Six Double-oh Mines

Submission to website: Monday, February 27, 10pm

Checkoff by LA/TA: Thursday, March 2, 10pm

This lab assumes you have Python 3.5 or newer installed on your machine. It is alright to use modules that are already imported for you in the template, but you should not add any imports to the code skeleton. Please use the Chromium or Firefox web browser.

Introduction

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 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.

6.00Mines

6.00Mines is played on a rectangular `n × m` board (where `n` indicates the number of rows and `m` the number of columns), 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, 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!

Our Implementation of *6.00Mines*

Game state

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

- `"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 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': [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]],
          'state': 'ongoing'}
```

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

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 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.

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.

```
>>> from lab import *
```

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', ' ', ' ', ' ', ' '],
 [' ', ' ', ' ', ' ', ' ', ' ']]
```

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).

Part 1: *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), 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` enable this behavior.

Here is how the output of `render_ascii` 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
```

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

Part 2: *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 appearance).

If you get stuck looking for opportunities for improvement, take a look at some of the suggestions in the `hints.pdf` file.

Part 3: *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 the first 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. You should be prepared to discuss your findings in detail with a staff member during a lab checkoff.
- Define a function `test_mines_implementation` that automates 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.

There are several behaviors we would like each implementation to perform correctly, and we'll give a name to each:

- the creation of a new board, including:
  - the dimensions should be initialized correctly (`'newgame_dimensions'`)
  - the board should be initialized correctly (`'newgame_board'`)
  - the mask should be initialized correctly (`'newgame_mask'`)
  - the game state should be initialized correctly (`'newgame_state'`)
- updating the board based on dig, including:
  - digging should modify the mask, not the game board (`'dig_mask'`)
  - digging should reveal the square that was dug (`'dig_reveal'`)
  - if a `0` is revealed during digging, all of its neighbors should automatically be dug as well (`'dig_neighbors'`)
  - dig should not do anything if performed on a game that is not ongoing (`'completed_dig_nop'`)
  - dig should not do anything if performed on a square that has already been dug (`'multiple_dig_nop'`)
  - digging should correctly report the number of tiles that were dug in all cases (`'dig_count'`)
  - digging a mine should result in the `'defeat'` state (`'defeat_state'`)
  - the game state should change to `'victory'` when there are no more safe squares to dig (`'victory_state'`)

You may assume that the `render` and `render_ascii` functions are implemented correctly in each implementation.

After looking over the implementations, you should implement the `test_mines_implementation` function. This function, which is described in detail in its docstring in the `lab.py` file, should test whether an implementation of the mines game correctly implements the desired behaviors, and it should return a description of which behaviors are implemented correctly and which are not.

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 or with thinking about how to structure this function, look at some of the hints in the `hints.pdf` file.

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. We will only use the same tests cases to auto-grade your work, except that there will be one additional test case on the server, which involves running your test suite on a perfectly-working implementation (so try to make your program as general as possible!).

Coding Points

This lab contains 21 test cases and 20 coding points. 5 of these coding points will be given based on your successful reorganization of the sample implementation from `lab.py`; 5 will be given for discussions of the various bugs in the alternative implementations in the `resources` directory; 5 will be given based on the coding style of your `render` and `render_ascii` functions; and 5 will be given based on test cases you use in your `test_mines_implementation` function.