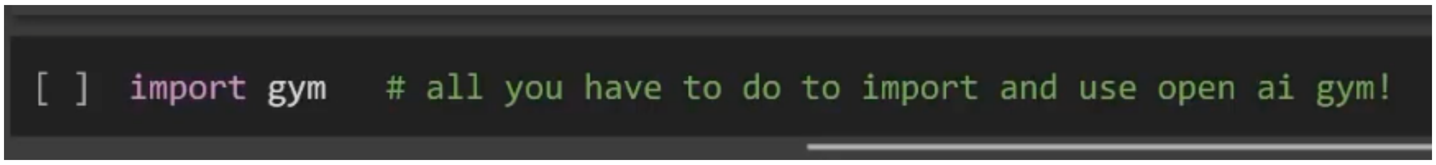
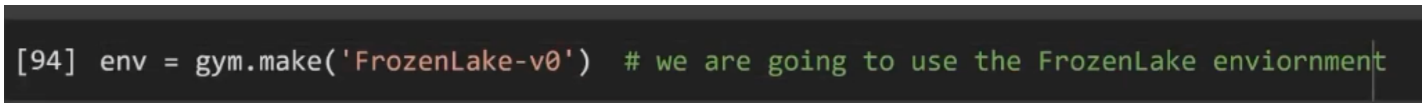
- ***using open AI gym to do Q-learning***
  - if the agent does not have a good balance of taking random actions and using learned actions -> the agent will always try to maximise its reward for the current state/action, leading to local maxima
  - -> he imports Open AI's gym



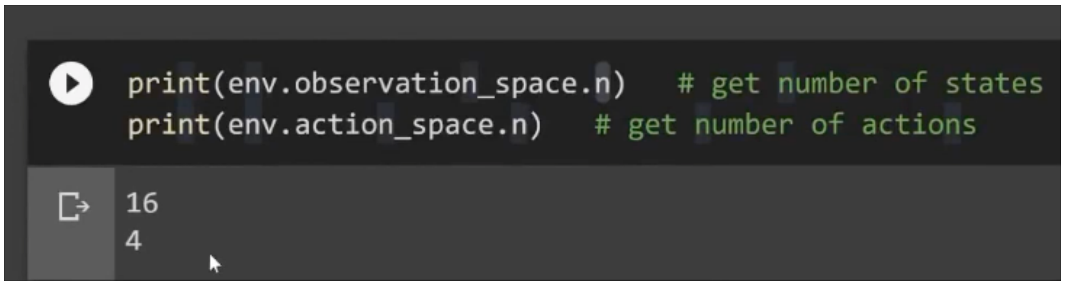
```
[ ]  import gym   # all you have to do to import and use open ai gym!
```

    - -> graphical environments for training re-enforcement models
  - ***-> then sets up the environment we are going to use***



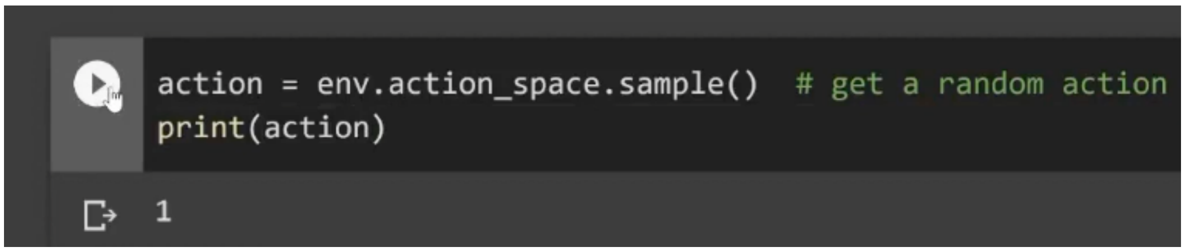
```
[94] env = gym.make('FrozenLake-v0')  # we are going to use the FrozenLake enviornment
```

    - -> there is an observation and action space for each variable
    - -> setting the number of states and the actions which can be taken
    - -> there are commands to move around the environment
    - -> you can also select a random action and return the value



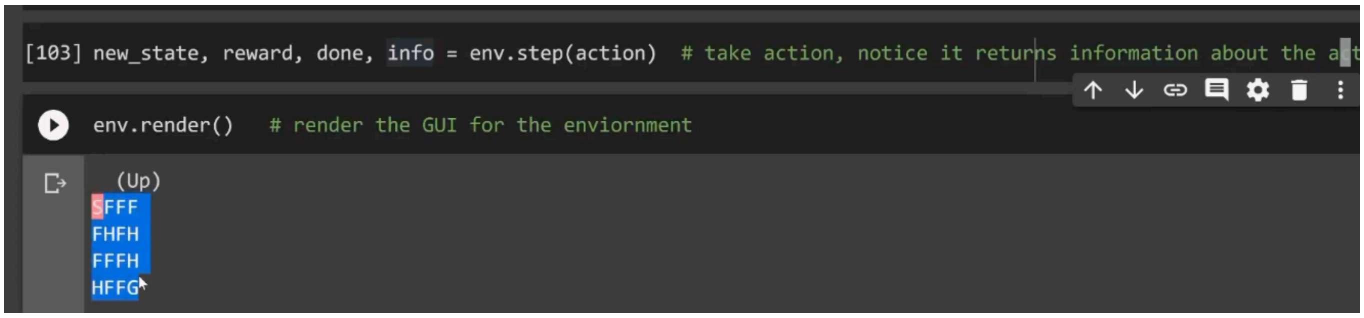
```
print(env.observation_space.n)   # get number of states
print(env.action_space.n)   # get number of actions

16
4
```

    - ***-> you can also take a random action in the action space***



```
action = env.action_space.sample()  # get a random action
print(action)

1
```

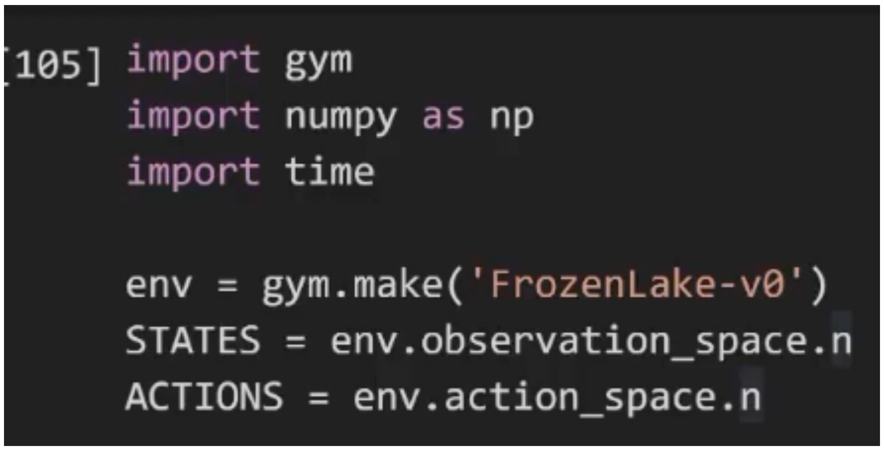
      - -> it returns the index of a random action
    - ***-> then taking the action and performing it in the environment***
      - -> the observation / state which we move into
      - -> this includes the reward and has 'done' (if the agent lost or won and is therefore no longer in a valid state in the environment)
    - ***-> watching the agent do the training***



```
[103] new_state, reward, done, info = env.step(action)  # take action, notice it returns information about the a

env.render()   # render the GUI for the enviornment

  (Up)
SFFF
FHFH
FFFH
HFFG
```

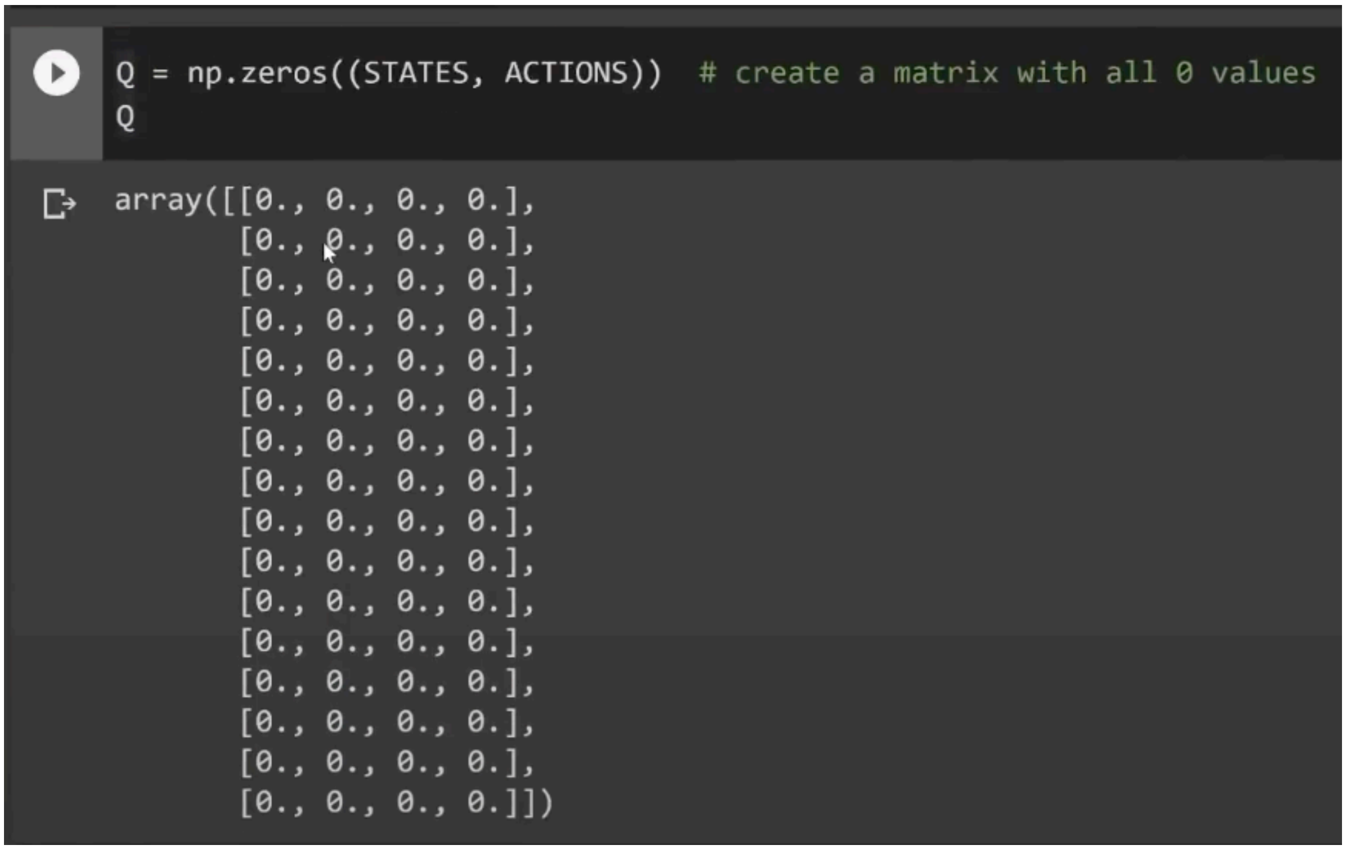
      - -> start vs frozen vs hole
      - -> re-running this
- ***-> implementing Q-learning to train the model***

- -> import the modules
- -> set the number of states equal to n
- -> similarly with the action space



```
[105] import gym
      import numpy as np
      import time

      env = gym.make('FrozenLake-v0')
      STATES = env.observation_space.n
      ACTIONS = env.action_space.n
```

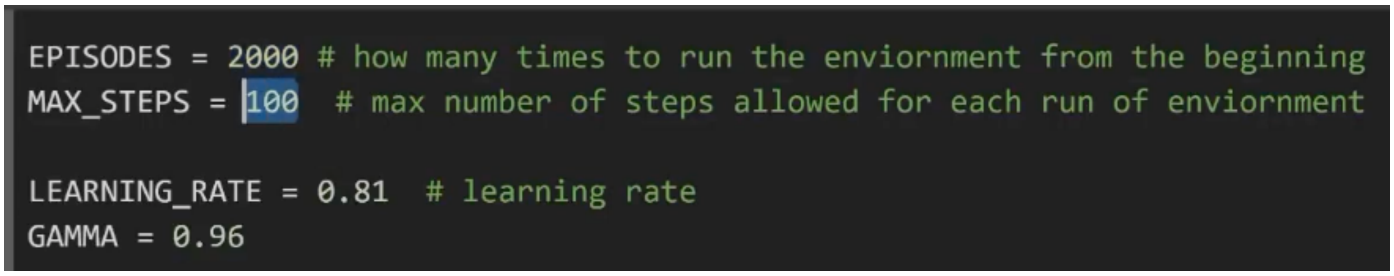
- -> initialising the Q-values / table
    - -> the actions are more likely to be random at the beginning of training the model



```
Q = np.zeros((STATES, ACTIONS))  # create a matrix with all 0 values
Q
```

```
array([[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.],
       [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.]])
```

- ***-> building a 16x4 matrix***
    - -> this matrix is the Q-value matrix
    - -> the rows are states, the columns are actions and the values are the rewards which the agent gets for taking that action in that state

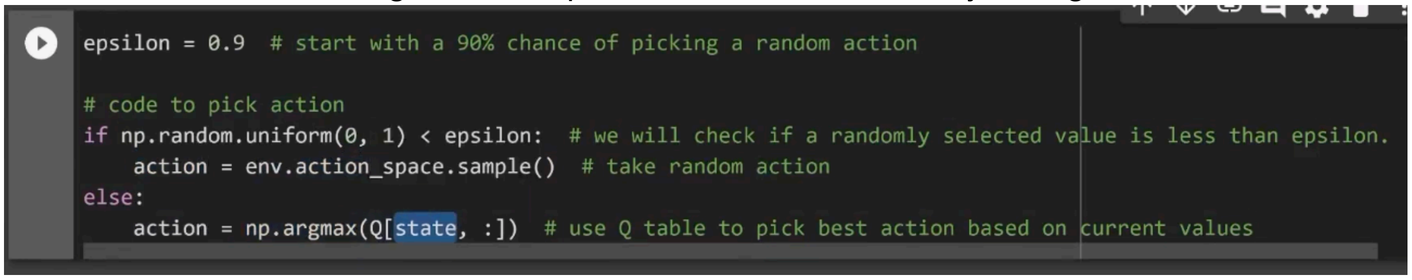


```
EPISODES = 2000 # how many times to run the enviornment from the beginning
MAX_STEPS = 100  # max number of steps allowed for each run of enviornment

LEARNING_RATE = 0.81  # learning rate
GAMMA = 0.96
```

- ***-> initialising the constants***
    - -> the number of episodes is the number of episodes you want to train the agent on

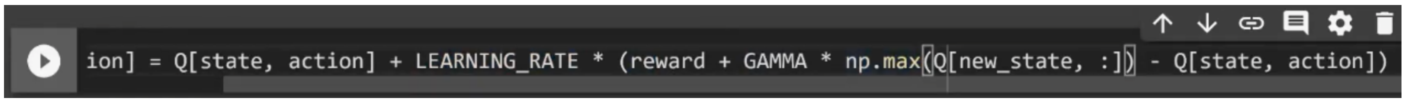
- -> max_steps is the maximum number of steps the agent can take before being cut off
  - -> to avoid getting stuck in a local minimum
- -> learning rate <- the higher this is the faster the model changes (it becomes less conservative in the choices it makes)

- ***-> picking an action***
  - -> you can either randomly pick an action or use the entire Q-value table to pick the most rewarding action
  - -> epsilon <- setting the percentage chance you will pick a random action
  - -> the model starts being able to explore in the environment by taking the actions



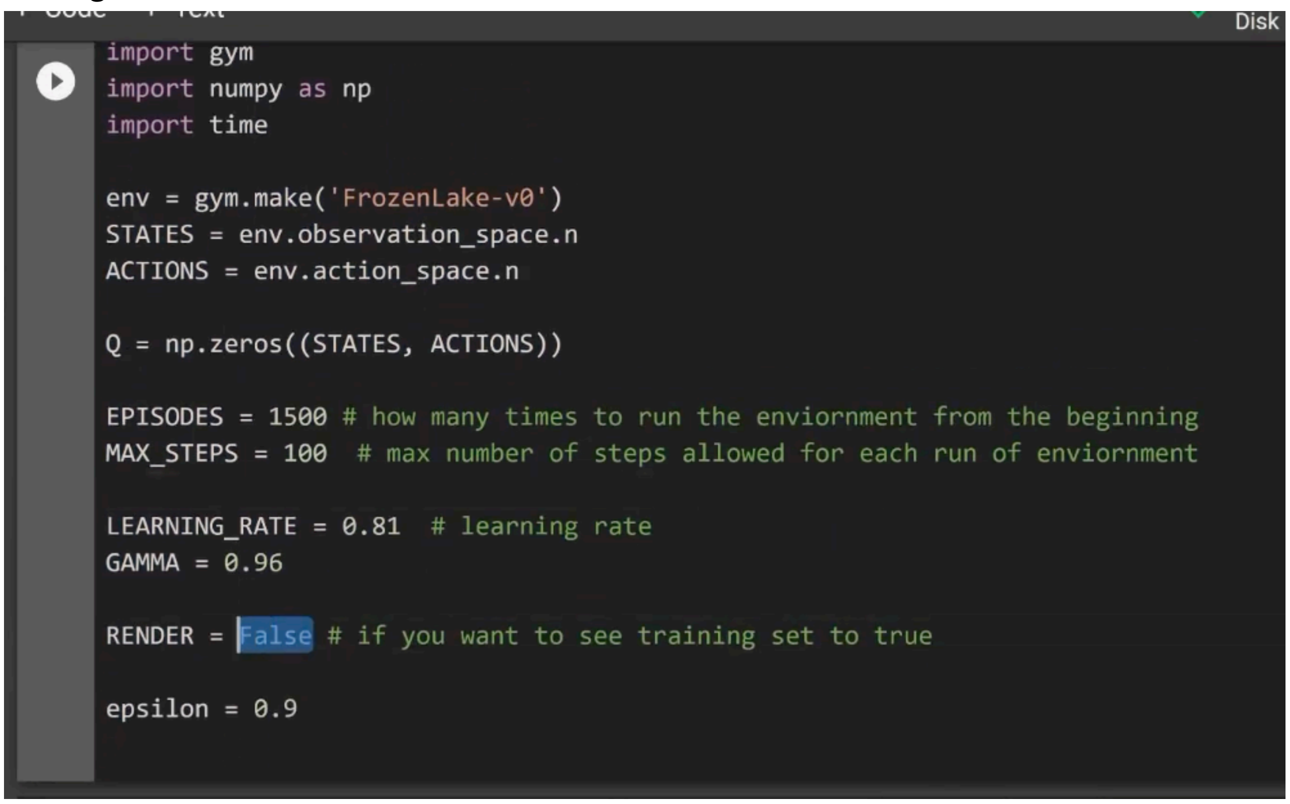
```
epsilon = 0.9  # start with a 90% chance of picking a random action

# code to pick action
if np.random.uniform(0, 1) < epsilon:  # we will check if a randomly selected value is less than epsilon.
    action = env.action_space.sample()  # take random action
else:
    action = np.argmax(Q[state, :])  # use Q table to pick best action based on current values
```

  - -> then decreasing the epsilons
    - -> np.random.uniform < epsilon
      - -> taking the argument max of the state row in the Q-table
      - -> finding the maximum value of the Q-table and returning it index



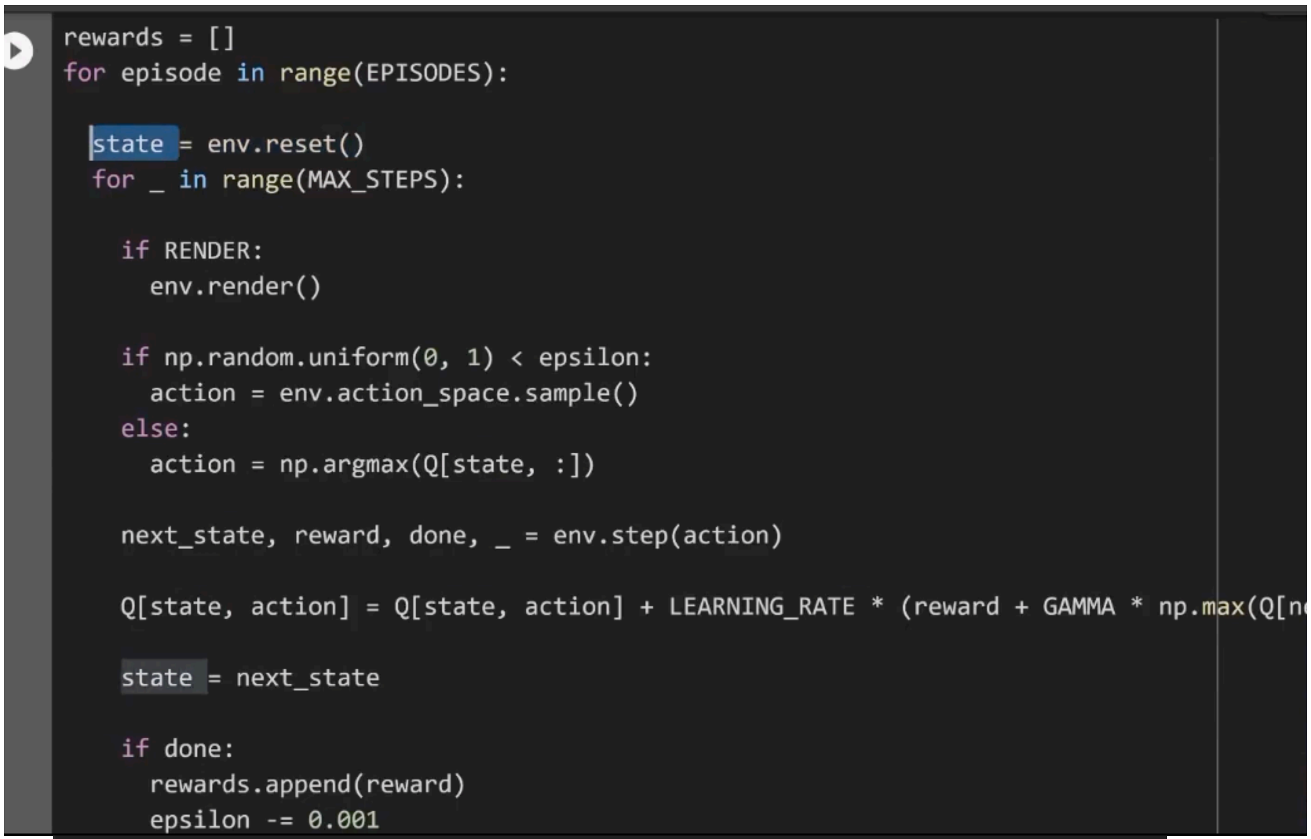
```
ion] = Q[state, action] + LEARNING_RATE * (reward + GAMMA * np.max(Q[new_state, :]) - Q[state, action])
```

- ***-> updating the Q values***

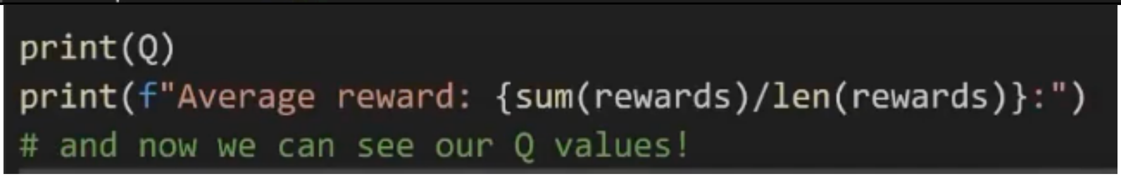


```
import gym
import numpy as np
import time

env = gym.make('FrozenLake-v0')
STATES = env.observation_space.n
ACTIONS = env.action_space.n

Q = np.zeros((STATES, ACTIONS))

EPISODES = 1500 # how many times to run the enviornment from the beginning
MAX_STEPS = 100  # max number of steps allowed for each run of enviornment

LEARNING_RATE = 0.81  # learning rate
GAMMA = 0.96

RENDER = False # if you want to see training set to true

epsilon = 0.9
```

- ***-> training the Q-table***
  - -> to make the model better you train it on more episodes
  - -> there is a rewards list

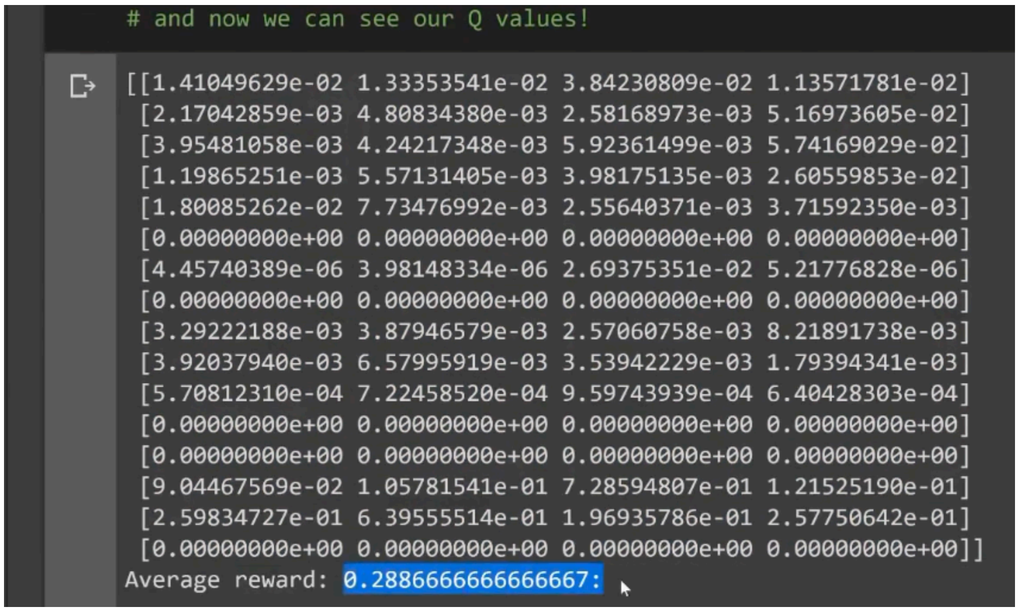


```
rewards = []
for episode in range(EPISODES):

  state = env.reset()
  for _ in range(MAX_STEPS):

    if RENDER:
      env.render()

    if np.random.uniform(0, 1) < epsilon:
      action = env.action_space.sample()
    else:
      action = np.argmax(Q[state, :])

    next_state, reward, done, _ = env.step(action)

    Q[state, action] = Q[state, action] + LEARNING_RATE * (reward + GAMMA * np.max(Q[n

    state = next_state

    if done:
      rewards.append(reward)
      epsilon -= 0.001
```



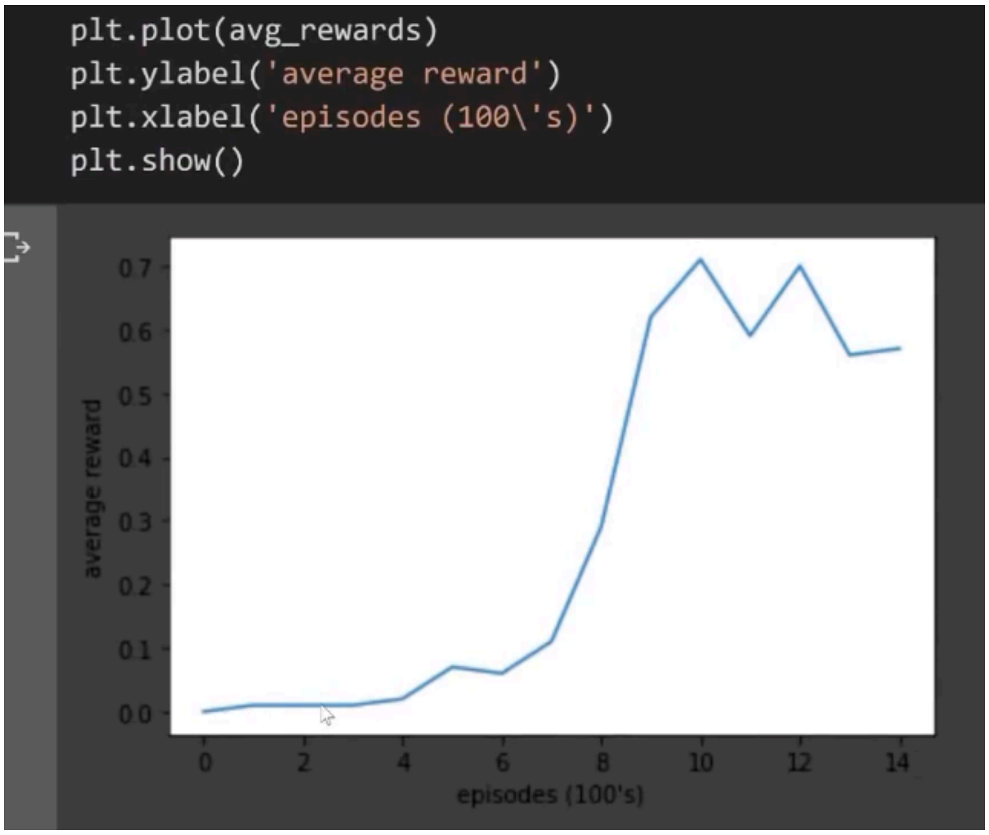
```
print(Q)
print(f"Average reward: {sum(rewards)/len(rewards)}:")
# and now we can see our Q values!
```

- -> explore the environment up to the maximum steps
- -> rendering the environment -> otherwise take an action
- -> an action is being taken for each time step
- -> next_state, reward, done <- taking the action
- -> then updating the Q-value using the equation
- -> then setting the current state to the next state -> so the agent is in a new state which it can start exploring
- -> appending whichever reward came from the last step into the rewards
- -> you get one reward if you move to a valid block and 0 if it doesn't work



```
# and now we can see our Q values!

[[1.41049629e-02 1.33353541e-02 3.84230809e-02 1.13571781e-02]
 [2.17042859e-03 4.80834380e-03 2.58168973e-03 5.16973605e-02]
 [3.95481058e-03 4.24217348e-03 5.92361499e-03 5.74169029e-02]
 [1.19865251e-03 5.57131405e-03 3.98175135e-03 2.60559853e-02]
 [1.80085262e-02 7.73476992e-03 2.55640371e-03 3.71592350e-03]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [4.45740389e-06 3.98148334e-06 2.69375351e-02 5.21776828e-06]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [3.29222188e-03 3.87946579e-03 2.57060758e-03 8.21891738e-03]
 [3.92037940e-03 6.57995919e-03 3.53942229e-03 1.79394341e-03]
 [5.70812310e-04 7.22458520e-04 9.59743939e-04 6.40428303e-04]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
 [9.04467569e-02 1.05781541e-01 7.28594807e-01 1.21525190e-01]
 [2.59834727e-01 6.39555514e-01 1.96935786e-01 2.57750642e-01]
 [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]]
Average reward: 0.2886666666666667:
```

        - -> the epsilon then changes
    - -> it then returns the average reward



```
plt.plot(avg_rewards)
plt.ylabel('average reward')
plt.xlabel('episodes (100\'s)')
plt.show()
```

    - -> the Q-values are stored in a matrix
        - -> the reward rate relative to the epsilon
        - -> you can also use the Q-table values (for example without updating them)
        - -> there are other ways of implementing this learning method
- -> next is a conclusion for the module