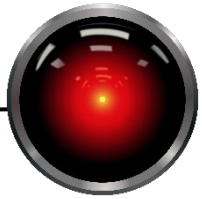


Part I – Do this part on your own. Remember, that as always, you must write answers in your own words. Failure to do so is plagiarism.

Questions 1-3, 10 points each, 4-6 are 20 points each

1. Williams and Kessler argue that pair programming only works if you are able to accept criticism. Briefly describe the anecdote that they provide to illustrate how being able to accept criticism leads to better code.
2. What is the role suggested by Williams and Kessler for the person who is not currently using the keyboard?
3. Why do Williams and Kessler suggest that taking turns at the keyboard is essential?
4. Would a landing agent for the SpaceX Falcon 9 reusable rocket be considered an episodic or sequential agent? Justify your answer.
5. Describe the difference between goal-based and utility-based agents.
6. Suppose a farm has an irrigation and soil management system that is controlled by a learning agent. The performance element has rules about watering, fertilizing, etc. Provide two examples of rules that might be modified by the agent's learning element and a trigger that might cause these rules to be invoked. You are free to assume any reasonable percepts as well as metrics from the performance element.

Part II – Getting your feet wet with Python & Agents (100 points), you may pair program

In this assignment, you will use a Pacman framework developed by UC Berkeley to implement a simple agent that will manage to evade Pacman ghosts. If you have never played Pacman, a brief synopsis is available on Wikipedia. The Berkeley version implements a variant of the game and may be downloaded from the assignment page on Canvas. Your task in this assignment is to implement a simple agent to help avoid the ghosts.

Part of this assignment will be reading other people's code. The pacman game can be started by executing pacman.py, e.g. python pacman.py. Pacman supports a wide variety of arguments, most of which you will not be needing in this assignment. For this assignment you only need to use one argument, the --pacman argument which specifies the name of your agent class. In this case, you will write an agent class called TimidAgent that must be stored in myAgents.py.

It is suggested that you look at one reflex agent that is provided for you, the LeftTurnAgent that is stored in pacmanagents.py. Try running it and look at the code to see what it is doing. It will probably be helpful to use a symbolic debugger to set breakpoints and examine values. Note that in most IDEs, you need to configure the IDE to start the program with your desired options. Here's an example in PyCharm for running LeftTurnAgent:

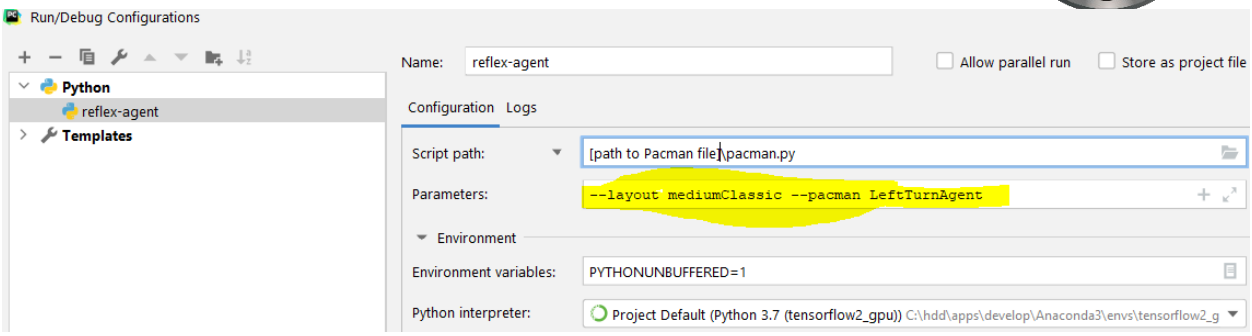


Figure 1 - Configuring command line options in PyCharm. Other IDEs have similar methods.

Note that this execution configuration sets the layout as well. It is not required, but you can play with any of the configurations in the layouts directory. Just for fun, you can specify KeyboardAgent (from keyboardAgents.py) if you want to play the game yourself using the keyboard cursor arrow keys.

TimidAgent should be based on the LeftTurnAgent with one difference. Each time the game invokes nextAction, nextAction will make a call for each ghost to see if the pacman is in danger. Your method TimidAgent.inDanger must have the signature in the skeleton code that is provided for you. It takes two formal arguments and one optional one. The first agent should be an AgentState representing the pacman. The second agent should be an AgentState of one of the ghosts. These can be obtained from the GameState object that will be bound to nextAction's formal argument state. See methods getPacmanState() and getGhostStates() in pacman.GameState which will be helpful for finding the agent states. The last argument, dist, specifies how close ghosts must be to the pacman before it is considered to be in danger and this value defaults to 3 if keyword formal argument is not passed..

We consider the pacman to be in danger when both of the following conditions are met:

1. The ghost and the pacman are in the same row or column.
2. The ghost is within dist units (formal argument to the method) of the pacman.
3. The ghost is not frightened (see the Ghost state for how to check for this).

When the pacman is in danger, inDanger returns the compass direction from the pacman to the ghost (Figure 2). Method inDanger returns the Directions.Stop when the pacman is not in danger.

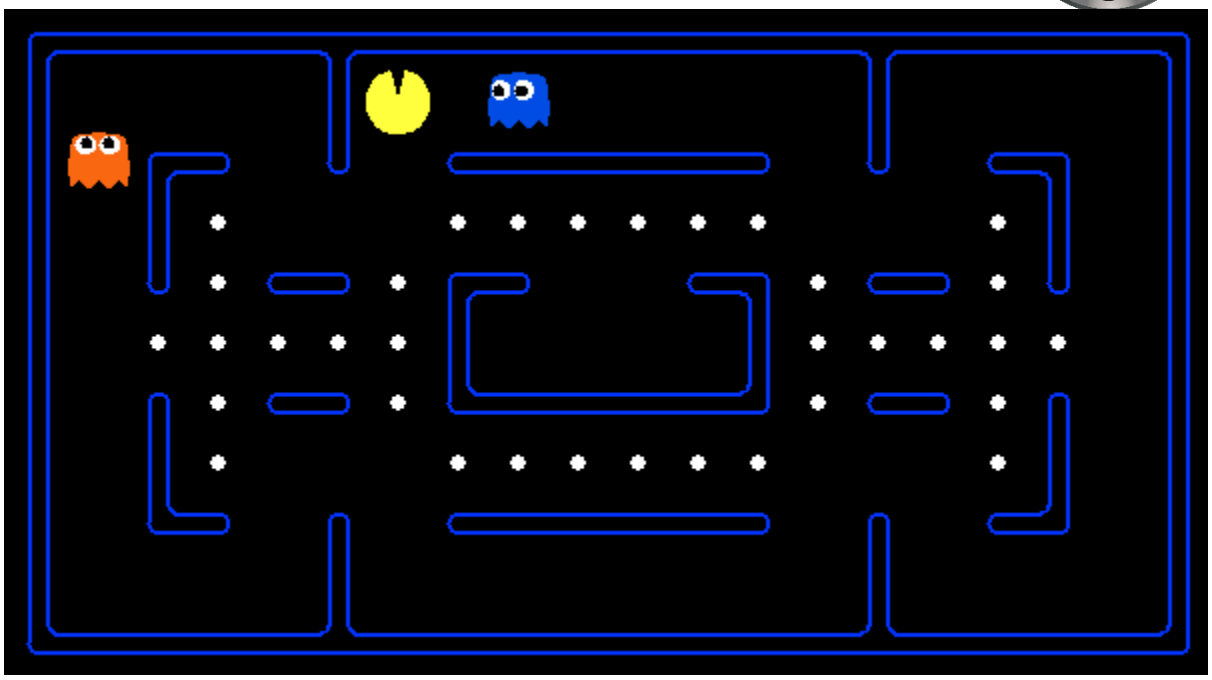


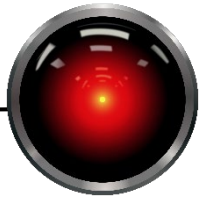
Figure 2 - Pacman is in danger from the East. inDanger() would return Directions.East when invoked with the pacman state and the blue ghost state. If the blue ghost was scared or if the ghost was farther than dist units away, inDanger would return Directions.Stop which indicates that no evasive action needs to be taken.

Method getAction should check for the pacman being in danger from each of the ghosts in the same order as they appear in the list returned by getGhostStates(). The first time the pacman is in danger, a decision is made. Based on the direction from which the pacman is in danger, we select a new direction. We check for legal directions in the following order: reversing the current direction, turning to the left, then turning to the right. If none of these are legal, we continue in the direction of the danger, or stop if no move is legal (only possible in contrived boards). In the example above, the pacman is in danger from the East. Reversing the danger direction is not legal, nor is heading North (left turn from East), but the right turn to the danger direction (South) is possible and TimidAgent should return Directions.South for this case.

When the pacman is not in danger, it should function similarly to the LeftTurnAgent. That is, it turns left whenever possible. If not possible it runs until it can't go any further in the current direction, then tries a right turn or U-turn. If no action is possible, sets the action to Directions.Stop.

There is a lot of code that is designed for scaffolding that you do not need to read. You should concentrate on reading and understanding the following files:

pacman.py – This is the program's entry module. When pacman.py is given as an argument to python, it will start executing code in the "if __name__ == ''__main__'' block. Get a feeling for GameState, there are useful get methods such as getGhostStates. You might want to read some of the other code here as well.



game.py – The Agent, AgentState and Directions classes are relevant to your assignment. Your TimidAgent should be derived from Agent, and both AgentState and Directions are useful for solving the problem.

pacmanAgent.py – Read LeftTurnAgent.

For an example of what I consider to be an appropriate level of commenting, please read LeftTurnAgent. Most of this code was written by others and the code is commented. Not all of this is at the level of commenting that I would like to see. However, experience on very large software projects has led me to see the value in good commenting and I have commented LeftTurnAgent to the level at which I would like to see you comment your code.

What to turn in: Submit myAgents.py to Canvas. Your file must be interpretable by Python, as it will be uploaded to a virtual machine that will check your assignment for correct output. It is critical that you stick to the provided interfaces as these will be called directly to test your code. If you modify the interface, the tests will break and you will not receive credit.

Warning: It is not difficult to make a smarter agent than this one, but you will be graded on TimidAgent being implemented to the specification above. If you are having fun and want to write a smarter agent, write another class (e.g. EinsteinAgent). We will be learning things throughout the semester that can help you make a smarter agent than this one.