Lab 3: Six Double-Oh Mines

The questions below are due on Monday September 25, 2017; 10:00:00 PM.

You are not logged in.

If you are a current student, please Log In (https://6009.csail.mit.edu/fall17/labs/lab3?loginaction=redirect) for full access to the web site.
Note that this link will take you to an external site (`https://oidc.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 3.5 or later installed on your machine.

The following file contains code and other resources as a starting point for this lab: `lab3.zip` (https://6009.csail.mit.edu/fall17/lab_distribution.zip?path=%5B%22fall17%22%2C+%22labs%22%2C+%22lab3%22%5D)

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 (0.5 points)
- passing the test cases from `test.py` (1.5 points), and
- a brief "checkoff" conversation with a staff member to discuss your code (2 points).

2) Introduction

International *Mines* tournaments have declined lately (http://www.minesweeper.info/wiki/Tournaments), 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 (http://www.minesweeper.info/tournaments.html) one! In preparation for the tournament, several people have written implementations of the *Mines* game. In order to prepare *yourself*, you decide to look over their implementations in detail.

2.1) *6.00Mines*

6.00Mines 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. 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 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, any time a `0` is revealed (a square surrounded by no mines), 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!

3) Our Implementation of *6.00Mines*

3.1) Game state

The state of an ongoing *6.00Mines* game is represented as a dictionary with four keys:

- `dimensions` : a list 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 bomb, and it is a number indicating the number of neighboring bombs otherwise.
- `mask` , a 2-dimensional array (implemented using nested lists) of Booleans. `game['mask'][r][c]` indicates whether the contents of square (r, c) 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 *6.00Mines* game state:

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

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

3.2) Game Logic

The game is implemented via three functions:

- `new_game(nrows, ncols, bombs)`
- `dig(game, row, col)`
- `render(game, xray)`

Each of these functions is documented in detail in `lab.py` and described succinctly below. Notice how each function comes with a detailed docstring (https://www.python.org/dev/peps/pep-0257/) documenting what it does.

- `new_game(nrows, ncols, bombs)` creates a new game state.
- `dig(game, row, col)` implements the digging logic and updates the game state if necessary, returning the number of tiles revealed on this move.
- `render(game, xray)` renders the game into a 2D grid (for display).
- `render_ascii(game, xray)` renders a game state as ASCII art (https://en.wikipedia.org/wiki/ASCII_art).

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. To help understand what happens, calls to `dump(game)` are inserted after each state-modifying step.

Running new game produces a new game object as described above:

```
>>> 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]
state: ongoing
>>> render(game)
[['_', '_', '_', '_', '_', '_'],
 ['_', '_', '_', '_', '_', '_'],
 ['_', '_', '_', '_', '_', '_'],
 ['_', '_', '_', '_', '_', '_'],
 ['_', '_', '_', '_', '_', '_'],
 ['_', '_', '_', '_', '_', '_']]
```

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

```
>>> dig(game, 1, 0)
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]
state: ongoing
>>> render(game)
[[' ', ' ', '1', '_', '_', '_'],
 [' ', ' ', '2', '_', '_', '_'],
 ['1', '1', '2', '_', '_', '_'],
 ['_', '_', '_', '_', '_', '_'],
 ['_', '_', '_', '_', '_', '_'],
 ['_', '_', '_', '_', '_', '_']]
```

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

```
>>> dig(game, 5, 4)
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]
state: ongoing
>>> render(game, False)
[[' ', ' ', '1', '_', '_', '_'],
 [' ', ' ', '2', '_', '3', '1'],
 ['1', '1', '2', '_', '2', ' '],
 ['_', '1', '1', '1', '1', ' '],
 ['1', '1', ' ', ' ', ' ', ' '],
 [' ', ' ', ' ', ' ', ' ', ' ']]
```

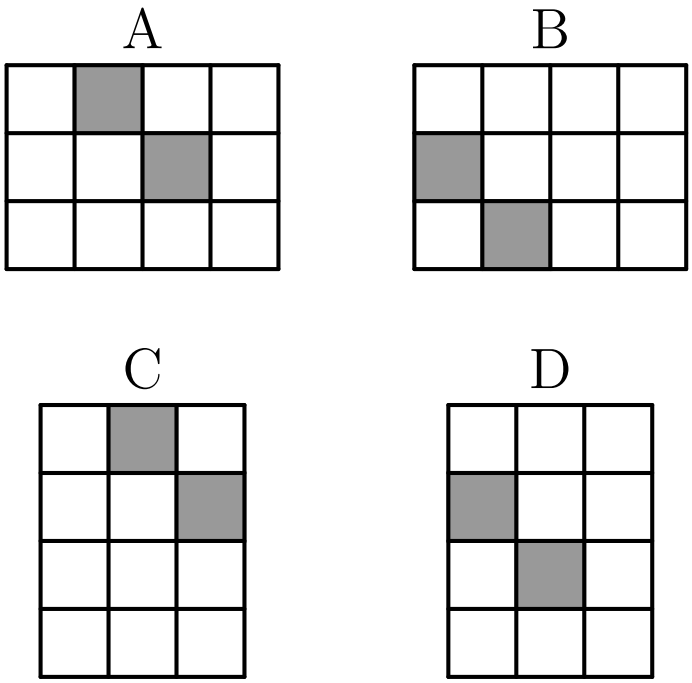
Emboldened by this success, the player then makes a fatal mistake and digs at `(0, 5)`, revealing a bomb:

```
>>> dig(game, 0, 5)
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]
state: defeat
>>> render(game)
[[' ', ' ', '1', '_', '_', '.'],
 [' ', ' ', '2', '_', '3', '1'],
 ['1', '1', '2', '_', '2', ' '],
 ['_', '1', '1', '1', '1', ' '],
 ['1', '1', ' ', ' ', ' ', ' '],
 [' ', ' ', ' ', ' ', ' ', ' ']]
```

3.4) Check Yourself

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

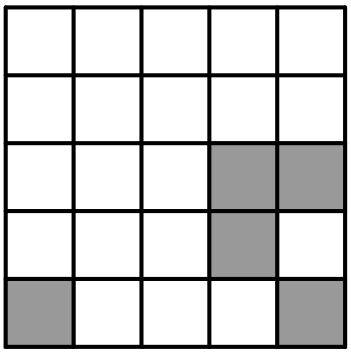
Consider the following four boards, where grey boxes represent bomb locations and white boxes represent open squares:



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

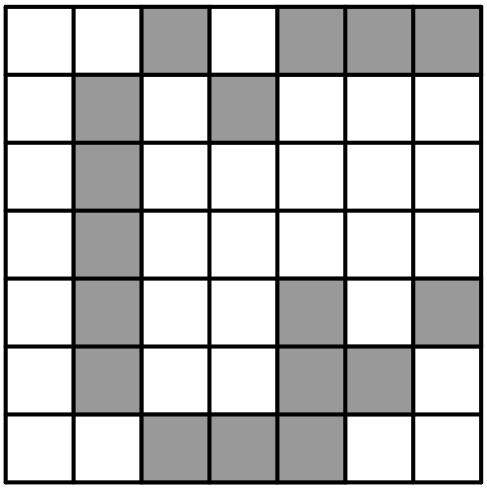
-- ▾

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



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

Consider the following board, where grey squares represent bombs 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:

4) What you need to do

As usual, you only 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, working implementation of the *6.00Mines* game. Your job in this lab is threefold:

1. Complete the implementation of *6.00Mines* in `lab.py` by writing the `render` and `render_ascii` functions;
2. Refactor the reference implementation of *6.00Mines* to make it easier to read/understand; and
3. Implement a set of functions that tests a *6.00Mines* implementation for correctness and returns a description of the bugs it contains (if any).

5) Render

Your first task is to complete the definitions of the `render` and `render_ascii` functions in `lab.py`. The details of the desired behavior for each function is described in that function's docstring. In addition, each offers a few small tests (in the form of doctests (https://docs.python.org/2/library/doctest.html)), which you can use as basic sanity 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.

Here is how the output of `render_ascii` might look (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
```

Once you have completed these functions, you can play your game in the browser by running `server.py` and connecting to `localhost:8000`.

6) Refactor

Although the implementation provided in `lab.py` is correct, 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 performs the same function but 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. Note that, after you make a change, your code should still pass all the same tests (since refactoring should not change the *behavior* of the code, merely its structure and appearance).

If you get stuck looking for opportunities for improvement, take a look at some of the suggestions we give, on this "hints" page (https://6009.csail.mit.edu/fall17/labs/lab3/hints).

7) Bug Hunt

Your final job (for this lab, at least!) has to do with debugging several implementations of the *6.00Mines* game. In the `resources` directory in the lab distribution, there are several implementations of the game, each of which may contain one or more bugs (or may be completely correct). Your goal in this last part of the lab is twofold:

- Examine the implementations in the `resources` directory in the lab distribution and try to enumerate all of the bugs in these implementations to the point where you'll be able to describe in detail both the nature of the bug(s) and a possible solution to each, but **you do not actually need to fix the bugs**. You

should be prepared to discuss your findings in detail with a staff member during a lab checkoff (you may wish to leave comments in those files near the bugs so that you remember them when the checkoff comes around).

- Complete the testing code near the bottom of `lab.py` to automate this process by programmatically testing whether a given implementation correctly implements the different behaviors described above.

Note that you can try playing the game with these implementations in `server.py`, which can help with finding bugs. If you add new implementations to the `resources` directory, they will also be available for you to use from the browser. You can switch between running with the rendering functions defined in the module and a fully working implementation of the rendering functions by clicking the button near the bottom of the page.

You may assume that the `render` and `render_ascii` functions are implemented correctly in each implementation, so there is no need to implement tests for them.

There are several behaviors we would like each implementation to perform correctly, and each has an associated method in the `TestMinesImplementation` class near the bottom of `lab.py`.

After looking over the implementations, you should fill in the definitions of the methods in the `TestMinesImplementation` class. Each test is described in detail in its docstring. The test should pass if that behavior is implemented correctly in the given implementation, and it should fail otherwise. Note that these behaviors should be considered *independently*; for example, a failure in an implementation's `new_game` method should not be taken into account when testing the `dig` implementation (i.e., tests for `dig` should test `dig` on well-formed inputs, regardless of the correctness of the other pieces of the implementation being tested).

You should also structure this code to avoid repetition where possible, and in such a way that each of the above tests can also be performed independently of the others.

If you are having trouble coming up with test cases, look at some of the hints on the "hints" page (https://6009.csail.mit.edu/fall17/labs/lab3/hints).

8) Auto-grader (unit tester)

As in the previous lab, we provide you with a `test.py` script to help you verify the correctness of your code. The script will call the required methods in `lab.py` and verify their output. Again, we encourage you to write your own test cases to further verify the correctness of your code and to help you diagnose any problems.

9) Code Submission

Select File `No file selected`

10) 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 `render` and `render_ascii`
- The changes you made to refactor `new_game`
- The changes you made to refactor `dig`
- The bugs you found in `mines1`

- The bugs you found in `mines2`
- The bugs you found in `mines3`
- The bugs you found in `mines4`
- The test cases you constructed related to `new_game`
- The test cases you constructed related to `dig`

10.1) Grade

You have not yet received this checkoff. When you have completed this checkoff, you will see a grade here.

```
\
/    /\__/\
\__=(  o_O )=
(________)
 |_ |_ |_ |_
```