Q-Learning Ttreasure hunt in a maze . The game is as follows :

You start at a given position, the starting state . From any state you can go left, right, up or down or stay in the same place provided you don't cross the premises of the maze. Each action will take you to a cell of the grid (a different state). Now, there is a treasure chest at one of the states (the goal state). Also, the maze has a pit of snakes in certain positions/states. Your goal is to travel from the starting state to the goal state by following a path that doesn't have snakes in it.

Start		Snakes	
-	Snakes	Snakes	
-	Snakes		
-	-	-	Treasure

Grid outline of the maze

When you place an agent in the grid(we will refer to it as our environment) it will first explore. It doesn't know what snakes are , neither does it know what or where the treasure is. So, to give it the idea of snakes and the treasure chest we will give some rewards to it after it takes each action. For every snake pit it steps onto we will give it a reward of -10. For the treasure we will give it a reward of +10. Now we want our agent to complete the task as fast as possible (to take the shortest route). For this, we will give rest of the states a reward of -1. Then we will tell it to maximise the score. Now as the agent explores , it learns that snakes are harmful for it, the treasure is good for it and it has to get the treasure as fast as possible. The '-' path in the figure shows the shortest path with maximum reward.

Q–Learning attempts to learn the value of being in a given state, and taking a specific action there.

What we will do is develop a table. Where the rows will be the states and the columns are the actions it can take. So, we have a 16x5 (80 possible state-action) pairs where each state is one cell of the maze-grid.
We start by initializing the table to be uniform (all zeros), and then as we observe the rewards we obtain for various actions, we update the table accordingly. We will update the table using the ***Bellman Equation .***

$$Q(s,a) = r + \gamma(max(Q(s',a'))$$

'S' represents the current state . 'a' represents the action the agent takes from the current state. ' S' ' represents the state resulting from the action. 'r' is the reward you get for taking the action and 'γ' is the discount factor. So, the Q value for the state 'S' taking the action 'a' is the sum of the instant reward and the discounted future reward (value of the resulting state). The discount factor 'γ' determines how much importance you want to give to future rewards. Say, you go to a state which is further away from the goal state, but from that state the chances of encountering a state with snakes is less, so, here the future reward is more even though the instant reward is less.
We will refer to each iteration(attempt made by the agent) as an episode. For each episode, the agent will try to achieve the goal state and for every transition it will keep on updating the values of the Q table.
Lets see how to calculate the Q table :
For this purpose we will take a smaller maze-grid for ease.

Start State - 1	State - 2
Snake State - 3	Treasure State - 4

The initial Q-table would look like (states along the rows and actions along the columns) :

```
   U D L R N
1[ 0 0 0 0 0 ]
2[ 0 0 0 0 0 ]
3[ 0 0 0 0 0 ]
4[ 0 0 0 0 0 ]
```

Q Matrix

U — up, D — down, L — left, R — right

The reward table would look like :

	U	D	L	R	N
1[	E	-10	E	-1	-1]
2[	E	+10	-1	E	-1]
3[	-1	E	E	+10	-1]
4[	-1	E	-10	E	+10]

R Matrix

Here, E represents NULL, i.e., there can be no such transitions.
Algorithm:

1. Initialise Q-matrix by all zeros. Set value for 'γ'. Fill rewards matrix.
2. For each episode. Select a random starting state (here we will restrict our starting state to state-1).
3. Select one among all possible actions for the current state (S).
4. Travel to the next state (S') as a result of that action (a).
5. For all possible actions from the state (S') select the one with the highest Q value.
6. Update Q-table using eqn.1 .
7. Set the next state as the current state.
8. If goal state reached then end.

Example : Lets say we start with state 1 . We can go either D or R. Say, we chose D . Then we will reach 3 (the snake pit). So, then we can go either U or R . So, taking γ = 0.8, we have :
Q(1,D) = R(1,D) + γ*[max(Q(3,U) & Q(3,R))]
Q(1,D) = -10 + 0.8*0 = -10
Here, max(Q(3,U) & Q(3,R)) = 0 as Q matrix not yet updated. -10 is for stepping on the snakes. So, new Q-table looks like :

	U	D	L	R	N
1[	0	-10	0	0	0]
2[	0	0	0	0	0]
3[	0	0	0	0	0]
4[	0	0	0	0	0]

Now, 3 is the starting state. From 3, lets say we go R. So, we go on 4 . From 4, we can go U or L .
Q(3,R) = R(3,R) + 0.8*[max(Q(4,U) & Q(4,L))]
Q(3,R) = 10 + 0.8*0 = 10

	U	D	L	R	N
1[	0	-10	0	0	0]
2[	0	0	0	0	0]
3[	0	0	0	10	0]
4[	0	0	0	0	0]

So, now we have reached the goal state 4. So, we terminate and more passes to let our agent understand the value of each state and action. Keep making passes until the values remain constant. This means that your agent has tried out all possible state-action pairs.
Implementation in python :

```python
import numpy as np
import random
import matplotlib.pyplot as plt

gamma = 0.8

#for matrices reward and q_matrix columns are in order (U, D, L, R, N)

# here the states are 0, 1, 2, 3 for convenience
# 0 is the starting state
# 1 is state to the right of 0
# 2 is the snake-pit state
# 3 is the treasure(goal state)

reward = np.array([[0, -10, 0, -1, -1],
                   [0, 10, -1, 0, -1],
                   [-1, 0, 0, 10, -1],
                   [-1, 0, -10, 0, 10]])

q_matrix = np.zeros((4,5))

# -1 represent invalid transitions
transition_matrix = np.array([[-1, 2, -1, 1, 1],
                              [-1, 3, 0, -1, 2],
                              [0, -1, -1, 3, 3],
                              [1, -1, 2, -1, 4]])

# for valid actions
# encoded up as 0, down as 1, left as 2, right as 3, no action as 4
# the rows are the states
valid_actions = np.array([[1, 3, 4],
                          [1, 2, 4],
                          [0, 3, 4],
                          [0, 2, 4]])

for i in range(1000): # 1000 episodes
    start_state = 0
    current_state = start_state
    while current_state != 3:
        action = random.choice(valid_actions[current_state])
        next_state = transition_matrix[current_state][action]
        future_rewards = []
        for action_nxt in valid_actions[next_state]:
            future_rewards.append(q_matrix[next_state][action_nxt])
        q_state = reward[current_state][action] + gamma*max(future_rewards)
        q_matrix[current_state][action] = q_state
        print(q_matrix)
        current_state = next_state
        if current_state == 3:
            print('goal state reached')

print('final q-matrix : ')
print(q_matrix)
```

Output for the last q_matrix :

```
final q-matrix :
[[  0.   -2.    0.    7.    7. ]
 [  0.   10.    4.6   0.    7. ]
 [  4.6   0.    0.   10.   -1. ]
 [  0.    0.    0.    0.    0. ]]
```

Credits: Aneek Das