Overview

In this mini-project, we will create a simulation of zombies and humans interacting on a grid. As in the movies, our zombies are hungry for human brains. As a result, zombies chase humans and humans flee from zombies. To keep our simulation manageable, the positions of the zombies and humans will be restricted to a grid. In our simulation, zombies are not very agile and can only move up, down, left or right in one step of the simulation. On the other hand, humans are more agile and can move in these four directions as well as the four neighboring diagonal directions. If a zombie catches a human by positioning itself in the same cell, the zombie enjoys some delicious human brains. Being a Computer Scientist, the human has plenty of brains to spare and continues to live on in our simulation ☺.

To enhance the realism of our simulation, some of the cells in this grid will be marked as impassable and restrict zombie/human movement so that they can not move through these cells. Our task in this simulation is to implement an **`Apocalypse`** class that encapsulates the core mechanisms of this simulation and that interacts with a GUI that we have created for visualizing the simulation in CodeSkulptor. This **`Apocalypse`** class is a sub-class of the **`Grid`** class and inherits the **`Grid`** class methods. Passable cells in the grid correspond to **`EMPTY`** cells while **`FULL`** cells are impassable. Humans and zombies can only inhabit passable cells of the grid. However, several humans and zombies may inhabit the same grid cell.

This **`Apocalypse`** class also includes two lists, one for zombies and one for humans. Note that the entries in each list are cell indices of the form **`(row, col)`** that represent the position of zombies/humans in the grid. Each step in the simulation will either update the positions of the zombies based on the state of the grid and the position of the humans or update the positions of the humans based on the state of the grid and the position of the zombies.

Testing your code

Your task for this mini-project is to implement the **`Apocalypse`** class described in detail below. Remember to test each method as you implement it using the testing philosophy discussed in class. You are also welcome to experiment with your simulation code in CodeSkulptor using our provided GUI. Note that the template contains two lines of the form

```
import poc_zombie_gui
poc_zombie_gui.run_gui(Apocalypse(30, 40))

```

that will load, create and run our GUI in CodeSkulptor. Once you are confident that your implementation works correctly, test your code using our Owltest test suite.

Remember that OwlTest uses Pylint to check that you have followed the coding style guidelines for this class. Deviations from these style guidelines will result in deductions from your final score. Please read the feedback from Pylint closely. If you have questions, feel free to consult this page and the class forums.

When you are ready to submit your code to be graded formally, submit your code to the CourseraTest page for this mini-project that is linked on the main assignment page.

Phase One

In phase one, we will implement the basic methods for the **`Apocalypse`** class. We suggest that you start from this template. Note that the **`Apocalypse`** class is a subclass of the **`Grid`** class and inherits all of its methods.

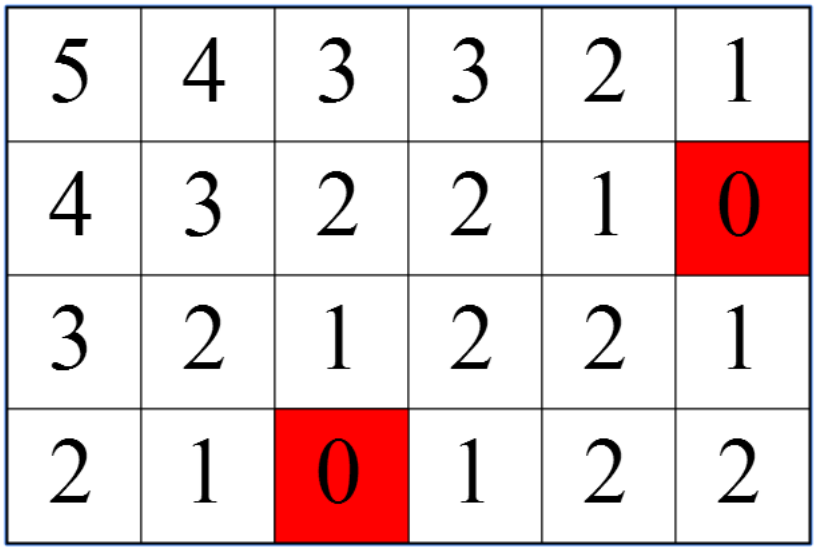
The template contains an implementation of the **`__init__`** method for the **`Apocalypse`** class. The initializer takes two required arguments **`grid_height`** and **`grid_width`**. The initializer also takes three optional arguments **`obstacle_list`**, **`zombie_list`**, and **`human_list`** which are lists of cells that initially contain obstacles, zombies and humans, respectively. For phase one, your task is to implement the remaining seven **`Apocalypse`** methods:

- **`def clear(self):`** Reset all cells in the grid to be passable and reinitialize the human and zombie lists to be empty. Remember that you can use the **`clear`** method from the **`poc_grid.Grid`** class to clear the grid of impassable cells. Examine the implementation of the **`__init__`** method for how to call this method.
- **`def add_zombie(self, row, col):`** Add a zombie to the zombie list at the supplied row and column.
- **`def num_zombies(self):`** Return the number of zombies in the zombie list.
- **`def zombies(self):`** Generator that allows you to iterate over zombies in the zombie list. Here, a zombie is a tuple of the form **`(row, col)`** indicating the zombie's location in the grid. The generator **must** yield the zombies in the order they were added (even if they have moved). Remember that you can use a generator to implement this method in one or two lines of code.
- **`def add_human(self, row, col):`** Add a human to the human list at the supplied row and column.
- **`def num_humans(self):`** Return the number of humans in the human list.
- **`def humans(self):`** Generator that allows you to iterate over humans in the human list. The generator **must** yield the humans in the order they were added (even if they have moved). Again, you can use a generator to implement this method in one or two lines of code.

Once you have successfully implemented these methods, you should be able to add zombies (red cells) and humans (green cells) to the simulation by toggling the button labelled **"Mouse click: ..."** and then clicking on the canvas. Cells that are occupied by both zombies and humans are purple.

Phase Two

Phase two is the core of this mini-project. Your task will be to compute a simple approximation of the distance from each cell in the grid to the nearest zombie (or human). This distance will correspond to the length of the *shortest* sequence of adjacent grid cells (a path) from the cell to a zombie. This 2D array of integer distances is a *distance field*. The image below shows an example of two (red) zombies on a 4×6 grid and the distances from each cell in the grid to the nearest zombie. Note that in this diagram, we are using the cell's four neighbors when determining whether cells are adjacent.



Observe that the distances in this example grow in a manner strikingly similar to the order in which cells are visited during breadth-first search. This observation is not a coincidence. In fact, this distance field was computed using breadth-first search. To compute this distance field, start by recalling the English description of breadth-first search.

```
while boundary is not empty:
    current_cell  ←  dequeue boundary
    for all neighbor_cell of current_cell:
        if neighbor_cell is not in visited:
            add neighbor_cell to visited
            enqueue neighbor_cell onto boundary

```

This description can be modified to compute a distance field during breadth-first search as follows:

- Create a new grid **`visited`** of the same size as the original grid and initialize its cells to be empty.
- Create a 2D list **`distance_field`** of the same size as the original grid and initialize each of its entries to be the product of the height times the width of the grid. (This value is larger than any possible distance.)
- Create a queue **`boundary`** that is a **copy** of either the zombie list or the human list. For cells in the queue, initialize **`visited`** to be **`FULL`** and **`distance_field`** to be zero. We recommend that you use our **`Queue`** class.
- Finally, implement a modified version of the BFS search described above. For each **`neighbor_cell`** in the inner loop, check whether the cell has not been visited and is passable. If so, update the **`visited`** grid and the **`boundary`** queue as specified. In this case, also update the neighbor's distance to be the distance to **`current_cell`** plus one (`distance_field[current_cell[0]][current_cell[1]] + 1`).

This method computes distances in exactly the order that the wild-fire spreads. When a **`neighbor_cell`** is added to the queue, that neighbor's distance from a start position is just the distance to the cell **`current_cell`** plus the one step to the neighbor. Working from the outline above, your task in phase two is to implement the method **`compute_distance_field`** as specified below:

- **`def compute_distance_field(self, entity_type):`** This method returns a 2D distance field computed using the four-way distance to entities of the given type (either **`ZOMBIE`** or **`HUMAN`**). Note that entries of the computed distance fields should be zero at the entities in the specified list. Non-zero distances should be computed using the shortest path computation based on breadth-first search described above. These shortest paths should avoid impassable cells.

Finally, if you are having trouble converting our English description above into Python, remember that the **`update_boundary()`** method from the wild-fire demo implements one step of breadth-first search.

Phase Three

In phase three, your task is to implement two final methods that update the positions of the zombies and humans, respectively.

- **`def move_humans(self, zombie_distance_field):`** This method updates the entries in the human list to model humans avoiding zombies. Each human either stays in its current cell or moves to a neighboring cell to maximize its distance from the zombies. Specifically, humans move to a cell that maximize their distance from the zombies according to the supplied **`zombie_distance_field`**. In the case where several cells shared the same maximal distance, we recommend (but do not require) choosing among these cells at random.
- **`def move_zombies(self, human_distance_field):`** This method updates the entries in the zombie list to model zombies chasing humans. Each zombie either stays in its current cell or moves to a neighboring cell to minimize its distance to the humans. Specifically, zombies moves to the cell that minimizes their distance to the humans according to the supplied **`human_distance_field`**. In the case where several cells shared the same minimal distance, we recommend choosing (but do not require) among these cells at random.

Once you have successfully implemented the three methods in phase two and phase three, the buttons in the GUI labelled **`"Zombies stalk`** and **`"Humans flee"`** should work. Zombies should stalk humans and humans should flee zombies. We encourage you to spend some time testing/experimenting with the simulation using our GUI. Once your code works, submit it to Owltest for grading.

Just for fun! Observations from the simulation

At a distance, zombies are quite good at finding humans (even ones hiding in buildings) since breadth-first search always searches the interiors of buildings (provided there is an entrance). The human distance field decreases steadily and always reaches zero at a delicious human brain. Humans, on the other hand, aren't so smart while fleeing since they greedily maximize the local distance away from the nearest zombies on each step. As a result, humans often run to the nearest corner of a building and cower as the zombies approach. (Hey, this is a pretty realistic simulation!)

However, in close quarters, humans have a large advantage since they can move diagonally. Unless cornered by multiple zombies, humans can easily out-maneuver a single zombie and avoid having their brains eaten. For packs of zombies inhabiting a single cell, the strategy for breaking ties in **`move_zombies`** is important. When the pack has a choice between moving to two neighboring cells that minimize distance, always choosing the one cell or the other causes the zombies to stay in clumped in a single cell. Having each zombie choose between the two cells at random causes the pack to tend to disperse.

This situation arises when the pack is chasing a human that is fleeing diagonally (say down and left). There are two directions (down or left) that are available to the zombies that minimize the distance to the human. Instead of having all of the zombies move in the same direction, each zombie should choose one of the two optimal directions randomly. With the random approach, some zombies go down and some zombies go left and the pack of zombies tends to spread out. This behavior makes the zombies more likely to catch the human when he/she ends up huddled in a corner.

There are several other interesting scenarios in the simulation that we will let you explore on your own. For example, you might consider creating a "zombie-proof" building that exploits the human's ability to move diagonally between obstacles. Feel free to post interesting scenarios that you discover in the forum. So, have fun and enjoy some "Brains!"

✓ Complete