HyperMines

Submission to website: Monday, March 6, 10pm

Checkoff by LA/TA: Thursday, March 9, 10pm

This lab assumes you have Python 3.5 installed on your machine. Please use the Chromium or Firefox web browser.

This lab has 18 tests and 3 coding points. The lab requires that you supply docstrings and doctests for all the functions you implement -- in fact, the first 3 tests check that you've done that! The coding points are earned as follows:

- 1 point for well-organized code that makes appropriate use of helper functions to avoid code repetition and enhance readability and testability.
- 1 point for providing informative and succinct docstrings. Please read https://www.python.org/dev/peps/pep-0257/ to learn about the semantics and conventions associated with Python docstrings. Note that the first line of a function's docstring is special: many Python development environments will display that line when the programmer starts to type a call to the function.
- 1 point for providing non-trivial doctests for each function's functionality.

Introduction

Now that you've mastered 2D mines, it's time for a small field trip :) There's just a small twist. 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 to three or four 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, 2D 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 this, of course. Here's the weather advisory for the week of the tournament:

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

.AN EXITING LOW PRESSURE SYSTEM WILL INCREASE NORTHWEST DIMENSIONAL
FLUX IN AND SOUTH OF THE EKANS RIVER BASIN ON YADRIF. 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.
```

This lab trains the following skills:

- Manipulating N-dimensional arrays implemented with nested lists
- Recursively exploring simple graphs and nested data structures
- Writing doctests and documenting functions

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`.

Directions

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`.

What you need to do

As usual, you only need to edit `lab.py` to complete this assignment.
You will need to correctly implement `nd_new_game(dims, bombs)`, `nd_dig(game, coords)`, `nd_render(game, xray)` to earn full credit for this lab. Just like before, **please write documentation and doctests for all new functions**.

If you're starting with a copy of your 2D Mines code, you can delete the testing functions you added for Part 3: Bug hunt. They aren't needed for this lab.

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

How to test your code

We provide three scripts to test and enjoy your code:

- `python3 simpletests.py` is an interactive script that will let you select which doctests to run.
- `python3 test.py` runs all the tests used for grading; these are the ones that the auto-grader will run on our servers.
- `python3 server.py` lets you play *HyperMines* in your browser! It uses your code to compute consecutive game states.

Good luck! As it turns out *HyperMines* is enjoyed here on Earth too -- check out http://gravitation3d.com/xezlec/.

Implementation

Game state

As before, the state of an ongoing*HyperMines* game is represented as a dictionary with four fields:

- `"dimensions"`, the board's dimensions (an arbitrary list of positive numbers in *HyperMines*)
- `"board"`, an N-dimensional array (implemented using nested lists) of integers and strings. In *HyperMines*, `game["board"][coordinate_1][...][coordinate_n]` is "." if the square with coordinate `[coordinate_1,...,coordinate_n]` contains a bomb.
- `"mask"`, an N-dimensional array (implemeted using nested lists) of Booleans. In *HyperMines*, `game["mask"][coordinate_1][...][coordinate_n]` indicates whether the contents of square `[coordinate_1,...,coordinate_n]` are visible to the player.
- `"state"`, a string containing 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* `"ongoing"`.

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

```
gameN = {'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 `dump(game)` function (included in `lab.py`) useful to print game states.

Game logic

Your task is to implement three functions: `nd_new_game(dims, bombs)`, `nd_dig(game, coords)`, and `nd_render(game, xray)`. These functions behave just like their 2D counterparts, and each of them is documented in detail in `lab.py`.

An example game

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

```
>>> from lab import *
>>> game = nd_new_game([3,3,2],[[1,2,0]])
>>> dump(game)
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.

```
>>> nd_dig(game, [2,1,0])
1
>>> dump(game)
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:

```
>>> nd_dig(game, [0,0,0])
11
>>> dump(game)
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:

```
>>> nd_dig(game, [1,2,0])
1
>>> dump(game)
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
```