A) Implement a basic driving agent

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 (now located in the archive folder)
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.

Update after Udacity review…
I added code to measure the performance of the agent driving randomly in the grid (with enforce_deadline=True). After 3 sets of trial runs, the agent was only able to reach the final destination approximately 20% of the time.

Performance (sample output from terminal)

Simulator.run(): Trial 99
Environment.reset(): Trial set up with start = (3, 5), destination = (3, 1), deadline = 20
RoutePlanner.route_to(): destination = (3, 1)
Environment.act(): Primary agent has reached destination!
# of moves: 18
# of penalties: 7
reward total: 0
destination reached: 23 out of 100

The agent reaches the final destination approximately 20% of the time.

B) Identify and update state

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

Filename: agent_code_for_**Q2**.py (now located in the archive folder)
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

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

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	Alpha	Gamma	Total Moves		Total Penalties	Cumulative Rewards	Penalty %		Destination Reached?
1	0.5	0.5	0.5	2236		(295)	2918	10.11%		64 / 100
2	0.5	0.5	0.2	2040		(278)	2748	10.12%		68 / 100
3	0.5	0.7	0.2	2044		(270)	2840	9.51%		74 / 100
4	**0.15**	**0.7**	**0.2**	**1644**		**(66)**	**3094**	**2.13%**		**91 / 100**

As you can see with the above table, I was able to improve the agent's performance by manipulating epsilon, alpha, and gamma values.

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 0.15

Since the last Udacity review, I created a decaying function for both epsilon and alpha values. This approach, for epsilon, reduced the amount of random actions as the trials increased. In turn, the agent relied more on learned q values as the trials progressed.

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.

As discussed previously, I created a decaying function for alpha. This approach reduced the alpha learning ratio to zero as the trials progressed. In turn, the agent relies more on past learning than on new leanings near the end of the 100-set trial.

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?

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 due to running red lights. Below is a sample of penalties incurred…

self.show_penalties [
[-1, {'light': 'red', 'oncoming': None, 'right': None, 'left': None}, 'forward', 1],
[-1, {'light': 'red', 'oncoming': None, 'right': None, 'left': None}, 'left', 1],
[-1, {'light': 'red', 'oncoming': None, 'right': None, 'left': None}, 'forward', 1],
[-1, {'light': 'red', 'oncoming': None, 'right': None, 'left': None}, 'left', 1],
[-1, {'light': 'red', 'oncoming': None, 'right': None, 'left': None}, 'left', 1],
[-1, {'light': 'red', 'oncoming': None, 'right': None, 'left': None}, 'forward', 1],
[-1, {'light': 'red', 'oncoming': None, 'right': None, 'left': None}, 'left', 1],
[-1, {'light': 'red', 'oncoming': None, 'right': None, 'left': None}, 'forward', 1],
….]

note: I removed the code that created the result above from my agent.py file. Although, I still have a copy of the code in my archive folder (and github of course).

It seems to me that the logic in the environment.py file would need to change to prevent the agent from running red lights. Currently, the program is only assigning a "-1" when the agent disregards a red light as it works to move closer to the final destination. In reality, the agent should be instructed to stop at a red light (action = None), unless it's a right turn.