**RIPHAH INTERNATIONAL UNIVERSITY, ISLAMABAD**

****

**Lab 9**

**Bachelors of Computer science – 6th semester**

**Subject:** Artificial Intelligence Lab

**Submitted to:** Ma’am Ayesha

**Submitted by:** Munaza Malik

**Sap Id:** 44956

**Date:** 10th April, 2025

**LAB TASKS**

**Question 01:**

Write a Python code using object-oriented classes to make a Simple Reflex Agent, as we have been doing in class. Below are the tasks that the agent must perform.

You manage a casino where a probabilistic game of wages is played. There are ‘n’ players and ‘n’ cards involved in this game. A card is mapped to a player based on the outcome of a roll of dice with-‘n’-faces. The problem is, you want to save as much money as you want therefore you want to fire your employee who hosts this game and rolls the dice. To save money, make an AI agent to replace the casino employee who performs the tasks below to host the game.

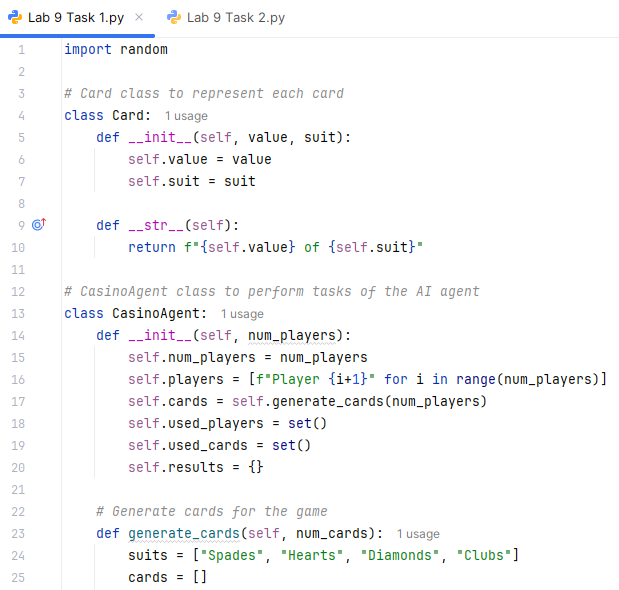
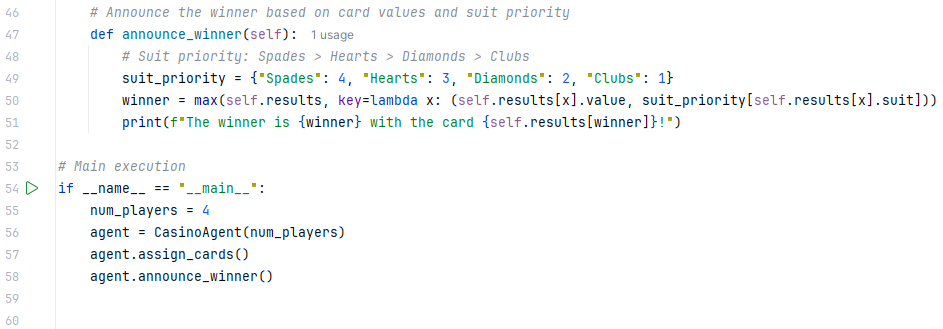
1. Identify number of the contestants
2. Add a same number of cards to the game
3. Perform the roll of two dice; one for players and the other for cards.
4. As per the value of the rolls, assign the corresponding card to the corresponding player.
5. Once a card is assigned to a players, both are invalid if the rolls of dice calls them again.
6. Announce the winner – the player with the highest card value, as per the legend below.

a. Cards Legend:

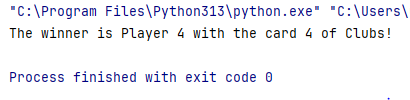
1. The bigger the number, the higher the priority
2. Spades > Hearts > Diamonds > Clubs

**Solution:**

import random  
  
*# Card class to represent each card*class Card:  
 def \_\_init\_\_(self, value, suit):  
 self.value = value  
 self.suit = suit  
  
 def \_\_str\_\_(self):  
 return f"{self.value} of {self.suit}"  
  
*# CasinoAgent class to perform tasks of the AI agent*class CasinoAgent:  
 def \_\_init\_\_(self, num\_players):  
 self.num\_players = num\_players  
 self.players = [f"Player {i+1}" for i in range(num\_players)]  
 self.cards = self.generate\_cards(num\_players)  
 self.used\_players = set()  
 self.used\_cards = set()  
 self.results = {}  
  
 *# Generate cards for the game* def generate\_cards(self, num\_cards):  
 suits = ["Spades", "Hearts", "Diamonds", "Clubs"]  
 cards = []  
 for i in range(num\_cards):  
 suit = suits[i % len(suits)]  
 cards.append(Card(i + 1, suit))  
 return cards  
  
 *# Roll a dice with n faces* def roll\_dice(self):  
 return random.randint(0, self.num\_players - 1)  
  
 *# Assign cards to players based on dice rolls* def assign\_cards(self):  
 while len(self.results) < self.num\_players:  
 player\_index = self.roll\_dice()  
 card\_index = self.roll\_dice()  
  
 if player\_index not in self.used\_players and card\_index not in self.used\_cards:  
 self.results[self.players[player\_index]] = self.cards[card\_index]  
 self.used\_players.add(player\_index)  
 self.used\_cards.add(card\_index)  
  
 *# Announce the winner based on card values and suit priority* def announce\_winner(self):  
 *# Suit priority: Spades > Hearts > Diamonds > Clubs* suit\_priority = {"Spades": 4, "Hearts": 3, "Diamonds": 2, "Clubs": 1}  
 winner = max(self.results, key=lambda x: (self.results[x].value, suit\_priority[self.results[x].suit]))  
 print(f"The winner is {winner} with the card {self.results[winner]}!")  
  
*# Main execution*if \_\_name\_\_ == "\_\_main\_\_":  
 num\_players = 4  
 agent = CasinoAgent(num\_players)  
 agent.assign\_cards()  
 agent.announce\_winner()

**Output:**



**Question 02:**

Write a program that includes the three different types of agents and their implementation in different scenarios.

* The program includes the implementation of goal-based agents.
* The program includes the implementation of the model-based agent.
* The program includes the implementation of a utility-based agent.

**Solution:**

# Base environment: a 1D grid with locations A, B, and C

environment = {

'A': 'dirty',

'B': 'clean',

'C': 'dirty'

}

# --------- 1. Goal-Based Agent ---------

def goal\_based\_agent(location):

print("\n--- Goal-Based Agent ---")

if environment[location] == 'dirty':

print(f"Location {location} is dirty. Cleaning it...")

environment[location] = 'clean'

else:

print(f"Location {location} is already clean.")

# --------- 2. Model-Based Agent ---------

model = {}

def model\_based\_agent(location):

print("\n--- Model-Based Agent ---")

perception = environment[location]

model[location] = perception

if model[location] == 'dirty':

print(f"Model: {location} is dirty. Cleaning it...")

environment[location] = 'clean'

model[location] = 'clean'

else:

print(f"Model: {location} is clean. Nothing to do.")

# --------- 3. Utility-Based Agent ---------

def utility(location):

return 1 if environment[location] == 'clean' else 0

def utility\_based\_agent():

print("\n--- Utility-Based Agent ---")

best\_location = max(environment.keys(), key=utility)

print(f"Most useful location to go: {best\_location}")

if environment[best\_location] == 'dirty':

print(f"Cleaning {best\_location} for high utility...")

environment[best\_location] = 'clean'

else:

print(f"{best\_location} is already clean. Utility is high.")

# --------- Run all Agents ---------

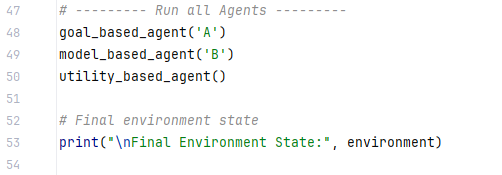
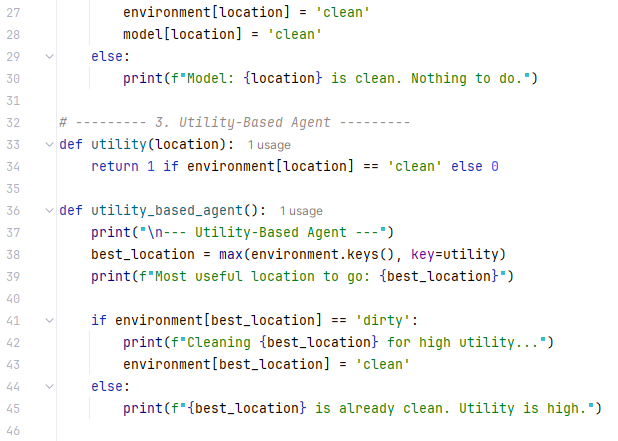
goal\_based\_agent('A')

model\_based\_agent('B')

utility\_based\_agent()

# Final environment state

print("\nFinal Environment State:", environment)



**Output:**

