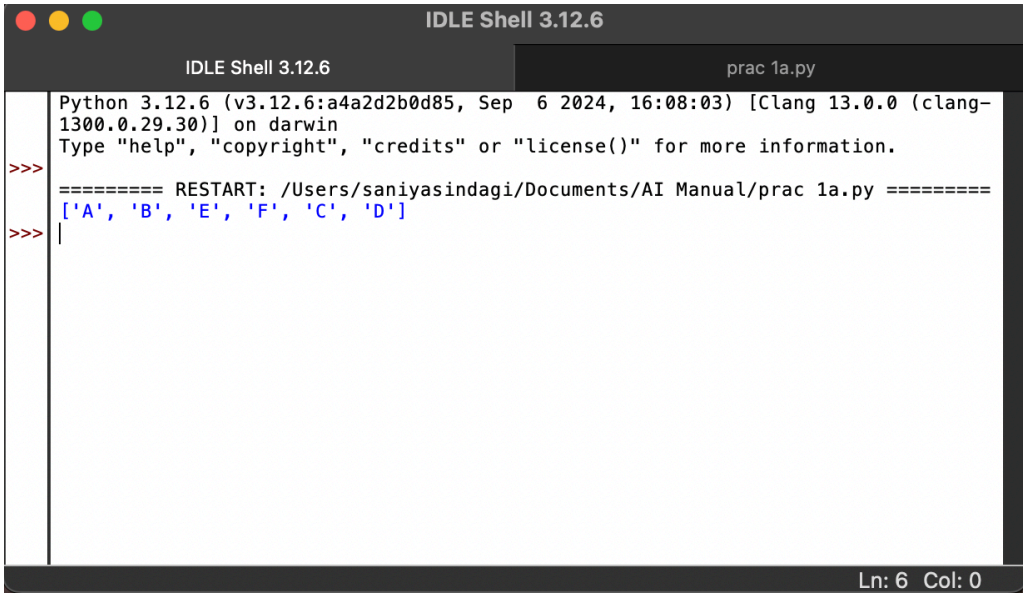
(1.a) Implement depth first search algorithm.

Code:

```
graph1 = {
   'A': set(['B', 'C']),
   'B': set(['A', 'D', 'E']), 'C': set(['A', 'F']),
   'D': set(['B']),
   'E': set(['B', 'F']),
   'F': set(['C', 'E']) }
def dfs(graph, node, visited):
   if node not in visited:
      visited.append(node)
      for n in graph[node]:
         dfs(graph,n, visited)
      return visited
visited = dfs(graph1,'A', [])
print(visited)
```

Output:



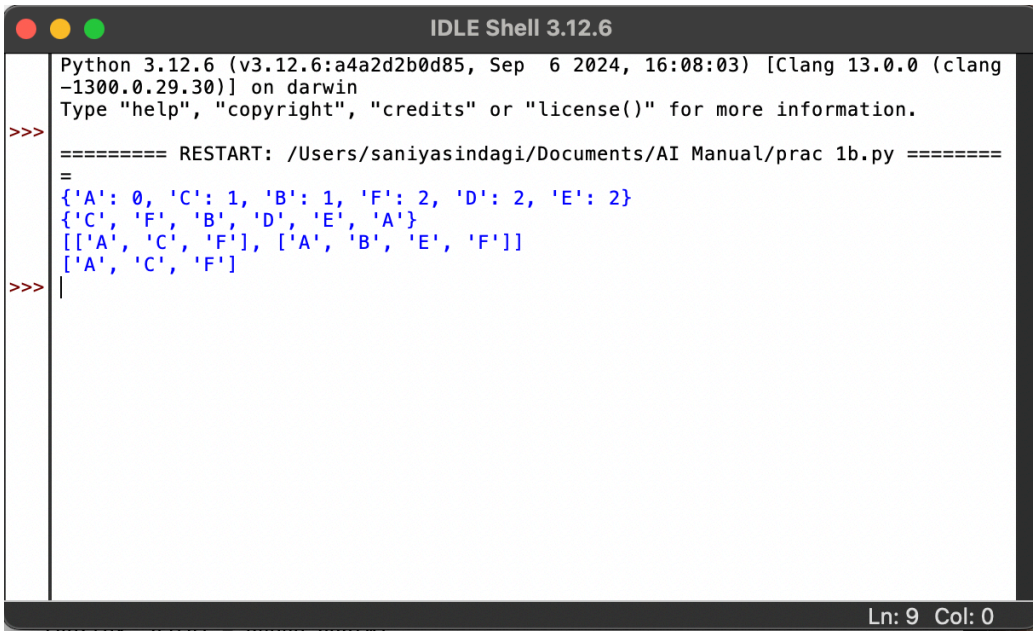
(1.b) Implement breadth first search algorithm.

Code:

```
#sample graph implemented as a dictionary
graph = {'A': set(['B', 'C']),
         'B': set(['A', 'D', 'E']),
         'C': set(['A', 'F']),
         'D': set(['B']),
         'E': set(['B', 'F']),
         'F': set(['C', 'E'])
         }

#Implement Logic of BFS
def bfs(start):
    queue = [start]
    levels={} #This Dict Keeps track of levels
    levels[start] = 0 #Depth of start node is 0
    visited = set(start)
    while queue:
        node = queue.pop(0)
        neighbours=graph[node]
        for neighbor in neighbours:
            if neighbor not in visited:
                queue.append(neighbor)
                visited.add(neighbor)
                levels[neighbor]= levels[node]+1
    print(levels) #print graph level
    return visited
print(str(bfs('A'))) #print graph node

#For Finding Breadth First Search Path
def bfs_paths(graph, start, goal):
    queue = [(start, [start])]
    while queue:
        (vertex, path) = queue.pop(0)
        for next in graph[vertex] - set(path):
            if next == goal:
                yield path + [next]
            else:
                queue.append((next, path + [next]))
result=list(bfs_paths(graph, 'A', 'F'))
print(result) #[['A', 'C', 'F'], ['A', 'B', 'E', 'F']]
```

```
#For finding shortest path
def shortest_path(graph, start, goal):
    try:
        return next(bfs_paths(graph, start, goal))
    except StopIteration:
        return None
result1=shortest_path(graph, 'A', 'F')
print(result1) #['A', 'C', 'F']
```

Output:



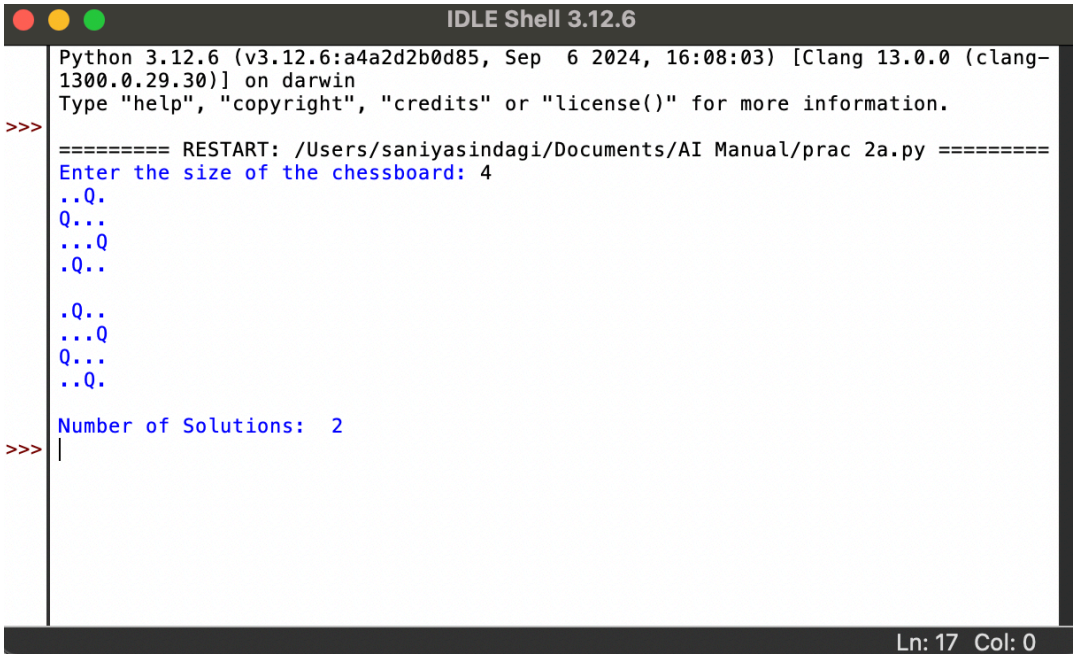
(2.a) Simulate 4-Queen/N-Queen problem.

Code:

```
def print_solutions(solutions):
    """Print all solutions in a human-readable format."""
    i=0
    if not solutions:
        print("No solutions exist")
        return
    for solution in solutions:
        for row in solution:
            print(row)
        print()  # Newline for separating solutions
        i=i+1
    print("Number of Solutions: ",i)

def solve_n_queens(n):
    """Find all solutions to the N-Queens problem."""
    def is_safe(board, row, col):
        """Check if placing a queen at (row, col) is safe."""
        # Check the current row and column
        for i in range(col):
            if board[row][i] == 'Q':
                return False

        # Check upper diagonal on the left side
        for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
            if board[i][j] == 'Q':
                return False

        # Check lower diagonal on the left side
        for i, j in zip(range(row, n, 1), range(col, -1, -1)):
            if board[i][j] == 'Q':
                return False
        return True
```

```
    def solve(board, col):
        """Use backtracking to find all solutions starting from the given column."""
        if col >= n:
            solutions.append([''.join(row) for row in board])
            return
        for i in range(n):
            if is_safe(board, i, col):
                board[i][col] = 'Q'
                solve(board, col + 1)
                board[i][col] = '.'  # Backtrack

    # Initialize an empty board
    board = [['.' for _ in range(n)] for _ in range(n)]
    solutions = []
    solve(board, 0)
    return solutions

# Example usage:
n = int(input("Enter the size of the chessboard: "))  # Size of the chessboard
solutions = solve_n_queens(n)
print_solutions(solutions)
```

Output:

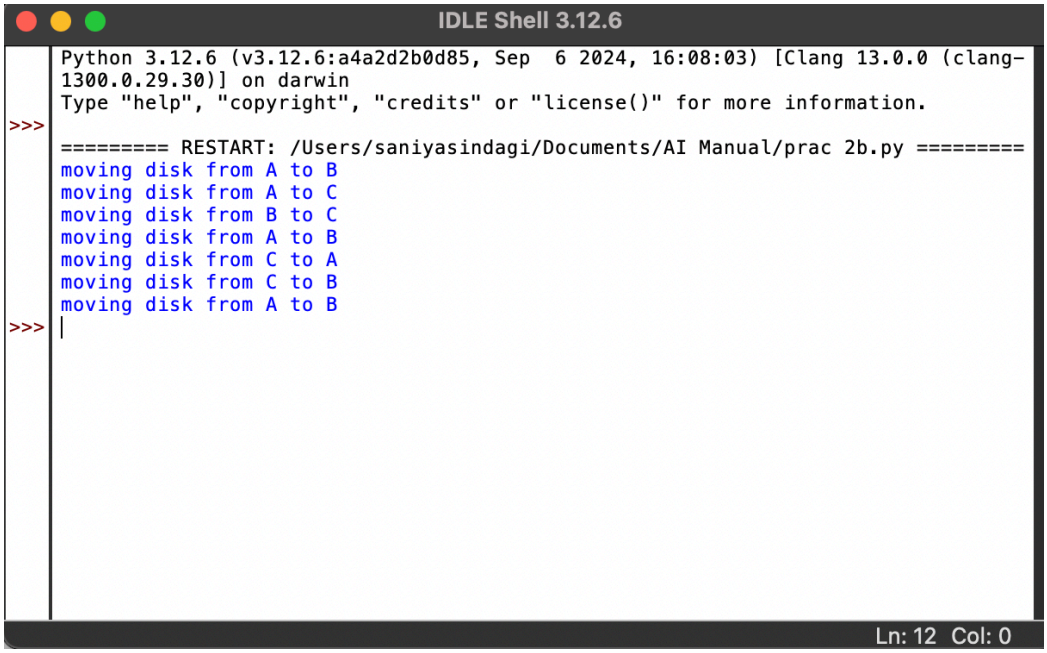


(2.b) Solve tower of Hanoi problem.

Code:

```
def moveTower(height,fromPole, toPole, withPole):
    if height >= 1:
        moveTower(height-1,fromPole,withPole,toPole)
        moveDisk(fromPole,toPole)
        moveTower(height-1,withPole,toPole,fromPole)
def moveDisk(fp,tp):
    print("moving disk from",fp,"to",tp)
moveTower(3,"A","B","C")
```

Output:

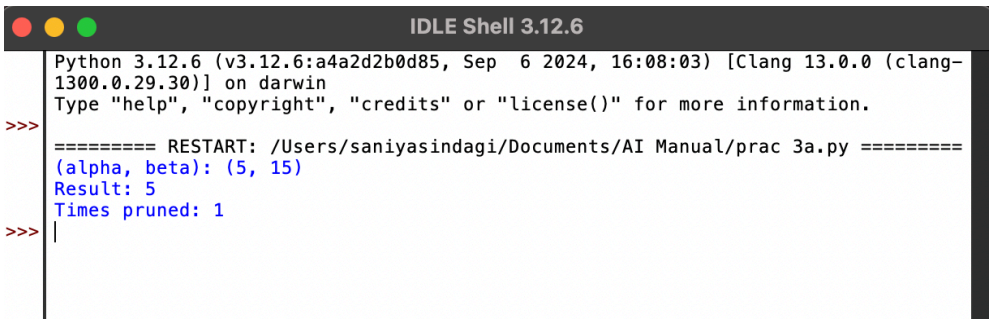


(3.a) Implement alpha beta search.

Code:

```
tree = [[[5, 1, 2], [8, -8, -9]], [[9, 4, 5], [-3, 4, 3]]]
root = 0
pruned = 0
def children(branch, depth, alpha, beta):
    global tree, pruned
    for i, child in enumerate(branch):
        if isinstance(child, list):
            nalpha, nbeta = children(child, depth + 1, alpha, beta)
            if depth % 2 == 0:
                alpha = max(alpha, nbeta)
                branch[i] = alpha
            else:
                beta = min(beta, nalpha)
        else:
            if depth % 2 == 0:
                alpha = max(alpha, child)
            else:
                beta = min(beta, child)
            if alpha >= beta:
                pruned += 1
                break
    if depth == root:
        tree = alpha if root == 0 else beta
    return alpha, beta
def alphabeta(in_tree=tree, start=root, upper=-15, lower=15):
    global pruned
    alpha, beta = children(in_tree, start, upper, lower)
    print(f"(alpha, beta): {alpha, beta}")
    print(f"Result: {tree}")
    print(f"Times pruned: {pruned}")
    return alpha, beta, tree, pruned
if __name__ == "__main__":
    alphabeta()
```

Output:

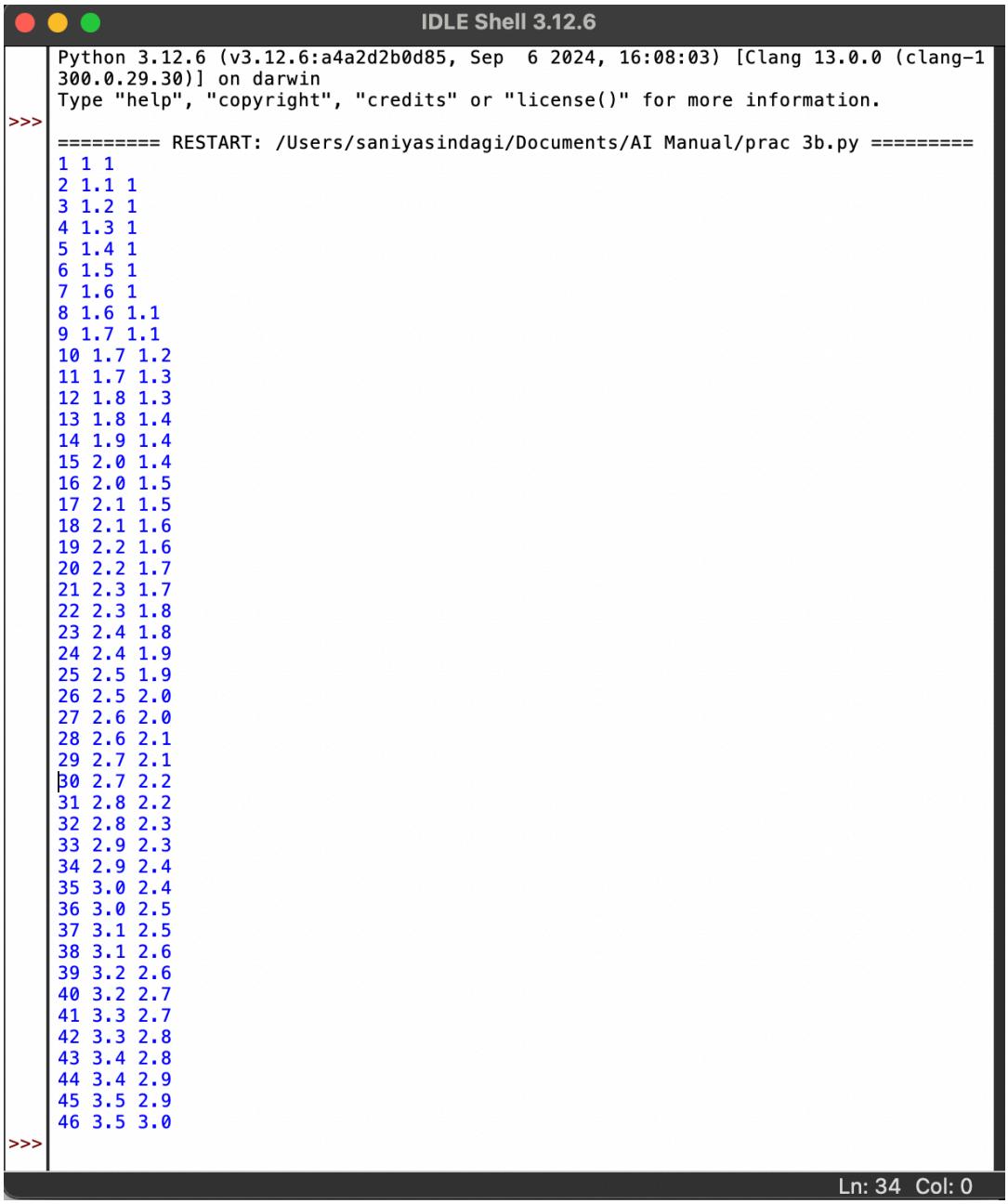


(3.b) Implement hill climbing problem.

Code:

```
import math

increment = 0.1
starting_point = [1, 1]
points = [[1, 5], [6, 4], [5, 2], [2, 1]]

def distance(x1, y1, x2, y2):
    return math.pow(x2 - x1, 2) + math.pow (y2 - y1, 2)

def sum_of_distances(x1, y1, points):
    return sum(distance(x1, y1, px, py) for px, py in points)

def new_distance(x1, y1, points):
    return [x1, y1, sum_of_distances(x1, y1, points)]

def new_points(minimum, *distances):
    return next((d[:2] for d in distances if d[2] == minimum), None)

min_distance = sum_of_distances(*starting_point, points)
flag = True
i = 1

while flag:
    d1 = new_distance(starting_point[0] + increment, starting_point[1], points)
    d2 = new_distance(starting_point[0] - increment, starting_point[1], points)
    d3 = new_distance(starting_point[0], starting_point[1] + increment, points)
    d4 = new_distance(starting_point[0], starting_point[1] - increment, points)

    print(i, round(starting_point[0], 2), round(starting_point[1], 2))

    minimum = min(d1[2], d2[2], d3[2], d4[2])

    if minimum < min_distance:
        starting_point = new_points(minimum, d1, d2, d3, d4)
        min_distance = minimum
        i += 1
    else:
        flag = False
```

Output:

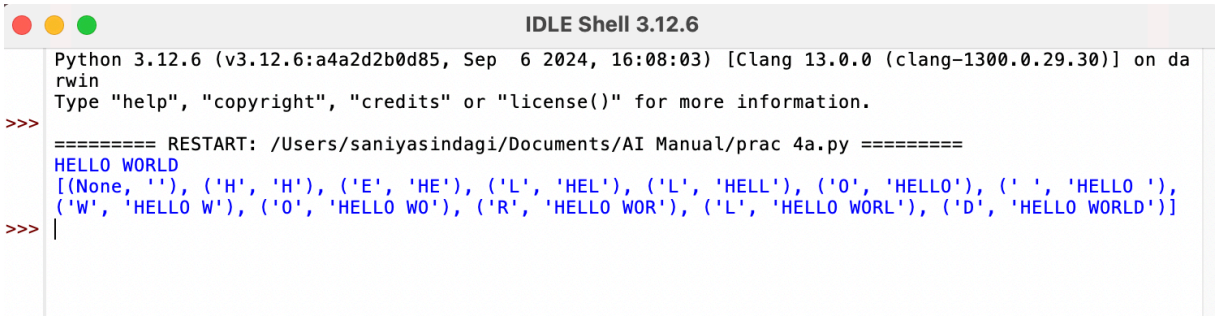

IDLE Shell 3.12.6
Python 3.12.6 (v3.12.6:a4a2d2b0d85, Sep 6 2024, 16:08:03) [Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========= RESTART: /Users/saniyasindagi/Documents/AI Manual/prac 3b.py =========
1 1 1
2 1.1 1
3 1.2 1
4 1.3 1
5 1.4 1
6 1.5 1
7 1.6 1
8 1.6 1.1
9 1.7 1.1
10 1.7 1.2
11 1.7 1.3
12 1.8 1.3
13 1.8 1.4
14 1.9 1.4
15 2.0 1.4
16 2.0 1.5
17 2.1 1.5
18 2.1 1.6
19 2.2 1.6
20 2.2 1.7
21 2.3 1.7
22 2.3 1.8
23 2.4 1.8
24 2.4 1.9
25 2.5 1.9
26 2.5 2.0
27 2.6 2.0
28 2.6 2.1
29 2.7 2.1
30 2.7 2.2
31 2.8 2.2
32 2.8 2.3
33 2.9 2.3
34 2.9 2.4
35 3.0 2.4
36 3.0 2.5
37 3.1 2.5
38 3.1 2.6
39 3.2 2.6
40 3.2 2.7
41 3.3 2.7
42 3.3 2.8
43 3.4 2.8
44 3.4 2.9
45 3.5 2.9
46 3.5 3.0
>>>
Ln: 34 Col: 0

(4.a) Implement A* algorithm.

Code:

```
from simpleai.search import SearchProblem, astar

GOAL = 'HELLO WORLD'
class HelloProblem(SearchProblem):
    def actions(self, state):
        if len(state) < len(GOAL):
            return list(' ABCDEFGHIJKLMNOPQRSTUVWXYZ')
        else:
            return []
    def result(self, state, action):
        return state + action
    def is_goal(self, state):
        return state == GOAL
    def heuristic(self, state):
        # how far are we from the goal?
        wrong = sum([1 if state[i] != GOAL[i] else 0
                     for i in range(len(state))])
        missing = len(GOAL) - len(state)
        return wrong + missing
problem = HelloProblem(initial_state='')
result = astar(problem)
print(result.state)
print(result.path())
```

Output:



(4.b) Solve water jug problem.

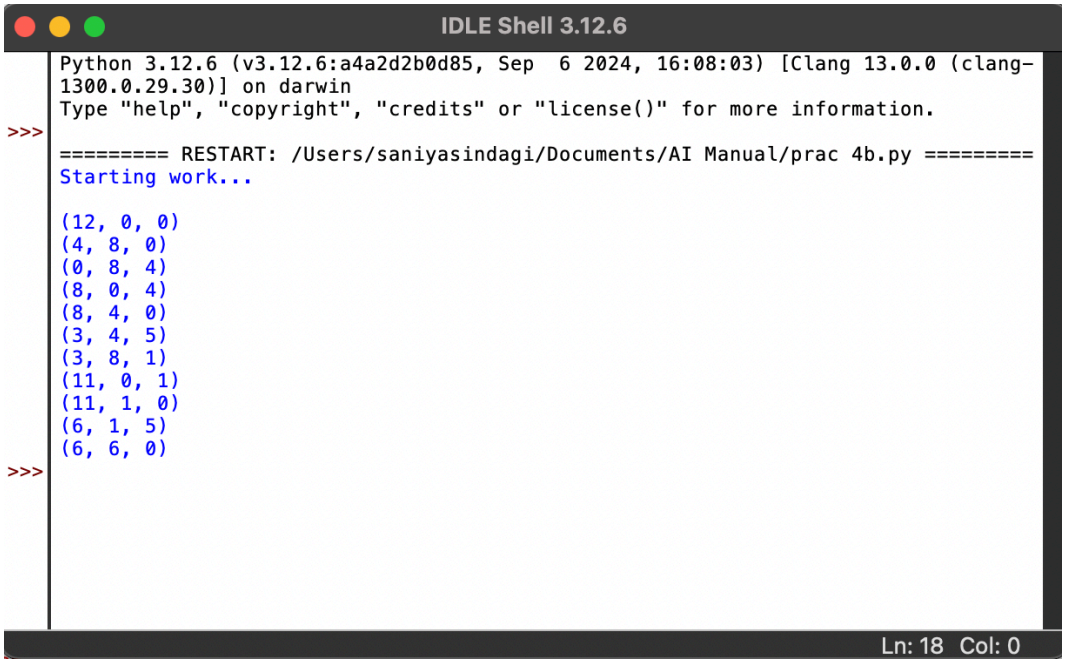
Code:

```
# 3 water jugs capacity -> (x,y,z) where x>y>z
# initial state (12,0,0)
# final state (6,6,0)

capacity = (12,8,5)
# Maximum capacities of 3 jugs -> x,y,z
x = capacity[0]
y = capacity[1]
z = capacity[2]
# to mark visited states
memory = {}

# store solution path
ans = []

def get_all_states(state):
    # Let the 3 jugs be called a,b,c
    a = state[0]
    b = state[1]
    c = state[2]

    if(a==6 and b==6):
        ans.append(state)
        return True

    # if current state is already visited earlier
    if((a,b,c) in memory):
        return False
    memory[(a,b,c)] = 1

    #empty jug a
    if(a>0):
        #empty a into b
        if(a+b<=y):
            if( get_all_states((0,a+b,c)) ):
                ans.append(state)
                return True
        else:
            if( get_all_states((a-(y-b), y, c)) ):
                ans.append(state)
                return True
```

```
        #empty a into c
        if(a+c<=z):
            if( get_all_states((0,b,a+c)) ):
                ans.append(state)
                return True
        else:
            if( get_all_states((a-(z-c), b, z)) ):
                ans.append(state)
                return True
  #empty jug b
if(b>0):
    #empty b into a
    if(a+b<=x):
        if( get_all_states((a+b, 0, c)) ):
            ans.append(state)
            return True
    else:
        if( get_all_states((x, b-(x-a), c)) ):
            ans.append(state)
            return True
#empty b into c
if(b+c<=z):
    if( get_all_states((a, 0, b+c)) ):
        ans.append(state)
        return True
else:
    if( get_all_states((a, b-(z-c), z)) ):
        ans.append(state)
        return True

#empty jug c
if(c>0):
    #empty c into a
    if(a+c<=x):
        if( get_all_states((a+c, b, 0)) ):
            ans.append(state)
            return True
    else:
        if( get_all_states((x, b, c-(x-a))) ):
            ans.append(state)
            return True
```

```
        #empty c into b
        if(b+c<=y):
            if( get_all_states((a, b+c, 0)) ):
                ans.append(state)
                return True
        else:
            if( get_all_states((a, y, c-(y-b))) ):
                ans.append(state)
                return True

    return False

initial_state = (12,0,0)
print("Starting work...\n")
get_all_states(initial_state)
ans.reverse()
for i in ans:
    print(i)
```

Output:



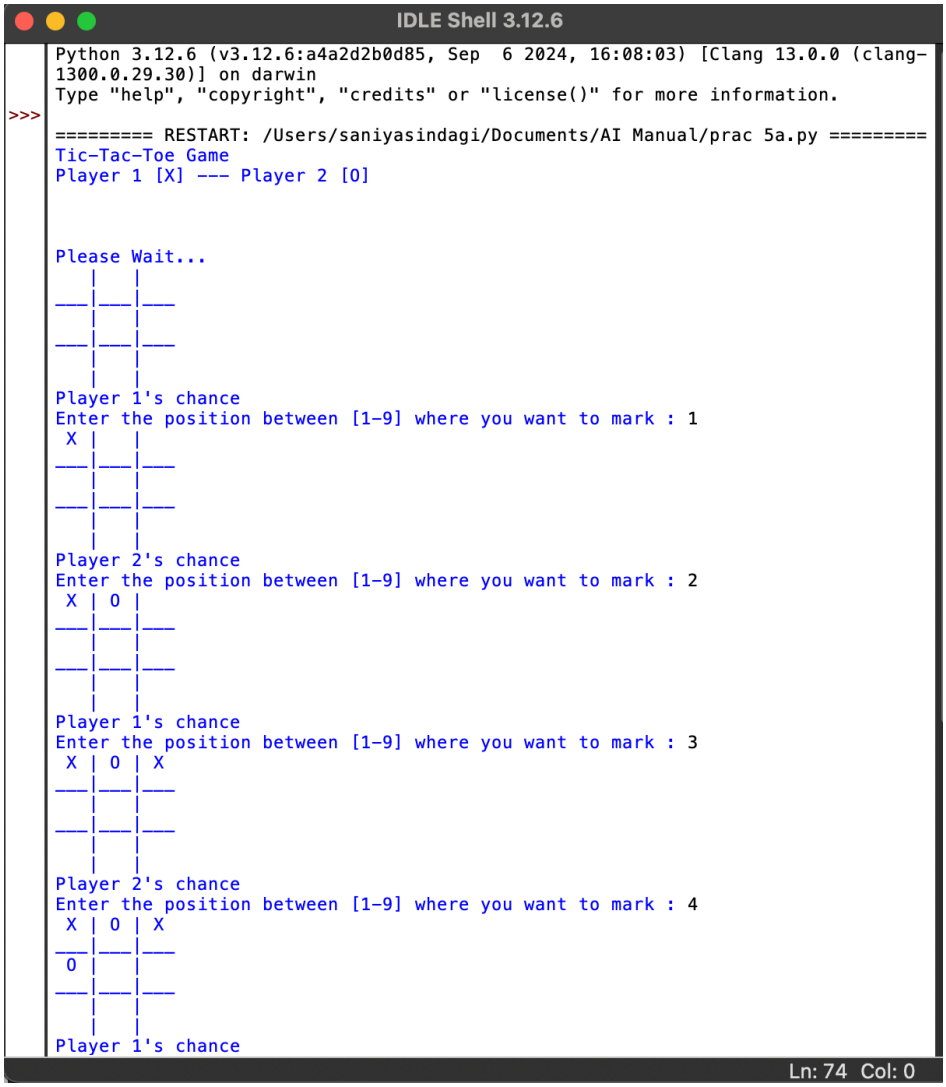
(5.a) Simulate tic-tac-toe game using min-max algorithm.

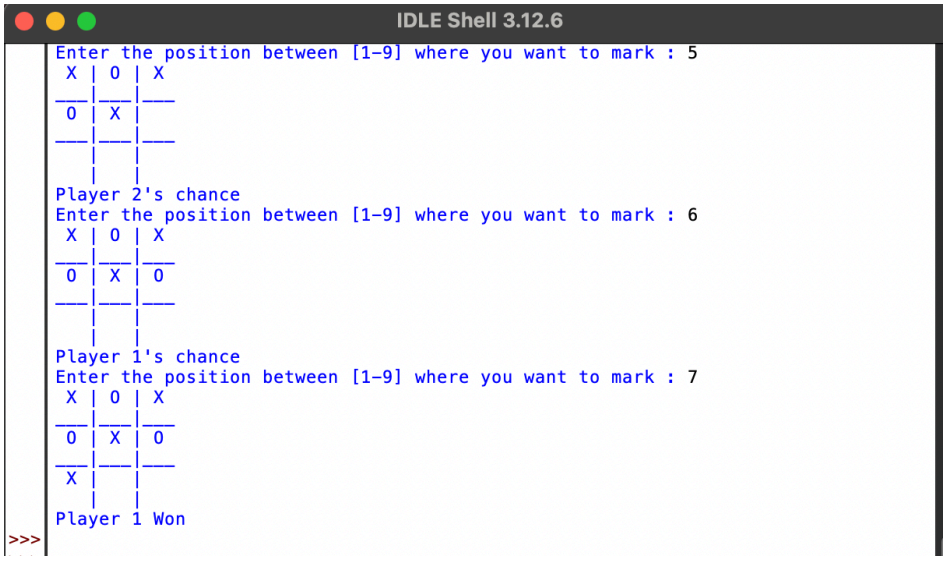
Code:

```
Import os
import time

board = [' ',' ',' ',' ',' ',' ',' ',' ',' ',' ']
player = 1

########win Flags##########
Win = 1
Draw = -1
Running = 0
Stop = 1
############################
Game = Running
Mark = 'X'

#This Function Draws Game Board
def DrawBoard():
    print(" %c | %c | %c " % (board[1],board[2],board[3]))
    print("___|___|___")
    print(" %c | %c | %c " % (board[4],board[5],board[6]))
    print("___|___|___")
    print(" %c | %c | %c " % (board[7],board[8],board[9]))
    print("   |   |   ")

#This Function Checks position is empty or not
def CheckPosition(x):
    if(board[x] == ' '):
        return True
    else:
        return False

#This Function Checks player has won or not
def CheckWin():
    global Game
    #Horizontal winning condition
    if(board[1] == board[2] and board[2] == board[3] and board[1] != ' '):
        Game = Win
    elif(board[4] == board[5] and board[5] == board[6] and board[4] != ' '):
        Game = Win
    elif(board[7] == board[8] and board[8] == board[9] and board[7] != ' '):
        Game = Win
    #Vertical Winning Condition
    elif(board[1] == board[4] and board[4] == board[7] and board[1] != ' '):
        Game = Win
    elif(board[2] == board[5] and board[5] == board[8] and board[2] != ' '):
```

```
        Game = Win
    elif(board[3] == board[6] and board[6] == board[9] and board[3] != ' '):
        Game=Win
    #Diagonal Winning Condition
    elif(board[1] == board[5] and board[5] == board[9] and board[5] != ' '):
        Game = Win
    elif(board[3] == board[5] and board[5] == board[7] and board[5] != ' '):
        Game=Win
    #Match Tie or Draw Condition
    elif(board[1]!=' ' and board[2]!=' ' and board[3]!=' ' and board[4]!=' ' and board[5]!=' ' and
board[6]!=' ' and board[7]!=' ' and board[8]!=' ' and board[9]!=' '):
        Game=Draw
    else:
        Game=Running

print("Tic-Tac-Toe Game")
print("Player 1 [X] --- Player 2 [O]\n")
print()
print()
print("Please Wait...")
time.sleep(1)
while(Game == Running):
    os.system('cls')
    DrawBoard()
    if(player % 2 != 0):
        print("Player 1's chance")
        Mark = 'X'
    else:
        print("Player 2's chance")
        Mark = 'O'
    choice = int(input("Enter the position between [1-9] where you want to mark : "))
    if(CheckPosition(choice)):
        board[choice] = Mark
        player+=1
        CheckWin()

os.system('cls')
DrawBoard()
if(Game==Draw):
    print("Game Draw")
elif(Game==Win):
    player-=1
    if(player%2!=0):
        print("Player 1 Won")
    else:
        print("Player 2 Won")
```

Output:


IDLE Shell 3.12.6
Python 3.12.6 (v3.12.6:a4a2d2b0d85, Sep 6 2024, 16:08:03) [Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: /Users/saniyasindagi/Documents/AI Manual/prac 5a.py ==========
Tic-Tac-Toe Game
Player 1 [X] --- Player 2 [O]
Please Wait...
Player 1's chance
Enter the position between [1-9] where you want to mark : 1
Player 2's chance
Enter the position between [1-9] where you want to mark : 2
Player 1's chance
Enter the position between [1-9] where you want to mark : 3
Player 2's chance
Enter the position between [1-9] where you want to mark : 4
Player 1's chance
Ln: 74 Col: 0


IDLE Shell 3.12.6
Enter the position between [1-9] where you want to mark : 5
Player 2's chance
Enter the position between [1-9] where you want to mark : 6
Player 1's chance
Enter the position between [1-9] where you want to mark : 7
Player 1 Won
>>>

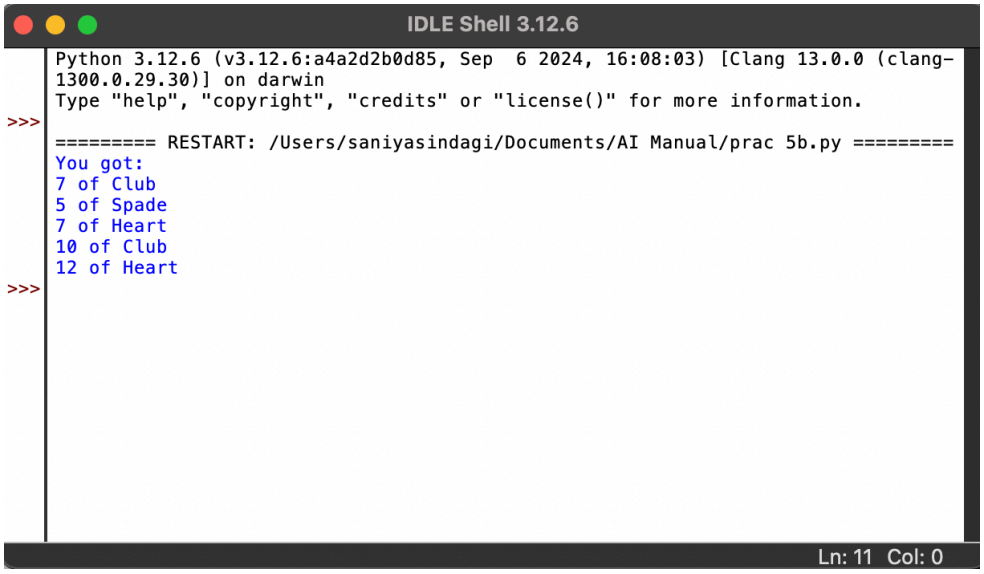
(5.b) Shuffle deck of cards.

Code:

(# Python program to shuffle a deck of card using the module random anddraw 5 cards

```
# import modules
import itertools, random

# make a deck of cards
deck = list(itertools.product(range(1,14),['Spade','Heart','Diamond','Club']))

# shuffle the cards
random.shuffle(deck)

# draw five cards
print("You got:")
for i in range(5):
    print(deck[i][0], "of", deck[i][1])
```

Output:



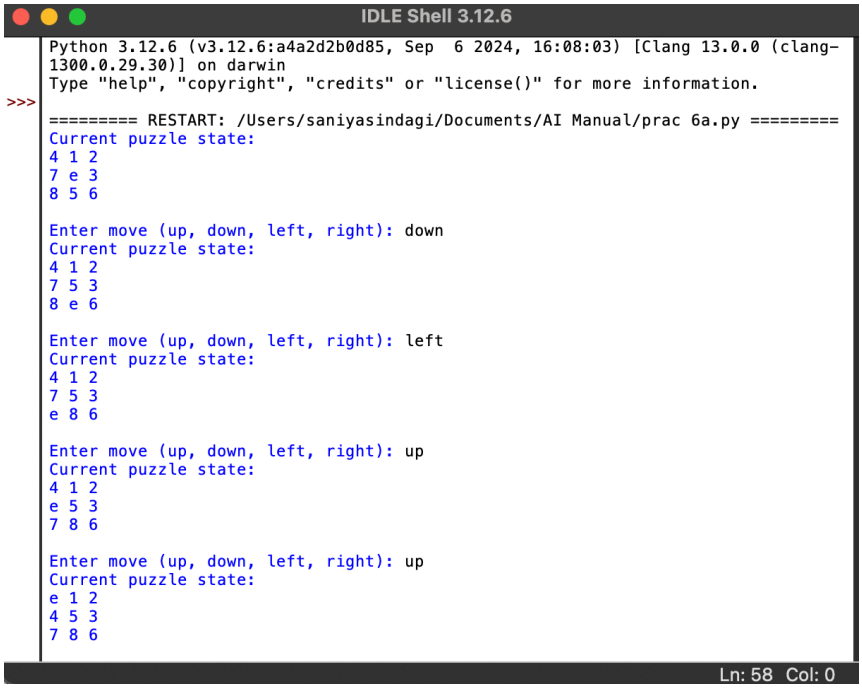
(6.a) Design an application to simulate number puzzle problem.

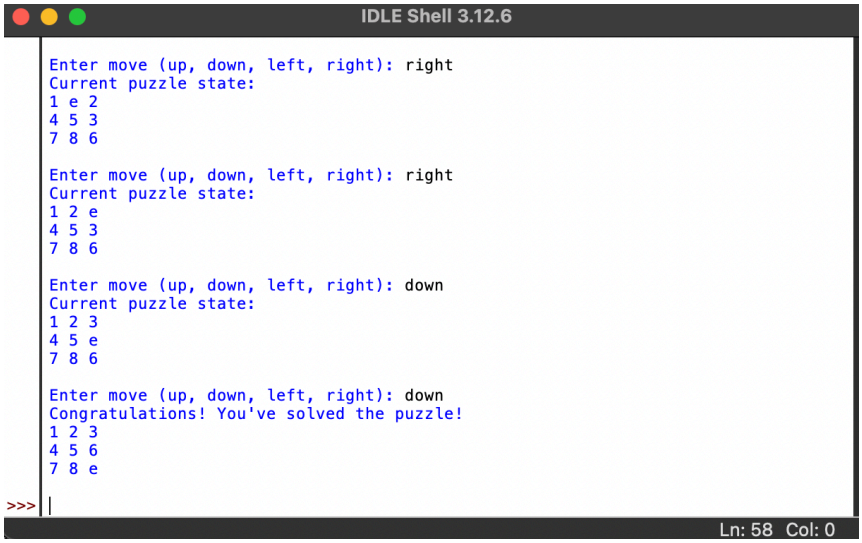
Code:

```
class Puzzle:
    def __init__(self, initial_state):
        self.grid = [list(row) for row in initial_state]
        self.empty_pos = self.find_empty_pos()

    def find_empty_pos(self):
        for r in range(3):
            for c in range(3):
                if self.grid[r][c] == 'e':
                    return (r, c)
        return None

    def move(self, direction):
        r, c = self.empty_pos
        if direction == 'up' and r > 0:
            self.swap(r, c, r-1, c)
        elif direction == 'down' and r < 2:
            self.swap(r, c, r+1, c)
        elif direction == 'left' and c > 0:
            self.swap(r, c, r, c-1)
        elif direction == 'right' and c < 2:
            self.swap(r, c, r, c+1)
        else:
            print("Invalid move")

    def swap(self, r1, c1, r2, c2):
        self.grid[r1][c1], self.grid[r2][c2] = self.grid[r2][c2], self.grid[r1][c1]
        self.empty_pos = (r2, c2)

    def is_solved(self):
        goal = [['1', '2', '3'], ['4', '5', '6'], ['7', '8', 'e']]
        return self.grid == goal

    def print_grid(self):
        for row in self.grid:
            print(" ".join(row))
        print()

def main():
    initial_state = [['4', '1', '2'], ['7', 'e', '3'], ['8', '5', '6']]
    puzzle = Puzzle(initial_state)

    while not puzzle.is_solved():
        print("Current puzzle state:")
        puzzle.print_grid()
```

```
        move = input("Enter move (up, down, left, right): ").strip().lower()
        if move in ['up', 'down', 'left', 'right']:
            puzzle.move(move)
        else:
            print("Invalid input. Please enter 'up', 'down', 'left', or 'right'.")

    print("Congratulations! You've solved the puzzle!")
    puzzle.print_grid()

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

Output:



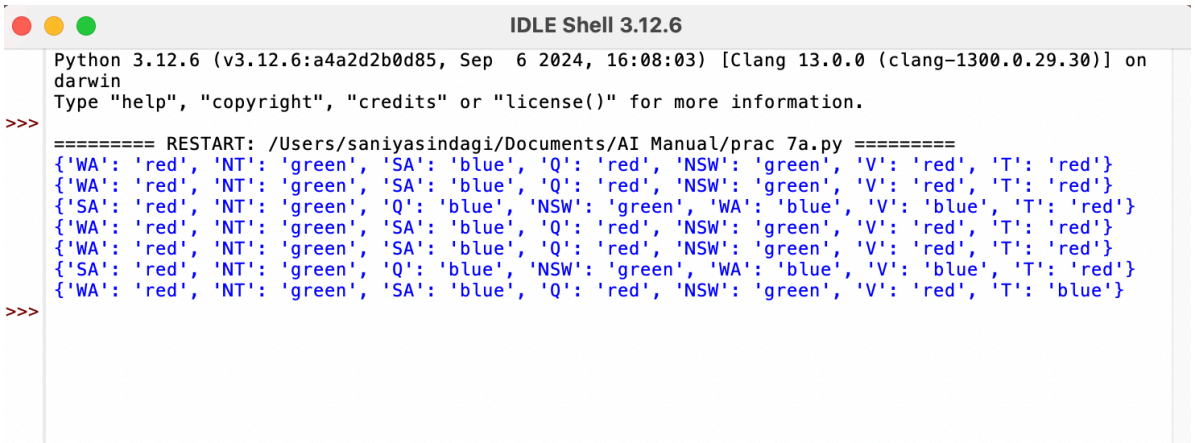


(7.a) Solve constraint satisfaction problem.

Code:

```
from __future__ import print_function
from simpleai.search import CspProblem, backtrack, min_conflicts,
MOST_CONSTRAINED_VARIABLE, HIGHEST_DEGREE_VARIABLE,
LEAST_CONSTRAINING_VALUE
variables = ('WA', 'NT', 'SA', 'Q', 'NSW', 'V', 'T')
domains = dict((v, ['red', 'green', 'blue']) for v in variables)
def const_different(variables, values):
 return values[0] != values[1] # expect the value of the neighbors to be different
constraints = [
 (('WA', 'NT'), const_different),
 (('WA', 'SA'), const_different),
 (('SA', 'NT'), const_different),
 (('SA', 'Q'), const_different),
 (('NT', 'Q'), const_different),
 (('SA', 'NSW'), const_different),
 (('Q', 'NSW'), const_different),
 (('SA', 'V'), const_different),
 (('NSW', 'V'), const_different),
]
my_problem = CspProblem(variables, domains, constraints)
print(backtrack(my_problem))
print(backtrack(my_problem,
variable_heuristic=MOST_CONSTRAINED_VARIABLE))
print(backtrack(my_problem,
variable_heuristic=HIGHEST_DEGREE_VARIABLE))
print(backtrack(my_problem,
value_heuristic=LEAST_CONSTRAINING_VALUE))
print(backtrack(my_problem,
variable_heuristic=MOST_CONSTRAINED_VARIABLE,
value_heuristic=LEAST_CONSTRAINING_VALUE))
print(backtrack(my_problem,
variable_heuristic=HIGHEST_DEGREE_VARIABLE,
value_heuristic=LEAST_CONSTRAINING_VALUE))
print(min_conflicts(my_problem))
```

Output:

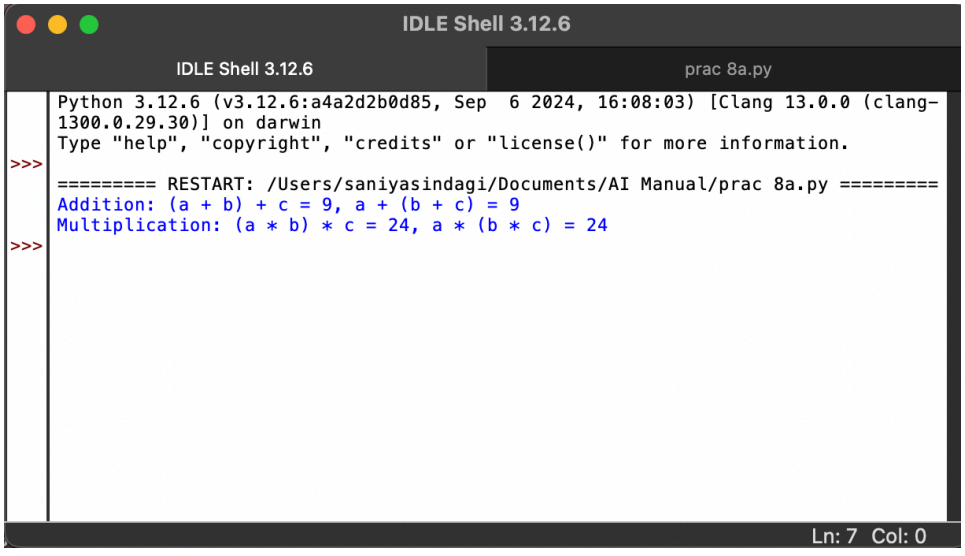


(8.a) Derive the expression based on Associatiive Law.

Code:

```
#Practical 8a
def associative_law_addition(a, b, c):
    # (a + b) + c == a + (b + c)
    left_side = (a + b) + c
    right_side = a + (b + c)
    return left_side, right_side

def associative_law_multiplication(a, b, c):
    # (a * b) * c == a * (b * c)
    left_side = (a * b) * c
    right_side = a * (b * c)
    return left_side, right_side

# Test values
a = 2
b = 3
c = 4

# Check Associative Law for Addition
add_left, add_right = associative_law_addition(a, b, c)
print(f"Addition: (a + b) + c = {add_left}, a + (b + c) = {add_right}")

# Check Associative Law for Multiplication
mul_left, mul_right = associative_law_multiplication(a, b, c)
print(f"Multiplication: (a * b) * c = {mul_left}, a * (b * c) = {mul_right}")
```

Output:

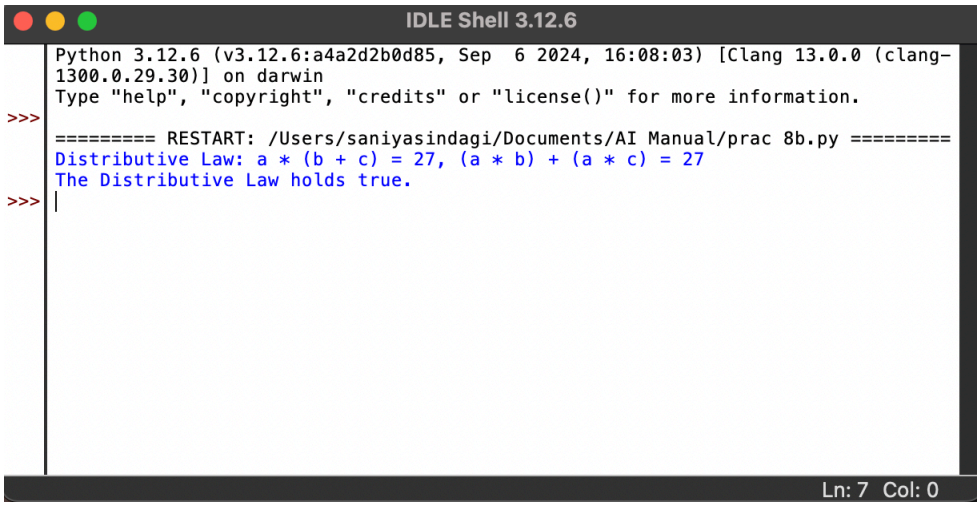


(8.b) Derive the expression based on Distributive Law.

Code:

```
#Practical 8b
def distributive_law(a, b, c):
    # Calculate both sides of the distributive law
    left_side = a * (b + c)           # Left side: a * (b + c)
    right_side = (a * b) + (a * c)    # Right side: (a * b) + (a * c)
    return left_side, right_side

# Test values
a = 3
b = 4
c = 5

# Check Distributive Law
left_result, right_result = distributive_law(a, b, c)
print(f"Distributive Law: a * (b + c) = {left_result}, (a * b) + (a * c) = {right_result}")

# Verify if both sides are equal
if left_result == right_result:
    print("The Distributive Law holds true.")
else:
    print("The Distributive Law does not hold true.")
```

Output:



(9.a) Derive the predicate.

Code:

```
# Define facts as a dictionary where key is the subject and value is the predicate
facts = {
    'Sachin': 'batsman',
    'batsman': 'cricketer'
}

# Function to derive the final fact from the initial fact
def derive_fact(starting_subject, target_predicate):
    current_subject = starting_subject
    while current_subject in facts:
        current_subject = facts[current_subject]
        if current_subject == target_predicate:
            return f"{starting_subject} is {target_predicate}"
    return "Cannot derive the fact"

# Define the starting subject and the target predicate
starting_subject = 'Sachin'
target_predicate = 'cricketer'

# Derive and print the fact
result = derive_fact(starting_subject, target_predicate)
print(result)
```

Output:

(10.a) Write a program which contains three predicates.

Code:

```
# Facts about individuals
male = {'John', 'Robert', 'Michael', 'Kevin'}
female = {'Mary', 'Patricia', 'Jennifer', 'Linda'}

parents = {
   'John': ['Michael', 'Sarah'],   # John is the father of Michael and Sarah
   'Mary': ['Michael', 'Sarah'],    # Mary is the mother of Michael and Sarah
   'Robert': ['Kevin'],             # Robert is the father of Kevin
   'Patricia': ['Kevin']            # Patricia is the mother of Kevin
}
def is_father(father, child):
   return father in male and child in parents.get(father, [])

def is_mother(mother, child):
   return mother in female and child in parents.get(mother, [])

def is_grandfather(grandfather, grandchild):
   return grandfather in male and any(is_father(parent, grandchild) or is_mother(parent, grandchild)
for parent in parents.get(grandfather, []))

def is_grandmother(grandmother, grandchild):
   return grandmother in female and any(is_father(parent, grandchild) or is_mother(parent,
grandchild) for parent in parents.get(grandmother, []))

def is_brother(sibling1, sibling2):
   return sibling1 in male and sibling2 in parents.get(sibling1, parents[sibling2])

def is_sister(sibling1, sibling2):
   return sibling1 in female and sibling2 in parents.get(sibling1, parents[sibling2])

def is_uncle(uncle, nephew):
   return uncle in male and (any(is_brother(uncle, parent) for parent in parents.get(nephew, [])) or
any(is_sister(uncle, parent) for parent in parents.get(nephew, [])))

def is_aunt(aunt, niece):
   return aunt in female and (any(is_brother(aunt, parent) for parent in parents.get(niece, [])) or
any(is_sister(aunt, parent) for parent in parents.get(niece, [])))

def is_cousin(cousin1, cousin2):
   return any(parent1 != parent2 and (parent1 in parents.get(cousin2, []) or parent2 in
parents.get(cousin1, []))
              for parent1 in parents.get(cousin1, [])
              for parent2 in parents.get(cousin2, []))
```

```
# Example Queries
print("Is John the father of Michael?", is_father('John', 'Michael'))
print("Is Mary the mother of Sarah?", is_mother('Mary', 'Sarah'))
print("Is Robert a grandfather of Michael?", is_grandfather('Robert', 'Michael'))
print("Is Patricia an aunt of Kevin?", is_aunt('Patricia', 'Kevin'))
```

Output: