Scripting Languages: Project 2

Deadline: November 22th 23:59, 2020

Project Guidelines

- This project is an individual project.
- Every task should have its own file named ”task1.py”, ”task2.py”...
- The program output should have exactly the same format as the provided sample output.
- If a specific aspect of the project is not working, you will still get a partial grade on the implemented parts of the project. In that case, it may be useful to add some comment statements on the problem and on the steps you tried to solve it.
- The project is corrected on 10 points. If you solve the bonus task 11, you can obtain a grade of 11/10. You can further raise that to 12/10 with the bonus task 12.

The assignment

In this project, you create a functional (but simple), terminal-based game. Minesweeper is a single-player game in which the player aims at clearing the cells in the map one by one without detonating any bombs. 10 bombs are randomly located in the 8 x 8 map. When a game starts, all cells are covered (marked as #), shown as follows.

```
    1 2 3 4 5 6 7 8
   ----------------
1 | # # # # # # # #
2 | # # # # # # # #
3 | # # # # # # # #
4 | # # # # # # # #
5 | # # # # # # # #
6 | # # # # # # # #
7 | # # # # # # # #
8 | # # # # # # # #
```

The player should enter two integers `row` and `column` to indicate which cell he or she wishes to uncover. The game should look as follows, player's input is marked blue.

```
    1 2 3 4 5 6 7 8
   ----------------
1 | # # # # # # # #
2 | # # # # # # # #
3 | # # # # # # # #
4 | # # # # # # # #
5 | # # # # # # # #
6 | # # # # # # # #
7 | # # # # # # # #
8 | # # # # # # # #
Uncover the cell at row: 6
                 column: 2

    1 2 3 4 5 6 7 8
   ----------------
1 | # # # # # # # #
2 | # # # # # # # #
3 | # # # # # # # #
4 | # # # # # # # #
5 | # # # # # # # #
6 | # . # # # # # #
7 | # # # # # # # #
8 | # # # # # # # #
Uncover the cell at row: 2
                 column: 5

    1 2 3 4 5 6 7 8
   ----------------
1 | # # # # # # # #
2 | # # # # . # # #
3 | # # # # # # # #
4 | # # # # # # # #
5 | # # # # # # # #
6 | # . # # # # # #
7 | # # # # # # # #
8 | # # # # # # # #
Uncover the cell at row: 3
                 column: 1

    1 2 3 4 5 6 7 8
   ----------------
1 | # # # # # # # #
2 | # # # # . # # #
```

```
3 | 2 # # # # # # #
4 | # # # # # # # #
5 | # # # # # # # #
6 | # . # # # # # #
7 | # # # # # # # #
8 | # # # # # # # #
Uncover the cell at row: 1
                 column: 7

    1 2 3 4 5 6 7 8
   ---------------
1 | # # # # # # 2 #
2 | # # # # . # # #
3 | 2 # # # # # # #
4 | # # # # # # # #
5 | # # # # # # # #
6 | # . # # # # # #
7 | # # # # # # # #
8 | # # # # # # # #
Uncover the cell at row: 3
                 column: 2

    1 2 3 4 5 6 7 8
   ---------------
1 | # # # # # # 2 *
2 | # * # # . # * #
3 | 2 * # # # # # #
4 | # # * # # # # #
5 | # # # * # # # #
6 | # . # # # # # #
7 | # # # # # * * *
8 | # * # # # # # #
Oh no! You lose!
Thank you for playing Minesweeper!
```

Task 1

Initialize two variables `empty_map` and `initial_map` that store internal representations of two maps. Take the approach of Battleship to build a similar internal representation. Each cell is a list of two elements `sign` and `uncovered` where `sign` is a character in {`1, 2, 3, 4, 5, 6, 7, 8, ., *`} and `uncovered` is a Boolean value `True` or `False`. When `sign` is `*`, a bomb is in the cell. When `sign` is `.`, no bomb is in the cell or in any neighboring cells. When `sign` is a number n, no bomb is in the cell, but exactly n neighboring cells contain a bomb. A cell's `uncovered` is originally set to `False`. When the player chooses to uncover a cell, its `uncovered` attribute is set to `True`. To store a map, you might need a nested list of three levels, i.e. a list of lists of lists: the two outer levels for the rows and columns, and the innermost level for storing `sign` and `uncovered`. You can assign `True` to all cells' `uncovered`.

The variable `empty_map` has a representation of an empty map that contains no bombs. It should correspond to

```
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
```

The variable `initial_map` has a representation of a map that contains 10 bombs, it should look like the following. Come up with your own map.

```
1 1 1 . . 1 2 *
2 * 2 . . 1 * 2
2 * 3 1 . 1 1 1
1 2 * 2 1 . . .
. 1 2 * 1 . . .
. . 1 1 2 2 3 2
1 1 1 . 1 * * *
1 * 1 . 1 2 3 2
```

Task 2

Write a function `set_bombs()` that returns a map with exactly 10 randomly distributed bombs. That is, `set_bombs()` can return different maps if called multiple times. All cells' `uncovered` flag is set to `False`. An example is as follows.

```
. . . . . . . *
. * . . . . * .
. * . . . . . .
. . * . . . . .
. . . * . . . .
. . . . . . . .
. . . . . * * *
. * . . . . . .
```

Task 3

Write a function `neighbors(row, column)` that takes two integers `row` and `column` between 0 and 7, indicating a position, and returns a list of lists, indicating all its neighboring cells' positions. Both `row` and `column` are assumed to be between 0 and 7. In the returned list, each position's row and column should be between 0 and 7. The order in the returned list does not matter. Notice that the length of the returned list should be either 3 (if the input position is at a corner), 5 (if the input position is next to a wall) or 8. For example, `neighbors(0, 0)` can return `[(1, 0), (1, 1), (0, 1)]` , `neighbors(4, 0)` can return `[(5,0), (3,0), (4,1), (3,1), (5,1)]`.

Task 4

Write a function `new_game()` that returns a new map with exactly 10 randomly distributed bombs and numbers of bombs nearby. It first calls `set_bombs()` to set the bombs, then loops over all cells in the map and uses `neighbors(row, column)` to calculate the numbers of bombs in the neighboring cells. All cells of the returned map are not uncovered. The map should look like the following.

```
1 1 1 . . 1 2 *
2 * 2 . . 1 * 2
2 * 3 1 . 1 1 1
1 2 * 2 1 . . .
. 1 2 * 1 . . .
. . 1 1 2 2 3 2
1 1 1 . 1 * * *
1 * 1 . 1 2 3 2
```

Task 5

Write a function `print_map(map)` that gets the representation of the current map as an argument and prints this map to the terminal. If a cell's `uncovered` flag is set to `True`, its `sign` is printed, otherwise the character `#` is printed, indicating the cell is not uncovered yet. When `print_map(map)` is called, the effect should be that we see the following on the screen.

```
    1 2 3 4 5 6 7 8
   ----------------
1 | # # 1 . . 1 2 #
2 | # # 2 . . 1 # #
3 | 2 # 3 1 . 1 1 1
4 | 1 2 # 2 1 . . .
5 | . 1 2 # 1 . . .
6 | . . 1 # 2 2 3 2
7 | 1 1 1 # # # # #
8 | # # # # # # # #
```

It is recommended (yet not necessary for this project) to use the following code in `print_map(map)` so that the map is always located nicely on the upper left of the terminal. The following code has the effect of clearing everything the terminal.

```
import os
os.system("cls" if os.name == "nt" else "clear")
```

Task 6

Write a function `get_input(map)` that gets a map and returns a list of two integers, indicating a cell's position. This function works as follows. It asks the player to enter and `row` and a `column`. If `row` or `column` is not between 1 and 8, or if the indicated cell is already uncovered, it asks the player to try again. This process repeats until it gets a desired position. Here is an example where the cell located at [1,8] is uncovered.

```
Uncover the cell at row: three
                 column: 1
Invalid row or column. Please try again.
Uncover the cell at row: 1
                 column: 8
The cell was already uncovered. Please try again.
Uncover the cell at row: 1
                 column: 11
Invalid row or column. Please try again.
Uncover the cell at row:
.
.
.
```

Task 7

Write a function `uncover(map, row, column)` that takes a map and two integers between 1 and 8, indicating the cell to be uncovered and returns a new map after uncovering some cells. You can assume that the cell indicated by `row` and `column` is valid, i.e. returned from `get_input(map)`.

The beginning of the assignment illustrates the behavior of `uncover(map, row, column)`. If the uncovered cell is a bomb, uncover all other cells that have a bomb. If the uncovered cell is not a bomb, uncover just the cell.

Task 8

Write a function `wins(map)` that takes a map and returns `True` if all non-bomb cells are uncovered and all 10 bomb cells are not uncovered. Otherwise, it returns `False`.

Task 9

Write a function `loses(map)` that takes a map and returns `True` if any bombs are uncovered. Otherwise, it returns `False`.

Task 10

Now write the `main` program for the game. This essentially consists of a loop, probably a while loop. The loop should have the following functionality: print the current map, ask a valid input from the player, uncover the cell indicated by the user, check whether the player wins or loses. If the player wins, print to the screen:

```
    1 2 3 4 5 6 7 8
   ----------------
1 | 1 1 1 . . 1 2 #
2 | 2 # 2 . . 1 # 2
3 | 2 # 3 1 . 1 1 1
4 | 1 2 # 2 1 . . .
5 | . 1 2 # 1 . . .
6 | . . 1 1 2 2 3 2
7 | 1 1 1 . 1 # # #
8 | 1 # 1 . 1 2 3 2
Congratulations! You win!
Thank you for playing Minesweeper!
```

If the player loses, print to the screen:

```
    1 2 3 4 5 6 7 8
   ----------------
1 | # # 1 . . 1 2 *
2 | # * 2 . . 1 * #
3 | 2 * 3 1 . 1 1 1
4 | 1 2 * 2 1 . . .
5 | . 1 2 * 1 . . .
6 | . . 1 # 2 2 3 2
7 | 1 1 1 # # * * *
8 | # * # # # # # #
Oh no! You lose!
Thank you for playing Minesweeper!
```

Then the program stops. Otherwise, it asks the player to enter a cell to uncover and performs the same actions.

Bonus task 11: A more fancy board layout

If you were able to do all assignments within the 8 hours that are intended for the project, you can try to change the layout of your game and make it look a bit more fancy. The map from task 4 could now look more like the following. The uncovered cells are now marked by a blank instead of `#`.

```
    1   2   3   4   5   6   7   8
  ---------------------------------
1 |   |   | 1 | . | . | 1 | 2 |   |
  ---------------------------------
2 |   |   | 2 | . | . | 1 |   |   |
  ---------------------------------
3 | 2 |   | 3 | 1 | . | 1 | 1 | 1 |
  ---------------------------------
4 | 1 | 2 |   | 2 | 1 | . | . | . |
  ---------------------------------
5 | . | 1 | 2 |   | 1 | . | . | . |
  ---------------------------------
6 | . | . | 1 |   | 2 | 2 | 3 | 2 |
  ---------------------------------
7 | 1 | 1 | 1 |   |   |   |   |   |
  ---------------------------------
8 |   |   |   |   |   |   |   |   |
  ---------------------------------
```

Bonus task 12: Extend the game!

If you were able to do all assignments within the 8 hours that are intended for the project, you can try to improve the gaming experience by extending `uncover(map, row, column)`. Recall that when a cell's `sign` is `.`, there is no bombs in any neighboring cells. This means that all the neighboring cells can be safely uncovered. Implement this functionality in `uncover(map, row, column)`. The function does not need to uncover neighbors of neighbors when some neighbor also contains a `.`.