Mines

You are not logged in.

Please Log In for full access to the web site.
Note that this link will take you to an external site (`https://shimmer.mit.edu`) to authenticate, and then you will be redirected back to this page.

Table of Contents

1) Preparation

This lab assumes you have Python 3.11 or later installed on your machine (3.11 recommended).

The following file contains code and other resources as a starting point for this lab: `mines.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.

Your raw score for this lab will be counted out of 5 points. Your score for the lab is based on:

- a style check (1 point) and
- passing the tests in `test.py` (4 points)

> **Reminder: Academic Integrity**
>
> Please also review the academic integrity policies before continuing. In particular, **note that you are not allowed to use any code other than that which you have written yourself, including code from online sources and code produced by advanced code-completion tools.**

2) Introduction

International *Minesweeper* tournaments have declined in popularity lately, but we're predicting a resurgence in popularity (maybe the next tournament could be even bigger than the legendary Budapest 2005 tournament?!). In anticipation of just such a resurgence, your roommate has written an implementation of the *Mines* game. In order to prepare *yourself*, you decide to look over your roommate's implementation in detail.

2.1) *Mines*

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

The game wouldn't be much of a game at all 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, any time a `0` is revealed (a square surrounded by no mines), the surrounding squares are also automatically revealed (they are, by definition, safe).

To get a feel for the game mechanics, it may be a good idea to use a search engine to find and play a few games of *Minesweeper* before looking at the code.

3) An Implementation of *Mines*

3.1) Game State

The state of an ongoing *Mines* game is represented as a dictionary with the following four keys:

- `'dimensions'`: a tuple containing the board's dimensions `(nrows, ncolumns)`
- `'board'`: a 2-dimensional array (implemented using nested lists) of integers and strings. `game['board'][r][c]` is `'.'` if square (r, c) contains a mine, and it is a number indicating the number of neighboring mines otherwise.
- `'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'`.

- `'visible'`: a 2-dimensional array (implemented using nested lists) of Booleans. `game['visible'][r][c]` indicates whether the contents of square (r, c) are visible to the player.

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

```
{
    'dimensions': (4, 3),
    'board': [[1,  '.',  2],
              [1,   2,  '.'],
              [1,   2,   1],
              ['.', 1,   0]],
    'state': 'ongoing',
    'visible': [[True, False, False],
                [False, True, False],
                [False, True, True],
                [False, True, True]],
}
```

The `dump` function (provided in `lab.py`) might be useful as you work through the exercises below. (You need not modify this function in any of the following sections.)

3.2) Game Logic

The game is implemented via four functions in `lab.py`:

- `new_game_2d` creates a new object to represent a game, given board dimensions and mine coordinates.
- `dig_2d` implements the digging logic (updating the game state if necessary) and returns the number of new tiles revealed from that move.
- `render_2d_locations` renders the game into a 2D grid (for display).
- `render_2d_board` renders a game state as ASCII art.

Each of these functions is documented in detail in `lab.py`. Notice how each function comes with a detailed docstring documenting what it does.

3.3) An Example Game

This section runs through an example game, showing which functions are called and what they should return in each case.

Calling `new_game_2d` produces a new game object as described above:

```
>>> game = new_game_2d(6, 6, [(3, 0), (0, 5), (1, 3), (2, 3)])
>>> dump(game)
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]
dimensions: (6, 6)
state: ongoing
visible:
    [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]
state: ongoing
>>> render_2d_locations(game)
[['_', '_', '_', '_', '_', '_'],
 ['_', '_', '_', '_', '_', '_'],
 ['_', '_', '_', '_', '_', '_'],
 ['_', '_', '_', '_', '_', '_'],
 ['_', '_', '_', '_', '_', '_'],
 ['_', '_', '_', '_', '_', '_']]
```

Assume the player first digs at `(1, 0)` by invoking the `dig_2d` method. The return value `9` indicates that 9 squares were revealed.

```
>>> dig_2d(game, 1, 0)
9
>>> dump(game)
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]
dimensions: (6, 6)
state: ongoing
visible:
    [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_2d_locations(game)
[[' ', ' ', '1', '_', '_', '_'],
 [' ', ' ', '2', '_', '_', '_'],
 ['1', '1', '2', '_', '_', '_'],
 ['_', '_', '_', '_', '_', '_'],
 ['_', '_', '_', '_', '_', '_'],
 ['_', '_', '_', '_', '_', '_']]
```

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

```
>>> dig_2d(game, 5, 4)
21
>>> dump(game)
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]
dimensions: (6, 6)
state: ongoing
visible:
    [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_2d_locations(game)
[[' ', ' ', '1', '_', '_', '_'],
 [' ', ' ', '2', '_', '3', '1'],
 ['1', '1', '2', '_', '2', ' '],
 ['_', '1', '1', '1', '1', ' '],
 ['1', '1', ' ', ' ', ' ', ' '],
 [' ', ' ', ' ', ' ', ' ', ' ']]
```

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

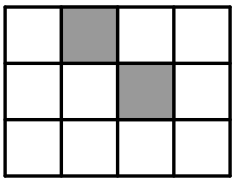
```
>>> dig_2d(game, 0, 5)
1
>>> dump(game)
board:
    [0, 0, 1, 1, 2, '.']
    [0, 0, 2, '.', 3, 1]
    [1, 1, 2, '.', 2, 0]
    ['.', 1, 1, 1, 1, 0]
    [1, 1, 0, 0, 0, 0]
    [0, 0, 0, 0, 0, 0]
dimensions: (6, 6)
state: defeat
visible:
    [True, True, True, False, False, True]
    [True, True, True, 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_2d_locations(game)
[[' ', ' ', '1', '_', '_', '.'],
 [' ', ' ', '2', '_', '3', '1'],
 ['1', '1', '2', '_', '2', ' '],
 ['_', '1', '1', '1', '1', ' '],
 ['1', '1', ' ', ' ', ' ', ' '],
 [' ', ' ', ' ', ' ', ' ', ' ']]
```

3.4) Check Yourself

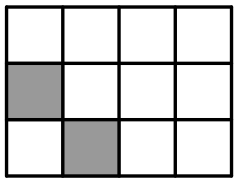
Before we dive into the code, answer the following questions about the representations and operations described above.

Consider the following four boards, where grey boxes represent mine locations and white boxes represent open squares:

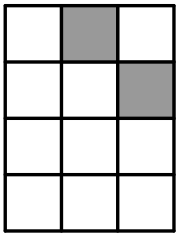
A



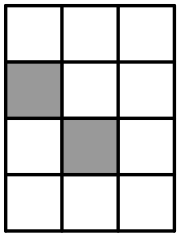
B



C



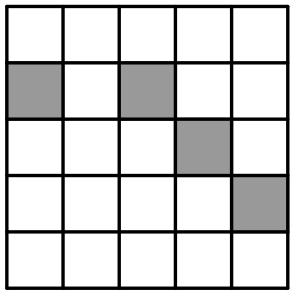
D



Which of these would result from calling `new_game_2d(3, 4, [(2, 1), (1, 0)])`?

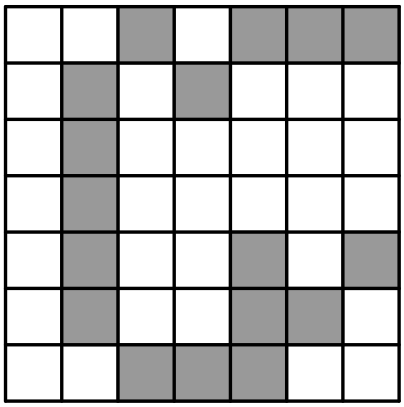
--

A new game is created with `5`, `5`, and a list `m` passed in as arguments, and it creates the following board layout, where grey squares represent mines and white squares represent open spaces:



What was the value of `m` that led to this board?

Consider the following board, where grey squares represent mines and white squares represent open spaces. Assuming that no squares had previously been revealed, if a player were to dig at location `(3, 0)`, how many squares in total would be revealed?



Enter an integer in the box:

On the same board, if a player were to dig at location `(2, 6)`, how many new squares would be revealed? Enter an integer in the box:

3.5) What You Need to Do

As usual, you will need to edit `lab.py` to complete this assignment. However, the nature of your changes to that file are somewhat different in this lab.

`lab.py` contains a nearly complete implementation of the *Mines* game. Your job in this part of the lab is twofold:

1. Complete the implementation of *Mines* in `lab.py` by writing the `render_2d_locations` and `render_2d_board` functions, and
2. Refactor the existing implementation of *Mines* to make it easier to read/understand.

Note that one of the test cases in `test.py` is designed to make sure that every function in `lab.py` has a docstring, so if you introduce any new helper functions, you should make sure to write an appropriate docstring for each!

Note also that the style check is not expected to pass until the whole lab is complete; so we should not expect it to be passing yet. However, you are more than welcome to come and talk with a staff member about your refactoring (what parts did you change and why?) to get feedback on this part as you are working! Refactoring the first part of the lab well will also likely make it easier to implement the next part of the lab.

3.5.1) Render

Your first task is to complete the definitions of the `render_2d_locations` and `render_2d_board`

functions in `lab.py`. The details of the desired behavior for each function are described in that function's docstring. In addition, each offers a few small tests (in the form of doctests), which you can use as basic checks.

Note that you can run the doctests without running `test.py` by running `python3 lab.py`. It is the lines at the bottom of `lab.py` that enable this behavior. You are also, of course, welcome to add additional doctests of your own (though you are not required to do so).

Here is how the output of `render_2d_board` might look (with `all_visible=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
```

Note that you can make a new line in Python with the character `"\n"`. For example

```
>>> print("Hello World\nHello World")
Hello World
Hello World
```

Once you have completed these functions, you can play the game in the browser by running `server_2d.py` and connecting to `http://localhost:6101`.

3.5.2) Refactor

The implementation provided in `lab.py` is entirely correct, but it is not written in a style that is particularly conducive to being read, understood, modified, or debugged. Your next task for this lab is to refactor the given code (i.e., to rewrite it so that it is expressed in a more concise, efficient, or understandable way).

In doing so, try to look for opportunities to avoid code repetition by introducing new loops and/or helper functions (among other things). Look also for opportunities to avoid redundant computation. There is a test

case in `test.py` that makes sure that each function/method in the implementation has a docstring. If you decide to add any new helper methods/functions while refactoring, please give them appropriate docstrings.

You may also find it helpful to read through the **section of the course readings on style** for suggestions.

If it is helpful for refactoring, you are welcome to add additional keys to the game dictionary; but it must still contain all of the keys mentioned above (and those values must behave as specified), and you will need to adjust the doctests so that they pass.

As you are refactoring the existing code, you can/should regularly run the doctests (and/or `test.py`) to make sure that your reorganization is not introducing errors into the existing code. You can also use the user interface to help find and fix bugs, and you are strongly encouraged to add some test cases of your own.

4) *HyperMines* (N-dimensional Mines)

Now that we've mastered 2-dimensional *Minesweeper*, we're going to add a little twist `:)`

Rather than playing on 2-dimensional boards, we're going to extend our *Mines* game to work on higher-dimensional boards. In this variant, which we'll call *HyperMines*, 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 (where n is the dimensionality of the space we're playing in).

4.1) Game Representation and Flow

4.1.1) Game State

Similarly to our 2-D representation, we will represent games as dictionaries, each of which should contain the following four keys:

- `'dimensions'`, the board's dimensions (an arbitrary tuple of positive integers)
- `'board'`, an N-dimensional array (implemented using nested lists) of integers and strings. In an game called `g`, `g['board'][x_0][...][x_k]` is `"."` if the square with coordinate $(x_0, \ldots, x_k)$ contains a mine.
- `'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'`.
- `'visible'`, an N-dimensional array (implemented using nested lists) of Booleans. In a game called `g`, `g['visible'][x_0][...][x_k]` indicates whether the contents of square $(x_0, \ldots, x_k)$ are visible to the player.

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]]],
    'visible': [[[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',
}
```

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

4.1.2) 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, it is **strongly encouraged**, before you dive into the code, to create some test cases of your own in `test.py`, to handle some more straightforward cases that are easier to reason about. For example, you may wish to include some tests on smaller boards. One suggestion would be to make one test for a 1-D game, one for a 2-D game, and one for a 3-D game. Each test could create a new game and perform at least 2 consecutive digs on that game, with different expected behaviors.

4.1.3) An Example Game

This section runs through an example game in 3D, 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.

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

The player tries digging at `(2,1,0)`, which reveals 1 tile.

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

... then at `(0,0,0)` which reveals 11 new tiles:

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

Emboldened by this success, the player then makes a fatal mistake and digs at `(1,2,0)`, revealing a mine:

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

```
state: defeat
visible:
    [[True, True], [True, True], [False, False]]
    [[True, True], [True, True], [True, False]]
    [[True, True], [True, True], [False, False]]
```

4.1.4) Check Yourself

To get a feel for working with an arbitrary number of dimensions, answer the following few questions about determining cells' neighbors. Note that each cell has up to $3^n - 1$ neighbors (not including itself or out of bound neighbors).

For these questions, it is okay to consider a cell to be its own neighbor, and 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 this lab, you might wish to consider a cell to be its own neighbor, or you may not.

> In a one-dimensional game with dimensions of `(10,)`, what are the neighbors of the coordinates `(5,)`? Enter a Python list of coordinates below:

> In a two-dimensional game with dimensions of `(10, 20)`, what are the neighbors of the coordinates `(5, 13)`? Enter a Python list of coordinates below:

> In a three-dimensional game with dimensions of `(10, 20, 3)`, what are the neighbors of the coordinates `(5, 13, 0)`? Enter a Python list of coordinates below:

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

4.2) Implementation

Now it's time to implement your *HyperMines* game! You should do so by filling in the skeletons of the `new_game_nd`, `dig_nd`, and `render_nd` functions in `lab.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-dimensional arrays represented as nested lists, and the questions above may also be helpful.

Note that, because of the sizes of some of these test cases, you may need to be a little bit careful about efficiency, as well.

Also note that on certain Windows installations of Python, you may encounter an error due to limitations on Python's stack size on Windows. If this happens to you, you can use the code submission box to check any tests that you cannot run locally.

How to Test Your Code

We provide three scripts to test and enjoy your code:

- `python3 lab.py` will run the doctests presented in the file.
- `python3 test.py` runs all the tests used for grading, as well as any additional test cases you put in `test.py`.
- `python3 server_nd.py` lets you play *HyperMines* in your browser! The server uses your code to compute consecutive game states and to render the results.

5) Refactoring Again

After having implemented *HyperMines*, you may notice that we now have a new opportunity to reorganize some of our two-dimensional code to take advantage of the (more-general) code we have written for the N-dimensional game. Try to reorganize your code for the 2-D game, where possible, to make use of these new structures (but make sure that it continues to function as expected!).

6) Code Submission

When you have tested your code sufficiently on your own machine, submit your modified `lab.py` using the `6.101-submit` script.

The following command should submit the lab, assuming that the last argument `/path/to/lab.py` is replaced by the location of your `lab.py` file:

```
$ 6.101-submit -a mines /path/to/lab.py
```

Running that script should submit your file to be checked. After submitting your file, information about the checking process can be found below:

> When this page was loaded, you had not yet made any submissions.
> If you make a submission, results should show up here automatically; or you may click here or reload the page to see updated results.