Hints for Six Double-oh Mines

This lab is difficult, and a lot of things are left fairly open-ended. As such, this document contains some hints for parts 2 and 3 of the lab. You are welcome to make use of these hints as you see fit, but you should do so **only after having given those portions of the lab a decent effort** (you're likely to find that you are able to make more progress without the hints than you may have thought!).

Of course, if you are still having trouble after using this document, there are plenty of opportunities to get help from a human (tutorials, lab sessions, office hours, Piazza, etc.)!

Part 2: *Refactor*

> *"A designer knows he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away."*
> -Antoine de Saint-Exupéry

1. In general, we will try to follow the so-called **DRY** principle (**D**on't **R**epeat **Y**ourself). Basically, the suggestion is that multiple fragments of code should not describe redundant logic. Rather, that logic can often be refactored to make use of a variable, a loop, or a function to avoid re-writing the same code multiple times.
2. If you find yourself re-writing the same short expression over and over, that might be a good sign that you can store that in a variable as an intermediate result. If you find yourself copy/pasting a block of code to compute a result, that might be an opportunity to define a function and/or to use a looping structure.
3. Also be on the lookout for redundant code. Code that, for example, performs multiple checks for the same condition, or reassigns a variable to have the exact same value, or something of that nature, can often be removed without affecting the behavior of the program.
4. Nested conditionals can often be rewritten as a single conditional to save on indentation using `and` or `or`.

Hints for Some (but not all) Opportunities for Improvement

Below are some specific suggestions for areas in which the implementation in `lab.py` could be improved. You do not need to follow these guidelines, but if you are stuck looking for opportunities for improvement, you may find the following suggestions helpful. Note also that this does not contain suggestions related to *all* of the things that could be improved; just a few to get you started.

1. The code in `new_game` for computing the number of neighboring bombs is massive, and the same block is repeated multiple times with little modification between repetitions. Try to restructure that piece of the code to avoid this kind of repetition.
2. In `new_game`, the blocks of code related to creating `"mask"` and creating `"board"` are very similar: they each create a 2-d array of a particular size and fill it with appropriate values. Try to restructure this by making a generic "helper" function, and creating both the board and the mask using that function.
3. The `reveal_squares` function contains a deeply-nested combination of `for` and `if` statements. Try to reorganize this function to minimize the nesting of these statements.
4. Several blocks throughout the code attempt to find the neighbors of a particular cell (specifically, only those that are "in bounds") to perform some operation on them. Try to replace these blocks by defining a helper function to perform this operation, and replacing these blocks with calls to that function.
5. There are several blocks throughout the code that loop over the entire game structure to determine the new *state* of the game after an update. Try to restructure this into a single helper function, which can be used to test for the state of a game based on the board and mask.

Part 3: *Bug Hunt*

> *"There is always one more bug."*
> -The Law of Cybernetic Entomology

1. If you are having trouble thinking about how to structure the `test_mines_implementation` function, take a look at the code in `testing_structure.py`. If you want to, you are welcome to copy this outline into your `lab.py` to use as a starting point.

2. For each of the behaviors described, try to think of *at least one test case* (an input game and an action or sequence of actions) for which a correct implementation will succeed but an incorrect implementation will fail. Then write a piece of code that tries the implementation in question on that test case, and compares the result against the value you would expect. Note that in some cases, you may need to test both the return value of the function *and* the game representation, since many of the functions modify the game state directly instead of (or in addition to) returning a value.

3. The implementations provided in `resources` exhibit some (but not all) of these bugs. Take a look at those (both by reading the code and by playing with them in `server.py`) if you're looking for some inspiration.

4. Your code should handle the case where a function returns an incorrect result in response to valid input, but it should also handle the case where the function raises an exception in response to valid input.

5. You are welcome to use the known working implementation when formulating your tests. For some tests, it may be easier to hard-code the expected output, but for others, it may be easier to compare the results from running with your implementation vs. the "gold standard" working implementation.

   If you *do* test using the gold standard, keep in mind that many of these functions will *modify* the game board that they are given. As such, it will be important to make sure that all of your operations are performed on the board states you want them to be performed on.