Computer Science 1 — CSci 1100
Lab 6 — Sudoku

Lab Overview

This lab uses the game of Sudoku to investigate the use of logic, nested lists, and nested loops. Please download the file `lab6files.zip` from the Submitty Course Materials section. This includes a Python utility for reading files, and several "games". You will use the utility function only in the final checkpoint. Until then, you will work with a fixed board. To get full credit for the lab you must show your progress to your TA/Mentor before leaving your lab session.

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 exactly one time. A Sudoku puzzle starts with some of the 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 0: Double loops

Before you start this lab, we will do a small exercise that will allow you to complete the rest of the lab much faster. This checkpoint is meant for an exercise and it will not be checked off.

The idea is that you will need to write a few loops that simply generate pairs of values corresponding to the indices of specific blocks in the Sudoku board. Once you have these loops in place, you can easily use them as indices for your code later. The best idea is to write them in a separate file. Try doing these both with `while` and `for` loops.

1. Write a loop to output the digits from 0 up to (and including) 8, all on one line.

   ```
   0 1 2 3 4 5 6 7 8
   ```

   As a hint, create an empty string called `line`, and then write a `for` loop that appends a string for each digit to `line`. After the `for` loop ends, print `line`.

2. Write a loop to generate pairs of values from 0 up to 8 (basically, for each value above, you will generate a second value between 0-8). As a special challenge, we added a space and line to separate each 3x3 block.

   ```
   0,0 0,1 0,2  0,3 0,4 0,5  0,6 0,7 0,8
   1,0 1,1 1,2  1,3 1,4 1,5  1,6 1,7 1,8
   2,0 2,1 2,2  2,3 2,4 2,5  2,6 2,7 2,8

   3,0 3,1 3,2  3,3 3,4 3,5  3,6 3,7 3,8
   4,0 4,1 4,2  4,3 4,4 4,5  4,6 4,7 4,8
   5,0 5,1 5,2  5,3 5,4 5,5  5,6 5,7 5,8

   6,0 6,1 6,2  6,3 6,4 6,5  6,6 6,7 6,8
   7,0 7,1 7,2  7,3 7,4 7,5  7,6 7,7 7,8
   8,0 8,1 8,2  8,3 8,4 8,5  8,6 8,7 8,8
   ```

   These will serve as the indices for the Sudoku board entries.

3. Write a loop to generate all the items in a given row, say row=2.

   ```
   2,0 2,1 2,2 2,3 2,4 2,5 2,6 2,7 2,8
   ```

4. Write a loop generate all items in a single column, say column=5.

   ```
   0,5 1,5 2,5 3,5 4,5 5,5 6,5 7,5 8,5
   ```

5. Finally, write a loop to generate the valid indices for the first `3x3` piece of the board.

   ```
   0,0 0,1 0,2
   1,0 1,1 1,2
   2,0 2,1 2,2
   ```

   Think about how you would modify this to generate the other 3x3 blocks. What are the starting and end indices?

You can see the patterns of how we can write these loops. During the rest of the lab, you can refer back to these examples as guidelines.

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.

Write nested `for` or `while` loops to print the whole board on the screen. You will first go through each row with one loop, then for each row, you will go through each column using a second loop (see index range 2 from Checkpoint 0). Print each item with space on both sides, and a | after every third item and third row. Remember, you have exactly 9 rows and 9 columns.

Here is the expected output:

```
-------------------------
| 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 |
-------------------------
```

Hint. Double loops can be difficult, so we recommend you start slowly and add complexity. This will also help you in the other parts. Start from your Checkpoint 0, step 2 code, but where you printed the indices, instead print the entry from the board `bd`. So if your loop variables are `i` and `j`, where you would be adding `str(i)+','+str(j)` to your output, instead add `bd[i][j]`.

Now, where you add the extra blank lines between every third row, add a line of twenty-five `'-'`s instead.

Finally, add a vertical bar `'|'` in place of the `' '` after every third column.

To complete Checkpoint 1: Show a TA or Mentor your code and output once you are finished.

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. You will then either tell the user, `This number cannot be added`, or if it can be added, change the board and reprint it.

To start Checkpoint 2, copy and paste your code from Checkpoint 1.

The actual work of Checkpoint 2 is the function `ok_to_add`. It has quite a few checks you will need to write. For example, if the user asks to put a 2 in row 1, column 8, the function should

- check if the row (1), column (8) and the value (2) are valid,

- check if row 1 contains a 2 already,

- check if column 8 contains a 2 already, and

- check if the 3x3 block starting at row 0, column 6 contains a 2 already.

What about the location (1,8)? Since we will be replacing that value if the check is valid, we want to ignore whatever is in that space. An easy way to do this is just to temporarily replace whatever is in row 1 column 8 with the empty space character `'.'`, running our checks, and then restoring the value before we return `True` or `False`.

For the Sudoku board from Checkpoint1, `ok_to_add` should return `False` because there is already a 2 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 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 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`. Either way, remember to restore the value at (1,8) first. Your Checkpoint 0 code for steps 3, 4 and 5 will help you write your loops.

Note that when `ok_to_add` returns `True`, it **does not** mean that the placement of the number is actually correct. It just means that there are not currently any conflicts with placing it in the chosen location.

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

Please come to lab for the last checkpoint.