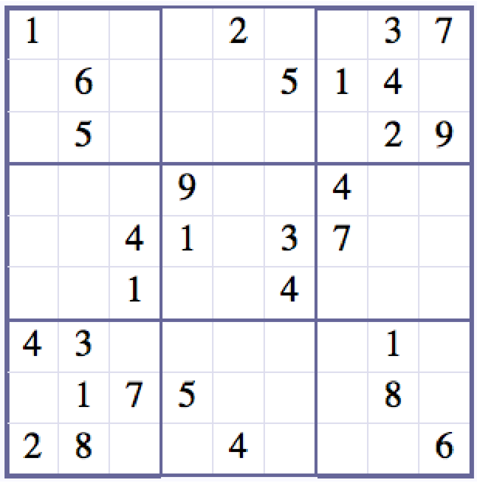
Computer Science 1 — CSci 1100
Lab 6 — Sudoku

Lab Overview

This lab uses the game of Sudoku to investigate the use of logic, nested lists, and nested loops. Please download the file `lab06_files.zip` from the Piazza site. 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. Of course, given we will cover files in Lecture 13, you can easily write this utility function yourself.

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 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 generates pairs of values of different types. 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. Now, we will use these to actually solve the lab.

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.

First, read each row as a list, and print each list on a single line. This is doable with a single loop.

Now, add the second loop for formatting each row. Go through each item in the row with a second loop, and construct a string containing your whole line. For each item, you will append a space before and after the item, and `|` at the beginning and end. Once done, print this string.

Now that you are printing reasonable lines, figure out how to add the `|` after every third column in the row.

Finally, add the code to print the line of hyphens (`-`) as needed. Always do things in small steps, and add complexity.

To complete Checkpoint 1: Show 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 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)? Well, you need to check that there is nothing there currently. But, it is better to move this check to outside of this function. This way, we can use this function for multiple purposes.

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`.

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

You are going to make a few changes to your code from Checkpoint 2 to complete this last checkpoint. First, you will use the utility given to you to read a board from file. Import `lab06_util`, prompt the user for a file name and read the board from the file using the `read_sudoku` function.

Next, you will write a Sudoku solution verifier by using 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 locations have a number and if `ok_to_add` is true for each position, then the solution is valid. Your job in Checkpoint 3 is to write the function `verify_board`. To verify your solution, you can use the files `solved.txt`, `unsolved1.txt` and `unsolved2.txt`.

To complete Checkpoint 3, combine all your code into a while loop that does the following.

```
ask user file name
read board from file
while (board is not solved according to verify_board):
    ask user for input to the puzzle
    if ok_to_add
         add value
    print the board
```

When you have tested your code properly, show it to a TA or mentor.

Extra Challenges

Here are two extra challenges for those of you who are 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 except for the satisfaction of great accomplishment!