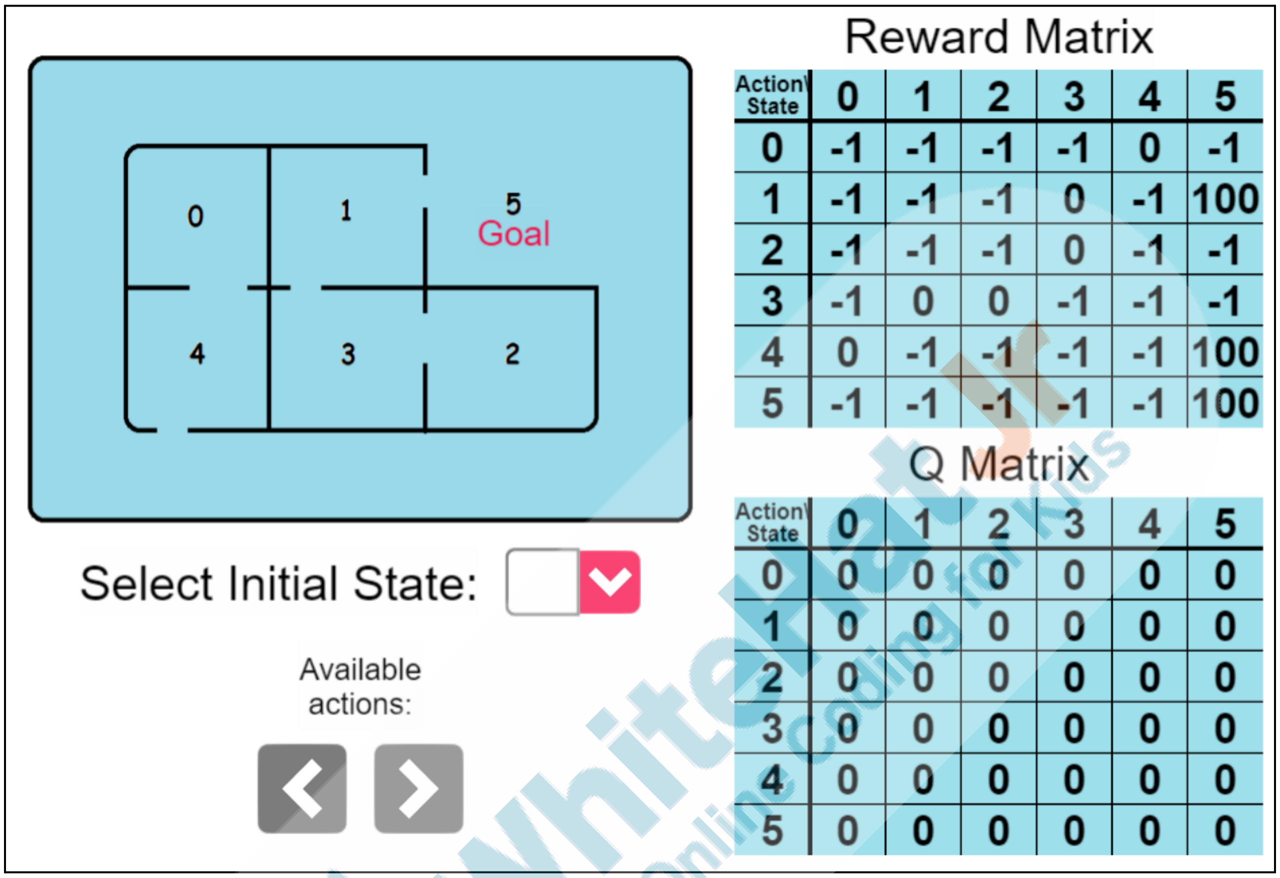
Topic	TRAIN RL MODEL	
Class Description	**The student will learn to train the Reinforcement Learning model using Reward and Q-matrix.**	
Class	**PRO C126**	
Class time	**45 mins**	
Goal	● Introduction to Episodes ● To train the RL model by running different episodes ● Find an optimal path to reach the goal	
Resources Required	● Teacher Resources: ○ Laptop with internet connectivity ○ Earphones with mic ○ Notebook and pen ○ Smartphone ● Student Resources: ○ Laptop with internet connectivity ○ Earphones with mic ○ Notebook and pen	
Class structure	**Warm-Up** **Teacher-Led Activity 1** **Student-Led Activity 1** **Wrap-Up**	**10 mins** **10 mins** **20 mins** **05 mins**

WARM-UP SESSION - 10 mins

Teacher Starts Slideshow
Slide 1 to 4
Refer to speaker notes and follow the instructions on each slide.

Teacher Action	**Student Action**

Hey <student's name>. How are you? It's great to see you! Are you excited to learn something new today? **Following are the WARM-UP session deliverables:** ● Greet the student. ● Revision of previous class activities. ● Quizzes.	**ESR**: Hi, thanks! Yes, I am excited about it! Click on the slide show tab and present the slides
WARM-UP QUIZ Click on In-Class Quiz	
Continue WARM-UP Session **Slide 5 to 13**	
Activity Details **Following are the session deliverables:** ● Appreciate the student. ● Narrate the story by using hand gestures and voice modulation methods to bring in more interest in students.	
Teacher Action	**Student Action**
Since now we know how to perform operations on matrices, today we are going to learn about training a Reinforcement learning model. For this let's go back and check the Reward matrix and Q-matrix that we had created. ***Note:*** *Open Teacher Activity 1 and show the matrices to the student.*	



Reward Matrix

Action\State	0	1	2	3	4	5
0	-1	-1	-1	-1	0	-1
1	-1	-1	-1	0	-1	100
2	-1	-1	-1	0	-1	-1
3	-1	0	0	-1	-1	-1
4	0	-1	-1	-1	-1	100
5	-1	-1	-1	-1	-1	100

Q Matrix

Action\State	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0

Depending upon these matrices we will create a model which will try to take different paths and the agent will learn by this trial and error method.	

Teacher Ends Slideshow

TEACHER-LED ACTIVITY - 10 mins

Teacher Initiates Screen Share

<u>ACTIVITY</u>

- **To code the model for running episodes**

Teacher Action	**Student Action**

Let us recall the problem statement. We have an agent in the environment. The agent has to reach the goal by taking the minimum number of actions.

So, we have to make sure that the agent learns by experimenting with different states.

When the agent starts from the current state till it finds the goal is known as an **Episode**. For example, playing an entire game is an episode since we are aiming to reach the goal.

So we'll create a function to run episodes and find a path for the agent to reach the goal. In the process, the agent will get trained on different paths. After learning, the agent will be able to find the best path to reach the goal from any current state.

We'll start with the previous class code.

Note: *Open Teacher Activity 2 for the Boilerplate code and run all the cells and simultaneously take the student through all the functions.*

Let's go through the functions that we had created previously.

1. Defined the **set_initial_state()** function: To set the initial state as a random state from **0** to **5**.
2. Created **Reward Matrix:** Assigned rewards for actions (-1,0,100).
3. Defined **get_action()** function: To decide the available actions.
4. Created **Q-matrix**.
5. Defined the **take_action()** function: To take the action from available states and update Q-matrix depending upon the reward for the action.

We had written a function called **take_action()** in the last class. Do you remember why we use it?

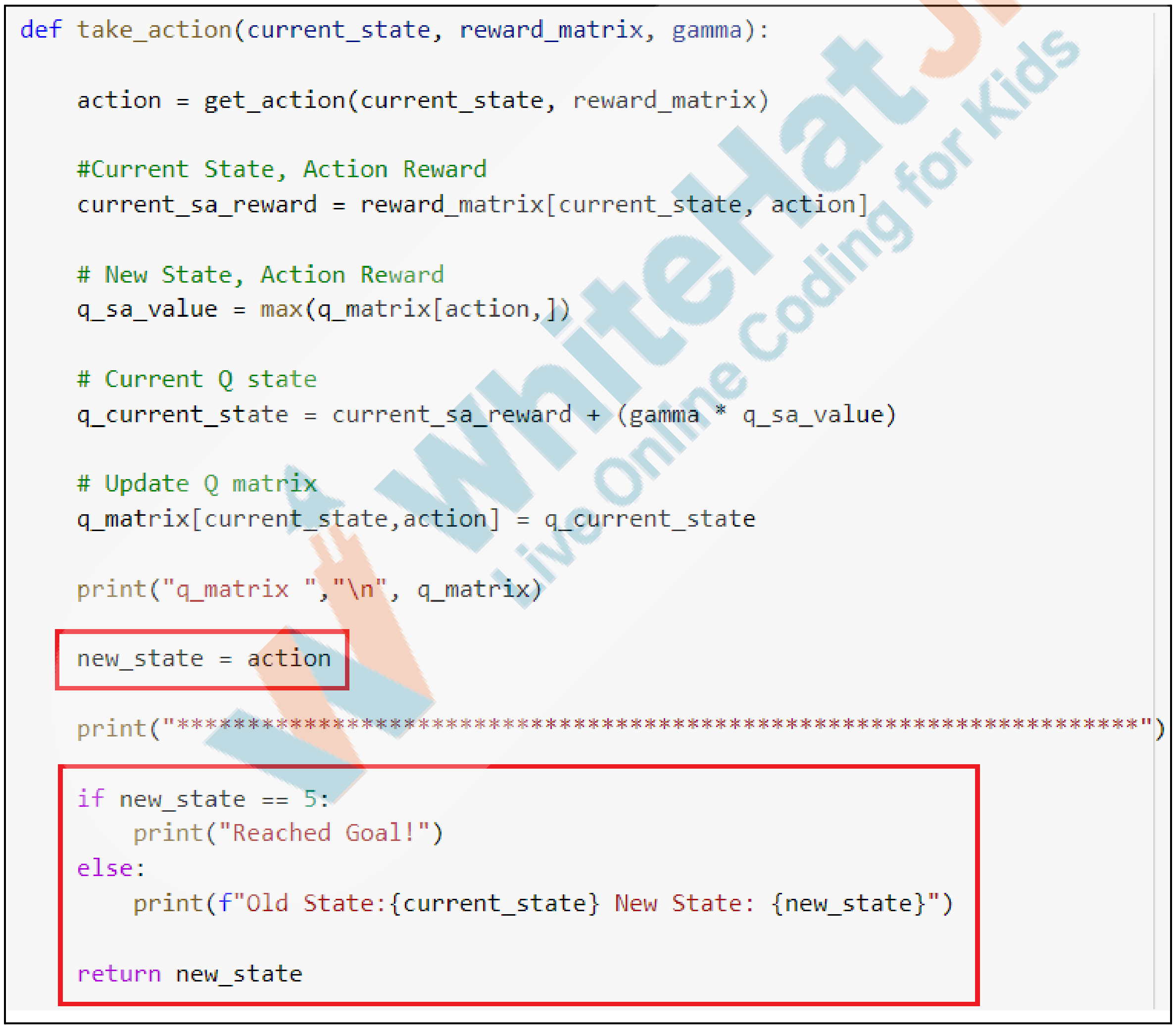
ESR: To update the **Q-matrix**.

Yes! So after updating the **Q-matrix** we'll be taking action.

The process will be repeated to find the Goal. If the agent reaches the goal then we have to end the episode.

Thus, each time an agent takes action, it moves to the next state, and thus its **previous action** becomes its **current state.**

Let's add a condition in the previously created **take_action()** function, if we reach the goal then the episode should end. Else, the action taken will become our new state.

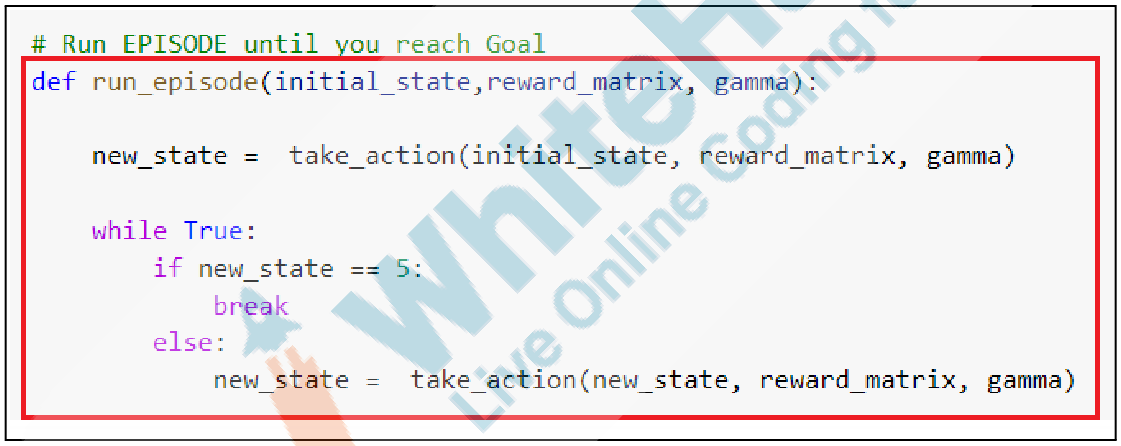


```python
def take_action(current_state, reward_matrix, gamma):

    action = get_action(current_state, reward_matrix)

    #Current State, Action Reward
    current_sa_reward = reward_matrix[current_state, action]

    # New State, Action Reward
    q_sa_value = max(q_matrix[action,])

    # Current Q state
    q_current_state = current_sa_reward + (gamma * q_sa_value)

    # Update Q matrix
    q_matrix[current_state,action] = q_current_state

    print("q_matrix ","\n", q_matrix)

    new_state = action

    print("********************************************************************")

    if new_state == 5:
        print("Reached Goal!")
    else:
        print(f"Old State:{current_state} New State: {new_state}")

    return new_state
```

But what if we again get state 5(Goal) in the else condition?

ESR: Check the variable new_state again if its goal,

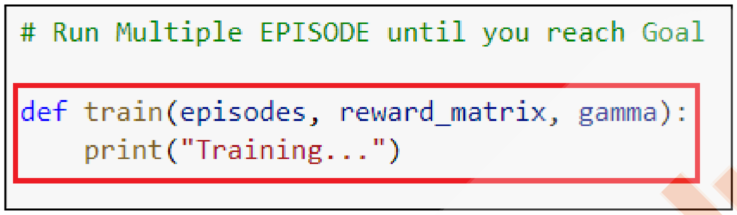
Exactly!! We'll be creating a function to run the episode. Also, we'll make sure that we have considered all the conditions for the agent while running an episode. 1. Define a function **run_episode()**. 2. Pass **initial_state**, **reward_matrix**, and **gamma**. We need these parameters because we'll be calling the **take_action()** function and getting the **new_state**. 3. We'll check the **new_state** here. If it is **5** then we won't proceed. 4. Else, we'll proceed to run the episode.	the agent has to stop.



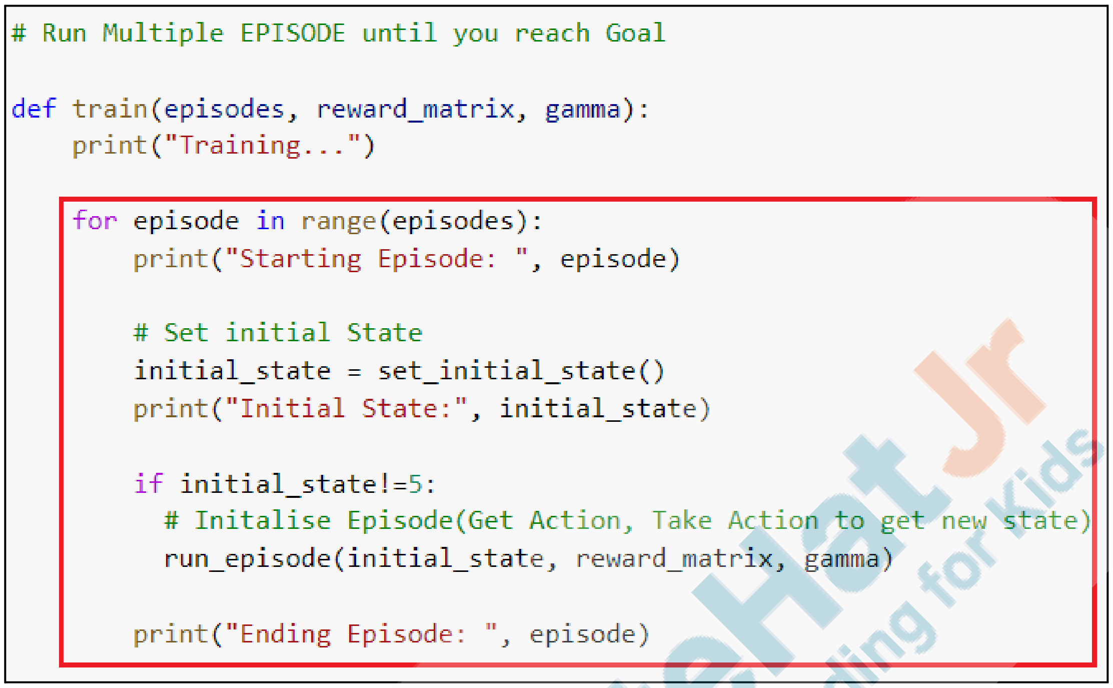
```
# Run EPISODE until you reach Goal
def run_episode(initial_state,reward_matrix, gamma):

    new_state =  take_action(initial_state, reward_matrix, gamma)

    while True:
        if new_state == 5:
            break
        else:
            new_state =  take_action(new_state, reward_matrix, gamma)
```

After this, we have to make the agent explore different states and reach the goal. We need a function that should run an episode. Which parameters do we need to run one episode? Yes!! So we'll have to take the reward matrix and gamma. Since we had written the function to calculate the Q-matrix every time the agent takes action, we'll call that function inside the **train()** function.	**ESR:** Reward matrix, Q-matrix, and gamma to calculate Bellman equation for updating Q-matrix.

We'll create a **train()** function that takes the number of episodes because we want to train the agent for multiple episodes and make it learn the best path.	

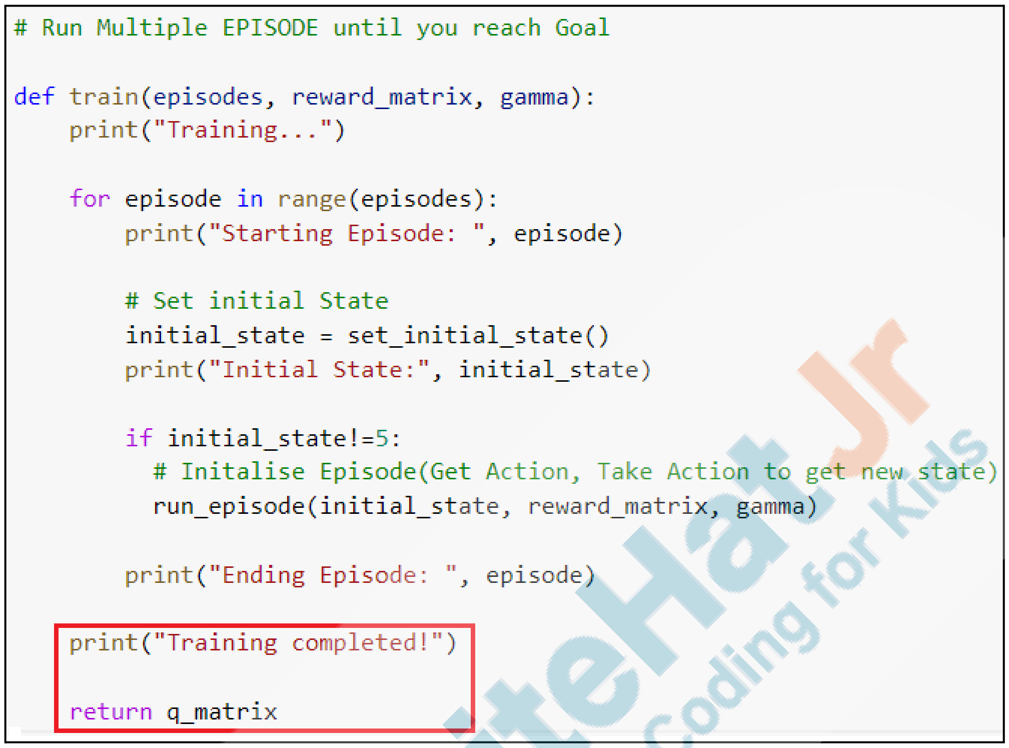
 `# Run Multiple EPISODE until you reach Goal`
 	
`def train(episodes, reward_matrix, gamma):`
`    print("Training...")`	

Next, define **for** loop to iterate for episodes. In this loop we'll call the functions we have created previously. 1. Start by printing the ongoing episode. 2. Call the **initial_state()** function. This will return the initial state randomly from **0** to **5**. 3. Check if the initial **state=5** or not. 4. Check if the agent is not in state **5**, we have to call the **run_episode()** function. This will call the **run_episode()** function and return the updated Q-matrix.	



```python
# Run Multiple EPISODE until you reach Goal

def train(episodes, reward_matrix, gamma):
    print("Training...")

    for episode in range(episodes):
        print("Starting Episode: ", episode)

        # Set initial State
        initial_state = set_initial_state()
        print("Initial State:", initial_state)

        if initial_state!=5:
          # Initalise Episode(Get Action, Take Action to get new state)
          run_episode(initial_state, reward_matrix, gamma)

        print("Ending Episode: ", episode)
```

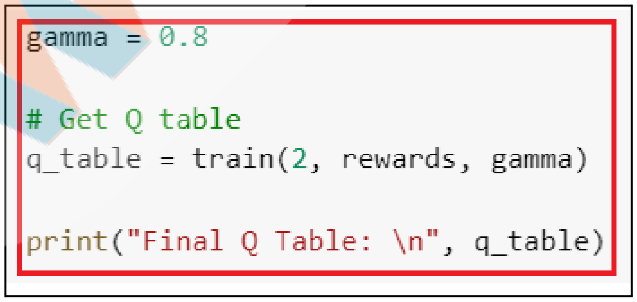
Once the goal is reached, we'll print that the episode is over. Also, we'll return the final updated Q-matrix for that particular episode.



```
# Run Multiple EPISODE until you reach Goal

def train(episodes, reward_matrix, gamma):
    print("Training...")

    for episode in range(episodes):
        print("Starting Episode: ", episode)

        # Set initial State
        initial_state = set_initial_state()
        print("Initial State:", initial_state)

        if initial_state!=5:
          # Initalise Episode(Get Action, Take Action to get new state)
          run_episode(initial_state, reward_matrix, gamma)

        print("Ending Episode: ", episode)

    print("Training completed!")

    return q_matrix
```

Let's check the output we get after calling the **train()** function. In this function, we'll provide gamma, number of episodes, and reward matrix.
As the **train()** function returns Q-matrix, we'll store it in the **q_table** variable and print this matrix.

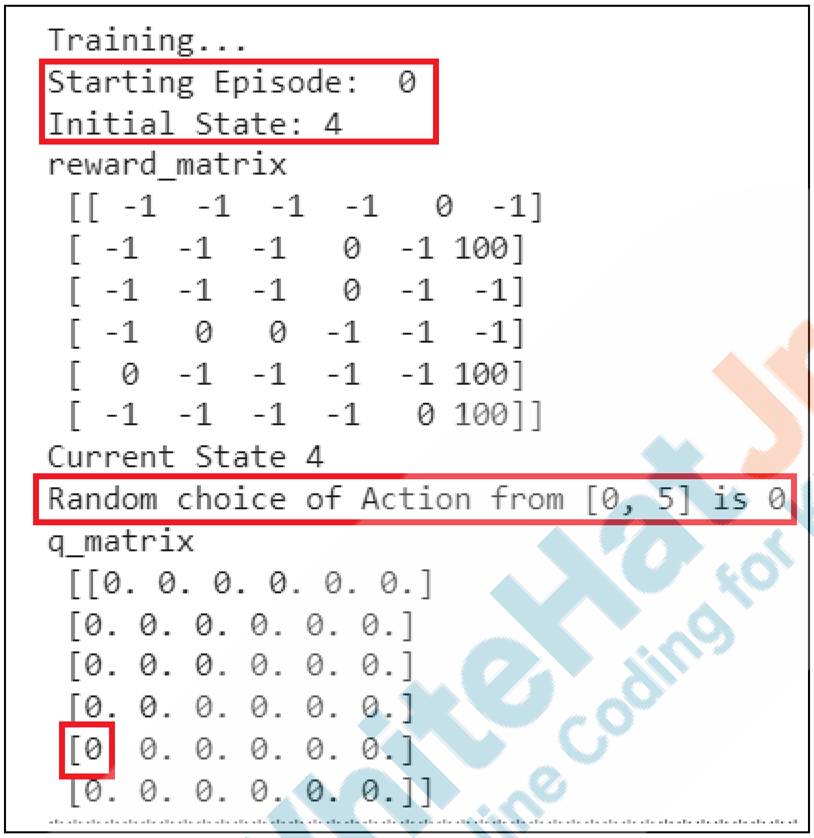
***Note:** The teacher will run for one or two episodes.*



```
gamma = 0.8

# Get Q table
q_table = train(2, rewards, gamma)

print("Final Q Table: \n", q_table)
```

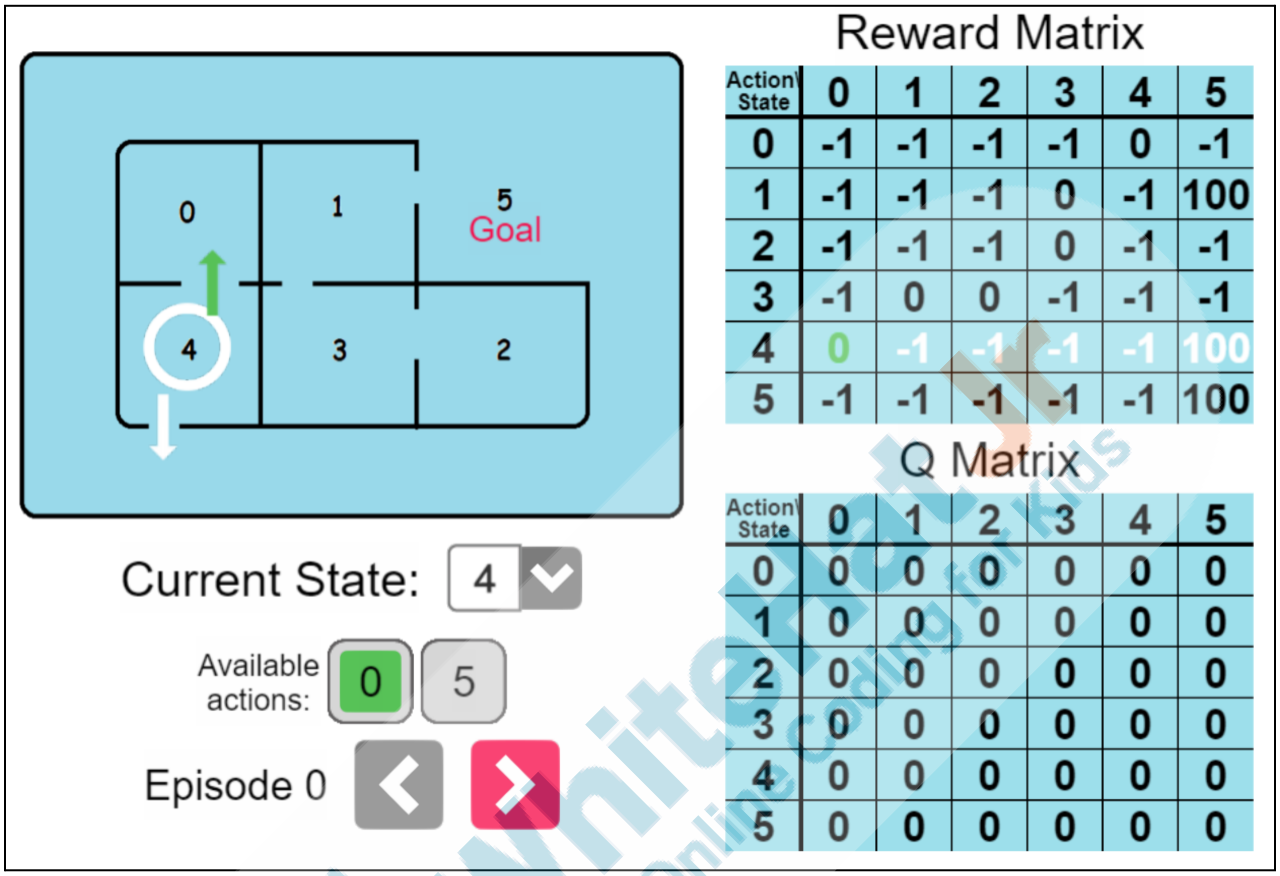
Let's run this and check the output.

***Note:** Open [Teacher Activity 3](#) simultaneously to make the student understand how Q values are updated.*



```
Training...
Starting Episode:  0
Initial State: 4
reward_matrix
 [[ -1  -1  -1  -1   0  -1]
 [ -1  -1  -1   0  -1 100]
 [ -1  -1  -1   0  -1  -1]
 [ -1   0   0  -1  -1  -1]
 [  0  -1  -1  -1  -1 100]
 [ -1  -1  -1  -1   0 100]]
Current State 4
Random choice of Action from [0, 5] is 0
q_matrix
 [[0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0  0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]]
```

So, we can see that for the 1st episode, we are getting an initial state. This is 4 so available states are 0 or 5. Among these, 0 is chosen randomly. With the help of the reward matrix, the Q-matrix is getting updated. Here, using the Bellman equation the rewards are calculated and placed at position (4,0) in the Q-matrix.



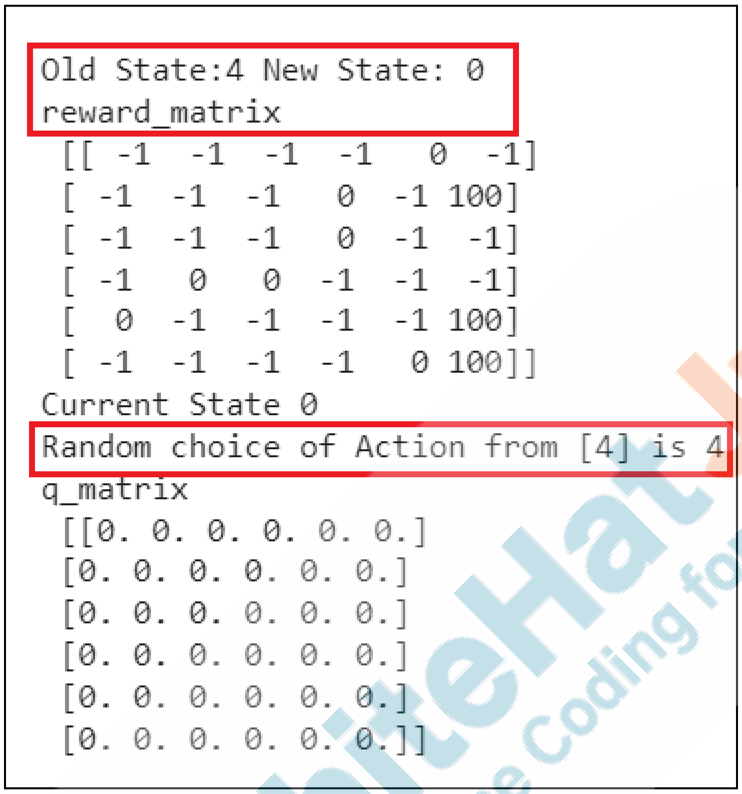
Reward Matrix

Action\State	0	1	2	3	4	5
0	-1	-1	-1	-1	0	-1
1	-1	-1	-1	0	-1	100
2	-1	-1	-1	0	-1	-1
3	-1	0	0	-1	-1	-1
4	0	-1	-1	-1	-1	100
5	-1	-1	-1	-1	-1	100

Q Matrix

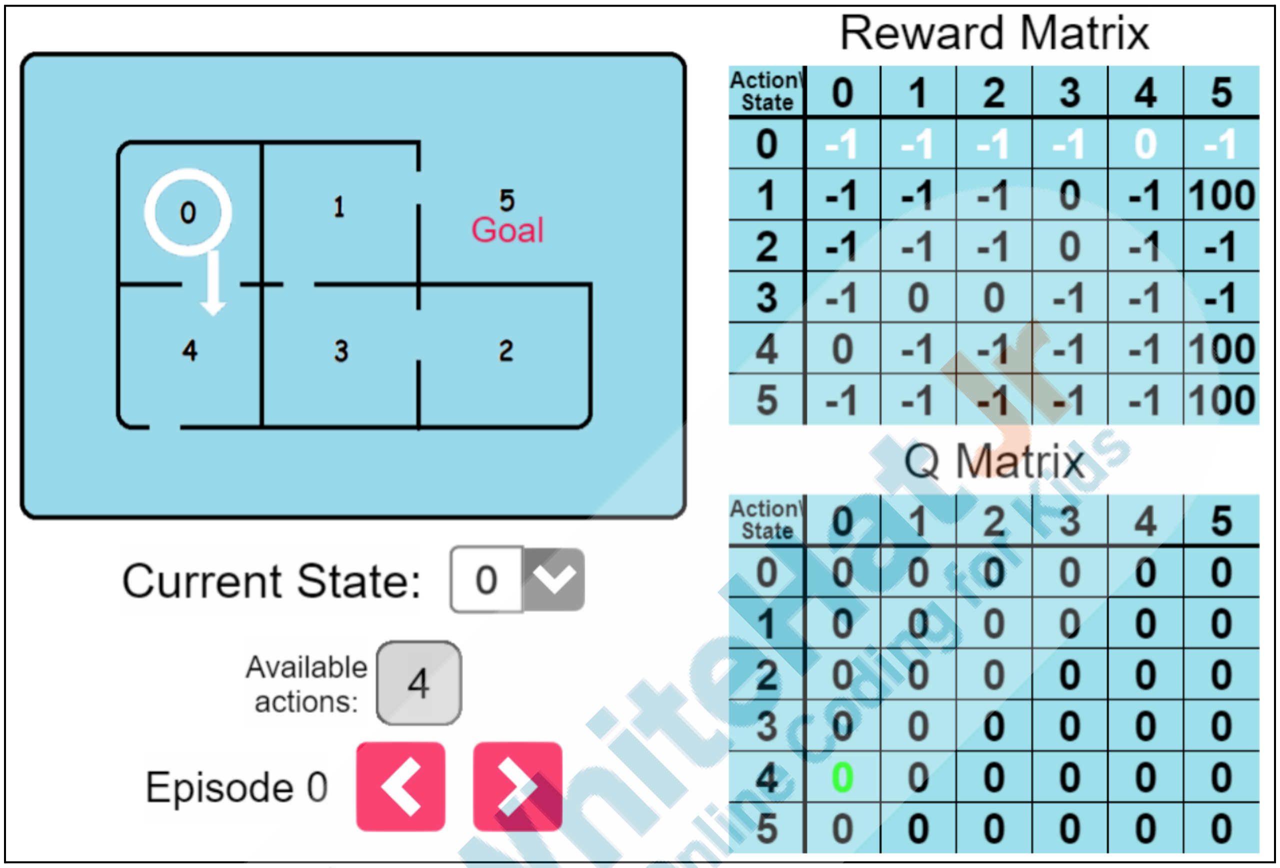
Action\State	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0

So, now 0 will be the new state. Thus, the available choice left is 4. So, the agent will come back to 4. At every step, the reward matrix is updated.



```
Old State:4 New State: 0
reward_matrix
 [[ -1  -1  -1  -1   0  -1]
 [ -1  -1  -1   0  -1 100]
 [ -1  -1  -1   0  -1  -1]
 [ -1   0   0  -1  -1  -1]
 [  0  -1  -1  -1  -1 100]
 [ -1  -1  -1  -1   0 100]]
Current State 0
Random choice of Action from [4] is 4
q_matrix
 [[0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0.]]
```

Simultaneously, we can check the output here.	



Current State: 0

Available actions: 4

Episode 0

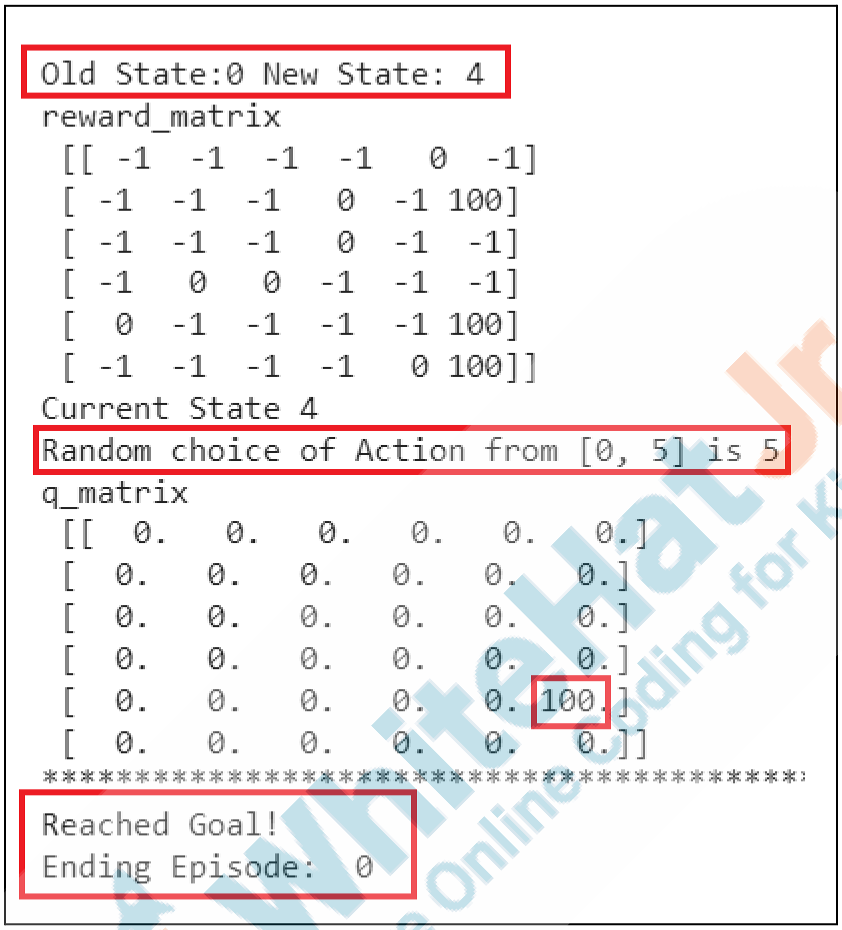
Reward Matrix

Action\State	0	1	2	3	4	5
0	-1	-1	-1	-1	0	-1
1	-1	-1	-1	0	-1	100
2	-1	-1	-1	0	-1	-1
3	-1	0	0	-1	-1	-1
4	0	-1	-1	-1	-1	100
5	-1	-1	-1	-1	-1	100

Q Matrix

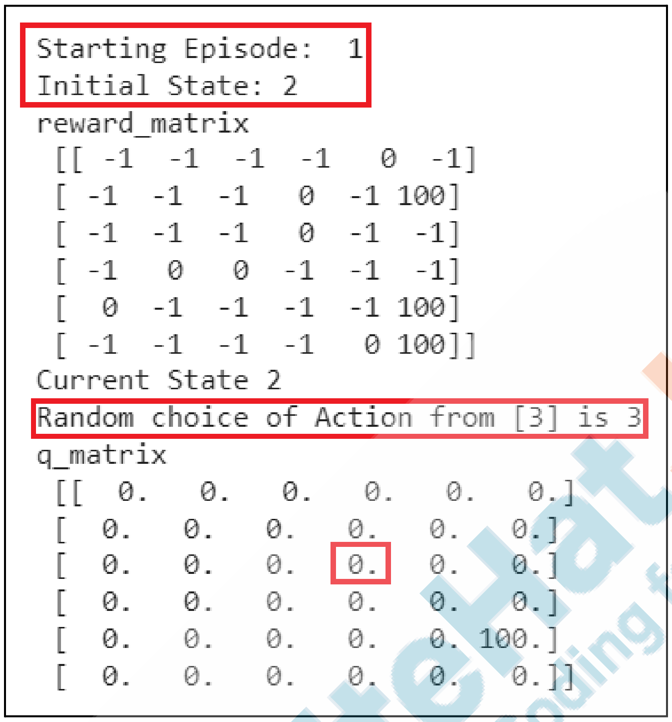
Action\State	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0

Again the agent will go to state 4 and the choice will be 0 or 5. This time if 5 is chosen then it will reach the goal and the episode will be ended.



```
Old State:0 New State: 4
reward_matrix
 [[ -1   -1   -1   -1    0   -1]
 [ -1   -1   -1    0   -1 100]
 [ -1   -1   -1    0   -1   -1]
 [ -1    0    0   -1   -1   -1]
 [  0   -1   -1   -1   -1 100]
 [ -1   -1   -1   -1    0 100]]
Current State 4
Random choice of Action from [0, 5] is 5
q_matrix
 [[  0.    0.    0.    0.    0.    0.]
 [  0.    0.    0.    0.    0.    0.]
 [  0.    0.    0.    0.    0.    0.]
 [  0.    0.    0.    0.    0.    0.]
 [  0.    0.    0.    0.    0. 100.]
 [  0.    0.    0.    0.    0.    0.]]
*****************************************
Reached Goal!
Ending Episode:  0
```

This way the process will be continued and then the episode will start. This time a new state will be chosen for the next episode. The process will continue till the time the Goal is reached.	



```
Starting Episode:  1
Initial State: 2
reward_matrix
 [[ -1  -1  -1  -1   0  -1]
 [ -1  -1  -1   0  -1 100]
 [ -1  -1  -1   0  -1  -1]
 [ -1   0   0  -1  -1  -1]
 [  0  -1  -1  -1  -1 100]
 [ -1  -1  -1  -1   0 100]]
Current State 2
Random choice of Action from [3] is 3
q_matrix
 [[  0.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0.   0.]
 [  0.   0.   0.   0.   0. 100.]
 [  0.   0.   0.   0.   0.   0.]]
```

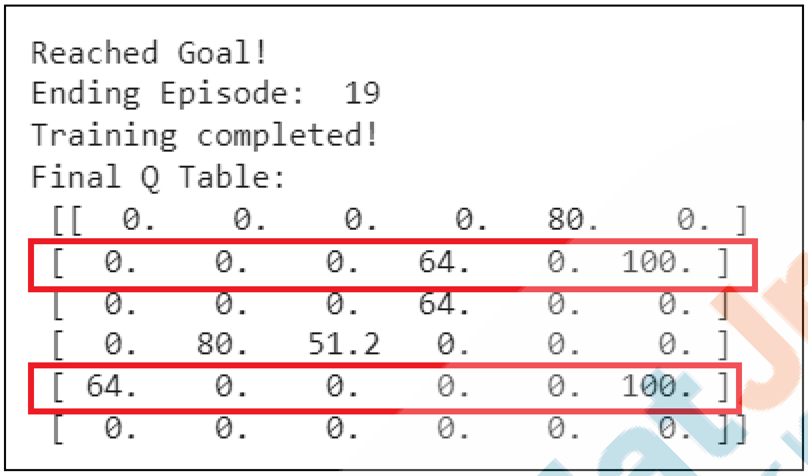
So, this way we'll be making the machine learn by itself. This is just like we are training the machine by trial and error method. Now, the final **q_table** will act as the **trained environment** for us. In the next step, we'll use this Q-matrix as a reference to make the move. Now you have to run the code for 20 episodes. This will train the agent well to find the best path for all the initial states.	

Teacher Stops Screen Share

So now it's your turn. Please share your screen with me.	

Teacher Starts Slideshow
Slide 14 to 17
Refer to speaker notes and follow the instructions on each slide.

We have one more class challenge for you. Can you solve it? Let's try. I will guide you through it.	
Teacher Ends Slideshow	
STUDENT-LED ACTIVITY - 20 mins	
● **Ask the student to press the ESC key to come back to the panel.** ● **Guide the student to start Screen Share.** ● **The teacher gets into Full Screen.**	
Student Initiates Screen Share	
ACTIVITY ● **Run the episodes multiple times to train the agent** ● **Code to find the optimal path using final Q-matrix**	
Teacher Action	**Student Action**
Open Student Activity 1 for the boilerplate code. Now write the code for the **run_episode()** function. also, Write the train function. Call the **train()** function and run at least 20 episodes. Why more episodes are required? ***Note:*** *Guide the student to write these functions and check the Q-table after completion of each episode.* So, when we look at the **q_table** we find that the rewards are calculated for different states and actions. For a particular state the maximum rewards are given to the action which should be taken to find the optimal path to reach the goal.	**ESR:** To train the model well and maximize the reward.



```
Reached Goal!
Ending Episode:  19
Training completed!
Final Q Table:
 [[  0.    0.    0.    0.   80.    0. ]
 [  0.    0.    0.   64.    0.  100. ]
 [  0.    0.    0.   64.    0.    0. ]
 [  0.   80.   51.2   0.    0.    0. ]
 [ 64.    0.    0.    0.    0.  100. ]
 [  0.    0.    0.    0.    0.    0. ]]
```

We have got the final Q-matrix now. You can see that the Q-matrix is updated.

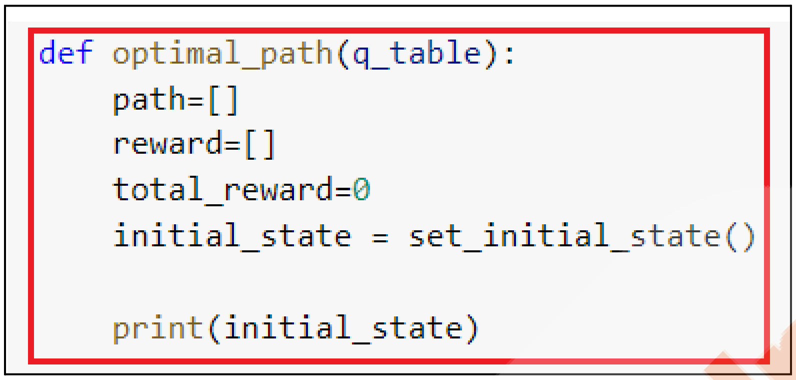
Say for state 1 the possible actions are 3 or 5, but 5 will make it reach the goal thus rewards are maximized at (3,5). Similarly, you can check for 4. This will now help the agent to find the optimal path.

***Note:** The values in the q-table may vary depending upon the initial states and actions taken during the training. Rewards for the goal will be always maximum.*

To choose the optimal path now we'll use our model which in our case is **q_table.**

To find the optimal path,

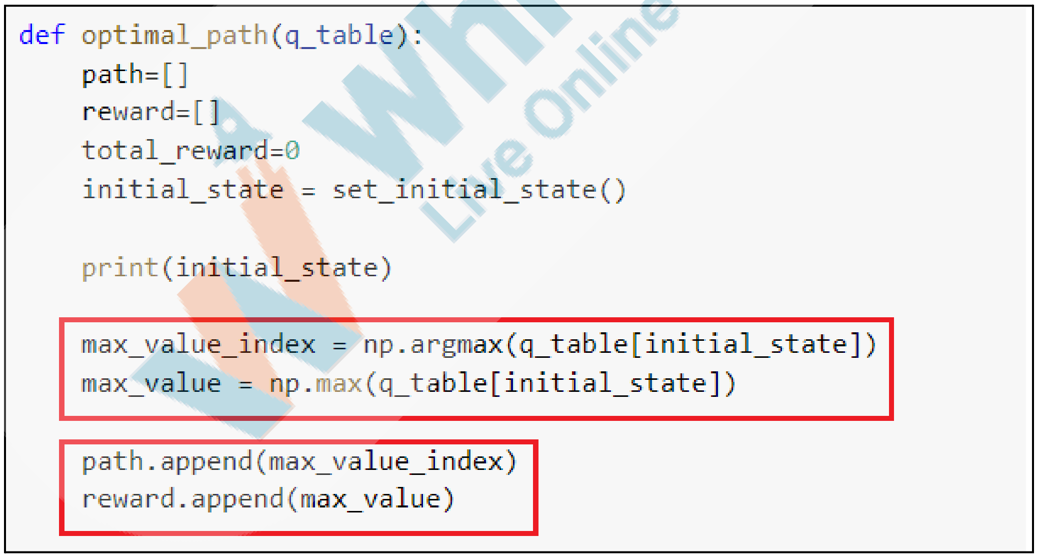
1. Create a function called **optimal path()**. This function requires **q_table** only.
2. Create two lists, one to store the states the agent will come across while moving from initial state to the goal. Next, to store the maximum reward.
3. Set **total_rewards=0**
4. Call the **set_initial_state()** function to get the initial state randomly.



```
def optimal_path(q_table):
    path=[]
    reward=[]
    total_reward=0
    initial_state = set_initial_state()

    print(initial_state)
```

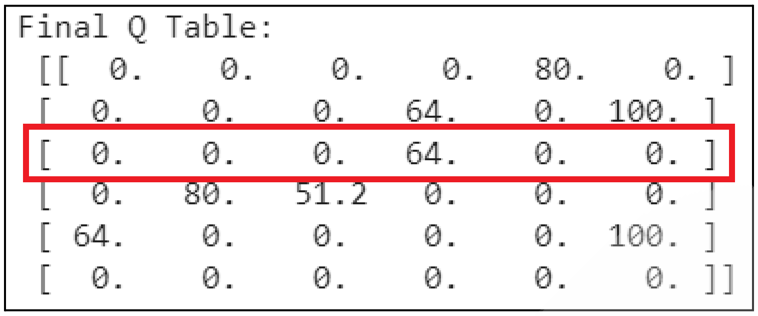
Now, whichever initial state is chosen, we need to find the maximum rewards for taking action. Thus, we'll look into the **q_table** and go to the index number of the initial state. We'll look for maximum rewards.

5. Use the **argmax()** function to find the index (**action**) at which maximum reward is present. Store this in the variable **max_value_index** variable.
6. Store the maximum rewards in variable **max_value**.
7. Append these values in path and reward respectively.



```
def optimal_path(q_table):
    path=[]
    reward=[]
    total_reward=0
    initial_state = set_initial_state()

    print(initial_state)

    max_value_index = np.argmax(q_table[initial_state])
    max_value = np.max(q_table[initial_state])

    path.append(max_value_index)
    reward.append(max_value)
```

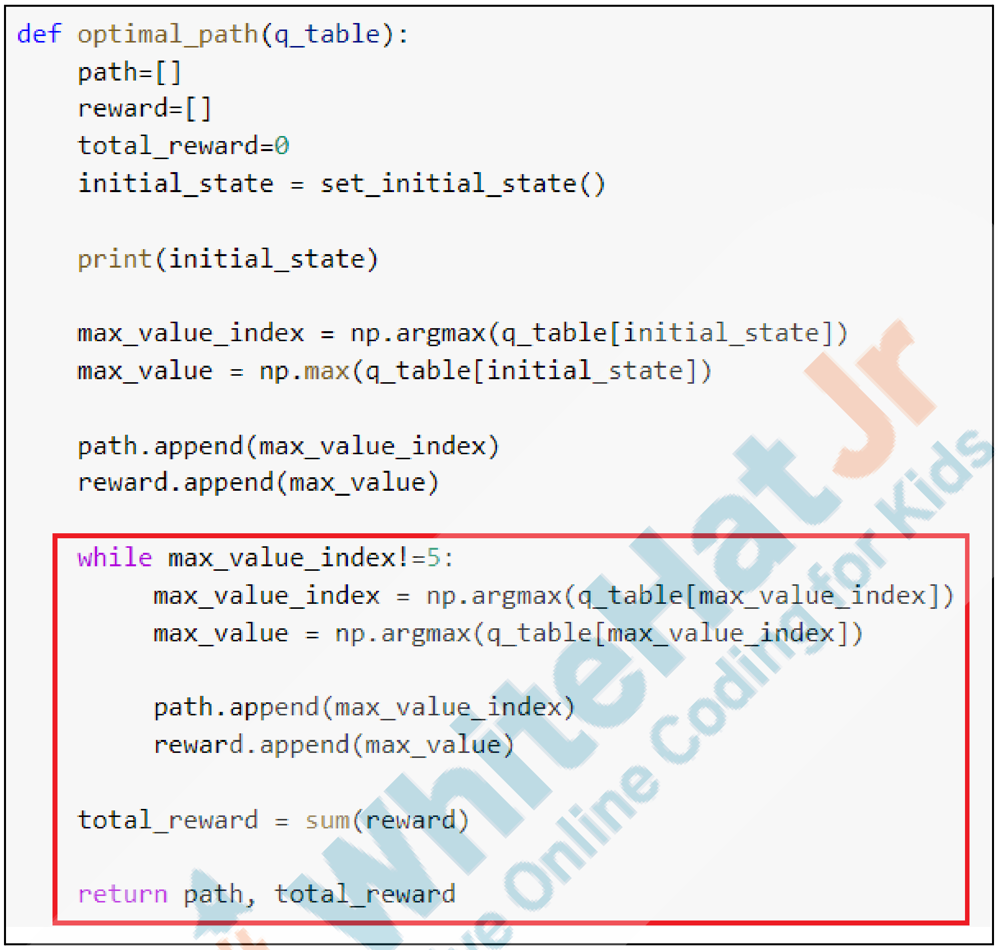
Referring to the **q_table** say if we initial state is 2, now the maximum value is present for action 3. So, the **max_value_index** will be 3 and the maximum reward is 64. Thus, the **max_value** is 64.



```
Final Q Table:
 [[  0.     0.     0.     0.    80.     0. ]
 [  0.     0.     0.    64.     0.   100. ]
 [  0.     0.     0.    64.     0.     0. ]
 [  0.    80.    51.2    0.     0.     0. ]
 [ 64.     0.     0.     0.     0.   100. ]
 [  0.     0.     0.     0.     0.     0. ]]
```

If the index or the action is **5** then the agent will reach the goal. So first we'll check whether the initial state is **5** or not. If not **5**, then proceed using the **while** loop. We'll follow the same steps as we did above.

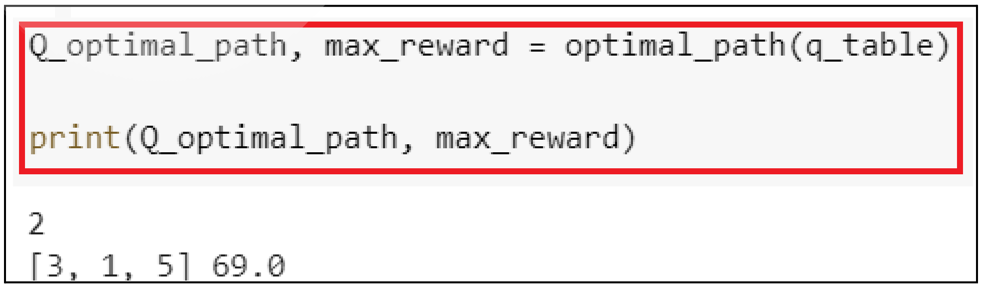
1. Find the maximum reward and its index.
2. Append these in the list's path and reward.
3. If the next index is **5** then the while loop will break.
4. The total reward will be the sum of all the rewards in the reward list.
5. The **optimal_path()** function will return the path and total reward.



```python
def optimal_path(q_table):
    path=[]
    reward=[]
    total_reward=0
    initial_state = set_initial_state()

    print(initial_state)

    max_value_index = np.argmax(q_table[initial_state])
    max_value = np.max(q_table[initial_state])

    path.append(max_value_index)
    reward.append(max_value)

    while max_value_index!=5:
        max_value_index = np.argmax(q_table[max_value_index])
        max_value = np.argmax(q_table[max_value_index])

        path.append(max_value_index)
        reward.append(max_value)

    total_reward = sum(reward)

    return path, total_reward
```

Let’s call this function to find the path and reward for any initial state. Since this function returns two values, we’ll store them in two variables called **Q_optimal_path** and **max_reward.**

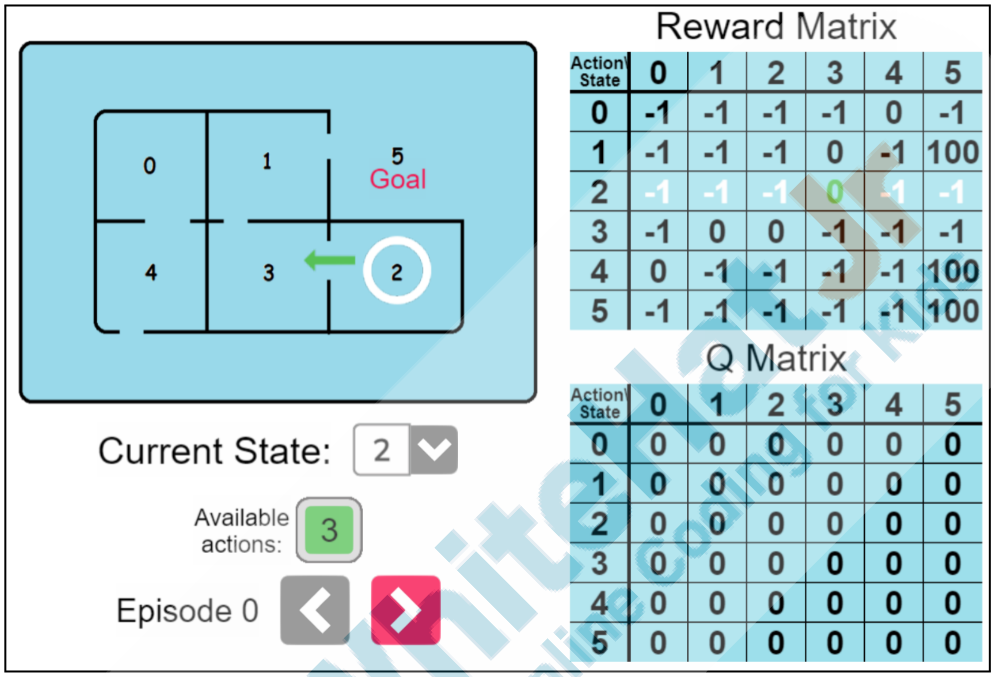
Print these values and cross-check the output.



```python
Q_optimal_path, max_reward = optimal_path(q_table)

print(Q_optimal_path, max_reward)
```

```
2
[3, 1, 5] 69.0
```

So the initial state is **2** and the optimal path is **3, 1, 5**. The rewards are calculated as **69**.
Let's cross-check this with Student Activity 2. For state 2 it will suggest 3:



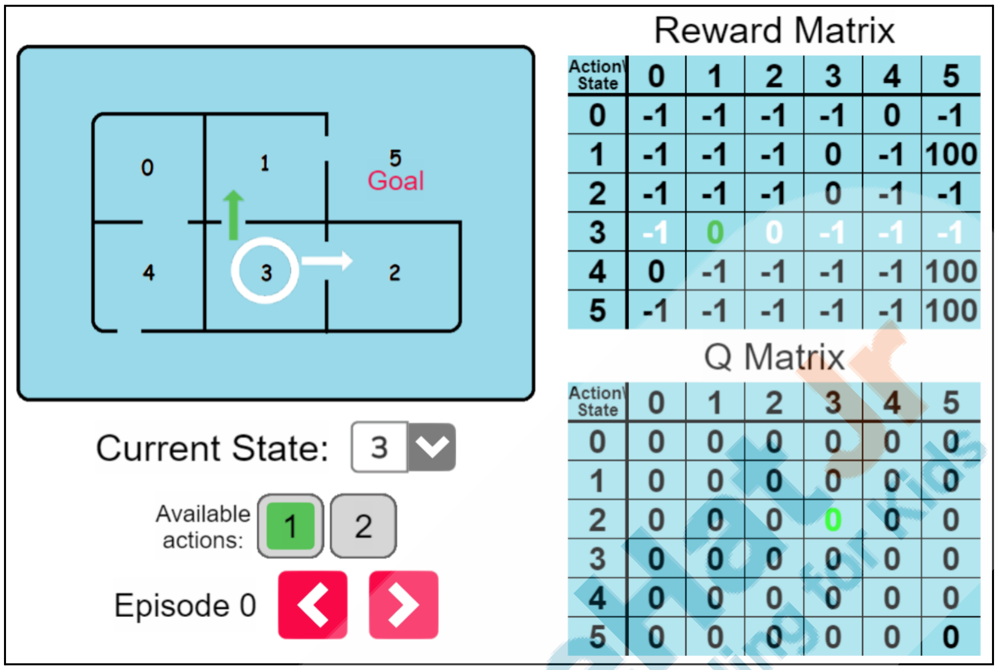
Reward Matrix

Action\State	0	1	2	3	4	5
0	-1	-1	-1	-1	0	-1
1	-1	-1	-1	0	-1	100
2	-1	-1	-1	0	-1	-1
3	-1	0	0	-1	-1	-1
4	0	-1	-1	-1	-1	100
5	-1	-1	-1	-1	-1	100

Q Matrix

Action\State	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0

For state 3 it has two choices either 1 or 2. It has chosen 1.



Reward Matrix

Action\State	0	1	2	3	4	5
0	-1	-1	-1	-1	0	-1
1	-1	-1	-1	0	-1	100
2	-1	-1	-1	0	-1	-1
3	-1	0	0	-1	-1	-1
4	0	-1	-1	-1	-1	100
5	-1	-1	-1	-1	-1	100

Q Matrix

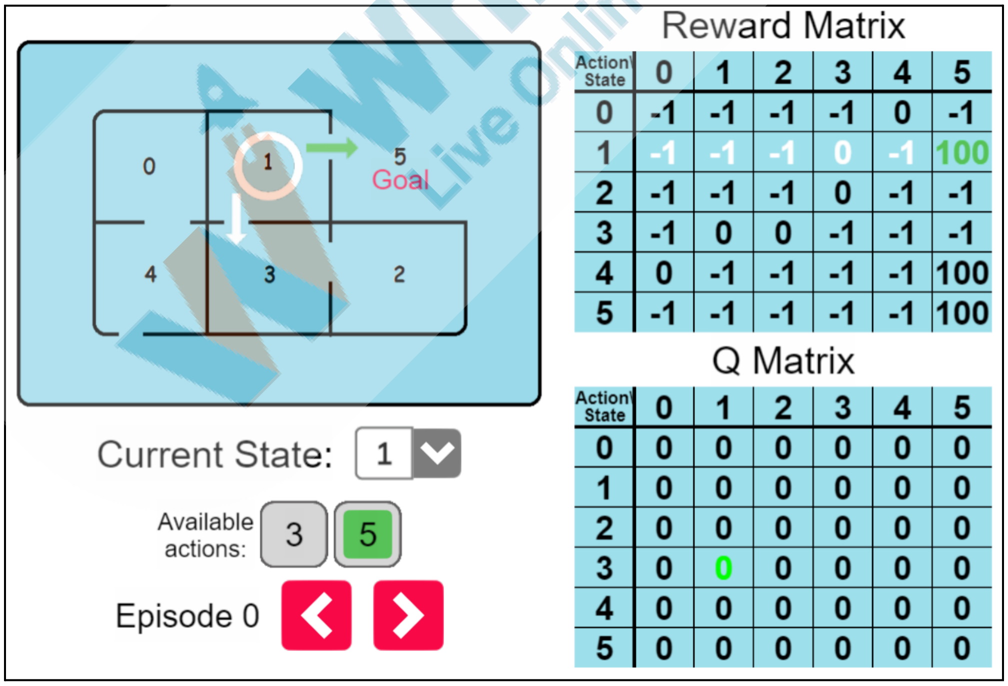
Action\State	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0

Current State: 3

Available actions: 1 2

Episode 0

For state 1, it has two choices either 3 or 5. If 5 is chosen then it'll be the goal!



Reward Matrix

Action\State	0	1	2	3	4	5
0	-1	-1	-1	-1	0	-1
1	-1	-1	-1	0	-1	100
2	-1	-1	-1	0	-1	-1
3	-1	0	0	-1	-1	-1
4	0	-1	-1	-1	-1	100
5	-1	-1	-1	-1	-1	100

Q Matrix

Action\State	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	0	0	0	0	0	0
4	0	0	0	0	0	0
5	0	0	0	0	0	0

Current State: 1

Available actions: 3 5

Episode 0

Great!!! We were able to train the RL model and also, the agent was able to find the optimal path for different states. This way any agent can be trained to take action in different environments.	
Teacher Guides Student to Stop Screen Share	
WRAP-UP SESSION - 05 mins	
Teacher Starts Slideshow **Slide 18 to 23**	
Activity details **Following are the WRAP-UP session deliverables:** ● Appreciate the student. ● Revise the current class activities. ● Discuss the quizzes.	
WRAP-UP QUIZ Click on In-Class Quiz	
Continue WRAP-UP Session **Slide 24 to 29**	
Activity Details **Following are the session deliverables:** ● Explain the facts and trivia ● Next class challenge ● Project for the day ● Additional Activity (Optional)	
FEEDBACK	

- **Appreciate and compliment the student for trying to learn a difficult concept.**
- **Get to know how they are feeling after the session.**
- **Review and check their understanding.**

Teacher Action	**Student Action**
You get "hats-off" for your excellent work! In the next class, we'll learn to access data from any webpage and explore planets beyond our galaxy.	*Make sure you have given at least 2 hats-off during the class for:* Creatively Solved Activities +10 Great Question +10 Strong Concentration +10

PROJECT OVERVIEW DISCUSSION

Refer the document below in Activity Links Sections

Teacher Clicks

ACTIVITY LINKS		
Activity Name	**Description**	**Links**
Teacher Activity 1	Available action	https://whitehatjrcontent.s3.ap-south-1.amazonaws.com/Teacher-Resources/COCOS_Applets/POC/Coding/SimpleQ-RL/noEpisodes/index.html
Teacher Activity 2	Boilerplate Code	https://colab.research.google.com/drive/13JNfnFZ0lRsvdTCgsobdLtQsidYf0Aah?usp=sharing
Teacher Activity 3	Run Episodes	https://whitehatjrcontent.s3.ap-south-1.amazonaws.com/Teacher-Resources/COCOS_Applets/POC/Coding/SimpleQ-RL/final/index.html
Teacher Activity 4	Reference Code	https://colab.research.google.com/drive/1l1ivjuBlZ0xrTq4O3uFbww4Hs1ErGuib?usp=sharing
Teacher Reference 1	Project	https://s3-whjr-curriculum-uploads.whjr.online/e6c672d0-5737-4f4b-a747-f1ac3cb5e754.pdf
Teacher Reference 2	Project Solution	https://colab.research.google.com/drive/1cbOPQq31uHcPB03enaKAZYm_BlHqeMNN?usp=sharing
Teacher Reference 3	Visual-Aid	https://s3-whjr-curriculum-uploads.whjr.online/e940c749-7564-4696-b84a-86103952a625.html
Teacher Reference 4	In-Class Quiz	https://s3-whjr-curriculum-uploads.whjr.online/f6d3eced-4a0c-4d4b-ad67-7d0967d65c83.pdf
Student Activity 1	Boilerplate Code	https://colab.research.google.com/drive/1IpkuDMnfWWxcdBjC1_emHF7DJYJiz2Tf?usp=sharing
Student Activity 2	Run Episodes	https://whitehatjrcontent.s3.ap-south-1.amazonaws.com/Teacher-Resources/COCOS_Applets/POC/Coding/SimpleQ-RL/final/index.html