Basics of Python

```
spam_amount = 0
print (spam_amount)
spam_amount = spam_amount + 4
if spam_amount > 0:
    print ("But I don't want any spam")
viking_song = "spam" * spam_amount
print (viking_song)
```

o/p :- 0
But I don't want any spam
spam spam spam spam

- = :- assignment operator
- we need not declare what type of value spam_amount is going to refer to.

```
type (spam_amount)
```
o/p: int

```
type (19.95)
```
o/p: float

- a // b ⇒ Quotient of a and b, removing fractional part
- a ** b ⇒ a raised to power of b.
- -a ⇒ negative of a

Eg:- print (5/2) — o/p: 2.5
print (5//2) — o/p: 2

⊛ PEDMAS :- Paranthesis,
Exponents,
Multiplication / Division,
Addition / Subtraction.

⊛ Built-in funtions :-

print (max (1, 2, 3))
o/p : 3

print (min (1, 2, 3))
o/p : 1

print (abs (-1))
o/p :- 1

⊛ print (float (10)) ⇒ o/p :- 10.0
print (int (3.33)) ⇒ 3
print (int ('807') + 1) → 808

PAGE EDGE
Date : / /

1 ⊛ write python program program using if-else statement for age

2 ⊛ Print integers in reverse order

3 ⊛ Print pattern 1, 2, 2, 3, 3, 3, 4, 4, 4, 4 ....

4 ⊛ Print pattern 1, 1, 2, 1, 2, 3, 1, 2, 3, 4 ....

5 ⊛ Power of a number $\Rightarrow x^n$

6 ⊛ Switch case problem.

Soln :-

1 ⊛

```
age = int (input ('type your age :'))
if age <13 :
    print ('You are a child')
elif age >=13 and age <18 :
    print ('You are a teenager')
else:
    print ('You are an adult')
```

2 ⊛

```
c = 0
number = int (input ('enter the number'))
while number != 0 :
        a = number % 10
        c = c * 10 + a
        number = number // 10
print (c)
```

o/p :- enter the number 1234
4321

3 ⊛
```
n = int(input('enter the number'))
    for i in range(1, n+1):
        for j in range(i):
            print(i, end=' ')
```

o/p : enter the number 4
1 2 2 3 3 3 4 4 4 4

4 ⊛
```
n = int(input('enter the number'))
for i in range(1, n+1):
    for j in range(1, i+1):
        print(j, end=', ')
```

o/p :- enter the number 4
1, 1, 2, 1, 2, 3, 1, 2, 3, 4

5 ⊛
```
n = int(input('enter the base'))
m = int(input('enter the exponent'))
result = n**m
print(f'power is {result}')
```

o/p : enter the base 2
enter the exponent 7
power is 128

⑦ lang = input ("What programming language would you like to learn")

match lang:
case "JS"
print ("You can become web developer")

case "PHP"
print ("You can become backend developer")

case _ :
print ("Sorry")

10/11/23

Tic-Tac-Toe game

```
board = [' '
' for x in
range(10)]

    def insertLetter(letter, pos):
        board[pos] = letter

    def spaceIsFree(pos):
        return board[pos] == ' '

    def printBoard(board):
        print('   |   |')
        print(' ' + board[1] + ' | ' + board[2] +
                    ' | ' + board[3])
        print('   |   |')
        print('-----------')
        print('   |   |')
        print(' ' + board[4] + ' | ' + board[5] +
                    ' | ' + board[6])
        print('   |   |')
        print('-----------')
        print('   |   |')
        print(' ' + board[7] + ' | ' + board[8] +
                    ' | ' + board[9])
        print('   |   |')

    def isWinner(bo, le):
        return (bo[7] == le and bo[8] == le
and bo[9] == le) or (bo[4] == le and
bo[5] == le and bo[1] == le) or
```

```
(bo[1] == le and bo[2] == le and bo[3] == le) or
        (bo[1] == le and bo[4] == le and bo[7] == le) or
(bo[2] == le and bo[5] == le and bo[8] == le) or
        (bo[3] == le and bo[6] == le and bo[9] == le) or
(bo[1] == le and bo[5] == le and bo[7] == le)

    def playerMove():
    run = True

while run:
    move = input('Please select a position to
      place an \'X\' (1-9):')
    try:
        move = int(move)
        if move > 0 and move < 10
            if spaceIsFree(move):
            run = False
            insertLetter('X', move)
        else:
            print('Sorry this space is occupied!')
        else:
            print('Please type a number within
                                the range!')
        except:
            print('Please type a number!')

    def compMove():
        possibleMoves = [x for x, letter in
enumerate(board) if letter == ' ' and x != 0]
        move = 0
```

Algorithm :-

① Initialize the board

```
board = [' ' for _ in range(10)]
```

② Insert a letter ('X' or 'O') into the board

```
def insertLetter(letter, pos):
```

③ Check if space on the board is free :

```
def spaceIsFree(pos):
    return board[pos] == ' '
```

④ print the board

```
def printBoard(board):
```

⑤ Check for a Winner

```
def isWinner(bo, le):
```

⑥ Player's move

```
def playerMove():
    Ask the player to choose a position
    to place their 'X' on the board.
```

⑦ Computer's Move

Implement the logic for the computers move

```
def compMove
```

PAGE EDGE
Date : / /

8- Puzzle using BFS:-

Algo :-

1. Initialize puzzle:
   - Create an instance of the 'Puzzle' class.
   - The initial board is generated randomly.

2. Loop until puzzle is solved:
   - While the puzzle is not solved (determined by the 'is_solved' method):
     - Display the current state of the board using 'print_board'.
     - Prompt the user to enter a move direction (up, down, left, right).
     - Call the 'move' method to attempt the move specified by the user.

3. Move method:
   - Get the index of the blank tile (0) on the board.
   - If the specified direction is valid:
     - Swap the blank tile with the adjacent tile in the specified direction.
     - If the move is invalid, print an error message.

4. Check for Valid Move:
   - Ensure the move direction is valid based on the current position of the blank tile.
   - For example, if moving 'up', check that the blank tile is not in the top row.

5. Check for puzzle solved:
- The 'is_solved' method compares the current board state with the goal state.
- If they match, the puzzle is considered solved.

6. User Interaction:
- The user interacts by entering move directions during each iteration of the loop.
- The loop continues until the puzzle is solved.

7. End Of Puzzle:
- Once the loop exits (puzzle is solved), display a congratulatory message.

Ex: i/p:-

1	2	3
4	5	0
6	7	8

o/p:-

1	2	3
4	5	6
7	8	9

The moves are [down, left, left, up, right, down, right, up, left, down, right, right].

PAGE EDGE
Date : / /

⊛ 8- Puzzle Iteration deepning search algorithm :-

1. Define PuzzleNode class :
- Each node represents a state of the puzzle
- The state is a 3×3 grid
- Store the parent node, the action taken to reach the current state, the depth and other necessary information.

2. Define helper function :
- 'get_blank_position (state)' : Find the position of the blank (0) in the puzzle.
- 'get_neighbours (node)' : Get valid neighbours for a given state.
- 'apply_action (state, action)' : Apply a given action to a state and return the new state
- 'print_solution (solution)' : Print the solution path.

3. Depth-limited search function :
- 'depth_limited_search (node, goal_state, depth_limit)' : ~~form~~ Perform a depth-limited search starting from the given node.
- If the goal state is found, return the node.
- If the depth limit is reached, return None.
- For each valid action, generate a child node and recursively call the function.
- If a solution is found in one of the branches, return the solution.

4. Iterative Deepening Search Function:

- 'iterative_deepening_search(initial_state, goal_state)': Perform iterative deepening search.
- Initialize depth_limit to 0.
- Loop until a solution is found:
  - Create a PuzzleNode for the initial state.
  - Call 'depth_limited_search' with the current depth limit.
  - 'apply_action(state, action)': Apply a given action to a state and return the new state.
  - 'print_solution(solution)': Print the solution path.
  - If a solution is found, print the solution and return the result.
  - Increment the depth limit and repeat the process

5. Main Function:

- Define the initial state of the puzzle and the goal state.
- Call 'iterative_deepening_search' with the initial and goal states.

PAGE EDGE
Date: / /

⑧ 8-Puzzle problem using the A* search algo

1. Define PuzzleNode class:
- Each node represents a state of the puzzle
- The state is a 3x3 grid.
- Store the parent node, the action taken to reach the current state, the cost to reach the current state, and the heuristic value.

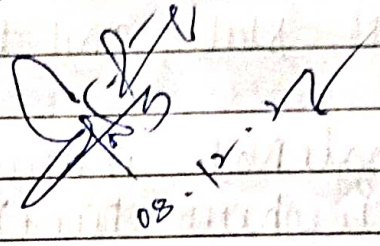
2. Define PuzzleNode methods:
- 'calculate_heuristic()': Calculate the heuristic value for the current state. In this example, we use Manhattan distance heuristic.
- '__lt__(self, other)': Define a comparison method for the priority queue in A* based on the total cost

3. Define helper functions:
- 'get_blank_position(state)': Find the position of the blank (0) in the puzzle.
- 'get_neighbors(node)': Get valid neighbors for a given state.
- apply_action(state, action): Apply a given action to a state and return the new state.

4. A star Algorithm*
- Initialize the initial state as a PuzzleNode
- Initialize a priority queue (min heap) to store nodes.
- Initialize a set to store explored states

5 Print solution:
- Print the actions & states along the path.

6 Main function:
- Define the initial state of the puzzle
- Call the A* algo with initial state.



PAGE EDGE
Date : / /

Vacuum Cleaner

Algorithm :-

function REFLEX-VACUUM-AGENT (location, status)
returns action

if status = Dirty then return Suck
else if location = A then return Right
else if location = B then return left

CODE :-

```
def reflex_agent(location, status):
if status == "Dirty":
return "Suck"
elif location == "A":
return "Right"
elif location == "B":
return "Left"

def vacuum_cleaner():
    location = input("Enter the current location (A or B): ")
    status = input("Enter the status of the location (Dirty or Clean): ")

    action = reflex_agent(location, status)
    print("Action:", action)

vacuum_cleaner()
```

CODE :-

```
def clean(floor):
i, j, row, col = 0, 0, len(floor), len(floor[0])
for i in range(row):
if(i%2) == 0:
for j in range(col):
 if (floor[i][j] == 1):
  print_f(floor, i, j) #clean()
else:
  for j in range(col-1, -1, -1):
   if (floor[i][j] == 1):
    print_f(floor, i, j)
    floor[i][j] = 0
  print_f(floor, i, j)
def print_f(floor, row, col):
  """A function to print the GRID, (row, col) represent
the current vacuum cleaner position"""
  print("The floor matrix is as below:")
    for r in range(len(floor)):
     for c in range(len(floor[r])):
       if r == row and c == col:
         print(f" >{floor[r][c]}<", end = '')
       else:
         print(f" {floor[r][c]} ", end = '')
      print(end='\n')
    print(end='\n')
  def main():
    floor = []
    m = int(input("Enter the No. of Rows: "))
    n = int(input("Enter the No. of Columns: "))
    print("Enter clean states for each cell (1-dirty,
0-clean)")
```

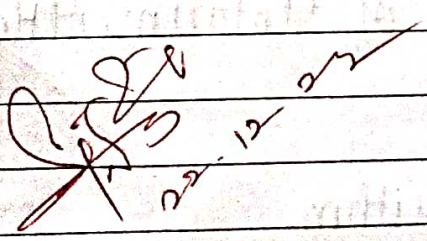
```
for i in range(m):
    f = list(map(int, input().split(" ")))
    floor.append(f)
print()
clean(floor)
main()
```

o/p :-

Enter no of rows = 1
no of columns = 2
clean status for each cell (1-dirty, 0-clean)
1 1

Floor matrix is as below:

>1< 1
clean
>0< 1
Right
0 >1<
clean
0 >0<
stay.



Knowledge Base Entailment

Pseudocode :

1. Tokenization :
   - Tokenize both 'sentence1' and 'sentence2' into individual words, considering punctuation and whitespace as delimiters.

2. Normalization :
   - Convert all words to lowercase to make the comparision case-insensitive.
   - Remove punctuation and handle variations in word forms.

3. Comparison :
   - Initialize a boolean variable, 'entailmentHolds' to true.
   - For each normalized word in 'sentence1':
     - Check if the word is present in the set of normalized words in 'sentence2'.
     - If any word is not found, set 'entailmentHolds' to false, exit the loop, and break.
   - Return the value of 'entailmentHolds' as the result.

4. End :
   - End the algorithm.

code :-

```
from sympy.logic.boolalg import Implies, Not
from sympy.abc import p, q

class PropositionalKnowledgeBase:
    def __init__(self):
        self.knowledge_base = set()

    def add_statement(self, statement):
        self.knowledge_base.add(statement)

    def check_entailment(self, query):
        return query.simplify() in self.knowledge_base

kb = PropositionalKnowledgeBase()

kb.add_statement(Implies(p, q))
kb.add_statement(Not(q))

query1 = p
query2 = Not(p)

result1 = kb.check_entailment(query1)
result2 = kb.check_entailment(query2)
```

o/p: Does 'p' logically follow from the KB?
True.

Knowledge base Resolution :-

Pseudocode :

Step 1 : Initialize Knowledge Base :

- Create a class knowledge Base to represent the KB
- Populate the KB with set of predefined questions & answers.

Step 2 : User interaction loop

- Enter a loop to interact with the user
- Prompt the user to ask a question or type exit to ~~the~~ end the program.

Step 3 : User input

- Receive user i/p for the question.

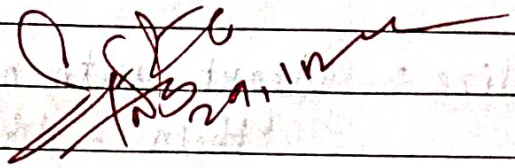
Step 4 : Knowledge Resolution algo

- Use the resolve_query method in kb class to find the answer to the user's question
- If an ans is found, display the ans

PAGE EDGE
Date : / /

Step 5 : Exit condition

If the user types exit, the loop and the prgm



Unification for first order logic :-

Algorithm :-

1. Initialize : Start with an empty substitution 'theta = {}'.

2. Comparision of Atoms : Compare the two atoms '["P", "x", "y"]' and '["P", "A", "B"]'.
   - The first elements "P" are the same.
   - The second elements '"x"' and '"A"' are different, so we need to unify them.

3. Unify variables : Unify the variable "x" with the term "A". Update 'theta' with this substitution : 'theta = {"x" : "A"}'.

4. Recursive Unification : Now, recursively unify the remaining elements:
   - For the third elements '"y"' and '"B"', unify the variable "y" with the term '"B"'. Update 'theta' : 'theta = {"x" : "A", "y" : "B"}'.

5. Result : The final substitution 'theta' is '{"x" : "A", "y" : "B"}'. The two expressions '{"P", "x", "y"}' and '["P", "A", "B"]' can be unified with this substitution.

code:-

```
class UnificationError (Exception):
    pass
def print_step (step, atom1, atom2, theta):
    print (f"\n Step { step}: ")
    print (f" Unifying {atom1} and {atom2} ")
    print (f" Current Substitution Theta: {theta}")

def unify_var (var, x, theta):
  if var in theta:
    return unify (theta [var], x, theta)
  elif x in theta:
    return unify (var, theta[x], theta)
  Use:
    theta[var] = x
    return theta

def unify (atom 1, atom 2, theta = None, step = 1):
  if theta is None:
    theta = { }

print_step (step, atom1, atom2, theta)

if atom1 == atom2:
return theta
elif isinstance (atom 1, str) and atom1. isalpha():
    return unify_var (atom1, atom2, theta)
elif isinstance (atom 1, list) and isinstance (atom2,
                                               list):
    for a1, a2 in zip (atom 1, atom 2:
        return theta
else:
```

raise UnificationError("cannot unify")

```
def print_step(step, atom1, atom2, theta)
    print(f"\n step {step}:")
    print(f"unifying {atom1} and {atom2}")
```

O/P :-

Step 1 :
Unifying ['P', 'x', 'y'] and ['P', 'A', 'B']
Initial substitution Theta : {}

Step 2 :
Unifying y and B
Current substitution Theta : {'x': 'A'}

Unification Successful. Substitution theta :
{'x': 'A', 'y': 'B'}

Convert a first order logic to conjunctive normal form

Algorithm:-

Step 1 : Eliminate Implications
Replace Implications with their equivalent forms using rule $p \rightarrow q \equiv \neg p \vee q$

Step 2 : Move negations inwards
Apply De morgan's law to move negation inwards.
For eg:
$\neg(p \wedge q) \equiv \neg p \vee \neg q$
$\neg(p \vee q) \equiv \neg p \wedge \neg q$

Step 3 : Distribute disjunctions over conjunctions

Step 4 : Convert to CNF form.

Code :

```
from sympy import symbols and, or, implies, not,
    to_cnf.

def eliminate_implications(formula):
    return formula.simplify()

def fol_to_cnf(fol_formula):
        formula_step1 = eliminate_implications
                        (fol_formula)
```

formula_step2 = move_negations
(formula_step1)

formula_step3 = distribute_disjunctions_over_
conjunctions (formula_step2)

~~formula_step4~~ =
cnf_formula = to_cnf (formula_step3)

return cnf_formula.

user_input = input ("enter a first order logic
formula:")
p, q, r = symbols ('p q r')
fol_formula = eval (user_input, {'p': p, 'q':
q, 'r': r})

cnf_formula = fol_to_cnf (fol_formula)
print ("\n FOL formula", fol_formula)
print ("CNF formula", cnf_formula)

Output:

Enter a first order logic: p & (~q | r)
FOL formula: And (p, Or (Not (q), r))
CNF formula: Or (And (p, r), and (p, ~q)) V
(∧ (p, r) ∧ (p, ~q))

PAGE EDGE
Date : / /

Write a KB consisting of FOL statements and prove the given query using forward chaining :-

Algorithm :-

1. Initialize knowledge base (KB) start with an empty knowledge base.

2. Add FOL statements to KB. Add relevant FOL statements to the knowledge base. Those statements represents facts and rules about the domain.

3. Define forward reasoning rules - Specify the rules for forward reasoning.

4. Initialize working memory. Create working memory to store the current state of the knowledge base during the reasoning process.

5. Ask Query. Formulate the query you want to prove.

6. Forward reasoning loop. Repeat the following steps until the query

   a. Iterate through each rule.
   b. Apply rules whose conditions match the current state.
   c. Add the conclusions of the applied.

7. Check Query in working memory. Verify if query is now be inferred.

8. Output result

If the query is proven output "Query is True otherwise output query is false".

Code :

```
class KnowledgeBase:
    def __init__(self):
        self.rules = []

def add_rule(self, conditions, conclusion):
    self.rules.append({"conditions": set(conditions),
                       "conclusion": conclusion})

def forward_reasoning(self, query):
    working_memory = set()
    unchanged = False

while not unchanged:
    unchanged = True
    for rule in self.rules:
        if rule['conditions'].issubset
(working_memory) and rule['conclusion'] not in
working_memory:

working_memory.add(rule['conclusion'])
unchanged = False

return query in working_memory.
```

```
kb = knowledgeBase()
kb.add_rule(["P", "Q"], "R")
kb.add_rule(["R", "S"], "T")
kb.add_rule(["T", "U"], "V")

query = "V"

result = kb.forward_reasoning(query)

if result:
    print("query is true")
else:
    print("Query is false")
```

O/P :-

query is false.