1. I created a matrix 48*2*8*4 for the states, 48 gridpoints,
light red or green, all combinations of traffic, and 4 directions
the car can face. Am i on the right track, having that many states (over 3000) ?
answer :- No, its wrong way and certaily no. of states will be far less than this.

2. 384 states - correct states.
96 - total perfect moves needed

3. say you don't carry over this optimistic assumption the first time you update
the Q-value, and let's ignore the future state (gamma is zero). Then these are
your options:

When you go to the state and perform a None action for the first time, your Q-value
for (state, None) drops to 0 (since that's always the reward for a None action).

When you make an incorrect/illegal move, the Q-value for (state, incorrect_move) drops
to a negative value (-0.5 or -1.0).

When you make the correct move, the Q-value for (state, correct_move) drops to 2.0.
So, you need at most 4 visits to the state to get a working Q-table entry for it.

-> That's 384 * 4 visits = 1536 steps needed to get all Q-table entries right. But
that's the best-case scenario in which you don't repeat any states, and as you note
there are lots of unvisited states in the course of a simulation.

4. from itertools import product

You can read the docs for more details but the gist is you pass product a bunch of
iterables and it does the logic of nested multiple loops for you.

product('ab', range(3)) --> ('a',0) ('a',1) ('a',2) ('b',0) ('b',1) ('b',2)

Another approach could be to fill in the dictionary as you go. Something like

```
for action in actions:
    if (state, action) not in q_table:
        q_table[(state, action)] = 0
    else:
        # do something different
```

An alternative Q-Table representation could be q_table[state] = [0, 0, 0, 0],
state is a tuple and the list contains the values of the 4 actions.

5. For instance, q_table.get((state, action), 0) would return q_table[(state, action)]
if (state, action) is a key in q_table, and 0 otherwise. So you can use q_table.get()
anywhere you would use the q-value for a state/action pair.

6. In order to do Q-Learning we need to know the next state/action pair, so that we can do:
max(a'){Q(s',a')},

That is, loop over the actions, go to (find out) the next state which those actions
take you to, and determine which respective Q(s',a') is the max value. (and thus pick
an optimal a' = next action for the policy.)

where s' is the next state and a' is the action that takes you to that next state
FROM THE PRESENT STATE.
So we need to know the present state = s, also. And we need to know R=reward for
going to this next state.

Then we can do: Q(s,a) = (1-alpha)Q(s,a) + alpha(R + gamma*Q(s',a'))
(ignoring epsilon-greedy for the moment)

Thus we need to know: present state = s, next state = s', and reward = R for going to the next state. Also we need to do this for all possible actions AND we need to do this prior to the simulator actually performing the action.

7. The epsilon-greedy policy tells you to pick a random action 1-x per cent of the time, and to follow the optimal a' x% of the time (according to max(a'){Q(s',a')}).

8. a is the action you take in state s that gets you to state s'. Then a' is the action you take in state s' to get you to the next state.

9. Remember the value that is propagating backward from the "next" state, Q(s',a') is the Q-value, not the reward value! So you don't need to get the reward for the state/action pair (s',a'), you just need to know what your current Q-values are for the state. These should start from some default that you set.

Also perhaps it's worth noting, a' is the action you intend to take in the state s'; it is not the action that took you to s'. So if you remember s and a, then a takes you to s', then you can look at your Q-table to pick an a' and propagate the Q-value backward.

10. action = max(self.q_table[self.state],
key=self.q_table[self.state].get)

11. alpha = 0.5
gamma = 0.6

12. import operator

```
stats = {'a':1000, 'b':3000, 'c': 100}
max(stats.iteritems(), key=operator.itemgetter(1))[0]
```

13. # TODO: Select action according to your policy

```
action = max(self.q_table[self.state], key=self.q_table[self.state].get)

# Calculate the Q_value
q_value = (1 - alpha) * self.q_table[self.state][action] + \
        alpha * (reward + gamma * max(self.q_table[state_new].values()))
```

14. The Q-value is the long term expected reward whereas the reward is the immediate reward for taking a state action pair.

15. The agent would choose 'NONE' eventually because it has reward=0, which is better than -ve and receive rewards from it continuously (exploiting). This is where we need to some times randomize the choice in order to 'escape' the local minima (exploring).

16. 'plan' (exploit) and 'learn' (explore)

17. Firstly, the actions in this environment are deterministic per se (and not probabilistic or stochastic like some of the examples cited on this thread).

But the environment is dynamic as it can change spontaneously without the agent taking any action (lights changing, other cars moving).

Here's the interesting part: Depending on how you define the agent's state, i.e.

if you include the state of the traffic light at the current intersection, next waypoint, etc., the next state resulting from taking an action can be different each time! You could thus treat the environment as being probabilistic/stochastic.

But none of this really affects why Q-learning is suitable here. The primary reason is that it is a model-free learning technique. There are a lot of states that make up the problem space (all combinations of traffic light states, next waypoints, other cars at the intersection, etc.), and with Q-learning you can avoid having to create an internal representation of all the transitions and expected rewards. Q-learning can work with deterministic as well as probabilistic environments equally well.

Of course, the algorithm will be more effective if you can generalize and reduce the number of states, or ignore inputs that don't matter. That's where your creativity comes into play!

18. There is one important reason that why you need to use a model-free algorithm like Q-Learning for this environment, because it doesn't care as much about what the next state is (it only looks at the best value possible from the next state).

19. here is an example of Q-table being stored as a dictionary of dictionaries :

```
{state1: {"action_1": 0.56,  "action_2": 2.31,  "action_3": -0.72}
 state2: {"action_1": 1.21,  "action_2": 0.13,  "action_3":  1.89}
 ...
 ...
 stateN: {"action_1": -0.27, "action_2": -0.57, "action_3":  3.73}
}
```

So, to illustrate how the API mentioned by @Jared works for this example is:

q[state1]["action_1"]

Returns the value: 0.56

20. Q(state, action) = R(state, action) + Gamma * Max[Q(next state, all actions)]

Our virtual agent will learn through experience, without a teacher (this is called unsupervised learning). The agent will explore from state to state until it reaches the goal. We'll call each exploration an episode. Each episode consists of the agent moving from the initial state to the goal state. Each time the agent arrives at the goal state, the program goes to the next episode.

The Q-Learning algorithm goes as follows:

1. Set the gamma parameter, and environment rewards in matrix R.
2. Initialize matrix Q to zero.
3. For each episode:
   Select a random initial state.
   Do While the goal state hasn't been reached.
    1. Select one among all possible actions for the current state.
    2. Using this possible action, consider going to the next state.
    3. Get maximum Q value for this next state based on all possible actions.
    4. Compute: Q(state, action) = R(state, action) + Gamma * Max[Q(next state, all actions)]
    5. Set the next state as the current state.
   End Do
  End For

The algorithm above is used by the agent to learn from experience. Each episode is

equivalent to one training session. In each training session, the agent explores the environment (represented by matrix R), receives the reward (if any) until it reaches the goal state. The purpose of the training is to enhance the 'brain' of our agent, represented by matrix Q. More training results in a more optimized matrix Q. In this case, if the matrix Q has been enhanced, instead of exploring around, and going back and forth to the same rooms, the agent will find the fastest route to the goal state.

The Gamma parameter has a range of 0 to 1 . If Gamma is closer to zero, the agent will tend to consider only immediate rewards. If Gamma is closer to one, the agent will consider future rewards with greater weight, willing to delay the reward.

To use the matrix Q, the agent simply traces the sequence of states, from the initial state to goal state. The algorithm finds the actions with the highest reward values recorded in matrix Q for current state:

Algorithm to utilize the Q matrix:

1. Set current state = initial state.
2. From current state, find the action with the highest Q value.
3. Set current state = next state.
4. Repeat Steps 2 and 3 until current state = goal state.

The algorithm above will return the sequence of states from the initial state to the goal state.

visit the following URL for the best explanation of Q-Learning :

http://mnemstudio.org/path-finding-q-learning-tutorial.htm

21. You should avoid using None since the objective of selecting random actions is to learn new states.