Lab 3: HyperMines

Table of Contents

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: `lab3.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:

- answering the questions on this page (0.3 points)
- passing the test cases from `test.py` under the time limit (1.7 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 your tests if they run to completion in under 100 seconds on the server.

Please also review the collaboration policy before continuing.

The questions on this page (including your code submission) are due at 4pm on Friday, Mar 1.

2) Introduction

READ THROUGH THIS ENTIRE LAB THOROUGHLY BEFORE STARTING

We recommend thinking through and answering all concept questions BEFORE beginning to code. Also read through all provided code before beginning to write your own.

International Mines tournaments have declined lately, but there is word that a new one is in the works, and rumor has it that it could be even bigger than the legendary Budapest 2005 tournament! In preparation for the tournament, Ben Bitdiddle has written an implementation of the Mines game. In order to prepare yourself, you decide to look over his implementation in detail.

2.1) *Mines*

Mines is played on a rectangular $r \times c$ board (where r indicates the number of rows and c the number of columns), covered with 1×1 square tiles. Some of these tiles hide secretly buried mines; all the other squares are safe. On each turn, the player removes one tile, revealing either a mine or a safe square. The game is won when all safe squares have been revealed, without revealing a single mine, and it is lost if a mine is revealed.

The game wouldn't be very entertaining if it only involved random guessing, so the following twist is added: when a safe square is revealed, that square is additionally inscribed with a number between `0` and `8` (inclusive), indicating the number of surrounding mines (when rendering the board, `0` is replaced by a blank, or `' '`). Additionally, any time a `0` is revealed (a square surrounded by no mines), all the surrounding squares are also automatically revealed (they are, by definition, safe).

2.2) Playing the Game

Run `python3 server2d.py` in your terminal and go to http://localhost:7000 in your browser to play a game of Mines!

We highly recommend playing a few games, especially if you have not previously played some variant of *Mines*.

2.3) Ben Bitdiddle's implementation

Once you've gotten some practice, you should also read through `lab2d.py` to look "under the hood" at Ben Bitdiddle's implementation, to get a better sense of how the game works.

The following methods are critical to the operation of the game:

- `__init__(dimensions, bombs)` - This starts a new *Mines* game with the correct internal values initialized. `dimensions` is a list representing the dimensions of the board, and `bombs` is a list of locations of bombs on the board.
- `dig(coords)` - This method digs up, or reveals, the square at the specified coordinates (`coords`) and then recursively digs up neighbors according to the rules of the game.
- `render(xray=False)` - This method returns a string representation of the current board. If `xray` is `True` (the default is `False`), all cells are shown regardless of whether they have already been dug.

Luckily for us, Ben really outdid himself with this implementation. In addition to the methods above, he has also the following helper class methods in the `MinesGame` class.

- `make_board(dimensions, elem)` - This method creates a 2-D list representing a *Mines* board of the provided `dimensions`, initializing each element in the 2-D list to `elem`.
- `is_in_bounds(coords)` - This method checks a specific coordinate (`coords`) to see whether it is a valid location on the current board.
- `neighbors(coords)` - This method returns a list of all of the valid coordinates which neighbor the provided coordinate (`coords`).
- `is_victory()` - This method checks the board to see whether it satisfies the victory condition.

- `dump()` - This method prints a human-readable representation of the current game. Useful for debugging your code! **PLEASE DO NOT EDIT THIS METHOD!**

His code is even well-documented (which is astounding given his track record in some other classes). You should familiarize yourself with his code, which can be found in the provided `lab2d.py` file.

3) *HyperMines*

Now that you've mastered 2-D mines, it's time to participate in the International Mines tournament! But wait! This year's tournament comes with a small twist. The tournament will be taking place on planet *Htrae*, and not on Earth! `:)` The Planet *Htrae* is not very different from Earth, except that space in the *Yklim way*, *Htrae*'s star cluster, doesn't have three dimensions — at least, not always: it fluctuates between 2 and, on the worst days, 60. In fact, it's not uncommon for an Htraean to wake up flat, for example, and finish the day in 7 dimensions — only to find themselves living in 3 or 4 dimensions on the next morning. It takes a bit of time to get used to, of course.

In any case, Htraeans are pretty particular about playing *Mines*. Kids on *Htrae* always play on regular, 2-D boards, but champions like to play on higher-dimensional boards, usually on as many dimensions as the surrounding space. Your code will have to support that flexibility, of course. Here's the weather advisory for the week of the tournament:

```
...VERY DIMENSIONAL IN SOUTHWEST OHADI ON YADIRF...

AN EXITING LOW PRESSURE SYSTEM WILL INCREASE NORTHWEST DIMENSIONAL
FLUX IN AND SOUTH OF THE EKANS RIVER BASIN ON YADIRF. ESIOB IS NOW
INCLUDED IN THE ADVISORY BUT THE STRONGEST FLUX WILL BE SOUTH AND
EAST OF MOUNTAIN EMOH TOWARD THE CIMAG VALLEY.

ZDI014>030-231330-016-
UPPER ERUSAERT VALLEY-SOUTHWEST HYPERLANDS-WESTERN CIMAG VALLEY-
1022 MP TDM UHT PES 22 6102

...DIMENSION ADVISORY REMAINS IN EFFECT FROM 10 MA TO 9 MP ON
YADIRF...

* DIMENSIONAL FLUX...30 TO 35 DIMENSIONS WITH GUSTS TO 45.

* IMPACTS...CROSSFLUXES WILL MAKE FOR DIFFICULT TRAVELLING
  CONDITIONS ON LOW-DIMENSIONAL ROADS.
```

Since most of the difficulty of this lab lies in implementing recursive functions, please do not use standard library modules that use recursion behind the scenes, such as `itertools`.

4) Rules

HyperMines is the Htraean twist on *Mines*. Unlike *Mines*, *HyperMines* is played on a board with an arbitrary number of dimensions. Everything works just the same as in *Mines*, except for the fact that each cell has up to $3^n - 1$ neighbors, instead of 8.

As usual, you mainly need to edit `lab.py` to complete this assignment, though we also ask you to add new test cases in `test.py`.

One of the implementation challenges in *HyperMines* is arbitrary-depth iteration. We include a **few hints** that you may find useful as you think about which recursive helper functions would be helpful in dealing with N-D arrays represented as nested lists.

5) How to test your code

We provide two scripts to test and enjoy your code:

- `python3 server.py` lets you play *HyperMines* in your browser (http://localhost:8000)! The server uses your code `lab.py` to compute consecutive game states.
- `python3 test.py` runs all the tests used for grading, as well as any additional test cases you put in `test.py`.

6) Implementation

6.1) Game state

You should represent games as `HyperMinesGame` instances with the following four attributes:

- `dimensions`: the board's dimensions represented as a list of positive numbers.
- `board`: an N-D array (a nested list of lists) of integers and strings.
  In a `HyperMinesGame` instance called `hmg`, `hmg.board[x_0][...][x_k]` is `"."` iff (if and only if) the square at position $(x_0, \ldots, x_k)$ contains a bomb. If that square does not contain a bomb, `hmg.board[x_0][...][x_k]` is an integer indicating the number of neighboring bombs.
- `mask`: an N-D array (a nested list of lists) of Booleans.
  In a `HyperMinesGame` instance called `hmg`, `hmg.mask[x_0][...][x_k]` is `True` iff the square at position $(x_0, \ldots, x_k)$ is revealed to the player.
- `state`: a string indicating the state of the game -- `"ongoing"` if the game is in progress, `"victory"` if the game has been won, and `"defeat"` if the game has been lost. The state of a new game is *always* initialized as `"ongoing"`.

For example, the following is a valid *HyperMines* game state:

```
dimensions = [4, 3, 2]
board = [[[1, 1], ['.', 2], [2, 2]],
         [[1, 1], [2, 2], ['.', 2]],
         [[1, 1], [2, 2], [1, 1]],
         [[1, '.'], [1, 1], [0, 0]]]
mask = [[[True, False], [False, False], [False, False]],
        [[False, False], [True, False], [False, False]],
        [[False, False], [True, True], [True, True]],
        [[False, False], [True, True], [True, True]]]
state = "ongoing"
```

You may find the `HyperMinesGame.dump` method (included in `lab.py`) useful to print game states.

6.2) Game logic

Your task is to implement three core methods within the body of the class: `__init__`, `dig`, and `render`. These functions behave just like their 2-D counterparts, and each of them is documented in detail in `lab.py`.

Additionally, skeleton code for optional helper methods are also provided:

- `get_coords()` - This method returns the value of a square given a list of coordinates.
- `set_coords()` - This method sets the value of a square on the game board given a list of coordinates and a value.

- `make_board(dimensions, elem)` - This method creates a 2-D list representing a *Mines* board of the provided `dimensions`, where each element in the 2-D list is initialized to `elem`.
- `is_in_bounds(coords)` - This method checks a specific coordinate (`coords`) to see whether it is a valid location on the current board.
- `neighbors(coords)` - This method returns a list of all of the valid coordinates which neighbor the provided coordinate (`coords`).
- `is_victory()` - This method checks whether the game satisfies the victory condition.
- `dump()` - This method prints a human-readable representation of the current game. Useful for debugging your code! **PLEASE DO NOT EDIT THIS METHOD!**

Most of these helper methods may seem familiar since they were used in Ben's code (`lab2d.py`) as well! Implementing these helper methods is completely **optional**. If you choose not to implement a helper method, you may delete the method from `lab.py`.

6.2.1) Testing

The test cases we will use to check your code in this lab are pretty complex, and so they are difficult to reason about.

To this end, create some test cases of your own in `test.py`, to handle some more straightforward cases that are easier to reason about.

Create at least 3 new, non-trivial tests in the `TestTiny` class in `test.py`: one for a 1-D game, one for a 2-D game, and one for a 3-D game. Each test should create a new game and perform at least 2 consecutive digs on that game, with different expected behaviors.

You should be prepared to discuss your new test cases during your checkoff.

6.3) An example game

This section runs through an example game in 3-D, showing which functions are called and what they should return in each case. To help understand what happens, calls to `game.dump()` are inserted after each state-modifying step where `game` is a `HyperMinesGame` instance.

```
>>> game = HyperMinesGame([3,3,2],[[1,2,0]])
>>> game.dump()
dimensions: [3, 3, 2]
board: [[0, 0], [1, 1], [1, 1]]
       [[0, 0], [1, 1], ['.', 1]]
       [[0, 0], [1, 1], [1, 1]]
mask:  [[False, False], [False, False], [False, False]]
       [[False, False], [False, False], [False, False]]
       [[False, False], [False, False], [False, False]]
state: ongoing
```

The player tries digging at `[2,1,0]`, which reveals 1 tile.

```
>>> game.dig([2,1,0])
1
>>> game.dump()
dimensions: [3, 3, 2]
board: [[0, 0], [1, 1], [1, 1]]
```

```
        [[0, 0], [1, 1], ['.', 1]]
        [[0, 0], [1, 1], [1, 1]]
mask:   [[False, False], [False, False], [False, False]]
        [[False, False], [False, False], [False, False]]
        [[False, False], [True, False], [False, False]]
state: ongoing
```

... then at `[0,0,0]` which reveals 11 new tiles:

```
>>> game.dig([0,0,0])
11
>>> game.dump()
dimensions: [3, 3, 2]
board: [[0, 0], [1, 1], [1, 1]]
       [[0, 0], [1, 1], ['.', 1]]
       [[0, 0], [1, 1], [1, 1]]
mask:  [[True, True], [True, True], [False, False]]
       [[True, True], [True, True], [False, False]]
       [[True, True], [True, True], [False, False]]
state: ongoing
```

Emboldened by this success, the player then makes a fatal mistake and digs at `[1,2,0]`, revealing a bomb:

```
>>> game.dig([1,2,0])
1
>>> game.dump()
dimensions: [3, 3, 2]
board: [[0, 0], [1, 1], [1, 1]]
       [[0, 0], [1, 1], ['.', 1]]
       [[0, 0], [1, 1], [1, 1]]
mask:  [[True, True], [True, True], [False, False]]
       [[True, True], [True, True], [True, False]]
       [[True, True], [True, True], [False, False]]
state: defeat
```

6.4) Check Yourself

To get a feel for working with an abitrary number of dimensions, answer the following few questions about determining cells' neighbors.

For these questions, it is okay to consider a cell to be its own neighbor, or it is also okay to consider it not to be its own neighbor. Depending on the structure of the rest of your code when implementing this or similar behaviors for Lab 3, you might wish to consider a cell to be its own neighbor, or you may not.

In a 1-D game with dimensions of `[10]`, what are the neighbors of the coordinates `[5]`? Enter a Python list of coordinates (in either tuples or lists) below:

([4],[6])

Submit

You have submitted this assignment 1 time.

This question is due on Thursday March 07, 2019 at 04:00:00 PM.

In a 2-D game with dimensions of `[10, 20]`, what are the neighbors of the coordinates `[5, 13]`? Enter a Python list of coordinates (in either tuples or lists) below:

([4,12],[4,13],[4,14],[5,12],[5,14],[6,12],[6,13],[6,14])

Submit

You have submitted this assignment 1 time.

This question is due on Thursday March 07, 2019 at 04:00:00 PM.

In a 3-D game with dimensions of `[10, 20, 3]`, what are the neighbors of the coordinates `[5, 13, 0]`? Enter a Python list of coordinates (in either tuples or lists) below:

([4,12,0],[4,12,1],[4,13,0],[4,13,1],[4,14,0],[4,14,1],[5,12,0],[

Submit

You have submitted this assignment 2 times.

This question is due on Thursday March 07, 2019 at 04:00:00 PM.

Take a careful look at your results for these questions. How do the results from one question help you solve the next?

7) Code Submission

Once you have debugged your code locally and are satisfied, upload your `lab.py` below:

Download Your Last Submission

Click to View Your Last Submission

Select File No file selected

Submit

You have submitted this assignment 2 times.

This question is due on Thursday March 07, 2019 at 04:00:00 PM.

After writing your own test cases, upload your `test.py` below.

File submission is factored into concept questions grading, and will be evaluated during the checkoff.

For fairness, only the file submitted here (not the local copy) can be used during the checkoff. Note that `test.py` has the same deadline as `lab.py`, which is before the checkoff deadline.

Download Your Last Submission

Click to View Your Last Submission

Select File No file selected

Submit

You have submitted this assignment 2 times.

This question is due on Thursday March 07, 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:

- Your implementation of `__init__`, including any helper functions.
- Your implementation of `dig`, including any helper functions.
- Your implementation of `render`, including any helper functions.
- The new test cases you added to `test.py`. File submission is required by the same deadline as `lab.py`'s, and local copies are NOT allowed.
- Play a game by running `server.py`, initializing a board, and digging a few times.

8.1) Grade

Grading:

- Concept questions (0.3 points): 0.3
- Tests (1.7 points): 1.7
- Checkoff (2 points): 2

Total: 4 Points (of 4)