Digital assignment -1

ARTIFICIAL INTELLIGENCE

RISHIKESH M RAMASUBRAMANIYAN - 22MIA1163

1. Implement simple facts using First-order logic in python using simple game

Overview

- A text-based adventure game implemented in Python using **First-Order Logic principles** for room navigation, inventory management, and interactions with objects and monsters.
- The player explores rooms, collects items, defeats a monster, and attempts to find treasure.

Key Components

Data Structure Design

- **Room Class**:
  - Represents a room with attributes:
    - `name` : Name of the room.
    - `contains` : List of items or entities present in the room (e.g., player, torch, key, monster).
    - `connected` : List of rooms connected to this room.
    - `dark` : Indicates if the room is dark and requires a light source to enter.
    - `locked` : Indicates if the room is locked and requires a key to enter.
- **GameWorld Class**:
  - Manages the overall game state, rooms, inventory, and player interactions.
  - Attributes:
    - `rooms` : Dictionary of room objects indexed by their names.
    - `player_inventory` : List of items collected by the player.
    - `current_room` : The room where the player currently resides.
    - `monster_defeated` : Boolean flag indicating if the monster is defeated.

Game Logic

Initialization (`GameWorld.__init__`)

- Sets up a **world map** with three interconnected rooms:
  1. **Room 1**: Contains the player and a torch, not dark or locked.
  2. **Room 2**: Contains a key and a monster, dark by default.
  3. **Room 3**: Contains the treasure but is locked.
- Player starts with a **sword** in their inventory.

Core Functionalities

Screen Management

- **`clear_screen()`** :
  - Clears the console for better readability (platform-specific).

Room Visualization

- **`display_matrix()`** :
  - Displays the connections between rooms in a matrix format.
  - Uses arrows to indicate room connectivity.
- **`display_room_details()`** :
  - Displays the details of each room, including:
    - Name and status (e.g., DARK, LOCKED).
    - Items or entities contained in the room.

Inventory Management

- **`display_inventory()`** :

- Lists the items currently in the player's inventory.
- Uses symbols to represent items visually.

Full Game State

- `display_game_state()`:
  - Combines all visualizations (room matrix, room details, inventory) to display the current state of the game.

Player Actions

Move Player (`move_player`)

- Moves the player to a connected room if the following conditions are satisfied:
  - The target room exists.
  - The room is connected to the current room.
  - The room is **not locked**, or the player has the key.
  - The room is **not dark**, or the player has a torch.

Take Item (`take_item`)

- Allows the player to pick up an item from the current room if:
  - The item exists in the room.
  - The item is not an immovable entity (e.g., monster or player).
  - The monster (if present) has been defeated.

Attack Monster (`attack_monster`)

- Allows the player to defeat the monster in the current room if:
  - The monster is present.
  - The player has a sword in their inventory.
- Updates the game state to replace the monster with a "defeated monster" symbol.

Main Game Loop (`main`)

1. Continuously displays the game state and waits for player input.
2. Supports the following commands:
   - **Move**: `move <room>` - Move to a connected room.
   - **Take**: `take <item>` - Pick up an item in the current room.
   - **Attack**: `attack monster` - Attack the monster in the room.
   - **Quit**: `quit` - Exit the game.
3. Validates input and calls the corresponding methods.
4. Adds delays (`time.sleep`) to improve user experience.

Gameplay Flow

1. Player starts in Room 1 with a sword and torch.
2. Player moves between rooms to explore.
3. Player must:
   - Defeat the monster in Room 2 to obtain the key.
   - Use the key to unlock Room 3 and collect the treasure.
4. The game ends when the player quits.

Special Features

- **First-Order Logic Concepts**:
  - Logical dependencies between game entities (e.g., locked rooms require keys, dark rooms require torches).
  - State changes based on player actions (e.g., defeating the monster to access the key).
- **Symbols for Better Visuals**:
  - Uses emojis to represent items and entities (e.g., 👤 for player, 🔑 for key).
- **Dynamic Game State Updates**:
  - Rooms, inventory, and interactions are updated in real-time based on player actions.

```
# implement simple facts using First-order logic in python using simple game

import os
```

```
from typing import Dict, List, Set
from dataclasses import dataclass
import time

@dataclass
class Room:
    name: str
    contains: List[str]
    connected: List[str]
    dark: bool = False
    locked: bool = False

class GameWorld:
    def __init__(self):
        self.rooms: Dict[str, Room] = {
            'room1': Room('Room 1', ['player', 'torch'], ['room2'], dark=False),
            'room2': Room('Room 2', ['key', 'monster'], ['room1', 'room3'], dark=True),
            'room3': Room('Room 3', ['treasure'], ['room2'], dark=False, locked=True)
        }
        self.player_inventory: List[str] = ['sword']
        self.current_room = 'room1'
        self.monster_defeated = False

    def clear_screen(self):
        os.system('cls' if os.name == 'nt' else 'clear')

    def get_item_symbol(self, item: str) -> str:
        symbols = {
            'player': '👤',
            'key': '🔑',
            'monster': '👻',
            'treasure': '💎',
            'sword': '⚔️',
            'torch': '🔦',
            'defeated_monster': '💀'
        }
        return symbols.get(item, item)

    def display_matrix(self):
        """Display room connections in matrix format"""
        print("\n=== Room Connection Matrix ===")
        print("      ", end="")
        for room_id in self.rooms:
            print(f"{room_id:^7}", end="")
        print("\n" + "      " + "-------" * len(self.rooms))

        for from_room in self.rooms:
            print(f"{from_room:^5}", end="")
            for to_room in self.rooms:
                if to_room in self.rooms[from_room].connected:
                    print("   →   ", end="")
                elif from_room in self.rooms[to_room].connected:
                    print("   ←   ", end="")
                else:
                    print("   ·   ", end="")
            print()
        print()

    def display_room_details(self):
        """Display detailed information about each room"""
        print("\n=== Room Details ===")
        for room_id, room in self.rooms.items():
            status = []
            if room.dark:
                status.append("DARK")
            if room.locked:
                status.append("LOCKED")

            status_str = f" [{' | '.join(status)}]" if status else ""
            current = " *" if room_id == self.current_room else ""

            print(f"\n{room.name}{status_str}{current}")
            print("-" * 20)

            if room.contains:
                print("Contains:", end=" ")
                for item in room.contains:
                    print(f"{self.get_item_symbol(item)} {item}", end=" ")
                print()

    def display_inventory(self):
        """Display player's inventory"""
        print("\n=== Inventory ===")
```

```
        if self.player_inventory:
            for item in self.player_inventory:
                print(f"{self.get_item_symbol(item)} {item}", end=" ")
            print()
        else:
            print("Empty")

    def display_game_state(self):
        """Display the complete game state"""
        self.clear_screen()
        print("=== Game World Visualization ===")
        print("(Use commands: move <room>, take <item>, attack monster, quit)")
        self.display_matrix()
        self.display_room_details()
        self.display_inventory()
        print("\n" + "=" * 40)

    def move_player(self, target_room: str) -> bool:
        """Attempt to move player to target room"""
        if target_room not in self.rooms:
            print("\nInvalid room!")
            return False

        current_room = self.rooms[self.current_room]
        target = self.rooms[target_room]

        if target_room not in current_room.connected:
            print("\nRooms are not connected!")
            return False

        if target.locked and 'key' not in self.player_inventory:
            print("\nRoom is locked and you don't have the key!")
            return False

        if target.dark and 'torch' not in self.player_inventory:
            print("\nRoom is dark and you don't have a light source!")
            return False

        # Removed the monster check - you can enter a room with a monster now

        # Move player
        current_room.contains.remove('player')
        target.contains.append('player')
        self.current_room = target_room

        # If there's a monster in the room, warn the player
        if 'monster' in target.contains and not self.monster_defeated:
            print("\nWarning: There's a monster in here! You should deal with it!")

        return True

    def take_item(self, item: str) -> bool:
        """Attempt to take item from current room"""
        current_room = self.rooms[self.current_room]
        if item not in current_room.contains:
            print("\nItem not found in this room!")
            return False

        if item in ['player', 'monster', 'defeated_monster']:
            print("\nYou can't take that!")
            return False

        if item == 'key' and not self.monster_defeated:
            print("\nThe monster is guarding the key! Defeat it first!")
            return False

        current_room.contains.remove(item)
        self.player_inventory.append(item)
        return True

    def attack_monster(self) -> bool:
        """Attack the monster if possible"""
        current_room = self.rooms[self.current_room]

        if 'monster' not in current_room.contains:
            print("\nThere's no monster here to attack!")
            return False

        if 'sword' not in self.player_inventory:
            print("\nYou need a sword to attack the monster!")
            return False
```

```
            # Remove monster and add defeated_monster
            current_room.contains.remove('monster')
            current_room.contains.append('defeated_monster')
            self.monster_defeated = True
            return True

def main():
    game = GameWorld()

    while True:
        game.display_game_state()

        command = input("\nEnter command: ").lower().split()
        if not command:
            continue

        if command[0] == 'quit':
            print("\nThanks for playing!")
            break

        if command[0] == 'attack':
            if len(command) == 2 and command[1] == 'monster':
                if game.attack_monster():
                    print("\nYou defeated the monster!")
                time.sleep(1)
                continue
            else:
                print("\nInvalid command! Use: attack monster")
                time.sleep(1)
                continue

        if len(command) < 2:
            print("\nInvalid command! Use: move <room>, take <item>, or attack monster")
            time.sleep(1)
            continue

        action, target = command[0], command[1]

        if action == 'move':
            if game.move_player(target):
                print(f"\nMoved to {target}!")
            time.sleep(1)

        elif action == 'take':
            if game.take_item(target):
                print(f"\nTook {target}!")
            time.sleep(1)

        else:
            print("\nInvalid command!")
            time.sleep(1)

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

```
=== Game World Visualization ===
(Use commands: move <room>, take <item>, attack monster, quit)

=== Room Connection Matrix ===
      room1  room2  room3
     --------------------
room1   ·      →      ·
room2   →      ·      →
room3   ·      →      ·


=== Room Details ===

Room 1 *
--------------------
Contains: 👤 player 🔦 torch

Room 2 [DARK]
--------------------
Contains: 🔑 key 👻 monster

Room 3 [LOCKED]
--------------------
Contains: 💎 treasure

=== Inventory ===
⚔️ sword

========================================
```

```
Enter command: TAKE TORCH

Took torch!
=== Game World Visualization ===
(Use commands: move <room>, take <item>, attack monster, quit)

=== Room Connection Matrix ===
      room1  room2  room3
     --------------------
room1   ·      →      ·
room2   →      ·      →
room3   ·      →      ·


=== Room Details ===

Room 1 *
--------------------
Contains: 👤 player

Room 2 [DARK]
--------------------
Contains: 🔑 key 👻 monster

Room 3 [LOCKED]
--------------------
Contains: 💎 treasure
```

2. Solving block world puzzle using planning techniques in python

Overview

- The program solves the **Block World Puzzle** using **state-space search** and **planning techniques**.
- It moves blocks between stacks or to a table to achieve a desired goal configuration from an initial configuration.

Key Components

BlockWorld Class

- Manages the block world puzzle, including the initial and goal states, and provides methods to perform actions and check solutions.

Class Attributes

- **`initial_state`**:
  - The starting configuration of blocks as a list of stacks.
  - Example: `[['A', 'B'], ['C'], []]` means:
    - Stack 1: Blocks `A` and `B` (B is on top of A).
    - Stack 2: Block `C`.
    - Stack 3: Empty.
- **`goal_state`**:
  - The desired block configuration.
  - Example: `[['A'], ['B', 'C'], []]`.
- **`max_table_size`**:
  - The maximum number of stacks allowed on the table.
- **`current_state`**:
  - Tracks the current configuration during the solving process.

Key Methods

1. `is_goal_state(self, state)`

- Compares the current state with the goal state.
- Returns `True` if the current state matches the goal state.

---

2. `get_possible_actions(self, state)`

- Generates a list of possible actions (moves) for a given state.
- Actions are tuples of the form `(block, from_stack, to_stack)`:
  - `block`: The block being moved.

  - `from_stack`: The stack where the block is moved from.
  - `to_stack`: The destination stack or `-1` (indicates moving to the table).
- Includes:
  - Moving a block from one stack to another.
  - Moving a block to the table (if allowed).

---

3. `apply_action(self, state, action)`

- Applies a given action to a state and returns the resulting new state.
- Steps:
  1. Removes the block from the source stack.
  2. Adds the block to the destination stack or the table (creates a new stack on the table if space permits).
  3. If moving to the table and the maximum allowed stacks are reached, the move is invalid (`None` is returned).

---

4. `display_state(self, state)`

- Visualizes the current block configuration.
- Steps:
  - Creates a 2D matrix to represent the stacks.
  - Prints each row of the matrix, showing blocks at their respective positions.
  - Uses `.` to represent empty spaces.

---

5. `solve(self)`

- Implements a **Breadth-First Search (BFS)** algorithm to find the solution:
  - Uses a queue to store states and the sequence of actions leading to those states.
  - Tracks visited states to avoid revisiting configurations.
  - Steps:
    1. Start with the initial state.
    2. For each state, generate all possible valid actions.
    3. Apply actions to create new states and add them to the queue.
    4. If the goal state is reached, return the sequence of actions.
    5. If the queue is empty and no solution is found, return `None`.

Features

- **Breadth-First Search**:
  - Ensures the shortest solution is found (minimum moves).
- **State Representation**:
  - Uses tuples to track visited states efficiently.
- **Visualization**:
  - Displays the block configurations after each move.

```python
# solving block world puzzle using planning techniques in python

class BlockWorld:
    def __init__(self, initial_state, goal_state, max_table_size):
        self.initial_state = initial_state
        self.goal_state = goal_state
        self.current_state = initial_state
        self.max_table_size = max_table_size

    def is_goal_state(self, state):
        return state == self.goal_state

    def get_possible_actions(self, state):
        actions = []
        for i, stack in enumerate(state):
            if stack:  # If the stack is not empty
                block = stack[-1]  # Get the top block
                # Move block to another stack
                for j in range(len(state)):
                    if i != j:  # Can't move to the same stack
                        actions.append((block, i, j))  # (block, from_stack, to_stack)
                # Move block to the table (if not already on the table)
                actions.append((block, i, -1))  # -1 indicates the table
        return actions
```

```
    def apply_action(self, state, action):
        block, from_stack, to_stack = action
        new_state = [stack.copy() for stack in state]  # Deep copy of current state
        new_state[from_stack].pop()  # Remove block from the from_stack
        if to_stack == -1:
            # Place block on the table (create a new stack if necessary)
            if len(new_state) - 1 < self.max_table_size:  # Check if we can add a new stack
                new_state.append([block])  # Create a new stack for the table
            else:
                # If the table is full, we cannot move the block to the table
                return None
        else:
            new_state[to_stack].append(block)  # Move block to the to_stack
        return new_state

    def display_state(self, state):
        max_height = max(len(stack) for stack in state)  # Find the maximum height of stacks
        matrix = [['.' for _ in range(len(state))] for _ in range(max_height)]  # Create a matrix filled with dots

        for col, stack in enumerate(state):
            for row in range(len(stack)):
                matrix[max_height - row - 1][col] = stack[row]  # Fill the matrix with blocks

        for row in matrix:
            print(' '.join(row))  # Print each row of the matrix
        print()  # Print a newline for better readability

    def solve(self):
        from collections import deque
        queue = deque([(self.initial_state, [])])  # (state, actions)
        visited = set()

        while queue:
            state, actions = queue.popleft()
            if tuple(map(tuple, state)) in visited:  # Check if state has been visited
                continue
            visited.add(tuple(map(tuple, state)))

            self.display_state(state)  # Display the current state

            if self.is_goal_state(state):
                return actions  # Return the sequence of actions to reach the goal

            for action in self.get_possible_actions(state):
                new_state = self.apply_action(state, action)
                if new_state is not None:  # Only add valid states
                    queue.append((new_state, actions + [action]))

        return None  # No solution found


# Example usage
initial_state = [['A', 'B'], ['C'], []]  # Initial configuration
goal_state = [['A'], ['B', 'C'], []]  # Desired configuration
max_table_size = 2  # Maximum number of stacks allowed on the table

block_world = BlockWorld(initial_state, goal_state, max_table_size)
solution = block_world.solve()

if solution:
    print("Solution found:")
    for action in solution:
        block, from_stack, to_stack = action
        if to_stack == -1:
            print(f"Move {block} to the table")
        else:
            print(f"Move {block} from stack {from_stack} to stack {to_stack}")
else:
    print("No solution found.")
```

```
B . .
A C .

. B .
A C .

A C B

C . .
B . .
A . .
```

```
B . .
A . C

. A .
. B .
. C .

. B .
. C A

. A .
. C B

. . A
. C B

C . .
A . B

. . C
A . B

A B C

. . B
A . C

B C A

. . B
. C A

. A .
B C .

. B .
. A .
. C .

. . A
C . B

. . C
. . A
. . B
```