1.Develop a logistics strategy analogous to the missionaries and cannibals problem to transport goods using python.

Define allowed groups that represent valid combinations of goods that can be transported together.
Take input from the user by prompting them to enter items to be transported.
Clean and convert the input by stripping extra spaces, converting to lowercase, and storing the items in a set for easier comparison.
Check if the input matches any allowed combination by seeing if the set of input items is a subset of any predefined group.
Return the transport status by accepting or rejecting the goods based on whether they match an allowed combination.

```
class LogisticsChecker:
    def __init__(self):
        # Allowed combinations that can be transported together
        self.allowed_combinations = [
            {"electronics", "furniture"},
            {"food", "medicines"},
            {"chemicals", "cleaning_supplies"},
            {"toys", "books"}
        ]

    def can_transport(self, items):
        items_set = set(items)
        for combo in self.allowed_combinations:
            if items_set.issubset(combo):
                return "Accepted"
        return "Rejected"

# Example usage
items = input("Enter items (comma-separated): ").split(',')
items = [item.strip().lower() for item in items]

checker = LogisticsChecker()
print(checker.can_transport(items))
```

2.Develop a water distribution system for gated community villas using a water jug problem in python.

Calculate total water needs by summing the number of people in all villas.
Distribute the total water supply proportionally based on the number of people in each villa.
Determine available space by calculating how much more water each villa can hold (capacity minus current water).
Assign water to each villa by taking the minimum of the calculated share and the available space.
Display the results by printing the assigned water amount for each villa.

```
def water_distribution(total_cap, villas):
    total_people = sum(v[2] for v in villas)  # Calculate total people
    water_distribution = []

    for cap, current, people in villas:
        # Calculate the share of water for each villa based on people
        share = (people / total_people) * total_cap
        # Calculate available space in the villa
        available = cap - current
        # Assign the minimum of share or available space
        water_distribution.append((cap, min(share, available)))

    return water_distribution

# Input and initialization
total_cap = 10000
num_villas = int(input("Enter number of villas: "))
villas = []

# Input details for each villa
for i in range(num_villas):
    cap = int(input(f"Enter capacity for villa {i+1}: "))
    cur = int(input(f"Enter current water for villa {i+1}: "))
    people = int(input(f"Enter number of people in villa {i+1}: "))
    villas.append((cap, cur, people))

# Display results
for i, (cap, water) in enumerate(water_distribution(total_cap, villas)):
    print(f"Villa {i+1}: {int(water)} liters of water")
```

3 Develop network routing analogous to the 8 queen's problem for traffic monitoring in smart cities using python.

Initialize the board with each row set to an empty position (indicated by -1).
Check if placing a sensor in the current position (row, col) is safe by verifying there are no conflicts with previous sensors.
Recursively attempt to place sensors row by row, trying all columns in each row.
Backtrack by resetting the position if placing a sensor does not lead to a solution, and continue the search.
Display the result when a solution is found by printing the board or stating no solution exists.

```
def is_safe(board, row, col):
    for i in range(row):
        if board[i] == col or abs(board[i] - col) == abs(i - row):
            return False
    return True

def solve(board, row, n):
    if row == n:
        print_matrix(board, n)
        return True

    for col in range(n):
        if is_safe(board, row, col):
            board[row] = col
            if solve(board, row + 1, n):
                return True
            board[row] = -1
    return False

def print_matrix(board, n):
    matrix = [[0] * n for _ in range(n)]
    for row in range(n):
        matrix[row][board[row]] = 1
    for row in matrix:
        print(" ".join(map(str, row)))

def traffic_monitoring(n):
    board = [-1] * n
    if not solve(board, 0, n):
        print("No solution found.")
    else:
        print("Solution found:")
```

```
# Take input for the size of the grid (number of intersections/sensors)
n = int(input("Enter the number of sensors (n): "))
traffic_monitoring(n)
```

4. Develop a tourism management system, similar to the traveling salesman problem using python .

Initialize variables to track the shortest distance and the best route path.
Iterate through available routes to find those that include both the source and destination cities.
Check the order of cities to ensure the source city appears before the destination city in the route.
Calculate the distance by summing up the distances between the source and destination cities in the selected route.
Display the shortest route and its distance once the best route is found or print a message if no valid route exists.

```
routes_data = [
    {"cities": ["Chennai", "Tiruchirappalli", "Madurai", "Kanyakumari"], "distances": [300, 200,
250]},
    {"cities": ["Chennai", "Vellore", "Tirunelveli", "Kanyakumari"], "distances": [150, 350, 300]},
    {"cities": ["Chennai", "Coimbatore", "Trichy", "Kanyakumari"], "distances": [250, 180, 270]},
    {"cities": ["Chennai", "Bangalore", "Mysore", "Kanyakumari"], "distances": [350, 220, 310]}
]

def find_best_route(src, dest):
    shortest = float('inf')
    best_route_path = []

    for route in routes_data:
        if src in route["cities"] and dest in route["cities"]:
            start, end = route["cities"].index(src), route["cities"].index(dest)
            if start < end:
                distance = sum(route["distances"][start:end])
                path = route["cities"][start:end+1]
                if distance < shortest:
                    shortest, best_route_path = distance, path

    print(f"Shortest Route: {' -> '.join(best_route_path)}, Distance: {shortest} km")

# Input for source and destination
src, dest = input("Enter source and destination: ").split()
find_best_route(src, dest)
```

5. Develop logics for the pick and place robotic arms for the food industry using prolog.

Define the items with their respective locations and destinations (plates) for each item.
Pick the item by identifying its location and displaying the action of picking it up from there.
Place the item by identifying the corresponding destination (plate) and displaying the action of placing the item onto it.
Combine both actions into a single process where the robot picks the item from its location and places it on its destination.
Execute the process by calling the combined pick-and-place action for any specified item to perform the task.

https://swish.swi-prolog.org/ // to run ?- pick_and_place(apple).

```
% Facts about items, their locations, and their types
item(apple).
item(banana).
item(chicken).
item(potato).

location(apple, shelf1).
location(banana, shelf2).
location(chicken, shelf3).
location(potato, shelf4).

destination(shelf1, plate1).
destination(shelf2, plate2).
destination(shelf3, plate3).
destination(shelf4, plate4).

% Logic for picking an item
pick_item(Item) :-
    item(Item),
    location(Item, Location),
    write('Picking up '), write(Item), write(' from '), write(Location), nl.

place_item(Item) :-
    item(Item),
    destination(Location, Destination),
    write('Placing '), write(Item), write(' onto '), write(Destination), nl.

% Combine pick and place actions for the robot arm
pick_and_place(Item) :-
    pick_item(Item),
    place_item(Item).
```

6.Implement a Decision tree classifier to detect fraudulent transactions in the stock market using python.

Prepare the dataset by organizing the transaction details and encoding the target labels (fraud as 1, normal as 0).
Extract features and target variables, where features include transaction amount and user age, and the target is the transaction type.
Split the data into training and testing sets to ensure the model is evaluated on unseen data.
Train the model using a Decision Tree Classifier on the training data to learn patterns for predicting fraud.
Evaluate the model by making predictions on the test data, calculating accuracy, and displaying the confusion matrix for performance assessment.

```
# Import necessary libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, confusion_matrix

# Generate a sample dataset (this would be replaced by real transaction data)
data = {
    'Transaction_Amount': [1000, 5000, 200, 3000, 7000, 400, 600, 8000, 2500, 15000],
    'User_Age': [25, 45, 23, 37, 41, 28, 35, 50, 30, 48],
    'Transaction_Type': ['Normal', 'Fraud', 'Normal', 'Normal', 'Fraud', 'Normal', 'Normal', 'Fraud',
'Normal', 'Fraud'],
}

df = pd.DataFrame(data)

# Encode the 'Transaction_Type' (Fraud -> 1, Normal -> 0)
df['Transaction_Type'] = df['Transaction_Type'].map({'Normal': 0, 'Fraud': 1})

# Features (X) and target (y)
X = df[['Transaction_Amount', 'User_Age']]
y = df['Transaction_Type']

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Initialize and train the Decision Tree Classifier
clf = DecisionTreeClassifier(random_state=42)
clf.fit(X_train, y_train)

# Make predictions
```

```
y_pred = clf.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
conf_matrix = confusion_matrix(y_test, y_pred)

# Output results
print(f"Accuracy: {accuracy*100:.2f}%")
print(f"Confusion Matrix:\n{conf_matrix}")
```

7. Develop an expert system for healthcare application to detect diseases and recommend treatment using python.

Collect symptoms by taking input from the user, and convert the symptoms to lowercase for consistency.
Check for known symptom combinations to detect common diseases (like fever + cough for Flu, headache + nausea for Migraine).
Match symptoms with specific conditions like chest pain for Heart Attack or painful urination for UTI.
Return the disease name and appropriate treatment advice based on the matched symptom pattern.
Handle unknown cases by suggesting the user consult a doctor if no known disease is detected.

```python
def detect_disease(symptoms):
    if "fever" in symptoms and "cough" in symptoms:
        return "Flu", "Rest, Stay hydrated."
    elif "headache" in symptoms and "nausea" in symptoms:
        return "Migraine", "Pain relievers, Rest."
    elif "chest pain" in symptoms:
        return "Heart Attack", "Seek emergency care."
    elif "painful urination" in symptoms:
        return "UTI", "Antibiotics."
    else:
        return "Unknown", "Consult a doctor."

# User input
symptoms = input("Enter your symptoms (comma-separated): ").lower().split(',')

# Detect and recommend
disease, treatment = detect_disease([s.strip() for s in symptoms])

print(f"Disease: {disease}")
print(f"Treatment: {treatment}")
```