A) Implement a basic driving agent

Implement the basic driving agent, which processes the following inputs at each time step:

- Next waypoint location, relative to its current location and heading,
- Intersection state (traffic light and presence of cars), and,
- Current deadline value (time steps remaining),

And produces some random move/action (None, 'forward', 'left', 'right'). Don't try to implement the correct strategy! That's exactly what your agent is supposed to learn.

Run this agent within the simulation environment with enforce_deadline set to False (see runfunction in agent.py), and observe how it performs. In this mode, the agent is given unlimited time to reach the destination. The current state, action taken by your agent and reward/penalty earned are shown in the simulator.

Q1 *- In your report, mention what you see in the agent's behaviour. Does it eventually make it to the target location?*

Filename: agent_code_for_**Q1**.py
For this task, the following lines of code were updated/added:
(line 29) options = ['forward', 'right', 'left', None]
(line 30) action = random.choice(options)

Observation and agent's behaviour:

- With the code change, the agent moves through the grid using random actions.
- Multiple rewards are given (-1, +0.5, +1, +2)
    - A negative reward of -1 is given when attempting to move left or forward on a red light
    - A reward of +0.5 is given when turning right on a red or green light
    - A reward of +1 is given when waiting at a red/green light
    - A reward of +2 is given when moving on a green light (forward, right, left)
- For the moment, it looks like there is no code preventing collisions with other cars.
- With this code, the agent will only haphazardly reach the destination. At the moment, there doesn't seem to be any motivation or reinforcement within the code to move towards the destination.

B) Identify and update state

Identify a set of states that you think are appropriate for modeling the driving agent. The main source of state variables are current inputs, but not all of them may be worth representing. Also, you can choose to explicitly define states, or use some combination (vector) of inputs as an implicit state.

At each time step, process the inputs and update the current state. Run it again (and as often as you need) to observe how the reported state changes through the run.

Q2 *- Justify why you picked these set of states, and how they model the agent and its environment.*

Filename: agent_code_for_**Q2**.py
For this task, the following lines of code were updated/added:
(line 26) self.state = (self.next_waypoint, inputs)
(line 27) print "self.state:", self.state

Code Justification:
I am selecting two state variables for modeling the current state of my agent: "next_waypoint" and "inputs". The "next_waypoint" state provides information on the agent's next step. The "next_waypoint" will be used to help guide my agent toward the destination target.

The "inputs" state variable provides information on the agent's surrounding environment:
- input 1: red/green light
- input 2: oncoming traffic
- input 3: traffic coming from the left
- input 4: traffic coming from the right

The "inputs" state variable will help me obey the rules of the road and avoid accidents with other agents.

Equipped with this information, additional code can added to select the most efficient path to the destination while avoiding accident's with other agent's.

I chose to exclude the "deadline" state to avoid rushing towards the destination and harming other agent's in the process.

C) Implement Q-Learning

Implement the Q-Learning algorithm by initializing and updating a table/mapping of Q-values at each time step. Now, instead of randomly selecting an action, pick the best action available from the current state based on Q-values, and return that. Each action generates a corresponding numeric reward or penalty (which may be zero). Your agent should take this into account when updating Q-values. Run it again, and observe the behavior.

Q3 - What changes do you notice in the agent's behaviour?

- With Q-Learning, the agent is able to reach the final destination more times than not.
- In the beginning, the agent requires to learn (explore) the environment, actions and rewards to start developing the optimal policy.
- Near the end of the trial, the agent becomes more efficient at reaching the final destination. However, a small number of penalties are incurred when random actions are performed instead of retrieving the optimal action from the Q_table.

D) Enhance the driving agent

Apply the reinforcement learning techniques you have learnt, and tweak the parameters (e.g. learning rate, discount factor, action selection method, etc.), to improve the performance of your agent. Your goal is to get it to a point so that within 100 trials, the agent is able to learn a feasible policy - i.e. reach the destination within the allotted time, with net reward remaining positive.

Q4 - Report what changes you made to your basic implementation of Q-Learning to achieve the final version of the agent. How well does it perform?

I experimented with three Q-learning variables for the final version of the agent:

1. Epsilon
2. Alpha
3. Gamma

Epsilon

I used epsilon to balance the amount of exploration vs exploitation the agent was going to do. Exploration of the environment would trigger when a random generated number would be below the epsilon value. Actual code:

```
if random.random() < self.epsilon_explore_vs_exploit:
        action = random.choice(self.actions) # explore
else:
        action = self.getAction(self.state) # exploit
```

If I set the epsilon value too low, such as 0, the agent would learn nothing and the destination would only be reached haphazardly. If I set the epsilon value too high, the agent would have difficulty improving its knowledge base as it would take a longer time to update q-values for the most optimal policy. I found a good balance for epsilon to be 0.2

In the future, it would be interesting to develop code where the epsilon value morphs over time. In other words, the epsilon value begins at 0.2 and grows closer to 1 when enough exploration has been done (please let me know if I need to develop this code for this assignment).

Alpha

I used the alpha variable to control the learning rate of the algorithm. Assigning a learning rate of 0 made the algorithm collapse as nothing was learned. Setting the alpha to 1 did not give favourable results either as the algorithm did not rely on any past learnings. I found a good balance for the alpha variable to be 0.7

Gamma

I used the gamma variable to discount future rewards. In other words, a low gamma digit (0) gives more importance to immediate rewards where as a high gamma digit

(1) gives more importance to long-term rewards. I found the sweet spot to this variable to be 0.2. I think this is because the next_waypoint functionality was already helping the agent move towards the final destination. I can see other environments and scenarios requiring a higher gamma digit (e.g. saving for retirement).

Q5 - Does your agent get close to finding an optimal policy, i.e. reach the destination in the minimum possible time, and not incur any penalties?
The formulas for updating Q-values can be found in this video.

Interesting question… prior to implementing Q-Learning, I was curious to see how an agent would behave using simple if statements to reach the destination (filename: agent_code_using_simple_if_statements.py). I created this file to see how Q-Learning would compare to coding simple if statements; It turns out that I was able to make the agent reach the final destination 100% of the time with no penalties incurred via simple if statements. With a better understanding of Q-Learning, I now realize that this is an unjust comparison. Creating a series of "if statements" for this game was easy. Creating a series of "if statements" for real-life scenarios would be exhausting. Although Q-Learning is meant for finite scenarios, I can understand the power behind the Q-Learning algorithm as it builds a state/action function based on it's own experiences (Q-values in a Q-table).

Back to Q5, I think I got close to finding an optimal policy for this assignment. The agent was able to reach the final destination more than 90% of the time. However, I did notice that the agent was still incurring penalties but I think this is due to the "random" variable in the code:

```
if random.random() < self.epsilon_explore_vs_exploit:
    action = random.choice(self.actions) # explore
else:
    action = self.getAction(self.state) # exploit
```

I added another piece of code to bypass the random functionality after 100 trials:

```
if self.trial_count < 100:
    # code…
else:
    action = self.getAction(self.state) # full exploit
```

with the above lines of code, the last 20 trials (from 100 to 120) would be processed solely from the q_table. With the randomness taken out of the equation, the penalties in each trial are next to nil.