Homework 2: High-level overview of an RL code
Project Title: “Self-Driving Cab”

Fatemeh Rahbari

1. High-Level Overview of the Code

This project is designed for a simulation of a self-driving cab. The major goal is to demonstrate reinforcement technique (RL) techniques can be used in a simplified environment to develop an efficient and safe approach to solving real-world problems. This cab is called Smartcab. The Smartcab's job is to pick up the passengers at one location and drop them off at another. So, the main tasks that the Smartcab will try to come up with are:

- Drop off the passenger at the right location: Save the passenger's time by taking the minimum time possible to drop off
- Take care of passenger's safety and traffic rules

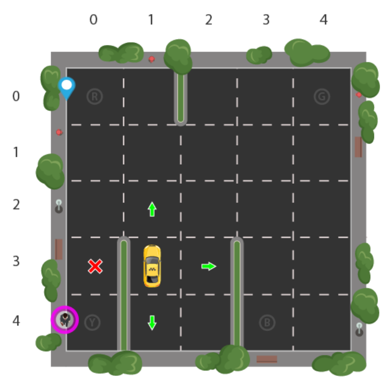
To model this problem using the RL model we need to determine rewards, states, and actions.

1. Rewards
Since the agent (the imaginary driver) is reward-motivated and is going to learn how to control the cab through trial experiences in the environment, we need to decide the rewards and/or penalties and their magnitude accordingly. So, we consider:

- The agent should receive a high positive reward for a successful drop-off because this behavior is highly desired
- The agent should be penalized if it tries to drop off a passenger in the wrong location.

A slight negative reward for not making it to the destination after every time step is considered since it is preferred for the agent to arrive late instead of making wrong moves trying to reach the destination as fast as possible.

2. State Space
The State Space is defined to be a set of all possible situations that the taxi could occupy. A grid area is considered a training area for the Smartcab where we are teaching it to transport people. It is assumed that Smartcab is the only vehicle in this parking lot. The parking lot is divided into a 5x5 grid, which gives us 25 possible taxi locations. These 25 locations are one part of the state space. There are four (4) locations where we can pick up and drop off a passenger: R, G, Y, B or [(0,0), (0,4), (4,0), (4,3)] in (row, col) coordinates. In the illustration below, passenger is in location Y and they wish to go to location R. So, the taxi environment has $5 \times 5 \times 5 \times 4 = 500$ total possible states.



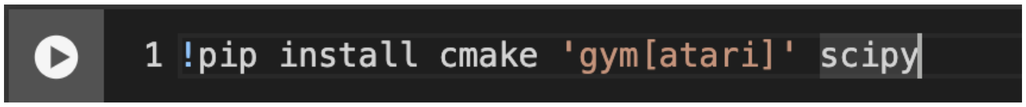
3. Action Space
The agent encounters one of the 500 states, and it takes action. The action in this case can be to move in a direction or decide to pick up/drop off a passenger. In other words, there are six possible actions: south, north, east, west, pickup, drop-off.

This is the action space: the set of all the actions that the agent can take in a given state. In the illustration above, the taxi cannot perform certain actions in certain states due to walls and -1 penalty is considered for every wall hit and the taxi won't move anywhere. This will just rack up penalties causing the taxi to consider going around the wall.

2. Identifying the Core Section and Implementation Explanation

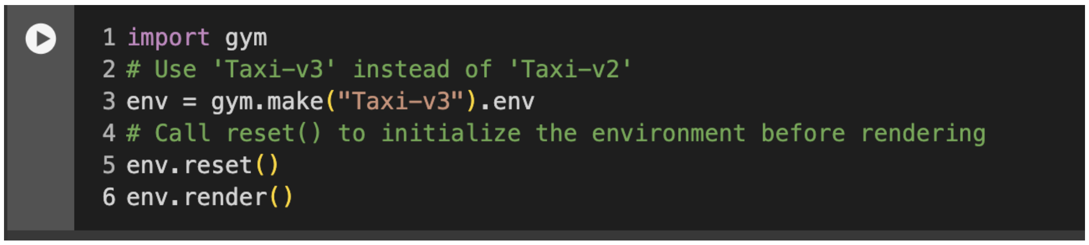
OpenAI Gym is used in this project and has this exact environment already built for us. OpenAI Gym which has been moved to Gymnasium, an API standard for reinforcement learning with a diverse collection of reference environments. Gym provides different game environments which we can plug into the code and test an agent. The library takes care of API for providing all the information that the agent would require, like possible actions, score, and current state. We just need to focus just on the algorithm part for the agent. The Gym environment called Taxi-V2, which all of the details explained above were pulled from. The objectives, rewards, and actions are all the same.

1. Gym's interface
We need to install gym first. Executing the following in a Jupyter notebook should work:

1. 

```
!pip install cmake 'gym[atari]' scipy
```

Once installed, we can load the game environment and render what it looks like:

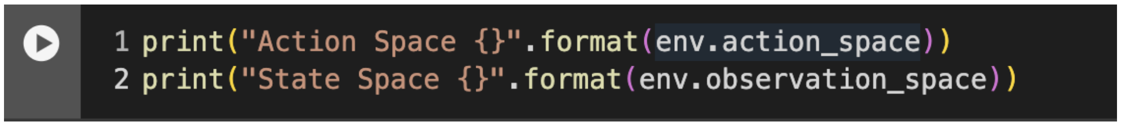
2. 

```python
import gym
# Use 'Taxi-v3' instead of 'Taxi-v2'
env = gym.make("Taxi-v3").env
# Call reset() to initialize the environment before rendering
env.reset()
env.render()
```

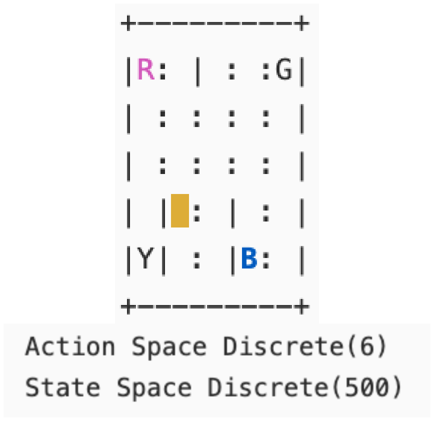
The core gym interface is env, which is the unified environment interface. The .env is used on the end of make to avoid training stopping at 200 iterations, which is the default for the new version of Gym (reference). The following are the env methods that would be quite helpful to us:

- **env.reset:** Resets the environment and returns a random initial state.
- **env.step(action):** Step the environment by one timestep. Returns
  - **observation**: Observations of the environment
  - **reward**: If your action was beneficial or not
  - **done**: Indicates if we have successfully picked up and dropped off a passenger, also called one *episode*
  - **info**: Additional info such as performance and latency for debugging purposes
- **env.render:** Renders one frame of the environment (helpful in visualizing the environment)

The restructured problem statement (from Gym docs) is as following: There are 4 locations (labeled by different letters), and the job is to pick up the passenger at one location and drop him off at another. We receive +20 points for a successful drop-off and lose 1 point for every time-step it takes. There is also a 10 point penalty for illegal pick-up and drop-off actions.

3. 

```python
print("Action Space {}".format(env.action_space))
print("State Space {}".format(env.observation_space))
```



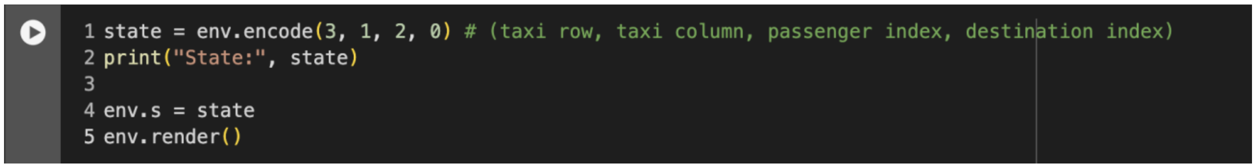
```
+---------+
|R: | : :G|
| : : : : |
| : : : : |
| |█: | : |
|Y| : |B: |
+---------+
Action Space Discrete(6)
State Space Discrete(500)
```

- The **filled square** represents the taxi, which is yellow without a passenger and green with a passenger.
- The **pipe ("|")** represents a wall which the taxi cannot cross.
- **R, G, Y, B** are the possible pickup and destination locations. The **blue letter** represents the current passenger pick-up location, and the **purple letter** is the current destination.

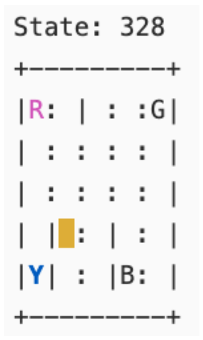
As verified by the prints, we have an **Action Space** of size 6 and a **State Space** of size 500. As you'll see, the RL algorithm won't need any more information than these two things. All we need is a way to identify a state uniquely by assigning a unique number to every possible state, and RL learns to choose an action number from 0-5 where:

- 0 = south
- 1 = north
- 2 = east
- 3 = west
- 4 = pickup
- 5 = dropoff

Recall that the 500 states correspond to a encoding of the taxi's location, the passenger's location, and the destination location. Reinforcement Learning will learn a mapping of **states** to the optimal **action** to perform in that state by *exploration*, i.e. the agent explores the environment and takes actions based off rewards defined in the environment. The optimal action for each state is the action that has the **highest cumulative long-term reward**. Illustration above, can be encoded state, and give it to the environment to render in Gym. Recall that we have the taxi at row 3, column 1, the passenger is at location 2, and the destination is location 0. Using the Taxi-v2 state encoding method, we can do the following:

4. 

```
state = env.encode(3, 1, 2, 0) # (taxi row, taxi column, passenger index, destination index)
print("State:", state)

env.s = state
env.render()
```

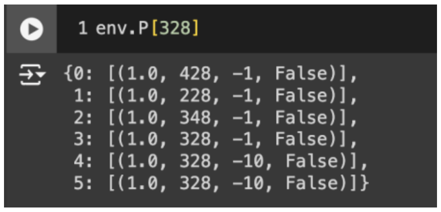


```
State: 328
+---------+
|R: | : :G|
| : : : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
```

2.The Reward Table

When the Taxi environment is created, there is an initial Reward table that's also created, called `P`. It like a matrix that has the number of states as rows and number of actions as columns, i.e. a states × actions matrix.

Since every state is in this matrix, we can see the default reward values assigned to the illustration's state:

5. 

```
env.P[328]
```

```
{0: [(1.0, 428, -1, False)],
 1: [(1.0, 228, -1, False)],
 2: [(1.0, 348, -1, False)],
 3: [(1.0, 328, -1, False)],
 4: [(1.0, 328, -10, False)],
 5: [(1.0, 328, -10, False)]}
```

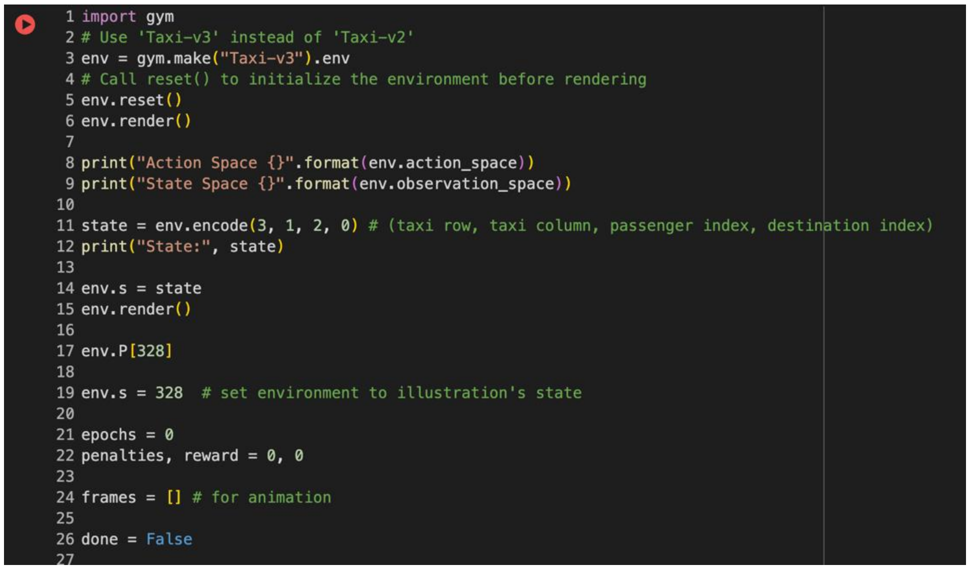
This dictionary has the structure {action: [(probability, nextstate, reward, done)]}.
A few things to note:

- The 0-5 corresponds to the actions (south, north, east, west, pickup, dropoff) the taxi can perform at the current state in the illustration.
- In this env, probability is always 1.0.
- The nextstate is the state we would be in if we take the action at this index of the dict
- All the movement actions have a -1 reward and the pickup/dropoff actions have -10 reward in this particular state. If we are in a state where the taxi has a passenger and is on top of the right destination, we would see a reward of 20 at the dropoff action (5)
- done is used to tell us when we have successfully dropped off a passenger in the right location. Each successfull dropoff is the end of an **episode**

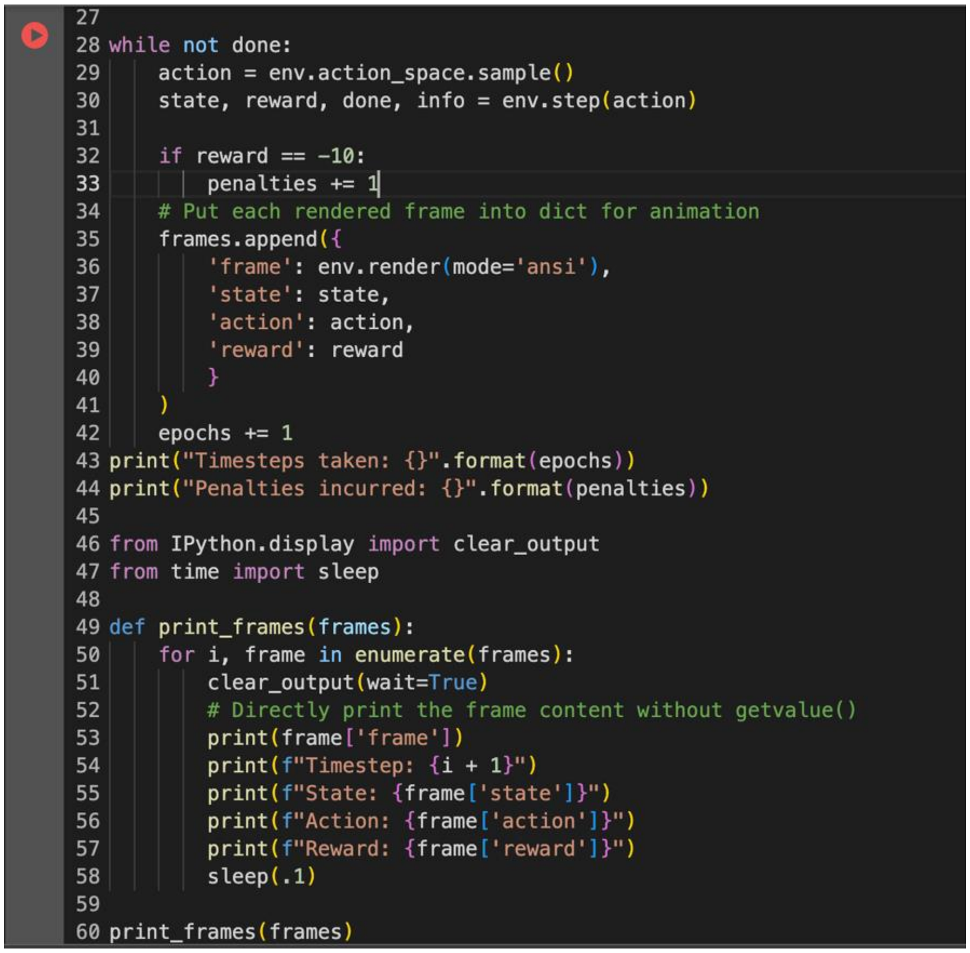
Note that if the agent chose to explore action two (2) in this state it would be going East into a wall. The source code has made it impossible to actually move the taxi across a wall, so if the taxi chooses that action, it will just keep accruing -1 penalties, which affects the **long-term reward**.

3. Solving the environment without Reinforcement Learning

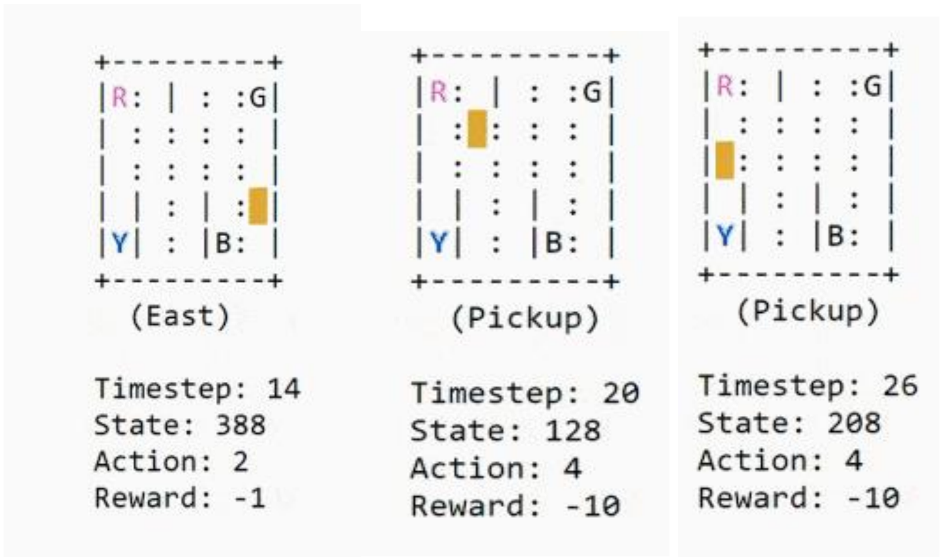
Let's see what would happen if we try to brute-force the way to solving the problem without RL. Since we have the P table for default rewards in each state, we can try to have the taxi navigate just using that.
We'll create an infinite loop which runs until one passenger reaches one destination (one **episode**), or in other words, when the received reward is 20. The env.action_space.sample() method automatically selects one random action from set of all possible actions.

6. 

```python
import gym
# Use 'Taxi-v3' instead of 'Taxi-v2'
env = gym.make("Taxi-v3").env
# Call reset() to initialize the environment before rendering
env.reset()
env.render()

print("Action Space {}".format(env.action_space))
print("State Space {}".format(env.observation_space))

state = env.encode(3, 1, 2, 0) # (taxi row, taxi column, passenger index, destination index)
print("State:", state)

env.s = state
env.render()

env.P[328]

env.s = 328  # set environment to illustration's state

epochs = 0
penalties, reward = 0, 0

frames = [] # for animation

done = False
```

7. 

```python

while not done:
    action = env.action_space.sample()
    state, reward, done, info = env.step(action)

    if reward == -10:
        penalties += 1
    # Put each rendered frame into dict for animation
    frames.append({
        'frame': env.render(mode='ansi'),
        'state': state,
        'action': action,
        'reward': reward
        }
    )
    epochs += 1
print("Timesteps taken: {}".format(epochs))
print("Penalties incurred: {}".format(penalties))

from IPython.display import clear_output
from time import sleep

def print_frames(frames):
    for i, frame in enumerate(frames):
        clear_output(wait=True)
        # Directly print the frame content without getvalue()
        print(frame['frame'])
        print(f"Timestep: {i + 1}")
        print(f"State: {frame['state']}")
        print(f"Action: {frame['action']}")
        print(f"Reward: {frame['reward']}")
        sleep(.1)

print_frames(frames)
```



```
+---------+
|R: | : :G|
| : : : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (East)

Timestep: 14
State: 388
Action: 2
Reward: -1
```

```
+---------+
|R: | : :G|
| : : : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (Pickup)

Timestep: 20
State: 128
Action: 4
Reward: -10
```

```
+---------+
|R: | : :G|
| : : : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (Pickup)

Timestep: 26
State: 208
Action: 4
Reward: -10
```

Not good. The agent takes thousands of timesteps and makes lots of wrong drop offs to deliver just one passenger to the right destination. This is because it is not *learning* from past experience. It can run this over and over, and it will never optimize. The agent has no memory of which action was best for each state, which is exactly what Reinforcement Learning will do for us.

4. Enter Reinforcement Learning

A simple RL algorithm called *Q-learning* which will give the agent some memory is used.

1. Intro to Q-learning
Essentially, Q-learning lets the agent use the environment's rewards to learn, over time, the best action to take in a given state.
In the Taxi environment, we have the reward table, P, that the agent will learn from. It does thing by looking receiving a reward for taking an action in the current state, then updating a *Q-value* to remember if that action was beneficial.
The values store in the Q-table are called a *Q-values*, and they map to a (state, action) combination.
A Q-value for a particular state-action combination is representative of the "quality" of an action taken from that state. Better Q-values imply better chances of getting greater rewards.
For example, if the taxi is faced with a state that includes a passenger at its current location, it is highly likely that the Q-value for pickup is higher when compared to other actions, like dropoff or north.
Q-values are initialized to an arbitrary value, and as the agent exposes itself to the environment and receives different rewards by executing different actions, the Q-values are updated using the equation:

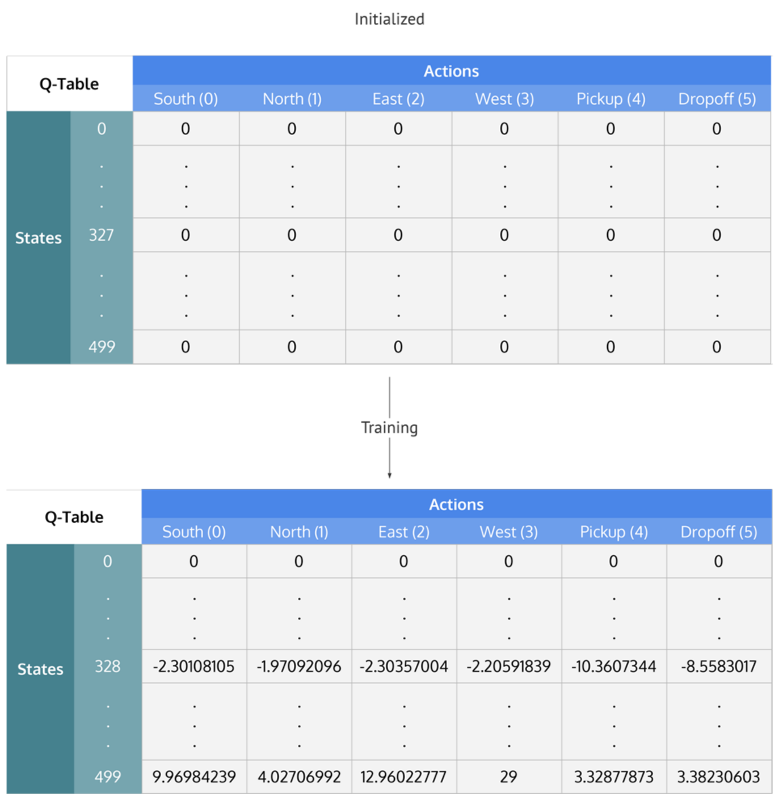
- $Q(\text{state},\text{action}) \leftarrow (1-\alpha)Q(\text{state},\text{action})+\alpha(\text{reward}+\gamma \text{maxa}Q(\text{next state},\text{all actions}))$

Where:

  - α (alpha) is the learning rate ($0<\alpha\leq1$) - Just like in supervised learning settings, α is the extent to which the Q-values are being updated in every iteration.
  - γ (gamma) is the discount factor ($0\leq\gamma\leq1$) - determines how much importance we want to give to future rewards. A high value for the discount factor (close to **1**) captures the long-term effective award, whereas, a discount factor of **0** makes the agent consider only immediate reward, hence making it greedy.

2. What is this saying?
We are assigning ($\leftarrow$), or updating, the Q-value of the agent's current *state* and *action* by first taking a weight ($1-\alpha$) of the old Q-value, then adding the learned value. The learned value is a combination of the reward for taking the current action in the current state, and the discounted maximum reward from the next state we will be in once we take the current action.
Basically, we are learning the proper action to take in the current state by looking at the reward for the current state/action combo, and the max rewards for the next state. This will eventually cause the taxi to consider the route with the best rewards strung together.
The Q-value of a state-action pair is the sum of the instant reward and the discounted future reward (of the resulting state). The way we store the Q-values for each state and action is through a **Q-table**

3. Q-Table
The Q-table is a matrix where we have a row for every state (500) and a column for every action (6). It's first initialized to 0, and then values are updated after training. Note that the Q-table has the same dimensions as the reward table, but it has a completely different purpose.



Initialized

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	327	0	0	0	0	0	0
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	499	0	0	0	0	0	0

Training

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	328	-2.30108105	-1.97092096	-2.30357004	-2.20591839	-10.3607344	-8.5583017
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	499	9.96984239	4.02706992	12.96022777	29	3.32877873	3.38230603

5. Summing up the Q-Learning Process

Breaking it down into steps, we get:

- Initialize the Q-table by all zeros.
- Start exploring actions: For each state, select any one among all possible actions for the current state (S).
- Travel to the next state (S') as a result of that action (a).
- For all possible actions from the state (S') select the one with the highest Q-value.
- Update Q-table values using the equation.
- Set the next state as the current state.
- If goal state is reached, then end and repeat the process.

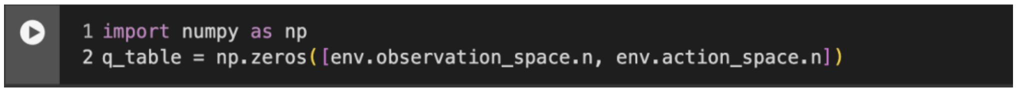
5. Exploiting learned values

After enough random exploration of actions, the Q-values tend to converge serving the agent as an action-value function which it can exploit to pick the most optimal action from a given state.
There's a tradeoff between exploration (choosing a random action) and exploitation (choosing actions based on already learned Q-values). We want to prevent the action from always taking the same route, and possibly overfitting, so we'll be introducing another parameter called ϵ "epsilon" to cater to this during training.
Instead of just selecting the best learned Q-value action, we'll sometimes favor exploring the action space further. Lower epsilon value results in episodes with more penalties (on average) which is obvious because we are exploring and making random decisions.

6. Implementing Q-learning in python

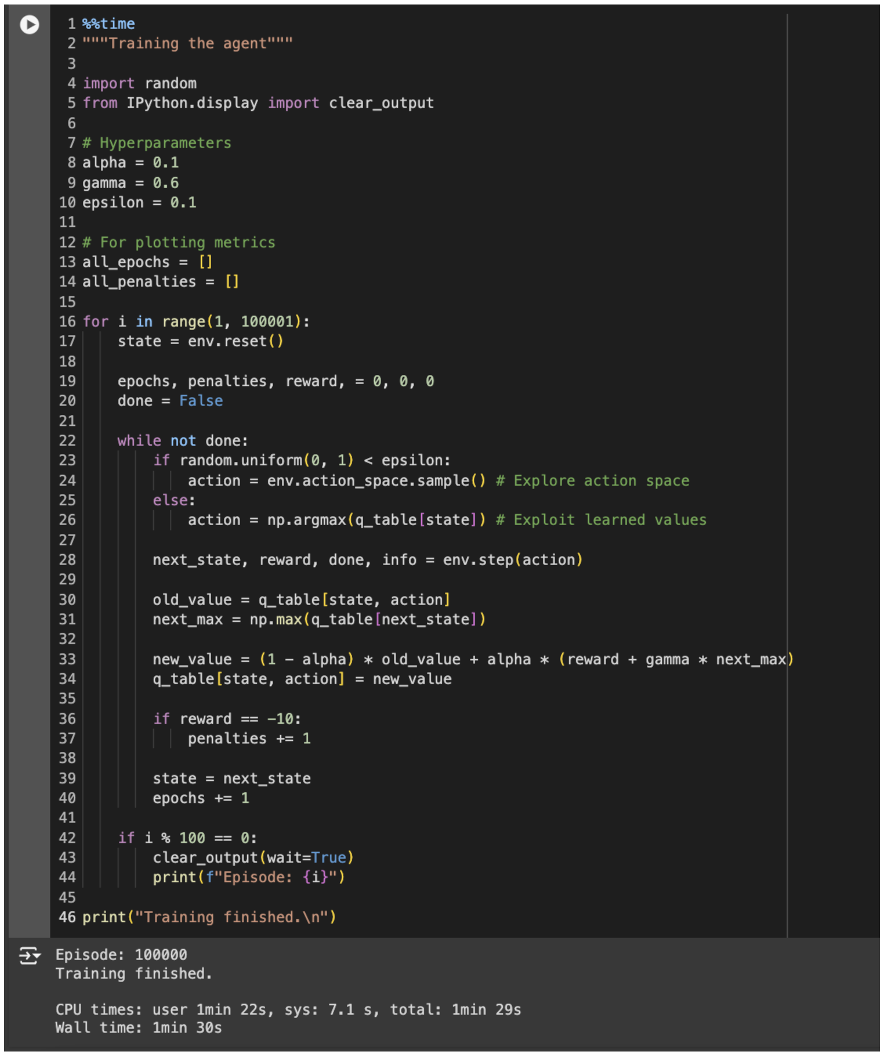
1. Training the Agent

First, we'll initialize the Q-table to a 500×6 matrix of zeros:

7.


```python
import numpy as np
q_table = np.zeros([env.observation_space.n, env.action_space.n])
```

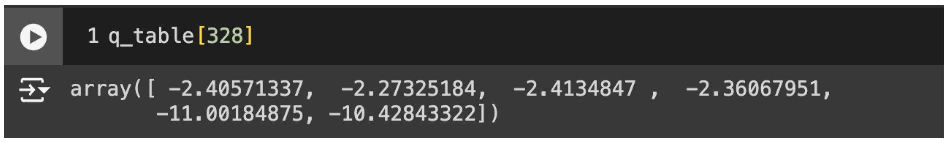
We can now create the training algorithm that will update this Q-table as the agent explores the environment over thousands of episodes.
In the first part of while not done, we decide whether to pick a random action or to exploit the already computed Q-values. This is done simply by using the epsilon value and comparing it to the random.uniform(0, 1) function, which returns an arbitrary number between 0 and 1.
We execute the chosen action in the environment to obtain the next_state and the reward from performing the action. After that, we calculate the maximum Q-value for the actions corresponding to the next_state, and with that, we can easily update the Q-value to the new_q_value:

8.


```python
%%time
"""Training the agent"""

import random
from IPython.display import clear_output

# Hyperparameters
alpha = 0.1
gamma = 0.6
epsilon = 0.1

# For plotting metrics
all_epochs = []
all_penalties = []

for i in range(1, 100001):
    state = env.reset()

    epochs, penalties, reward, = 0, 0, 0
    done = False

    while not done:
        if random.uniform(0, 1) < epsilon:
            action = env.action_space.sample() # Explore action space
        else:
            action = np.argmax(q_table[state]) # Exploit learned values

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

        old_value = q_table[state, action]
        next_max = np.max(q_table[next_state])

        new_value = (1 - alpha) * old_value + alpha * (reward + gamma * next_max)
        q_table[state, action] = new_value

        if reward == -10:
            penalties += 1

        state = next_state
        epochs += 1

    if i % 100 == 0:
        clear_output(wait=True)
        print(f"Episode: {i}")

print("Training finished.\n")
```

```
Episode: 100000
Training finished.

CPU times: user 1min 22s, sys: 7.1 s, total: 1min 29s
Wall time: 1min 30s
```

Now that the Q-table has been established over 100,000 episodes, let's see what the Q-values are at the illustration's state:

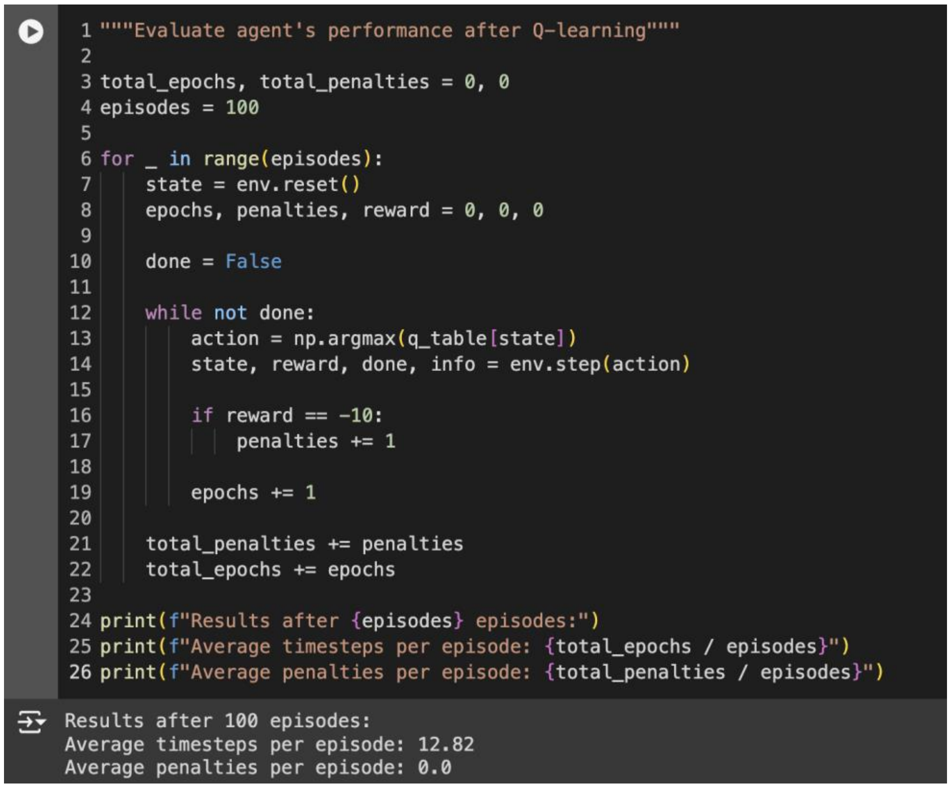
9. 

```
q_table[328]
```

```
array([ -2.40571337,  -2.27325184,  -2.4134847 ,  -2.36067951,
       -11.00184875, -10.42843322])
```

The max Q-value is "north" (-1.971), so it looks like Q-learning has effectively learned the best action to take in the illustration's state!

2. Evaluating the agent
We don't need to explore actions any further, so now the next action is always selected using the best Q-value:

10. 

```
"""Evaluate agent's performance after Q-learning"""

total_epochs, total_penalties = 0, 0
episodes = 100

for _ in range(episodes):
    state = env.reset()
    epochs, penalties, reward = 0, 0, 0

    done = False

    while not done:
        action = np.argmax(q_table[state])
        state, reward, done, info = env.step(action)

        if reward == -10:
            penalties += 1

        epochs += 1

    total_penalties += penalties
    total_epochs += epochs

print(f"Results after {episodes} episodes:")
print(f"Average timesteps per episode: {total_epochs / episodes}")
print(f"Average penalties per episode: {total_penalties / episodes}")
```

```
Results after 100 episodes:
Average timesteps per episode: 12.82
Average penalties per episode: 0.0
```

We can see from the evaluation, the agent's performance improved significantly, and it incurred no penalties, which means it performed the correct pickup/dropoff actions with 100 different passengers.

11. Comparing the Q-learning agent to no Reinforcement Learning

With Q-learning agent commits errors initially during exploration but once it has explored enough (seen most of the states), it can act wisely maximizing the rewards making smart moves. Let's see

how much better the Q-learning solution is when compared to the agent making just random moves.
We evaluate the agents according to the following metrics,

- **Average number of penalties per episode:** The smaller the number, the better the performance of the agent. Ideally, we would like this metric to be zero or very close to zero.
- **Average number of timesteps per trip:** We want a small number of timesteps per episode as well since we want the agent to take minimum steps(i.e. the shortest path) to reach the destination.
- **Average rewards per move:** The larger the reward means the agent is doing the right thing. That's why deciding rewards is a crucial part of Reinforcement Learning. In the case, as both timesteps and penalties are negatively rewarded, a higher average reward would mean that the agent reaches the destination as fast as possible with the least penalties"

Measure	**Random agent's performance**	**Q-learning agent's performance**
Average rewards per move	-3.9012092102214075	0.6962843295638126
Average number of penalties per episode	920.45	0.0
Average number of timesteps per trip	2848.14	12.38

These metrics were computed over 100 episodes. And as the results show, the Q-learning agent nailed it!