,4,5,-1,7,8]

depth = 1

iddfs(src, target, depth)

src = [1, 2, 3, 4, 5, 6, 7, 8, -1]

target = [-1, 1, 2, 3, 4, 5, 6, 7, 8]

for i in range(1, 100):

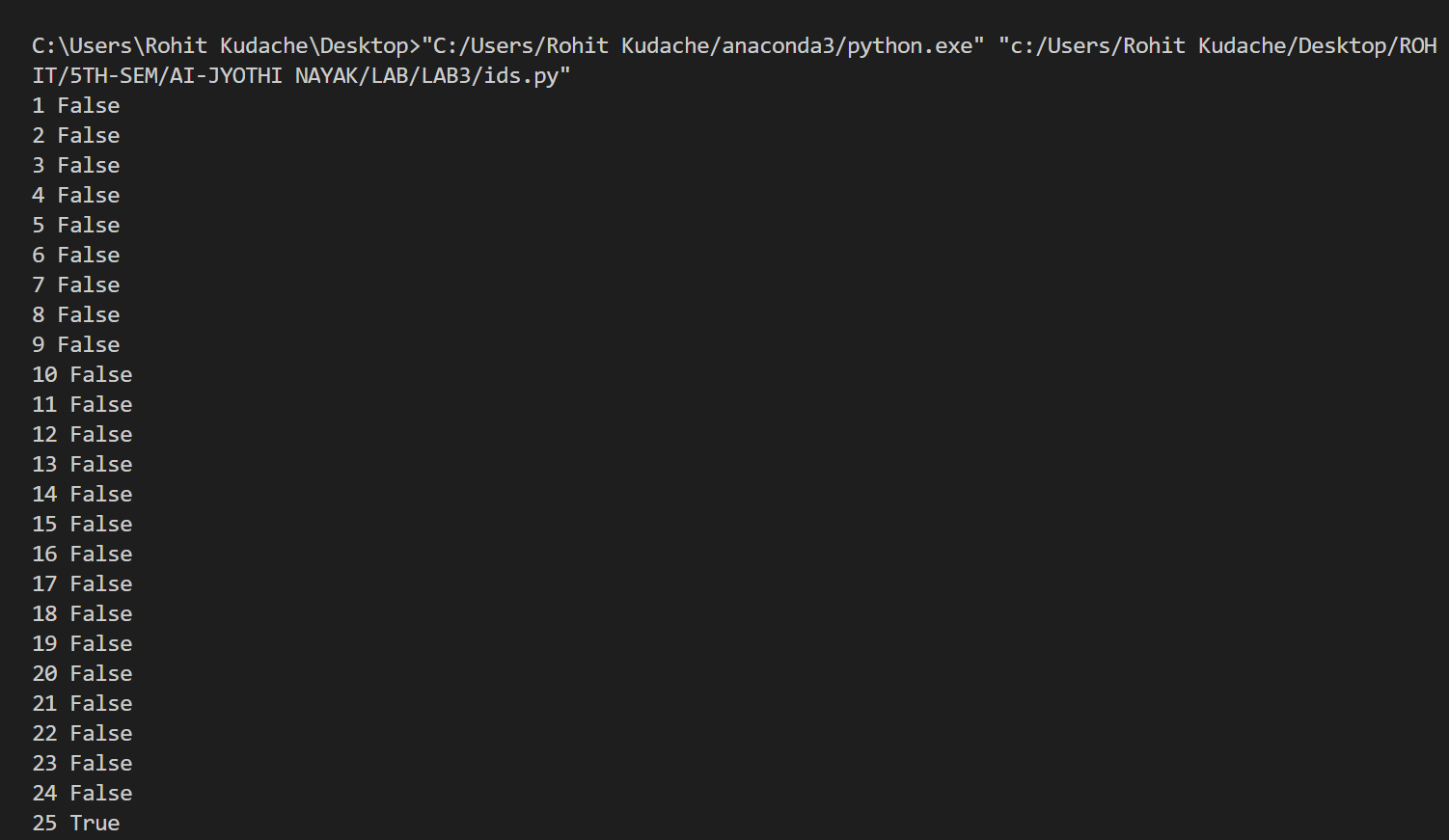
val = iddfs(src,target,i)

print(i, val)

if val == True:

break

**OUTPUT:**

****

**LAB 4:**

# This class represents a node

class Node:

# Initialize the class

def \_\_init\_\_(self, position: (), parent: ()):

self.position = position

self.parent = parent

self.g = 0 # Distance to start node

self.h = 0 # Distance to goal node

self.f = 0 # Total cost

# Compare nodes

def \_\_eq\_\_(self, other):

return self.position == other.position

# Sort nodes

def \_\_lt\_\_(self, other):

return self.f < other.f

# Print node

def \_\_repr\_\_(self):

return ('({0},{1})'.format(self.position, self.f))

# Draw a grid

def draw\_grid(map, width, height, spacing=2, \*\*kwargs):

for y in range(height):

for x in range(width):

print('%%-%ds' % spacing % draw\_tile(map, (x, y), kwargs), end='')

print()

# Draw a tile

def draw\_tile(map, position, kwargs):

# Get the map value

value = map.get(position)

# Check if we should print the path

if 'path' in kwargs and position in kwargs['path']: value = '+'

# Check if we should print start point

if 'start' in kwargs and position == kwargs['start']: value = '@'

# Check if we should print the goal point

if 'goal' in kwargs and position == kwargs['goal']: value = '$'

# Return a tile value

return value

# A\* search

def astar\_search(map, start, end):

# Create lists for open nodes and closed nodes

open = []

closed = []

# Create a start node and an goal node

start\_node = Node(start, None)

goal\_node = Node(end, None)

# Add the start node

open.append(start\_node)

# Loop until the open list is empty

while len(open) > 0:

# Sort the open list to get the node with the lowest cost first

open.sort()

# Get the node with the lowest cost

current\_node = open.pop(0)

# Add the current node to the closed list

closed.append(current\_node)

# Check if we have reached the goal, return the path

if current\_node == goal\_node:

path = []

while current\_node != start\_node:

path.append(current\_node.position)

current\_node = current\_node.parent

# path.append(start)

# Return reversed path

return path[::-1]

# Unzip the current node position

(x, y) = current\_node.position

# Get neighbors

neighbors = [(x - 1, y), (x + 1, y), (x, y - 1), (x, y + 1)]

# Loop neighbors

for next in neighbors:

# Get value from map

map\_value = map.get(next)

# Check if the node is a wall

if (map\_value == '#'):

continue

# Create a neighbor node

neighbor = Node(next, current\_node)

# Check if the neighbor is in the closed list

if (neighbor in closed):

continue

# Generate heuristics (Manhattan distance)

neighbor.g = abs(neighbor.position[0] - start\_node.position[0]) + abs(

neighbor.position[1] - start\_node.position[1])

neighbor.h = abs(neighbor.position[0] - goal\_node.position[0]) + abs(

neighbor.position[1] - goal\_node.position[1])

neighbor.f = neighbor.g + neighbor.h

# Check if neighbor is in open list and if it has a lower f value

if (add\_to\_open(open, neighbor) == True):

# Everything is green, add neighbor to open list

open.append(neighbor)

# Return None, no path is found

return None

# Check if a neighbor should be added to open list

def add\_to\_open(open, neighbor):

for node in open:

if (neighbor == node and neighbor.f >= node.f):

return False

return True

# The main entry point for this module

def main():

# Get a map (grid)

map = {}

chars = ['c']

start = None

end = None

width = 0

height = 0

# Open a file

fp = open('maze-grid.txt', 'r')

# Loop until there is no more lines

while len(chars) > 0:

# Get chars in a line

chars = [str(i) for i in fp.readline().strip()]

# Calculate the width

width = len(chars) if width == 0 else width

# Add chars to map

for x in range(len(chars)):

map[(x, height)] = chars[x]

if (chars[x] == '@'):

start = (x, height)

elif (chars[x] == '$'):

end = (x, height)

# Increase the height of the map

if (len(chars) > 0):

height += 1

# Close the file pointer

fp.close()

# Find the closest path from start(@) to end($)

path = astar\_search(map, start, end)

print()

print(path)

print()

draw\_grid(map, width, height, spacing=1, path=path, start=start, goal=end)

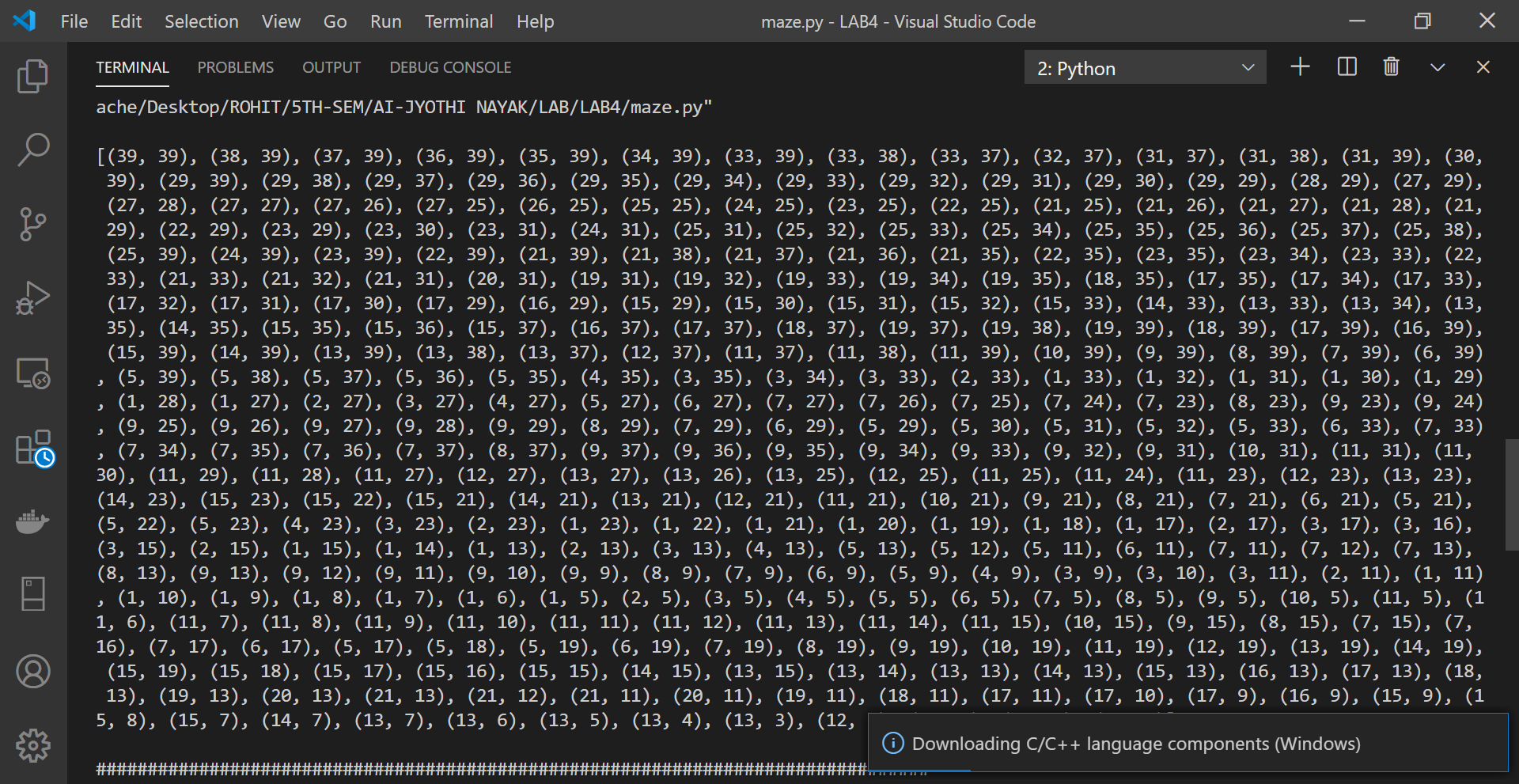
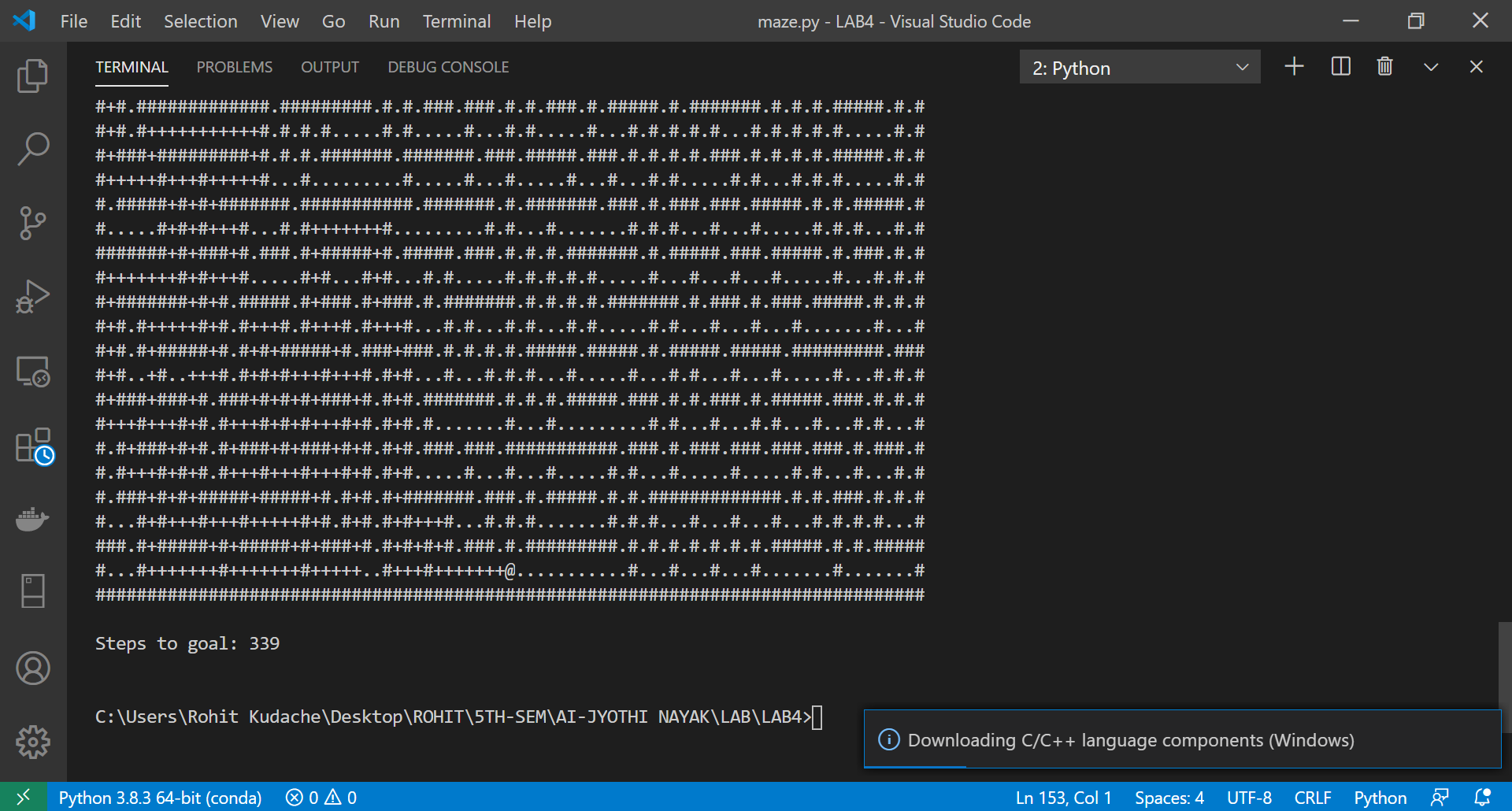
print()

print('Steps to goal: {0}'.format(len(path)))

print()

# Tell python to run main method

if \_\_name\_\_ == "\_\_main\_\_": main()

**OUTPUT:**

**LAB 5:**

#INSTRUCTIONS

#Enter LOCATION A/B in captial letters

#Enter Status O/1 accordingly where 0 means CLEAN and 1 means DIRTY

def vacuum\_world():

# initializing goal\_state

# 0 indicates Clean and 1 indicates Dirty

goal\_state = {'A': '0', 'B': '0'}

cost = 0

location\_input = input("Enter Location of Vacuum \t") #user\_input of location vacuum is placed

status\_input = input("Enter status of"+" " + location\_input + "\t") #user\_input if location is dirty or clean

status\_input\_complement = input("Enter status of other room \t")

initial\_state = {'A' : status\_input , 'B' : status\_input\_complement}

print("Initial Location Condition" + str(initial\_state))

if location\_input == 'A':

# Location A is Dirty.

print("Vacuum is placed in Location A")

if status\_input == '1':

print("Location A is Dirty.")

# suck the dirt and mark it as clean

goal\_state['A'] = '0'

cost += 1 #cost for suck

print("Cost for CLEANING A " + str(cost))

print("Location A has been Cleaned.")

if status\_input\_complement == '1':

# if B is Dirty

print("Location B is Dirty.")

print("Moving right to the Location B. ")

cost += 1 #cost for moving right

print("COST for moving RIGHT" + str(cost))

# suck the dirt and mark it as clean

goal\_state['B'] = '0'

cost += 1 #cost for suck

print("COST for SUCK " + str(cost))

print("Location B has been Cleaned. ")

else:

print("No action" + str(cost))

# suck and mark clean

print("Location B is already clean.")

if status\_input == '0':

print("Location A is already clean ")

if status\_input\_complement == '1':# if B is Dirty

print("Location B is Dirty.")

print("Moving RIGHT to the Location B. ")

cost += 1 #cost for moving right

print("COST for moving RIGHT " + str(cost))

# suck the dirt and mark it as clean

goal\_state['B'] = '0'

cost += 1 #cost for suck

print("Cost for SUCK" + str(cost))

print("Location B has been Cleaned. ")

else:

print("No action " + str(cost))

print(cost)

# suck and mark clean

print("Location B is already clean.")

else:

print("Vacuum is placed in location B")

# Location B is Dirty.

if status\_input == '1':

print("Location B is Dirty.")

# suck the dirt and mark it as clean

goal\_state['B'] = '0'

cost += 1 # cost for suck

print("COST for CLEANING " + str(cost))

print("Location B has been Cleaned.")

if status\_input\_complement == '1':

# if A is Dirty

print("Location A is Dirty.")

print("Moving LEFT to the Location A. ")

cost += 1 # cost for moving right

print("COST for moving LEFT" + str(cost))

# suck the dirt and mark it as clean

goal\_state['A'] = '0'

cost += 1 # cost for suck

print("COST for SUCK " + str(cost))

print("Location A has been Cleaned.")

else:

print(cost)

# suck and mark clean

print("Location B is already clean.")

if status\_input\_complement == '1': # if A is Dirty

print("Location A is Dirty.")

print("Moving LEFT to the Location A. ")

cost += 1 # cost for moving right

print("COST for moving LEFT " + str(cost))

# suck the dirt and mark it as clean

goal\_state['A'] = '0'

cost += 1 # cost for suck

print("Cost for SUCK " + str(cost))

print("Location A has been Cleaned. ")

else:

print("No action " + str(cost))

# suck and mark clean

print("Location A is already clean.")

# done cleaning

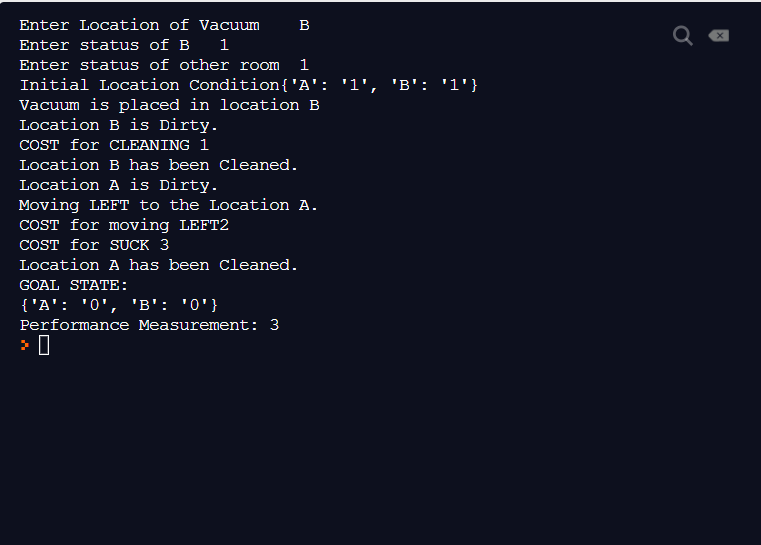
print("GOAL STATE: ")

print(goal\_state)

print("Performance Measurement: " + str(cost))

vacuum\_world()

**OUTPUT:**

****

**LAB 6:**

combinations=[(True,True, True),(True,True,False),(True,False,True),(True,False, False),(False,True, True),(False,True, False),(False, False,True),(False,False, False)]

variable={'p':0,'q':1, 'r':2}

kb=''

q=''

priority={'~':3,'v':1,'^':2}

def input\_rules():

global kb, q

kb = (input("Enter rule: "))

q = input("Enter the Query: ")

def entailment():

global kb, q

print('\*'\*10+"Truth Table Reference"+'\*'\*10)

print('kb','alpha')

print('\*'\*10)

for comb in combinations:

s = evaluatePostfix(toPostfix(kb), comb)

f = evaluatePostfix(toPostfix(q), comb)

print(s, f)

print('-'\*10)

if s and not f:

return False

return True

def isOperand(c):

return c.isalpha() and c!='v'

def isLeftParanthesis(c):

return c == '('

def isRightParanthesis(c):

return c == ')'

def isEmpty(stack):

return len(stack) == 0

def peek(stack):

return stack[-1]

def hasLessOrEqualPriority(c1, c2):

try:

return priority[c1]<=priority[c2]

except KeyError:

return False

def toPostfix(infix):

stack = []

postfix = ''

for c in infix:

if isOperand(c):

postfix += c

else:

if isLeftParanthesis(c):

stack.append(c)

elif isRightParanthesis(c):

operator = stack.pop()

while not isLeftParanthesis(operator):

postfix += operator

operator = stack.pop()

else:

while (not isEmpty(stack)) and hasLessOrEqualPriority(c, peek(stack)):

postfix += stack.pop()

stack.append(c)

while (not isEmpty(stack)):

postfix += stack.pop()

return postfix

def evaluatePostfix(exp, comb):

stack = []

for i in exp:

if isOperand(i):

stack.append(comb[variable[i]])

elif i == '~':

val1 = stack.pop()

stack.append(not val1)

else:

val1 = stack.pop()

val2 = stack.pop()

stack.append(\_eval(i,val2,val1))

return stack.pop()

def \_eval(i, val1, val2):

if i == '^':

return val2 and val1

return val2 or val1

#Test 1

input\_rules()

ans = entailment()

if ans:

print("The Knowledge Base entails query")

else:

print("The Knowledge Base does not entail query")

#Test 2

input\_rules()

ans = entailment()

if ans:

print("The Knowledge Base entails query")

else:

print("The Knowledge Base does not entail query")

**OUTPUT :**

****

**LAB 7:**

import re

def negate(term):

return f'~{term}' if term[0] != '~' else term[1]

def reverse(clause):

if len(clause) > 2:

t = split\_terms(clause)

return f'{t[1]}v{t[0]}'

return ''

def split\_terms(rule):

exp = '(~\*[PQRS])'

terms = re.findall(exp, rule)

return terms

def contradiction(query, clause):

contradictions = [ f'{query}v{negate(query)}', f'{negate(query)}v{query}']

return clause in contradictions or reverse(clause) in contradictions

def resolve(kb, query):

temp = kb.copy()

temp += [negate(query)]

steps = dict()

for rule in temp:

steps[rule] = 'Given.'

steps[negate(query)] = 'Negated conclusion.'

i = 0

while i < len(temp):

n = len(temp)

j = (i + 1) % n

clauses = []

while j != i:

terms1 = split\_terms(temp[i])

terms2 = split\_terms(temp[j])

for c in terms1:

if negate(c) in terms2:

t1 = [t for t in terms1 if t != c]

t2 = [t for t in terms2 if t != negate(c)]

gen = t1 + t2

if len(gen) == 2:

if gen[0] != negate(gen[1]):

clauses += [f'{gen[0]}v{gen[1]}']

else:

if contradiction(query,f'{gen[0]}v{gen[1]}'):

temp.append(f'{gen[0]}v{gen[1]}')

steps[''] = f"Resolved {temp[i]} and {temp[j]} to {temp[-1]}, which is in turn null. \

\nA contradiction is found when {negate(query)} is assumed as true. Hence, {query} is true."

return steps

elif len(gen) == 1:

clauses += [f'{gen[0]}']

else:

if contradiction(query,f'{terms1[0]}v{terms2[0]}'):

temp.append(f'{terms1[0]}v{terms2[0]}')

steps[''] = f"Resolved {temp[i]} and {temp[j]} to {temp[-1]}, which is in turn null. \

\nA contradiction is found when {negate(query)} is assumed as true. Hence, {query} is true."

return steps

for clause in clauses:

if clause not in temp and clause != reverse(clause) and reverse(clause) not in temp:

temp.append(clause)

steps[clause] = f'Resolved from {temp[i]} and {temp[j]}.'

j = (j + 1) % n

i += 1

return steps

def resolution(kb, query):

kb = kb.split(' ')

steps = resolve(kb, query)

print('\nStep\t|Clause\t|Derivation\t')

print('-' \* 30)

i = 1

for step in steps:

print(f' {i}.\t| {step}\t| {steps[step]}\t')

i += 1

def main():

print("Enter the kb:")

kb = input()

print("Enter the query:")

query = input()

resolution(kb,query)

#test 1

#(P^Q)<=>R : (Rv~P)v(Rv~Q)^(~RvP)^(~RvQ)

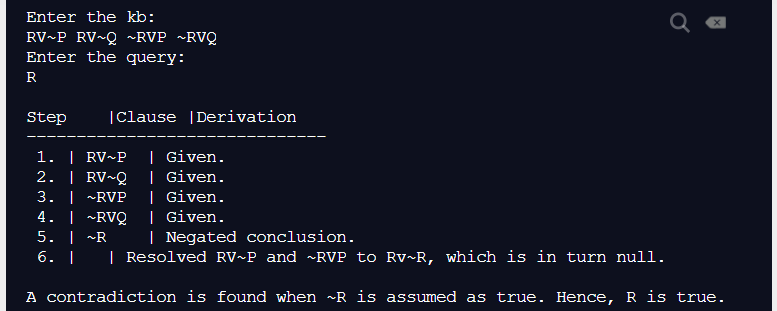
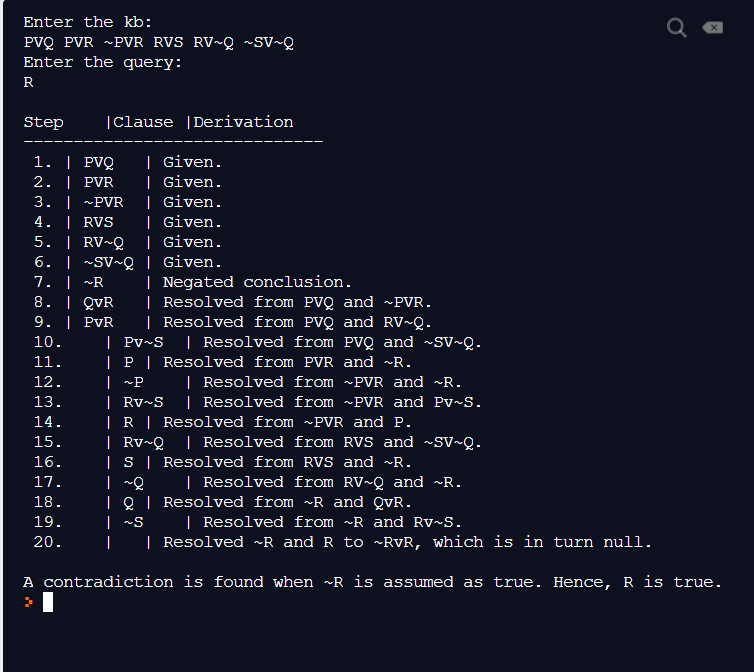
main()

#test 2

#(P=>Q)=>Q, (P=>P)=>R, (R=>S)=>~(S=>Q)

main()

**OUTPUT:**

****

**LAB 8:**

import re

def getAttributes(expression):

expression = expression.split("(")[1:]

expression = "(".join(expression)

expression = expression.split(")")[:-1]

expression = ")".join(expression)

attributes = expression.split(',')

return attributes

def getInitialPredicate(expression):

return expression.split("(")[0]

def isConstant(char):

return char.isupper() and len(char) == 1

def isVariable(char):

return char.islower() and len(char) == 1

def replaceAttributes(exp, old, new):

attributes = getAttributes(exp)

predicate = getInitialPredicate(exp)

for index, val in enumerate(attributes):

if val == old:

attributes[index] = new

return predicate + "(" + ",".join(attributes) + ")"

def apply(exp, substitutions):

for substitution in substitutions:

new, old = substitution

exp = replaceAttributes(exp, old, new)

return exp

def checkOccurs(var, exp):

if exp.find(var) == -1:

return False

return True

def getFirstPart(expression):

attributes = getAttributes(expression)

return attributes[0]

def getRemainingPart(expression):

predicate = getInitialPredicate(expression)

attributes = getAttributes(expression)

newExpression = predicate + "(" + ",".join(attributes[1:]) + ")"

return newExpression

def unify(exp1, exp2):

if exp1 == exp2:

return []

if isConstant(exp1) and isConstant(exp2):

if exp1 != exp2:

print(f"{exp1} and {exp2} are constants. Cannot be unified")

return []

if isConstant(exp1):

return [(exp1, exp2)]

if isConstant(exp2):

return [(exp2, exp1)]

if isVariable(exp1):

return [(exp2, exp1)] if not checkOccurs(exp1, exp2) else []

if isVariable(exp2):

return [(exp1, exp2)] if not checkOccurs(exp2, exp1) else []

if getInitialPredicate(exp1) != getInitialPredicate(exp2):

print("Cannot be unified as the predicates do not match!")

return []

attributeCount1 = len(getAttributes(exp1))

attributeCount2 = len(getAttributes(exp2))

if attributeCount1 != attributeCount2:

print(f"Length of attributes {attributeCount1} and {attributeCount2} do not match. Cannot be unified")

return []

head1 = getFirstPart(exp1)

head2 = getFirstPart(exp2)

initialSubstitution = unify(head1, head2)

if not initialSubstitution:

return []

if attributeCount1 == 1:

return initialSubstitution

tail1 = getRemainingPart(exp1)

tail2 = getRemainingPart(exp2)

if initialSubstitution != []:

tail1 = apply(tail1, initialSubstitution)

tail2 = apply(tail2, initialSubstitution)

remainingSubstitution = unify(tail1, tail2)

if not remainingSubstitution:

return []

return initialSubstitution + remainingSubstitution

def main():

print("Enter the first expression")

e1 = input()

print("Enter the second expression")

e2 = input()

substitutions = unify(e1, e2)

print("The substitutions are:")

print([' / '.join(substitution) for substitution in substitutions])

main()

print(" ")

print("------------------- ")

print(" ")

main()

print(" ")

print("------------------- ")

print(" ")

main()

print(" ")

print("------------------- ")

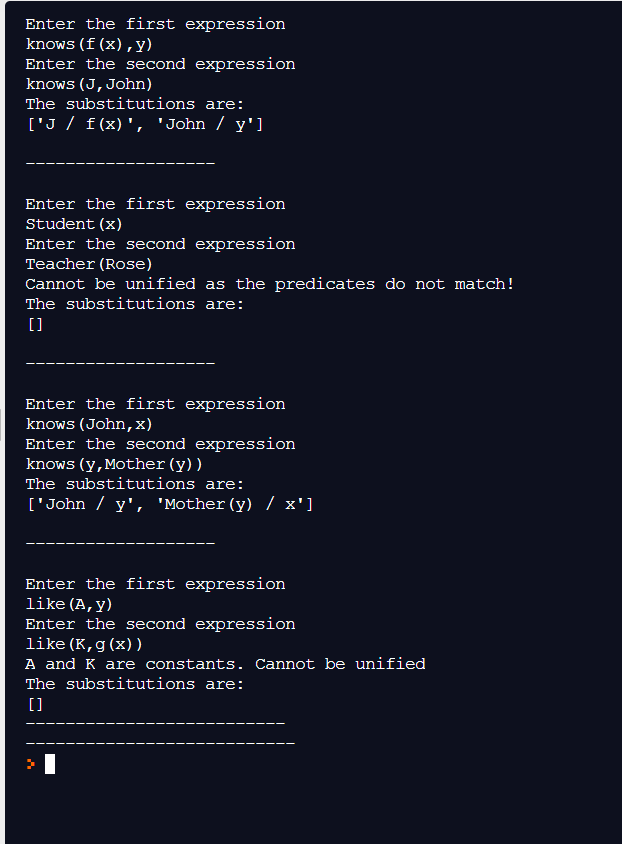
print(" ")

main()

print("-------------------------- ")

print("---------------------------")

**OUTPUT:**

****

**LAB 9:**

import re

print("Enter FOL")

def remove\_brackets(source, id):

reg = '\(([^\(]\*?)\)'

m = re.search(reg, source)

if m is None:

return None, None

new\_source = re.sub(reg, str(id), source, count=1)

return new\_source, m.group(1)

class logic\_base:

def \_\_init\_\_(self, input):

self.my\_stack = []

self.source = input

final = input

while 1:

input, tmp = remove\_brackets(input, len(self.my\_stack))

if input is None:

break

final = input

self.my\_stack.append(tmp)

self.my\_stack.append(final)

def get\_result(self):

root = self.my\_stack[-1]

m = re.match('\s\*([0-9]+)\s\*$', root)

if m is not None:

root = self.my\_stack[int(m.group(1))]

reg = '(\d+)'

while 1:

m = re.search(reg, root)

if m is None:

break

new = '(' + self.my\_stack[int(m.group(1))] + ')'

root = re.sub(reg, new, root, count=1)

return root

def merge\_items(self, logic):

reg0 = '(\d+)'

reg1 = 'neg\s+(\d+)'

flag = False

for i in range(len(self.my\_stack)):

target = self.my\_stack[i]

if logic not in target:

continue

m = re.search(reg1, target)

if m is not None:

continue

m = re.search(reg0, target)

if m is None:

continue

for j in re.findall(reg0, target):

child = self.my\_stack[int(j)]

if logic not in child:

continue

new\_reg = "(^|\s)" + j + "(\s|$)"

self.my\_stack[i] = re.sub(new\_reg, ' ' + child + ' ', self.my\_stack[i], count=1)

self.my\_stack[i] = self.my\_stack[i].strip()

flag = True

if flag:

self.merge\_items(logic)

class ordering(logic\_base):

def run(self):

flag = False

for i in range(len(self.my\_stack)):

new\_source = self.add\_brackets(self.my\_stack[i])

if self.my\_stack[i] != new\_source:

self.my\_stack[i] = new\_source

flag = True

return flag

def add\_brackets(self, source):

reg = "\s+(and|or|imp|iff)\s+"

if len(re.findall(reg, source)) < 2:

return source

reg\_and = "(neg\s+)?\S+\s+and\s+(neg\s+)?\S+"

m = re.search(reg\_and, source)

if m is not None:

return re.sub(reg\_and, "(" + m.group(0) + ")", source, count=1)

reg\_or = "(neg\s+)?\S+\s+or\s+(neg\s+)?\S+"

m = re.search(reg\_or, source)

if m is not None:

return re.sub(reg\_or, "(" + m.group(0) + ")", source, count=1)

reg\_imp = "(neg\s+)?\S+\s+imp\s+(neg\s+)?\S+"

m = re.search(reg\_imp, source)

if m is not None:

return re.sub(reg\_imp, "(" + m.group(0) + ")", source, count=1)

reg\_iff = "(neg\s+)?\S+\s+iff\s+(neg\s+)?\S+"

m = re.search(reg\_iff, source)

if m is not None:

return re.sub(reg\_iff, "(" + m.group(0) + ")", source, count=1)

class replace\_iff(logic\_base):

def run(self):

final = len(self.my\_stack) - 1

flag = self.replace\_all\_iff()

self.my\_stack.append(self.my\_stack[final])

return flag

def replace\_all\_iff(self):

flag = False

for i in range(len(self.my\_stack)):

ans = self.replace\_iff\_inner(self.my\_stack[i], len(self.my\_stack))

if ans is None:

continue

self.my\_stack[i] = ans[0]

self.my\_stack.append(ans[1])

self.my\_stack.append(ans[2])

flag = True

return flag

def replace\_iff\_inner(self, source, id):

reg = '^(.\*?)\s+iff\s+(.\*?)$'

m = re.search(reg, source)

if m is None:

return None

a, b = m.group(1), m.group(2)

return (str(id) + ' and ' + str(id + 1), a + ' imp ' + b, b + ' imp ' + a)

class replace\_imp(logic\_base):

def run(self):

flag = False

for i in range(len(self.my\_stack)):

ans = self.replace\_imp\_inner(self.my\_stack[i])

if ans is None:

continue

self.my\_stack[i] = ans

flag = True

return flag

def replace\_imp\_inner(self, source):

reg = '^(.\*?)\s+imp\s+(.\*?)$'

m = re.search(reg, source)

if m is None:

return None

a, b = m.group(1), m.group(2)

if 'neg ' in a:

return a.replace('neg ', '') + ' or ' + b

return 'neg ' + a + ' or ' + b

class de\_morgan(logic\_base):

def run(self):

reg = 'neg\s+(\d+)'

flag = False

final = len(self.my\_stack) - 1

for i in range(len(self.my\_stack)):

target = self.my\_stack[i]

m = re.search(reg, target)

if m is None:

continue

flag = True

child = self.my\_stack[int(m.group(1))]

self.my\_stack[i] = re.sub(reg, str(len(self.my\_stack)), target, count=1)

self.my\_stack.append(self.doing\_de\_morgan(child))

break

self.my\_stack.append(self.my\_stack[final])

return flag

def doing\_de\_morgan(self, source):

items = re.split('\s+', source)

new\_items = []

for item in items:

if item == 'or':

new\_items.append('and')

elif item == 'and':

new\_items.append('or')

elif item == 'neg':

new\_items.append('neg')

elif len(item.strip()) > 0:

new\_items.append('neg')

new\_items.append(item)

for i in range(len(new\_items) - 1):

if new\_items[i] == 'neg':

if new\_items[i + 1] == 'neg':

new\_items[i] = ''

new\_items[i + 1] = ''

return ' '.join([i for i in new\_items if len(i) > 0])

class distributive(logic\_base):

def run(self):

flag = False

reg = '(\d+)'

final = len(self.my\_stack) - 1

for i in range(len(self.my\_stack)):

target = self.my\_stack[i]

if 'or' not in self.my\_stack[i]:

continue

m = re.search(reg, target)

if m is None:

continue

for j in re.findall(reg, target):

child = self.my\_stack[int(j)]

if 'and' not in child:

continue

new\_reg = "(^|\s)" + j + "(\s|$)"

items = re.split('\s+and\s+', child)

tmp\_list = [str(j) for j in range(len(self.my\_stack), len(self.my\_stack) + len(items))]

for item in items:

self.my\_stack.append(re.sub(new\_reg, ' ' + item + ' ', target).strip())

self.my\_stack[i] = ' and '.join(tmp\_list)

flag = True

if flag:

break

self.my\_stack.append(self.my\_stack[final])

return flag

class simplification(logic\_base):

def run(self):

old = self.get\_result()

for i in range(len(self.my\_stack)):

self.my\_stack[i] = self.reducing\_or(self.my\_stack[i])

# self.my\_stack[i] = self.reducing\_and(self.my\_stack[i])

final = self.my\_stack[-1]

self.my\_stack[-1] = self.reducing\_and(final)

return len(old) != len(self.get\_result())

def reducing\_and(self, target):

if 'and' not in target:

return target

items = set(re.split('\s+and\s+', target))

for item in list(items):

if ('neg ' + item) in items:

return ''

if re.match('\d+$', item) is None:

continue

value = self.my\_stack[int(item)]

if self.my\_stack.count(value) > 1:

value = ''

self.my\_stack[int(item)] = ''

if value == '':

items.remove(item)

return ' and '.join(list(items))

def reducing\_or(self, target):

if 'or' not in target:

return target

items = set(re.split('\s+or\s+', target))

for item in list(items):

if ('neg ' + item) in items:

return ''

return ' or '.join(list(items))

def merging(source):

old = source.get\_result()

source.merge\_items('or')

source.merge\_items('and')

return old != source.get\_result()

def run(input):

all\_strings = []

# all\_strings.append(input)

zero = ordering(input)

while zero.run():

zero = ordering(zero.get\_result())

merging(zero)

one = replace\_iff(zero.get\_result())

one.run()

all\_strings.append(one.get\_result())

merging(one)

two = replace\_imp(one.get\_result())

two.run()

all\_strings.append(two.get\_result())

merging(two)

three, four = None, None

old = two.get\_result()

three = de\_morgan(old)

while three.run():

pass

all\_strings.append(three.get\_result())

merging(three)

three\_helf = simplification(three.get\_result())

three\_helf.run()

four = distributive(three\_helf.get\_result())

while four.run():

pass

merging(four)

five = simplification(four.get\_result())

five.run()

all\_strings.append(five.get\_result())

return all\_strings

inputs = input().split('\n')

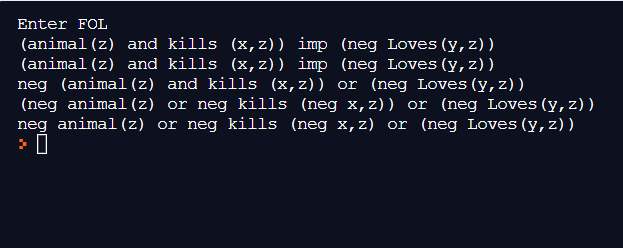
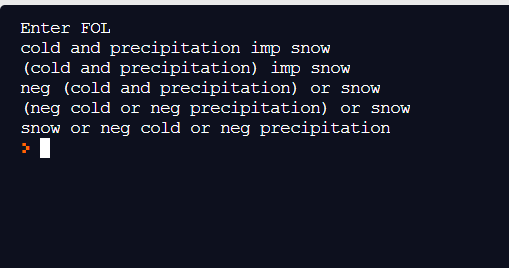
for input in inputs:

for item in run(input):

print(item)

# output.write('\n')

**OUTPUT:**

****

**LAB 10:**

import re

def isVariable(x):

return len(x) == 1 and x.islower() and x.isalpha()

def getAttributes(string):

expr = '\([^)]+\)'

matches = re.findall(expr, string)

return matches

def getPredicates(string):

expr = '([a-z~]+)\([^&|]+\)'

return re.findall(expr, string)

class Fact:

def \_init\_(self, expression):

self.expression = expression

predicate, params = self.splitExpression(expression)

self.predicate = predicate

self.params = params

self.result = any(self.getConstants())

def splitExpression(self, expression):

predicate = getPredicates(expression)[0]

params = getAttributes(expression)[0].strip('()').split(',')

return [predicate, params]

def getResult(self):

return self.result

def getConstants(self):

return [None if isVariable(c) else c for c in self.params]

def getVariables(self):

return [v if isVariable(v) else None for v in self.params]

def substitute(self, constants):

c = constants.copy()

f = f"{self.predicate}({','.join([constants.pop(0) if isVariable(p) else p for p in self.params])})"

return Fact(f)

class Implication:

def \_init\_(self, expression):

self.expression = expression

l = expression.split('=>')

self.lhs = [Fact(f) for f in l[0].split('&')]

self.rhs = Fact(l[1])

def evaluate(self, facts):

constants = {}

new\_lhs = []

for fact in facts:

for val in self.lhs:

if val.predicate == fact.predicate:

for i, v in enumerate(val.getVariables()):

if v:

constants[v] = fact.getConstants()[i]

new\_lhs.append(fact)

predicate, attributes = getPredicates(self.rhs.expression)[0], str(getAttributes(self.rhs.expression)[0])

for key in constants:

if constants[key]:

attributes = attributes.replace(key, constants[key])

expr = f'{predicate}{attributes}'

return Fact(expr) if len(new\_lhs) and all([f.getResult() for f in new\_lhs]) else None

class KB:

def \_init\_(self):

self.facts = set()

self.implications = set()

def tell(self, e):

if '=>' in e:

self.implications.add(Implication(e))

else:

self.facts.add(Fact(e))

for i in self.implications:

res = i.evaluate(self.facts)

if res:

self.facts.add(res)

def query(self, e):

facts = set([f.expression for f in self.facts])

i = 1

print(f'Querying {e}:')

for f in facts:

if Fact(f).predicate == Fact(e).predicate:

print(f'\t{i}. {f}')

i += 1

def display(self):

print("All facts: ")

for i, f in enumerate(set([f.expression for f in self.facts])):

print(f'\t{i+1}. {f}')

def main():

kb = KB()

print("Enter KB: (enter e to exit)")

while True:

t = input()

if(t == 'e'):

break

kb.tell(t)

print("Enter Query:")

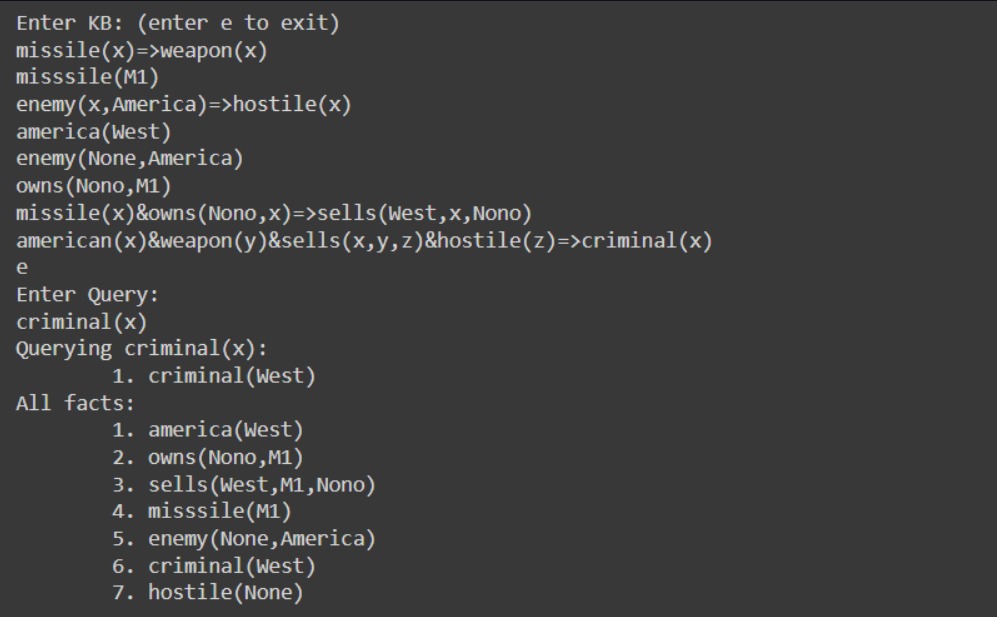
q = input()

kb.query(q)

kb.display()

main()

**OUTPUT:**

****