

Answer the questions in the spaces provided on the question sheets. If you run out of room for an answer, continue on the back of the page.

Name: \_\_\_\_\_

## Python Concepts

1. We will now review some basic dictionary operations.<sup>1</sup> Dictionaries are a useful data structure for implementing blackboard architectures. The blackboard is an area of memory that any expert (or agent) may use to read from and write to.

(a) Create an empty dictionary called `blackboard`.

(b) Add the key `players` and set the value to 4 to indicate that there are four players.

(c) Print the number of players as known to the blackboard.

(d) Increment the number of players in the game and store it back to the `blackboard`.

2. Write a function `avatarPositionToTile(avatar)`. This function will perform what is essentially the inverse of `tilePositionToPixel(tile)` in that it will return a column and row tuple of the tile when given a pixel position of an avatar.

## Blackboard Architecture

<sup>1</sup><http://docs.python.org/tutorial/datastructures.html>

3. In this question, you will create two **blackboards** for the **boy** and **girl** avatars. Since the **boy** will be the player, his blackboard will be called `player_bb`. The other blackboard will be called `enemy_bb`.

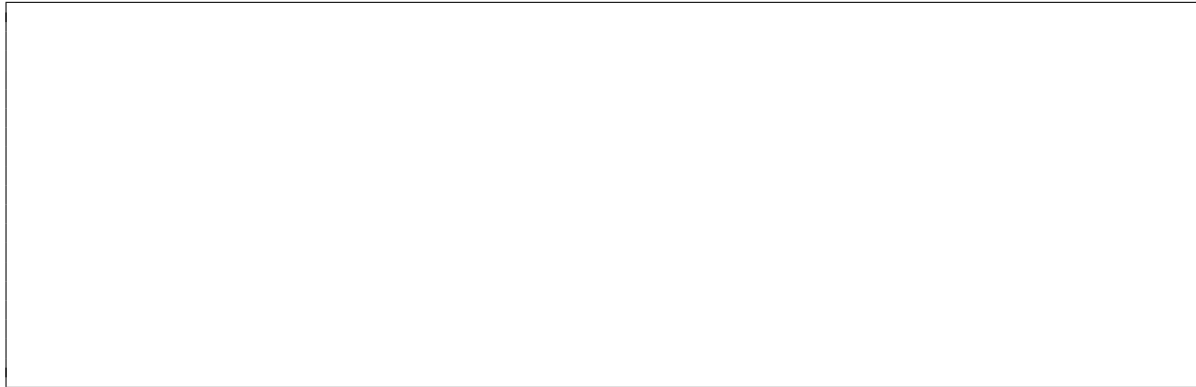
- (a) Initialize the `player_bb` with the properties `health => 100` and `ammo => 20`.

- (b) Initialize the `enemy_bb` with the properties `health => 100`, `ammo => 10`, `time_passed => 0`, `delay => 0`, `action => 'Idle'`, `panic => None`. It may not be obvious at this point what all these different properties do, but we will get to them as the need arises.

- (c) Near the beginning of the game loop, store `time_passed_seconds` in `enemy_bb` (for this, use the key value of `time_passed`). The blackboard location `enemy_bb['delay']` will similarly keep track of the **accumulated** time so far. The purpose of these properties are to intentionally slow down the AI to have more human-like performance.

## Pygame Concepts

4. Add a `KEYDOWN` event for `K_r` so that it makes a call to `randomEvent(level)`. This is a provided function which randomly places a health item and an ammo item on the screen. There can be at most one health item and one ammo item on the board at any given time.
5. Unfortunately, nothing actually appears on the screen when calling this function, since it must be `blit` inside of the game loop. If `items['health']` (and `items['ammo']`) are not `None`, then `blit` the corresponding surface to the screen. Note that the corresponding surfaces are called `health_image` and `ammo_image`. Since the value of the `items` dictionary contains the center position of the object, use this as the position of the item. Don't forget to re-position the item, since `blit` occurs from the top left of the surface, not the center.



## Decision Trees

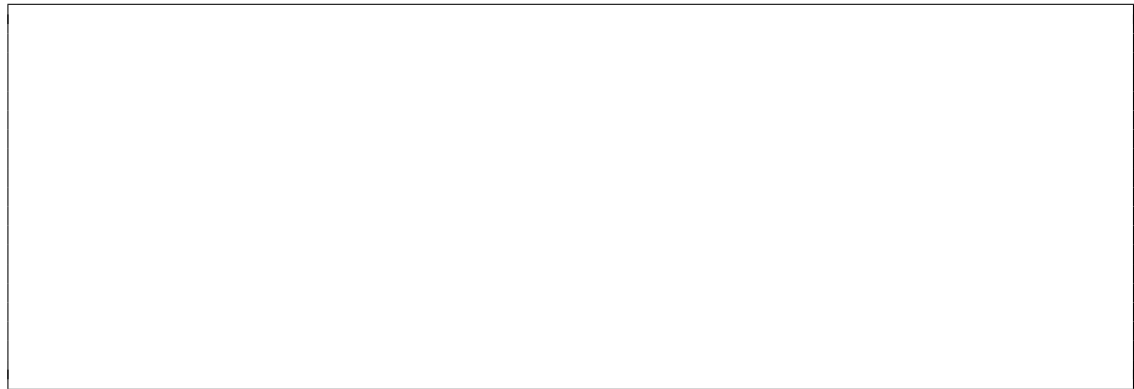
6. In this section, we will implement a hard-coded (conditional-based) decision tree. Such a decision tree implementation can be generalized, and we will do so in a future lecture. For now, assume the game has a `boy` and a `girl` avatar where the girl avatar is an AI.
  - (a) Uncomment the call to the `statusline` in the game loop. This will display the status line containing the health and ammo information of the avatars.
  - (b) Similarly, uncomment the `grabItem` function calls. This code allows the avatar to pick up items.
  - (c) Add a `KEYDOWN` event for `K_SPACE`. When the player presses this key, and if the player has enough ammo, and if the AI is in the same tile as the player, then subtract one unit of ammo from yourself and five units of health from the AI. Test that this functions correctly before continuing.



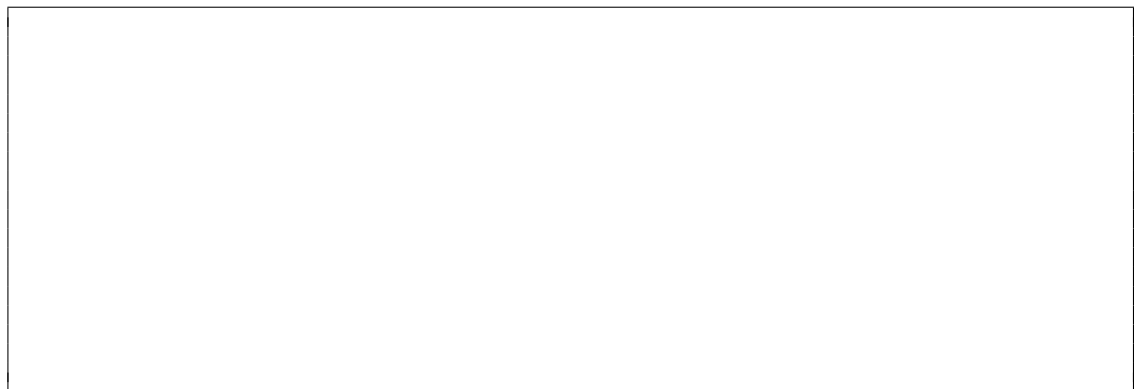
7. You will now draw a decision tree for the enemy AI. Assume that the decision tree has available to it the following actions: `fire`, `seek`, `ammo`, `health`, and `panic`. Your decision tree should represent the following criteria, from highest (root) to lowest (leaf) priority:
  - (a) If the health is less than 20, and if there is a health item on the screen, then perform the `health` action. The `health` action seeks the health item.



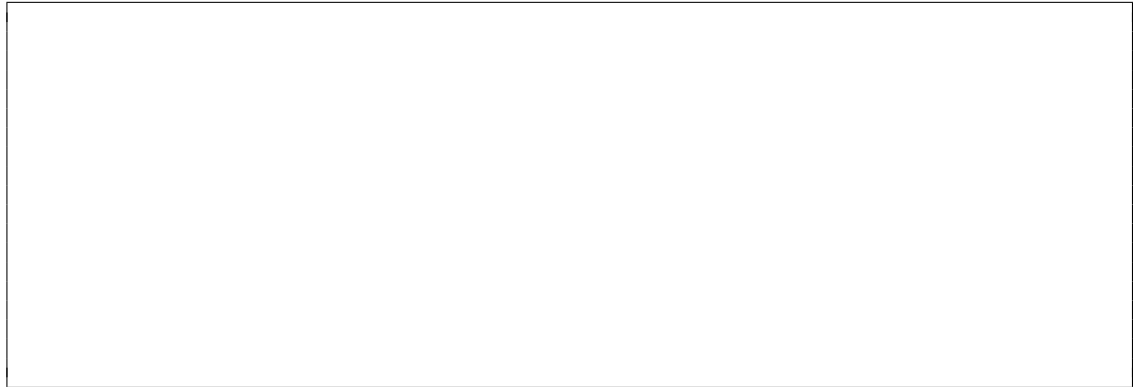
- (b) Otherwise, if the ammo is greater than 0, and the player is in the same tile as the AI, then perform the **fire** action. The **fire** action will fire upon the player. Internally, the fire action throttles so that it does not rapid-fire, but this detail is not present in the decision tree.



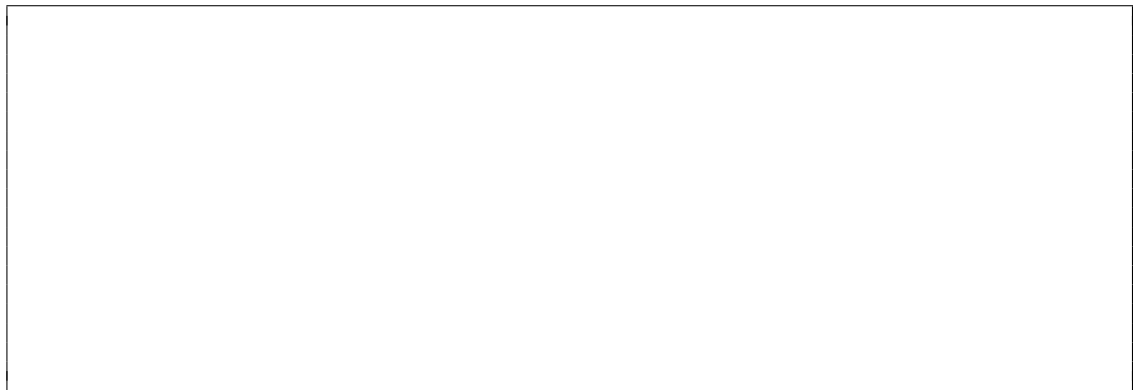
- (c) If the AI is out of ammo, and if an ammo item is available, perform the **ammo** action. This action seeks to pick up the ammo item.



- (d) The AI will perform **panic** when health is less than 20, it is out ammo, and no items are available for pickup. When the AI panics, it will always flee at most two tiles away from the player. It can choose the direction to flee arbitrarily.



- (e) If none of the above conditions are true, then the AI will **seek** the player.



8. Use Python to program the decision tree from the previous question. Your logic should be implemented within the `aiDecisionTree` function.
9. Finally, you can update the status bar state within `aiDecisionTree` with code such as `enemy_bb['action'] = 'Fire'`. This will update the status line to indicate that the AI is firing.

