Q learning tutorial from here:
https://www.youtube.com/watch?v=TiAXhVAZQI8

There are two reinforcement learning algorithms: value based methods and policy based methods.
End of the day: we want to maximize total reward.
Value based determines a value function that quantifies total reward. From this method it will determine an optimal policy

Policy based algorithms are based directly on optimal policy. A policy is how an agent behaves in a given situation.

Q learning is a value based method.

Value based methods determines a policy which maximizes total reward.
There are inputs and outputs to a value function.

There are state value functions (which can be represented by V(s) and state action value functions (which can be represented by Q(s, a))

A state is a snapshot of the environment, while an action is the decision taken by this agent in an environment.

In a state value function, a function will take in a state and output a real number. A state action value function will take the state and action as input and output a real number. The number returned is a q value.

The number returned from a state essentially says “how good is it to be in the state s”. On the other hand, a state action value function says “how good is it to be in the state s and take an action a in this state?”

It’s easy to think of q learning as a table. You’ll have a variety of states and possible actions at each of those states. Each cell will be the q value for that given state and action.

Example of this

Q	a1	a2	a3
s1	Q(s1,a1)	Q(s1,a2)	Q(s1,a3)
s2	Q(s2,a1)	Q(s2,a2)	Q(s2,a3)
s3	Q(s3,a1)	Q(s3,a2)	Q(s3,a3)

Also let's say we have the following as well (arbitrary rewards for each particular state) - this particular example is straight from the video.

Q	left	right	up	down
s1	-0.5	~~1~~ 0.815	2.1	1.3
s2	0.5	0.75	-0.5	~~1.5~~ 1.267
s6	-1.2	1.2	0.7	1.7

So here, Q(s1,a1) represents the reward when action 1 is taken at state 1. We want the total value to be maximized.

If these were filled with -1s besides Q(s3,a3) which has +10 and Q(s3,a2), which has -10, how do we get to Q(s3,a3)? We want the agent to learn the optimal target policy.

To learn this optimal policy, we use q learning.
Let's say that initially, the agent just explores its environment and chooses something at random chance - the agent makes a decision based on behavior policy.

If the agent made the random decision to go right, we translate into another state. At this point, we can calculate the observed Q value.

The bellman equation:

$$Q(S_1, \text{right})_{\text{observed}} = R(S_2) + \gamma \max_a Q(S_2, a)$$

Q(s1, right) represents what happens when we start at s1 and go to the right. It is given by the sum of the reward at s2 plus the maximum future q value for this state. Gamma serves as a discount factor, which shows how much to value the current reward over future ones.

Looking at the q table, we get the maximum q value from going down on state 2.
Given a discount rate of 0.1, we get the following: -1 + 0.1 * 1.5 = -0.85
-1 = what happened when we went right from state 1 (the first table)
0.1 = the discount rate
1.5 = the maximum reward attainable from an action in state 2

We got -0.85 as a reward, but the actual reward that should be obtained is 1. This error is known as a temporal difference error. We are comparing q values of two different time steps and the difference is the error.

TD error = observed - expected -> -0.85 - 1 = -1.85
Now, we update the table. So Q(s1, right) = Q(s1, right) + ß x TD error, where ß represents the learning rate/ step size. It represents how much we are willing to change these q values in a particular step. Higher value = faster learning.
Say that ß = 0.1. So, Q(s1, right) = 1 + 0.1 * -1.85 = 0.815. We update the s1 right in the second table. (In these notes I strikethrough it)

We are now in S2 (top center). We can go left, down, or right from here. We are following a random policy and decide to go down. On taking the action, agent goes to another state, say, s6, and receives a reward based on this move.

Calculate the observed Q value: -1 + 0.1 * 1.7 = -0.83.

Here, the observed value was actually 1.5. So our TD value is -0.83 - 1.5 = -2.33 (observed - expected).

We now update the table based on this. So, with our same learning value / step size, we have 1.5 + 0.1 * -2.33 = 1.267

These actions are repeated until we get to the -10 or +10 spot. What we have done so far is “one episode”. We can perform multiple episodes choosing random actions until the values from the q table begin to come more stable. The values from the second table eventually go to particular numbers, which can lead the agent to the highest reward on the table (being able to reach the +10). The Q value dictates the policy to get the reward, the target policy, and it does this by following the behavior policy (which was choosing things at random in our case).

Q learning with python:
Source: https://www.geeksforgeeks.org/q-learning-in-python/
(Scroll to the bottom, the first part is similar to above)

First define the environment. We initialize the q-table by giving a possible number of states, actions, and a target state. This Q-table is first initialized with zeros.

We then must set various factors. In Q-learning, we have the learning rate (how much to value new information), a discount factor (how much to value current reward, lower = value more, higher = value less), an exploration probability (chance of selecting an action at random rather than the one with the highest q-value, and the number of rounds of training we go through (when we reset, that's defined as a round).

Now, we actually perform the the q-learning algorithm.

Start with a for loop that goes until hitting the maximum number of epochs declared (if you wanted to run until stopping the program manually, you could use a while loop instead).

First start from a random state. Then, check if this state is the goal state (we're done if this is the case).

In the likely case it is not:

- Choose an action with the epsilon-greedy strategy
- We either will have the off-chance of choosing a random value in the table or we say action = np.argmax(Q_table[current_state]). In numpy, argmax returns the index (or indices) of the highest value in the table.
- We then proceed to move to the next state - (next_state = (current_state + 1)
- We can have a simple reward function. Give a +1 reward for reaching the goal and 0 otherwise. (But if we want to make something optimal, like quantum gates, we should have a negative reward for each additional step it takes to reach the goal).
- Finally, we update the Q-value with the Q-learning rule. This looks like the following: Q_table[current_state, action] += learning_rate * (reward + discount_factor * np.max(Q_table[next_state]) - Q_table[current_state, action])