QUESTION1: Observe what you see with the agent's behavior as it takes random actions. Does the **smartcab** eventually make it to the destination? Are there any other interesting observations to note?

After setting random actions for the primary agent, from the output below we can see that the smartcab reaches its destination time to time. It is reached frequently after the deadline.
We can observe that in some cases the reward is negative:
- it is -1.0 when the smartcab makes an invalid move for the game
- it is -0.5 when the smartcab does not take the optimal direction towards the target destination.

```
Simulator.run(): Trial 5
Environment.reset(): Trial set up with start = (1, 1), destination = (7, 5), deadline = 50
RoutePlanner.route_to(): destination = (7, 5)
LearningAgent.update(): deadline = 50, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'left': None}, action = forward, reward = -0.5
LearningAgent.update(): deadline = 49, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None}, action = left, reward = -1.0
LearningAgent.update(): deadline = 48, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None}, action = forward, reward = -1.0
LearningAgent.update(): deadline = 47, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None}, action = right, reward = -0.5
...
...
LearningAgent.update(): deadline = 4, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'left': None}, action = forward, reward = -0.5
LearningAgent.update(): deadline = 3, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None}, action = None, reward = 0.0
LearningAgent.update(): deadline = 2, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'left': None}, action = None, reward = 0.0
LearningAgent.update(): deadline = 1, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'left': None}, action = None, reward = 0.0
Environment.act(): Primary agent has reached destination!
LearningAgent.update(): deadline = 0, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'left': None}, action = right, reward = 12.0
```

QUESTION2: What states have you identified that are appropriate for modeling the **smartcab** and environment? Why do you believe each of these states to be appropriate for this problem?

The environment can be modelled with the following states:
- the traffic light (inputs['light']): for the reward when going right on red and the penalty when going left or forward on red.
- the oncoming traffic (inputs['oncoming']): for the agent being penalized when turning left on green light with oncoming traffic as this can cause an accident.
- the left value (inputs['left']): is required when making a right turn on the red light, so an accident can be avoided .

The planner can be modelled with the state:
- next_waypoint (next_waypoint()) which tells the desirable direction to take.

I choose to not include in the state the case of an oncoming car on the right (inputs['right']). In fact, in that situation, the two cars aren't even on a collision course.

Not including the current deadline value (time steps remaining) for the current agent state, is like considering the agent is given unlimited time to reach the destination, or, looking in another way, like the only factors to consider would be waypoint and game rules.
Since Markov Chain Learning processes are defined in a setting where only the current state matters for describing the decision factors, we are not able to consider the deadline in practice for the Qlearning process.

One way would be adding to the state *the number of allowed not optimal moves* for reaching the destination. This is possible considering the deadline and the destination, which is given to the agent by the method *reset().*
We would have a very large number of states, consisting in all possible combinations, increasing by the factor of a maximum possible deadline for each of the possible destination distances.
The learning process would be very long at the beginning, but this would be a doable way.
The next state is defined clearly by subtracting 1 to the current *number of allowed not optimal moves.*

I would not say that the agent, when this number is low, could learn to not observe game rules and get directly to the destination.
This is already happening, since in the last transition from current state to destination, the environment is not decreasing the value of final reward if the game rules are not observed.
Moreover in this learning model everything is moved by the current *waypoint.* The agent, as observed earlier is updating and looking at Q values of the current state and next state.
In our model, every time we reach the destination, we also give high Q values for the previous state of the destination. This could influence future decisions in any case, considering deadline or not.
We could have added a check, and using a standard 2.0 reward, for calculating the new Qval on reaching the destination.

A better way would be for example introducing another factor in the formula. This factor "*number of allowed not optimal moves"* can be together with the *"exploration_factor".*

QUESTION3: What changes do you notice in the agent's behavior when compared to the basic driving agent when random actions were always taken? Why is this behavior occurring?

The driving agent can reach its destination remarkably faster than choosing random directions.
It has been developed for maximizing the reward at each state. This, for the first trips, is not always true, as the agent has a exploration_factor (epsilon) that allows taking into consideration also moves that consists in a negative reward. In this way, after the learning_factor overcomes the exploration_factor, the agent starts to be trained for optimal choices in the target direction without evading the rules of the game.

The exploration_rate is decreased on every passed trip.
The code below simply compares the final factor with a random vales.
The agent is expected to take 90% of greedy decision after 5/6 completed steps.

```
        random_factor = self.exploration_rate / (self.passed_trips + self.exploration_rate)

        if random.random() < random_factor: # Compare it with random 0..1 values
            action = random.choice(self.directions) # Random action for exploring the enviroment
        else:
            action =  self.greedy_action(state) #Pick the best value for Q - Greedy action
```

QUESTION4: Report the different values for the parameters tuned in your basic implementation of Q-Learning. For which set of parameters does the agent perform best? How well does the final driving agent perform?

For begin my tests i kept the exploration rate to 0.5 because I cannot predict its influence on reward and pass rate before evaluating learning rate and discount factor.

I start with this configuration

```
Learning Rate: 0.3, Discount Factor: 0.7, exploration_rate 0.5,
        Pass Rate: 97.0%, Explored States: 51, Mean Reward: 22.03, Number of Trials: 100

Learning Rate: 0.4, Discount Factor: 0.6, exploration_rate 0.5,
       Pass Rate: 99.0%, Explored States: 56, Mean Reward: 22.055, Number of Trials: 100

Learning Rate: 0.5, Discount Factor: 0.5, exploration_rate 0.5,
       Pass Rate: 98.0%, Explored States: 50, Mean Reward: 21.6, Number of Trials: 100

Learning Rate: 0.6, Discount Factor: 0.4, exploration_rate 0.5,
       Pass Rate: 99.0%, Explored States: 62, Mean Reward: 22.185, Number of Trials: 100

Learning Rate: 0.7, Discount Factor: 0.3, exploration_rate 0.5,
       Pass Rate: 100.0%, Explored States: 59, Mean Reward: 22.2, Number of Trials: 100
```

I saw that as we increase the learning rate, that the mean reward and the passing rate has averagely increased. A second try confirmed it.
However with `Learning Rate: 0.7, Discount Factor: 0.3` the results where somewhat unstable, so I considered for testing the exploration rate this values: `Learning Rate: 0.6, Discount Factor: 0.4`

```
Learning Rate: 0.6, Discount Factor: 0.4, exploration_rate 0.5,
       Pass Rate: 100.0%, Explored States: 56, Mean Reward: 22.66, Number of Trials: 100

Learning Rate: 0.6, Discount Factor: 0.4, exploration_rate 0.6,
       Pass Rate: 98.0%, Explored States: 55, Mean Reward: 22.22, Number of Trials: 100

Learning Rate: 0.6, Discount Factor: 0.4, exploration_rate 0.7,
       Pass Rate: 99.0%, Explored States: 62, Mean Reward: 22.925, Number of Trials: 100

Learning Rate: 0.6, Discount Factor: 0.4, exploration_rate 0.8,
       Pass Rate: 99.0%, Explored States: 49, Mean Reward: 22.45, Number of Trials: 100
```

Choosing slightly higher values exploration rates consisted in an higher number of explored states and thus in an higher mean reward.

For the optimal settings I choose:

```
Learning Rate: 0.6, Discount Factor: 0.4, exploration_rate 0.7,
```

QUESTION5: 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? How would you describe an optimal policy for this problem?

It is really important in our scenario to evaluate what criteria we want to give precedence to:

- mean reward → optimal criteria for example in a pacman scenario
- pass rate → optimal criteria for example in videogame level-quest scenario.

However in a car driving scenario not having an accident is clearly more important of the two criteria above.

This is clearly non the policy followed by our model.
In the example trips below, the last two of a sample 100, we have a violation of game rules:

```
Environment.act(): Primary agent has reached destination!
LearningAgent.update(): deadline = 22, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None},
action = forward, reward = 12.0
Simulator.run(): Trial 96
..
..
..
LearningAgent.update(): deadline = 19, inputs = {'light': 'green', 'oncoming': 'right', 'right': None, 'left': None},
action = forward, reward = 12.0
Simulator.run(): Trial 98
Environment.reset(): Trial set up with start = (8, 2), destination = (5, 4), deadline = 25
RoutePlanner.route_to(): destination = (5, 4)
LearningAgent.update(): deadline = 25, inputs = {'light': 'red', 'oncoming': 'left', 'right': None, 'left': None},
action = forward, reward = 2.0
LearningAgent.update(): deadline = 24, inputs = {'light': 'red', 'oncoming': 'left', 'right': None, 'left': None},
action = None, reward = 0.0
LearningAgent.update(): deadline = 23, inputs = {'light': 'red', 'oncoming': 'left', 'right': None, 'left': None},
action = None, reward = 0.0
LearningAgent.update(): deadline = 22, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'left': None},
action = forward, reward = 2.0
LearningAgent.update(): deadline = 21, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'left': None},
action = forward, reward = 2.0
LearningAgent.update(): deadline = 20, inputs = {'light': 'green', 'oncoming': 'right', 'right': None, 'left': None},
action = left, reward = -1.0
LearningAgent.update(): deadline = 19, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None},
action = None, reward = 0.0
LearningAgent.update(): deadline = 18, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None},
action = None, reward = 0.0
LearningAgent.update(): deadline = 17, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None},
action = None, reward = 0.0
LearningAgent.update(): deadline = 16, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'left': None},
action = left, reward = 2.0
Environment.act(): Primary agent has reached destination!
LearningAgent.update(): deadline = 15, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'left': None},
action = forward, reward = 12.0
Simulator.run(): Trial 99
Environment.reset(): Trial set up with start = (6, 5), destination = (2, 3), deadline = 30
RoutePlanner.route_to(): destination = (2, 3)
LearningAgent.update(): deadline = 30, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'left': None},
action = right, reward = 2.0
LearningAgent.update(): deadline = 29, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'left': None},
action = right, reward = 2.0
LearningAgent.update(): deadline = 28, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'left': None},
action = None, reward = 0.0
LearningAgent.update(): deadline = 27, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None},
action = None, reward = 0.0
LearningAgent.update(): deadline = 26, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None},
action = None, reward = 0.0
LearningAgent.update(): deadline = 25, inputs = {'light': 'red', 'oncoming': 'right', 'right': None, 'left': None},
action = forward, reward = -1.0
LearningAgent.update(): deadline = 24, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None},
action = forward, reward = 2.0
LearningAgent.update(): deadline = 23, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None},
action = None, reward = 0.0
LearningAgent.update(): deadline = 22, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None},
action = None, reward = 0.0
LearningAgent.update(): deadline = 21, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'left': None},
action = forward, reward = 2.0
```

```
LearningAgent.update(): deadline = 20, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None},
action = forward, reward = 2.0
LearningAgent.update(): deadline = 19, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None},
action = right, reward = 2.0
LearningAgent.update(): deadline = 18, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None},
action = None, reward = 0.0
LearningAgent.update(): deadline = 17, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None},
action = None, reward = 0.0
LearningAgent.update(): deadline = 16, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None},
action = None, reward = 0.0
LearningAgent.update(): deadline = 15, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None},
action = forward, reward = 2.0
LearningAgent.update(): deadline = 14, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None},
action = None, reward = 0.0
LearningAgent.update(): deadline = 13, inputs = {'light': 'red', 'oncoming': None, 'right': None, 'left': None},
action = None, reward = 0.0
Environment.act(): Primary agent has reached destination!
LearningAgent.update(): deadline = 12, inputs = {'light': 'green', 'oncoming': None, 'right': None, 'left': None},
action = forward, reward = 12.0
Learning Rate: 0.6, Discount Factor: 0.4, exploration_rate 0.7, Pass Rate: 96.0%, Explored States: 61, Mean Reward:
22.555, Number of Trials: 100
```

Like explained in question 2, it is possible to see that the Qvalue for some states has been clearly influenced by the reaching of the destination in a previous trip.

This caused the Qvalue to be high, and the "*apparent*" utility of the next state even taking a penalty.

Generally speaking the agent follows the directions of the planner, but in these not so rare cases, it does not resolve in a *None* move, but moves towards a state whose Qvalue has been increased by reaching the destination in previous trips.

This can be confirmed by looking at previous trips in the log. We could prevent this by a change in the update() function code if requested.