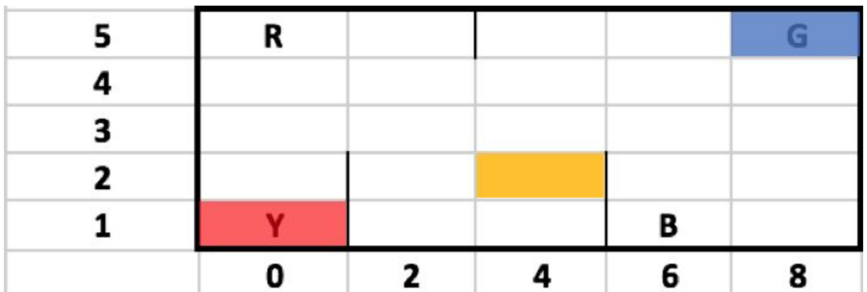
ELEN 6885 Reinforcement Learning Coding Assignment (Part 1, 2, 3)

Taxi Problem Overview

There are 4 locations (labeled by different letters) and your job is to pick up the passenger at one location and drop him off in another. You receive +20 points for a successful drop-off, and lose 1 point for every timestep it takes. There is also a 10 point penalty for illegal pick-up and drop-off actions.



- blue: passenger
- magenta: destination
- yellow: empty taxi
- green: full taxi
- other letters: locations

Please put your code into the block marked by:

##############################
YOUR CODE STARTS HERE
YOUR CODE ENDS HERE

##############################
You should not edit anything outside of the block.

Playing with the environment

Run the cell below to get a feel for the environment by moving your agent(the taxi) by taking one of the actions at each step.

In [14]:
```
from gym.wrappers import Monitor
import gym
import random
import numpy as np
import time
from helpers import utils
```

In [3]:

```python
"""
You can test your game now.
Input range from 0 to 5:
    0 : South (Down)
    1 : North (Up)
    2 : East (Right)
    3 : West (Left)
    4: Pick up
    5: Drop off
    6: exit_game
"""
GAME = "Taxi-v3"
env = gym.make(GAME)
env = Monitor(env, "taxi_simple", force=True)
s = env.reset()
steps = 100
for step in range(steps):
    env.render()
    action = int(input("Please type in the next action:"))
    if action==6:
        break
    s, r, done, info = env.step(action)
    print('state:',s)
    print('reward:',r)
    print('Is state terminal?:',done)
    print('info:',info)

# close environment and monitor
env.close()
```

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

Please type in the next action:6
```

1.1 Incremental implementation of average

We've finished the incremental implementation of average for you. Please call the function to estimate with 1/step step size and fixed step size to compare the difference between these two on a simulated Bandit problem.

In [4]:
```python
def estimate(OldEstimate, StepSize, Target):
    '''An incremental implementation of average.
    OldEstimate : float
    StepSize : float
    Target : float
    '''
    NewEstimate = OldEstimate + StepSize * (Target - OldEstimate)
    return NewEstimate
```

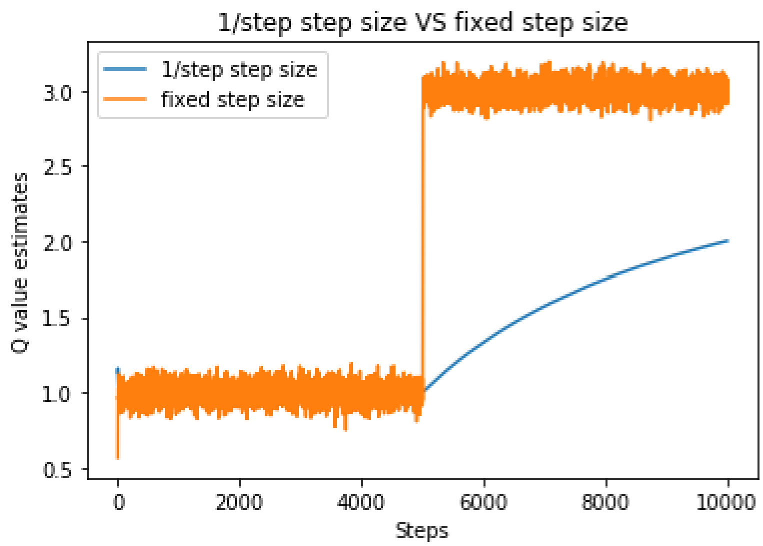
In [5]:
```python
random.seed(6885)
numTimeStep = 10000
q_h = np.zeros(numTimeStep + 1) # Q Value estimate with 1/step step si
q_f = np.zeros(numTimeStep + 1) # Q value estimate with fixed step siz
FixedStepSize = 0.5 #A large number to exaggerate the difference
for step in range(1, numTimeStep + 1):
    if step < numTimeStep / 2:
        r = random.gauss(mu = 1, sigma = 0.1)
    else:
        r = random.gauss(mu = 3, sigma = 0.1)

    #TIPS: Call function estimate defined in ./helpers/utils.py
    ############################
    # YOUR CODE STARTS HERE
    q_h[step] = estimate(q_h[step-1], 1/step, r)
    q_f[step] = estimate(q_f[step-1], FixedStepSize, r)

    # YOUR CODE ENDS HERE
    ############################

q_h = q_h[1:]
q_f = q_f[1:]
```

Plot the two Q value estimates. (Please include a title, labels on both axes, and legends)

In [5]:

```
import matplotlib.pyplot as plt
##########################
# YOUR CODE STARTS HERE
plt.title('1/step step size VS fixed step size')
plt.plot(q_h, label ='1/step step size')
plt.plot(q_f, label = 'fixed step size')
plt.xlabel('Steps')
plt.ylabel('Q value estimates')
plt.legend()
# YOUR CODE ENDS HERE
##########################
```

Out[5]: <matplotlib.legend.Legend at 0x1dca7ce9dd8>



1.2 ϵ-Greedy for Exploration

In Reinforcement Learning, we are always faced with the dilemma of exploration and exploitation. ϵ-Greedy is a trade-off between them. You are supposed to implement Greedy and ϵ-Greedy. We combine these two policies in one function by treating Greedy as ϵ-Greedy where $\epsilon = 0$. Edit the function epsilon_greedy the following block.

In [6]:
```
def epsilon_greedy(value, e, seed = None):
    '''
    Implement Epsilon-Greedy policy.

    Inputs:
    value: numpy ndarray
    A vector of values of actions to choose from
    e: float
    Epsilon
    seed: None or int
    Assign an integer value to remove the randomness

    Outputs:
    action: int
    Index of the chosen action
    '''
    assert len(value.shape) == 1
    assert 0 <= e <= 1

    if seed != None:
          np.random.seed(seed)

    ############################
    # YOUR CODE STARTS HERE
    p = np.random.uniform(low=0.0, high=1.0, size=None)
    #print(p)
    if p < e:
        action = random.randint(0,len(value)-1)
    else:
        action =  random.choice(np.where(value == value.max())[0])

    # YOUR CODE ENDS HERE
    ############################
    return action
```

In [7]:
```
np.random.seed(6885) #Set the seed forreproducability
q = np.random.normal(0, 1, size = 5)
############################
# YOUR CODE STARTS HERE
e_greedy_action = epsilon_greedy(q,.1, 6885)
greedy_action = epsilon_greedy(q,0, 6885)
# YOUR CODE ENDS HERE
############################
print('Values:')
print(q)
print('Greedy Choice =', greedy_action)
print('Epsilon-Greedy Choice =', e_greedy_action)
```

```
Values:
[ 0.61264537  0.27923079 -0.84600857  0.05469574 -1.09990968]
Greedy Choice = 0
Epsilon-Greedy Choice = 0
```

You should get the following results:

Values:

```
[ 0.61264537 0.27923079 -0.84600857 0.05469574 -1.09990968]
Greedy Choice = 0
Epsilon-Greedy Choice = 0
```

1.3 Exploration VS. Exploitation

Try to reproduce Figure 2.2 (the upper one is enough) of the Sutton's book based on the experiment described in Chapter 2.3.

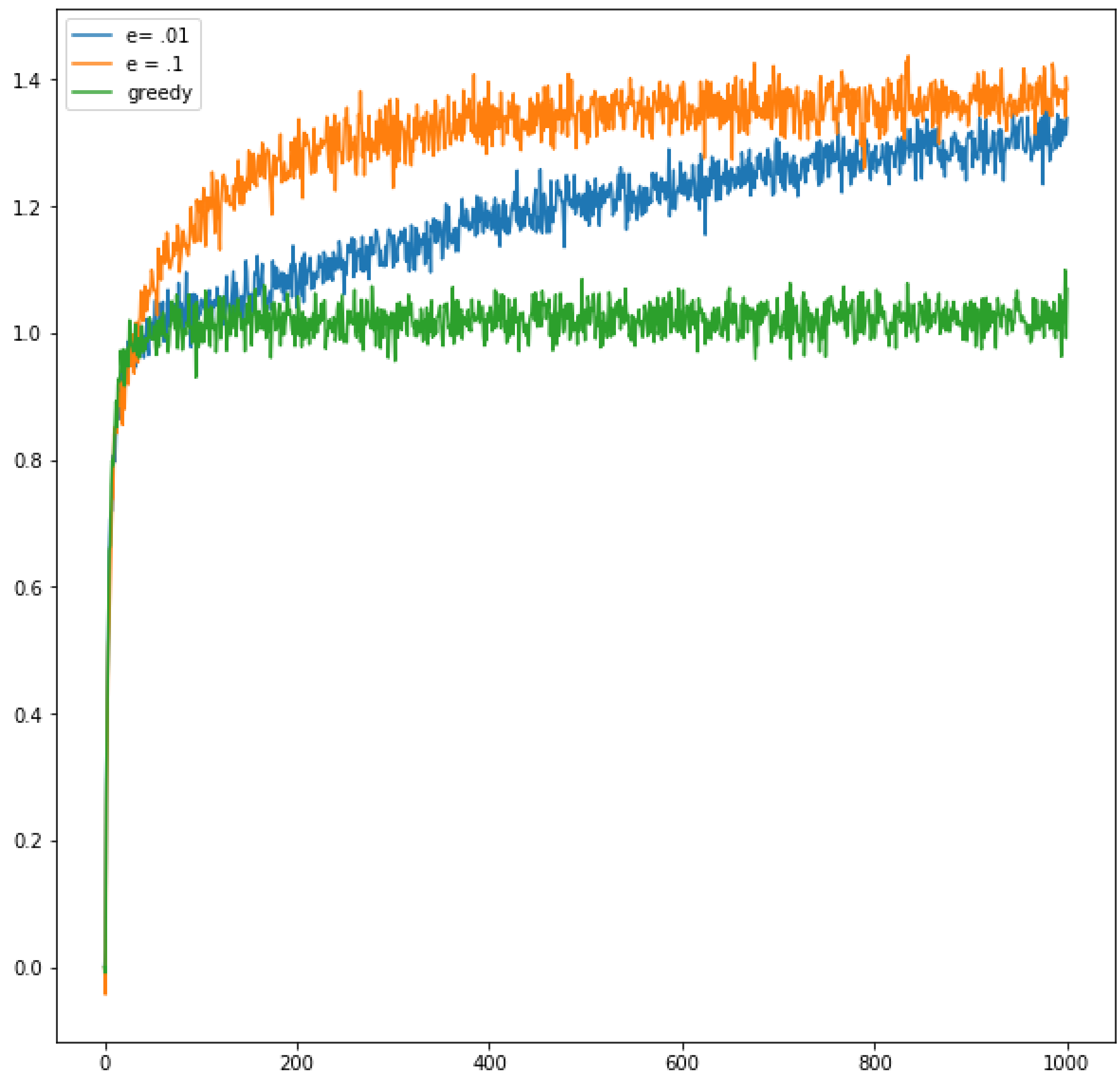
In [11]:
```
# Do the experiment and record average reward acquired in each time st
###########################
# YOUR CODE STARTS HERE
np.random.seed(100) #Set the seed forreproducability
trials = 2000
numTimeStep = 1000

e_final_results = np.zeros(numTimeStep + 1)
e2_final_results = np.zeros(numTimeStep + 1)
greedy_final_results = np.zeros(numTimeStep + 1)



for trial in range(1, trials+1):
    e_qestimate = np.zeros(10)
    e2_qestimate = np.zeros(10)
    g_qestimate = np.zeros(10)
    q_arms = np.random.normal(0, 1, size = 10)

    e_counter = {}
    e2_counter = {}
    greedy_counter = {}

    for step in range(1, numTimeStep +1):

        e_greedy_action = epsilon_greedy(e_qestimate,.01)
        e_greedy_action_2 = epsilon_greedy(e2_qestimate, .1)
        greedy_action = epsilon_greedy(g_qestimate, 0)

        reward_e = q_arms[e_greedy_action] + np.random.normal(0,1)
        reward_e2 = q_arms[e_greedy_action_2] + np.random.normal(0,1)
        reward_greedy = q_arms[greedy_action] + np.random.normal(0,1)

        if e_greedy_action in e_counter:
            e_counter[e_greedy_action] += 1
        else:
            e_counter[e_greedy_action] = 1

        if e_greedy_action_2 in e2_counter:
            e2_counter[e_greedy_action_2] += 1
        else:
            e2_counter[e_greedy_action_2] = 1

        if greedy_action in greedy_counter:
            greedy_counter[greedy_action] += 1
        else:
            greedy_counter[greedy_action] = 1


        e_qestimate[e_greedy_action] = estimate(e_qestimate[e_greedy_a
        e2_qestimate[e_greedy_action_2] = estimate(e2_qestimate[e_gree
        g_qestimate[greedy_action] = estimate(g_qestimate[greedy_actic

        e_final_results[step] = estimate(e_final_results[step], 1/tria
        e2_final_results[step] =  estimate(e2_final_results[step], 1/t
        greedy_final_results[step] =  estimate(greedy_final_results[st
```

```
# YOUR CODE ENDS HERE
##########################
```

In [12]:

```
# Plot the average reward
##########################
# YOUR CODE STARTS HERE
plt.figure(figsize=(10,10))
plt.plot(e_final_results, label = 'e= .01')
plt.plot(e2_final_results, label = 'e = .1')
plt.plot(greedy_final_results, label = 'greedy')
plt.legend()

# YOUR CODE ENDS HERE
##########################
```

Out[12]: <matplotlib.legend.Legend at 0x1cd1765d630>



Question 2

In this question, you will implement the value iteration and policy iteration algorithms to solve the Taxi game problem

2.1 Model-based RL: value iteration

For this part, you need to implement the helper functions action_evaluation(env, gamma, v), and extract_policy(env, v, gamma) in utils.py. Understand action_selection(q) which we have implemented.
Use these helper functions to implement the value_iteration algorithm below.

In [7]:
```
import numpy as np
from helpers import utils
np.random.seed(6885) #Set the seed forreproducability
def value_iteration(env, gamma, max_iteration, theta):
    """
    Implement value iteration algorithm. You should use extract_policy

    Parameters
    ----------
    env: OpenAI env.
            env.P: dictionary
                    the transition probabilities of the environment
                    P[state][action] is tuples with (probability, next
            env.nS: int
                    number of states
            env.nA: int
                    number of actions
    gamma: float
            Discount factor.
    max_iteration: int
            The maximum number of iterations to run before stopping.
    theta: float
            Determines when value function has converged.
    Returns:
    ----------
    value function: np.ndarray
    policy: np.ndarray
    """
    V = np.zeros(env.nS)
    ############################
    # YOUR CODE STARTS HERE
    for i in range(max_iteration):
        q = utils.action_evaluation(env,gamma,V)
        V_prev = V
        V = np.max(q,axis=1)
        if np.abs(V-V_prev).max() < theta:
            break
    policy = utils.extract_policy(env,V,gamma)
    # YOUR CODE ENDS HERE
    ############################

    return V, policy

```

After implementing the above function, read and understand the functions implemented in evaluation_utils.py, which we will use to evaluate our value iteration policy

In [9]:
```
from helpers import evaluation_utils
import gym
GAME = "Taxi-v3"
env = gym.make(GAME)
V_vi, policy_vi = value_iteration(env, gamma=0.95, max_iteration=6000,
# visualize how the agent performs with the policy generated from valu
evaluation_utils.render_episode(env, policy_vi)
```

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

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

In [10]:
```
# evaluate the performance of value iteration over 100 episodes
evaluation_utils.avg_performance(env, policy_vi)
```

Out[10]: 8.323232323232324

2.2 Model-based RL: policy iteration

In this part, you are supposed to implement policy iteration to solve the Taxi game problem.

In [8]:

```
from helpers import utils
def policy_iteration(env, gamma, max_iteration, theta):
    """Implement Policy iteration algorithm.

    You should use the policy_evaluation and policy_improvement metho
    implement this method.

    Parameters
    ----------
    env: OpenAI env.
            env.P: dictionary
                    the transition probabilities of the environment
                    P[state][action] is tuples with (probability, nex
            env.nS: int
                    number of states
            env.nA: int
                    number of actions
    gamma: float
            Discount factor.
    max_iteration: int
            The maximum number of iterations to run before stopping.
    theta: float
            Determines when value function has converged.
    Returns:
    ----------
    value function: np.ndarray
    policy: np.ndarray
    """
    policy = np.zeros(env.nS, dtype=int)
    ############################
    # YOUR CODE STARTS HERE
    for i in range(max_iteration):
        V = policy_evaluation(env, policy, gamma, theta)
        policy, policy_stable  = policy_improvement(env, V, policy, g
        if policy_stable:
            break
    # YOUR CODE ENDS HERE
    ############################

    return V, policy


def policy_evaluation(env, policy, gamma, theta):
    """Evaluate the value function from a given policy.

    Parameters
    ----------
    env: OpenAI env.
            env.P: dictionary
                    the transition probabilities of the environment
                    P[state][action] is tuples with (probability, nex
            env.nS: int
                    number of states
            env.nA: int
                    number of actions

```

```
    gamma: float
            Discount factor.
    policy: np.array
            The policy to evaluate. Maps states to actions.
    max_iteration: int
            The maximum number of iterations to run before stopping.
    theta: float
            Determines when value function has converged.
    Returns
    -------
    value function: np.ndarray
            The value function from the given policy.
    """
    P = env.P
    V = np.zeros(env.nS)
    ############################
    # YOUR CODE STARTS HERE
    while True:
        delta = 0
        for s in range(env.nS):
            a = policy[s]
            q_s_a = 0
            for i in range(len(P[s][a])):
                next_state_tuple = P[s][a][i]
                v_next_state = V[next_state_tuple[1]]
                p_next_state = next_state_tuple[0]
                reward_next_state = next_state_tuple[2]
                q_s_a+=  p_next_state*(reward_next_state + gamma*v_ne
            delta = max(delta, np.abs(q_s_a - V[s]))
            V[s] = q_s_a

        if delta < theta:
            break
    # YOUR CODE ENDS HERE
    ############################
    return V


def policy_improvement(env, value_from_policy, policy, gamma):
    """Given the value function from policy, improve the policy.

    Parameters
    ----------
    env: OpenAI env
            env.P: dictionary
                    the transition probabilities of the environment
                    P[state][action] is tuples with (probability, nex
            env.nS: int
                    number of states
            env.nA: int
                    number of actions

    value_from_policy: np.ndarray
            The value calculated from the policy
    policy: np.array
            The previous policy.
    gamma: float
```

```
                Discount factor.

        Returns
        -------
        new policy: np.ndarray
                An array of integers. Each integer is the optimal action
                in that state according to the environment dynamics and t
                given value function.
        stable policy: bool
                True if the optimal policy is found, otherwise false
        """
        ############################
        # YOUR CODE STARTS HERE
        nS = env.nS
        nA = env.nA
        q = np.zeros((nS, nA))
        P=env.P
        policy_stable = False
        v = value_from_policy
        for s in range(nS):
            for a in range(nA):
                ############################
                # YOUR CODE STARTS HERE
                #[probability, nextstate, reward, terminal]=P[s][a]
                q_s_a = 0
                for i in range(len(P[s][a])):
                    next_state_tuple = P[s][a][i]
                    v_next_state = v[next_state_tuple[1]]
                    p_next_state = next_state_tuple[0]
                    reward_next_state = next_state_tuple[2]
                    q_s_a+=  p_next_state*(reward_next_state + gamma*v_ne
                q[s][a] = q_s_a

        new_policy = utils.action_selection(q)

        if np.array_equal(policy, new_policy):
            policy_stable = True

        # YOUR CODE ENDS HERE
        ############################

        return new_policy, policy_stable
```

In [64]:
```
## Testing out policy iteration policy for one episode
GAME = "Taxi-v3"
evaluation_utils.render_episode(env, policy_vi)
env = gym.make("Taxi-v3")
V_pi, policy_pi = policy_iteration(env, gamma=0.95, max_iteration=6000
```

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

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

In [63]:
```
# visualize how the agent performs with the policy generated from poli
evaluation_utils.render_episode(env, policy_pi)
```

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

+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (East)
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (North)
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (Pickup)
+---------+
|R: | : :G|
| : | : :_|
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (South)
+---------+
|R: | : :G|
| : | : : |
| : : : :_|
| | : | : |
|Y| : |B: |
+---------+
  (South)
+---------+
|R: | : :G|
| : | : : |
| : : :_: |
| | : | : |
```

```
|Y| : |B: |
+---------+
  (West)
+---------+
|R: | : :G|
| : | : : |
| : :_: : |
| | : | : |
|Y| : |B: |
+---------+
  (West)
+---------+
|R: | : :G|
| : | : : |
| :_: : : |
| | : | : |
|Y| : |B: |
+---------+
  (West)
+---------+
|R: | : :G|
| : | : : |
|_: : : : |
| | : | : |
|Y| : |B: |
+---------+
  (West)
+---------+
|R: | : :G|
| : | : : |
| : : : : |
|_| : | : |
|Y| : |B: |
+---------+
  (South)
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (South)
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (Dropoff)
Episode reward: 9.000000
```

In [65]:
```
# evaluate the performance of policy iteration over 100 episodes
print(evaluation_utils.avg_performance(env, policy_pi))
```

```
8.484848484848484
```

Part 3: Q-learning and SARSA

3.1 Model-free RL: Q-learning

In this part, you will implement Q-learning.

In [23]:

```
def QLearning(env, num_episodes, gamma, lr, e):
    """
    Implement the Q-learning algorithm following the epsilon-greedy ex
    Inputs:
    env: OpenAI Gym environment
            env.P: dictionary
                    P[state][action] are tuples of tuples tuples with
                    probability: float
                    nextstate: int
                    reward: float
                    terminal: boolean
            env.nS: int
                    number of states
            env.nA: int
                    number of actions
    num_episodes: int
            Number of episodes of training
    gamma: float
            Discount factor.
    lr: float
            Learning rate.
    e: float
            Epsilon value used in the epsilon-greedy method.
    Outputs:
    Q: numpy.ndarray
    """

    Q = np.zeros((env.nS, env.nA))

    ############################
    # YOUR CODE STARTS HERE
    # Set the percent you want to explore
    for i in range(num_episodes):

        s = env.reset()


        done = False
        while not done:
            if np.random.uniform(0, 1) < e:
                """
                Explore: select a random action
                """
                a = np.random.choice(range(env.nA))
            else:
                """
                Exploit: select the action with max value (future rewa
                """
                a = np.argmax(Q[s])

            next_s, r, done, _ = env.step(a)
            next_a = np.argmax(Q[next_s])
            Q[s][a] = Q[s][a] + lr*(r + gamma*Q[next_s][next_a] - Q[s]

            s = next_s

```

```
    # YOUR CODE ENDS HERE
    ############################

    return Q
```

In [17]:

```
def render_episode_Q(env, Q):
    """Renders one episode for Q functionon environment.

      Parameters
      ----------
      env: gym.core.Environment
        Environment to play Q function on.
      Q: np.array of shape [env.nS x env.nA]
        state-action values.
    """

    episode_reward = 0
    state = env.reset()
    done = False
    while not done:
        env.render()
        time.sleep(0.5)
        action = np.argmax(Q[state])
        state, reward, done, _ = env.step(action)
        episode_reward += reward

    print ("Episode reward: %f" %episode_reward)
```

In [24]:

```
Q = QLearning(env = env.env, num_episodes = 1000, gamma = 1, lr = 0.1,
print('Action values:')
print(Q)
```

```
Action values:
[[ 0.          0.          0.          0.          0.          0.
]
 [-3.19794405 -3.87267875 -4.05310345 -3.7965246   7.81255557 -5.1332576
]
 [-0.46050136  0.02574472 -1.7852255  -0.95707712 14.41700161 -1.9119776
7]
 ...
 [-1.1        -1.05841259 -1.1        -1.08586767 -2.         -2.
]
 [-2.58165982 -2.64337669 -2.6556984  -0.57262706 -2.94391081 -3.9352165
]
 [-0.2        -0.2        -0.2         1.909      -1.         -1.
]]
```

In [29]:
```
# Uncomment the following to evaluate your result, comment them when y
env = gym.make('Taxi-v3')
Q = QLearning(env = env.env, num_episodes = 1000, gamma = 1, lr = 0.1,
render_episode_Q(env.env,Q)
```

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

+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (North)
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (West)
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (West)
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (West)
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (South)
+---------+
|R: | : :G|
| : | : : |
```

```
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (South)
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (Pickup)
+---------+
|R: | : :G|
| : | : : |
| : : : : |
|_| : | : |
|Y| : |B: |
+---------+
  (North)
+---------+
|R: | : :G|
| : | : : |
|_: : : : |
| | : | : |
|Y| : |B: |
+---------+
  (North)
+---------+
|R: | : :G|
|_: | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (North)
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (North)
Episode reward: 9.000000
```

3.2 Model-free RL: SARSA

In this part, you will implement Sarsa.

In [19]:

```
def SARSA(env, num_episodes, gamma, lr, e):
    """
    Implement the SARSA algorithm following epsilon-greedy exploration
    Inputs:
    env: OpenAI Gym environment
            env.P: dictionary
                    P[state][action] are tuples of tuples tuples with
                    probability: float
                    nextstate: int
                    reward: float
                    terminal: boolean
            env.nS: int
                    number of states
            env.nA: int
                    number of actions
    num_episodes: int
            Number of episodes of training
    gamma: float
            Discount factor.
    lr: float
            Learning rate.
    e: float
            Epsilon value used in the epsilon-greedy method.
    Outputs:
    Q: numpy.ndarray
            State-action values
    """
    Q = np.zeros((env.nS, env.nA))
    ############################
    # YOUR CODE STARTS HERE
    for i in range(num_episodes):

        s = env.reset()
        if np.random.uniform(0, 1) < e:
            """
            Explore: select a random action
            """
            a = np.random.choice(range(env.nA))
        else:
            """
            Exploit: select the action with max value (future reward)
            """
            a = np.argmax(Q[s])

        done = False
        while not done:

            next_s, r, done, _ = env.step(a)

            if np.random.uniform(0, 1) < e:
                """
                Explore: select a random action
                """
                next_a = np.random.choice(range(env.nA))
            else:
                """
```

```
                Exploit: select the action with max value (future rewa
                """
                next_a = np.argmax(Q[next_s])
            Q[s][a] = Q[s][a] + lr*(r + gamma*Q[next_s][next_a] - Q[s]

            s = next_s
            a = next_a

    # YOUR CODE ENDS HERE
    ############################

    return Q
```

In [20]:

```
def render_episode_Q(env, Q):
    """Renders one episode for Q functionon environment.

      Parameters
      ----------
      env: gym.core.Environment
        Environment to play Q function on.
      Q: np.array of shape [env.nS x env.nA]
        state-action values.
    """

    episode_reward = 0
    state = env.reset()
    done = False
    while not done:
        env.render()
        time.sleep(0.5)
        action = np.argmax(Q[state])
        state, reward, done, _ = env.step(action)
        episode_reward += reward

    print ("Episode reward: %f" %episode_reward)
```

In [21]:

```
Q = SARSA(env = env.env, num_episodes = 1000, gamma = 1, lr = 0.1, e =
print('Action values:')
print(Q)
```

```
Action values:
[[ 0.          0.          0.          0.          0.          0.
]
 [-4.74823032 -4.52974963 -4.50450374 -3.6320741  -0.68727041 -7.3560936
6]
 [-2.4731041  -1.37530551 -2.33613848  0.12065845  9.84693682 -5.0100564
]
 ...
 [-0.95104891 -0.53842485 -0.98139445 -0.93855415 -4.82619345 -2.09
]
 [-3.30326141 -3.12965199 -3.32168319 -3.33724226 -5.5078639  -5.3846284
]
 [-0.281      -0.299      -0.29        8.14927588 -1.9        -1.91
]]
```

In [22]:

```
# Uncomment the following to evaluate your result, comment them when y
env = gym.make('Taxi-v3')
render_episode_Q(env.env,Q)
```

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

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

```
| | : | : |
|Y| : |B: |
+---------+
  (South)
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : |_: |
|Y| : |B: |
+---------+
  (South)
+---------+
|R: | : :G|
| : | : : |
| : : : : |
| | : | : |
|Y| : |B: |
+---------+
  (South)
Episode reward: 12.000000
```