

Lab 8: 6.009 Zoo

Table of Contents

- 1) Preparation
- 2) Introduction
- 3) Technical Overview
 - 3.1) Debugging with the Web UI
 - 3.2) Grid System
 - 3.3) Formations
 - 3.4) Game State
- 4) Game Setup
 - 4.1) Serialization and Deserialization
- 5) Animals
 - 5.1) The Path
- 6) Zookeepers
 - 6.1) Placement
 - 6.2) Throwing Food
- 7) Code Submission
- 8) Checkoff

1) Preparation

This lab assumes you have Python 3.5 or later installed on your machine.

The following file contains code and other resources as a starting point for this lab: [lab8.zip](#)

Most of your changes should be made to `lab.py`, which you will submit at the end of this lab. Importantly, you should not add any imports to the file.

This lab is worth a total of 4 points. Your score for the lab is based on:

- correctly answering the questions on this page (0.5 points)
- passing the test cases from `test.py` under the time limit (1.5 points), and
- a brief "checkoff" conversation with a staff member to discuss your code (2 points).

For this lab, you will only receive credit for a test case if it runs to completion under the time limit on the server.

The questions on this page (including your code submission) are due at 4pm on Friday, Apr 26.

2) Introduction

After a successful stint at MIT, your skills are in global demand! In particular, the world-renowned 6.009 Zoo is in dire need of a manager to help feed its increasingly hungry animal population. They reach out to you and, being the animal lover that you are, you accept. Since you are an avid student of [Fordism](#), you come up with an [assembly line](#) process for feeding all of the animals in an efficient and speedy manner. In this lab and the next one, we will build a [tower-defense game](#) to model this problem! Here is a picture of what the complete game will look like:



Your game will be played on a two-dimensional grid. On the edges of this grid there will be a start where animals will appear and an end where they will disappear. The animals will move along some defined path connecting these two coordinates. Your job is to feed the animals before they reach the end by placing zookeepers along the path. Zookeepers cost money, which players earn by feeding animals. If an animal gets close enough to a zookeeper, the zookeeper will toss food at the animal, thus feeding it.

To get an idea of how similar tower-defense games work, feel free to play one of the many tower-defense games available online! Tower-defense games are a broad genre, so there are many options. The staff is fond of [Bloons Tower Defense](#).

While a game is hard to describe in full detail, we have restricted our tests to exactly the aspects described on this page. Beyond that, you may make your own design decisions and should be prepared to discuss them during your checkoff. We have presented information in the order which we think maps most intuitively to implementation order, but you need not follow this order. If you find yourself overwhelmed by the many aspects of the game, attempt to break them down into pieces to create a modular design. Have fun!

3) Technical Overview

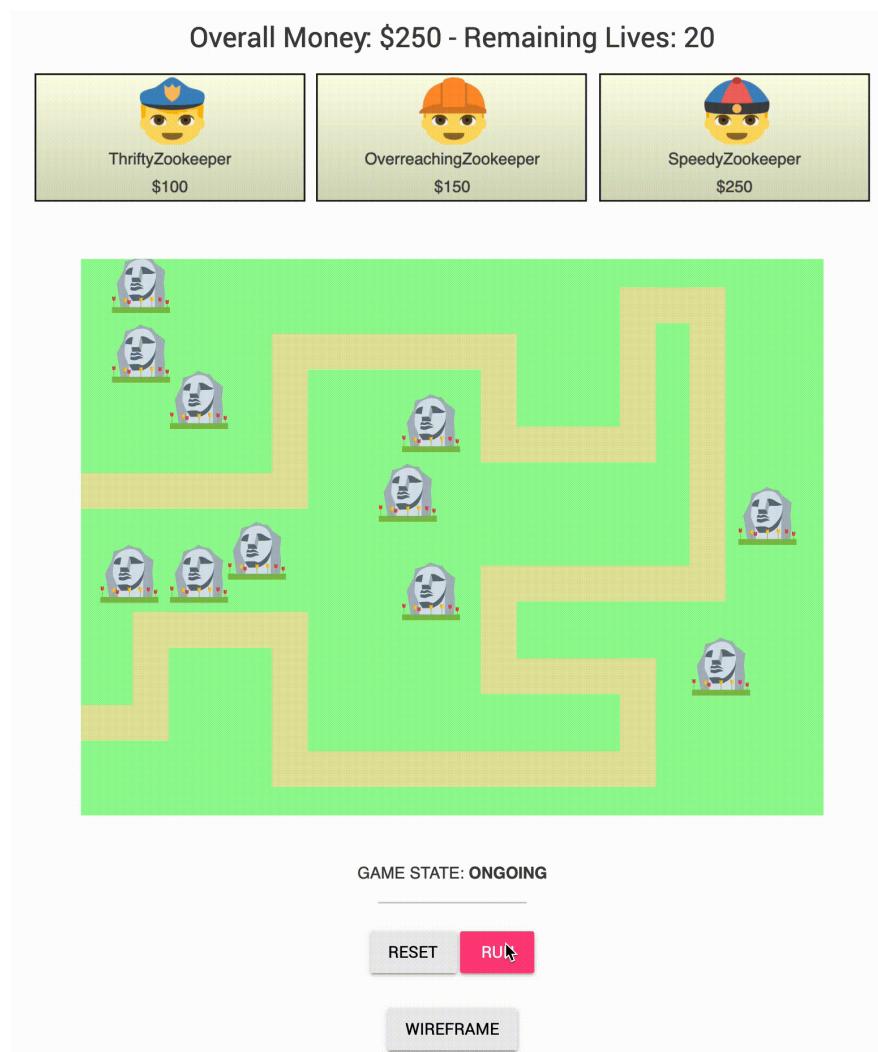
3.1) Debugging with the Web UI

As you get started, remember that our web user interface (UI) is a powerful debugging ally. To use it, run `python3 server.py` and use your web browser to navigate to <localhost:8000>. You will need to restart `server.py` in order to reload your code if you make changes. Note that the gif below is displaying a simulation in a slowed-down version.

When the server is first started, it calls your render function exactly once. This call to render occurs before any timesteps take place. The UI will display the results of this render. If you wish to begin the server, you can press the "Run" button located below the grid. This calls your timestep function, followed by another render to update the screen. As long as "Run" is selected, the server will continuously call timestep and render in sequence (as fast as it can).

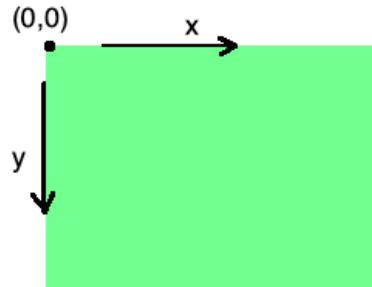
Feel free to edit `lab8/resources/maps/design-your-own-zoo.json` (or create more similar `.json` files) for more fun. By selecting `design-your-own-zoo.json` as your map on the UI, you will be able to visualize your own zoo. Other files in the `maps` directory are used for testing, so you shouldn't modify values in those.

If you run into issues where your code does not seem to be matching with the server, we recommend clearing your browser's cache or opening the UI in a new incognito or private session.



3.2) Grid System

The game world is a two-dimensional coordinate plane. We consider the top-left corner of the grid as $(0, 0)$, with x s increasing towards the right and y s increasing towards the bottom.



3.3) Formations

The game will be structured using blocks known as *formations*. A formation can represent any object in the game with a position, size, and texture. Textures determine how the formation looks when rendered in the UI. They are string codes like defined in the `Constants.TEXTURES`¹ dictionary of `lab.py`. Here are renderings of some of the textures we will use:



This shows one of the zookeepers, a rock, food, and an animal.

Some formations might be moving formations. Those may have both an x -directional and a y -directional velocity, in units of distance traversed per timestep.

Throughout the game, collision detection will be based on the overlap between these rectangular formations. One formation intersects (i.e. collides) with another if any part of it overlaps with the other. Shared edges and corners do not count as intersections.

Note that formations may leave the board during the game. We say a formation has left the board if its center is no longer on the board. Coordinates on edges and corners of the board (such as $(0, 0)$) are still on the board.

Hint: Consider making a Formation class.

3.4) Game State

A Game is made up of three parts: a collection of formations, the zoo's money balance, and a global clock. Every time the clock ticks, the state of the Game is updated by appropriately modifying the money balance and all formations. This makes it simple to describe the Game as a class with three methods:

- `__init__(self, game_info)` initializes the game. `game_info` is a dictionary which contains the following necessary information for generating the game grid:
 - 'width': The integer width of the game grid.
 - 'height': The integer height of the game grid.
 - 'rocks': The set of tuples of rock coordinates. The coordinates are of the centers of the rocks, which have dimensions `Constants.ROCK_WIDTH` and `Constants.ROCK_HEIGHT`.
 - 'path_corners': An ordered list of coordinate tuples. The first coordinate is the start of the path, the last coordinate is the end (both of which lie on the edges of the gameboard), and the other coordinates are corners on the path. Recall that "the path" refers to the unique path that animals follow as they traverse the board.
 - 'money': The player's money balance, in dollars.
 - 'spawn_interval': The interval (in timesteps) for spawning animals to the game. If the first call to timestep has count 0 and the spawn interval is I, an animal is spawned at timesteps with count 0, I, 2*I, ...
 - 'animal_speed': The magnitude of the speed at which the animals move along the path, in units of distance traversed per timestep.
 - 'num_allowed_unfed': The number of animals allowed to finish the path unfed without the player being defeated.
- `timestep(self, mouse)` advances the game state by one time step based on player input `mouse`. In other words, `timestep` is responsible for simulating the evolution of the world across one unit of time. In each timestep, **if the game is ongoing**, the following changes or updates will occur, **in the listed order**:
 1. Compute any changes in formation locations, and remove any off-board formations.
 2. Handle any food-animal collisions, and remove the fed animals and the eaten food.
 3. Throw new food if possible.

- 4. Spawn a new animal from the path's start if needed.
- 5. Handle mouse input, which is the integer tuple coordinate of a player's click, the string label of a particular zookeeper type, or None .
- 6. Redeem one dollar per animal fed this timestep.
- 7. Check for the losing condition.
- `render(self)` tells our UI the state of the game by returning a dictionary of information (described in [Section 4.1](#)).

These three methods are needed for the UI to display the game and are the only code directly called by the test cases. Of course, your solution should make use of many additional functions and classes (our solution has about 40 methods!). Doing so will simplify Lab 9, in which you'll extend your game. We expect to see large variation among solutions, which exemplifies the flexibility of data abstraction when programming larger systems.

See the later sections of this write-up for more detail on the steps that we have only sketched so far.

4) Game Setup

4.1) Serialization and Deserialization

Before we dive into game details, we need to understand how the Game's state is communicated with the UI. The code skeleton that we have provided you is fairly minimal in order to provide flexibility. Therefore, there needs to be some standardized way of digesting the game's state. This logic is handled via the `render` function. In short, `render` outputs the game's state in a way that can be reconstructed later. This process is known as [serialization](#). Conversely, the process of taking this normalized value and breaking it down into meaningful information is known as [deserialization](#).

Your first task will be twofold:

1. [Deserialize](#) `game_info` in Game's `__init__` and store all useful properties.
2. [Serialize](#) the state of Game in `render`.

More specifically, the UI will expect `render` to return a dictionary with the following entries:

- 'formations': A list of dictionaries in any order, each one representing a formation. Each dictionary is of the form `{'loc': (x, y), 'texture': texture, 'size': (width, height)}` where `(x, y)` is the center coordinate of the formation, `texture` is its texture, and `width` and `height` are its dimensions. The list should contain the formations of all animals, zookeepers, rocks, and food (but not the path or game board).
- 'money': The amount of money, in dollars, the player has available to hire zookeepers.
- 'status': The current state of the game which can be '`ongoing`' or '`defeat`'. It is '`ongoing`' initially.
- 'num_allowed_remaining': The number of animals which are still allowed to exit the board before the game status is '`defeat`'. (This number is less than or equal to the initial cap of `game_info['num_allowed_unfed']`.)

5) Animals

An animal is spawned (generated) on the first timestep. The new animal's center aligns with the path's start coordinate center. Another animal should be spawned on every `game_info['spawn_interval']`th timestep thereafter. An animal makes its way along the path with speed `game_info['animal_speed']`. Conveniently, since they start on an integer coordinate and travel with integer speed on non-diagonal path segments, animals will always have integer coordinate positions.

If an animal's center is no longer on the board or is fed by one unit of food, it simply disappears from the game: your render should no longer return it. If more than `game_info['num_allowed_unfed']` animals leave the board, then the game status

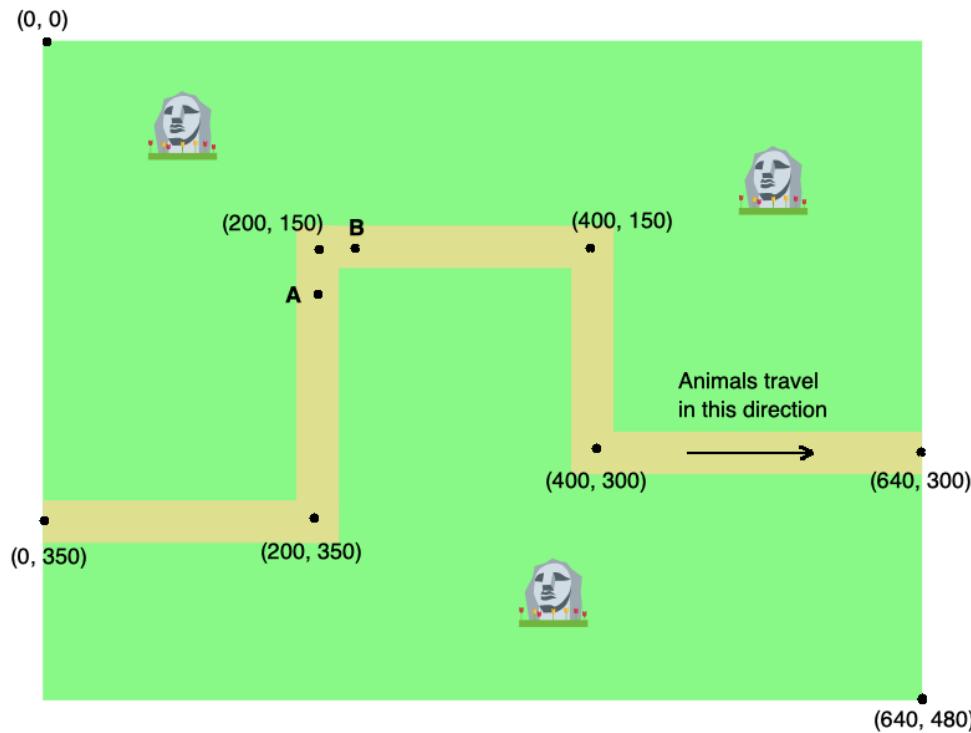
permanently becomes 'defeat', and the game comes to an end. Once this happens, the dictionary returned in `render` should no longer change, even after any number of calls to timesteps.

5.1) The Path

Each segment of the path must be parallel to one of the board edges. That is, every pair of consecutive coordinates in `game_info['path_corners']` share either their x- or y-coordinate. Additionally, the path is guaranteed not to intersect any of the rocks. The path has a thickness given by `Constants.PATH_THICKNESS`.

You're welcome to process `game_info['path_corners']` and store the path however you see fit.

Here's a board with just a path and some rocks, labeled with various coordinates and the direction of animal travel.



If `game_info['animal_speed']` is equal to the distance from **A** to $(200, 150)$ plus the distance from $(200, 150)$ to **B**, then an animal at **A** during one timestep will be at **B** in the next timestep. Animals do turn path corners, even within a single timestep. The distance animals travel is measured *along the path*.

What are the start and end coordinates in the above game? Enter your answer as an ordered list of tuples.

You have submitted this assignment 1 time.

This question is due on Friday April 26, 2019 at 04:00:00 PM.

Enter the representation of the path as it would be given in `game_info['path_corners']`.

```
[(0,350),(200,350),(200,150),(400,150),(400,300),(640,300)]
```

You have submitted this assignment 1 time.

This question is due on Friday April 26, 2019 at 04:00:00 PM.

6) Zookeepers

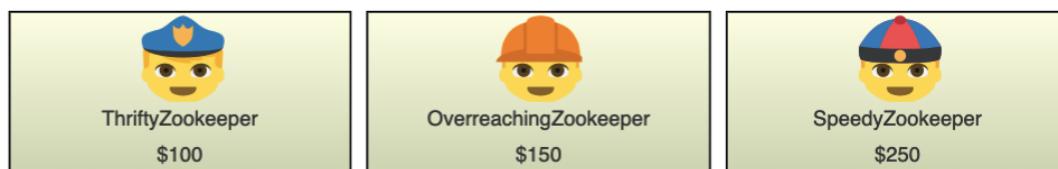
Zookeepers are how the player prevents animals from getting to the path's end without being fed! Zookeepers are placed by the player at permanent locations on the board, and can see animals that enter its range of sight. [Section 6.2](#) explains how zookeepers decide which animal to try to feed and where to throw food.

From your extensive research on zookeeper psychology and physiology, you know that there are exactly three types of zookeepers. They differ in their ability to see animals at long distances, their ability to throw food quickly, and their willingness to work for different amounts of money.

- 'ThriftyZookeeper' has price 100, sight range 100, and throw speed 15
- 'OverreachingZookeeper' has price 150, sight range 150, and throw speed 5
- 'SpeedyZookeeper' has price 250, sight range 50, and throw speed 20

The zookeeper price (`price`) is the amount of the one-time payment needed to hire the zookeeper; the radius of its range of sight (`range`) is the maximum distance away an animal can be such that the zookeeper can see (and therefore decide to throw food at) it; and the throw speed (`throw_speed_mag`) is the speed at which the zookeeper's thrown food travels. The speed gives the total distance traveled per timestep of each unit of food thrown from the zookeeper, no matter the direction in which it is thrown. The `Constants` class in `lab.py` already defines each zookeeper's `price`, `range`, and `throw_speed_mag` for you.

Here are their textures, shown within the UI's zookeeper choice panel:



6.1) Placement

To place a zookeeper, the user first clicks on the zookeeper type they want on the UI's zookeeper choice panel. The UI will interpret the location of this click and pass it into the `mouse` argument of the `timestep` function as one of the zookeeper labels ('SpeedyZookeeper', 'ThriftyZookeeper', or 'OverreachingZookeeper'). Notice that your Game therefore need not know anything about the rendering or handling of the zookeeper choice panel -- good abstraction!

After having chosen a zookeeper type, the user's next click indicates the center of the location at which they wish to place the zookeeper, and is passed into `mouse` as an integer tuple coordinate. Upon receiving this coordinate, you should **first** check whether the user can afford to pay the zookeeper, i.e. if the player's balance is greater than or equal to the price of the zookeeper. If it is not, raise a `NotEnoughMoneyError`, (defined for you in `lab.py`). If the player has enough money, you should **then** check whether the location is valid. A location is not valid if the zookeeper placed there would overlap with the path or any existing rocks or zookeepers. If you have enough money and the clicked location is valid, then the zookeeper is bought and placed there.

If the player has enough money to place the selected zookeeper, allow for multiple click attempts until the placement is valid. If the user would like to place multiple new zookeepers, they must reselect the zookeeper type from the zookeeper choice panel after every valid placement. This means that you should only "deselect" a keeper if keeper placement is successful.

Your implementation should deal with indecisive players: if a player clicks on multiple different zookeeper types before placing a zookeeper, the last-clicked type should be taken.

6.2) Throwing Food

Once a zookeeper is placed, they can begin helping the player feed animals. They do this by throwing food. An animal falls in a zookeeper's range if its center lies inside of or exactly on the edge of the zookeeper's circular range, which is centered at the zookeeper's center. At any given timestep, some animal(s) may be in the range of a particular zookeeper. If this is the case, then the zookeeper needs to choose one animal to attempt to feed and may throw (only) a single unit of food. If there are many animals in range, then the zookeeper chooses to aim for the animal which, among those in range, is probably hungriest: the one which has traveled furthest along the path. *Note that being furthest along the path is not necessarily the same as being closest to the path's last coordinate distance-wise.*

For clarity, we'll call the animal for which the zookeeper decides to aim the `aim_animal`. Luckily, zookeepers are quite diligent in making sure their food reaches the animals. To make it more likely that the food will reach (and therefore feed) the `aim_animal`, zookeepers consider where the `aim_animal` will be in the future. Food only reaches and feeds an animal if the food (a formation with dimensions `Constants.FOOD_WIDTH` and `Constants.FOOD_HEIGHT`) intersects the animal formation.

Here's more detail on their algorithm:²

1. For each integer coordinate `coord` which lies on the center of the path:
 - Disregard this `coord` if the animal has already passed it or is currently there. Otherwise, proceed.
 - Compute how long (in possibly-fractional number of timesteps) it will be before the center of `aim_animal` reaches `coord`, assuming `aim_animal` keeps moving at its current speed.
 - Compute how long (also in possibly-fractional number of timesteps) it would be before the center of food launched in the direction of `coord` would reach `coord`.
2. Throw food in the direction of the coordinate which has the smallest difference between the times that the hypothetical thrown food and the `aim_animal` would reach it.
3. Give the food the total magnitude of speed allowed by the zookeeper type.

Thrown food starts at the center point of the zookeeper that throws it. If the food intersects with an animal, then the animal is fed, and both the animal and the food disappear from the board, regardless of whether that animal is still in the zookeeper range or not. The player earns one dollar for feeding one animal. If the food does not hit an animal, it simply disappears when its center is no longer on the board.

If multiple pieces of food intersect with an animal in a given timestep, remove all those pieces of food. (Our animals are very hungry!) But make sure to only gain one dollar per animal fed, not per piece of food eaten.

Note that the algorithm may cause a food item to end up at a nonintegral coordinate position. Our UI accepts noninteger coordinates, and renders them properly. Do not round at any point, and do not cast any values to integers.

7) Code Submission

[Download Your Last Submission](#)

[Click to View Your Last Submission](#)

Select File

No file selected

Submit

You have submitted this assignment 3 times.

This question is due on Friday April 26, 2019 at 04:00:00 PM.

8) Checkoff

Once you are finished with the code, please come to a tutorial, lab session, or office hour and add yourself to the queue asking for a checkoff. **You must be ready to discuss your code and test cases in detail before asking for a checkoff.**

You should be prepared to demonstrate your code (which should be well-commented, should avoid repetition, and should make good use of helper functions). In particular, be prepared to discuss:

- Implementation of timestep
- Internal representation of the path
- Storage of information about different zookeepers
- Logic to detect intersections (of food with animals, keepers with path/rocks, etc.)
- Food throwing algorithm
- Seeing your game in action on the UI!

Grade

Grading:

- Concept questions (0.5 points): 0.5
- Tests (1.5 points): 1.5
- Checkoff (2 points): 2

Total: 4 Points (of 4)

Footnotes

¹ It's conventional to name constants with all-caps in Python.

² This algorithm has some interesting subtleties (which you do not need to worry about). If you're interested, notice that it considers points on the path at which animals may never actually be, since animals may "skip" points on the path by moving more than one point forward per timestep. This suggests that the algorithm can be improved for performance, especially in games with high animal speeds. Notice also that the algorithm treats the animal as a point, which it is in fact not. In our game, animal formations are squares. It's interesting to consider how this algorithm performs on formations with less symmetric shapes.

