SUPERMARKET AGENT SETUP

Authors: Daniel Kasenberg, Teah Markstone, and Matthias Scheutz

Synopsis: Set up an autonomous agent connected to the supermarket simulation environment that can perform the shopping task.

Download the following dependencies:

- DIARC.jar (from Canvas)
- Agent.java (from Canvas)

Running the simulation (with keyboard input or with Java agents)

To run the simulation with autonomous agents controlling the avatar, you first need to start the socket listener in one terminal:

<python-command> socket_env.py

Put the java class file together with the DIARC.jar in the same directory and compile the agent file using:

javac -cp DIARC.jar Agent.java

Then you can run using the agent using

java -cp .:DIARC.jar Agent Agent

(Note that you need “Agent” twice! Also, use “;” instead of “:” and put the paths in double quotes on Windows machines!)

This agent will print out info about the game and the spin indefinitely until you stop it with Ctrl+C.

Actions and environment dynamics

Below is a table of the actions available in the supermarket environment (left column), again the corresponding key to press in keyboard input to perform the action (middle column) and action names to be used in the Java code for your agent:

Action	**Input Key (main.py)**	**Java action**
North	Up arrow	goNorth()
South	Down arrow	goSouth()
West	Left arrow	goWest()
East	Right arrow	goEast()
No-op (do nothing)	<none>	nop()
Interact with object	Enter/Return	interactWithObject()
Toggle holding cart	‘c’	toggleShoppingCart()
Cancel interaction	‘b’	cancelInteraction()
View shopping list	‘l’	(None: observation includes complete shopping list)
View current inventory	‘i’	(None: observation includes all necessary information)
Get last state observation	(none)	getLastObservation()

General facts

- Objects are solid; you can't walk through walls, shelves, or shopping carts
- When holding a shopping cart, that cart is an extension of you for collision purposes; the cart won't pass through solid objects either
- Shopping lists are randomly generated when the game initializes, and have anywhere between 1 and 12 items
- Shopping carts can each carry a maximum of 12 food items
- Food items must be picked up and placed in the cart one at a time. If you need three apples, you must do the following three times: go to the shelf, pick up an apple, go to the cart, put the apple in the cart.
- The player can leave their shopping cart anywhere in the store, and can even have multiple shopping carts: any item in any of your carts counts towards your inventory
- You can only *hold* one shopping cart at a time
- Putting food back on the wrong shelf does not mean you'll pick up the wrong food when you interact with that shelf later (if you were in a grocery store in the canned foods aisle looking for soup and saw 15 cans of soup and one apple, would you pick up the apple?)
- You can't leave a cart above or below one checkout, and then expect your food to be purchased when you interact with the other checkout. This should more or less go without saying.
- In the keyboard input mode, you can exit the game with the ESC key, or by leaving through the exit door. You can also Ctrl+C out of the Python program or "X out" of the window.
- In autonomous agent mode, you can exit the game by leaving through the exit door, Ctrl+C in your terminal, or "X out" of the window.

Observations

In autonomous agent mode, you will need to access the state observations which are contained in the the Observation class (see below):

- The observation keeps track of the states of the individual objects: **players**, **carts**, **shelves**, **registers**, **cart returns**, **counters**, and **checkouts**. There's an array for each class of object.
    - All objects have a position (a 2D array x and y). Objects other than carts and players also have a width and height. Be careful in how you interpret this; just because a person is colliding with an object doesn't mean that object.x < player.x < object.x + object.width, etc. Take a look at the collision() and canInteract() methods for various object classes to get a sense of how this actually works.
    - Shelves and counters have a particular food associated with them, represented as a string ("apples", etc.)
    - Players and carts are facing a particular direction (0=north, 1=south, 2=east, 3=west).
    - Players may be holding a particular food (field "holding_food", will be null if not holding any food).
    - Players have a shopping list shopping_list of food strings (matching those on particular shelves). There's a corresponding list list_quant of the number of each type of food on the list. For example, if player.shopping_list[i].equals("apples"), then player.list_quant[i] is the number of apples on the shopping list.
    - The Player's curr_cart is the index in observation.carts of the cart the player is holding, or -1 if the player is not holding a cart.
    - Each cart has an owner (who first picked the cart up from the cart return) and a separate variable last_held (who last touched the cart). This probably won't matter in single-agent settings, but would matter in multi-agent games.
    - Each cart has a capacity. This'll just be 12.
    - Each cart has variables contents and contents_quant (same format as a player's shopping list, but for the unpurchased contents of a cart) and purchased_contents and purchased_quant (same format, but for the *purchased* contents of a cart).
- The observation contains a little information about the "interactive stage" of the game, which can presumably help you remember to call interactWithObject() an appropriate number of times:
    - If the user is not currently interacting with an object (i.e., if no interaction message is on the screen), interactive_stage=-1 and total_stages=0.

    - Otherwise, the user is interacting with some object; interactive_stage is the (zero-indexed) current stage, and total_stages will be 1 or 2 depending on if the interaction in question is a zero-stage interaction.
- The observation has some helper methods that may come in handy. You can rely on the individual objects' collision() and canInteract() methods (they're identical to those in the python simulation); be a bit more wary of the other ones. The helper methods are still very much under active development.

```
public class Observation {
    public Player[] players;
    public Cart[] carts;
    public Shelf[] shelves;
    public Counter[] counters;
    public Register[] registers;
    public CartReturn[] cartReturns;
    int interactive_stage;
    int total_stages;

    public static boolean defaultCollision(InteractiveObject obj, double x, double y, double xMargin, double
       yMargin) {
        // Objects are usually a bit bigger than their official width/height estimates
        return obj.position[0] - xMargin < x && x < obj.position[0] + obj.width + xMargin &&
               obj.position[1] - yMargin < y && y < obj.position[1] + obj.height + yMargin;
    }

    public static boolean defaultCollision(InteractiveObject obj, double x, double y) {
        return defaultCollision(obj, x, y, 0.55, 0.55);
    }


    public static boolean defaultCanInteract(InteractiveObject obj, Player player, double range) {
        double x = player.position[0];
        double y = player.position[1];
        switch(player.direction) {
            case 0: // Player is facing north, a spot just above the player collides with the object. etc.
                return obj.collision(x, y-range);
            case 1: // South
                return obj.collision(x, y+range);
            case 2: // East
                return obj.collision(x + range, y);
```

```
        case 3: // West
            return obj.collision(x - range, y);
    }
    return false;
}
public static boolean defaultCanInteract(InteractiveObject obj, Player player) {
    return defaultCanInteract(obj, player, 0.5);
}

public abstract class InteractiveObject {
    public double width;
    public double height;

    public double[] position;

    public abstract boolean collision(double x, double y);
    public abstract boolean canInteract(Player player);
}

public class Shelf extends InteractiveObject {
    public String food;

@Override
public boolean collision(double x, double y) {
    return defaultCollision(this, x, y, 0.75, 0.55);
}

@Override
public boolean canInteract(Player player) {
        return player.direction <= 1 && defaultCanInteract(this, player);
    }
}
```

```
public class Register extends InteractiveObject {
    @Override
    public boolean collision(double x, double y) {
        return defaultCollision(this, x, y);
    }

    @Override
    public boolean canInteract(Player player) {
        return defaultCanInteract(this, player);
    }
}

public class Counter extends InteractiveObject {
    public String food;

    @Override
    public boolean collision(double x, double y) {
        return defaultCollision(this, x, y);
    }



    @Override
    public boolean canInteract(Player player) {
        return defaultCanInteract(this, player);
    }
}

public class CartReturn extends InteractiveObject {
    @Override
    public boolean collision(double x, double y) {
        return defaultCollision(this, x, y, 0.7, 0.55);
```

```
    }

    @Override
    public boolean canInteract(Player player) {
        return player.direction == 1 && defaultCanInteract(this, player);
    }
}

public class Player {
    public int index;
    public double[] position;
    public int direction; // NORTH is 0, SOUTH is 1, EAST is 2, WEST is 3

    // This is the *index* of the current cart in the carts array.
    public int curr_cart;
    public String[] shopping_list;
    public int[] list_quant;
    public String holding_food;
}



public class Cart extends InteractiveObject {
    public int direction;
    public int owner;
    public int last_held;
    public int capacity;

    public String[] contents;
    public int[] contents_quant;
    public String[] purchased_contents;
    public int[] purchased_quant;
```

```
    @Override
    public boolean collision(double x, double y) {
    // Collisions between players and carts are weird, and don't follow the bounding-box rules.
    // This is because the different directions of sprites are different sizes, etc, and
    //following the rules would make them look bad.
        switch(this.direction) {
            case 0: // North
                return this.position[0]-0.5 <= x && x <= this.position[0] + 0.5
                    && this.position[1]- 1.6 <= y && y <= this.position[1] - 0.2;
            case 1: // South
                return this.position[0]-0.5 <= x && x <= this.position[0] + 0.5 &&
                        this.position[1] +0.2 <= y && y <= this.position[1] + 1.6;
            case 2: // East
                return this.position[1]-0.5 <= y && y <= this.position[1] + 0.5
                        && this.position[0] +0.2 <= x && x <= this.position[0] + 1.3;
            case 3:
                return this.position[1]-0.5 <= y && y <= this.position[1] + 0.5
                        && this.position[0] - 1.3 <= x && x <= this.position[0] - 0.2;
        }
        return false;
    }
    @Override
    public boolean canInteract(Player player) {
        return defaultCanInteract(this, player);
    }
}

public boolean northOfCartReturn(int playerIndex) {
    return this.players[playerIndex].position[1] < 17.5;
}
```

```
public boolean southOfCartReturn(int playerIndex) {
    return this.players[playerIndex].position[1] > 17.8;

}

public boolean atCartReturn(int playerIndex) {
    double x = this.players[playerIndex].position[0];
    double y = this.players[playerIndex].position[1];
    return y >= 17.5 && y <= 17.8 && x >= 1.25 && x <= 2.25;
}

public boolean inAisleHub(int playerIndex) {
    return this.players[playerIndex].position[0] >= 3.5
            && this.players[playerIndex].position[0] <= 4.5;
}

public boolean inRearAisleHub(int playerIndex) {
    return this.players[playerIndex].position[0] >= 15.5
            && this.players[playerIndex].position[0] <= 16.5;
}

public boolean besideCounters(int playerIndex) {
    return this.players[playerIndex].position[0] >= 17.5;
}

public boolean inAisle(int playerIndex, int aisleIndex) {
    double x = this.players[playerIndex].position[0];
    double y = this.players[playerIndex].position[1];
    double aisleLeft = 5.5;
    double aisleRight = 15.5;
    double aisleTop = 0.5 + 4.*(aisleIndex - 1);
    double aisleBottom = 0.5 + 4.*aisleIndex;
```

```
        return y >= aisleTop && y <= aisleBottom
                && x >= aisleLeft && x <= aisleRight;
    }

    public boolean belowAisle(int playerIndex, int aisleIndex) {
        double y = this.players[playerIndex].position[1];
        double aisleBottom = 0.5 + 4.*aisleIndex - 1.5;
        return y >= aisleBottom;
    }


    public boolean aboveAisle(int playerIndex, int aisleIndex) {
        double y = this.players[playerIndex].position[1];
        double aisleTop = 0.5 + 4.*aisleIndex - 2;
        return y <= aisleTop;
    }

    public boolean westOf(Player player, InteractiveObject obj) {
        return player.position[0] <= obj.position[0];
    }

    public boolean eastOf(Player player, InteractiveObject obj) {
        return player.position[0] >= obj.position[0] + obj.width;
    }
}
```