**NNFS LAB 1: Intelligent Agents**

**Reinforcement Learning - taxi problem**

Group 1:

Parth Deshpande 191060022

Ishan Deshpande 191060021

Dev Sharma 191060023

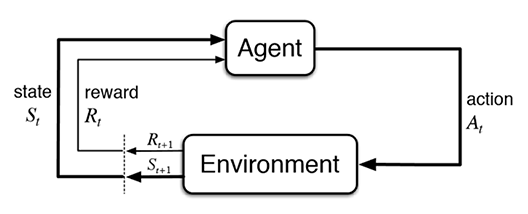
Kedar Daithankar 191060019

**Reinforcement Learning*:***

Reinforcement learning (RL) is an area of machine learning concerned with how intelligent agents ought to take actions in an environment in order to maximize the notion of cumulative reward. It involves an agent interacting with its surrounding environment to determine what is the best action to take. The environment could be uncertain, complex, and the agent’s behaviour can also be probabilistic, not deterministic.

Reinforcement learning is one of three basic machine learning paradigms, alongside supervised learning and unsupervised learning.

**Taxi** is one of many environments available on OpenAI Gym. These environments are used to develop and benchmark reinforcement learning algorithms.



In a RL problem, the agent would want to maximize its total rewards. It constantly interacts with the environment and explore to gather information about what reward it gets from executing an action as specific states, then search for the optimal action to take at each state.

**Q Learning** is a type of Value-based learning algorithms. The agent’s objective is to optimize a “Value function” suited to the problem it faces. In a Q learning process we have a **Q-table** that stores the Q value for each state and each possible action, the agent explores the environment and make update to the Q values iteratively.

Learning Process:

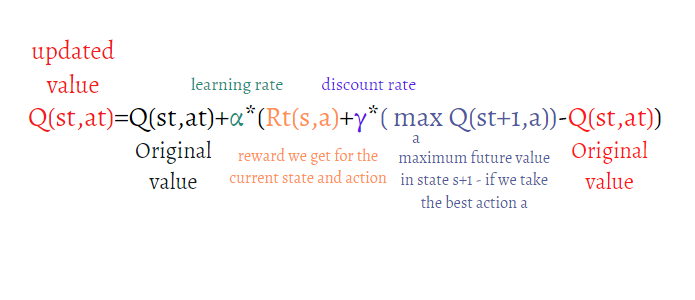
**Step 1**: Initialize all Q values in the Q- table to 0, the agent has no knowledge about the environment it is in.

**Step 2**: Explore the space: The agent keeps exploring the environment by executing actions at the states it is in.

***Exploration vs Exploitation:***

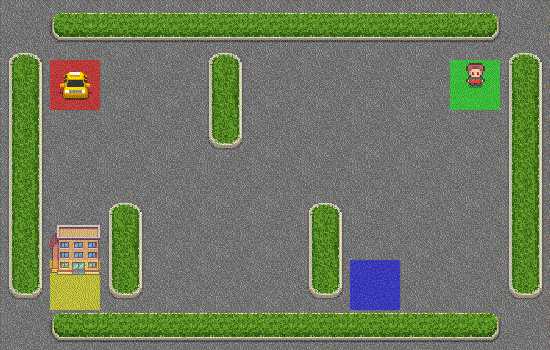
We can let our agent explore to update our Q-table using the Q-learning algorithm. As our agent learns more about the environment, we can let it use this knowledge to take more optimal actions and converge faster - known as exploitation. During exploitation, our agent will look at its Q-table and select the action with the highest Q-value (instead of a random action). Over time, our agent will need to explore less, and start exploiting what it knows instead.

**Step 3:** observe the reward - When exploring, the agent would observe what reward it gets from executing a particular action (at) in state (st) to go to next state (st+1).

**Step 4**: After observing the reward, the agent then updates the value function for the particular state and action pair using the following formula, this returns an updated Q-table.

TAXI PROBLEM:

There are four designated locations in the grid world indicated by **Red**, **Green**, **Yellow**, and **Blue**. When the episode starts, the taxi starts off at a random square and the passenger is at a random location. The taxi drives to the passenger’s location, picks up the passenger, drives to the passenger’s destination (another one of the four specified locations), and then drops off the passenger. Once the passenger is dropped off, the episode ends.



**Actions**

There are **6 discrete deterministic actions**:

* 0: move south
* 1: move north
* 2: move east
* 3: move west
* 4: pickup passenger
* 5: drop off passenger

There are **500 discrete states** since there are **25 taxi positions**, 5 possible locations of the passenger

**Passenger locations**:

* 0: Red
* 1: Green
* 2: Yellow
* 3: Blue
* 4: in taxi

**Destinations:**

* 0: Red
* 1: Green
* 2: Yellow
* 3: Blue

**Rewards**

-1 per step unless other reward is triggered.

+20 delivering passenger.

-10 executing “pickup” and “drop-off” actions illegally.

**Q-tables**:

For storing the points/Maximum expected future rewards we use q-tables. Q-table consists of number of **rows** same as the number of states of the agent in that environment & number of **columns** are the total number of actions it can take.

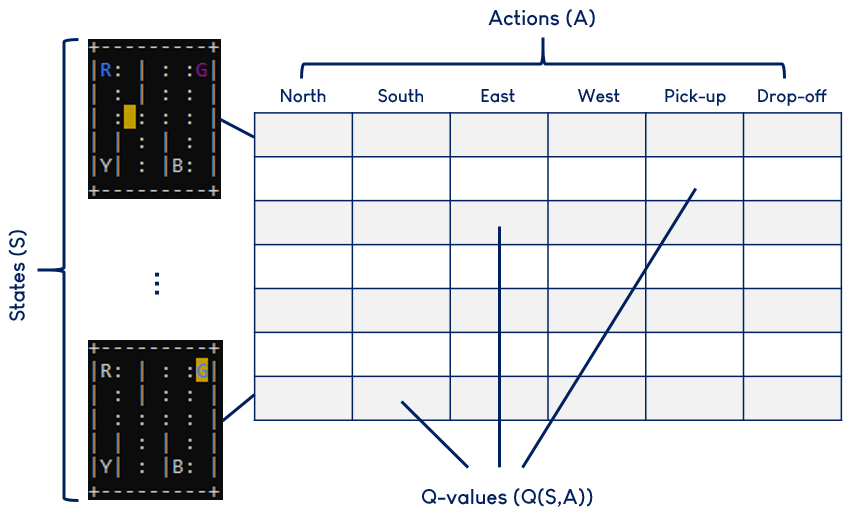
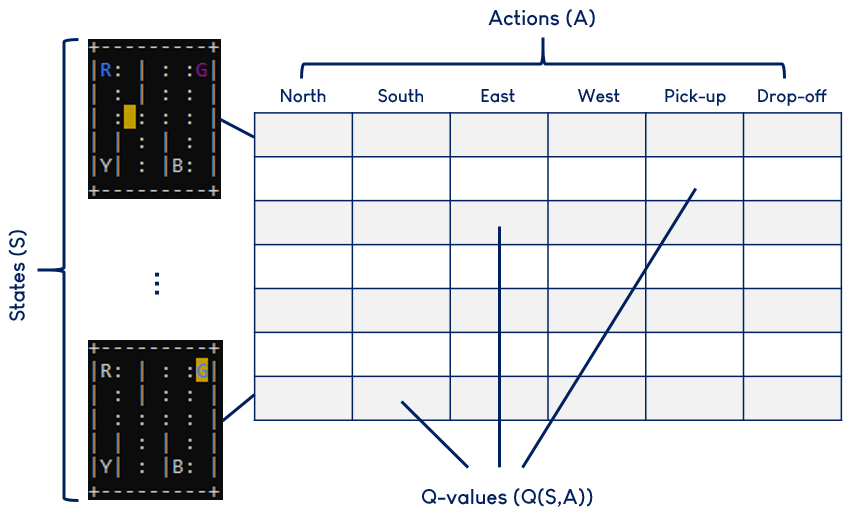
In this example, Agent can have 500 states and it can perform 6 actions as explained above.

So, the dimensions of q-table are **(500 x 6)**

Each row corresponds to a unique state in the 'Taxi' environment

Each column corresponds to an action our agent can take

Each cell corresponds to the Q-value for that state-action pair - a higher Q-value means a higher maximum reward our agent can expect to get if it takes that action in that state.



The Q-learning algorithm will help our agent update the current Q-value (Q(St,At)) with its observations after taking an action. I.e. increase Q if it encountered a positive reward, or decrease Q if it encountered a negative one.

**Why discount factor**?

The discount factor (a number between 0-1) is a clever way to scale down the rewards more and more after each step so that, the total sum remains bounded.

**Implementation:**

https://github.com/heisenberg-88/nnfs\_lab1\_Reinforcement-Learning---taxi-problem

There were some changes in the recent versions Taxi-V3 api, so we had to change the code accordingly.

In the new version, to make the environment work, we have added render\_mode as an additional argument according to the documentation.

streets = gym.make("Taxi-v3",render\_mode="human").env

also,

we have to call env.reset() before any env.render()

In every episode, we have to call env.reset().In new taxi-v3,

this function returns **current state number** & other info, probability, etc

state, info = streets.reset(

the state value ranges from 1 to 500

we can access q-table data using this**. q\_table[state].**

This returns a numpy array consisting of 6 columns corresponding to 6 actions.

The process for training our agent will look like:

* Initialising our Q-table with 0's for all Q-values.
* Let our agent play Taxi over a large number of games.
* Continuously update the Q-table using the Q-learning algorithm and an exploration-exploitation strategy.

Here's the full implementation:

Printway function is used for printing the steps.

def printway(index):  
 value = ""  
 if index==0:  
 value = "down"  
 elif index==1:  
 value ="up"  
 elif index==2:  
 value ="right"  
 elif index==3:  
 value ="left"  
 elif index==4:  
 value ="pickup passenger"  
 elif index==5:  
 value ="drop off passenger"  
 return value



Showing the state number and the corresponding action array.

We have created two functions for 2 different scenarios:

1. q\_learning\_new
2. q\_learning\_precalc

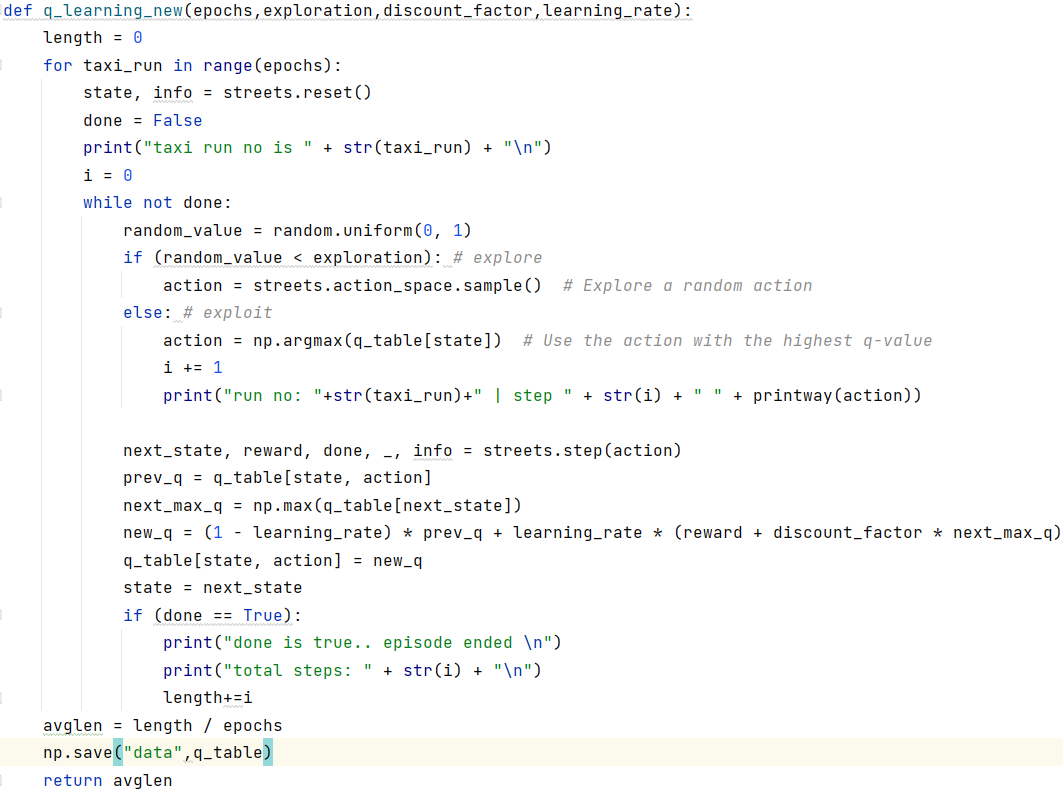
q\_learning\_new is used to compute the q\_table when the agent does not have any prior exploration.

q\_learning\_precalc is used when we have q\_table updated for some number of epochs. This function can be used when we want to start the q learning from a existing q\_table.

**q\_learning\_new.py**

here q\_table with zeros is created for initialization.

Streets taxiV3 environment is created from gym.make().



This function returns the average length avglen of the steps taken during each episode for all epochs. Adds number of steps ‘i’ to the lengths and then divides by epochs. length+=i ,avglen = length/epochs

This shows the current epoch and the steps/actions that agent is taking.

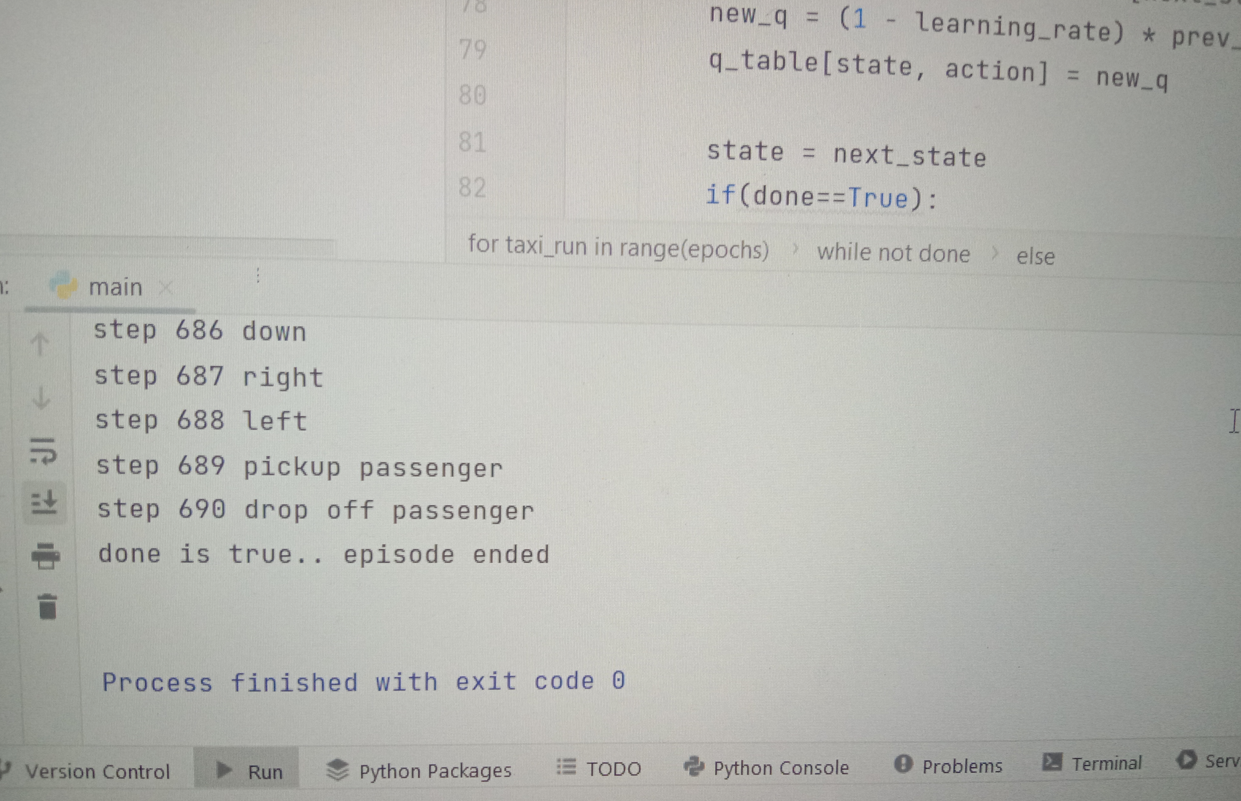
print("run no: "+str(taxi\_run)+" | step " + str(i) + " " +printway(action))

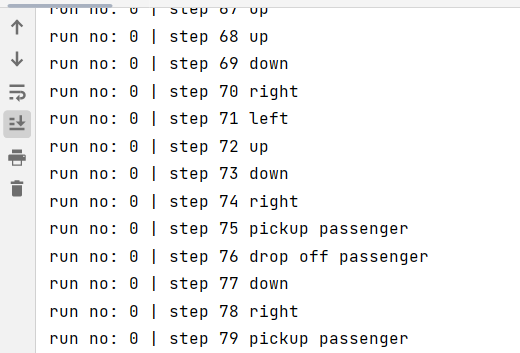
in old taxiv3, step() returns 4 values with the done boolean. But, in new taxiv3 api, env.step() returns 5 values: Observation – next\_state,

Reward, terminated – done bool replaced, truncated, info.

To get proper return values, changes are made accordingly.

next\_state, reward, done, \_, info = streets.step(action)

Steps showing till each episode ends



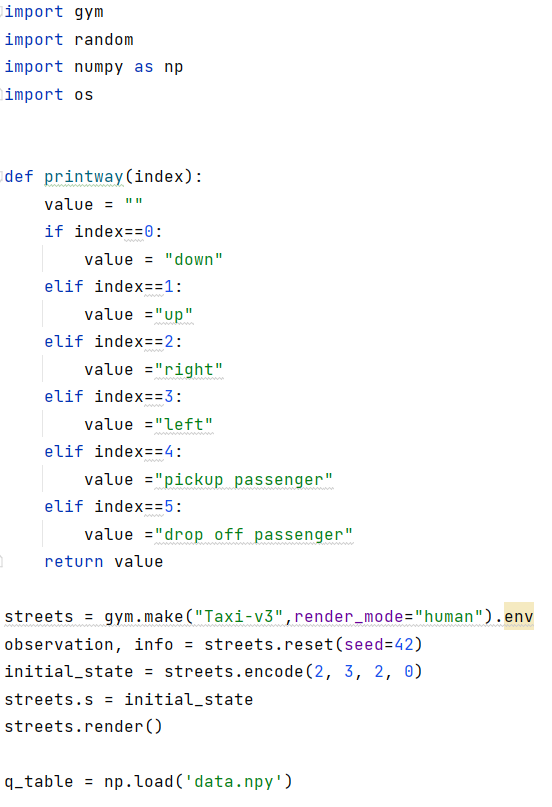
After all epochs/taxi runs, the q\_table is stored in data.npy format.

np.save("data",q\_table)



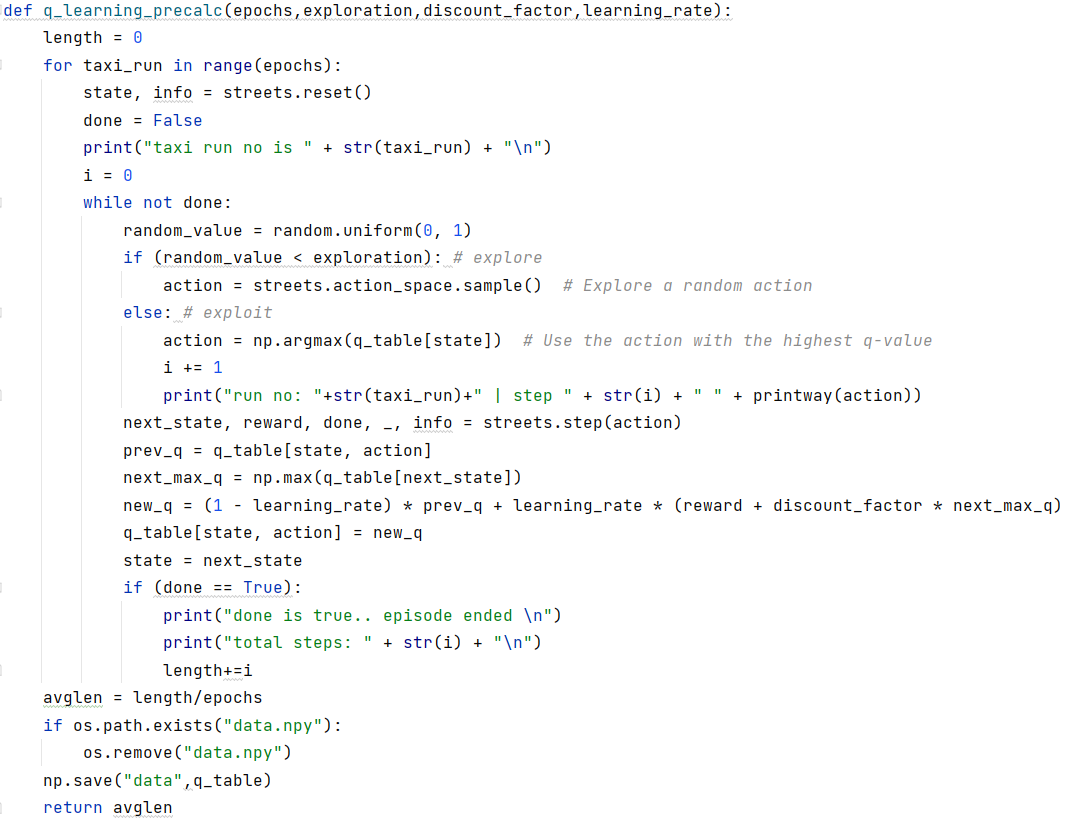
**q\_learning\_precalc.py**

This is same as previous one but the change is that , we can use this if we want to use previously generated q\_table and continue learning from that instance.



Here we are loading the precomputed q\_table using

np.load('data.npy')



avglen is returned by calculating length/epochs. Length is incremented at every step.

At the end of an episode total steps are added to length

length+=i

and then average is calculated  
avglen = length/epochs

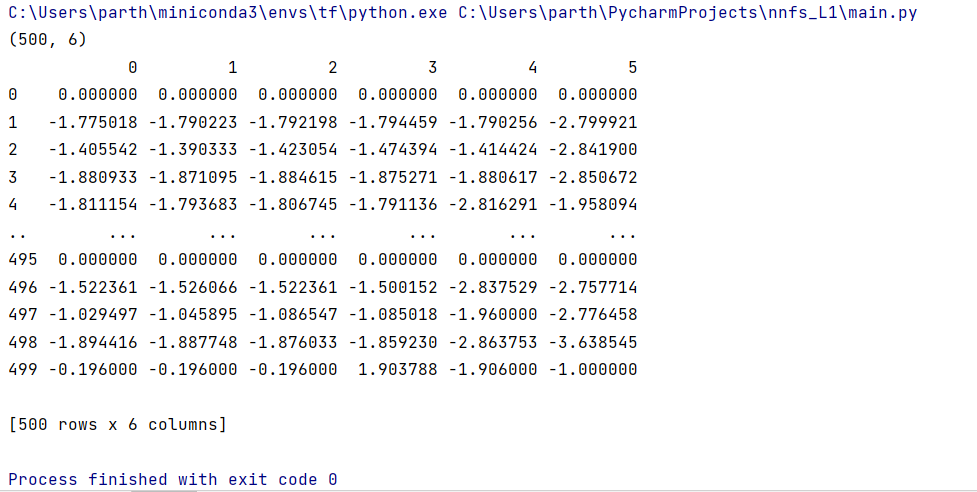
here, the old q\_table npy file is deleted and new npy data of q\_table is created.

if os.path.exists("data.npy"):  
 os.remove("data.npy")  
np.save("data",q\_table)

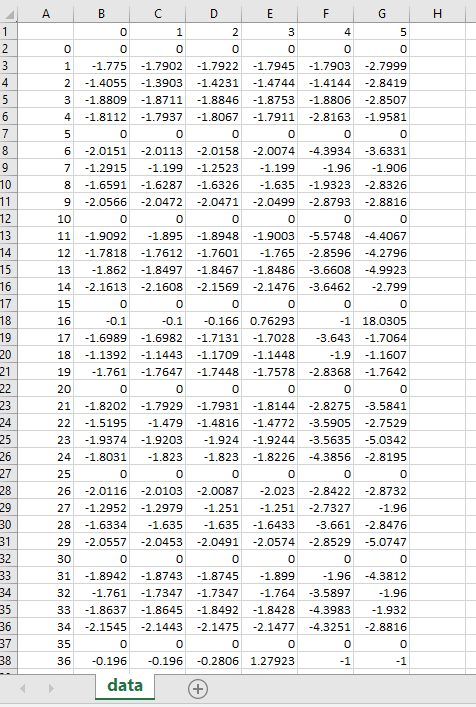
**viewdata.py**

****

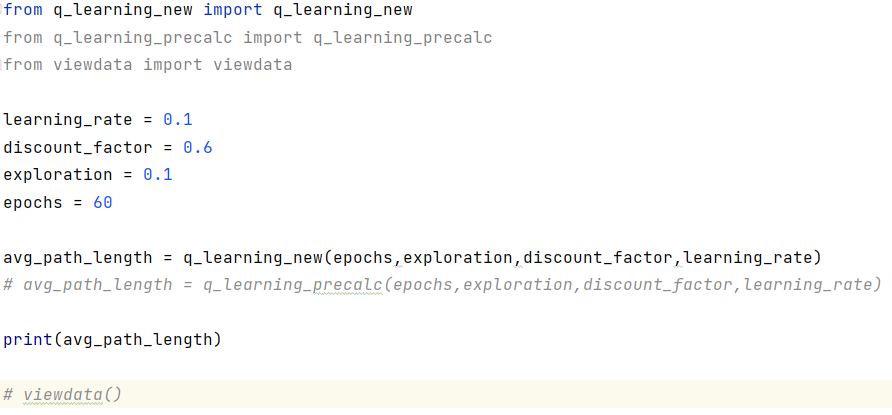
Viewdata() function is used to create and show the q\_table dataframe. Q\_table can also be exported to csv format using this function for further use.

**q\_table**

**csv q\_table data:**

****

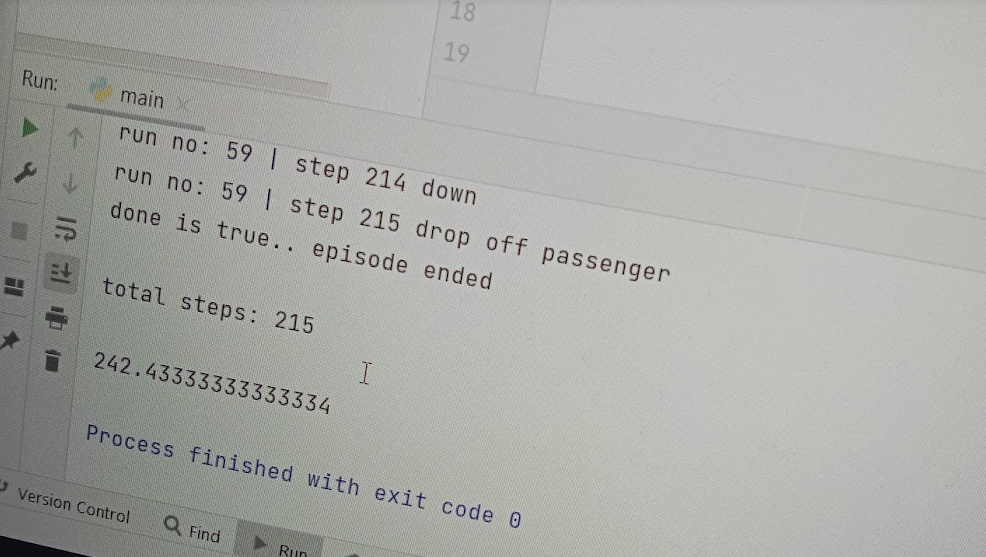
**……… till 500 rows (500 states)**

**main.py**

here, we’ve imported the 2 functions from 2 files mentioned above (i.e. q\_learning\_new.py & q\_learning\_precalc.py)

and viewdata.py file is also imported to interact with q\_table.

Avg length of steps returned from q\_learning functions is stored in avg\_path\_length.

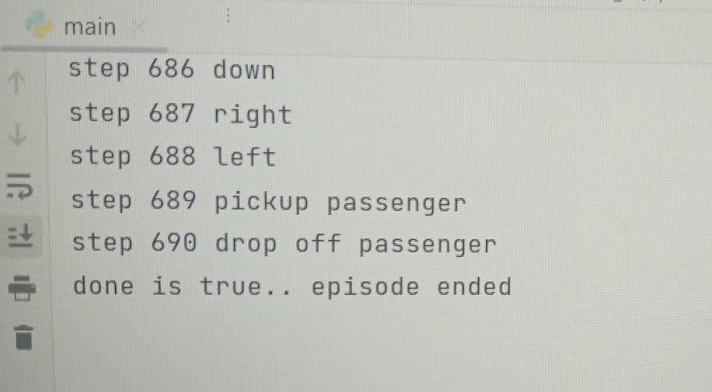
here avg length is 242 steps.

**Observations:**

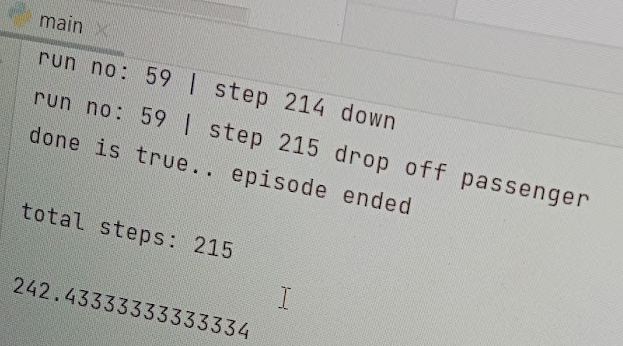
When the learning is done for lesser number of steps, the taxi requires more steps for getting to the right spot.

When more learning is done for 1000s of epochs , then we get better results and less steps.

We can also tweak discount rate for getting better results.

**1st epoch**:

**60th epoch: average steps length = 242.43**

****