1) age = int (input("enter your age"))
if age ≤ 10:
    print("Hello")
elif age > 11 and age < 30:
    print ("Padma from badvel")
else:
    print ("sushitha from kdp")

output:
enter your age 14
~~Hello~~ padma from badvel

2) Printing list of integers in reverse order:

Programs

input_list = input("Enter a list of integers separated by spaces:")
integer_list = [int(x) for x in input_list.split()]
print ("original list :", integer_list)
print("Reversed list :", integer_list[::-1])

output:

Enter a list of integers separated by spaces : 1 23 45 18

original list : [1, 23, 45, 18]

Reversed list : [18, 45, 23, 1]

3) Printing the Pattern

i) 1
2 2
3 3 3
4 4 4 4
5 5 5 5 5

until the user input

Program:

```
num_rows = int(input("Enter the number of rows :"))
for i in range (1, num_rows +1):
    for j in range(i):
        print(i, end =" ")
    print()
```

output:

Enter the number of rows : 7

1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
6 6 6 6 6 6
7 7 7 7 7 7 7

ii) 1
1 2
1 2 3
1 2 3 4
1 2 3 4 5 .. untill the user inpat

program:

```
num_rows = int(input("Enter the number
                      of rows: "))
for i in range(1, num_rows+1):
    for j in range(1, i+1):
        print(j, end = " ")
    print()
```

output: Enter the number of rows : 7

1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
1 2 3 4 5 6 7

4) print power of a number

x and n are inputs.

$x^n$.

program:

```
base = int(input("Enter the base (x):"))
exponent = int(input("Enter the exponent(n):"))
result = base ** exponent  : Pow(x,y)
print(f"{base} raised to power of
        {exponent} is : {result}").
```

output:

```
Enter the base (x) : 2
Enter the exponent (n) : 2
2 raised to the power of 2 is : 4
```

10/11/23

Program 1 : Implementing Tic-Tac-Toe Game

Algorithm:

1. Initialize the Board :
   create a 3x3 Grid to represent the Tic Tac Toe board.
2. Print the Board :
   - create a function to display the current state of the board.
3. Player Input :
   - Take Input from two players, alternating between 'x' and 'o'.
   - Ensure the input is within the valid range (0 to 2 for both row and column).
   - check if the selected cell is already occupied ; if yes, ask for input again.
4. Update the Board :
   update the board with player's symbol at the chosen position.
5. check for a winner :
   - After each move, check if the current player has won.

- check rows, columns and diagnols for three matching symbols.

6. check for a Tie:
   - If the board is full and no player has won; declare a tie.

7. Repeat:
   Repeat steps 2-6 untill a player wins or the game ends in a tie.

8. End of the game:

Program:

```
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(' '+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 (

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
                a range!')

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=[]
```

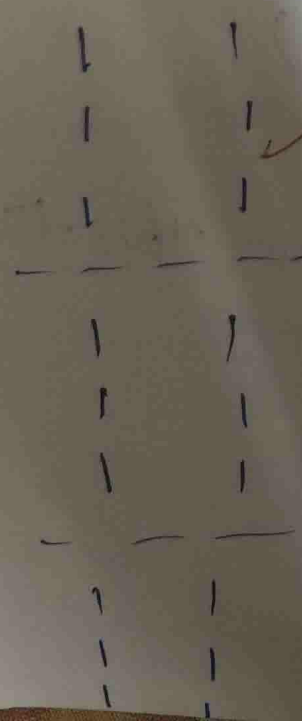
```
    if(len(edgesopen)>0:
        move=selectRandom(edgesopen)
    return move
def selectRandom(li):
    import random
    ln=len(li)
    r=random.randrange(0,ln)

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
```

```
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
```

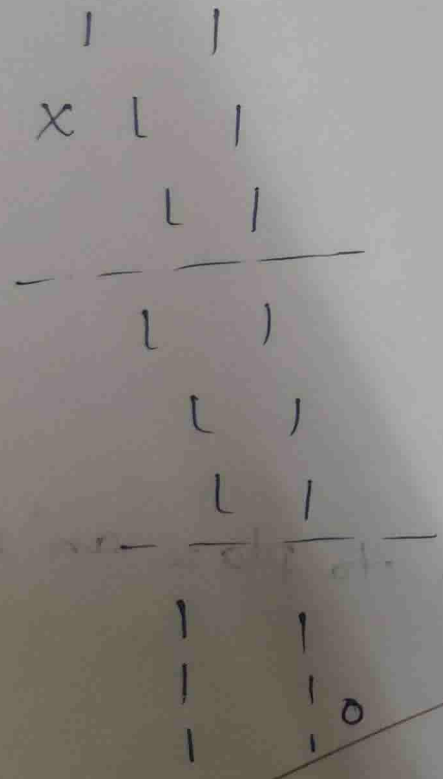
output:

```
Do you want to play again? (Y/N) y
- - - - - - - - -
Welcome to Tic Tac Toe!
```

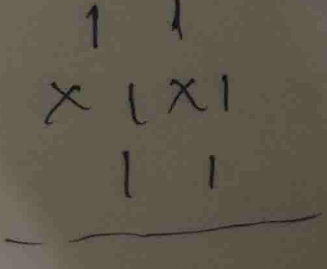


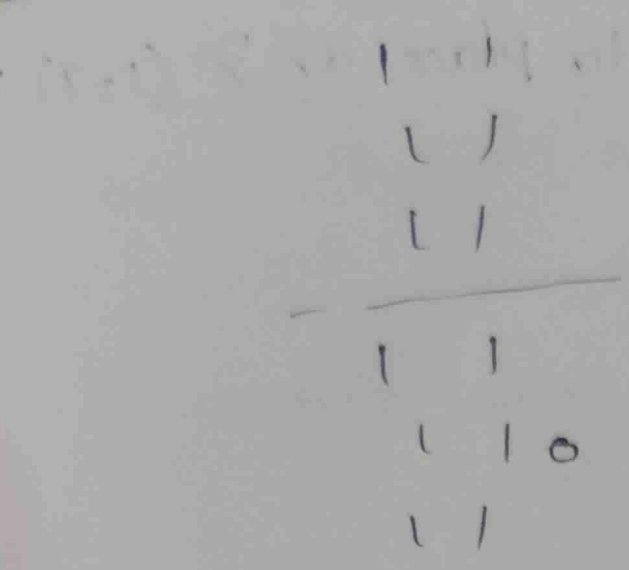
Please Select a Position to place an 'x' (1-9) :1

computer placed an 'o' in Position 9:

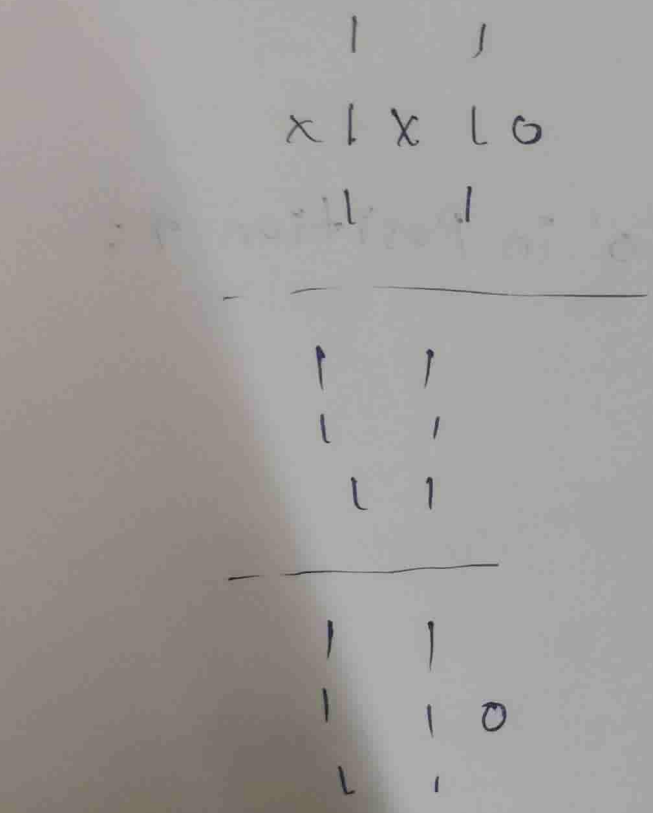


Please Select a Position to place an 'x'(1-9):2

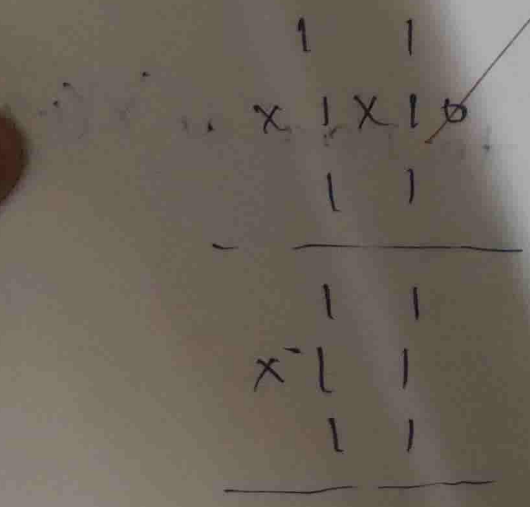




computer placed an 'o' in position 3:



Please select a position to place an 'x' (1-9):



| |
| | O
| |

computer placed an 'o' in position 6:.

| |

sorry, O's won this time!

Do you want to play again? (Y/N)

N

Lab 2: 8 Puzzle program

Algorithm:

1. Initialize Puzzle
   - create an Instance of the 'Puzzle8' class
   - The Initial board is generated randomly
2. Loop until Puzzle is Solved
   - while the puzzle is not solved (determined by the 'is_solved' method):
   - current state of board : print_board
   - Prompt the user to enter a move direction (up, down, left, right).
3. Move Method:
   - Get the Index of the blank tile(0) on the board.
   - If the specified direction is valid swap the blank tile with the adjacent tile otherwise print error Message.
4. check for Valid Move :
   - Ensure the Move direction is valid based on current Position of blank tile

5. check for Puzzle solved:
   - is_solved method compares the current board state.
   - If they match, Puzzle is solved.

6. User Interaction:
   - The loop continues until the Puzzle is solved.

7. End of Puzzle:
   - once the loop exists (Puzzle is solved)

Program:

```
import heapq
class PuzzleNode:
    def __init__(self, state, parent=None,
                 action=None, cost=0):
        self.state = state
        self.parent = parent
        self.action = action
        self.cost = cost
        self.heuristic = self.calculate_heuristic()
        self.
    def calculate_heuristic(self):
        goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
        h = 0
```

```
for i in range(3):
    for j in range(3):
        if self.state[i][j] != 0:
            goal_row, goal_col = divmod(self.state
                                        [i][j]-1, 3)
            h += abs(i-goal_row) + abs(j-goal_col)
return h
def __lt__(self, other):
    return (self.cost + self.heuristic) <
           (other.cost + other.heuristic)

def get_neighbors(node):
    i, j = get_blank_position(node.state)
    neighbors = []
    if i > 0:
        neighbors.append(('down', (i+1, j)))
    if i < 2:
        neighbors.append(('up', (i+1, j)))
    if j > 0:
        neighbors.append(('right', (i, j-1)))
    if j < 2:
        neighbors.append(('left', (i, j+1)))
```

```
return neighbors

def Print_solution(solution):
    current = solution
    Path = []
    while current is not None:
        Path.append((current.action, current.State))
    print('\n')

def astar(initial_state):
    initial_node = Puzzlenode(initial_state)

    while frontier:
        current_node = heapq.heappop(frontier)
        if current_node = heapq.heappop(frontier)
        if current_node.state == [(1, 2, 3), (4, 5, 6),
                                  (7, 8, 0)]:
        print("Goal State reached!")
```

output:

Goal State reached!

Action: None

[1, 2, 3]

[4, 0, 6]

[7, 5, 8]

Action : up

[1, 2, 3]

[4, 5, 6]

[7, 0, 8]

Action : left

[1, 2, 3]

[4, 5, 6]

[7, 8, 0]

24. 11. 23

Lab 3: 8 Puzzle program using Iterative deepening Search Algorithm.

Algorithm:

1) Define Puzzle Node class:
   - Each node represents a state of the Puzzle (3x3) Grid.
   - Store the parent node, the action taken to reach the current state, the depth.

2) Define helper functions:
   - get_blank_position(state): Find the position of the blank (0) in the puzzle.
   - get_neighbors(node): Get valid neighbors for a given state.
   - apply_action(state, action): Apply a given action to a state and return new state.

3) Depth_limited Search function:
   - depth_limited_search(node, goal_state, depth_limit): perform a dls starting from given node.
   - If the goal state is found, return node, If the dls is reached, return None.

4. Iterative Deepening Search Function:

- iterative_deepening_search (initial_state, goal_state): perform iterative deepening search.
- Initialize depth_limit to 0.
- If solution found for initial state, print result.

5. Main function:

- Define Initial State of Puzzle & goal state.
- call 'iterative_deepening_Search with the initial & goal states.

Program:

```
import copy
Goal_state = [[1, 2, 3], [ 8, 0, 4], [7, 6, 5]]
Move-up, Move down, Move left,
                Move right = U, D, L, R
Moves = [Move up, Move Down,
                Move-left, Move-right]
def print_puzzle (state):
    for row in state:
        print (row)
    print()
```

```
def find-Puzzle(state):
    for row in state:
        print(row)
    print()

def find-blank(state):
    for i in range(3):
        for j in range(3):
            if state(i)(j) == 0;
                return i,j

def Move(state, direction):
    i, j = find-blank(state)
    new-state = copy.deepcopy(state)

    if direction == Move up and i>0:
        new-state(i)(j), new state(i-1)(j):
                new-state(i-1)(j),
                    new-state(i)(j)

    elif direction == Move-Down and i<2:
        new-state(i)(j), new_state(i+1)(j) =
                new state(i+1)(j), newstate(i)(j

    return new-state

def is-goal(state):
        return state == Goal-State
```

```
def depth_limited_search (state, depth, path):
    if depth == 0:
        return None
    if is_goal(state):
        return path
    return None
def iterative_deepening_search( initial_state):
    depth = 0
    while True:
        if result is not None:
            return result
        depth += 1
initial_state = [(1, 2, 3), (0, 8, 4), (7, 6, 5)]
print ("Initial State:")
if solution_path is not None:
    print("Solution found!")
if solution_path is not None:
```

```
print_Puzzle(state)

else:
print("No Solution found")
```

---

output:

```
Initial state
[1,2,3]
[0,8,4]
[7,6,5]
Solution found

step 1:
[1,2,3]
[0,8,4]
[7,6,5]

step 2:
[1,2,3]
[8,0,4]
[7,6,5]
```

8/12/23

8 Puzzle using A* Search Algorithm:

Algorithm:

1) Define PuzzleNode class:
   - Each node represents a state of the Puzzle (3x3 Grid)
   - Store the parent node, the action taken to reach the current state, the cost and heuristic value.

2) Define PuzzleNode Methods:
   - calculate_heuristic() : calculate heuristic value for current state.
   - __lt__(self, other) : Define comparison method for priority queue in A*.

3) Define Helper functions:
   - get_blank_position(state):
   - get_neighbors(node)
   - apply_action(state, action)

4) A* Search Algorithm:
   - Initialize the initial state as PuzzleNode.
   - Initialize a priority queue to store Nodes.

- Pop the node with the lowest cost + heuristic from the priority queue.
- If the current state is goal state, print the solution path & exit.

5) print solution
- print the actions & states along path.

6) Main function:
- Define the initial state of the puzzle
- call the A* algorithm with initial state.

Program:

```
class Node:
    def __init__(self, data, level, fval):
        self.data = data
        self.level = level
        self.fval = fval

    def generate_child(self):
        x, y = self.find(self.data, '-')
        val_list = [[x, y-1], [x, y+1], [x-1, y],
                    [x+1, y]]
        children = []
        for i in val_list:
            child = self.shuffle
```

```
def shuffle(self, puz, x1, y1, x2, y2):
    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):
    temp = []
    for i in root:
        t = []
        for j in root i:
            t.append(j)
        temp.append(t)
    return temp
class Puzzle:
    def __init__(self, size):
        self.n = size
```

```
        self.open = []
        self.clase = []
    def accept(self):
        puz = []
        for i in range(0, self.n):
            temp = input().split(" ")
            puz.append(temp)
        return puz

    def process(self):
        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)
        self.open.append(start)
        print("\n\n")
        while True:
            cur = self.open(0)
            print(" ")

puz = puzzle(3)
puz = process()
```

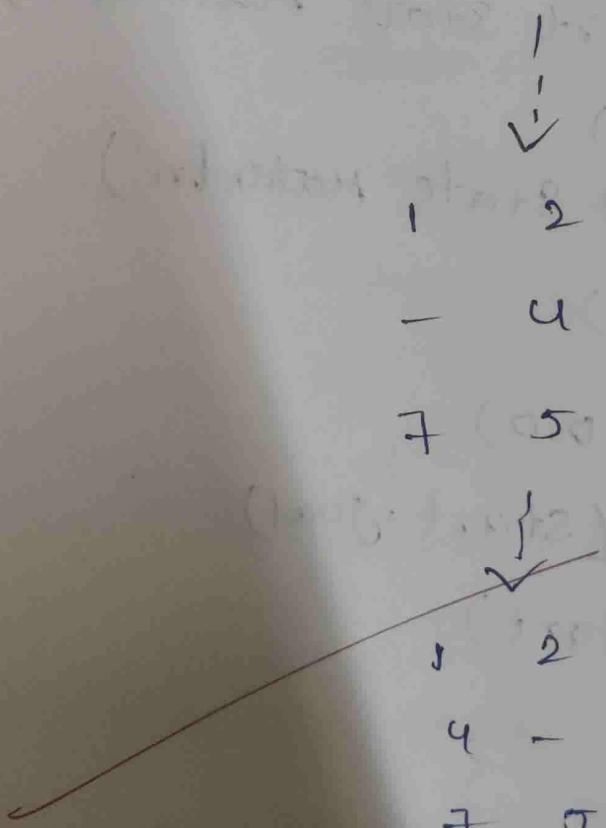
output:

Enter the Start State Matrix

1 2 3
4 6
7 5 8

Enter the goal State Matrix

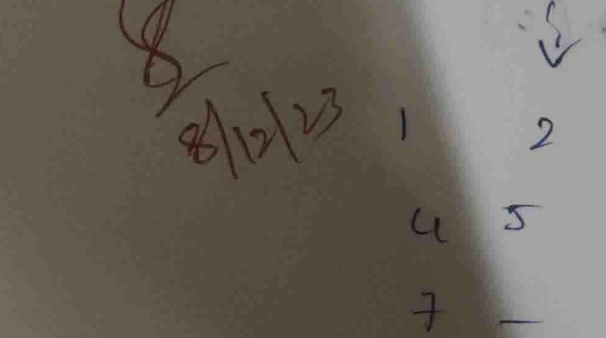
1 2 3
4 5 6
7 8 0



1 2 3
- 4 5
7 5 8



1 2 3
4 - 6
7 5 8



1 2 3
4 5 6
7 - 8

→ 1 2 3
4 5 6
7 8 -

8/12/23

Lab 4: Vacuum cleaner Agent

Algorithm:

1) Initialize:
Place the vacuum cleaner in a starting position.

2) Sensors:
Use sensors to detect the current state of the environment (whether a location is dirty or clean)

3) Action:
- If the current location is dirty, clean it.
- If the current location is clean, move to an adjacent location.
- Repeat steps 2 and 3 untill all locations are clean.

4) Environment interaction:
- Interact with the environment to determine the status of each location.
- Mark cleaned locations.

5) Movement:
- Move to the next location (if any) based on a predefined path or a strategy.

6. Repeat:

- Repeat steps 2-5 untill all locations are clean.

Program:

```
def clean(floor):
    i,j,row,col=0,0,len(floor),len(floor[0])
    for i in range(row):
        if(i%2==0):
            for j in range(col):
                if(floor[i][j]==1):
                    print_F(floor,i,j)
                    floor[i][j]=0
                print_F(floor,i,j)
        else:
            for j in range(col-1,-1,-1):
                if(floor[i][j]==1):
                    print_F(floor,i,j)
                    floor[i][j]=0
                print_F(floor,i,j)
def print_F(floor,row,col):
    print("The floor matrix is as below:")
    for r in range(len(floor)):
        for c in range(len(floor[r])):
```

```
    if r==row and c==col:
        print(f">{floor[r][c]}<", end='')
    else:
        print(f" {floor[r][c]} ", end='')
    print(end='\n')
print(end='\n')

def main():
    floor=[]
    m=int(input("Enter the No.of Rows:"))
    n=int(input("Enter the No.of columns:"))
    print("Enter clean status for each cell
          (1-dirty, 0-clean)")

    for i in range(m):
        f=list(map(int,input().split(" ")))
        floor.append(f)

    print()
    clean(floor)

#Test 1
#floor=[[1,0,0,0],
#       [0,1,0,1],
#       [1,0,1,1]]
#clean(floor)
main()
```

Output:

Enter the No. of Rows : 2

Enter the No. of columns : 2

Enter clean status for each cell (1-dirty, 0-clean)

1

0

The floor Matrix is as below:

>1<

0

The floor matrix is as below:

>0<

0

The floor matrix is as below:

0

>0<

# Lab 5: Knowledge base entailment

## Algorithm:

1) Tokenization:
   - Tokenize both 'sentence 1' and 'sentence 2' into individual words, considering punctuation and whitespace as delimiters.

2) Normalization:
   - convert all words to lowercase to make the comparison case-insensitive.
   - Remove punctuation and handle variations in word forms (e.g., stemming or lemmatization).

3) comparison:
   - Initialize a boolean variable, 'entailment Holds', to true.
   - for each normalized word in 'sentence 1':
     - check if the word is present in the Sentence 2.

4) End:
   End of algorithm.
   algo will return true if the entailment holds false if it doesn't.

Program:

```
from sympy.logic.boolalg import Implies,
from sympy.abc import P, Q

class PropositionalKnowledgeBase:
    def __init__(self):
        Self.knowledge_base = set()

    def add_statement(self, statement):
        self.Knowledge_base.add(statement)

    def check_entailment(self, query):
        return query.simplify() in self.knowledge
                                    -base

kb = PropositionalKnowledgeBase()

kb.add_Statement(Implies(P, Q))
kb.add_statement(Not(Q))

Query1 = P
Query2 = Not(P)
```

```
result1 = kb.check_entailment(query1)
result2 = kb.check_entailment(query2)

print(f"Does '{query1}' logically follow
      from the knowledge base? {result1}")
print(f"Does '{query2}' logically follow
      from the knowledge base? {result2}")
```

output:

```
Does 'p' logically follow from the
      knowledge base? True
Does '-p' logically follow from the
      knowledge base? True
```

# Knowledge Base Resolution:

## Algorithm:

1. Initialize Knowledge Base:
   - Create a class 'KnowledgeBase' to represent the knowledge base.
   - Populate the knowledge base with a set of predefined questions and answers.
2. User interaction loop:
   - Enter a loop to interact with user.
   - prompt the user to ask a question or type 'exit' to end the program.
3. User input:
   - Receive user input for the question.
4. Knowledge Resolution:
   - Use the 'resolve_query' method in the 'KnowledgeBase' class to find the answer to the user's question.
   - If an answer is found, display the answer.

otherwise, inform the user that the information is not available.

5. Exit condition:
   - If the user types 'exit', exit the loop and end the program.

Program:

```
def resolution(knowledge_base, query):
    clauses = knowledge_base + {frozenset({-query})}
    while True:
        new_clauses = set()
        for i in range(len(clauses)):
            for j in range(i+1, len(clauses)):
                resolvents = resolve(clauses[i],
                                     clauses[j])
                if frozenset() in resolvents:
                    return True
                new_clauses.update(resolvents)
        if new_clauses.issubset(clauses):
            return False
        clauses.update(new_clauses)
```

```
def resolve(ci, cj):
    resolvents = set()

    for li in ci:
        for lj in cj:
            if li == -lj or -li == lj:
                resolvent = (ci - {li}) | (cj - {lj})
                resolvents.add(frozenset(resolvent))

    return resolvents

Knowledge_base = []
while True:
    clause = input("Enter a clause for the
                    Knowledge base (or 'done' to
                    finish: ").strip()
    if clause.lower() == 'done':
        break
    knowledge_base.append("input("Enter
                    the query:"))
```

result = resolution(knowledge_base, query)
print("Is {query} entailed by the knowledge base? {result}")

output:

Enter a claus for the knowledge base: {P, -Q}
" " " : {-P, R}
" " " : {Q, -R}
" " : done

Enter the query : P
Is P entailed by the knowledge base? False

Lab 6: Unification in first order logic.

Algorithm:

1) Initialize: start with an empty substitution 'theta = {}'.

2) Comparison of Atoms: compare the two atoms '("P", "x", "y")' and '["P", "A", "B"]'.

- The first elements "P" are the same.
- The second elements "x" and "A" are different, so we need to unify them.

3) Unify Variables: Unify the variables "x" and with the term A: update 'theta' with this substitution.

4) Recursive Unification: Now, recursively unify the remaining elements.

{"x", "A", "y", "B"}

5) Result: The final substitution 'theta' is {x, A, y, B}. The two expressions

["P", "X", "Y"] and ["P", "A", "B"]. can be unified with this substitution.

Program:

```
class UnificationError(Exception):
    Pass
def print_step(step, atom1, atom2, theta):
    print(f"\nStep{step}:")
    Print(f"Unifying {atom1} and {atom2}")
    Print(f"Current Substitution Theta:
            {theta}")
def unify_var(var, x, theta):
    if var in theta:
        return unify(theta[var], x, theta)
    elif x in theta:
        return unify(var, theta[x], theta)
    else:
        theta[var] = x
        return theta
def unify(atom1, atom2, theta=None, step=1):
    if theta is None:
        theta = { }
```

```
    if atom1 == atom2:
        return theta
    else:
        raise UnificationError("Cannot Unify")
```

Output:

Step 1:

Unifying ['P', 'X', 'Y'] and ['P', 'A', 'B']

Current Substitution Theta: {}

Step 2:

Unifying P and P

current Substitution Theta: {}

Step 3:

Unifying X and A

current Substitution Theta: {}

Step 4:

Unifying Y and B

current Substitution Theta: {'X': 'A'}

Unification Successful.

Substitution theta: {'X', A', 'Y': 'B'}

convert a given first order logic statement into conjunctive Normal Form (CNF)

Algorithm:

1) Eliminate Implications:

Replace Implication with their equivalent forms using the rule $p \Rightarrow q \equiv \neg p \vee q$.

2) Move Negations inwards:

- Apply De Morgan's laws to move negations inwards.

3) Distribute Disjunctions over conjunctions:

- Distribute disjunctions ($\vee$) over conjuctions ($\wedge$) using the distributive property.

4) convert to CNF form:

- Ensure that the formula is in CNF by applying and additional simplifications if needed.

Program:

```
from sympy import symbols, And, Or, Implies, Not, to_cnf

def eliminate_implications(formula):
```

```
    return formula.subs( Implies(P,Q),
                         Or(Not(P),Q))

def Move_negations_inwards(formula):
    return formula.simplify()

def fol_to_cnf(fol_formula):
    formula_step1 = eliminate_implications
                    (fol_formula)
    formula_step2 = Move_negations_inwards
```

output:

```
Enter a first order logic : P&(~Q|R)
Fol formula : And(P,Or,Not(Q),R))
CNF formula : Or (And(P,R) And
                  (P,~Q))
```

create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.

Algorithm:

1) Knowledge Base class: (Initialize)
   - def init (self): Initiati
   create a knowledge base object with an empty list of statements.

2) Add statements:
   Add first order logic (FOL) statements to the knowledge base using add_statement method.

3) Forward Reasoning:
   - set uncharged to false to enter the loop.
   - while uncharge is False.
     - set uncharged to True.

4) Example usage:
   - create a knowledge Base object.
   - Add FOL statements to the knowledge base.
   - Ask a query to be proven using forward reasoning.

Program:

```
class KnowledgeBase:
    def __init__(self):
        self.statement = []

    def add_statement(self, statement):
        self.statements.append(statement)

    def forward_reasoning(self, query):
        working_memory = set()
        unchanged = false

while not unchanged:
    unchanged = True
    for rule in self.statements:
        if rule.conditions.issubset(working_
                memory)
            unchanged = False

return query in working_memory
```

#Example usage:

```
kb = KnowledgeBase()
kb.add_statement({"p", "q"}).kb.add_
                    statement({"q", "r"})

query = {"r"}

result = kb.forward_reasoning(query)

if result:
  print("Query is true")

else:
  print("Query is false")
```

Output:

Query is false

25.01.24