Teach a Taxi to pick up and drop off passengers at the right locations with Reinforcement Learning

In [ ]:

In [3]:
```
import gym
import numpy as np
import pickle, os
```

In [4]:
```
env = gym.make("Taxi-v3")
```

In [5]:
```
state = env.reset()
```

In [6]:
```
state
```

Out[6]: 33

In [7]:
```
env.observation_space.n
```

Out[7]: 500

In [8]:
```
env.render()
```

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

In [9]:
```
state
```

Out[9]: 33

In [7]:
```
env.render()
```

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

Possible Actions

down (0), up (1), right (2), left (3), pick-up (4), and drop-off (5)

In [10]:
```
n_states = env.observation_space.n
n_actions = env.action_space.n
```

In [11]:
```
n_actions
```

Out[11]: 6

In [12]:
```
n_states
```

Out[12]: 500

In [13]:
```
env.env.s = 120
```

In [14]:
```
env.render()
```

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

In [15]:
```
env.step(5)
```

Out[15]: (120, -10, False, {'prob': 1.0})

How good does behaving completely random do?

In [16]:
```
state = env.reset()
counter = 0
g = 0
reward = None
```

In [17]:
```
while reward != 20:
    state, reward, done, info = env.step(env.action_space.sample())
    counter += 1
    g += reward
```

In [18]:
```
print("Solved in {} Steps with a total reward of {}".format(counter,g))
```

```
Solved in 1245 Steps with a total reward of -4806
```

Let's look at just one episode and see how the Q values change after each step using the formula below

In [19]:
```
Q = np.zeros([n_states, n_actions])
```

In [20]:
```
episodes = 1
G = 0
alpha = 0.618
```

In [21]:
```
for episode in range(1,episodes+1):
    done = False
    G, reward = 0,0
    state = env.reset()
    firstState = state
    print("Initial State = {}".format(state))
    while reward != 20:
        action = np.argmax(Q[state])
        state2, reward, done, info = env.step(action)
        Q[state,action] += alpha * (reward + np.max(Q[state2]) - Q[state,action])
        G += reward
        state = state2
```

```
Initial State = 82
```

In [22]:
```
finalState = state
```

Let's look at the first step:

In [23]:
```
firstState
```

Out[23]: 82

Let's look at the final step:

In [24]:
```
finalState
```

Out[24]: 410

In [25]:
```
Q
```

Out[25]:
```
array([[ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         0.        ],
       [-1.236     , -1.236     , -1.617924, -1.236     , -1.236     ,
        -6.18      ],
       ...,
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         0.        ],
       [-0.618     , -0.618     ,  0.        ,  0.        ,  0.        ,
         0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         0.        ]])
```

Let's run over multiple episodes so that we can converge on a optimal policy

In [26]:
```
episodes = 2000
rewardTracker = []
```

In [27]:
```
G = 0
alpha = 0.618
```

In [28]:
```
for episode in range(1,episodes+1):
    done = False
    G, reward = 0,0
    state = env.reset()
    while done != True:
        action = np.argmax(Q[state])
        state2, reward, done, info = env.step(action)
        Q[state,action] += alpha * ((reward + (np.max(Q[state2])) - Q[state,action]))
        G += reward
        state = state2

    if episode % 100 == 0:
        print('Episode {} Total Reward: {}'.format(episode,G))
```

```
Episode 100 Total Reward: -41
Episode 200 Total Reward: 3
Episode 300 Total Reward: 10
Episode 400 Total Reward: 3
Episode 500 Total Reward: 8
Episode 600 Total Reward: 5
Episode 700 Total Reward: 5
Episode 800 Total Reward: 9
Episode 900 Total Reward: 7
Episode 1000 Total Reward: 12
Episode 1100 Total Reward: 5
Episode 1200 Total Reward: 14
Episode 1300 Total Reward: 6
Episode 1400 Total Reward: 6
Episode 1500 Total Reward: 10
Episode 1600 Total Reward: 9
Episode 1700 Total Reward: 6
Episode 1800 Total Reward: 6
Episode 1900 Total Reward: 6
Episode 2000 Total Reward: 11
```

Now that we have learned the optimal Q Values we have developed a optimal policy and have no need to train the agent anymore

In [29]:
```
state = env.reset()
done = None
```

In [30]:
```
while done != True:
    # We simply take the action with the highest Q Value
    action = np.argmax(Q[state])
    state, reward, done, info = env.step(action)
    env.render()
```

```
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (North)
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (North)
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (North)
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (North)
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (East)
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (Pickup)
+---------+
|R: | :[G:|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (West)
+---------+
|R: | : :G|
| : | :[: |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (South)
+---------+
|R: | : :G|
| : | : : |
| : :[: : |
| | : | : |
|Y| : |B: |
+---------+
  (South)
+---------+
|R: | : :G|
| : | : : |
| : :[: : |
| | : | : |
|Y| : |B: |
+---------+
  (West)
+---------+
|R: | : :G|
| : | : : |
| :[: : : |
| | : | : |
|Y| : |B: |
+---------+
  (West)
+---------+
|R: | : :G|
| : | : : |
|[: : : : |
| | : | : |
|Y| : |B: |
+---------+
  (North)
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (North)
+---------+
|[: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (West)
+---------+
|[: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (Dropoff)
```

In [ ]:

In [31]:
```
with open("smartTaxi_qTable.pkl", 'wb') as f:
    pickle.dump(Q, f)
```

In [32]:
```
with open("smartTaxi_qTable.pkl", 'rb') as f:
    Qtest = pickle.load(f)
```

In [ ]: