# Smart Cab

**Implement a basic driving agent**

Q1: In your report, mention what you see in the agent’s behavior. Does it eventually make it to the target location?

**Identify and update state**

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

**Implementing Q-Learning**

*Q3:* What changes do you notice in the agent’s behavior?

**Enhance the driving agent**

Q4a: 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?

Q4b: 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?

**Implement a basic driving agent**

The driving agent needs to make a decision about where to go. I will consider 4 different ‘strategies’:

* stubborn: agent always makes a left turn
* random: agent takes a random action out of ‘left’, ‘right’, ‘forward’, or ‘stay’ (no action taken)
* guided: agent always follows the planner if light is green
* qlearner: agent uses reinforcement learning (Q-learning) to find the ‘best’ action to take

Figure 1 shows paths followed by an agent following each of the strategy, whereas Figure 2 shows the cumulative reward, respectively. I fixed the code so that they all start from the same location and have the same destination.

Figure 1

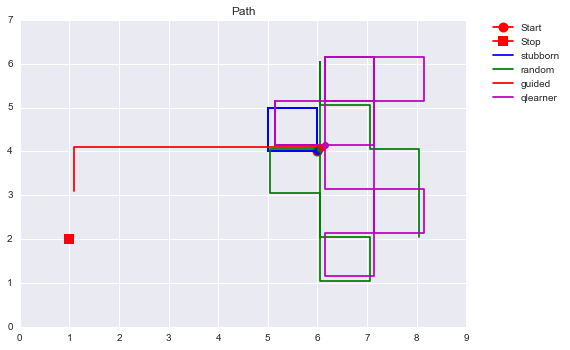
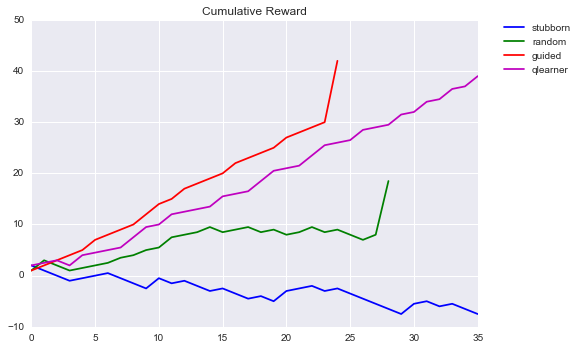


Figure 2



As we can see the stubborn driver keeps looping around without going anywhere. The random driver was actually lucky in this case and ended up at the destination. As a preview, the Q-learner is exploring the space but is not reaching its destination. More comments on this are given below.

For this section I will consider the strategy ‘guided’, which essentially follows the planner:

# Guided

action= None

if inputs['light']=='green':

action = self.next\_waypoint

Running several trials I observe that it normally reaches the destination in time before the deadline.

**Identify and update state**

The state should be a function of what type of inputs we are sensing from the environment. While distance to the destination, etc. could be useful information I assume these remain unknown and we can only sense the traffic light and oncoming traffic. The simplest definition of state (I can think of) is the color of the traffic light. This makes the ‘state space’ limited to 2 states.

One could also add a state that represents the intersection, generically:

* free: when there is no incoming traffic
* busy: if there is any car, left, right, or incoming

This would make the state space be 4-dimensional: red/green x free/busy. If one wanted to use all the available information the state space would be 2 x 4 = 8-dimensional.

To start, I picked the simplest description of the environment: red or green. This will help building intuition of the algorithm.

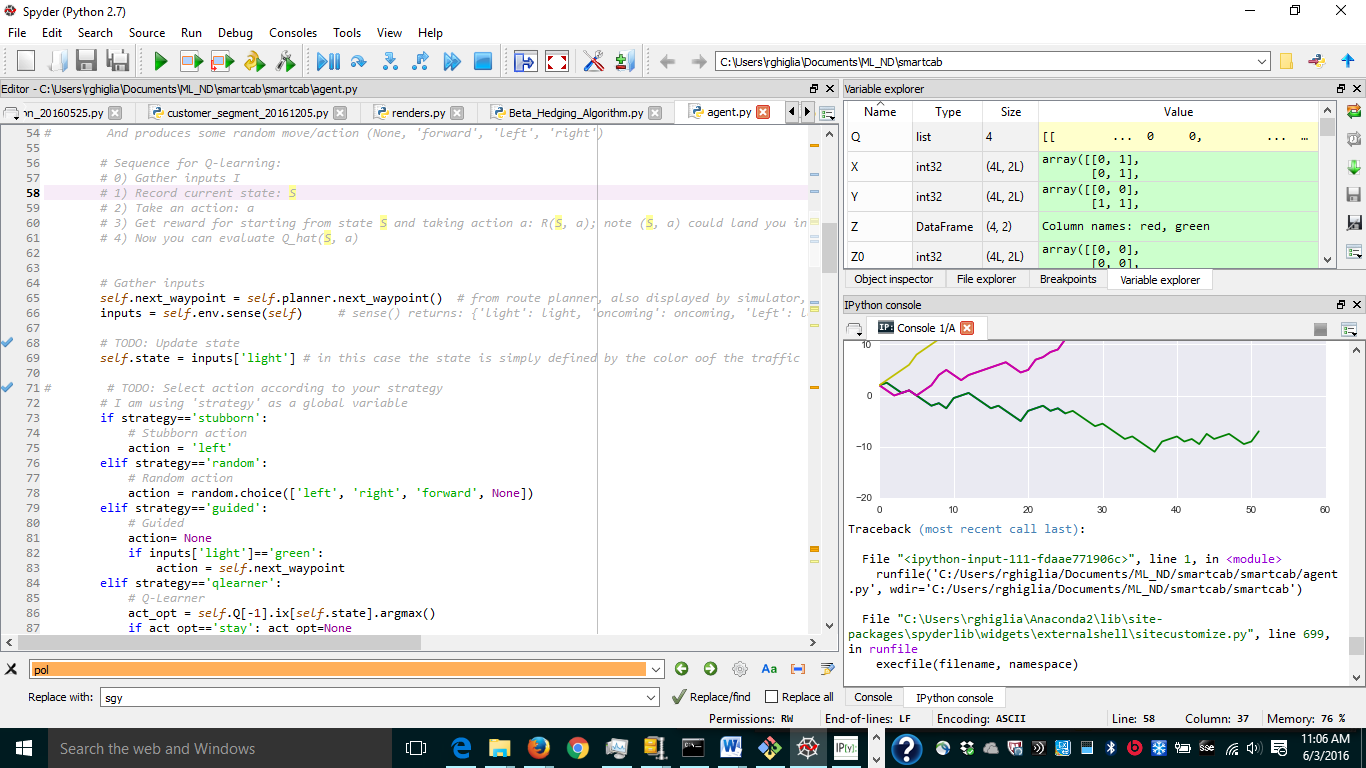
The questions asks about the ‘agent and its environment’; I not sure if I should interpret it as those two in isolation or in combination. The explanation below tries to address both interpretations.

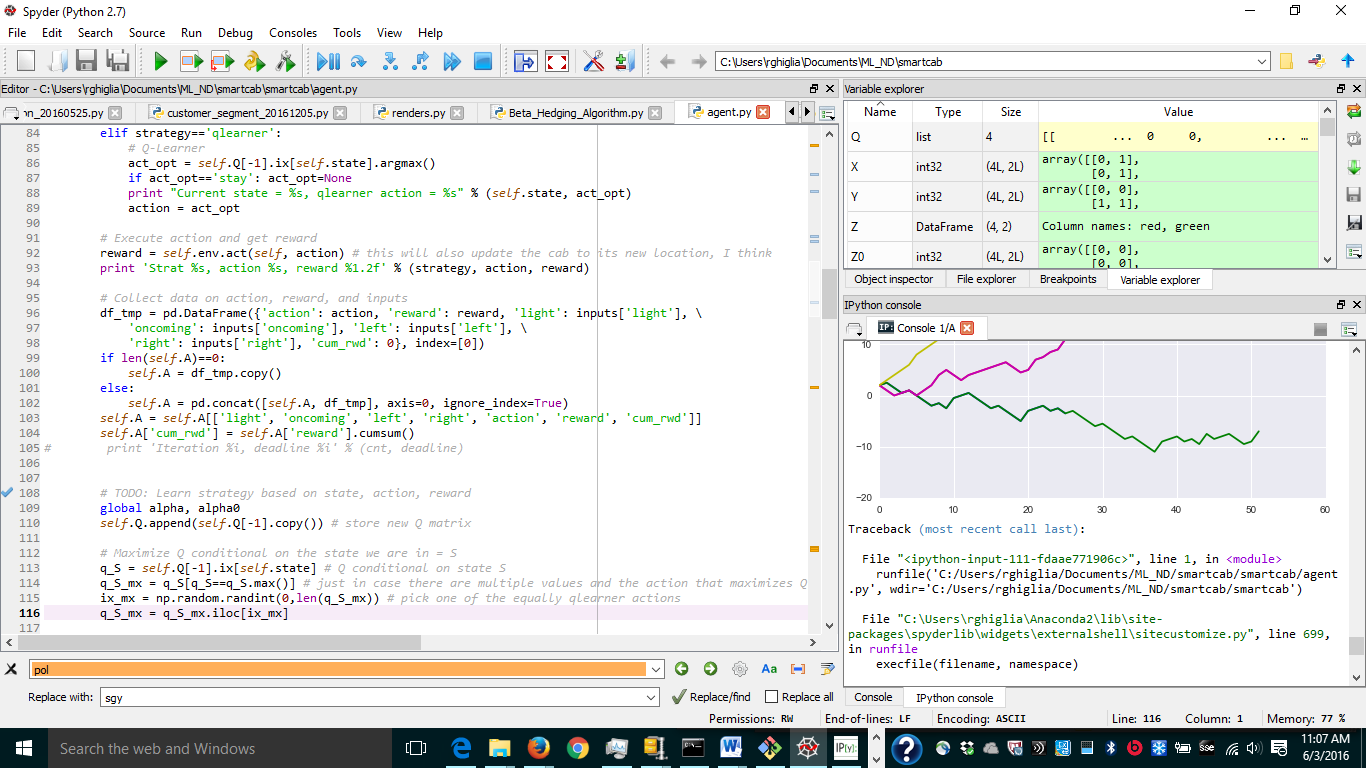
These inputs do not describe the agent though. It seems to me none of the available inputs are descriptive of the agent himself. Examples of that would be the distance to the target. Although, strictly speaking, the latter would also not be describing the agent independently but rather the relation of the agent to the environment. A state that would be strictly relative to the agent could be something like the agent’s health affecting his ability to drive or something.

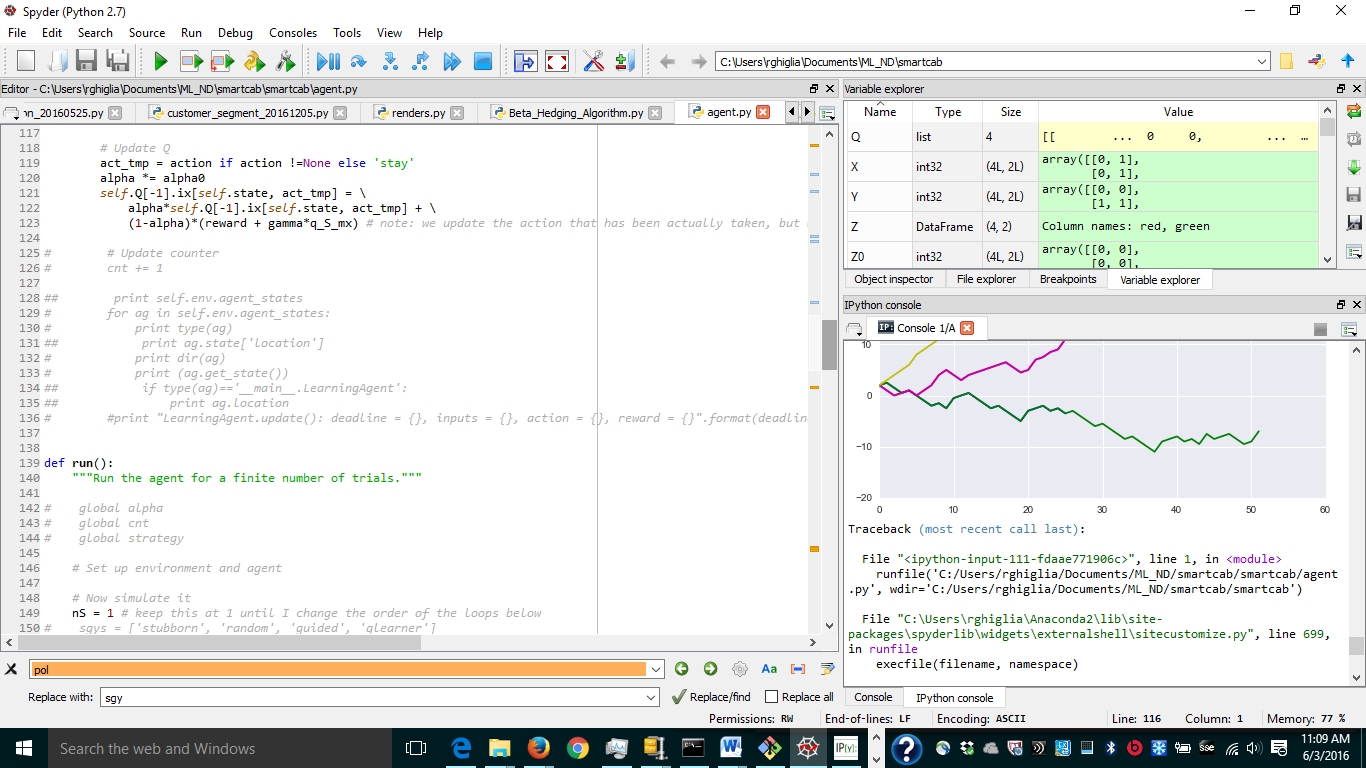
**Implementing Q-Learning**

The ‘guided’ strategy in sense is a bit like cheating because the planner has likely more global (i.e. extended) information than the agent and can therefore be more effective. The ‘Q-learning’ strategy estimates an optimal combination of state and action by trial and error. It takes actions, gets a reward and incorporates (updates) that information into an estimate of how best to proceed.

Figure 3







As such, the agent wanders around a lot more and the success rate is a lot lower than the ‘guided’ strategy. It is hard to describe the actual behavioral changes without collecting more data. From visual inspection, it seems that in the ‘guided’ strategy the agent tries to go on more straight-line paths.

I ran 5 simulations and kept track of the paths for each strategy (Figure 4, start = black circle, destination= black square)[[1]](#footnote-1). The results confirm that observation. The guided strategy gets there in a most linear way, whereas the q-learner wanders around a lot more. Occasionally, the Q-learner does better than the guided driver, however that is probably mostly by chance[[2]](#footnote-2). The purely random driver never reached the destination.

Figure 4

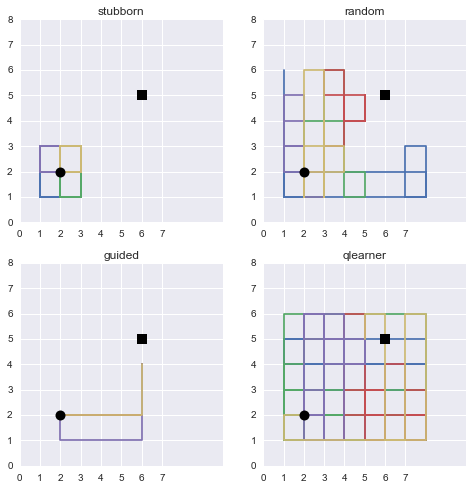
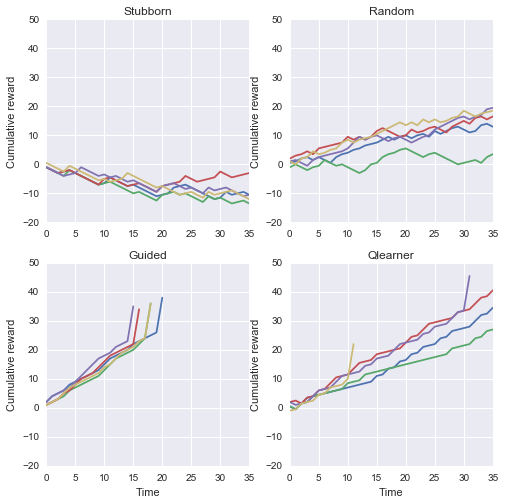


Figure 5



In this small sample the success rate is 100% for the guided learner, 40% for the Q-learner and 0 for the others.

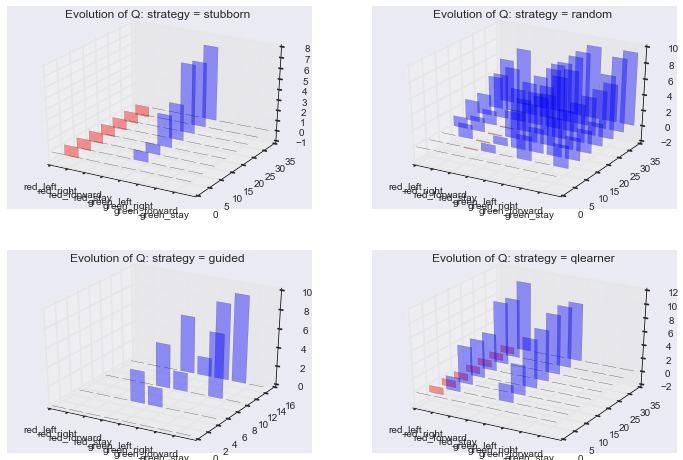
**Enhance the driving agent**

We can visualize the estimated value function Q for all strategies as shown in Figure 6. Q is really a function of 3 variables: state, action, and the updating step. I flattened the state-action variables so that we can see all 8 components of the Q matrix and their evolution over time.

As we might expect, the stubborn strategy only visits 2 state-action combinations: red-left and green-left. The red-left is consistently a bad choice and is maxed by other alternatives that have at least the value of zero as Q was initialized to 0.

The random strategy visits a lot of state-action combinations, whereas the guided and q-learning strategies only a few. However, it is noticeable that q-learning strategy switches from preferring green\_left to red\_right a couple of times.

Figure 6



**Enhance the driving agent**

In this section I will focus on the q-learning agent. As a test case, I ran 100 simulations and obtained the following success rate:

guided 0.99

qlearner 0.26

Alpha is set to 0.5 and decays at 0.5^t.

self.Q[-1].ix[self.state, act\_tmp] = \

alpha\*self.Q[-1].ix[self.state, act\_tmp] + \

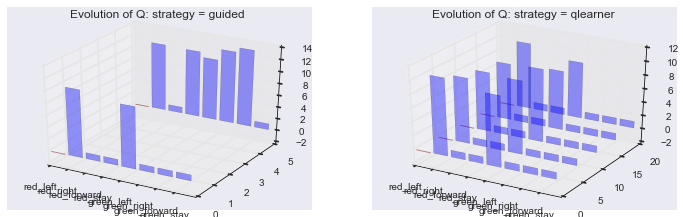
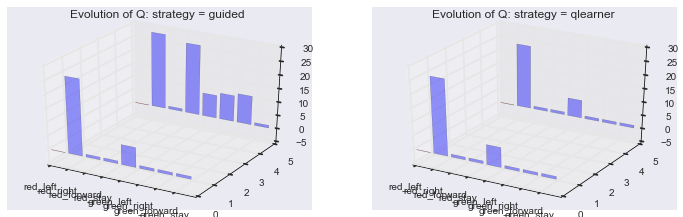
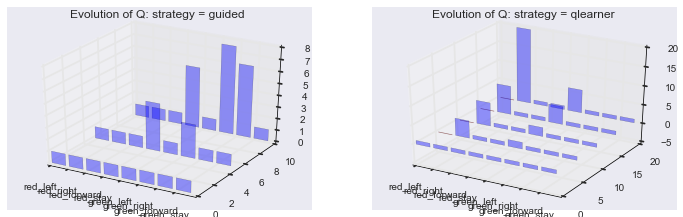
(1-alpha)\*(reward + gamma\*q\_S\_mx)

A larger alpha creates more inertia, i.e. the function Q incorporates new information at a slower rate. Gamma controls the contribution of the future expected utility compared to the immediate reward and is set to 0.9. Q has been initialized to zero.

The first improvement is to initialize Q to a larger number so that more of the space gets sampled. With Q = 10, the success rate went down to 0.02, whereas with Q=1 it went up to 0.7.

I then implemented a cross-trip learning whereby the best estimate of Q is transferred from one trip to the next, i.e. the Q function that is learned during a trip becomes the starting estimate of the Q function at the start of the next iteration. Figure 7 shows the evolution of the Q function estimate in the first trip (top left panel), the second (top right panel) and the last trip (bottom left panel). The success rate was 0.5.

Figure 7



As far as reaching the destination in the minimum time, I collected some statistics that measure the efficiency of reaching the destination. An efficiency of 1 would indicate that the target has been reached in 1 step, in general efficiency = 1 – time spent / deadline

Efficiency:

guided qlearner

0 0.520000 0.24

1 0.600000 0.36

2 0.700000 0.75

3 0.550000 0.00

4 0.657143 0.00

5 0.133333 0.90

6 0.520000 0.68

7 0.800000 0.00

8 0.650000 0.00

9 0.750000 0.00

The table shows that at times the q-learner reaches the destination faster than the guided solution. It does not reach the destination all the time though.

The following table shows all the actions and rewards taken during the last iteration and it shows that no action was taken that resulted in a negative reward.

# light oncoming left right action reward cum\_rwd

0 red None None None right 0.5 0.5

1 green None None None left 2.0 2.5

2 red None None None right 0.5 3.0

3 red None None None right 0.5 3.5

4 green None None None left 0.5 4.0

5 red None None None right 0.5 4.5

6 green None None None left 0.5 5.0

7 green None None None left 0.5 5.5

8 red None None None right 0.5 6.0

9 green None None None left 2.0 8.0

10 green None None None left 0.5 8.5

11 green None None None left 0.5 9.0

12 green None None left left 0.5 9.5

13 red None None None right 0.5 10.0

14 green None None None left 0.5 10.5

15 red None None None right 2.0 12.5

16 red None None None right 0.5 13.0

17 red None None None right 0.5 13.5

18 red None None None right 2.0 15.5

19 green None None None left 0.5 16.0

20 red None None None right 0.5 16.5

1. Note that the paths do not show the very last step because the code exists before recording the very last step. [↑](#footnote-ref-1)
2. One can identify reaching the target by the big jump in the reward function. [↑](#footnote-ref-2)