1.8 PUZZLE

class Puzzle8:

def \_\_init\_\_(self, initial\_state=None):

self.goal\_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

if initial\_state:

self.state = initial\_state

else:

self.state = self.goal\_state.copy()

self.moves = 0

def get\_user\_input(self):

print("Enter the initial state of the puzzle row by row, use '0' for the blank tile:")

self.state = []

for \_ in range(3):

row = input().split()

self.state.append([int(cell) for cell in row])

def get\_valid\_moves(self):

row, col = self.find\_blank()

valid\_moves = []

if row > 0:

valid\_moves.append((-1, 0)) # Move the blank tile up

if row < 2:

valid\_moves.append((1, 0)) # Move the blank tile down

if col > 0:

valid\_moves.append((0, -1)) # Move the blank tile left

if col < 2:

valid\_moves.append((0, 1)) # Move the blank tile right

return valid\_moves

def move(self, direction):

row, col = self.find\_blank()

dr, dc = direction

self.state[row][col], self.state[row + dr][col + dc] = self.state[row + dr][col + dc], self.state[row][col]

self.moves += 1

def find\_blank(self):

for i, row in enumerate(self.state):

if 0 in row:

return i, row.index(0)

def \_\_str\_\_(self):

return "\n".join(" ".join(str(cell) for cell in row) for row in self.state)

# Example usage

puzzle = Puzzle8()

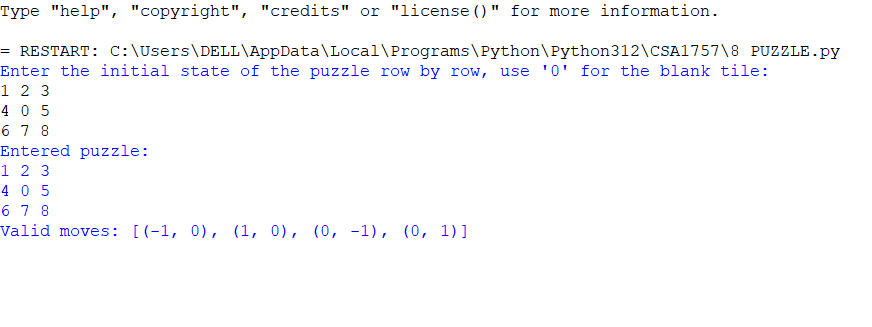
puzzle.get\_user\_input()

print("Entered puzzle:")

print(puzzle)

print("Valid moves:", puzzle.get\_valid\_moves())

OUTPUT



2.8 QUEENS PROBLEM

print ("Enter the number of queens")

N = int(input())

board = [[0]\*N for \_ in range(N)]

def attack(i, j):

for k in range(0,N):

if board[i][k]==1 or board[k][j]==1:

return True

for k in range(0,N):

for l in range(0,N):

if (k+l==i+j) or (k-l==i-j):

if board[k][l]==1:

return True

return False

def N\_queens(n):

if n==0:

return True

for i in range(0,N):

for j in range(0,N):

if (not(attack(i,j))) and (board[i][j]!=1):

board[i][j] = 1

if N\_queens(n-1)==True:

return True

board[i][j] = 0

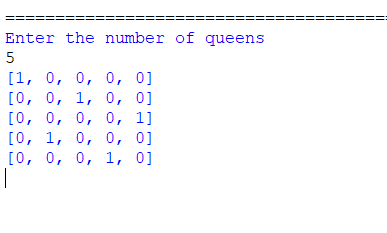
return False

N\_queens(N)

for i in board:

print (i)

OUTPUT



3.WATER JUG

from collections import deque

def Solution(a, b, target):

m = {}

isSolvable = False

path = []

q = deque()

q.append((0, 0))

while (len(q) > 0):

u = q.popleft()

if ((u[0], u[1]) in m):

continue

if ((u[0] > a or u[1] > b or

u[0] < 0 or u[1] < 0)):

continue

path.append([u[0], u[1]])

m[(u[0], u[1])] = 1

if (u[0] == target or u[1] == target):

isSolvable = True

if (u[0] == target):

if (u[1] != 0):

path.append([u[0], 0])

else:

if (u[0] != 0):

path.append([0, u[1]])

sz = len(path)

for i in range(sz):

print("(", path[i][0], ",",

path[i][1], ")")

break

q.append([u[0], b])

q.append([a, u[1]])

for ap in range(max(a, b) + 1):

c = u[0] + ap

d = u[1] - ap

if (c == a or (d == 0 and d >= 0)):

q.append([c, d])

c = u[0] - ap

d = u[1] + ap

if ((c == 0 and c >= 0) or d == b):

q.append([c, d])

q.append([a, 0])

q.append([0, b])

if (not isSolvable):

print("Solution not possible")

if \_\_name\_\_ == '\_\_main\_\_':

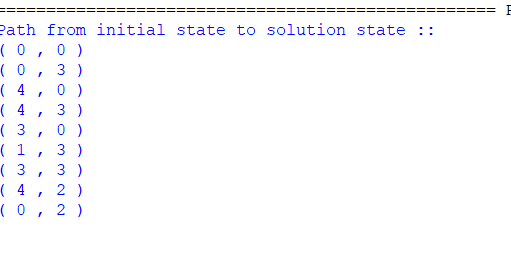
Jug1, Jug2, target = 4, 3, 2

print("Path from initial state "

"to solution state ::")

Solution(Jug1, Jug2, target)

OUTPUT



4. CRIPT ARITHMETIC

import itertools

def get\_value(word, substitution):

s = 0

factor = 1

for letter in reversed(word):

s += factor \* substitution[letter]

factor \*= 10

return s

def solve2(equation):

left, right = equation.lower().replace(' ', '').split('=')

left = left.split('+')

letters = set(right)

for word in left:

for letter in word:

letters.add(letter)

letters = list(letters)

digits = range(10)

for perm in itertools.permutations(digits, len(letters)):

sol = dict(zip(letters, perm))

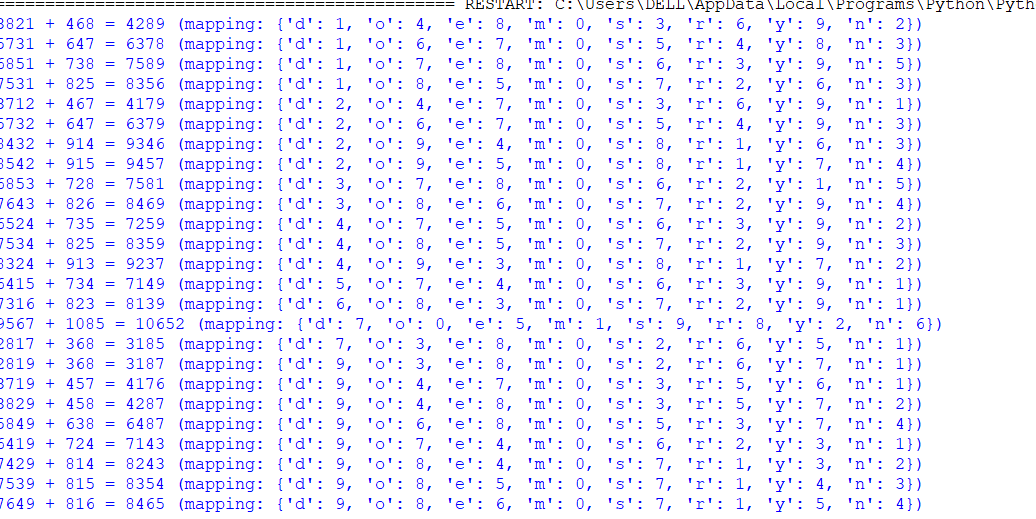
if sum(get\_value(word, sol) for word in left) == get\_value(right, sol):

print(' + '.join(str(get\_value(word, sol)) for word in left) + " = {} (mapping: {})".format(get\_value(right, sol), sol))

if \_\_name\_\_ == '\_\_main\_\_':

solve2('SEND + MORE = MONEY')

OUTPUT



5.MISSIONAROIES AND CANNIBALS

print("\n")

print("\tGame Start\nNow the task is to move all of them to right side of the river")

print("rules:\n1. The boat can carry at most two people\n2. If cannibals num greater than missionaries then the cannibals would eat the missionaries\n3. The boat cannot cross the river by itself with no people on board")

lM = 3 #lM = Left side Missionaries number

lC = 3 #lC = Laft side Cannibals number

rM=0 #rM = Right side Missionaries number

rC=0 #rC = Right side cannibals number

userM = 0 #userM = User input for number of missionaries for right to left side travel

userC = 0 #userC = User input for number of cannibals for right to left travel

k = 0

print("\nM M M C C C | --- | \n")

try:

while(True):

while(True):

print("Left side -> right side river travel")

#uM = user input for number of missionaries for left to right travel

#uC = user input for number of cannibals for left to right travel

uM = int(input("Enter number of Missionaries travel => "))

uC = int(input("Enter number of Cannibals travel => "))

if((uM==0)and(uC==0)):

print("Empty travel not possible")

print("Re-enter : ")

elif(((uM+uC) <= 6)and((lM-uM)>=0)and((lC-uC)>=0)):

break

else:

print("Wrong input re-enter : ")

lM = (lM-uM)

lC = (lC-uC)

rM += uM

rC += uC

print("\n")

for i in range(0,lM):

print("M ",end="")

for i in range(0,lC):

print("C ",end="")

print("| --> | ",end="")

for i in range(0,rM):

print("M ",end="")

for i in range(0,rC):

print("C ",end="")

print("\n")

k +=1

if(((lC==3)and (lM == 1))or((lC==3)and(lM==2))or((lC==2)and(lM==1))or((rC==3)and (rM == 1))or((rC==3)and(rM==2))or((rC==2)and(rM==1))):

print("Cannibals eat missionaries:\nYou lost the game")

break

if((rM+rC) == 6):

print("You won the game : \n\tCongrats")

print("Total attempt")

print(k)

break

while(True):

print("Right side -> Left side river travel")

userM = int(input("Enter number of Missionaries travel => "))

userC = int(input("Enter number of Cannibals travel => "))

if((userM==0)and(userC==0)):

print("Empty travel not possible")

print("Re-enter : ")

elif(((userM+userC) <= 2)and((rM-userM)>=0)and((rC-userC)>=0)):

break

else:

print("Wrong input re-enter : ")

lM += userM

lC += userC

rM -= userM

rC -= userC

k +=1

print("\n")

for i in range(0,lM):

print("M ",end="")

for i in range(0,lC):

print("C ",end="")

print("| <-- | ",end="")

for i in range(0,rM):

print("M ",end="")

for i in range(0,rC):

print("C ",end="")

print("\n")

if(((lC==3)and (lM == 1))or((lC==3)and(lM==2))or((lC==2)and(lM==1))or((rC==3)and (rM == 1))or((rC==3)and(rM==2))or((rC==2)and(rM==1))):

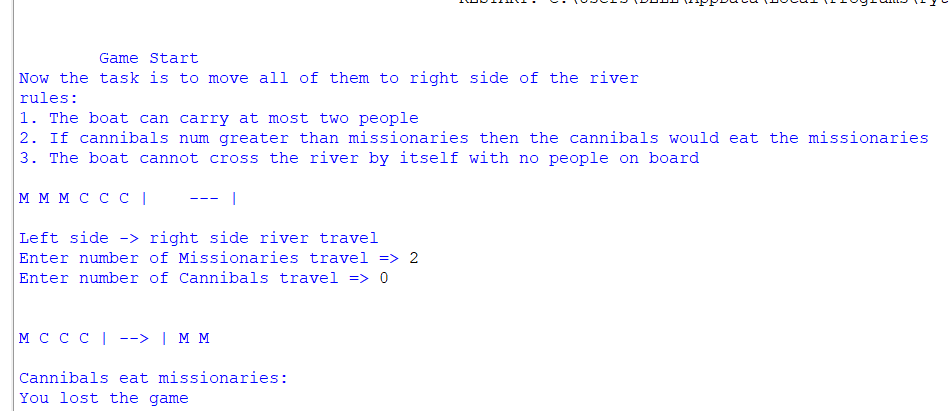
print("Cannibals eat missionaries:\nYou lost the game")

break

except EOFError as e:

print("\nInvalid input please retry !!")

OUTPUT



6.VACCUM CLEANER

class VacuumCleaner:

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

self.position = 0

self.dirt\_positions = [2, 4, 7]

def move(self, position):

self.position = position

print("Moving to position", self.position)

def clean(self, position):

if position in self.dirt\_positions:

self.dirt\_positions.remove(position)

print("Cleaning dirt at position", position)

def check\_dirt(self):

print("Dirt positions:", self.dirt\_positions)

def clean\_all\_dirt(self):

while self.dirt\_positions:

current\_position = self.dirt\_positions[0]

self.move(current\_position)

self.clean(current\_position)

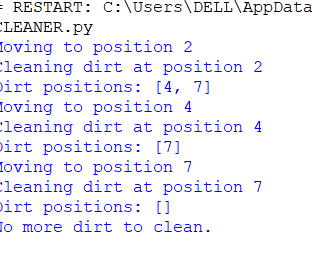
self.check\_dirt()

print("No more dirt to clean.")

vacuum = VacuumCleaner()

vacuum.clean\_all\_dirt()

OUTPUT



7.BFS

from collections import deque

def bfs(graph, start\_node):

visited = set()

queue = deque([start\_node])

print("BFS traversal order:")

while queue:

node = queue.popleft()

if node not in visited:

print(node, end=" ")

visited.add(node)

queue.extend(neighbor for neighbor in graph[node] if neighbor not in visited)

graph = {

'A': ['B', 'C'],

'B': ['A', 'D', 'E'],

'C': ['A', 'F'],

'D': ['B'],

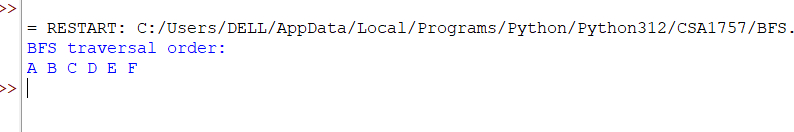
'E': ['B', 'F'],

'F': ['C', 'E']

}

bfs(graph, 'A')

OUTPUT



8.DFS

def dfs(graph, start\_node, visited=None):

if visited is None:

visited = set()

print(start\_node, end=" ")

visited.add(start\_node)

for neighbor in graph[start\_node]:

if neighbor not in visited:

dfs(graph, neighbor, visited)

graph = {

'A': ['B', 'C'],

'B': ['A', 'D', 'E'],

'C': ['A', 'F'],

'D': ['B'],

'E': ['B', 'F'],

'F': ['C', 'E']

}

print("DFS traversal order:")

dfs(graph, 'A')

OUTPUT

