Question 1:

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?

Answer 1:

In my case, with random actions smartcab reached destination in about 20 times. Which is quite a good result from my point of view if you make decision randomly. Unfortunately, there is a persistent amount of penalties for a smartcab. And this is not good. Let's see if we can improve it by applying Reinforcement Learning techniques.

Question 2:

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?

Answer 2:

Here are a list of appropriate inputs for modeling the smartcab and environment:

1. **Light**. Possible states: Green, Red. Green light means that agent can perform next action with exception on left turn. Red light means that agent cannot perform action with exception on right turn.

2. **Oncoming**. Possible states: None, forward, left, right. Defines whether there is oncoming traffic or not and which direction it goes. With oncoming traffic agent may not turn left with green light or turn right with red light.

3. **Right**. Possible states: None, forward, left, right. Defines whether there is traffic from the right of the agent or not and which direction it goes. Right-of-way rules don't take right-side traffic into account so this property is unnecessary.

4. **Left**. Possible states: None, forward, left, right. Defines whether there is traffic from the left of the agent or not and which direction it goes. Traffic from left going forward means that agent cannot turn right on red light.

5. **Next waypoint**. Possible states: forward, left, right. Defines which direction agent should go to reach the destination. Without this feature agent won't know where to go.

6. **Deadline**. Meaningless feature since agent looks for optimal route without taking deadline state into account. This feature cannot force agent make other decision.

The agent has four valid actions: *None, left, right, forward*.

Agent can perform action *None* if light is red and there is a traffic from left going forward. Based on that inputs **Light** and **Left** required.

Agent can perform action *left* if light is green and there is no oncoming traffic going forward or turning left. Based on that inputs **Light** and **Oncoming** required.

Agent can perform action *forward* if light is green. Based on that input **Light** required.

Agent can perform action *right* if light is green or red and there is no traffic from left going forward. Based on that inputs **Light** and **Left** required.

Another required input is **Next waypoint**. Information about **Light** and traffic just tells us information about state on a street. But it doesn't give us any information about where to go. So **Next waypoint** input is mandatory.

Other input features don't give us any useful information. Right-of-way rules don't mention traffic to the right, so this is unnecessary information that doesn't need to be in the state for the agent to learn the optimal policy. Deadline input doesn't give us any useful information as well. Since rewards for correct and incorrect actions are constant this feature cannot make any impact on a decision where to go. Based on that both these features are not important.

So the state will consist of 4 features (Light (Red, Green), Oncoming (None, forward, left, right), Left (None, forward, left, right), Next waypoint (forward, left, right)). By counting them we get 2*4*4*3=**96** unique states. An overall amount of all possible states is equal to 2*4*4*4*3=**384**. Such a big number of states doesn't seem reasonable since it requires much more calculation.

Question 3:

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?

Answer 3:

Agent which performed random actions just walked through the grid and nothing else. Reaching destination was just a matter of luck. One time agent could turn left, make several actions forward and then get back where he started. However, agent didn't break any rule. It also surprised that sometimes agent actually reached a destination.

At the beginning when I launched Clever Agent it behaves quite same. It made circles before reaching a destination. Turned in incorrect direction. Sometimes it didn't reach destination at all. But each trial he accomplished better and better. After I started to use Q-value to choose next action, the number of times when agent reached destination grew up dramatically from 20 to 98. Fortunately, there was a way to simplify Q-learning algorithm. Since environment produces an immediate reward for each action which makes agent closer to destination it is enough to make a decision and ignore long-term reward by setting discount factor equal to zero.

Question 4:

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?

Answer 4:

During work on a project I developed 4 different agents:

1. **Random Agent.** Execution cmd: `python smartcab/agent.py random_agent.RandomAgent`. Agent chooses best actions randomly. I was very surprised that agent reached its destination in 20% of trials. From my perspective this is a very good result.
2. **Clever Agent**. Execution command: `python smartcab/agent.py clever_agent.CleverAgent`. Agent chooses next action based on Q-Value. The higher Q-Value action has, the better is the action. As I mentioned in **Answer 3** the discount factor is set to zero, since long-term reward doesn't make any impact. Q- Learning Function learning rate is a time inversion. The more time agent drives, the lower is learning rate.
3. **Positive Agent**. Execution cmd: `python smartcab/agent.py positive_agent.PositiveAgent`. This agent uses strategy optimism-in-face-of-uncertainty. Each time agent finds an action without a Q-Value it expects that it has a very high one. Learning rate is same as for **Clever Agent**.
4. **Learner Agent**. Execution command: `python smartcab/agent.py learner_agent.LearnerAgent`. This agent uses same Q-Learning function as **Clever Agent** but different learning rate algorithm. Instead of taking time inverse (1/1, 1/2, 1/3 etc.) it inverses square root of time (1/sqrt(1), 1/sqrt(2), 1/sqrt(3) etc). By applying this algorithm the learning rate doesn't reduce so fast.

Below presented results of each agent:

	Random Agent	Clever Agent	Positive Agent	Learner Agent
Mean penalty per 100 trials	1581.4	27.35	40.4	29.6
Std. Dev. penalty per 100 trials	75.96	5.35	23.94	9.48
Mean trial time	27.79	13.33	13.28	13.23
Std.Dev. trial time	1.19	0.84	0.79	0.58
Mean success rate per 100 trials	0.20	0.9905	0.98	0.995
Std.Dev. success rate per 100 trials	0.027	0.009	0.014	0.007

So here we have results for 4 agents. It's completely clear that Random Agent is an outsider. Agent reaches destination in only 20% of trials, while the rest does it in 98%-99.5%. If we look on success rate of each agent, than it looks like Learner Agent is the best one. Even though it's just a 0.5%, from my perspective in such critical area like autopilot it is valuable. On the other hand, mean penalty of Learner Agent and penalty's standard deviation are higher for Learner Agent, while Clever Agent has the lowest numbers. As far as I understand, it means that Learner Agent may require more data to learn than Clever Agent. So if I was need to choose one of those agents, probably it will depend on how many training data I have. Between those 3 Positive Agent looks like the worst choice. It has lowest success rate and high penalty number.

Modifying the Learning Rate decay

In this section I will try to modify learning rate decay to find best configuration. To be able to do that I will modify my current algorithm, so there will be a place for tuning. So while original algorithm looked like this: `learning_rate = 1.0/sqrt(self.time)` the new one has an additional multiplier

which could be tuned. Here is the updated version of this algorithm: `learning_rate = 1.0/sqrt(self.time * self.mult)`. The multipliers list is following: [0.25, 0.3, 0.35, 0.5, 0.75, 1, 1.25, 1.45, 1.5, 1.55, 1.6]. I ran 20 trail sets, 100 trails each for each multiplier. All code presented in script `optimal_agent.py` which could be run as following python smartcab/optimal_agent.py. Here are statistic for each multiplier:

Mult	Mean Penalty	Median Penalty	Std.Dev Penalty	Mean Trial Time	Median Trial Time	Std.Dev Trial Time	Mean Success Rate	Median Success Rate	Std.Dev Success Rate
0.25	29.6	28.0	4.24	13.32	13.47	0.63	0.9915	0.99	0.0072
0.3	28.1	27.5	4.77	13.15	13.16	0.56	0.98	0.99	0.0099
0.35	31.35	30.5	7.49	13.63	13.47	0.68	0.99	0.99	0.0118
0.5	29.7	31.0	5.56	13.25	13.24	0.70	0.9925	0.99	0.0062
0.75	33.8	31.5	8.02	13.02	13.16	0.52	0.9925	0.99	0.0082
1	29.6	30.0	3.49	13.35	13.24	0.71	0.9915	0.99	0.0085
1.25	28.3	30.0	3.30	13.14	13.00	0.85	0.995	0.995	0.005
1.45	29.25	26.5	9.36	13.03	13.06	0.64	0.993	0.995	0.0078
1.5	28.2	28.0	3.68	13.23	13.26	0.61	0.993	0.995	0.0084
1.55	31.0	27.5	10.95	13.32	13.41	0.65	0.9915	0.995	0.0096
1.6	29.3	29.5	3.68	13.35	13.28	0.79	0.99	0.99	0.0083

As we can see above, the multiplier **1.25**(highlighted in blue) has the best in class both mean and median success rate, and one of the lowest in class average penalty. Moreover, I found very interesting behavior. Multiplier values between **1.0** and **1.6** (both exclusive) have highest mean and median success rate. So from my perspective best multiplier should be somewhere from this range. In this benchmark multiplier **1.25** showed the best result, since it has the lowest success rate standard deviation, which means that all success rate values are very close to the average.

Question 5:

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?

Answer 5:

As a conclusion I would choose **Learner Agent** as the best one. This agent showed best results, which were quite close to the original **Clever Agent** configuration, but in such serious area like self-driving vehicles even such a small difference is very important.

From my point of view optimal policy for this problem is as mentioned in question “reach destination in the minimum possible time and not incur any penalties”. This means that for each trial agent should find the shortest way to destination point. Following game rules, this path shouldn’t have any penalties, since each action makes you closer to the destination. Below presented performance tests for each agent.

Agent	Lower Quartile	Whole Dataset	Upper Quartile
Random Agent	0.044	0.0155	0.004
Clever Agent	0.7	0.791	0.84

Positive Agent	0.754	0.819	0.848
Learner Agent	0.762	0.82	0.888

Lower Quartile – Column represents how many optimal trials was accomplished by agent on first 25 trials divided by this amount of trials.

Whole Dataset - Column represents how many optimal trials was accomplished on a whole dataset comparing to a a overall amount of completed trials.

Upper Quartile - Column represents how many optimal trials was accomplished by agent on last 25 trials divided by this amount of trials.

This table demonstrates that **Learner Agent** regularly shows best results comparing to other agents as well as it learns from its experience. The more trials agent passed, the higher is chance that next trial will be completed with optimal route.