Index

Program 1

Aim: Introduction to Python Programming language

What is Python?

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

It is used for:

web development (server-side),

software development,

mathematics,

system scripting.

What can Python do?

Python can be used on a server to create web applications.

Python can be used alongside software to create workflows.

Python can connect to database systems. It can also read and modify files.

Python can be used to handle big data and perform complex mathematics.

Python can be used for rapid prototyping, or for production-ready software development.

Why Python?

Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).

Python has a simple syntax similar to the English language.

Python has syntax that allows developers to write programs with fewer lines than some other programming languages.

Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.

Python can be treated in a procedural way, an object-oriented way or a functional way.

Good to know

The most recent major version of Python is Python 3, which we shall be using in this tutorial. However, Python 2, although not being updated with anything other than security updates, is still quite popular.

In this tutorial Python will be written in a text editor. It is possible to write Python in an Integrated Development Environment, such as Thonny, Pycharm, Netbeans or Eclipse which are particularly useful when managing larger collections of Python files.

Python Syntax compared to other programming languages

Python was designed for readability, and has some similarities to the English language with influence from mathematics.

Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.

Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

Program 2

Aim: Implementation of vectors and matrices operations

Use NumPy to create a one-dimensional array:

```
# Load library
import numpy as np

# Create a vector as a row
vector_row = np.array([1, 2, 3])

# Create a vector as a column
vector_column = np.array([[1],
                          [2],
                          [3]])
```

Use NumPy to create a two-dimensional array:

```
# Load library
import numpy as np

# Create a matrix
matrix = np.array([[1, 2],
                   [1, 2],
                   [1, 2]])
```

Use the T method:

```
# Load library
import numpy as np

# Create matrix
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Transpose matrix
matrix.T
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
```

Use flatten:

```
# Load library
import numpy as np

# Create matrix
matrix = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])

# Flatten matrix
matrix.flatten()
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Program 3

Aim: WAP to implement DFS using Python

Theory:

Depth First Traversal (or DFS) for a graph is similar to Depth First Traversal of a tree. The only catch here is, that, unlike trees, graphs may contain cycles (a node may be visited twice). To avoid processing a node more than once, use a boolean visited array. A graph can have more than one DFS traversal.

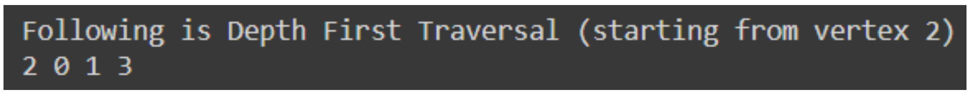
Code:

```
# Python3 program to print DFS traversal
# from a given graph
from collections import defaultdict



# This class represents a directed graph using
# adjacency list representation
class Graph:

    # Constructor
    def __init__(self):

        # Default dictionary to store graph
        self.graph = defaultdict(list)



    # Function to add an edge to graph
    def addEdge(self, u, v):
        self.graph[u].append(v)
```

```
# A function used by DFS
def DFSUtil(self, v, visited):

        # Mark the current node as visited
        # and print it
        visited.add(v)
        print(v, end=' ')

        # Recur for all the vertices
        # adjacent to this vertex
        for neighbour in self.graph[v]:
                if neighbour not in visited:
                        self.DFSUtil(neighbour, visited)


# The function to do DFS traversal. It uses
# recursive DFSUtil()
def DFS(self, v):

        # Create a set to store visited vertices
        visited = set()

        # Call the recursive helper function
        # to print DFS traversal
        self.DFSUtil(v, visited)
```

```
# Driver's code
if __name__ == "__main__":
        g = Graph()
        g.addEdge(0, 1)
        g.addEdge(0, 2)
        g.addEdge(1, 2)
        g.addEdge(2, 0)
        g.addEdge(2, 3)
        g.addEdge(3, 3)

        print("Following is Depth First Traversal (starting from vertex 2)")

        # Function call
        g.DFS(2)
```

Output:



```
Following is Depth First Traversal (starting from vertex 2)
2 0 1 3
```

Program 4

Aim: WAP to implement BFS for 8 puzzle problem using Python

Theory: We can search the state space tree using a **breadth-first approach**. It always locates the **goal state that is closest to the root**. However, the algorithm tries the **same series of movements as DFS** regardless of the initial state.

Code:

```
struct list_node
{
  list_node *next;
  // Helps in tracing path when answer is found
  list_node *parent;
  float cost;
}
algorithm LCSearch(list_node *t)
{
  // Search t for an answer node
  // Input: Root node of tree t
  // Output: Path from answer node to root
  if (*t is an answer node)
  {
     print(*t);
     return;
  }

  E = t; // E-node
  Initialize the list of live nodes to be empty;
  while (true)
  {
    for each child x of E
    {
       if x is an answer node
       {
         print the path from x to t;
         return;
       }
       Add (x); // Add x to list of live nodes;
       x->parent = E; // Pointer for path to root
    }
    if there are no more live nodes
    {
      print ("No answer node");
      return;
    }

    // Find a live node with least estimated cost
    E = Least();
    // The found node is deleted from the list of
    // live nodes
  }
}
```

Output:

```
1  2  3
5  6  0
7  8  4

1  2  3
5  0  6
7  8  4

1  2  3
5  8  6
7  0  4

1  2  3
5  8  6
0  7  4
```

Program 5

Aim: WAP to implement BFS for water jug problem using Python

Theory:

You are given a m liter jug and a n liter jug where 0<m<n . Both the jugs are initially empty. The jugs don't have markings to allow measuring smaller quantities. You have to use the jugs to measure d liters of water where d<n . Determine the minimum no of operations to be performed to obtain d liters of water in one of jug.

Code:

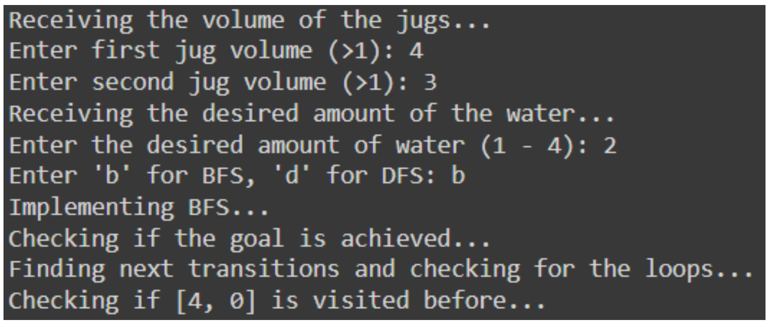
```
import collections
#This method return a key value for a given node.
#Node is a list of two integers representing current state of the jugs
def get_index(node):
   return pow(7, node[0]) * pow(5, node[1])
get_index([4,0])
#This method accepts an input for asking the choice for type of searching required i.e. BFS or DFS.
#Method return True for BFS, False otherwise
def get_search_type():
   s = input("Enter 'b' for BFS, 'd' for DFS: ")
   #TODO:convert the input into lowercase using lower() method
   s = s.lower()
   while s != 'b' and s != 'd':
     s = input("The input is not valid! Enter 'b' for BFS, 'd' for DFS: ")
     s = s[0].lower()
   #TODO: Return True for BFS option selected
   return s == 'b'
#This method accept volumes of the jugs as an input from the user.
#Returns a list of two integeres representing volumes of the jugs.
```

```
def get_jugs():
    print("Receiving the volume of the jugs...")
    #TODO: Create an empty list
    jugs = []

    temp = int(input("Enter first jug volume (>1): "))
    while temp < 1:
        temp = int(input("Enter a valid amount (>1): "))
    #TODO: Append the input quantity of jug into jugs list
    jugs.append(temp)

    temp = int(input("Enter second jug volume (>1): "))
    while temp < 1:
        temp = int(input("Enter a valid amount (>1): "))

    #TODO: Append the input quantity of jug into jugs list
    jugs.append(temp)

    #TODO: Return the list
    return jugs
#This method accepts the desired amount of water as an input from the user whereas
#the parameter jugs is a list of two integers representing volumes of the jugs
#Returns the desired amount of water as goal
def get_goal(jugs):

    print("Receiving the desired amount of the water...")

    #TODO: Find the maximum capacity of jugs using max()
    max_amount = max(jugs)
    s = "Enter the desired amount of water (1 - {0}): ".format(max_amount)
    goal_amount = int(input(s))
    #TODO: Accept the input again from the user if the bound of goal_amount is outside the limit between [1,max_amount]
    while goal_amount < 1 or goal_amount > max_amount:
        goal_amount = int(input("Enter a valid amount (1 - {0}): ".format(max_amount)))
    #TODO:Return the goal amount of water
    return goal_amount
#This method checks whether the given path matches the goal node.
#The path parameter is a list of nodes representing the path to be checked
#The goal_amount parameter is an integer representing the desired amount of water
def is_goal(path, goal_amount):

    print("Checking if the goal is achieved...")

    #TODO: Return the status of the latest path matches with the goal_amount of another jug
    return path[-1][0] == goal_amount
#This method validates whether the given node is already visited.
#The parameter node is a list of two integers representing current state of the jugs
#The parameter check_dict is   a dictionary storing visited nodes
def been_there(node, check_dict):

    print("Checking if {0} is visited before...".format(node))

    #TODO: Return True whether a given node already exisiting in a dictionary, otherwise False. Use get() method of dictionary
    return check_dict.get(node)
#This method returns the list of all possible transitions
#The parameter jugs is a list of two integers representing volumes of the jugs
#The parameter path is a list of nodes represeting the current path
#The parameter check_dict is a dictionary storing visited nodes
def next_transitions(jugs, path, check_dict):

    print("Finding next transitions and checking for the loops...")
    #TODO: create an empty list
    result = []
    next_nodes = []
    node = []
    a_max = jugs[0]
    b_max = jugs[1]

    #TODO: initial amount in the first jug using path parameter
    a = path[-1][0]
    #TODO: initial amount in the second jug using path parameter
    b = path[-1][1]

    #Operation Used in Water Jug problem
    # 1. fill in the first jug
    node.append(a_max)
    node.append(b)
    if not been_there(node, check_dict):
        next_nodes.append(node)
    node = []

    # 2. fill in the second jug
    #TODO: Append with the initial amount of water in first jug
    node.append(a)
    #TODO: Append with the max amount of water in second jug
    node.append(b-max)
    #TODO: Check if node is not visited then append the node in next_nodes. Use
been_there(..,..) method
    if not been_there(node, check_dict):
        next_nodes.append(node)
    node = []
```

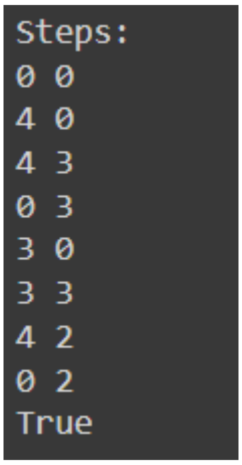
```
# 3. second jug to first jug
node.append(min(a_max, a + b))
node.append(b - (node[0] - a))  # b - ( a' - a)
if not been_there(node, check_dict):
   next_nodes.append(node)
node = []

# 4. first jug to second jug
#TODO: Append the minimum between the a+b and b_max
node.append(min(a + b, b_max))
node.insert(0, a - (node[0] - b))
if not been_there(node, check_dict):
   next_nodes.append(node)
node = []

# 5. empty first jug
#TODO: Append 0 to empty first jug
node.append(0)
#TODO:Append b amount for second jug
node.append(b)
if not been_there(node, check_dict):
   next_nodes.append(node)
node = []

# 6. empty second jug
#TODO:Append a amount for first jug
node.append(a)
#TODO: Append 0 to empty second jug
node.append(0)
if not been_there(node, check_dict):
        next_nodes.append(node)

    # create a list of next paths
    for i in range(0, len(next_nodes)):
        temp = list(path)
        #TODO: Append the ith index of next_nodes to temp
        temp.append(next_nodes[i])
        result.append(temp)

    if len(next_nodes) == 0:
        print("No more unvisited nodes...\nBacktracking...")
    else:
        print("Possible transitions: ")
        for nnode in next_nodes:
            print(nnode)
    #TODO: return result
    return result

def transition(old, new, jugs):

    #TODO: Get the amount of water from old state/node for first Jug
    a = old[0]
    #TODO: Get the amount of water from old state/node for second Jug
    b = old[1]
    #TODO: Get the amount of water from new state/node for first Jug
    a_prime = new[0]
    #TODO: Get the amount of water from new state/node for second Jug
    b_prime = new[1]
    #TODO: Get the amount of water from jugs representing volume for first Jug
    a_max = jugs[0]
    #TODO: Get the amount of water from jugs representing volume for second Jug
    b_max = jugs[1]

    if a > a_prime:
        if b == b_prime:
            return "Clear {0}-liter jug:\t\t\t".format(a_max)
        else:
            return "Pour {0}-liter jug into {1}-liter jug:\t".format(a_max, b_max)
    else:
        if b > b_prime:
            if a == a_prime:
                return "Clear {0}-liter jug:\t\t\t".format(b_max)
            else:
                return "Pour {0}-liter jug into {1}-liter jug:\t".format(b_max, a_max)
        else:
            if a == a_prime:
                return "Fill {0}-liter jug:\t\t\t".format(b_max)
            else:
                return "Fill {0}-liter jug:\t\t\t".format(a_max)

def print_path(path, jugs):

    print("Starting from:\t\t\t\t", path[0])
    for i in  range(0, len(path) - 1):
        print(i+1,":", transition(path[i], path[i+1], jugs), path[i+1])
def search(starting_node, jugs, goal_amount, check_dict, is_breadth):

    if is_breadth:
        print("Implementing BFS...")
    else:
        print("Implementing DFS...")
```

```
goal = []
#TODO: SET accomplished to be False
accomplished = False

#TODO: Call a deque() using collections
q = collections.deque()
q.appendleft(starting_node)

while len(q) != 0:
    path = q.popleft()
    check_dict[get_index(path[-1])] = True
    if len(path) >= 2:
        print(transition(path[-2], path[-1], jugs), path[-1])
    if is_goal(path, goal_amount):
        #TODO: Set accomplished to be True
        accomplished = True
        goal = path
        break

    #TODO: Call next_transitions method for generating the further nodes
    next_moves = next_transitions(jugs, path, check_dict)
    #TODO: Iterate over the next_moves list
    for i in next_moves:
        if is_breadth:
            q.append(i)
        else:
            q.appendleft(i)

if accomplished:
    print("The goal is achieved\nPrinting the sequence of the moves...\n")
    print_path(goal, jugs)
    else:
        print("Problem cannot be solved.")
if __name__ == '__main__':
    starting_node = [[0, 0]]
    #TODO: Call the get_jugs() method
    jugs = get_jugs()
    #TODO: Call the get_goal() method
    goal_amount = get_goal(jugs)
    #TODO: Create an empty dictionary
    check_dict = {}
    #TODO: call the get_search_type() method
    is_breadth = get_search_type()
    #TODO: Call the search method with the required parameters
    search(starting_node, jugs, goal_amount, check_dict, is_breadth)
```

Output:



```
Receiving the volume of the jugs...
Enter first jug volume (>1): 4
Enter second jug volume (>1): 3
Receiving the desired amount of the water...
Enter the desired amount of water (1 - 4): 2
Enter 'b' for BFS, 'd' for DFS: b
Implementing BFS...
Checking if the goal is achieved...
Finding next transitions and checking for the loops...
Checking if [4, 0] is visited before...
```



```
Steps:
0 0
4 0
4 3
0 3
3 0
3 3
4 2
0 2
True
```

Program 6

Aim: WAP to implement A* algorithm in python

Theory:

A* search is the most commonly known form of best-first search. It uses heuristic function h(n), and cost to reach the node n from the start state g(n). It has combined features of UCS and greedy best-first search, by which it solve the problem efficiently. A* search algorithm finds the shortest path through the search space using the heuristic function. This search algorithm expands less search tree and provides optimal result faster. A* algorithm is similar to UCS except that it uses g(n)+h(n) instead of g(n).

Code:

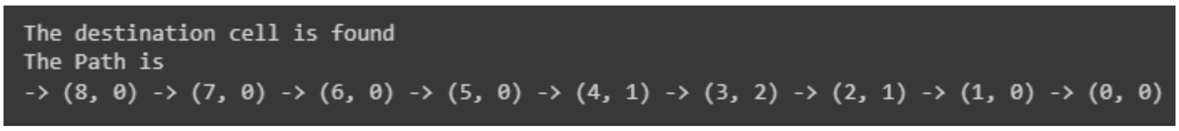
```
import math

import heapq


# Define the Cell class

class Cell:

        def __init__(self):

                self.parent_i = 0 # Parent cell's row index

                self.parent_j = 0 # Parent cell's column index

                self.f = float('inf') # Total cost of the cell (g + h)

                self.g = float('inf') # Cost from start to this cell

                self.h = 0 # Heuristic cost from this cell to destination


# Define the size of the grid

ROW = 9

COL = 10


# Check if a cell is valid (within the grid)

def is_valid(row, col):
    return (row >= 0) and (row < ROW) and (col >= 0) and (col < COL)

# Check if a cell is unblocked
def is_unblocked(grid, row, col):
    return grid[row][col] == 1

# Check if a cell is the destination
def is_destination(row, col, dest):
    return row == dest[0] and col == dest[1]

# Calculate the heuristic value of a cell (Euclidean distance to destination)
def calculate_h_value(row, col, dest):
    return ((row - dest[0]) ** 2 + (col - dest[1]) ** 2) ** 0.5

# Trace the path from source to destination
def trace_path(cell_details, dest):
    print("The Path is ")
    path = []
    row = dest[0]
    col = dest[1]

    # Trace the path from destination to source using parent cells
    while not (cell_details[row][col].parent_i == row and cell_details[row][col].parent_j ==
col):
        path.append((row, col))
        temp_row = cell_details[row][col].parent_i
        temp_col = cell_details[row][col].parent_j
        row = temp_row
        col = temp_col

    # Add the source cell to the path
    path.append((row, col))
```

```
        # Reverse the path to get the path from source to destination
        path.reverse()

        # Print the path
        for i in path:
                print("->", i, end=" ")
        print()

# Implement the A* search algorithm
def a_star_search(grid, src, dest):
        # Check if the source and destination are valid
        if not is_valid(src[0], src[1]) or not is_valid(dest[0], dest[1]):
                print("Source or destination is invalid")
                return

        # Check if the source and destination are unblocked
        if not is_unblocked(grid, src[0], src[1]) or not is_unblocked(grid, dest[0], dest[1]):
                print("Source or the destination is blocked")
                return

        # Check if we are already at the destination
        if is_destination(src[0], src[1], dest):
                print("We are already at the destination")
                return

        # Initialize the closed list (visited cells)
        closed_list = [[False for _ in range(COL)] for _ in range(ROW)]
        # Initialize the details of each cell
        cell_details = [[Cell() for _ in range(COL)] for _ in range(ROW)]

        # Initialize the start cell details
        i = src[0]
```

```
    j = src[1]
    cell_details[i][j].f = 0
    cell_details[i][j].g = 0
    cell_details[i][j].h = 0
    cell_details[i][j].parent_i = i
    cell_details[i][j].parent_j = j

    # Initialize the open list (cells to be visited) with the start cell
    open_list = []
    heapq.heappush(open_list, (0.0, i, j))

    # Initialize the flag for whether destination is found
    found_dest = False

    # Main loop of A* search algorithm
    while len(open_list) > 0:
        # Pop the cell with the smallest f value from the open list
        p = heapq.heappop(open_list)

        # Mark the cell as visited
        i = p[1]
        j = p[2]
        closed_list[i][j] = True

        # For each direction, check the successors
        directions = [(0, 1), (0, -1), (1, 0), (-1, 0), (1, 1), (1, -1), (-1, 1),
(-1, -1)]
        for dir in directions:
            new_i = i + dir[0]
            new_j = j + dir[1]

            # If the successor is valid, unblocked, and not visited
```

```
                if is_valid(new_i, new_j) and is_unblocked(grid, new_i, new_j) and not
closed_list[new_i][new_j]:
                    # If the successor is the destination
                    if is_destination(new_i, new_j, dest):
                        # Set the parent of the destination cell
                        cell_details[new_i][new_j].parent_i = i
                        cell_details[new_i][new_j].parent_j = j
                        print("The destination cell is found")
                        # Trace and print the path from source to destination
                        trace_path(cell_details, dest)
                        found_dest = True
                        return
                    else:
                        # Calculate the new f, g, and h values
                        g_new = cell_details[i][j].g + 1.0
                        h_new = calculate_h_value(new_i, new_j, dest)
                        f_new = g_new + h_new

                        # If the cell is not in the open list or the new f value is
smaller
                        if cell_details[new_i][new_j].f == float('inf') or
cell_details[new_i][new_j].f > f_new:
                            # Add the cell to the open list
                            heapq.heappush(open_list, (f_new, new_i, new_j))
                            # Update the cell details
                            cell_details[new_i][new_j].f = f_new
                            cell_details[new_i][new_j].g = g_new
                            cell_details[new_i][new_j].h = h_new
                            cell_details[new_i][new_j].parent_i = i
                            cell_details[new_i][new_j].parent_j = j

    # If the destination is not found after visiting all cells
    if not found_dest:
        print("Failed to find the destination cell")


def main():
    # Define the grid (1 for unblocked, 0 for blocked)
    grid = [
        [1, 0, 1, 1, 1, 1, 0, 1, 1, 1],
        [1, 1, 1, 0, 1, 1, 1, 0, 1, 1],
        [1, 1, 1, 0, 1, 1, 0, 1, 0, 1],
        [0, 0, 1, 0, 1, 0, 0, 0, 0, 1],
        [1, 1, 1, 0, 1, 1, 1, 0, 1, 0],
        [1, 0, 1, 1, 1, 1, 0, 1, 0, 0],
        [1, 0, 0, 0, 0, 1, 0, 0, 0, 1],
        [1, 0, 1, 1, 1, 1, 0, 1, 1, 1],
        [1, 1, 1, 0, 0, 0, 1, 0, 0, 1]
    ]

    # Define the source and destination
    src = [8, 0]
    dest = [0, 0]

    # Run the A* search algorithm
    a_star_search(grid, src, dest)

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

Output:



```
The destination cell is found
The Path is
-> (8, 0) -> (7, 0) -> (6, 0) -> (5, 0) -> (4, 1) -> (3, 2) -> (2, 1) -> (1, 0) -> (0, 0)
```

Program 7

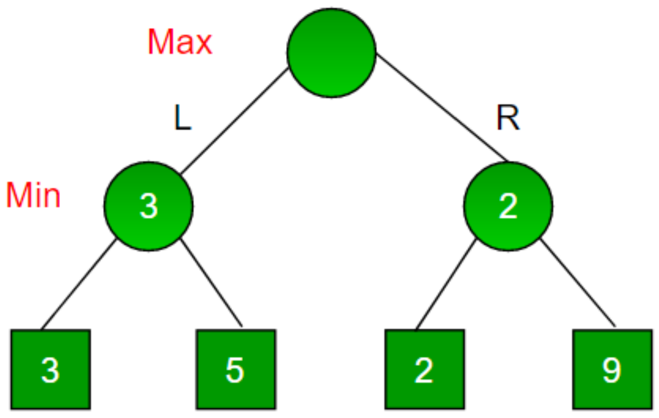
Aim: WAP to implement Tic-Tac-Toe game problem.

Theory:

Minimax is a artifical intelligence applied in two player games, such as tic-tac-toe, checkers, chess and go. This games are known as zero-sum games, because in a mathematical representation: one player wins (+1) and other player loses (-1) or both of anyone not to win (0).

Minimax is a type of adversarial search algorithm for generating and exploring game trees. It is mostly used to solve zero-sum games where one side's gain is equivalent to other side's loss, so adding all gains and subtracting all losses end up being zero.

Adversarial search differs from conventional searching algorithms by adding opponents into the mix. Minimax algorithm keeps playing the turns of both player and the opponent optimally to figure out the best possible move.



The above tree shows two possible scores when maximizer makes left and right moves.

Code:

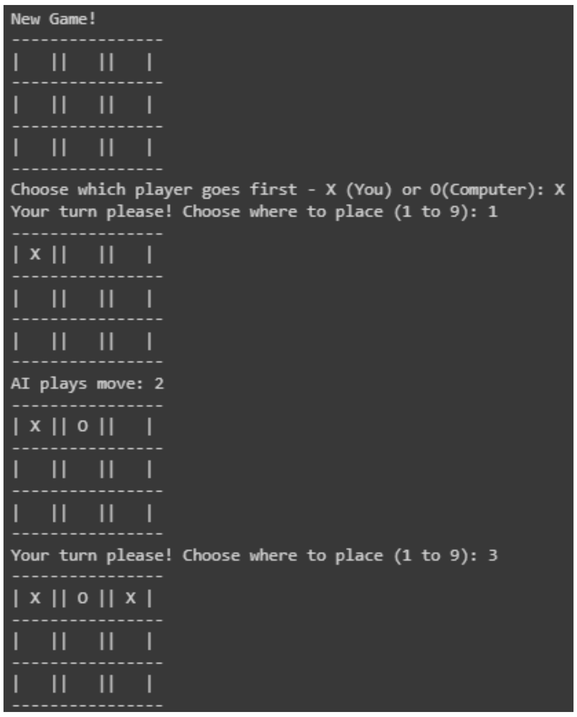
```
#Import the necessary libraries
import numpy as np
from math import inf as infinity

#Set the Empty Board
game_state = [[' ',' ',' '],
             [' ',' ',' '],
             [' ',' ',' ']]
#Create the Two Players as 'X'/'O'
players = ['X','O']

#Method for checking the correct move on Tic-Tac-Toe
def play_move(state, player, block_num):
    if state[int((block_num-1)/3)][(block_num-1)%3] == ' ':
        #TODO: Assign the player move on the current position of Tic-Tac-Toe if condition is
True
        state[int((block_num-1)/3)][(block_num-1)%3] = player
    else:
        block_num = int(input("Block is not empty, ya blockhead! Choose again: "))
        #TODO: Recursively call the play_move
        play_move(state, player, block_num)

#Method to copy the current game state to new_state of Tic-Tac-Toe
def copy_game_state(state):
    new_state = [[' ',' ',' '],[' ',' ',' '],[' ',' ',' ']]
    for i in range(3):
        for j in range(3):
            #TODO: Copy the Tic-Tac-Toe state to new_state
                new_state[i][j] = state[i][j]
    #TODO: Return the new_state
    return new_state
```

```
#Method to check the current state of the Tic-Tac-Toe
def check_current_state(game_state):
    #TODO: Set the draw_flag to 0
    draw_flag = 0
    for i in range(3):
        for j in range(3):
            if game_state[i][j] == ' ':
                draw_flag = 1

    if draw_flag == 0:
        return None, "Draw"

    # Check horizontals in first row
    if (game_state[0][0] == game_state[0][1] and game_state[0][1] == game_state[0][2] and
game_state[0][0] != ' '):
        return game_state[0][0], "Done"
    #TODO: Check horizontals in second row
    if (game_state[1][0] == game_state[1][1] and game_state[1][1] == game_state[1][2] and
game_state[1][0] != ' '):
        return game_state[1][0], "Done"
    #TODO: Check horizontals in third row
    if (game_state[2][0] == game_state[2][1] and game_state[2][1] == game_state[2][2] and
game_state[2][0] != ' '):
        return game_state[2][0], "Done"

    # Check verticals in first column
    if (game_state[0][0] == game_state[1][0] and game_state[1][0] == game_state[2][0] and
game_state[0][0] != ' '):
        return game_state[0][0], "Done"
    # Check verticals in second column
    if (game_state[0][1] == game_state[1][1] and game_state[1][1] == game_state[2][1] and
game_state[0][1] != ' '):
        return game_state[0][1], "Done"
    # Check verticals in third column
    if (game_state[0][2] == game_state[1][2] and game_state[1][2] == game_state[2][2] and
game_state[0][2] != ' '):
        return game_state[0][2], "Done"


    # Check left diagonal
    if (game_state[0][0] == game_state[1][1] and game_state[1][1] == game_state[2][2] and
game_state[1][1] != ' '):
        return game_state[1][1], "Done"
    # Check right diagonal
    if (game_state[0][2] == game_state[1][1] and game_state[1][1] == game_state[2][0] and
game_state[1][1] != ' '):
        return game_state[1][1], "Done"


    return None, "Not Done"


#Method to print the Tic-Tac-Toe Board
def print_board(game_state):
    print('----------------')
    print('| ' + str(game_state[0][0]) + ' || ' + str(game_state[0][1]) + ' || ' + str(game_state[0][2]) + '
|')
    print('----------------')
    print('| ' + str(game_state[1][0]) + ' || ' + str(game_state[1][1]) + ' || ' + str(game_state[1][2]) + '
|')
    print('----------------')
    print('| ' + str(game_state[2][0]) + ' || ' + str(game_state[2][1]) + ' || ' + str(game_state[2][2]) + '
|')
    print('----------------')


#Method for implement the Minimax Algorithm
def getBestMove(state, player):
    #TODO: call the check_current_state method using state parameter
    winner_loser , done = check_current_state(state)
```

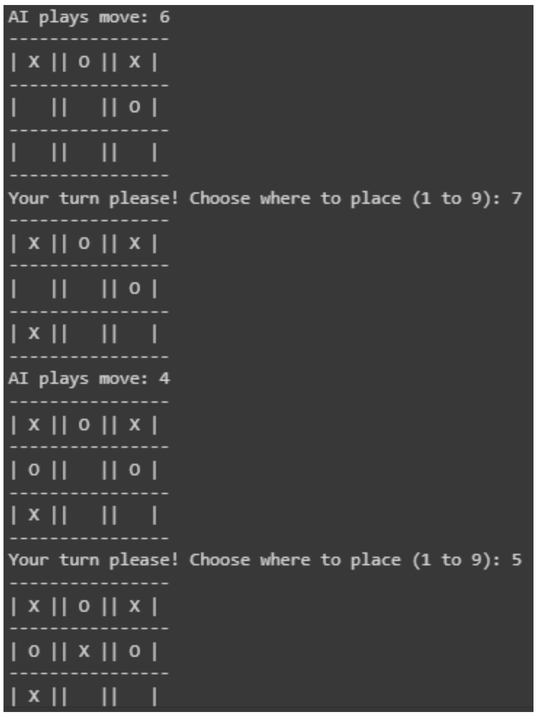
```
#TODO:Check condition for winner, if winner_loser is 'O' then Computer won
#else if winner_loser is 'X' then You won else game is draw
if done == "Done" and winner_loser == '0':
    return 1
elif done == "Done" and winner_loser == 'X':
    return -1
elif done == "Draw":
    return 0

#TODO: set moves to empty list
moves = []
#TODO: set empty_cells to empty list
empty_cells = []

#Append the block_num to the empty_cells list
for i in range(3):
    for j in range(3):
        if state[i][j] == ' ':
            empty_cells.append(i*3 + (j+1))

#TODO:Iterate over all the empty_cells
for empty_cell in empty_cells:
    #TODO: create the empty dictionary
    move = {}

    #TODO: Assign the empty_cell to move['index']
    move['index'] = empty_cell

    #Call the copy_game_state method
    new_state = copy_game_state(state)

    #TODO: Call the play_move method with new_state,player,empty_cell
```

```
        play_move(new_state, player, empty_cell)


        #if player is computer
        if player == 'O':
            #TODO: Call getBestMove method with new_state and human player ('X') to make
more depth tree for human
            result = getBestMove(new_state, 'X')
            move['score'] = result
        else:
            #TODO: Call getBestMove method with new_state and computer player('O') to make
more depth tree for computer
            result = getBestMove(new_state, 'O')
            move['score'] = result


        moves.append(move)


    # Find best move
    best_move = None
    #Check if player is computer('O')
    if player == "O":
        #TODO: Set best as -infinity for computer
        best = float("-inf")
        for move in moves:
            #TODO: Check if move['score'] is greater than best
            if move['score'] > best:
                best = move['score']
                best_move = move['index']
    else:
        #TODO: Set best as infinity for human
        best = float("inf")
        for move in moves:
            #TODO: Check if move['score'] is less than best
            if move['score'] < best:
                best = move['score']
                best_move = move['index']

    return best_move

# Now PLaying the Tic-Tac-Toe Game
play_again = 'Y'
while play_again == 'Y' or play_again == 'y':
    #Set the empty board for Tic-Tac-Toe
    game_state = [[' ',' ',' '],
              [' ',' ',' '],
              [' ',' ',' ']]
    #Set current_state as "Not Done"
    current_state = "Not Done"
    print("\nNew Game!")

    #print the game_state
    print_board(game_state)

    #Select the player_choice to start the game
    player_choice = input("Choose which player goes first - X (You) or O(Computer): ")

    #Set winner as None
    winner = None

    #if player_choice is ('X' or 'x') for humans else for computer
    if player_choice == 'X' or player_choice == 'x':
        #TODO: Set current_player_idx is 0
        current_player_idx = 0
    else:
        #TODO: Set current_player_idx is 1
        current_player_idx = 1


    while current_state == "Not Done":
        #For Human Turn
        if current_player_idx == 0:
            block_choice = int(input("Your turn please! Choose where to place (1 to 9): "))
            #TODO: Call the play_move with parameters as game_state
,players[current_player_idx], block_choice
            play_move(game_state, players[current_player_idx], block_choice)
        else:   # Computer turn
            block_choice = getBestMove(game_state, players[current_player_idx])
            #TODO: Call the play_move with parameters as game_state
,players[current_player_idx], block_choice
            play_move(game_state, players[current_player_idx], block_choice)
            print("AI plays move: " + str(block_choice))
        print_board(game_state)
        #TODO: Call the check_current_state function for game_state
        winner, current_state = check_current_state(game_state)
        if winner is not None:
            print(str(winner) + " won!")
        else:
            current_player_idx = (current_player_idx + 1)%2


        if current_state is "Draw":
            print("Draw!")


    play_again = input('Wanna try again?(Y/N) : ')
    if play_again == 'N':
        print('Thank you for playing Tic-Tac-Toe Game!!!!!!!')
```

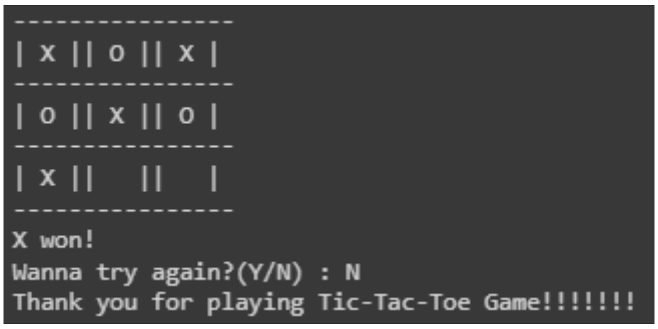
Output:



```
New Game!
----------------
|   ||   ||   |
----------------
|   ||   ||   |
----------------
|   ||   ||   |
----------------
Choose which player goes first - X (You) or O(Computer): X
Your turn please! Choose where to place (1 to 9): 1
----------------
| X ||   ||   |
----------------
|   ||   ||   |
----------------
|   ||   ||   |
----------------
AI plays move: 2
----------------
| X || O ||   |
----------------
|   ||   ||   |
----------------
|   ||   ||   |
----------------
Your turn please! Choose where to place (1 to 9): 3
----------------
| X || O || X |
----------------
|   ||   ||   |
----------------
|   ||   ||   |
----------------
```



```
AI plays move: 6
----------------
| X || O || X |
----------------
|   ||   || O |
----------------
|   ||   ||   |
----------------
Your turn please! Choose where to place (1 to 9): 7
----------------
| X || O || X |
----------------
|   ||   || O |
----------------
| X ||   ||   |
----------------
AI plays move: 4
----------------
| X || O || X |
----------------
| O ||   || O |
----------------
| X ||   ||   |
----------------
Your turn please! Choose where to place (1 to 9): 5
----------------
| X || O || X |
----------------
| O || X || O |
----------------
| X ||   ||   |
```



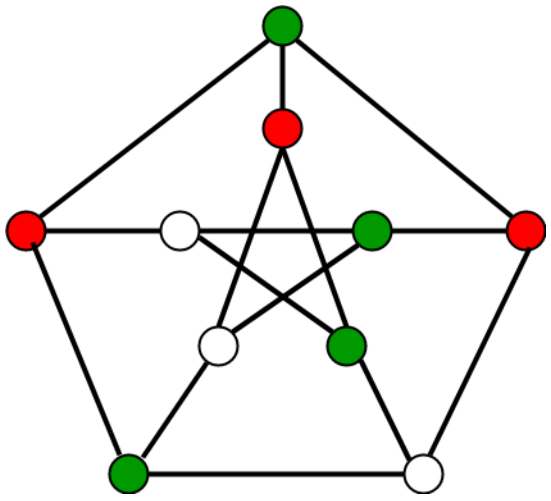
```
----------------
| X || O || X |
----------------
| O || X || O |
----------------
| X ||   ||   |
----------------
X won!
Wanna try again?(Y/N) : N
Thank you for playing Tic-Tac-Toe Game!!!!!!!
```

Program 8

Aim: Implement Graph coloring problem using python

Theory:

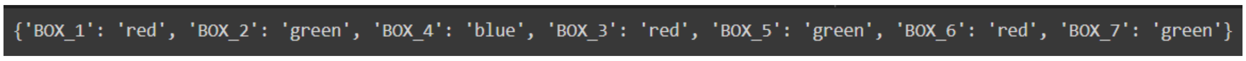
Graph coloring refers to the problem of coloring vertices of a graph in such a way that no two adjacent vertices have the same color. This is also called the vertex coloring problem. If coloring is done using at most m colors, it is called m-coloring.



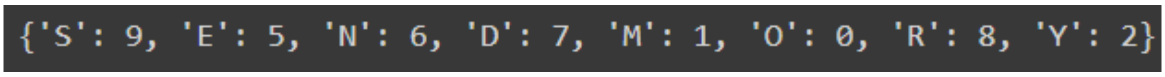
Code:

```
#Import the necessary libraries
from typing import Generic, TypeVar, Dict, List, Optional
from abc import ABC, abstractmethod
#Declares a type variable V as variable type and D as domain type
V = TypeVar('V') # variable type
D = TypeVar('D') # domain type
#This is a Base class for all constraints
class Constraint(Generic[V, D], ABC):
    # The variables that the constraint is between
    def __init__(self, variables: List[V]) -> None:
        self.variables = variables
    # This is an abstract method which must be overridden by subclasses
    @abstractmethod
    def satisfied(self, assignment: Dict[V, D]) -> bool:
        ...
# A constraint satisfaction problem consists of variables of type V
# that have ranges of values known as domains of type D and constraints
# that determine whether a particular variable's domain selection is valid
class CSP(Generic[V, D]):
    def __init__(self, variables: List[V], domains: Dict[V, List[D]]) -> None:
        # variables to be constrained
        #TODO: Assign variables parameter to self.variables
        self.variables: List[V] = variables
        # domain of each variable
        #TODO: Assign domains parameter to self.domains
        self.domains: Dict[V, List[D]] = domains
        #TODO: Assign an empty dictionary to self.constraints
        self.constraints: Dict[V, List[Constraint[V, D]]] = {}
        #TODO:Iterate over self.variables
        for variable in self.variables:
            self.constraints[variable] = []
            #TODO:If the variable is not in domains, then raise a LookupError("Every variable
should have a domain assigned to it.")
            if variable not in self.domains:
                raise LookupError("Every variable should have a domain assigned to it.")
    #This method add constraint to variables as per their domains
    def add_constraint(self, constraint: Constraint[V, D]) -> None:
        for variable in constraint.variables:
            if variable not in self.variables:
                raise LookupError("Variable in constraint not in CSP")
            else:
                self.constraints[variable].append(constraint)
    # Check if the value assignment is consistent by checking all constraints
    # for the given variable against it
    def consistent(self, variable: V, assignment: Dict[V, D]) -> bool:
        #TODO: Iterate over self.constraints[variable]
        for constraint in self.constraints[variable]:
            #TODO: if constraint not satisfied then return False
            if not constraint.satisfied(assignment):
                return False
        #TODO: otherwise return True
        return True

    #This method is performing the backtracking search to find the result
    def backtracking_search(self, assignment: Dict[V, D] = {}) -> Optional[Dict[V, D]]:
        # assignment is complete if every variable is assigned (our base case)
        if len(assignment) == len(self.variables):
            return assignment

        # get all variables in the CSP but not in the assignment
        unassigned: List[V] = [v for v in self.variables if v not in assignment]

        # get the every possible domain value of the first unassigned variable
        first: V = unassigned[0]
        #TODO: Iterate over self.domains[first]
        for value in self.domains[first]:
            local_assignment = assignment.copy()
            #TODO: Assign the value
            local_assignment[first] = value
            # if we're still consistent, we recurse (continue)
            if self.consistent(first, local_assignment):
                #TODO: recursively call the self.backtracking_search method based on the
  local_assignment
                result: Optional[Dict[V, D]] = self.backtracking_search(local_assignment)
            # if we didn't find the result, we will end up backtracking
            if result is not None:
                return result
        return None
#MapColoringConstraint is a subclass of Constraint class
class MapColoringConstraint(Constraint[str, str]):
    def __init__(self, place1: str, place2: str) -> None:
        super().__init__([place1, place2])
        self.place1: str = place1
        self.place2: str = place2
    #Define the abstract method satisfied in subclass
    def satisfied(self, assignment: Dict[str, str]) -> bool:
        # If either place is not in the assignment then it is not
        # yet possible for their colors to be conflicting
        if self.place1 not in assignment or self.place2 not in assignment:
            return True
        # check the color assigned to place1 is not the same as the
        # color assigned to place2
        return assignment[self.place1] != assignment[self.place2]
#Main starts
if __name__ == "__main__":
    #Initializes the variables as per the regions of the graph
    variables: List[str] = ["BOX_1", "BOX_2", "BOX_4",
                            "BOX_3", "BOX_5", "BOX_6", "BOX_7"]
    #TODO: Initialize the domain as empty dictionary
    domains: Dict[str, List[str]] = {}
    for variable in variables:
        #Initialize the domain of each variable
        domains[variable] = ["red", "green", "blue"]
    #Instantiate the object of CSP
    csp: CSP[str, str] = CSP(variables, domains)
    #Add constraints to the given MAP problem
    csp.add_constraint(MapColoringConstraint("BOX_1", "BOX_2"))
    csp.add_constraint(MapColoringConstraint("BOX_1", "BOX_4"))
    csp.add_constraint(MapColoringConstraint("BOX_4", "BOX_2"))
    csp.add_constraint(MapColoringConstraint("BOX_3", "BOX_2"))
    csp.add_constraint(MapColoringConstraint("BOX_3", "BOX_4"))
    csp.add_constraint(MapColoringConstraint("BOX_3", "BOX_5"))
    csp.add_constraint(MapColoringConstraint("BOX_5", "BOX_4"))
    csp.add_constraint(MapColoringConstraint("BOX_6", "BOX_4"))
    csp.add_constraint(MapColoringConstraint("BOX_6", "BOX_5"))
    csp.add_constraint(MapColoringConstraint("BOX_6", "BOX_7"))
    #Finding the solution to the problem by calling the backtracking_search() method
    solution: Optional[Dict[str, str]] = csp.backtracking_search()
    if solution is None:
        print("No solution found!")
    else:
        print(solution)
#SendMoreMoneyConstraint is a subclass of Constraint class
class SendMoreMoneyConstraint(Constraint[str, int]):
    def __init__(self, letters: List[str]) -> None:
        super().__init__(letters)
        self.letters: List[str] = letters

    def satisfied(self, assignment: Dict[str, int]) -> bool:
        # if there are duplicate values then it's not a solution
        if len(set(assignment.values())) < len(assignment):
            return False

        # if all variables have been assigned, check if it adds correctly
        if len(assignment) == len(self.letters):
            s: int = assignment["S"]
            e: int = assignment["E"]
            n: int = assignment["N"]
            d: int = assignment["D"]
            m: int = assignment["M"]
            o: int = assignment["O"]
            r: int = assignment["R"]
            y: int = assignment["Y"]
            send: int = s * 1000 + e * 100 + n * 10 + d
            more: int = m * 1000 + o * 100 + r * 10 + e
            money: int = m * 10000 + o * 1000 + n * 100 + e * 10 + y
            return send + more == money
        return True # no conflict
if __name__ == "__main__":
    letters: List[str] = ["S", "E", "N", "D", "M", "O", "R", "Y"]
    possible_digits: Dict[str, List[int]] = {}
    for letter in letters:
        possible_digits[letter] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    possible_digits["M"] = [1]  # so we don't get answers starting with a 0
    csp: CSP[str, int] = CSP(letters, possible_digits)
    csp.add_constraint(SendMoreMoneyConstraint(letters))
    solution: Optional[Dict[str, int]] = csp.backtracking_search()
    if solution is None:
        print("No solution found!")
    else:
        print(solution)
```

Output:



```
{'BOX_1': 'red', 'BOX_2': 'green', 'BOX_4': 'blue', 'BOX_3': 'red', 'BOX_5': 'green', 'BOX_6': 'red', 'BOX_7': 'green'}
```



```
{'S': 9, 'E': 5, 'N': 6, 'D': 7, 'M': 1, 'O': 0, 'R': 8, 'Y': 2}
```

Program 10

Aim: WAP to implement DFS for water jug problem using Python

Theory:

The Water Jug Problem is approached using the Depth-First Search (DFS) algorithm. The algorithm starts with an empty stack and pushes the initial state (both jugs empty) onto the stack. While the stack is not empty, it pops a state from the stack, checks if the state represents the desired quantity, generates all possible next states from the current state, and pushes them onto the stack. This process continues until the stack becomes empty or the desired quantity is found.

Code:

```
def solveWaterJugProblem(capacity_jug1, capacity_jug2, desired_quantity):
    stack = []
    stack.append((0, 0))  # Initial state: both jugs empty

    while stack:
        current_state = stack.pop()

        if current_state[0] == desired_quantity or current_state[1] == desired_quantity:
            return current_state

        next_states = generateNextStates(current_state, capacity_jug1, capacity_jug2)
        stack.extend(next_states)

    return "No solution found"

def generateNextStates(state, capacity_jug1, capacity_jug2):
    next_states = []

    # Fill Jug 1
    next_states.append((capacity_jug1, state[1]))

    # Fill Jug 2
    next_states.append((state[0], capacity_jug2))

    # Empty Jug 1
    next_states.append((0, state[1]))

    # Empty Jug 2
    next_states.append((state[0], 0))

    # Pour water from Jug 1 to Jug 2
    pour_amount = min(state[0], capacity_jug2 - state[1])
    next_states.append((state[

0] - pour_amount, state[1] + pour_amount))

    # Pour water from Jug 2 to Jug 1
    pour_amount = min(state[1], capacity_jug1 - state[0])
    next_states.append((state[0] + pour_amount, state[1] - pour_amount))

    return next_states
```

Program 11

Aim: Tokenization of word and Sentences with the help of NLTK package

Theory:

NLTK is a leading platform for building Python programs to work with human language data. It provides easy-to-use interfaces to over 50 corpora and lexical resources such as WordNet, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning, wrappers for industrial-strength NLP libraries, and an active discussion forum.

Code:

```
#Import the necessary libraries

import nltk                              # Python library for NLP

from nltk.corpus import twitter_samples    # sample Twitter dataset from NLTK

import matplotlib.pyplot as plt           # library for visualization

import random                              # pseudo-random number generator

import numpy as np

# downloads sample twitter dataset. execute the line below if running on a local machine.

nltk.download('twitter_samples')

# select the set of positive and negative tweets

all_positive_tweets = twitter_samples.strings('positive_tweets.json')

all_negative_tweets = twitter_samples.strings('negative_tweets.json')

#TODO: Print the size of positive and negative tweets

print('Number of positive tweets: ', len(all_positive_tweets))

print('Number of negative tweets: ', len(all_negative_tweets))


#TODO:Print the type of positive and negative tweets, using type()

print('\nThe type of all_positive_tweets is: ', type(all_positive_tweets))

print('The type of a tweet entry is: ', type(all_positive_tweets[0]))
```

```
#PLOT the positive and negative tweets in a pie-chart
# Declare a figure with a custom size
fig = plt.figure(figsize=(5, 5))

# labels for the two classes
labels = 'Positives', 'Negative'

# Sizes for each slide
sizes = [len(all_positive_tweets), len(all_negative_tweets)]

# Declare pie chart, where the slices will be ordered and plotted counter-clockwise:
plt.pie(sizes, labels=labels, autopct='%1.1f%%',
        shadow=True, startangle=90)

# Equal aspect ratio ensures that pie is drawn as a circle.
plt.axis('equal')

# Display the chart
plt.show()
#TODO: Display a random tweet from positive and negative tweet. The size of positive and
negative tweets are 5000 each.
#Generate a random number between 0 and 5000 using random.randint()
# print positive in greeen
print('\033[92m' + all_positive_tweets[np.random.randint(0,5000)])

# print negative in red
print('\033[91m' + all_negative_tweets[np.random.randint(0,5000)])
# Our selected sample. Complex enough to exemplify each step
tweet = all_positive_tweets[2277]
print(tweet)
# download the stopwords from NLTK
nltk.download('stopwords')
```

```
#Import the necessary libraries
import re                              # library for regular expression operations
import string                          # for string operations

from nltk.corpus import stopwords          # module for stop words that come with NLTK
from nltk.stem import PorterStemmer        # module for stemming
from nltk.tokenize import TweetTokenizer   # module for tokenizing strings
print('\033[92m' + tweet)
print('\033[94m')

# remove old style retweet text "RT"
tweet2 = re.sub(r'^RT[\s]+', '', tweet)

# remove hyperlinks
tweet2 = re.sub(r'\https?:\/\/.*[\r\n]*', '', tweet2)

# remove hashtags
# only removing the hash # sign from the word
tweet2 = re.sub(r'#', '', tweet2)

print(tweet2)
print()
print('\033[92m' + tweet2)
print('\033[94m')

# instantiate tokenizer class
tokenizer = TweetTokenizer(preserve_case=False, strip_handles=True,
                           reduce_len=True)

# tokenize tweets
tweet_tokens = tokenizer.tokenize(tweet2)
```

```
print()
print('Tokenized string:')
print(tweet_tokens)
#Import the english stop words list from NLTK
stopwords_english = stopwords.words('english')

print('Stop words\n')
print(stopwords_english)

print('\nPunctuation\n')
print(string.punctuation)
print()
print('\033[92m')
print(tweet_tokens)
print('\033[94m')

#TODO: Create the empty list to store the clean tweets after removing stopwords and
punctuation
tweets_clean = []

#TODO: Remove stopwords and punctuation from the tweet_tokens
for word in tweet_tokens: # Go through every word in your tokens list
    if (word not in stopwords_english and  # remove stopwords
        word not in string.punctuation):  # remove punctuation
        #TODO: Append the clean word in the tweets_clean list
        tweets_clean.append(word)

print('removed stop words and punctuation:')
print(tweets_clean)
print()
print('\033[92m')
print(tweets_clean)
print('\033[94m')

# Instantiate stemming class
stemmer = PorterStemmer()

#TODO: Create an empty list to store the stems
tweets_stem = []

#TODO: Itearate over the tweets_clean fot stemming
for word in tweets_clean:
   #TODO:call the stem function for stemming the word
   stem_word = stemmer.stem(word)           # stemming word
   #TODO:Append the stem_word in tweets_stem list
   tweets_stem.append(stem_word)

print('stemmed words:')
print(tweets_stem)
def process_tweet(tweet):
   """Process tweet function.
   Input:
      tweet: a string containing a tweet
   Output:
      tweets_clean: a list of words containing the processed tweet

   """
   stemmer = PorterStemmer()
   stopwords_english = stopwords.words('english')
   # remove stock market tickers like $GE
   tweet = re.sub(r'\$\w*', '', tweet)
   # remove old style retweet text "RT"
   tweet = re.sub(r'^RT[\s]+', '', tweet)
   # remove hyperlinks
```

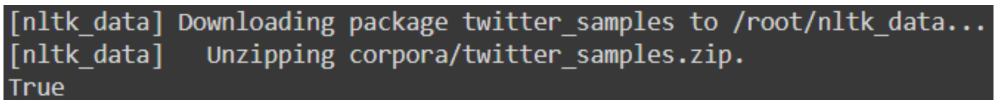
```
    tweet = re.sub(r'https?:\/\/.*[\r\n]*', '', tweet)
    # remove hashtags
    # only removing the hash # sign from the word
    tweet = re.sub(r'#', '', tweet)
    # tokenize tweets
    tokenizer = TweetTokenizer(preserve_case=False, strip_handles=True,
                               reduce_len=True)
    tweet_tokens = tokenizer.tokenize(tweet)

    tweets_clean = []
    for word in tweet_tokens:
        if (word not in stopwords_english and  # remove stopwords
                word not in string.punctuation):  # remove punctuation
            # tweets_clean.append(word)
            stem_word = stemmer.stem(word)  # stemming word
            tweets_clean.append(stem_word)

    return tweets_clean
# choose the same tweet
tweet = all_positive_tweets[2277]

print()
print('\033[92m')
print(tweet)
print('\033[94m')

#TODO: call the process_tweet function
tweets_stem = process_tweet(tweet)

print('preprocessed tweet:')
print(tweets_stem) # Print the result
#TODO: Concatenate the lists, 1st part is the positive tweets followed by the negative
```

```
tweets = all_positive_tweets + all_negative_tweets

# let's see how many tweets we have
print("Number of tweets: ", len(tweets))
# make a numpy array representing labels of the tweets
labels = np.append(np.ones((len(all_positive_tweets))), np.zeros((len(all_negative_tweets))))
dictionary = {'key1': 1, 'key2': 2}
# Add a new entry
dictionary['key3'] = -5

# Overwrite the value of key1
dictionary['key1'] = 0

print(dictionary)
# Square bracket lookup when the key exist
print(dictionary['key2'])
# The output of this line is intended to produce a KeyError
print(dictionary['key8'])
# This prints a message because the key is not found
if 'key7' in dictionary:
    print(dictionary['key7'])
else:
    print('key does not exist!')

# This prints -1 because the key is not found and we set the default to -1
print(dictionary.get('key7', -1))
def build_freqs(tweets, ys):
    """Build frequencies.
    Input:
        tweets: a list of tweets
        ys: an m x 1 array with the sentiment label of each tweet
            (either 0 or 1)
```

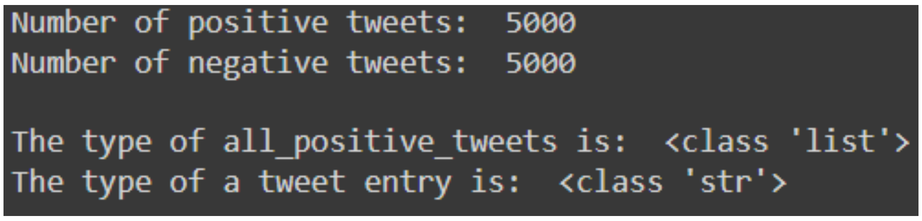
```
   Output:
       freqs: a dictionary mapping each (word, sentiment) pair to its
       frequency
   """
   # Convert np array to list since zip needs an iterable.
   # The squeeze is necessary or the list ends up with one element.
   # Also note that this is just a NOP if ys is already a list.
   yslist = np.squeeze(ys).tolist()


   # Start with an empty dictionary and populate it by looping over all tweets
   # and over all processed words in each tweet.
   #TODO:Create the empty dictionary
   freqs = {}
   for y, tweet in zip(yslist, tweets):
       #TODO: Iterate over all the words returned by calling the process_tweet() function for
each tweet
       for word in process_tweet(tweet):
           pair = (word, y)
           #TODO: If pair matches in the dictionary, then increment the count of corresponding
pair.
           if pair in freqs:
               freqs[pair] +=1
           else:
               #TODO: If pair does not matches in the dictionary, then set the count of
corresponding pair as 1.
               freqs[pair] = 1


   #TODO: Return the dictionary
   return freqs
#TODO: Call the build_freqs function to create frequency dictionary based on tweets and labels
freqs = build_freqs(tweets, labels)


#TODO: Display the data type of freqs
```

```
print(f'type(freqs) = {type(freqs)}')


#TODO: Display the length of the dictionary
print(f'len(freqs) = {len(freqs)}')
#TODO: prinf all the key-value pair of frequency dictionary
for key, value in freqs.items():
    print(key, value)
# select some words to appear in the report. we will assume that each word is unique (i.e. no
duplicates)
keys = ['happi', 'merri', 'nice', 'good', 'bad', 'sad', 'mad', 'best', 'pretti',
        '❤', ':)', ':(', '😒', '😬', '😄', '😍', '♛',
        'song', 'idea', 'power', 'play', 'magnific']




#TODO: Create the empty list as list representing our table of word counts, where
#each element consist of a sublist with this pattern: [<word>, <positive_count>,
<negative_count>].
data = []


#TODO:Iterate over each word in keys
for word in keys:


    # initialize positive and negative counts
    pos = 0
    neg = 0


    # retrieve number of positive counts
    if (word, 1) in freqs:
        pos = freqs[(word, 1)]


    # retrieve number of negative counts
    if (word, 0) in freqs:
        neg = freqs[(word, 0)]

    # append the word counts to the table
    data.append([word, pos, neg])

data
fig, ax = plt.subplots(figsize = (8, 8))

# convert positive raw counts to logarithmic scale. we add 1 to avoid log(0)
x = np.log([x[1] + 1 for x in data])

# do the same for the negative counts
y = np.log([x[2] + 1 for x in data])

# Plot a dot for each pair of words
ax.scatter(x, y)

# assign axis labels
plt.xlabel("Log Positive count")
plt.ylabel("Log Negative count")

# Add the word as the label at the same position as you added the points just before
for i in range(0, len(data)):
    ax.annotate(data[i][0], (x[i], y[i]), fontsize=12)

ax.plot([0, 9], [0, 9], color = 'red') # Plot the red line that divides the 2 areas.
plt.show()
```

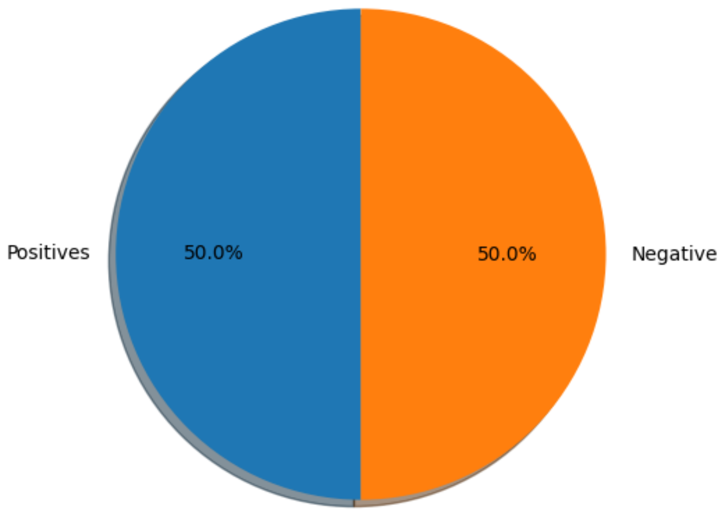
Output:

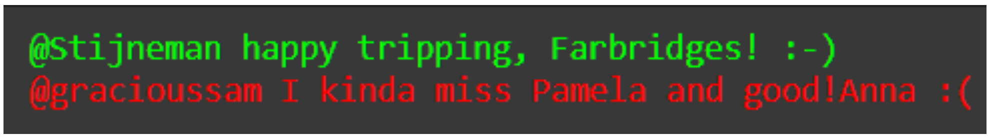


```
[nltk_data] Downloading package twitter_samples to /root/nltk_data...
[nltk_data]   Unzipping corpora/twitter_samples.zip.
True
```

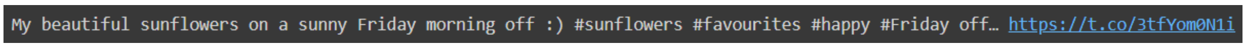


```
Number of positive tweets:  5000
Number of negative tweets:  5000

The type of all_positive_tweets is:  <class 'list'>
The type of a tweet entry is:  <class 'str'>
```

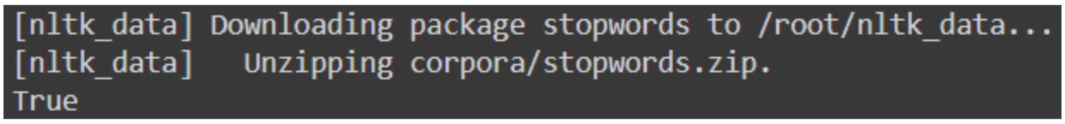




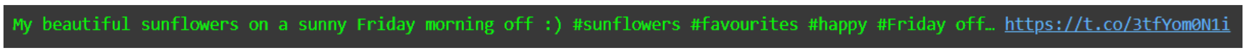
```
@Stijneman happy tripping, Farbridges! :-)
@gracioussam I kinda miss Pamela and good!Anna :(
```



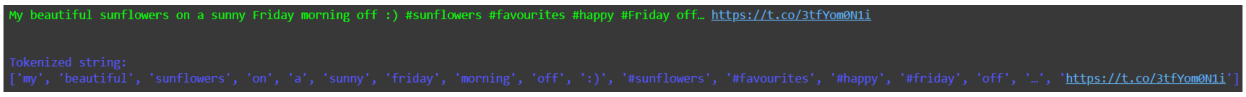
```
My beautiful sunflowers on a sunny Friday morning off :) #sunflowers #favourites #happy #Friday off… https://t.co/3tfYom0N1i
```



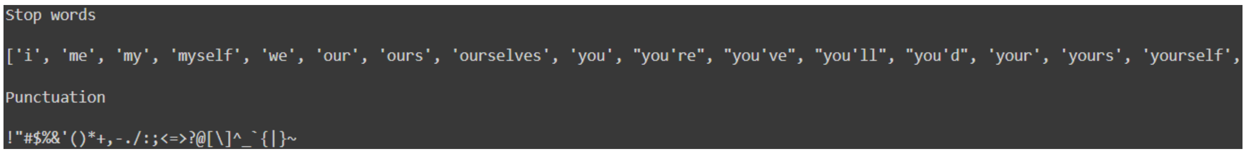
```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
True
```



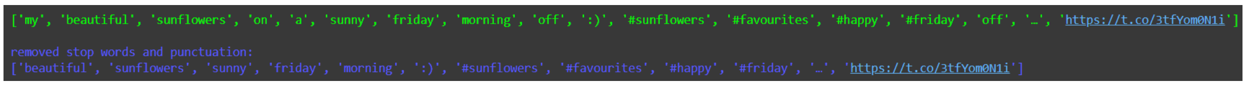
```
My beautiful sunflowers on a sunny Friday morning off :) #sunflowers #favourites #happy #Friday off… https://t.co/3tfYom0N1i
```



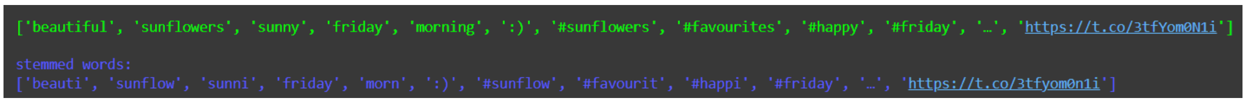
```
My beautiful sunflowers on a sunny Friday morning off :) #sunflowers #favourites #happy #Friday off… https://t.co/3tfYom0N1i


Tokenized string:
['my', 'beautiful', 'sunflowers', 'on', 'a', 'sunny', 'friday', 'morning', 'off', ':)', '#sunflowers', '#favourites', '#happy', '#friday', 'off', '…', 'https://t.co/3tfYom0N1i']
```



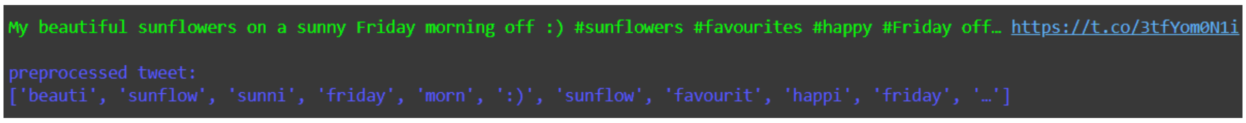
```
Stop words

['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "you've", "you'll", "you'd", 'your', 'yours', 'yourself',

Punctuation

!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
```



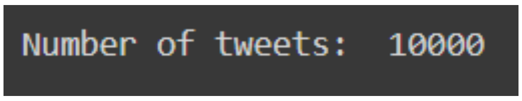
```
['my', 'beautiful', 'sunflowers', 'on', 'a', 'sunny', 'friday', 'morning', 'off', ':)', '#sunflowers', '#favourites', '#happy', '#friday', 'off', '…', 'https://t.co/3tfYom0N1i']

removed stop words and punctuation:
['beautiful', 'sunflowers', 'sunny', 'friday', 'morning', ':)', '#sunflowers', '#favourites', '#happy', '#friday', '…', 'https://t.co/3tfYom0N1i']
```



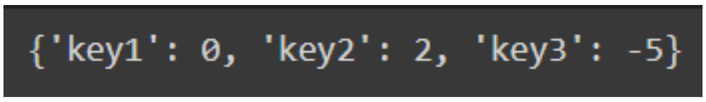
```
['beautiful', 'sunflowers', 'sunny', 'friday', 'morning', ':)', '#sunflowers', '#favourites', '#happy', '#friday', '…', 'https://t.co/3tfYom0N1i']

stemmed words:
['beauti', 'sunflow', 'sunni', 'friday', 'morn', ':)', '#sunflow', '#favourit', '#happi', '#friday', '…', 'https://t.co/3tfyom0n1i']
```



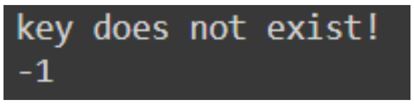
```
My beautiful sunflowers on a sunny Friday morning off :) #sunflowers #favourites #happy #Friday off… https://t.co/3tfYom0N1i

preprocessed tweet:
['beauti', 'sunflow', 'sunni', 'friday', 'morn', ':)', 'sunflow', 'favourit', 'happi', 'friday', '…']
```



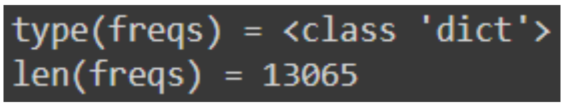
```
Number of tweets:  10000
```



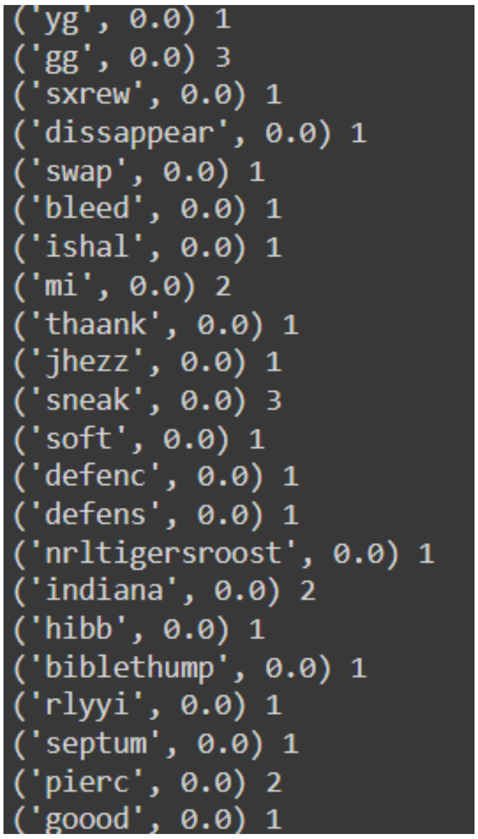
```
{'key1': 0, 'key2': 2, 'key3': -5}
```



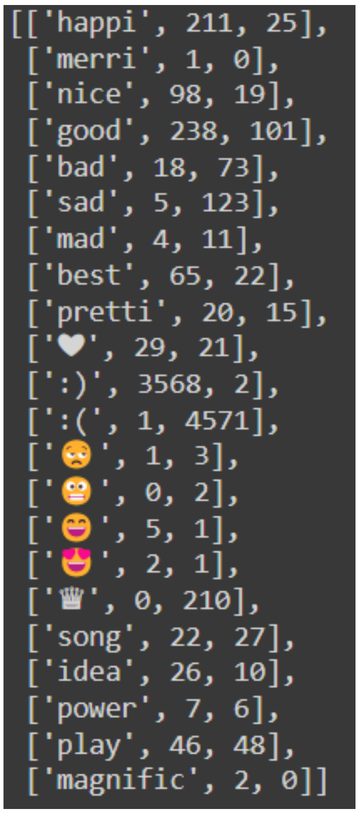
```
key does not exist!
-1
```



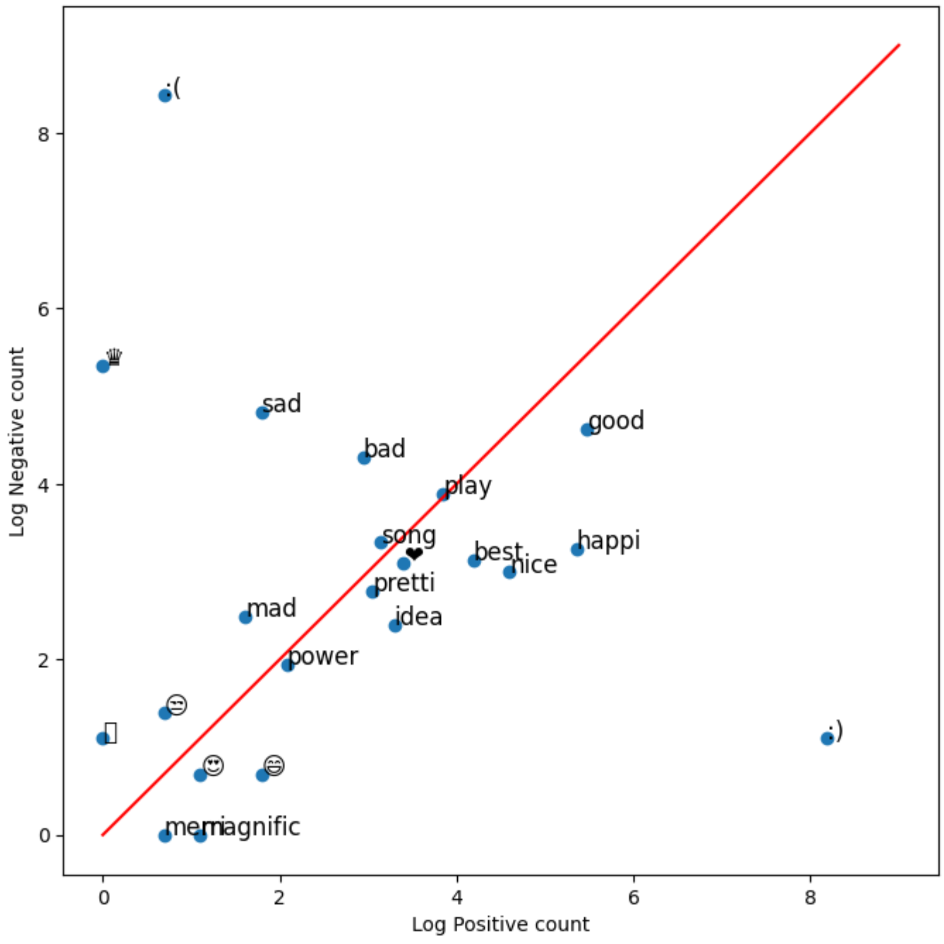
```
type(freqs) = <class 'dict'>
len(freqs) = 13065
```



```
('yg', 0.0) 1
('gg', 0.0) 3
('sxrew', 0.0) 1
('dissappear', 0.0) 1
('swap', 0.0) 1
('bleed', 0.0) 1
('ishal', 0.0) 1
('mi', 0.0) 2
('thaank', 0.0) 1
('jhezz', 0.0) 1
('sneak', 0.0) 3
('soft', 0.0) 1
('defenc', 0.0) 1
('defens', 0.0) 1
('nrltigersroost', 0.0) 1
('indiana', 0.0) 2
('hibb', 0.0) 1
('biblethump', 0.0) 1
('rlyyi', 0.0) 1
('septum', 0.0) 1
('pierc', 0.0) 2
('goood', 0.0) 1
```



```
[['happi', 211, 25],
 ['merri', 1, 0],
 ['nice', 98, 19],
 ['good', 238, 101],
 ['bad', 18, 73],
 ['sad', 5, 123],
 ['mad', 4, 11],
 ['best', 65, 22],
 ['pretti', 20, 15],
 ['🤍', 29, 21],
 [':)', 3568, 2],
 [':(', 1, 4571],
 ['😟', 1, 3],
 ['😬', 0, 2],
 ['😄', 5, 1],
 ['😍', 2, 1],
 ['♛', 0, 210],
 ['song', 22, 27],
 ['idea', 26, 10],
 ['power', 7, 6],
 ['play', 46, 48],
 ['magnific', 2, 0]]
```


:(
♛
sad
good
bad
play
song
❤
best
nice
happi
pretti
mad
idea
power
😑
:)
😍
😄
merri
magnific
Log Negative count
Log Positive count