Mines

Introduction

International *Mines* tournaments have declined lately, but there is word that a new one is in the works — rumor has it that it could be even bigger than the legendary Budapest 2005 one, in no small part because inhabitants of planet *Htrae* might be attending.

What better way to get prepared for the tournament than to write your own implementation? 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 it, 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 2- and 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`.

Part 2 of this lab is tricky; please plan accordingly.

Directions

Rules

Mines

Mines is played on a rectangular `n × m` board, covered with `1×1` square tiles. `k` of these tiles hide a secretly buried mine; all the other squares are safe. On each turn the player removes one tile, revealing either a mine or a safe square. The game completes when all safe tiles have been removed, without revealing a single mine.

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, indicating the number of surrounding mines (when rendering the board, `0` is replaced by a blank). Additionally, if removing a tile reveals a `0` (a square surrounded by no mines), then the surrounding squares are also automatically revealed (they are, by definition, safe).

Feel free to play one of the implementations available online to get a better feel of the game!

HyperMines

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. This lab has 2 parts. In the first one, you will implement *Mines* (in 2D); in the second one, you will implement the fully general *HyperMines*. You can jump straight to *HyperMines*, but beware: implementing *Mines* is much easier and will give you useful insight for *HyperMines*, so we recommend doing that first.

Part 1: *Mines*

You will need to correctly implement `new_game(nrows, ncols, bombs)`, `dig(game, row, col)`, `render(game, xray)`, and `render_ascii(game)` to earn full credit for this part (these functions are described below). Additionally, we ask that you write a small documentation string for each new function that you introduce, along with a few tests in doctest format.

Part 2: *HyperMines*

You will need to correctly implement `nd_new_game(nrows, ncols, bombs)`, `nd_dig(game, row, col)`, `nd_render(game, xray)` to earn full credit for this part. Just like before, please write documentation and doctests for all new functions.

How to test your code

We provide three scripts to test and enjoy your code:

- `python simpletests.py` is an interactive testing script. You can pick which function to test, and it will show the input, and the output that your function produced, along with a diagnostic. Choosing "all tests" runs a battery of simple tests on your code, including the doctests in your `lab.py` and the example game below. This script produces detailed error messages to help with debugging.
- `python test.py` runs all the tests used for grading; these are the ones that the auto-grader will run on our servers.
- `python server.py` lets you play *Mines* in your browser! It uses your code to compute consecutive game states.

Good luck!

Implementation

Game state

The state of an ongoing *Mines* or *HyperMines* game is represented as a dictionary with three fields:

- `"dimensions"`, the board's dimensions (`[nrows, ncolumns]` in *Mines*, an arbitrary list in *HyperMines*)
- `"board"`, an N-dimensional array (implemented using nested lists) of integers and strings. In *Mines*, `game["board"][r][c]` is "." if square `(r, c)` contains a bomb, and a number indicating the number of neighboring bombs otherwise. In *HyperMines* things are similar, with `game["board"][coordinate_1][...][coordinate_n]` instead of `game["board"][r][c]`.
- `"mask"`, an N-dimensional array (implemeted using nested lists) of Booleans. In *Mines*, `game["mask"][r][c]` indicates whether the contents of square `(r, c)` are visible to the player. In *HyperMines* things are similar, with `game["mask"][coordinate_1][...][coordinate_n]` instead of `game["mask"][r][c]`.

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

```
python game2D = {'dimensions': [3, 4], 'board': [[1, '.', 2], [1, 2, '.'], [1, 2, 1], ['.', 1, 0]], 'mask': [[True, False, False], [False, True, False], [False, True, True], [False, True, True]]}
```

And the following is a valid *HyperMines* game state:

```
python 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]]]}
```

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

Game logic

Part 1

Your task for part 1 is to implement four functions: `new_game(nrows, ncols, bombs)`, `dig(game, row, col)`, `render(game, xray)`, and `render_ascii(game)`. Each of these functions is documented in detail in `lab.py`, and described succinctly below. Notice how each function comes with a detailed docstring documenting what it does. In addition, each function (in `lab.py`) comes with a few small tests (these are called doctests), which you can use as basic sanity checks.

- `new_game(nrows, ncols, bombs)` creates a new game state:

  ```python def new_game(nrows, ncols, bombs): """Start a new game.

  Return a game state dictionary, with the "board" and "mask" fields
  adequately initialized.

  Args:
     num_rows (int): Number of rows
     num_cols (int): Number of columns
     bombs (list): List of bombs, given in (row, column) pairs

  Returns:
     A game state dictionary
  """
  ```

- `dig(game, row, col)` implements the digging logic, and checks whether the game is over:

  ```python def dig(game, row, col): """Recursively dig up (row, col) and neighboring squares.

  Update game["mask"] to reveal (row, col); then recursively reveal (dig up)
  its neighbors, as long as (row, col) does not contain and is not adjacent to
  a bomb.  Return a pair: the first element indicates whether the game is over
  using a string equal to "victory", "defeat", or "ongoing", and the second
  one is a number indicates how many squares were revealed.

  The first element is "defeat" when at least one bomb is visible on the board
  after digging (i.e. game["mask"][bomb_location] == True), "victory" when all
  safe squares (squares that do not contain a bomb) and no bombs are visible,
  and "ongoing" otherwise.

  Args:
     game (dict): Game state
     row (int): Where to start digging (row)
     col (int): Where to start digging (col)

  Returns:
     Tuple[str,int]: A pair of game status and number of squares revealed
  ```

```
    """
```

```

- `render(game, xray)` renders the game into a 2D grid (for display):

  ``` python def render(game, xray=False): """Prepare a game for display.

  ```
  Returns a two-dimensional array (list of lists) of "_" (hidden squares), "."
  (bombs), " " (empty squares), or "1", "2", etc. (squares neighboring bombs).
  game["mask"] indicates which squares should be visible. If xray is True (the
  default is False), game["mask"] is ignored and all cells are shown.

  Args:
     game (dict): Game state
     xray (bool): Whether to reveal all tiles or just the ones allowed by
                  game["mask"]

  Returns:
     A 2D array (list of lists)
  """
  ```

  ```

- `render_ascii(game, xray)` renders a game state as ASCII art:

  ```python def render_ascii(game, xray=False): """Render a game as ASCII art.

  ```
  Returns a string-based representation of argument "game". Each tile of the
  game board should be rendered as in the function "render(game)".

  Args:
     game (dict): Game state
     xray (bool): Whether to reveal all tiles or just the ones allowed by
                  game["mask"]

  Returns:
     A string-based representation of game
  """
  ```

  ```

  Here is how the output might look like (with `xray=True`):

  ```
   1.1112.1 1112..1 1.22.11...1 2221.211 1.12.31 112.2112432 1.111211 111111 222 2.2 111 1.1 111 1.1 14.3 123211221
  11112.1 111 2..2 1..212..1 1.11.21 2.31 1223.32221211111 111 2.3111.1 111111111 111 112.1111 1.11.22.21111 1.1 111
  111112.4.11.1 11211 12.323342 1.1111 113.3... 1122.1 111 2.33.3 2.31 111 1.1 111222 112.2 1.1 11211111 112.1 .1111 111
  1.11.1 1.211
  ```

### Part 2

Your task for part 2 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 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.

```python

> from lab import *

```

After you start `server.py`, our code runs your `new_game` function with a randomly generated list of bombs:

```python

> game = new*game(6, 6, [(3, 0), (0, 5), (1, 3), (2, 3)]) dump(game) dimensions: [6, 6] board: [0, 0, 1, 1, 2, '.'] [0, 0, 2, '.', 3, 1] [1, 1, 2, '.', 2, 0] ['.', 1, 1, 1, 1, 0] [1, 1, 0, 0, 0, 0] [0, 0, 0, 0, 0, 0] mask: [False, False, False, False, False, False] [False, False, False, False, False, False] [False, False, False, False, False, False] [False, False, False, False, False, False] [False, False, False, False, False, False] [False, False, False, False, False, False] render(game) [['', '', '', '', '', ''], ['', '', '', '', '', ''], ['', '', '', '', '', ''], ['', '', '', '', '', ''], ['', '', '', '', '', ''], ['', '', '', '', '', '']] print(render*ascii(game))

```

Assume the player first digs at `(1, 0)`; our code calls your `dig()` function. The return value `('ongoing', 9)` indicates that the game is ongoing, and that 9 squares were revealed.

```python

> dig(game, 1, 0) ('ongoing', 9) dump(game) dimensions: [6, 6] board: [0, 0, 1, 1, 2, '.'] [0, 0, 2, '.', 3, 1] [1, 1, 2, '.', 2, 0] ['.', 1, 1, 1, 1, 0] [1, 1, 0, 0, 0, 0] [0, 0, 0, 0, 0, 0] mask: [True, True, True, False, False, False] [True, True, True, False, False, False] [True, True, True, False, False, False] [False, False, False, False, False, False] [False, False, False, False, False, False] [False, False, False, False, False, False] render(game) [[' ', ' ', '1', '', '', ''], [' ', ' ', '2', '', '', ''], ['1', '1', '2', '', '', ''], ['', '', '', '', '', ''], ['', '', '', '', '', ''], ['', '', '', '', '', '']] print(render*ascii(game)) 1___ 2___ 112___

```

… then at `(5, 4)` (which reveals 21 new squares):

```python

> dig(game, 5, 4) ('ongoing', 21) dump(game) dimensions: [6, 6] board: [0, 0, 1, 1, 2, '.'] [0, 0, 2, '.', 3, 1] [1, 1, 2, '.', 2, 0] ['.', 1, 1, 1, 1, 0] [1, 1, 0, 0, 0, 0] [0, 0, 0, 0, 0, 0] mask: [True, True, True, False, False, False] [True, True, True, False, True, True] [True, True, True, False, True, True] [False, True, True, True, True, True] [True, True, True, True, True, True] [True, True, True, True, True, True] render(game, False) [[' ', ' ', '1', '', '', ''], [' ', ' ', '2', '', '3', '1'], ['1', '1', '2', '', '2', ' '], ['', '1', '1', '1', '1', ' '], ['1', '1', ' ', ' ', ' ', ' '], [' ', ' ', ' ', ' ', ' ', ' ']] print(render*ascii(game)) 1__ 231 1122 _1111 11

```

Emboldened by this success, the player then makes a fatal mistake and digs at `(0, 5)`, revealing a bomb:

```python

> dig(game, 0, 5) ('defeat', 1) dump(game) dimensions: [6, 6] board: [0, 0, 1, 1, 2, '.'] [0, 0, 2, '.', 3, 1] [1, 1, 2, '.', 2, 0] ['.', 1, 1, 1, 1, 0] [1, 1, 0, 0, 0, 0] [0, 0, 0, 0, 0, 0] mask: [True, True, True, False, False, True] [True, True, True, False, True, True] [True, True, True, False, True, True] [False, True, True, True, True, True] [True, True, True, True, True, True] [True, True, True, True, True, True] render(game) [[' ', ' ', '1', '', '', '.'], [' ', ' ', '2', '', '3', '1'], ['1', '1', '2', '', '2', ' '], ['', '1', '1', '1', '1', ' '], ['1', '1', ' ', ' ', ' ', ' '], [' ', ' ', ' ', ' ', ' ', ' ']] print(render*ascii(game)) 1_. 231 112_2 _1111 11

```
```