Artificial Intelligence

AI-2002

Lab 02

Agents and Environment

National University of Computer & Emerging Sciences – NUCES – Karachi

National University of Computer & Emerging Sciences - NUCES - Karachi

FAST School of Computing

Course Code: AI-2002	Artificial Intelligence Lab

1. Objective

1. Introduction to Agents in different Environments, exposing students to different AI Problems and solutions.
2. Types of Agents and the reasons to use them in different environments.
3. Solve some basic AI problems using the python programming language.

2. Introduction to Agents

An intelligent agent (IA) is an entity that makes a decision that enables artificial intelligence to be put into action. It can also be described as a software entity that conducts operations in the place of users or programs after sensing the environment. It uses actuators to initiate action in that environment.

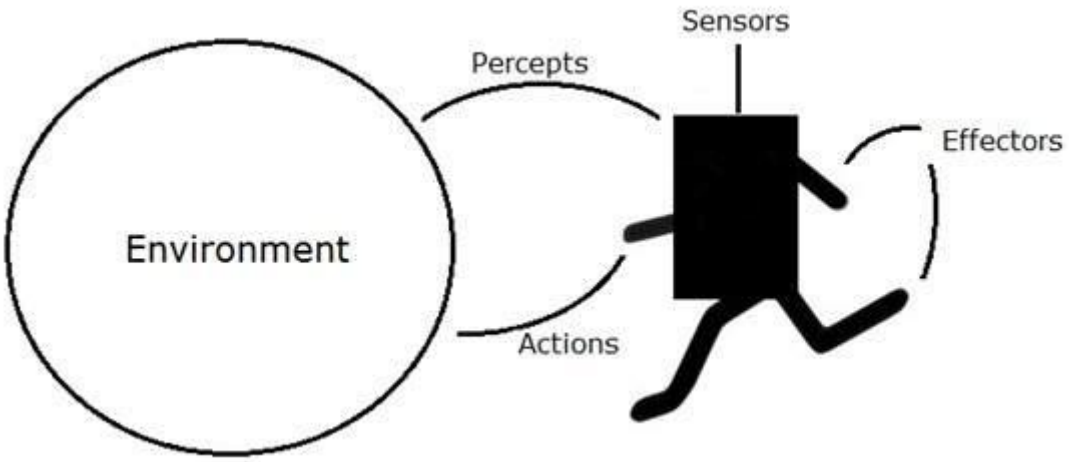
- This agent has some level of autonomy that allows it to perform specific, predictable, and repetitive tasks for users or applications.
- It's also termed as 'intelligent' because of its ability to learn during the process of performing tasks.
- The two main functions of intelligent agents include perception and action. Perception is done through sensors while actions are initiated through actuators.
- Intelligent agents consist of sub-agents that form a hierarchical structure. Lower-level tasks are performed by these sub-agents.
- The higher-level agents and lower-level agents form a complete system that can solve difficult problems through intelligent behaviors or responses.

2.1 How intelligent agents work

Intelligent agents work through three main components: sensors, actuators, and effectors. Getting an overview of these components can improve our understanding of how intelligent agents work.

- **Sensors:** These are devices that detect any changes in the environment. This information is sent to other devices. In artificial intelligence, the environment of the system is observed by intelligent agents through sensors.
- **Actuators:** These are components through which energy is converted into motion. They perform the role of controlling and moving a system. Examples include rails, motors, and gears.

- **Effectors:** The environment is affected by effectors. Examples include legs, fingers, wheels, display screen, and arms.



2.2 Examples of Agents

- **A Software agen**t has Keystrokes, file contents, received network packages which act as sensors and displays on the screen, files, sent network packets acting as actuators. A Human-agent has eyes, ears, and other organs which act as sensors, and hands, legs, mouth, and other body parts acting as actuators.
- **A Robotic agen**t has Cameras and infrared range finders which act as sensors and various motors acting as actuators.

2.3 Relating AI-Based Agents to Human Behavior

Artificial Intelligence (AI) agents are designed to mimic human decision-making processes, often drawing parallels with how people act and respond in daily life. **Reflex agents** in AI, like humans' instinctive reactions, respond immediately to stimuli without deeper thought, such as an automated braking system in cars that stops when an obstacle is detected. **Model-based agents** use stored knowledge to make informed decisions, similar to how people rely on experience, like checking the weather before dressing for the day. **Goal-based agents** are comparable to humans setting objectives and taking steps to achieve them, such as planning a trip or saving for a vacation. **Utility-based agents** evaluate multiple factors to optimize outcomes, just as individuals balance cost,

quality, and convenience when choosing products or services. Lastly, **learning agents** improve performance over time by gathering feedback, much like humans learn from mistakes and experiences to refine their skills. This relationship highlights how AI systems emulate human cognitive functions to solve problems, automate tasks, and enhance decision-making in real-world applications.

highlights how AI systems emulate human cognitive functions to solve problems, automate tasks, and enhance decision-making in real-world applications.

Real-World Behaviors Reflecting AI Agent Principles

1. Reflex Agent (Immediate, automatic responses)

- Pulling your hand away after touching a hot object.
- Blinking when something moves rapidly toward your face.
- Ducking when a ball is thrown unexpectedly at you.
- Swerving your car to avoid an obstacle on the road.
- Sneezing when exposed to dust or allergens.

2. Model-Based Agent (Using past knowledge to inform decisions)

- Looking outside or checking the weather app before dressing for the day.
- Turning on headlights when driving in a tunnel because you know it will be dark.
- Remembering a store's layout to quickly find an item on a return visit.
- Closing windows when it starts to rain based on previous experiences.
- Adjusting room temperature by using the thermostat when feeling cold or hot.

3. Goal-Based Agent (Actions aimed at achieving specific goals)

- Planning a study schedule to pass an upcoming exam.
- Organizing a shopping list to save time and avoid unnecessary purchases.
- Mapping a route to reach a new location efficiently.
- Scheduling workouts to achieve fitness goals.
- Saving money for a specific purpose, like a vacation or a car.

4. Utility-Based Agent (Maximizing overall satisfaction or utility)

- Choosing between two jobs by considering salary, location, and work-life balance.
- Selecting a phone by weighing features like camera quality, battery life, and price.

- Picking a restaurant based on food quality, ambiance, and cost.
- Deciding which movie to watch by balancing reviews and your mood.
- Choosing between walking or driving based on distance, time, and weather conditions.

5. Learning Agent (Improving actions based on feedback)

- Learning to bake a cake better after receiving feedback from previous attempts.
- Adjusting study techniques after noticing which methods improve test scores.
- Refining a presentation style after audience feedback.
- Improving navigation skills after getting lost and finding a better route.
- Learning a new language through practice and correcting mistakes over time.

2.3 Working of Agents

When designing any intelligent agent system, it is essential to create three main components:

1. **The Environment:**
    - The external system with which the agent interacts. It provides percepts (sensory inputs) and responds to the agent's actions.

```
class Environment:
    def __init__(self, initial_state):
        self.initial_state = initial_state #Initial state could be
fixed or random

    def get_percept(self):
      #initial condition of environment , that would be perceived
by agent
      pass
```

2. **The Agent:**
    - The decision-making entity that perceives its environment and takes actions to achieve a goal.

```
class SimpleReflexAgent:
    def __init__(self):
        pass

    def act(self, percept):
        #Determine action based on the initial percept
        pass
```

3. **The Agent Program:**
    - The internal mechanism that determines the actions the agent takes based on its precepts.

This interaction allows the agent to move within the environment and perform its functions effectively. The following example demonstrates these components with a simple reflex agent.

```
def run_agent(agent, environment):
    # The agent reacts to the initial stimulus/Percept
    percept = environment.get_percept()
    action = agent.act(percept)
    print(f"Percept: {percept}, Action: {action}")
```

```
# Create instances of agent and environment
agent = SimpleReflexAgent()
```

```
environment = Environment(initial_state=0)  # Start with any
initial condition (high/low , 1/0, True/False , high/med/low etc
based on scenario)

# Run the agent in the environment (only once)
run_agent(agent, environment)
```

The given code illustrates a basic framework for agent-environment interaction, where the behavior of both the environment and the agent can be easily modified. If we update the initial conditions in the Environment class, it will directly affect the get_percept function, as the agent perceives the environment based on these initial conditions.

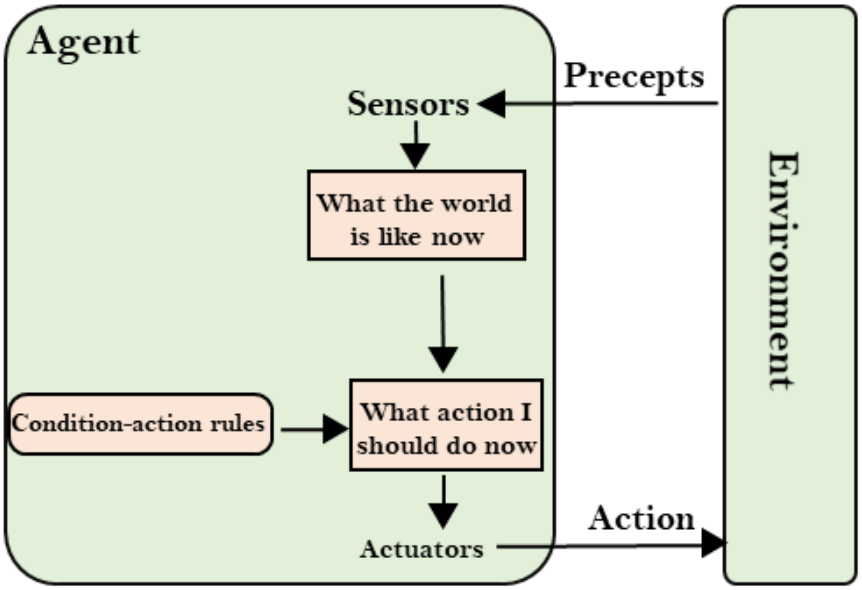
- Furthermore, the environment can be extended to operate in a 2D space, allowing the addition of obstacles, various terrains, or dynamic elements. These enhancements can make the simulation more realistic and complex.
- On the agent side, the behavior and decision-making strategy (e.g., goal-based, reflex-based, or utility-based) can be modified by updating the logic within the act method of the agent's class. For instance, a goal-based agent would require state tracking and goal prioritization, while a utility-based agent would involve calculating the best outcome.

This flexible structure enables experimentation with different agent strategies and diverse environment configurations.

3. Types of Agents

3.1 Simple Reflex Agent

Click Here (Learn about Simple Reflex Agent)



3.1.1 Simple Reflex Agent Code Example #1 (Reflex-Based Hand-Pulling Agent:)

This agent simulates a human-like reflex action when encountering a hot object. The agent perceives the environment to check if the object is hot and reacts accordingly. If the object is hot, the agent pulls its hand away immediately. If the object is cool, the agent keeps its hand near the object. The agent operates purely on the current percept without memory or prediction, performing actions based solely on the immediate stimulus.

This agent simulates a human-like reflex action when encountering a hot object. The agent perceives the environment to check if the object is hot and reacts accordingly. If the object is hot, the agent pulls its hand away immediately. If the object is cool, the agent keeps its hand near the object. The agent operates purely on the current percept without memory or prediction, performing actions based solely on the immediate stimulus.

Environment Class: Environment

Represents the environment containing the hot object, which can either be "Hot" or "Cool."

- **heat_level**: Tracks the temperature state of the object, which can be "High" (Hot) or "Low" (Cool).
- **get_percept()**: Returns the current state of the object as either "Hot" or "Cool".

```
class Environment:
    def __init__(self, heat_level='High'):
        self.heat_level = heat_level

    def get_percept(self):
        """Return the heat level of the object as the percept."""
        return 'Hot' if self.heat_level == 'High' else 'Cool'
```

Agent Class: SimpleReflexAgent

Decides actions based on percepts (the current state of the environment, specifically the heat of the object).

- **act(percept)**:
  - If the percept is "Hot", the agent performs a reflex action to "Pull hand away."
  - If the percept is "Cool", the agent keeps its hand near the object.

```
class SimpleReflexAgent:
    def __init__(self):
        pass

    def act(self, percept):
        """Determine action based on the percept (heat level)."""
        if percept == 'Hot':
            return 'Pull hand away, you touched the hot object'
        else:
            return 'You have not touched any hot object , No need to
pull away'
```

Simulation Function: run_agent(agent, environment)

Simulates the interaction between the agent and the environment. The agent performs an action based on the current state of the environment (whether the object is hot or cool).

- **Steps**: The simulation is a one-time interaction between the agent and the environment. The agent reacts to the heat stimulus once, either pulling the hand away or keeping it near the object.

```
def run_agent(agent, environment):
    # The agent reacts to the heat stimulus only once
    percept = environment.get_percept()
    action = agent.act(percept)
    print(f"Percept: {percept}, Action: {action}")


# Create instances of agent and environment
agent = SimpleReflexAgent()
environment = Environment(heat_level='Low')  # Start with a hot object

# Run the agent in the environment (only once)
run_agent(agent, environment)
```

Working

The Environment provides the state of an object that can either be "Hot" or "Cool." The agent, upon perceiving the object's state:

- **If the object is hot**: The agent pulls its hand away.
- **If the object is cool**: The agent keeps its hand near the object.

The simulation function triggers this reflex action once, as a real-life human would react to the hot object immediately and not repeatedly.

3.1.2 Simple Reflex Agent Code Example #2 (Vacuum Cleaner)

Design a reflex-based cleaning agent that perceives the cleanliness of a room. If the room is dirty, the agent cleans it. If the room is already clean, it does nothing except confirming the cleanliness. The agent does not have memory or predictive capabilities and reacts solely to the current percept.

Environment Class: Environment

Represents the state of the environment, which can be either "Dirty" or "Clean".

- state: Tracks the room's cleanliness status.
- get_percept(): Returns the current state of the room.
- clean_room(): Changes the state to "Clean".

```
class Environment:
    def __init__(self, state='Dirty'):
        self.state = state

    def get_percept(self):
        return self.state

    def clean_room(self):
        self.state = 'Clean'
```

Agent Class: SimpleReflexAgent

Decides actions based on percepts (the current state of the environment).

- act(percept):
    - Returns "Clean the room" if percept is "Dirty".
    - Returns "Room is already clean" if percept is "Clean".

```
class SimpleReflexAgent:
    def __init__(self):
        pass

    def act(self, percept):
        if percept == 'Dirty':
            return 'Clean the room'
        else:
            return 'Room is already clean'
```

Simulation Function: run_agent(agent, environment, steps)

- Simulates interactions between the agent and the environment for a specified number of steps.
- Prints the percept and action for each step.

```
def run_agent(agent, environment, steps):
    for step in range(steps):
        percept = environment.get_percept()
        action = agent.act(percept)
        print(f"Step {step + 1}: Percept - {percept}, Action - {action}")
        if percept == 'Dirty':
            environment.clean_room()


# Create instances of agent and environment
agent = SimpleReflexAgent()
environment = Environment()

# Run the agent in the environment for 5 steps
run_agent(agent, environment, 5)
```

Working

The **Environment** provides the state of a 1D room, which can either be **"Dirty"** or **"Clean"**. The **Agent** reacts to this percept:

- If **"Dirty",** it cleans the room, updating the state to **"Clean".**
- If **"Clean"**, it does nothing but acknowledges the room's cleanliness.

The **Simulation Function** repeatedly checks the room's state and triggers the agent's action for a fixed number of steps. Initially, the room is dirty; hence, the agent cleans it. For the remaining steps, since the room is already clean, the agent only confirms the cleanliness without further actions.

Practice Task # 1

A. Modify the code so that the initial state of the environment ('Dirty' or 'Clean') is randomly set when the program starts. The state should be chosen at the beginning of the program and should vary each time it runs.
B. Modify the code to make the environment's state change randomly after each step. The state should switch between 'Dirty' and 'Clean' after each iteration of the loop or after the agent performs an action. This will simulate a dynamic environment where the state is unpredictable throughout the agent's execution.

3.1.3 Simple Reflex Agent Code Example #3 (2D Grid-Based Vacuum Cleaner Simulation: Smart Cleaning Robot)

We have a 3x3 grid (labeled from 'a' to 'i'), and the vacuum cleaner agent is tasked with cleaning the dirty spots in the grid. The initial state of the grid is as follows:

The grid looks like this:

a(0)	b(1)	c(2)
d(3)	e(4)	f(5)
g(6)	h(7)	i(8)

- The agent starts at position 'a' (position 0).

A 👽	b	c
d	e	f
g	h	i

Environment Class: Environment
Represents the grid-based environment where the agent operates.

- **Attributes:**
  - grid: A list representing a 3x3 grid with "Clean" and "Dirty" states.
    - Initial state: ['Clean', 'Dirty', 'Clean', 'Clean', 'Dirty', 'Dirty', 'Clean', 'Clean', 'Clean'].
- **Methods:**
  - get_percept(position):
    - Returns the state ("Clean" or "Dirty") of the given grid position.
  - clean_room(position):

- Sets the state of the room at the given position to "Clean".
- display_grid(agent_position):
  - Displays the grid in a 3x3 layout, showing the agent's current position with " 👽 ".

```
class Environment:
    def __init__(self):
        # Create a 3x3 grid, where 'b', 'e', and 'f' are dirty
        self.grid = ['Clean', 'Dirty', 'Clean',
                     'Clean', 'Dirty', 'Dirty',
                     'Clean', 'Clean', 'Clean']

    def get_percept(self, position):
        # Return the state of the current position
        return self.grid[position]

    def clean_room(self, position):
        # Clean the room at the given position
        self.grid[position] = 'Clean'

    def display_grid(self, agent_position):
        # Display the current state of the grid in a 3x3 format
        print("\nCurrent Grid State:")
        grid_with_agent = self.grid[:]  # Copy the grid
        grid_with_agent[agent_position] = "👽"  # Place the agent at
the current position
        for i in range(0, 9, 3):
            print(" | ".join(grid_with_agent[i:i + 3]))
        print()  # Extra line for spacing
```

Agent Class: SimpleReflexAgent

The agent decides actions based on percepts (the current state of the environment).

- **Attributes:**
  - position: Keeps track of the agent's current position on the grid (starting at position 0).
- **Methods:**

- **act(percept, grid):**
  - Returns "Clean the room" if the percept is "Dirty".
  - Updates the grid to "Clean" at the current position.
  - Returns "Room is clean" if the percept is "Clean".
- **move():**
  - Moves the agent to the next position (right to left, top to bottom).
  - Stops at the last position (index 8).

```
class SimpleReflexAgent:
    def __init__(self):
        self.position = 0  # Start at 'a' (position 0 in the grid)

    def act(self, percept, grid):
        # If the current position is dirty, clean it
        if percept == 'Dirty':
            grid[self.position] = 'Clean'
            return 'Clean the room'
        else:
            return 'Room is clean'

    def move(self):
        # Move to the next position in the grid
        if self.position < 8:
            self.position += 1
        return self.position
```

Simulation Function: run_agent(agent, environment, steps)
Simulates the agent's operation in the environment for a specified number of steps.

- **Parameters:**
  1. agent: An instance of SimpleReflexAgent.
  2. environment: An instance of Environment.
  3. steps: Number of steps the simulation runs (9 steps cover the entire grid).
- **Process:**
  1. The agent perceives the state of the current position.
  2. It performs an action based on the percept ("Clean the room" or "Room is clean").
  3. The grid is displayed showing the agent's current position.

4. The agent moves to the next position.

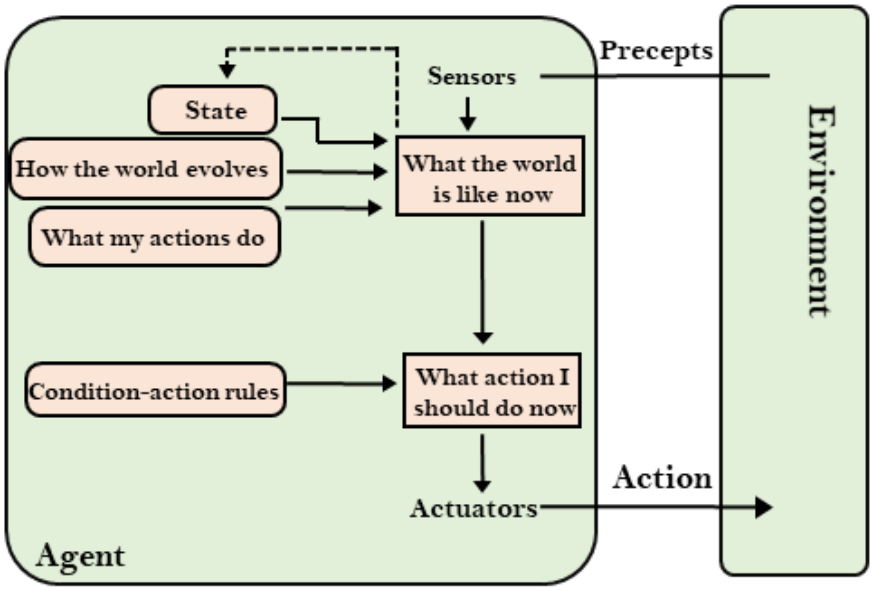
```
def run_agent(agent, environment, steps):
    for step in range(steps):
        percept = environment.get_percept(agent.position)
        action = agent.act(percept, environment.grid)
        print(f"Step {step + 1}: Position {agent.position} -> Percept
- {percept}, Action - {action}")
        environment.display_grid(agent.position)  # Display the grid
state with agent position

        if percept == 'Dirty':
            environment.clean_room(agent.position)

        agent.move()
# Create instances of agent and environment
agent = SimpleReflexAgent()
environment = Environment()

# Run the agent in the environment for 9 steps (to cover the 3x3 grid)
run_agent(agent, environment, 9)
```

Output

```
Step 1: Position 0 -> Percept - Clean, Action - Room is clean

Current Grid State:
🤖 | Dirty | Clean
Clean | Dirty | Dirty
Clean | Clean | Clean

Step 2: Position 1 -> Percept - Dirty, Action - Clean the room

Current Grid State:
Clean | 🤖 | Clean
Clean | Dirty | Dirty
Clean | Clean | Clean

Step 3: Position 2 -> Percept - Clean, Action - Room is clean

Current Grid State:
Clean | Clean | 🤖
Clean | Dirty | Dirty
Clean | Clean | Clean

Step 4: Position 3 -> Percept - Clean, Action - Room is clean

Current Grid State:
Clean | Clean | Clean
🤖 | Dirty | Dirty
Clean | Clean | Clean

Step 5: Position 4 -> Percept - Dirty, Action - Clean the room

Current Grid State:
Clean | Clean | Clean
Clean | 🤖 | Dirty
Clean | Clean | Clean

Step 6: Position 5 -> Percept - Dirty, Action - Clean the room

Current Grid State:
Clean | Clean | Clean
Clean | Clean | 🤖
Clean | Clean | Clean
```

3.2 Model Based Agent

Click Here (Learn about Model Based Agent)



3.2.1 Model Based Agent Code Example #1 (Vacuum Cleaner)

```
class ModelBasedAgent:
    def __init__(self):
        self.model = {}
 # d= {'current': clean}
    def update_model(self, percept):
        self.model['current'] = percept
        print(self.model)

    def predict_action(self):
        if self.model['current'] == 'Dirty':
            return 'Clean the room'
        else:
            return 'Room is clean'

    def act(self, percept):
        self.update_model(percept)
        return self.predict_action()
```

```
class Environment:
    def __init__(self, state='Dirty'):
        self.state = state

    def get_percept(self):
        return self.state

    def clean_room(self):
        self.state = 'Clean'


def run_agent(agent, environment, steps):
    for step in range(steps):
        percept = environment.get_percept()
        action = agent.act(percept)
        print(f"Step {step + 1}: Percept - {percept}, Action -
{action}")
        if percept == 'Dirty':
            environment.clean_room()


# Create instances of agent and environment
agent = ModelBasedAgent()
environment = Environment()

# Run the agent in the environment for 5 steps
run_agent(agent, environment, 5)
```

Output	Step 1: Percept - Dirty, Action - Clean the room Step 2: Percept - Clean, Action - Room is clean Step 3: Percept - Clean, Action - Room is clean Step 4: Percept - Clean, Action - Room is clean Step 5: Percept - Clean, Action - Room is clean

3.2.2 Model Based Agent Code Example #2 (Closing Window When it Starts to Rain)

In this task, the agent needs to make a decision to close the windows based on its past experiences. The agent will use a model of the environment to make informed decisions. It will remember whether the windows are open or closed, and it will also track whether it has previously encountered rain. This model allows the agent to predict the future state of the environment based on past experiences.

Environment Class: Environment

Represents the state of the environment, which can be affected by weather conditions (rain) and the state of the windows (open or closed).

- **rain**: Tracks whether it is currently raining ("Yes" or "No").
- **windows_open**: Tracks whether the windows are open ("Open" or "Closed").
- **get_percept()**: Returns the percept, which is the current state of rain and window status.
- **close_windows()**: Changes the state of the windows to "Closed".

```
class Environment:
    def __init__(self, rain='No', windows_open='Open'):
        self.rain = rain
        self.windows_open = windows_open

    def get_percept(self):
        """Returns the current percept (rain status and window
status)."""
        return {'rain': self.rain, 'windows_open': self.windows_open}

    def close_windows(self):
        """Closes the windows if they are open."""
        if self.windows_open == 'Open':
            self.windows_open = 'Closed'
```

Agent Class: ModelBasedAgent

Decides actions based on a model of the environment, using both current and past experiences. The agent keeps track of whether it has previously encountered rain and whether the windows were open.

- **act(percept)**: The agent checks if it is raining and whether the windows are open:
  - If it is raining and the windows are open, the agent will close the windows.
  - If it is not raining, the agent will do nothing but acknowledge that no action is needed.
- The agent uses a **model** to keep track of its past experiences with rain and window states.

```
class ModelBasedAgent:
    def __init__(self):
        self.model = {'rain': 'No', 'windows_open': 'Open'}

    def act(self, percept):
        """Decides action based on the model and current percept."""
        # Update the model with the current percept
        self.model.update(percept)

        # Check the model to decide action
        if self.model['rain'] == 'Yes' and self.model['windows_open']
== 'Open':
            return 'Close the windows'
        else:
            return 'No action needed'
```

Agent Class: ModelBasedAgent

Decides actions based on a model of the environment, using both current and past experiences. The agent keeps track of whether it has previously encountered rain and whether the windows were open.

- **act(percept)**: The agent checks if it is raining and whether the windows are open:
  - If it is raining and the windows are open, the agent will close the windows.

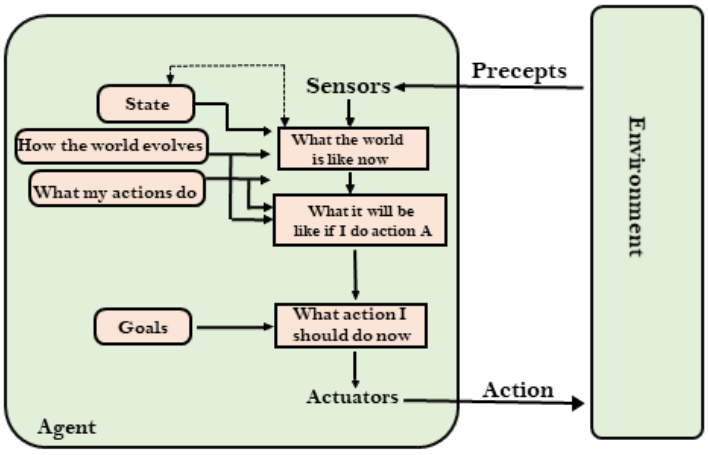
- If it is not raining, the agent will do nothing but acknowledge that no action is needed.

- The agent uses a **model** to keep track of its past experiences with rain and window states.

```
def run_agent(agent, environment, steps):
    for step in range(steps):
        # Get the current percept from the environment
        percept = environment.get_percept()

        # Agent makes a decision based on the current percept
        action = agent.act(percept)

        # Print the current percept and the agent's action
        print(f"Step {step + 1}: Percept - {percept}, Action -
{action}")

        # If the agent decided to close the windows, update the
environment
        if action == 'Close the windows':
            environment.close_windows()

# Create instances of agent and environment
agent = ModelBasedAgent()
environment = Environment(rain='Yes', windows_open='Open')  # It's
raining and windows are open

# Run the agent in the environment for 5 steps
run_agent(agent, environment, 5)
```

3.3 Goal Based Agent

Click Here (Learn about Goal Based Agent)



3.3.1 Goal Based Agent Code Example #1 (From Scratch)

```
class GoalBasedAgent:
    def __init__(self):
        self.goal = 'Clean'

    def formulate_goal(self, percept):
        if percept == 'Dirty':
            self.goal = 'Clean'
        else:
            self.goal = 'No action needed'

    def act(self, percept):
        self.formulate_goal(percept)
        if self.goal == 'Clean':
            return 'Clean the room'
        else:
            return 'Room is clean'


class Environment:
    def __init__(self, state='Dirty'):
```

```
        self.state = state

    def get_percept(self):
        return self.state

    def clean_room(self):
        self.state = 'Clean'


def run_agent(agent, environment, steps):
    for step in range(steps):
        percept = environment.get_percept()
        action = agent.act(percept)
        print(f"Step {step + 1}: Percept - {percept}, Action -
{action}")
        if percept == 'Dirty':
            environment.clean_room()


# Create instances of agent and environment
agent = GoalBasedAgent()
environment = Environment()

# Run the agent in the environment for 5 steps
run_agent(agent, environment, 5)
```

Output	Step 1: Percept - Dirty, Action - Clean the room Step 2: Percept - Clean, Action - Room is clean Step 3: Percept - Clean, Action - Room is clean Step 4: Percept - Clean, Action - Room is clean Step 5: Percept - Clean, Action - Room is clean

3.4 Utility Based Agent

Click Here (Learn about Utility Based Agent)

3.4.1 Utility Based Agent Code Example #1 (Vacuum Cleaner)

```
class UtilityBasedAgent:
    def __init__(self):
        self.utility = {'Dirty': -10, 'Clean': 10}

    def calculate_utility(self, percept):
        return self.utility[percept]

    def select_action(self, percept):
        if percept == 'Dirty':
            return 'Clean the room'
        else:
            return 'No action needed'

    def act(self, percept):
        action = self.select_action(percept)
        return action


class Environment:
    def __init__(self, state='Dirty'):
        self.state = state

    def get_percept(self):
        return self.state

    def clean_room(self):
        self.state = 'Clean'


def run_agent(agent, environment, steps):
    total_utility = 0
    for step in range(steps):
```

```
        percept = environment.get_percept()
        action = agent.act(percept)
        utility = agent.calculate_utility(percept)
        print(f"Step {step + 1}: Percept - {percept}, Action -
{action}, Utility - {utility}")
        total_utility += utility
        if percept == 'Dirty':
            environment.clean_room()
    print("Total Utility:", total_utility)


# Create instances of agent and environment
agent = UtilityBasedAgent()
environment = Environment()

# Run the agent in the environment for 5 steps
run_agent(agent, environment, 5)
```

Output

```
Step 1: Percept - Dirty, Action - Clean the room, Utility - -
10
Step 2: Percept - Clean, Action - No action needed, Utility -
10
Step 3: Percept - Clean, Action - No action needed, Utility -
10
Step 4: Percept - Clean, Action - No action needed, Utility -
10
Step 5: Percept - Clean, Action - No action needed, Utility -
10
Total Utility: 30
```

3.4.2 Utility Based Agent Code Example #1 (Choosing a Movie to Watch)

In this task, the agent chooses a movie to watch by considering multiple factors, such as reviews and mood. A utility-based agent uses a utility function to evaluate different

options and select the one that maximizes overall satisfaction. This agent balances between positive reviews and the current mood to make an optimal decision.

Environment Class: Environment

Represents the environment that provides a set of movies and their respective reviews.

- **movies**: A dictionary where keys are movie titles and values are review scores (e.g., ratings from 1 to 10).
- **get_percept()**: Returns the list of movies and their review scores.

```
class Environment:
    def __init__(self, rain='No', windows_open='Open'):
        self.rain = rain
        self.windows_open = windows_open

    def get_percept(self):
        """Returns the current percept (rain status and window
status)."""
        return {'rain': self.rain, 'windows_open': self.windows_open}

    def close_windows(self):
        """Closes the windows if they are open."""
        if self.windows_open == 'Open':
            self.windows_open = 'Closed'
```

Agent Class: UtilityBasedAgent

Decides which movie to watch by using a utility function that balances the review score and a mood factor.

- **mood_factor**: Represents how much mood influences the choice (a float between 0 and 1). Higher values indicate stronger mood influence.
- **utility(movie, review)**: Computes utility by combining review scores and mood. For simplicity, we assume:
    - Utility = (review score) * (mood factor)
- **act(percept)**: Chooses the movie with the highest utility value.

```
class UtilityBasedAgent:
    def __init__(self, mood_factor=0.7):
        self.mood_factor = mood_factor

    def utility(self, review):
        """Compute utility based on review score and mood factor."""
        return review * self.mood_factor

    def act(self, percept):
        """Decides which movie to watch based on utility."""
        best_movie = None
        best_utility = -float('inf')

        for movie, review in percept.items():
            movie_utility = self.utility(review)
            if movie_utility > best_utility:
                best_movie = movie
                best_utility = movie_utility

        return best_movie
```

Simulation Function: run_agent(agent, environment)

Simulates a single decision by the agent based on the available movies and their reviews.

- **Steps**: Since this is a one-time decision, there's no loop required.

```
def run_agent(agent, environment):
    percept = environment.get_percept()
    best_choice = agent.act(percept)
    print(f"Available Movies: {percept}")
    print(f"Best Movie to Watch: {best_choice}")


# Create instances of agent and environment
agent = UtilityBasedAgent(mood_factor=0.8)
environment = Environment({'Movie A': 7, 'Movie B': 9, 'Movie C': 5})
```

```
# Run the agent in the environment
run_agent(agent, environment)
```

3.5 Learning Based Agent

Click Here (Learn about Learning Based Agent)

3.5.1 Learning Based Agent Code Example

```
import random

class LearningBasedAgent:
    def __init__(self, actions):
        self.Q = {}
        self.actions = actions
        self.alpha = 0.1  # Learning rate
        self.gamma = 0.9  # Discount factor
        self.epsilon = 0.1  # Exploration rate

    def get_Q_value(self, state, action):
        return self.Q.get((state, action), 0.0)

    def select_action(self, state):
        if random.uniform(0, 1) < self.epsilon:
            return random.choice(self.actions)
        else:
            return max(self.actions, key=lambda a:
self.get_Q_value(state, a))

    def learn(self, state, action, reward, next_state):
        old_Q = self.get_Q_value(state, action)
        best_future_Q = max([self.get_Q_value(next_state, a) for a in
self.actions])
        self.Q[(state, action)] = old_Q + self.alpha * (reward +
self.gamma * best_future_Q - old_Q)

    def act(self, state):
        action = self.select_action(state)
```

```
        return action

class Environment:
    def __init__(self, state='Dirty'):
        self.state = state

    def get_percept(self):
        return self.state

    def clean_room(self):
        self.state = 'Clean'
        return 10

    def no_action_reward(self):
        return 0

def run_agent(agent, environment, steps):
    for step in range(steps):
        percept = environment.get_percept()
        action = agent.act(percept)
        if percept == 'Dirty':
            reward = environment.clean_room()
            print(f"Step {step + 1}: Percept - {percept}, Action -
{action}, Reward - {reward}")
        else:
            reward = environment.no_action_reward()
            print(f"Step {step + 1}: Percept - {percept}, Action -
{action}, Reward - {reward}")
        next_percept = environment.get_percept()
        agent.learn(percept, action, reward, next_percept)


# Create instances of agent and environment
agent = LearningBasedAgent(['Clean the room', 'No action needed'])
environment = Environment()

# Run the agent in the environment for 5 steps
run_agent(agent, environment, 5)
```

Lab Tasks

Task # 1

A company's security system consists of nine critical components (A through I). Each component can be either safe or vulnerable due to security flaws. A security agent is responsible for scanning the entire system, identifying vulnerabilities, and patching them to prevent attacks.

You are tasked with simulating a cybersecurity exercise using the given environment and agent. Complete the following steps:

- **Initial System Check:**
  - The system environment is initialized with random vulnerabilities. Display the initial state of the system, showing which components are safe and which are vulnerable.
- **System Scan:**
  - The security agent scans each component. If a component is vulnerable, the agent logs a warning and adds it to a list for patching. If it is secure, a success message is logged.
- **Patching Vulnerabilities:**
  - After the scan, the agent patches all vulnerable components, marking them as safe. Display messages indicating which components have been patched.
- **Final System Check:**
  - Display the system's final state, confirming that all vulnerabilities have been patched.

Task # 2

A data center runs multiple servers, each hosting different services. These services are either underloaded, balanced, or overloaded based on the system's load. The Load Balancer Agent is responsible for redistributing tasks across the servers to ensure optimal system performance.

Task:

- Create a system with 5 servers.
- Each server has a load state of "Underloaded", "Balanced", or "Overloaded".
- The Load Balancer Agent must scan the system and move tasks from overloaded servers to underloaded ones to balance the load.
- After balancing the load, display the updated load status of each server.

Task # 3

A network of servers has a set of backup tasks that need to be completed regularly. Some backups are successful, while others fail. The Backup Management Agent is responsible for ensuring all failed backups are retried and completed.

Task:

- Create a list of backup tasks with either "Completed" or "Failed" statuses.
- The Backup Management Agent scans for failed backups and retries them.
- After retrying, display the updated task statuses.

Task # 4 (utility based agent)

A cybersecurity exercise is being conducted for a company's security system, which consists of nine critical components (A through I). Each component of the system can either be Safe or have Vulnerabilities of varying severity. The company wants to ensure that its system remains secure, but it only has access to a basic security service that can patch Low Risk Vulnerabilities. High Risk Vulnerabilities require purchasing a premium security service to patch.

In this scenario, the goal is to simulate how a Utility-Based Security Agent scans and patches the system based on the vulnerabilities detected and the available resources (limited patching service).

- **Initial System Check:**
    - Initialize the system environment with random vulnerabilities (Safe, Low Risk Vulnerable, and High Risk Vulnerable).

  - Display the initial state of the system, showing which components are Safe and which have Vulnerabilities.
- **System Scan:**
  - The security agent will scan each component.
  - If a component is Vulnerable, the agent logs a warning.
  - If it is Safe, a success message is logged.
- **Patching Vulnerabilities:**
  - The agent will patch all Low Risk Vulnerabilities.
  - The agent will log a message for High Risk Vulnerabilities indicating the need for premium service to patch them.
- **Final System Check:**
  - Display the system's final state to confirm that all Low Risk Vulnerabilities have been patched.
  - The High Risk Vulnerabilities will remain unresolved unless the premium service is purchased.

Task # 5 (Goal based agent)

In a hospital, a delivery robot is tasked with delivering medicines to patients, assisting nurses, and performing other related activities in an efficient manner. The goal of the robot is to automatically move through hospital corridors, pick up medicines, deliver them to the correct patient rooms, and perform various tasks such as scanning patient IDs or alerting staff.

- **Components:**
  - Agent: The hospital delivery robot, which can move around, interact with patient rooms, pick up medicines, deliver them, and alert nurses or doctors when needed.

- **Environment:** The hospital layout, including:
  - Corridors
  - Patient rooms
  - Nurse stations
  - Medicine storage areas

- **Actions:**
  - Move to a location (room, station, etc.).
  - Pick up medicine from storage.
  - Deliver medicine to the patient's room.
  - Scan patient ID for verification.
  - Alert staff for critical situations.

- **Perceptions:**
  - Room numbers (where the robot should deliver the medicine).
  - Patient schedules (timing for when patients need their medicines).

- Medicine type (specific medicines to be delivered to patients).
- Staff availability (alerts if staff assistance is needed).

Goal-Based Agent Approach:
Goal: Deliver medicine to patients based on a schedule and room number, while ensuring all deliveries are correctly made. The robot must scan the patient's ID before delivering and alert nurses or doctors if needed.

Task # 6

In a building with multiple rooms, a firefighting robot has been deployed to save lives and prevent damage. The building is represented by a 3x3 grid, where each cell corresponds to a room. Some rooms contain fires, and others are safe.

- The robot starts at room 'a' and must move across all rooms, from 'a' to 'j', detecting and extinguishing any fires along the way.
- The robot needs to be aware of which rooms have fire and must extinguish them by changing the room's status from "fire" to "safe."
- The robot needs to continuously display the environment's status after each move and indicate when fires are detected and extinguished.
- **Initialization**:
  - Implement a 3x3 grid where rooms 'a', 'b', 'd', 'f', 'g', 'h' are safe (no fire), and rooms 'c', 'e', 'j' contain fires.
  - The robot starts at room 'a' and follows a predefined path: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'j'].
- **Robot Movement**:
  - The robot should move from room to room in the specified path.
  - At each room, the robot must check if there is a fire:
    - If there is a fire, extinguish it and update the room's status to safe.
    - If there is no fire, move to the next room.
- **Displaying the Environment**:
  - After each move, display the current status of the environment.
  - Use symbols like " 🔥 " for fire and " " (space) for a safe room to represent the environment visually.
- **Final Output**:

- After the robot has completed its movement, display the final status of all rooms (with all fires extinguished).