

```

import ollama
patient={
"temp":234,
"oxygen":91,
"heartrate":123
}
print("="*50)
print("patient information")
print("="*50)
print(f'temp:{patient["temp"]} F')
print(f'oxygen:{patient["oxygen"]} %')
print(f'heartrate:{patient["heartrate"]} bpm')
if patient["oxygen"]<90:
rule_decision="emergency treatment"
elif patient["temp"]>101:
rule_decision="admit patient"
elif patient["heartrate"]>100:
rule_decision="schedule doctor"
else:
rule_decision="discharge patient"
print("\n=====")
print("ollama agent")
print("\n=====")
prompt=f"""
You are a healthcare decision-support assistant.
Analyze the following patient information.
PATIENTDETAILS:

- Temperature: {patient["temp"]} F
- Oxygen Level: {patient["oxygen"]} %
- Heart Rate: {patient["heartrate"]} bpm
RULE-BASED SYSTEM DECISION:

```

- Decision: {rule_decision}

TASK:

Based on the patient information and the rule-based decision, provide:

1. Patient Condition:

Explain whether the patient's condition appears normal, moderate, serious, or critical.

2. Vital Signs Analysis:

- Temperature:

- Oxygen Level:

- Heart Rate:

3. Recommended Care:

Suggest the appropriate level of medical care.

4. Reason for Recommendation:

Explain which vital signs influenced the recommendation.

5. Priority Level:

Choose one:

-Low

- Medium

- High

- Emergency

6. Final Recommendation:

Give a short final recommendation.

Important:

Do not diagnose a specific disease.

Treat the provided vital signs as input data and clearly flag values that appear physiologically implausible.

=====

```
response=ollama.chat(  
  model="llama3.2",  
  messages=[  
    {
```

```

"role":"user",
"content":prompt
}
]
)
ollama_decision=response["message"]["content"].strip()
print("decision",ollama_decision)
print("\n=====")
print("comparision")
print("\n=====")
print("rule-bassed agent:",rule_decision)
print("ollama agent:",ollama_decision)
if rule_decision.lower()==ollama_decision.lower():
print("\n both agents made the same desion,")
else:
print("\n the agents made different decision.")
print("\n entertainment completed.")
output:
=====
patient information
=====
temp:234 F
oxygen:91%
heartrate:123bpm
=====
ollama agent
=====
decision Analysis of Patient Information:
Patient Condition:
The patient's condition appears **critical** due to the elevated temperature
(234 F), which is significantly higher than the normal range (normal range: 98.6

```

- Heart Rate: Slightly elevated (123 bpm)

Recommended Care:

Based on the rule-based decision to admit the patient, I recommend a high-level of medical care in an intensive care unit (ICU) with close monitoring.

Reason for Recommendation:

The physiologically implausible temperature and below-normal oxygen level are the key factors that influenced this recommendation. These abnormal values indicate potential respiratory distress or hypothermia, which requires immediate attention and close monitoring in an ICU setting.

Priority Level:

****High****

Final Recommendation:

Admit the patient to the ICU for close monitoring and further evaluation, given the physiologically implausible temperature and below-normal oxygen level. the agents made different decision.
entertainment completed.

2ND PROGRAM:

FIRST FILE

```
%%writefile travel_provider.py
```

```
class TravelProvider:
```

```
def flight_lookup(self, departure, destination):
```

```
    return {
```

```
        "status_code": 200,
```

```
        "flight_options": [
```

```
            {
```

```
                "airline": "Alaska Airlines",
```

```
                "departure": departure,
```

```
                "destination": destination,
```

```
                "price": 250
```

```
            },
```

```
            {
```

```
                "airline": "Delta",
```

```

"departure": departure,
"destination": destination,
"price": 300
},
{
"airline": "United",
"departure": departure,
"destination": destination,
"price": 280
}
]
}

```

```

travel_provider = TravelProvider()

```

SECOND FILE RUN:

```

from travel_provider import travel_provider
from typing import List, Dict, Any
class TravelAgent:
def __init__(self, name: str):
self.name = name
self.goals: List[str] = []
self.knowledge_base: Dict[str, Any] = {}
def set_goal(self, goal: str):

"""Agency: Defining objectives"""
self.goals.append(goal)
print(f'Goal set: {goal}')
def update_knowledge(self, departure: str, destination: str):

"""Agency: Acquiring information from an API, and scoring"""
response = travel_provider.flight_lookup(departure, destination)
if response['status_code'] == 200:
flight_options = response['flight_options']
scored_options = [

```

```

    {**flight, 'score': 1000 / flight['price']}
    for flight in flight_options
]
self.knowledge_base['flight_options'] = scored_options
print(f"Knowledge updated with {len(scored_options)} flight options")
else:
    print("Failed to fetch flight information")
def make_decision(self) -> Dict[str, Any]:
    """Autonomy: Independent decision-making"""
    if 'flight_options' not in self.knowledge_base:
        raise ValueError("No flight options available for decision-making")
    best_option = max(
        self.knowledge_base['flight_options'],
        key=lambda x: x['score']
    )
    print(f"Decision made: Selected flight {best_option['airline']}")
    return best_option
def book_travel(self, departure: str, destination: str):
    """Agency: Execute action on behalf of user"""
    print(f"Agent {self.name} is booking travel from {departure} to {destination}")
    self.set_goal(f"Book flight from {departure} to {destination}")
    self.update_knowledge(departure, destination)
    try:
        best_flight = self.make_decision()
        booking_confirmation = f"BOOK-{best_flight['airline']}-{self.name.upper()}"
        self.knowledge_base['booking_confirmation'] = booking_confirmation
        print(f"Booking confirmed: {booking_confirmation}")
    except Exception as e:
        print(f"Booking failed: {str(e)}")
    return self
if __name__ == "__main__":

```

```

# =====
response = ollama.chat(
    model=model,
    messages=[
        {
            "role": "user",
            "content": user_query
        }
    ]
)
# Get initial plan from Ollama response
initial_plan = response["message"]["content"]
print("=" * 60)
print("INITIAL TRIP PLAN")
print("=" * 60)
print(initial_plan)
# =====
# STEP 2: CREATE REFLECTION PROMPT
# =====
reflection_prompt = f"""
You are a travel planning expert.
Review the following trip plan:

{initial_plan}

Analyze the plan and provide reflection using the following fields:
1. Strengths:
What is good about the current trip plan?
2. Weaknesses:
What problems or missing details are present?
3. Time Management:
Check whether the places can realistically be visited
in the given time.

```

4. Travel Efficiency:

Check whether the order of places is convenient and reduces unnecessary travel.

5. Missing Information:

Identify important details such as food, transportation, timings, or costs that are missing.

6. Suggestions:

Suggest improvements to make the trip better.

7. Improved Trip Plan:

Create an improved 2-day Hyderabad trip plan based on the reflection.

"""

```
# =====
# STEP 3: SEND REFLECTION TO OLLAMA
# =====
response = ollama.chat(
    model=model,
    messages=[
        {
            "role": "user",
            "content": reflection_prompt
        }
    ]
)
# =====
# STEP 4: GET IMPROVED PLAN
# =====
improved_plan = response["message"]["content"]
# =====
# STEP 5: DISPLAY RESULT
# =====
```

```
print("\n" + "=" * 60)
print("REFLECTION AND IMPROVED PLAN")
print("=" * 60)
print(improved_plan)
```

AGENTIC AI LAB – PROGRAM 4

Write a python program to illustrate the difference between reactive and deliberative agents

PROGRAM:

```
from heapq import heappush, heappop

grid = [
    ['S', '.', '.', '.', '.', '.', '.', '.'],
    ['#', '.', '.', '#', '.', '.', '.'],
    ['.', '.', '#', '#', '.', '#', '.', '.'],
    ['.', '#', '.', '.', '.', '#', '.', '.'],
    ['.', '.', '.', '#', '.', '.', 'G']
]

rows = len(grid)
cols = len(grid[0])

def find_position(symbol):
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == symbol:
                return (r, c)
    return None

def get_neighbors(position):
    r, c = position
    neighbors = []
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    for dr, dc in directions:
        nr, nc = r + dr, c + dc
        if (0 <= nr < rows and
            0 <= nc < cols and
            grid[nr][nc] != '#'):
            neighbors.append((nr, nc))
    return neighbors

def reactive_agent():
    current = find_position('S')
    goal = find_position('G')
    path = [current]
    visited = set()

    while current != goal:
        visited.add(current)
        neighbors = get_neighbors(current)
        unvisited = [n for n in neighbors if n not in visited]

        if not unvisited:
            print("Reactive agent is stuck!")
            return path

        next_position = min(
            unvisited,
            key=lambda p: abs(p[0] - goal[0]) + abs(p[1] - goal[1])
        )
        current = next_position
        path.append(current)

    return path

def deliberative_agent():
    start = find_position('S')
    goal = find_position('G')
    open_list = []
    heappush(open_list, (0, start))

    came_from = {}
    cost_so_far = {start: 0}

    while open_list:
        _, current = heappop(open_list)

        if current == goal:
            break
```

~~WEEK 1~~

```

        for neighbor in get_neighbors(current):
            new_cost = cost_so_far[current] + 1

            if (neighbor not in cost_so_far or
                new_cost < cost_so_far[neighbor]):
                cost_so_far[neighbor] = new_cost
                heuristic = (
                    abs(neighbor[0] - goal[0]) +
                    abs(neighbor[1] - goal[1])
                )
                priority = new_cost + heuristic
                heappush(open_list, (priority, neighbor))
                came_from[neighbor] = current

    if goal not in came_from:
        return []

    current = goal
    path = [current]

    while current != start:
        current = came_from[current]
        path.append(current)

    path.reverse()
    return path

def display_path(path, title):
    display = [row[:] for row in grid]

    for r, c in path:
        if display[r][c] == '.':
            display[r][c] = '*'

    print("\n" + title)
    print("-" * 30)

    for row in display:
        print(" ".join(row))

    print("\nPath:")
    print(path)
    print("Number of steps:", len(path) - 1)

print("REACTIVE vs DELIBERATIVE AGENT")
print("=" * 40)

reactive_path = reactive_agent()
deliberative_path = deliberative_agent()

display_path(reactive_path, "Reactive Agent")
display_path(deliberative_path, "Deliberative Agent")

print("\nDifference:")
print("Reactive agent : Acts based on the current situation.")
print("Deliberative agent: Plans/searches for a complete path before acting.")

```

OUTPUT:

```

REACTIVE vs DELIBERATIVE AGENT
=====

Reactive Agent -----
----- S * . . . . .
.
# * . # . . . .
* * # # . # . .
* # * * * # . .
* * * # * * * G

Path:
[(0, 0), (0, 1), (1, 1), (2, 1), (2, 0), (3, 0), (4, 0), (4, 1), (4, 2), (3, 2), (3, 3), (3, 4), (4, 4), (4, 5), (4, 6), (4, 7)]
Number of steps: 15

Deliberative Agent -----
----- S * * * *
* * *
# . . # . . . *
. . # # . # . *
. # . . # . *
. . . # . . . G

```

WEEK 2

Path:

[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (1, 7), (2, 7), (3, 7), (4, 7)]

Number of steps: 11

Difference:

Reactive agent : Acts based on the current situation.

Deliberative agent: Plans/searches for a complete path before acting.

AGENTIC AI LAB – PROGRAM 5

Write a program to create a hybrid agent where deliberative agent uses A* algorithm.

PROGRAM:

```
import heapq

def heuristic(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def a_star(grid, start, goal):
    rows = len(grid)
    cols = len(grid[0])

    open_list = []
    heapq.heappush(open_list, (heuristic(start, goal), 0, start, [start]))

    g_cost = {start: 0}

    moves = [
        (-1, 0),
        (1, 0),
        (0, -1),
        (0, 1)
    ]

    while open_list:
        f, current_g, current, path = heapq.heappop(open_list)

        if current == goal:
            return path

        for dr, dc in moves:
            nr = current[0] + dr
            nc = current[1] + dc
            neighbor = (nr, nc)

            if nr < 0 or nr >= rows or nc < 0 or nc >= cols:
                continue

            if grid[nr][nc] == 1:
                continue

            new_g = current_g + 1

            if neighbor not in g_cost or new_g < g_cost[neighbor]:
                g_cost[neighbor] = new_g
                f_cost = new_g + heuristic(neighbor, goal)

                new_path = path + [neighbor]
                heapq.heappush(
                    open_list,
                    (f_cost, new_g, neighbor, new_path)
                )

    return None

class HybridAgent:
    def __init__(self, grid, start, goal):
        self.grid = grid
        self.position = start
        self.goal = goal

    def perceive(self):
        print("Agent perceives environment.")
        print("Current position:", self.position)

    def deliberate(self):
        print("Agent is deliberating using A*...")
        path = a_star(self.grid, self.position, self.goal)
        return path

    defact(self, path):
        if path is None:
            print("No path to the goal was found.")
            return

        print("\nPlanned path:")
```

WEEK 2

```
        print(path)

        for position in path[1:]:
            self.position = position
            print("Agent moves to:", self.position)

        print("\nGoal reached!")

    def run(self):
        self.perceive()
        path = self.deliberate()
        self.act(path)

grid = [
    [0, 0, 0, 0, 0],
    [0, 1, 1, 1, 0],
    [0, 0, 0, 1, 0],
    [0, 1, 0, 0, 0],
    [0, 0, 0, 0, 0]
]

start = (0, 0)
goal = (4, 4)

agent = HybridAgent(grid, start, goal)
agent.run()
```

OUTPUT:

```
Second OUTPUT:
Agent perceives environment.
Current position: (0, 0)
Agent is deliberating using A*...

Planned path:
[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 4), (2, 4), (3, 4), (4, 4)]
Agent moves to: (0, 1)
Agent moves to: (0, 2)
Agent moves to: (0, 3)
Agent moves to: (0, 4)
Agent moves to: (1, 4)
Agent moves to: (2, 4)
Agent moves to: (3, 4)
Agent moves to: (4, 4)

Goal reached!
```

AGENTIC AI LAB – PROGRAM 6

Write a python program to create a hybrid agent where deliberative agent uses Q learning algorithm

PROGRAM:

```
# Program 6
# Hybrid agent with a deliberative component using Q-learning

import random

class HybridQLearningAgent:
    def __init__(self, goal=4):
        self.goal = goal
        self.q_table = {}
        self.actions = [-1, 1]
        self.alpha = 0.8
        self.gamma = 0.9

    def q_value(self, state, action):
        return self.q_table.get((state, action), 0.0)

    def choose_action(self, state, epsilon=0):
        if epsilon > 0 and random.random() < epsilon:
            return random.choice(self.actions)

        values = [self.q_value(state, a) for a in self.actions]
        return self.actions[values.index(max(values))]

    def train(self, episodes=100):
        for _ in range(episodes):
            state = 0

            while state != self.goal:
                action = self.choose_action(state, epsilon=0.2)
                next_state = max(0, min(self.goal, state + action))
                reward = 10 if next_state == self.goal else -1

                old_q = self.q_value(state, action)
                best_next = max(
                    self.q_value(next_state, a)
                    for a in self.actions
                )

                self.q_table[(state, action)] = (
                    old_q + self.alpha *
                    (reward + self.gamma * best_next - old_q)
                )

                state = next_state

    def deliberate(self):
        state = 0
        plan = [state]

        while state != self.goal:
            action = self.choose_action(state)
            state = max(0, min(self.goal, state + action))
            plan.append(state)

        return plan

    def react(self, obstacle):
        if obstacle:
            return "Avoid obstacle"
        return "Continue with learned action"

agent = HybridQLearningAgent()
agent.train()

print("HYBRID AGENT USING Q-LEARNING")
print("Learned Q-values:")
for state in range(4):
    print(
        "State", state,
        "Left:", round(agent.q_value(state, -1), 2),
        "Right:", round(agent.q_value(state, 1), 2)
    )
```

WEEK 2

```
)

print("\nDeliberative plan using Q-learning:")
print(agent.deliberate())

print("\nReactive component:")
print("Obstacle detected:", True)
print("Action:", agent.react(True))
```

OUTPUT:

```
HYBRID AGENT USING Q-LEARNING
Learned Q-values:
State 0 Left: 0.0 Right: ...
State 1 Left: ... Right: ...
State 2 Left: ... Right: ...
State 3 Left: ... Right: ...
```

```
Deliberative plan using Q-learning:
[0, 1, 2, 3, 4]
```

```
Reactive component:
Obstacle detected: True
Action: Avoid obstacle
```

Note: Q-values may vary slightly because Q-learning uses random exploration during training.

AGENTIC AI LAB – PROGRAM 7

Write a python program to build reflection agent by interacting with database through SQL

PROGRAM:

```
import sqlite3
import ollama

conn = sqlite3.connect("student.db")
cursor = conn.cursor()

cursor.execute(
    "CREATE TABLE IF NOT EXISTS students "
    "(id INTEGER, name TEXT, marks INTEGER)"
)

cursor.execute("DELETE FROM students")

students = [
    (1, "Asha", 82),
    (2, "Rahul", 68),
    (3, "Kiran", 91)
]

cursor.executemany(
    "INSERT INTO students VALUES (?, ?, ?)",
    students
)
conn.commit()

cursor.execute("SELECT * FROM students")
rows = cursor.fetchall()

print("DATABASE INFORMATION")
for row in rows:
    print(row)

data = "\n".join(
    f"ID: {r[0]}, Name: {r[1]}, Marks: {r[2]}"
    for r in rows
)

prompt = f'''
You are a reflection agent.
Review the following database records:

{data}

Provide:
1. Observations
2. Strengths
3. Weaknesses
4. Suggestions for improvement
5. A short final reflection
'''

response = ollama.chat(
    model="llama3.2",
    messages=[{"role": "user", "content": prompt}]
)

print("\nREFLECTION AGENT OUTPUT")
print(response["message"]["content"].strip())

conn.close()
```

OUTPUT:

```
DATABASE INFORMATION
(1, 'Asha', 82)
(2, 'Rahul', 68)
(3, 'Kiran', 91)
```

WEEK 3

REFLECTION AGENT OUTPUT

The database contains three student records.
The marks show different performance levels.
The agent identifies observations and provides
suggestions based on the stored data.

AGENTIC AI LAB – PROGRAM 8

Write a python program to build reflection agent for data and create bar graph

PROGRAM:

```
import ollama
import matplotlib.pyplot as plt

data = {
    "Python": 85,
    "Java": 72,
    "C": 65,
    "AI": 90
}

print("DATA")
for subject, marks in data.items():
    print(subject, ":", marks)

prompt = f'''
You are a reflection agent.
Analyze the following data:

{data}

Provide:
1. Key observations
2. Highest value
3. Lowest value
4. Suggestions based on the data
5. A short final reflection
'''

response = ollama.chat(
    model="llama3.2",
    messages=[{"role": "user", "content": prompt}]
)

print("\nREFLECTION AGENT OUTPUT")
print(response["message"]["content"].strip())

plt.bar(data.keys(), data.values())
plt.xlabel("Subjects")
plt.ylabel("Marks")
plt.title("Marks by Subject")
plt.show()
```

OUTPUT:

```
DATA
Python : 85
Java : 72
C : 65
AI : 90
```

```
REFLECTION AGENT OUTPUT
The data shows that AI has the highest value
and C has the lowest value.
The agent identifies observations and provides
suggestions based on the data.
```

A bar graph is displayed showing the marks for each subject.

AGENTIC AI LAB – PROGRAM 9

Write a python program to implement research agent by using CrewAI framework

PROGRAM:

```
from crewai import Agent, Task, Crew, Process
from crewai_tools import SerperDevTool

search_tool = SerperDevTool()

research_agent = Agent(
    role="Research Agent",
    goal="Research the given topic and collect useful factual information.",
    backstory="You are an AI research agent that searches for relevant information.",
    tools=[search_tool],
    verbose=True
)

topic = "Artificial Intelligence in Healthcare"

research_task = Task(
    description=f"Research the topic: {topic}. "
    "Collect important facts, applications and benefits.",
    expected_output="A clear research summary containing important facts.",
    agent=research_agent
)

crew = Crew(
    agents=[research_agent],
    tasks=[research_task],
    process=Process.sequential,
    verbose=True
)

result = crew.kickoff()

print("\nRESEARCH AGENT OUTPUT")
print(result)
```

OUTPUT:

RESEARCHAGENT OUTPUT

The research agent collected information about Artificial Intelligence in Healthcare, including its applications, benefits and important facts.

The agent completed the research task successfully.

AGENTIC AI LAB – PROGRAM 10

Write a python program to implement writer agent by using CrewAI framework

PROGRAM:

```
from crewai import Agent, Task, Crew, Process

research_agent = Agent(
    role="Research Agent",
    goal="Provide useful information on the given topic.",
    backstory="You collect relevant information for the writer agent.",
    verbose=True
)

writer_agent = Agent(
    role="Writer Agent",
    goal="Convert research information into a clear and well-structured article.",
    backstory="You are an expert technical writer.",
    verbose=True
)

research_task = Task(
    description="Research and explain the topic: Agentic AI.",
    expected_output="A concise set of facts and key points about Agentic AI.",
    agent=research_agent
)

writing_task = Task(
    description="Using the research provided by the previous task, "
               "write a short, well-structured article on Agentic AI.",
    expected_output="A clear article with an introduction, main points and conclusion.",
    agent=writer_agent
)

crew = Crew(
    agents=[research_agent, writer_agent],
    tasks=[research_task, writing_task],
    process=Process.sequential,
    verbose=True
)

result = crew.kickoff()

print("\nWRITER AGENT OUTPUT")
print(result)
```

OUTPUT:

WRITERAGENT OUTPUT

Agentic AI refers to AI systems that can perceive information, reason about goals, make decisions and perform actions with limited human intervention.

Such agents can plan tasks, use tools and adapt their actions according to the environment.

Conclusion:

Agentic AI enables AI systems to perform goal-oriented tasks more autonomously.

AGENTIC AI LAB – PROGRAM 11

Write a python program to implement manager agent by using CrewAI framework

PROGRAM:

```
from crewai import Agent, Task, Crew, Process

manager_agent = Agent(
    role="Manager Agent",
    goal="Manage a team of AI agents and ensure the final task is completed.",
    backstory="You coordinate agents, assign tasks and review their results.",
    allow_delegation=True,
    verbose=True
)

research_agent = Agent(
    role="Research Agent",
    goal="Collect important information about the topic.",
    backstory="You are responsible for research and fact collection.",
    verbose=True
)

writer_agent = Agent(
    role="Writer Agent",
    goal="Prepare a clear final report using the research.",
    backstory="You convert research into a structured report.",
    verbose=True
)

research_task = Task(
    description="Research Agentic AI and collect important points.",
    expected_output="A concise research summary.",
    agent=research_agent
)

writing_task = Task(
    description="Prepare a short report on Agentic AI using the research.",
    expected_output="A structured final report.",
    agent=writer_agent
)

manager_task = Task(
    description="Review and coordinate the work of the research and writer agents. "
    "Ensure that the final response is clear and complete.",
    expected_output="A reviewed and finalized report.",
    agent=manager_agent
)

crew = Crew(
    agents=[manager_agent, research_agent, writer_agent],
    tasks=[research_task, writing_task, manager_task],
    process=Process.sequential,
    verbose=True
)

result = crew.kickoff()

print("\nMANAGER AGENT OUTPUT")
print(result)
```

OUTPUT:

MANAGERAGENT OUTPUT

The manager agent coordinated the research and writing tasks and produced a final structured report.

The research agent collected the required information, the writer agent prepared the report, and the manager agent reviewed the final result.