1. Write a program in python to implement simple arithmetic operations.

```
num1 = input('Enter first number: ')
num2 = input('Enter second number: ')
sum = float(num1) + float(num2)
min = float(num1) - float(num2)
mul = float(num1) * float(num2)
div = float(num1) / float(num2)
print('The sum of {0} and {1} is {2}'.format(num1, num2, sum))
print('The subtraction of {0} and {1} is {2}'.format(num1, num2,
min))
print('The multiplication of {0} and {1} is {2}'.format(num1, num2,
mul))
print('The division of {0} and {1} is {2}'.format(num1, num2, div))
```

Output:

```
Enter first number: 55
Enter second number: 23


The sum of 55 and 23 is 78.0
The subtraction of 55 and 23 is 32.0
The multiplication of 55 and 23 is 1265.0
The division of 55 and 23 is 2.391304347826087
```

2. Write a program to implement a vacuum cleaner agent with basic functions.

```
import random
def display(room):
    print(room)
room = [
    [1, 1, 1, 1],
    [1, 1, 1, 1],
    [1, 1, 1, 1],
    [1, 1, 1, 1],
]
print("All the rooom are dirty")
display(room)

x =0
y= 0
while x < 4:
    while y < 4:
```

```
        room[x][y] = random.choice([0,1])
        y+=1
    x+=1
    y=0

print("Before cleaning the room I detect all of these random dirts")
display(room)
x =0
y= 0
z=0
while x < 4:
    while y < 4:
        if room[x][y] == 1:
            print("Vaccum in this location now,",x, y)
            room[x][y] = 0
            z+=1
        y+=1
    x+=1
    y=0
pro= (100-((z/16)*100))
print("Room is clean now, Thanks for using")
display(room)
print('performance=',pro,'%')
```

Output:

```
Before cleaning the room I detect all of these random dirts
[[0, 0, 1, 1], [1, 0, 1, 1], [1, 0, 1, 0], [0, 0, 1, 1]]


Vaccum in this location now, 0 2
Vaccum in this location now, 0 3
Vaccum in this location now, 1 0
Vaccum in this location now, 1 2
Vaccum in this location now, 1 3
Vaccum in this location now, 2 0
Vaccum in this location now, 2 2
Vaccum in this location now, 3 2
Vaccum in this location now, 3 3
Room is clean now, Thanks for using
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
performance= 43.75 %
```

3. Write a Python Program to find the shortest distance between any two places using a A* search algorithm. Repeat the experiment for different Graphs.

```
class PQueue():
  def __init__(self):
    self.dict = {}
    self.keys = []
    self.sorted = False
  def push(self, k, v):
    self.dict[k] = v
    self.sorted = False
  def _sort(self):
    self.keys = sorted(self.dict, key=self.dict.get,reverse=True)
    self.sorted = True
  def pop(self):
    try:
        if not self.sorted:
            self._sort()
        key = self.keys.pop()
        value = self.dict[key]
        self.dict.pop(key)
        return key, value
    except:
        return None
def path_costs(path):
    c = {}
    with open(path, 'r') as file:
        for line in file:
            line = line.split(", ")
            v = int(line.pop())
            e1 = line.pop()
            e2 = line.pop()
            print(" VALUE OF V, e1, e2", v, " ",e1, " ", e2 )
            if e1 not in c:
                c[e1] = {}
            if e2 not in c:
                c[e2] = {}
            c[e1][e2] = c[e2][e1] = v

    print(" C VALUE", c)
    return c
def a_star(start, goal, h, g):
    frontier = PQueue()
# pushing path and cost to pqueue
    frontier.push(start, h[start])
    while True:
# poping path with least cost
```

```
        path, cost = frontier.pop()
        print(path+ " " +str(cost))
# splitting out end node in path
        end = path.split("->")[-1]
# removing heuristic value of end node from cost
        cost -= h[end]
        if goal == end:
            break
        for node, weight in g[end].items():
# adding edge weight(cost) and node heuristic to total cost
            new_cost = cost + weight + h[node]
            new_path = path + "->" + node
# adding new path and cost to pqueue
            frontier.push(new_path, new_cost)

a_star('Arad', 'Bucharest', heuristics('/home/Heuristics.txt'),
path_costs('/home/Paths.txt'))
```

output:

```
VALUE OF V, e1, e2 75   Zerind   Arad
 VALUE OF V, e1, e2 140   Sibiu   Arad
 VALUE OF V, e1, e2 118   Timisoara   Arad
 VALUE OF V, e1, e2 71   Oradea   Zerind
 VALUE OF V, e1, e2 151   Sibiu   Oradea
 VALUE OF V, e1, e2 111   Lugoj   Timisoara
 VALUE OF V, e1, e2 99   Fagaras   Sibiu
 VALUE OF V, e1, e2 80   Rimnicu Vilcea   Sibiu
 VALUE OF V, e1, e2 70   Mehadia   Lugoj
 VALUE OF V, e1, e2 211   Bucharest   Fagaras
 VALUE OF V, e1, e2 97   Pitesti   Rimnicu Vilcea
 VALUE OF V, e1, e2 146   Craiova   Rimnicu Vilcea
 VALUE OF V, e1, e2 75   Dobreta   Mehadia
 VALUE OF V, e1, e2 101   Pitesti   Bucharest
 VALUE OF V, e1, e2 85   Urziceni   Bucharest
 VALUE OF V, e1, e2 90   Giurgiu   Bucharest
 VALUE OF V, e1, e2 138   Craiova   Pitesti
 VALUE OF V, e1, e2 120   Dobreta   Craiova
 VALUE OF V, e1, e2 98   Hirsova   Urziceni
 VALUE OF V, e1, e2 142   Vaslui   Urziceni
 VALUE OF V, e1, e2 86   Eforie   Hirsova
 VALUE OF V, e1, e2 92   Lasi   Vaslui
 VALUE OF V, e1, e2 87   Neamt   Lasi


 C VALUE {'Zerind': {'Arad': 75, 'Oradea': 71}, 'Arad': {'Zerind': 75,
'Sibiu': 140, 'Timisoara': 118}, 'Sibiu': {'Arad': 140, 'Oradea': 151,
'Fagaras': 99, 'Rimnicu Vilcea': 80}, 'Timisoara': {'Arad': 118,
'Lugoj': 111}, 'Oradea': {'Zerind': 71, 'Sibiu': 151}, 'Lugoj':
{'Timisoara': 111, 'Mehadia': 70}, 'Fagaras': {'Sibiu': 99,
'Bucharest': 211}, 'Rimnicu Vilcea': {'Sibiu': 80, 'Pitesti': 97,
'Craiova': 146}, 'Mehadia': {'Lugoj': 70, 'Dobreta': 75}, 'Bucharest':
```

```
{'Fagaras': 211, 'Pitesti': 101, 'Urziceni': 85, 'Giurgiu': 90},
'Pitesti': {'Rimnicu Vilcea': 97, 'Bucharest': 101, 'Craiova': 138},
'Craiova': {'Rimnicu Vilcea': 146, 'Pitesti': 138, 'Dobreta': 120},
'Dobreta': {'Mehadia': 75, 'Craiova': 120}, 'Urziceni': {'Bucharest':
85, 'Hirsova': 98, 'Vaslui': 142}, 'Giurgiu': {'Bucharest': 90},
'Hirsova': {'Urziceni': 98, 'Eforie': 86}, 'Vaslui': {'Urziceni': 142,
'Lasi': 92}, 'Eforie': {'Hirsova': 86}, 'Lasi': {'Vaslui': 92, 'Neamt':
87}, 'Neamt': {'Lasi': 87}}

Arad 366
Arad->Sibiu 393
Arad->Sibiu->Rimnicu Vilcea 413
Arad->Sibiu->Fagaras 415
Arad->Sibiu->Rimnicu Vilcea->Pitesti 417
Arad->Sibiu->Rimnicu Vilcea->Pitesti->Bucharest 41
```

4. Write a program to implement a Minimax decision-making algorithm, typically used in a turn-based, two player games. The goal of the algorithm is to find the optimal next move.

```python
import math
def minimax(tree, depth):
  max_turn = bool(depth % 2)
  for _ in range(depth):
    zipped = zip(tree[::2], tree[1::2])
    if max_turn:
        tree = [max(a, b) for a, b in zipped] # max player
    else:
        tree = [min(a, b) for a, b in zipped] # min player
    max_turn = not max_turn # swapping turns
  return tree[0]
A = [-1, 4, 2, 6, -3, -5, 0, 7]
depth = math.ceil(math.log(len(A), 2))
print(f"Result = {minimax(A, depth)}")
```

output:

```
Result = 4
```

5. Write a program to implement Alpha Beta pruning in Python. The algorithm can be applied to any depth of tree by not only pruning the tree leaves but also the entire subtree. Order the nodes in the tree such that the best nodes are checked first from the shallowest node.

```
maximum, minimum=1000,-1000
def fun_alphabeta(d,node,maxp,v,A,B):
  if d==3:
    return v[node]
  if maxp:
    best =minimum
    for i in range(0,2):
      value=fun_alphabeta(d+1,node*2+i,False,v,A,B)
      best=max(best,value)
      A=max(A,best)
      if B<=A:
        break
    return best
  else:
    best=maximum
    for i in range(0,2):
      value=fun_alphabeta(d+1,node*2+i,True,v,A,B)
      best=min(best,value)
      A=min(A,best)
      if B<=A:
        break
    return best
scr=[]
x= int(input("Enter total number of leaf node:"))
for i in range(x):
  y=int(input("Enter node value"))
  scr.append(y)
d=int(input("Enter depth value:"))
node=int(input("Enter node value:"))
print("The optimal value
is:",fun_alphabeta(d,node,True,scr,minimum,maximum))
```

output:

```
Enter total number of leaf node:8
Enter node value2
Enter node value3
Enter node value5
Enter node value9
Enter node value0
Enter node value1
Enter node value7
Enter node value5
Enter depth value:0
Enter node value:0
The optimal value is: 3
```

6. Write a program to solve 4 Queens Problem.

```
# Taking number of queens as input from user
print ("Enter the number of queens")
N = int(input())
# here we create a chessboard
# NxN matrix with all elements set to 0
board = [[0]*N for _ in range(N)]
def attack(i, j):
    #checking vertically and horizontally
    for k in range(0,N):
        if board[i][k]==1 or board[k][j]==1:
            return True
    #checking diagonally
    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:

```
Enter the number of queens
4
[0, 1, 0, 0]
[0, 0, 0, 1]
[1, 0, 0, 0]
[0, 0, 1, 0]
```

7. Implementation of Tic Tac Toe game here, the player needs to take turns marking the spaces in a 3x3 grid with their own marks, if 3 consecutive marks (Horizontal, Vertical, Diagonal) are formed then the player who owns these moves get won. Noughts and Crosses or X's and O's abbreviations can be used to play.

```
import os
turn = 'X'
win = False
spaces = 9
def draw(board):
    for i in range(6, -1, -3):
        print(' ' + board[i] + '|' +
              board[i+1] + '|' + board[i+2])
def takeinput(board, spaces, turn):
    pos = -1
    print(turn + "'s turn:")
    while pos == -1:
        try:
            print("Pick position 1-9:")
            pos = int(input())
            if(pos < 1 or pos > 9):
                pos = -1
            elif board[pos - 1] != ' ':
                pos = -1
        except:
            print("enter a valid position")
    spaces -= 1
    board[pos - 1] = turn
    if turn == 'X':
        turn = 'O'
    else:
        turn = 'X'
    return board, spaces, turn
def checkwin(board):
# could probably make this better
    for i in range(0, 3):
        r = i*3
        if board[r] != ' ':
            if board[r] == board[r+1] and board[r+1] == board[r+2]:
                return board[r]
# columns
        if board[i] != ' ':
            if board[i] == board[i+3] and board[i] == board[i+6]:
                return board[i]
# diagonals
    if board[0] != ' ':
        if (board[0] == board[4] and board[4] == board[8]):
```

```
            return board[0]
    if board[2] != ' ':
        if (board[2] == board[4] and board[4] == board[6]):
            return board[2]
    return 0
board = [' ']*9
while not win and spaces:
    draw(board)
    board, spaces, turn = takeinput(board, spaces, turn)
    win = checkwin(board)
    os.system('cls')
draw(board)
if not win and not spaces:
    print("draw")
elif win:
    print(f'{win} wins')
    input()
```

output:

```
| |
  | |
  | |
X's turn:
Pick position 1-9:
1
  | |
  | |
 X| |
O's turn:
Pick position 1-9:
9
  | |O
  | |
 X| |
X's turn:
Pick position 1-9:
4
  | |O
 X| |
 X| |
O's turn:
Pick position 1-9:
7
 O| |O
 X| |
 X| |
X's turn:
Pick position 1-9:
3
 O| |O
 X| |
```

```
 X| |X
O's turn:
Pick position 1-9:
8
 O|O|O
 X| |
 X| |X
O wins
2
```

8. Implement unification in first order logic.

```python
def get_index_comma(string):
    """
    Return index of commas in string
    """

    index_list = list()
    # Count open parentheses
    par_count = 0

    for i in range(len(string)):
        if string[i] == ',' and par_count == 0:
            index_list.append(i)
        elif string[i] == '(':
            par_count += 1
        elif string[i] == ')':
            par_count -= 1

    return index_list


def is_variable(expr):
    """
    Check if expression is variable
    """

    for i in expr:
        if i == '(':
            return False

    return True


def process_expression(expr):
        # Remove space in expression
    expr = expr.replace(' ', '')
```

```
    # Find the first index == '('
    index = None
    for i in range(len(expr)):
        if expr[i] == '(':
            index = i
            break

    # Return predicate symbol and remove predicate symbol in expression
    predicate_symbol = expr[:index]
    expr = expr.replace(predicate_symbol, '')

    # Remove '(' in the first index and ')' in the last index
    expr = expr[1:len(expr) - 1]

    # List of arguments
    arg_list = list()

    # Split string with commas, return list of arguments
    indices = get_index_comma(expr)

    if len(indices) == 0:
        arg_list.append(expr)
    else:
        arg_list.append(expr[:indices[0]])
        for i, j in zip(indices, indices[1:]):
            arg_list.append(expr[i + 1:j])
        arg_list.append(expr[indices[len(indices) - 1] + 1:])

    return predicate_symbol, arg_list


def get_arg_list(expr):

    _, arg_list = process_expression(expr)

    flag = True
    while flag:
        flag = False

        for i in arg_list:
            if not is_variable(i):
                flag = True
                _, tmp = process_expression(i)
                for j in tmp:
                    if j not in arg_list:
                        arg_list.append(j)
                arg_list.remove(i)

    return arg_list
```

```
def check_occurs(var, expr):
    """
    Check if var occurs in expr
    """

    arg_list = get_arg_list(expr)
    if var in arg_list:
        return True

    return False


def unify(expr1, expr2):

    # Step 1:
    if is_variable(expr1) and is_variable(expr2):
        if expr1 == expr2:
            return 'Null'
        else:
            return False
    elif is_variable(expr1) and not is_variable(expr2):
        if check_occurs(expr1, expr2):
            return False
        else:
            tmp = str(expr2) + '/' + str(expr1)
            return tmp
    elif not is_variable(expr1) and is_variable(expr2):
        if check_occurs(expr2, expr1):
            return False
        else:
            tmp = str(expr1) + '/' + str(expr2)
            return tmp
    else:
        predicate_symbol_1, arg_list_1 = process_expression(expr1)
        predicate_symbol_2, arg_list_2 = process_expression(expr2)

        # Step 2
        if predicate_symbol_1 != predicate_symbol_2:
            return False
        # Step 3
        elif len(arg_list_1) != len(arg_list_2):
            return False
        else:
            # Step 4: Create substitution list
            sub_list = list()

            # Step 5:
```

```
            for i in range(len(arg_list_1)):
                tmp = unify(arg_list_1[i], arg_list_2[i])

                if not tmp:
                    return False
                elif tmp == 'Null':
                    pass
                else:
                    if type(tmp) == list:
                        for j in tmp:
                            sub_list.append(j)
                    else:
                        sub_list.append(tmp)

            # Step 6
            return sub_list


if __name__ == '__main__':
    # Data 1
    f1 = 'p(b(A), X, f(g(Z)))'
    f2 = 'p(Z, f(Y), f(Y))'

    # Data 2
    # f1 = 'Q(a, g(x, a), f(y))'
    # f2 = 'Q(a, g(f(b), a), x)'

    # Data 3
    # f1 = 'Q(a, g(x, a, d), f(y))'
    # f2 = 'Q(a, g(f(b), a), x)'

    result = unify(f1, f2)
    if not result:
        print('Unification failed!')
    else:
        print('Unification successfully!')
        print(result)
```

output:

```
Unification successfully!
['b(A)/Z', 'f(Y)/X', 'g(Z)/Y']
```

9. Write a program to implement a knowledge base consisting of first order logic statements and prove the query using forward reasoning.

```
from __future__ import generators
import re
import agents
from utils import *
class KB:
  def __init__(self, sentence=None):
      abstract

  def tell(self, sentence):
        "Add the sentence to the KB"
      abstract

  def ask(self, query):
      try:
            return self.ask_generator(query).next()
      except StopIteration:
            return False
  def ask_generator(self, query):
        "Yield all the substitutions that make query true."
      abstract

  def retract(self, sentence):
        "Remove the sentence from the KB"
      abstract
class PropKB(KB):
    "A KB for Propositional Logic.  Inefficient, with no indexing."

    def __init__(self, sentence=None):
        self.clauses = []
        if sentence:
            self.tell(sentence)

    def tell(self, sentence):
        "Add the sentence's clauses to the KB"
        self.clauses.extend(conjuncts(to_cnf(sentence)))

    def ask_generator(self, query):
        "Yield the empty substitution if KB implies query; else False"
        if not tt_entails(Expr('&', *self.clauses), query):
            return
        yield {}

    def retract(self, sentence):
        "Remove the sentence's clauses from the KB"
```

```
        for c in conjuncts(to_cnf(sentence)):
            if c in self.clauses:
                self.clauses.remove(c)


class KB_Agent(agents.Agent):
    """A generic logical knowledge-based agent. [Fig. 7.1]"""
    def __init__(self, KB):
        t = 0
        def program(percept):
            KB.tell(self.make_percept_sentence(percept, t))
            action = KB.ask(self.make_action_query(t))
            KB.tell(self.make_action_sentence(action, t))
            t = t + 1
            return action
        self.program = program

    def make_percept_sentence(self, percept, t):
        return(Expr("Percept")(percept, t))

    def make_action_query(self, t):
        return(expr("ShouldDo(action, %d)" % t))

    def make_action_sentence(self, action, t):
        return(Expr("Did")(action, t))
class Expr:
   def __init__(self, op, *args):
        "Op is a string or number; args are Exprs (or are coerced to
Exprs)."
        assert isinstance(op, str) or (isnumber(op) and not args)
        self.op = num_or_str(op)
        self.args = map(expr, args) ## Coerce args to Exprs

    def __call__(self, *args):
        """Self must be a symbol with no args, such as
Expr('F').  Create a new
        Expr with 'F' as op and the args as arguments."""
        assert is_symbol(self.op) and not self.args
        return Expr(self.op, *args)

    def __repr__(self):
        "Show something like 'P' or 'P(x, y)', or '~P' or '(P | Q |
R)'"
        if len(self.args) == 0: # Constant or proposition with arity 0
            return str(self.op)
        elif is_symbol(self.op): # Functional or Propositional operator
            return '%s(%s)' % (self.op, ', '.join(map(repr,
self.args)))
        elif len(self.args) == 1: # Prefix operator
```

```
            return self.op + repr(self.args[0])
        else: # Infix operator
            return '(%s)' % (' '+self.op+' ').join(map(repr,
self.args))

    def __eq__(self, other):
        """x and y are equal iff their ops and args are equal."""
        return (other is self) or (isinstance(other, Expr)
            and self.op == other.op and self.args == other.args)

    def __hash__(self):
        "Need a hash method so Exprs can live in dicts."
        return hash(self.op) ^ hash(tuple(self.args))
```