Computer Science 1 — CSci 1100
Lab 7 — Sudoku

Lab Overview

This lab uses the game of Sudoku to investigate the use of logic, nested lists, and nested for loops. Please download the file `lab07_files.zip` from the Piazza site. This includes two Python programs, and several "games".

The Game of Sudoku

Sudoku is a popular logic puzzle, often called a "wordless crossword". There are many books and websites for Suduko. The following puzzle was taken from `http://www.websudoku.com`, where you can learn a bit more about the rules of the puzzle

1				2			3	7
	6				5	1	4	
	5						2	9
			9			4		
		4	1		3	7		
		1			4			
4	3						1	
	1	7	5				8	
2	8			4				6

In a Sudoku solution, each row, each col, and each 3x3 *block* has each of the numbers 1-9 appearing exactly one time. A Sudoku puzzle starts with some of squares having numbers, and there is generally only one way the remaining squares may be filled in legally. Sometimes finding this solution is easy. Other times it seems impossible.

Checkpoint 1: Representing and Building the Board

We will represent the Sudoku board as a list of lists of single character strings. Start by looking at the code in `check1.py`. It has an example board, stored in the variable `bd`. Each `'.'` is an empty location on the Sudoku board. The code prints the length of `bd`, the length of the 0-th list stored in `bd`, the entry in row 0, column 0, and finally the entry in row 8, column 8. Go ahead and run this code, and make sure you understand the output you are seeing.

Part 1a: Write nested `for` loops to print the row, column and number for each non-empty location in `bd`. The first five lines of output should be something like

```
(0,0) has 1
```

```
(0,4) has 2
(0,7) has 3
(0,8) has 7
(1,1) has 6
```

Part 1b: In the second half of Checkpoint 1, you will get started working with the main code for this lab. Please open up Python file `sudoku_start.py`. This is where you will write most of your code. The main code asks the user for the name of the file containing the initial Sudoku board to work with. It then calls the function to read the board and another to print it. Your job in the rest of this checkpoint is to write and test `read_board`. You can use the parameter `block_size` if you would like, it is merely there for convenience.

Take a look at the file `board.txt` and the format of each line, which has `.` and numbers separated by spaces. From here you should be able to apply techniques we've already used to convert each input line to a list, append it to the board list, and return the Sudoku board at the end. Here is the expected output:

```
Enter the file for Sudoku moves ==> board.txt
-------------------------
| 1 . . | . 2 . | . 3 7 |
| . 6 . | . . 5 | 1 4 . |
| . 5 . | . . . | . 2 9 |
-------------------------
| . . . | 9 . . | 4 . . |
| . . 4 | 1 . 3 | 7 . . |
| . . 1 | . . 4 | . . . |
-------------------------
| 4 3 . | . . . | . 1 . |
| . 1 7 | 5 . . | . 8 . |
| 2 8 . | . 4 . | . . 6 |
-------------------------
```

To complete Checkpoint 1: Show your code and output from both 1a and 1b. Test all three Sudoku initial examples provided.

Checkpoint 2: Assigning Numbers to Cells

Recall that the completed Sudoku board has no repeated numbers in a row, in a column, or in any 3x3 block. In Checkpoint 2, your code will ask the user of the program to enter a row (starting at index 0), a column (starting at index 0), and a number. It will then call function `ok_to_add`, which you must write, to check to see if the number can safely be added to that particular row and column based on the current contents of the Sudoku board. To start Checkpoint 2, removing the commenting around the Checkpoint 2 code in the main code of `sudoku_start.py`. Note that this code also allows users to erase an entry in the board by entering a `.` instead of a digit.

The actual work of Checkpoint 2 is the function `ok_to_add`. It has quite a few checks. For example, if the user asks to put a 5 in row 4, column 6, the function should

- check row 4 to see if there is already a 5,

- check column 6 to see if there is already a 5, and
- check the 3x3 block starting at row 3, column 6 to see if there is already a 5.

For the Sudoku board in `board.txt`, `ok_to_add` should return `False` because there is already a 5 in the 3x3 block. `ok_to_add` should also check to see if the row index, the column index and the number are also legal values — remember that users make typing mistakes!

The function `ok_to_add` will have separate for loops to check the row, to check the column, and to check the 3x3 block. The latter is the hardest because you need to find the lowest row and column indices in the block and then write nested for loops to check all 9 locations. The code should return `False` immediately when it finds a mistake, but it should wait until all checking is complete before returning `True`.

Note that when `ok_to_add` returns `True`, it **does not** mean that the placement of the number is actually correct. It is really just a sanity check.

To complete Checkpoint 2, show a TA or mentor your code, and the results of testing a full range of possible mistakes.

Checkpoint 3: Sudoku Verifier

We would like to make a Sudoku solution verifier by utilizing the `ok_to_add` function. The verification of a correct solution to a Sudoku board can be broken down into two steps:

- Verify that there are no empty (".") spaces in the board.
- Verify that for every number in the solution, it is `ok_to_add` in its current position. For example, if there is a 3 in position (0,1), we want to make sure that `ok_to_add(0,1,3,board)`.

If all numbers are not empty and `ok_to_add` in their current position, then the solution is valid. Your job in Checkpoint 3 is to complete the function `verify_board`. To verify your solution, you can use the files `solved.txt`, `unsolved1.txt` and `unsolved2.txt`.

To complete Checkpoint 3, show a TA or mentor your code and demonstrate the correctness of the function `verify_board`.

Extra Challenges

Here are two extra challenges for those of you who are really ambitious:

1. In theory, a Sudoku can be any NxN block grid, where N $>$ 0. The most common Sudoku is a 3x3 block board, but there do exist 4x4 and 5x5 Sudoku puzzles. As an extra challenge, rewrite Checkpoints 1b, 2 and 3 so that they support any sized Sudoku boards. You might start by modifying `print_board`, and include a parameter for the `block_size` (in an ordinary 9x9 Sudoku, `block_size` is 3). You can test your code with the file `bigger.txt`, which is a 4x4 block Sudoku.
2. If you are really ambitious, write an automatic solver. For each cell, keep a list of the possible digits that can be assigned to that cell. When there is only one possibility,

say the digit 3, then 3 must be removed from the list of possible digits for the other cells in the row, in the column and in the block. The trick is to keep a list of which cells have been reduced to one entry.

This solution will not allow you to automatically solve all Sudoku puzzles — multi-cell reasoning is needed for this — but it should be able to solve almost anything labeled "medium" or easier.

Unfortunately, no extra credit is being offered for implementing this.