def Flames**(**name1,name2**)**:

namestr = name1 + name2

for c in namestr:

if namestr.count**(**c**)** != 1:

namestr = namestr.replace**(**c,""**)**

print**(**"FLAMES...."**)**

print**(**"F = Friend \nL = Love \nA = Affection \nM = Marriage \nE = Enemy \nS = Siblings \n\n"**)**

number = len**(**namestr**)** % 6

rel = ""

if number == 1:

rel += "Friends"

elif number == 2:

rel += "Love"

elif number == 3:

rel += "Affection"

elif number == 4:

rel += "Marriage"

elif number == 5:

rel += "Enemy"

elif number == 0:

rel += "Siblings"

else:

pass

return rel

n1 = input**(**"Enter your name : "**)**.upper**()**

n2 = input**(**"Enter name of your crush : "**)**.upper**()**

print**(**f"Your Relationship is : {Flames(n1,n2)}"**)**

|  |
| --- |
|  |
| class Hangman(): |
|  | def \_\_init\_\_(self): |
|  | print "Welcome to 'Hangman', are you ready to die?" |
|  | print "(1)Yes, for I am already dead.\n(2)No, get me outta here!" |
|  | user\_choice\_1 = raw\_input("->") |
|  |  |
|  | if user\_choice\_1 == '1': |
|  | print "Loading nooses, murderers, rapists, thiefs, lunatics..." |
|  | self.start\_game() |
|  | elif user\_choice\_1 == '2': |
|  | print "Bye bye now..." |
|  | exit() |
|  | else: |
|  | print "I'm sorry, I'm hard of hearing, could you repeat that?" |
|  | self.\_\_init\_\_() |
|  |  |
|  | def start\_game(self): |
|  | print "A crowd begins to gather, they can't wait to see some real" |
|  | print "justice. There's just one thing, you aren't a real criminal." |
|  | print "No, no. You're the wrong time, wrong place type. You may think" |
|  | print "you're dead, but it's not like that at all. Yes, yes. You've" |
|  | print "got a chance to live. All you've gotta do is guess the right" |
|  | print "words and you can live to see another day. But don't get so" |
|  | print "happy yet. If you make 6 wrong guess, YOU'RE TOAST! VAMANOS!" |
|  | self.core\_game() |
|  |  |
|  | def core\_game(self): |
|  | guesses = 0 |
|  | letters\_used = "" |
|  | the\_word = "pizza" |
|  | progress = ["?", "?", "?", "?", "?"] |
|  |  |
|  | while guesses < 6: |
|  | guess = raw\_input("Guess a letter ->") |
|  |  |
|  | if guess in the\_word and not in letters\_used: |
|  | print "As it turns out, your guess was RIGHT!" |
|  | letters\_used += "," + guess |
|  | self.hangman\_graphic(guesses) |
|  | print "Progress: " + self.progress\_updater(guess, the\_word, progress) |
|  | print "Letter used: " + letters\_used |
|  | elif guess not in the\_word and not(in letters\_used): |
|  | guesses += 1 |
|  | print "Things aren't looking so good, that guess was WRONG!" |
|  | print "Oh man, that crowd is getting happy, I thought you" |
|  | print "wanted to make them mad?" |
|  | letters\_used += "," + guess |
|  | self.hangman\_graphic(guesses) |
|  | print "Progress: " + "".join(progress) |
|  | print "Letter used: " + letters\_used |
|  | else: |
|  | print "That's the wrong letter, you wanna be out here all day?" |
|  | print "Try again!" |
|  |  |
|  |  |
|  |  |
|  | def hangman\_graphic(self, guesses): |
|  | if guesses == 0: |
|  | print "\_\_\_\_\_\_\_\_ " |
|  | print "| | " |
|  | print "| " |
|  | print "| " |
|  | print "| " |
|  | print "| " |
|  | elif guesses == 1: |
|  | print "\_\_\_\_\_\_\_\_ " |
|  | print "| | " |
|  | print "| 0 " |
|  | print "| " |
|  | print "| " |
|  | print "| " |
|  | elif guesses == 2: |
|  | print "\_\_\_\_\_\_\_\_ " |
|  | print "| | " |
|  | print "| 0 " |
|  | print "| / " |
|  | print "| " |
|  | print "| " |
|  | elif guesses == 3: |
|  | print "\_\_\_\_\_\_\_\_ " |
|  | print "| | " |
|  | print "| 0 " |
|  | print "| /| " |
|  | print "| " |
|  | print "| " |
|  | elif guesses == 4: |
|  | print "\_\_\_\_\_\_\_\_ " |
|  | print "| | " |
|  | print "| 0 " |
|  | print "| /|\ " |
|  | print "| " |
|  | print "| " |
|  | elif guesses == 5: |
|  | print "\_\_\_\_\_\_\_\_ " |
|  | print "| | " |
|  | print "| 0 " |
|  | print "| /|\ " |
|  | print "| / " |
|  | print "| " |
|  | else: |
|  | print "\_\_\_\_\_\_\_\_ " |
|  | print "| | " |
|  | print "| 0 " |
|  | print "| /|\ " |
|  | print "| / \ " |
|  | print "| " |
|  | print "The noose tightens around your neck, and you feel the" |
|  | print "sudden urge to urinate." |
|  | print "GAME OVER!" |
|  | self.\_\_init\_\_() |
|  |  |
|  | def progress\_updater(self, guess, the\_word, progress): |
|  | i = 0 |
|  | while i < len(the\_word): |
|  | if guess == the\_word[i]: |
|  | progress[i] = guess |
|  | i += 1 |
|  | else: |
|  | i += 1 |
|  |  |
|  | return "".join(progress) |
|  |  |
|  | game = Hangman() |
|  |  |

**def** generateSquare(n):

    magicSquare **=** [[0 **for** x **in** range(n)]

**for** y **in** range(n)]

     i **=** n **//** 2

    j **=** n **-** 1

     num **=** 1

**while** num <**=** (n **\*** n):

**if** i **==** **-**1 **and** j **==** n:

            j **=** n **-** 2

            i **=** 0

**else**:

**if** i < 0:

                i **=** n **-** 1

**if** j **==** n:

                j **=**

**if** magicSquare[int(i)][int(j)]:

            j **=** j **-** 2

            i **=** i **+** 1

**continue**

**else**:

            magicSquare[int(i)][int(j)] **=** num

            num **=** num **+** 1

        j **=** j **+** 1

        i **=** i **-** 1

**print**("Magic Square for n =", n)

**print**("Sum of each row or column",

          n **\*** (n **\*** n **+** 1) **//** 2, "\n")

**for** i **in** range(0, n):

**for** j **in** range(0, n):

**print**('%2d ' **%** (magicSquare[i][j]),

                  end**=**'')

            # To display output

            # in matrix form

**if** j **==** n **-** 1:

                print()

# Driver Code

# Works only when n is odd

n **=** 7

generateSquare(n)

**class** QueueEntry(object):

**def** \_\_init\_\_(self, v**=**0, dist**=**0):

        self.v **=** v

        self.dist **=** dist

'''This function returns minimum number of

dice throws required to. Reach last cell

from 0'th cell in a snake and ladder game.

move[] is an array of size N where N is

no. of cells on board. If there is no

snake or ladder from cell i, then move[i]

is -1. Otherwise move[i] contains cell to

which snake or ladder at i takes to.'''

**def** getMinDiceThrows(move, N):

    # The graph has N vertices. Mark all

    # the vertices as not visited

    visited **=** [False] **\*** N

    # Create a queue for BFS

    queue **=** []

    # Mark the node 0 as visited and enqueue it

    visited[0] **=** True

    # Distance of 0't vertex is also 0

    # Enqueue 0'th vertex

    queue.append(QueueEntry(0, 0))

    # Do a BFS starting from vertex at index 0

    qe **=** QueueEntry()  # A queue entry (qe)

**while** queue:

        qe **=** queue.pop(0)

        v **=** qe.v  # Vertex no. of queue entry

        # If front vertex is the destination

        # vertex, we are done

**if** v **==** N **-** 1:

**break**

        # Otherwise dequeue the front vertex

        # and enqueue its adjacent vertices

        # (or cell numbers reachable through

        # a dice throw)

        j **=** v **+** 1

**while** j <**=** v **+** 6 **and** j < N:

            # If this cell is already visited,

            # then ignore

**if** visited[j] **is** False:

                # Otherwise calculate its

                # distance and mark it

                # as visited

                a **=** QueueEntry()

                a.dist **=** qe.dist **+** 1

                visited[j] **=** True

                a.v **=** move[j] **if** move[j] !**=** **-**1 **else** j

                queue.append(a)

            j **+=** 1

**return** qe.dist

# driver code

N **=** 30

moves **=** [**-**1] **\*** N

# Ladders

moves[2] **=** 7

moves[4] **=** 25

moves[10] **=** 21

moves[19] **=** 28

# Snakes

moves[26] **=** 0

moves[20] **=** 8

moves[16] **=** 3

moves[18] **=** 6

**print**("Min Dice throws required is {0}".

      format(getMinDiceThrows(moves, N)))