Python Battle

TINST 310 – Computational Problem Solving

Background:

The purpose of Python Battle is to teach Python programming and algorithmic design at a basic level, while also allowing the programmer to be as creative and advanced in her/his coding as (s)he would like.

The programmer will need to have a very basic understanding of classes (object-oriented programming), some basic data types, as well as a fundamental knowledge of algorithmic design and how to implement them in Python.

The programmer will code a basic to complex artificial intelligence by following a class template and putting logic inside a single “turn()” function that will be called by the main game (see: gameplay). The programmer will have to consider logic and algorithmic design in order to ensure his or her bot will appropriately maneuver the map and battle the opponent bot.

Knowledge required: Classes (basic knowledge of creating one and calling properties), calling functions (no parameters needed), loops (while, for), conditional statements (if), navigating lists & tuples (how indexes work), and creating variables

Gameplay:

Objective:

The objective of Python Battle is to program an artificial intelligence for a battle bot that will defeat another battle bot that is placed against it in a match. The two AI controlled battle bots fight each other on a 10x10 grid by traversing the grid and using attacks against their opponent.

Once the Python Battle game has been launched, the program will ask for the names of the two AIs that will be loaded. The program will then run each player’s AI code one at a time, determining what the bots will do for each turn based upon each player’s individual code. This back and forth turn-based gameplay will continue until one of the two bots is determined the winner of the match.

There are two ways to achieve victory:

1. Attack the other bot until its health reaches or drops below 0.
2. 200 turns (100 each bot) has been reached, and no winner has been determined. At that point the winning bot will be:
   1. The bot with the most health
   2. If health is even, the bot who controls the most squares with its respective color. Squares are captured once a bot moves over it. If a bot moves over a square captured by the other bot, the first bot takes over the second bot’s square.

**Bots:**

There are two bots; one red, and one blue. Each bot will be controlled via a player’s AI code.



Each bot starts with three important characteristics:

1. **100 Health** – Each bot needs health to be able to continue the match and starts each match with 100 health. Once a bot’s health falls to 0 or below, that bot has officially lost the match. Health can be replenished via a repair ability (see: abilities).
2. **20 Energy** – Each bot starts the match with 20 energy and reaches a maximum of 100 at any given point throughout the match. Energy is gained through various abilities, and is depleted while accessing several “special” abilities. All of this is discussed further in the abilities section.
3. **5 Teleport Charges** – Each bot has a teleporter built in, but they only have limited use per match. Two special abilities make use of the teleporter, but require teleport charges in order to function. Once the five charges are depleted, the teleport abilities can no longer be used.

Map and Movement:

Each match of Python Battle is played out on a 10x10 grid. There are several maps that vary with “wall” placement throughout the grid. Walls are any space on the grid covered with a purple border and UW logo on the inside. Bots are not able to traverse through any square that is a wall. Logic is being written to place a random number of walls on random positions on the map.

When the game is launched, each bot is positioned randomly on the map facing a random direction. It is up to the programmer to ensure his or her AI can traverse the grid properly.

Each bot has the ability to move forward, move backward, turn left, or turn right. Each of these actions results in the ending of the turn for the bot. In addition to movement functions, each bot is allowed to freely call functions allowing the bot to “check” any space they’d like. Typically, this will either be checking the spot in front of the bot, or getting the enemy’s location. It is important to note that the functions that check the status of a space on the grid do not result in the loss of turn for the bot, as they are integral functions for a bot to effectively navigate the grid. Lastly, each bot object has two properties that are important in determining the bot’s current placement on the grid. One contains the current grid space, the other the current direction the bot is facing.

Code:

The following is all essential code needed to properly program an effective AI within Python Battle:

AI Class:

When writing AI code for a battle bot code, it is important to understand how the code will be called by the overall program.

A programmer must first create a class called AI(). The Python Battle game creates an overall robot object that inputs a player’s AI class as one of its properties. Second, every AI class must have a turn() function that contains all logic that dictates the bot’s actions. Other than that, all other self-created functions, properties, etc. are up to the programmer.

No properties are required for the \_\_init\_\_ Python function when creating the class, but properties are highly recommended for keeping track of various aspect of the game. An example / suggestion: each bot only is allowed five teleports per match and any attempts after five will waste a bot’s turn; keeping track of how many teleports a bot has used (or has left) would be extremely beneficial.

Some sample code to get a player / programmer started:

1. **class** AI:
2. **def** \_\_init\_\_(self):
3. **pass**
4. **def** turn(self):
5. # AI logic goes here

Calling Functions and Properties:

It is important to know how to properly call all functions and properties. As mentioned previously, the AI class will be used when creating an overall “robot” object. The AI class is one property of the robot class along with several others, as well as all function. All properties and functions of importance will be mentioned later in this section.

Due to the above AI class being a property of an overarching class, we must call the robot object prior to any function or property as part of that class, i.e. self.robot.property or self.robot.function()

Movement:

These functions and properties are all related to moving a battle bot along the playing grid:

Properties:

***Calling a property does not use a turn.***

Position:

This is a direct property of the robot class and contains a bot’s current location on the grid. This property is stored as a tuple and returns two numbers; the first is the location along the x axis, and the second is location along the y axis. For example, (7, 4) would indicate the location that is the 7th space from the left and the 4th space from the top.

**Example:**

1. position = self.robot.position
2. **print**(position)
4. Output:
5. (7, 4)

Rotation:

This is a direct property of the robot class and contains a bot’s current rotation (the direction the bot is facing) on the grid. This property is stored as an integer and contains a value of 0, 1, 2, or 3. 0 indicates the bot is facing upwards, 1 to the right, 2 facing downward, and 3 to the left.

**Example:**

1. rotation = self.robot.rotation
2. **print**(rotation)
4. Output:
5. 1
6. # This would indicate the bot is facing right

Health:

This is a direct property of the robot class and contains a bot’s current health. If this number falls to 0 or below, the game is over and the opponent has won the match. This number will always be between 1 and 100 when called. Each bot always starts the match with 100 health.

**Example:**

1. health = self.robot.health
2. **print**(health)
4. Output:
5. 100

Energy:

This is a direct property of the robot class and contains a bot’s current energy. Energy is used when calling the special ability functions (discussed later). This number will always be between 0 and 100 when called. Each bot begins each match with 20 energy.

**Example:**

1. energy = self.robot.energy
2. **print**(energy)
4. Output:
5. 100

Functions:

lookInFront():

This function checks the space immediately in front of the bot and returns one of three values – “wall”, “bot”, or “clear” depending on what is in the space. This function is imperative for the AI to work properly. For instance, a bot’s AI would need to know if the opponent is in front of them before either attacking or attempting to move into that space.

***This function does not use a turn.***

**Example:**

1. check\_space = self.robot.lookInFront()
2. **print**(check\_space)
4. Output:
5. "clear"

lookAtSpace(space):

This function checks a particular space on the grid that is passed into the function. In order to work properly, a tuple of coordinates must be passed into the function. Just like the lookInFront() function, lookAtSpace() returns “wall”, “bot”, or “clear.” This function might be useful for advanced navigation AI.

***This function does not use a turn.***

**Example:**

1. space = (3, 3)
2. check\_space = self.robot.lookAtSpace(space):
3. **print**(check\_space)
5. Output:
6. "wall"

locateEnemy():

This function locates the enemy bot’s position and rotation, and returns those values as a tuple. The first position value is a tuple in itself, and the second rotation value is an integer. This function is essential for trying to hunt down an enemy.

***This function does not use a turn.***

**Example:**

1. enemy\_location = self.robot.locateEnemy()
2. **print**(enemy\_location)
3. **print**(enemy\_location[0])
4. **print**(enemy\_location[1])
6. Output:
7. # Full tuple returned
8. ((6,4), 3)
9. # First item in the tuple -- a tuple with enemy coordinates
10. (6, 4)
11. # Second item in the tuple, the enemy's rotation value
12. 3

goForth():

This function moves the bot one space directly in front of the direction the bot is facing if that space is clear, e.g. a bot on (1, 1) facing right will move to (2, 1). If the space in front of the bot is not a “clear” space, the function call will still constitute as a turn but no action will be performed.

***This function uses a turn and will give the bot +10 energy up to 100 if below 100.***

**Example:**

1. self.robot.goForth()

goBack():

This function moves the bot one space directly behind the bot if that space is clear, e.g. a bot on (5, 5) facing right will move to (4, 5). If the space behind the bot is not a “clear” space, the function call will still constitute as a turn but no action will be performed.

***This function uses a turn and will give the bot +10 energy up to 100 if below 100.***

**Example:**

1. self.robot.goBack()

turnLeft():

This function rotates the bot 90 degrees counter-clockwise so that the bot rotates to the left. Each time the function is called, it is considered a turn so a maximum of 90 degrees of rotation can only be performed per turn, i.e. a 180 degree rotation cannot be performed in a single turn.

***This function uses a turn and will give the bot +10 energy up to 100 if below 100.***

**Example:**

1. self.robot.turnLeft()

turnRight():

This function rotates the bot 90 degrees clockwise so that the bot rotates to the right. Each time the function is called, it is considered a turn so a maximum of 90 degrees of rotation can only be performed per turn, i.e. a 180 degree rotation cannot be performed in a single turn.

***This function uses a turn and will give the bot +10 energy up to 100 if below 100.***

**Example:**

1. self.robot.turnRight()

Non-Movement Functions:

Attack():

This is a basic attack function. If called when the opponent is in the space directly in front of the bot, it will cause 10 damage to the opponent reducing the opponent’s health by 10. This ability can always be used and down not require energy.

***This function uses a turn, reduces the opponent’s health by 10 if they are in front of the bot, and will give the bot +10 energy up to 100 if below 100.***

**Example:**

1. self.robot.attack()

charge():

This function ends a bots turn without performing any other action that would otherwise end a bot’s turn. For staying in one spot, the bot “charges” and gains a bonus 10 energy on top of the normal 10 energy given for a total of 20 energy added to the bot’s total for that turn.

***This function uses a turn, and will give the bot +20 energy up to 100 if below 100.***

**Example:**

1. self.robot.charge()

repair():

This function ends a bots turn without performing any other action that would otherwise end a bot’s turn. The bot stays in one spot and “repairs” 10 units of health at the expense of 20 energy.

***Calling this function will always end a bot’s turn, but it will only be successful if the bot has enough energy to perform the action.***

***This function uses a turn, and will give the bot +10 health up to 100 if below 100 if successful. It will use 20 energy if 20 energy is available.***

**Example:**

1. self.robot.repair()

escape():

This function teleports a bot to a random location on the grid facing a random direction. This function is useful if the bot is very low on health and is about to lose the match. This ability costs 30 energy to use. This function also uses a “teleport” token and a bot must have one available in order to use this ability.

***Calling this function will always end a bot’s turn, but it will only be successful if the bot has enough energy to perform the action as well as a teleport token.***

***This function uses a turn, and the bot will be teleported to a random location if successful. It will use 30 energy and a teleport token if both are available.***

**Example:**

1. self.robot.escape()

teleport():

This function teleports a bot directly behind the opponent. Teleport is especially useful for AIs that simply try to hunt down and follow its opponent, as two turns are required to rotate 180 degrees. This ability costs 80 energy to use. This function also uses a “teleport” token and a bot must have one available in order to use this ability.

***Calling this function will always end a bot’s turn, but it will only be successful if the bot has enough energy to perform the action as well as a teleport token.***

***This function uses a turn, and the bot will be teleported directly behind its opponent if successful. It will use 80 energy and a teleport token if both are available.***

**Example:**

1. self.robot.teleport()

toss():

This function attacks the space directly in front of the bot. If the opponent is in that space, this attack will hit the opponent, inflicting 20 damage, and “tossing” the opponent to a random location on the grid facing a random direction. This ability requires 70 energy to perform.

***Calling this function will always end a bot’s turn, but it will only be successful if the bot has enough energy to perform the action as well as the opponent being in the space directly in front of the bot.***

***This function uses a turn, and inflicts 20 damage on the opponent as well as tosses them to a random location on the grid if the attack is successful. It will use 70 energy if 70 energy is available.***

**Example:**

1. self.robot.toss()

crush():

This function attacks the space directly in front of the bot. If the opponent is in that space, this attack will hit the opponent, inflicting between 15 and 25 damage at random. This ability requires 40 energy to perform.

***Calling this function will always end a bot’s turn, but it will only be successful if the bot has enough energy to perform the action as well as the opponent being in the space directly in front of the bot.***

***This function uses a turn, and inflicts 15-25 damage on the opponent if the attack is successful. It will use 40 energy if 40 energy is available.***

**Example:**

1. self.robot.crush()

luckyshot():

This function attempts to attack the opponent from anywhere on the map. If successful, the attack will damage the opponent anywhere between 5 and 30 damage. However, this attack also has the chance to “backfire,” damaging the bot making the attack instead of its opponent. Each bot’s chance of backfire starts at 0% and increases anywhere between 1-20% per shot. It is important to note that this increase happens ***BEFORE*** taking your first shot. If a shot backfires rather than hits an opponent, it will damage the bot anywhere between 5 and 15 health. Luckyshot requires 50 energy to use.

***Calling this function will always end a bot’s turn, but it will only be successful if the bot has enough energy to perform the action.***

***This function uses a turn, and will either inflict 5-30 damage on an opponent or “backfire” and cause 5-15 damage to the bot making the attack. The chance of backfire increases with each use. There is no way to know a bot’s current backfire chance percentage.***

**Example:**

1. self.robot.luckyshot()