COSC 130 – Project 03 Instructions

Introduction

The goal of this project is to develop an automated sudoku solver. If you are unfamiliar with sudoku puzzles, you can find a description of them by clicking on the following link: Rules for Sudoku

General Instructions

Create a Python script file named **`Sudoku.py`** and a Jupyter notebook file named **`Project_03_YourLastName.ipynb`**. You will use the script to define your functions and the notebook will be used to load the script and to test your functions.

Please download the files **`puzzle01.txt`**, **`puzzle02.txt`**, **`puzzle03.txt`**, and **`puzzle04.txt`**, storing these in the same directory as your script file and notebook file. It is important that these files are all in the same directory. Otherwise, your code will not run correctly when I run it.

You will represent a given state for a sudoku puzzle as a list of nine lists. Each of the 9 sub-lists will contain 9 elements, and will represent a single row in the puzzle. Empty cells will be indicated by storing a value of 0 in the location representing that cell. As an example, the list displayed on the left below would represent the puzzle displayed on the right.

```
puzzle = [
    [8, 0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 3, 6, 0, 0, 0, 0, 0],
    [0, 7, 0, 0, 9, 0, 2, 0, 0],
    [0, 5, 0, 0, 0, 7, 0, 0, 0],
    [0, 0, 0, 0, 4, 5, 7, 0, 0],
    [0, 0, 0, 1, 0, 0, 0, 3, 0],
    [0, 0, 1, 0, 0, 0, 0, 6, 8],
    [0, 0, 8, 5, 0, 0, 0, 1, 0],
    [0, 9, 0, 0, 0, 0, 4, 0, 0]
]
```

```
+-------+-------+-------+
| 8 . . | . . . | . . . |
| . . 3 | 6 . . | . . . |
| . 7 . | . 9 . | 2 . . |
+-------+-------+-------+
| . 5 . | . . 7 | . . . |
| . . . | . 4 5 | 7 . . |
| . . . | 1 . . | . 3 . |
+-------+-------+-------+
| . . 1 | . . . | . 6 8 |
| . . 8 | 5 . . | . 1 . |
| . 9 . | . . . | 4 . . |
+-------+-------+-------+
```

Instructions for the script file begin on the next page.

Instructions for the Script File

Define functions with the following names: **`load_puzzle()`**, **`display_puzzle()`**, **`get_next()`**, **`get_options()`**, **`copy_puzzle()`**,and **`solve()`**. Descriptions of each of these functions are provided below.

`load_puzzle()`

This function should accept a single parameter named **`path`**. This parameter is expected to be a string indicating the path to a text file containing the initial state of a sudoku puzzle. The file will be expected to contain 9 lines of text, each of which contains 9 numbers separated by spaces. The function should open the file and process the lines within. The function should return a list of lists, with each sub-list containing the 9 values on any one line of the file, stored as integers.

The function should complete the desired task by performing the following steps:

1. Use **`open()`** and **`read()`** to read in the contents of the file indicated by **`path`** as a string.
2. Split the string from the previous step on the newline character, storing the result in a list named **`lines`**.
3. Create an empty list named **`puzzle`**.
4. Loop over the elements of **lines**. Perform the following steps for each string in **lines**:
    a. Split the string on spaces, storing the result in a list named **`token_strings`**.
    b. Create a list named **`token_ints`** that contains the values from **`token_strings`**, but coerced to integers. You can do this with a loop or a list comprehension.
    c. Append the list **`token_ints`** to **`puzzle`**.
5. Return **`puzzle`**.

`display_puzzle()`

This function should accept a single parameter named **`puzzle`**. This parameter is expected to be a list of lists representing a puzzle state. The function should print the puzzle state using the format displayed above on the right.

The function should complete the desired task by performing the following steps:

1. Loop over the elements **`puzzle`**. Fir each list in **`puzzle`**, perform the following steps:
    a. If considering the list at index 0, 3, or 6, print the following string: **`'+-------+-------+-------+'`**
    b. Create a string named **`row`** that is initially set equal to **`'| '`**. We will concatenate values to this string to build up the desired output for the current row.
    c. Loop over the elements of the current list, which will be int. For each int value, do the following:
        i. If the current value is zero, concatenate the string **`'. '`** to **`row`**.
        ii. Otherwise, coerce the value to a string, and concatenate that and a space to **`row`**.
        iii. If considering the element at index 2, 5, or 8, concatenate the string **`'| '`** to **`row`**.
    d. After the inner loop finishes, print **`row`**.
2. After the outer loop finishes, print the bottom border of the puzzle: **`'+-------+-------+-------+'`**

This function does not need to return a value.

`get_next()`

This function should accept two parameters named **`row`** and **`col`**. These parameters are expected to be integers 0 – 8 representing the location of a specific cell. The function should return two integers, which should be the row and column indices for the next cell in the puzzle, assuming we are moving through the rows left-to-right and top-to-bottom.

- If **`col`** is less than 8, then we are not at the last column and need to advance one more column. The function should return **`row`** and **`col+1.`**
- If **`col`** is equal to 8, and **`row`** is less than 8, the we are in the last column of a row, but are not at the last row. To advance to the next row, the function should return **`row+1`** and **`0`**.
- If **`col`** and **`row`** are both equal to 8, then we are at the last row and the last column and can advance no further. The function should return **`None`** and **`None`**.

`copy_puzzle()`

This function should accept a single parameter named **`puzzle`**, which is intended to represent a puzzle state. The function should create a deep copy of the list **`puzzle`**.

To accomplish this, the function should perform the following steps:

1. Create an empty list named **`new_puzzle`**.
2. Loop over the items in **`puzzle`**, which are all lists. For each sub-list, create a copy using the **`copy()`** method. Append the new copy to **`new_puzzle`**.
3. Return **`new_puzzle`**.

`get_options()`

This function should accept three parameters named **`puzzle`**, **`row`**, and **`col`**. The parameters **`row`** and **`col`** should be integers indicating the position of a cell in the puzzle represented by **`puzzle`**. If the cell contains a value, then the function should return **None**. If the cell is empty, then the function should return a list of valid numbers that could be placed in that cell. It should do this by scanning the row, column, and 3x3 block that contain the cell, eliminating from the possible options any values that appear in those collections of cells.

For example, if **puzzle** is the list provided in the first page of these instructions, then:

- **`get_options(puzzle, 2, 4)`** should return **None**, since the cell in row 2, column 4 contains a 9.
- **`get_options(puzzle, 2, 5)`** should return **`[1, 3, 4, 8]`**.
    - 2, 7, and 9 are NOT valid options since they appear in the same row as this cell.
    - 5 and 7 are NOT valid options since they appear in the same column as this cell.
    - 6 and 9 are NOT valid options since they appear in the same block as this cell.

The steps below explain one way to accomplish the desired tasks:

1. Check to see if the cell of **`puzzle`** indicated by the values **`row`** and **`col`** contains a value greater than 0. If it does, then return **None**.
2. Create an empty list named **`used`**. We will fill this list with the non-zero numbers that are contained the the row, column, and 3x3 block that contain the indicated cell.
3. Loop over the indicated row of **`puzzle`**. Append each non-zero value in that row to **`used`**.
4. Loop over the indicated column of **`puzzle`**. Append each non-zero value in that column to **`used`**.
5. Use nested loops to scan over the block containing the current cell. Append each non-zero value in that column to **`used`**. To scan over the desired block, you will have to identify the index of the row and column that start the block. For example:
    - If **`row`** is 0, 1, or 2, then the first row in the block is at index 0.
    - If **`row`** is 3, 4, or 5, then the first row in the block is at index 3.
    - If **`row`** is 6, 7, or 8, then the first row in the block is at index 6.

    Similar rules fold for the starting column. You can us the following code to calculate these starting values:

    ```
    start_row = 3*int(row/3)
    start_col = 3*int(col/3)
    ```
6. After completing steps 2 – 5, the list **`used`** will contain all of the **invalid** options. We now wish to identify the **valid** options. Create an empty list named **`options`**.
7. Loop over the digits 1 – 9. Check to see if each digit appears in **`used`**. If it does not, then append it to **`options`**.
8. Return **`options`**.

`solve()`

This function should accept three parameters named **`puzzle`**, **`row`**, and **`col`**. The parameter **`puzzle`** is expected to be a list representing a puzzle state, while **`row`** and **`col`** are expected to be integers indicating the position of a specific cell in the puzzle. The default values of **`row`** and **`col`** should be set to 0, indicating that the function will begin the serach for a solution by looking at the upper-left cell.

The function should return a list of lists representing the solved state of the puzzle, if one exists. If no solution exists, then the function should return **`None`**.

The function should find the solution to the puzzle recursively by following the steps below:

1. Check to see if the cell indicated by the parameters **`row`** and **`col`** is blank (which would be indicated by a value of 0 stored at the appropriate location in **`puzzle`**). If this cell is **not** blank, then we will move to the next cell, if one exists. In this case, perform the following steps:
    a. Use **`get_next()`** to obtain the position of the next cell. Store the resulting values in **`next_row`** and **`next_col`**.
    b. Check to see if **`next_row`** is **`None`**. If so, then we are at the last cell, and will have finished construction the solution. Return **`puzzle`**, which will store the completed solution.
    c. If **`next_row`** is not **`None`**, then we will move on to the next cell. Call **`solve()`** again, passing it **`puzzle`**, **`next_row`**, and **`next_col.`**

2. If the current cell is blank, then we will identify the possible values that could be placed within this cell. Call **`get_options()`**, passing it **`puzzle`** and the location of the current cell. Store the results in a list named **`options`**.

3. If **`options`** is empty, then there are no value digits that could be placed in this cell. Return **`None`** to indicate that the current puzzle state is not solvable. This will end this branch of the recursion, causing the function to backtrack and consider different values for previously filled-in cells.

4. If **`options`** is not empty, then loop over the digits in this list. Perform the following steps for each such value:
    a. Use **`copy_puzzle()`** to create a new copy of the puzzle, named **`new_puzzle`**.
    b. Use **`row`** and **`col`** to set the entry of **`new_puzzle`** corresponding to the current cell to the current value of **`options`** that is being considered. We are filling in a valid number in this cell and will check to see if this leads to a solution.
    c. Call **`solve()`**, passing it **`new_puzzle`**, **`row`**, and **`col`**. Store the result in a variable named **`result`**. This is the step in which we attempt to see if the new puzzle state leads to a solution. Note that if a solution was found, then **`result`** will contain that solution. If no possible solution can be found using the current state, then result will contain **`None`**.
    d. If **`result`** is not equal to **`None`**, then this means that a solution was eventually found. Return **`result`**, which should store the solved puzzle state.
    e. If **`result`** is equal to **`None`**, then no additional action should be taken. In this case, the loop will continue executing, and the function will try out different values for the current cell.

Instructions for the Notebook

Any set of instructions you see in this document with an orange bar to the left will indicate a place where you should create a markdown cell. If no instructions are provided regarding formatting, then the text should be unformatted.

Any set of instructions you see with a blue bar to the left will provide instructions for creating a single code cell.

Assignment Header

Create a markdown cell with a level 1 header that reads: "COSC 130 – Project 03". Add your name below that in bold.

Introduction

Create a markdown cell with a level 2 header that reads "Introduction". Add unformatted text explaining the purpose of this project. You may paraphrase the statement in the first paragraph of this document. Explain that you will be by running the script in which you have created the functions that you will be using.

We will start by running the script.

Use the Magic command **`%run -i Sudoku.py`** to run the contents of your script.

Puzzle 1

Create a markdown cell with a level 2 header that reads "**Puzzle 1**". Add unformatted text explaining that you will test your function on the puzzle whose initial state is stored in the file **`puzzle01.txt`**, and that you will begin by loading the puzzle from the file.

Use **`load_puzzle()`** to load the contents of **`puzzle01.txt`** into a list named **`puzzle01`**. Use **`display_puzzle()`** to print the puzzle.

Add a markdown cell explaining that you will now attempt to find a solution to the puzzle.

Use **`solve()`** to attempt to find the solution of the puzzle stored in **`puzzle01`**. If a solution exists, then use **`display_puzzle()`** to print the solution. Otherwise, print the text "`No solution exists for this puzzle.`"

Puzzle 2, Puzzle 3, and Puzzle 4

Repeat the steps detailed for Puzzle 1, but using the information stored in the files **`puzzle02.txt`**, **`puzzle03.txt`**, and **`puzzle04.txt`**.

Submission Instructions

When you are done, click **Kernel > Restart and Run All**. If any cell produces an error, then manually run every cell after that one, in order. Save your notebook, and then export the notebook as an HTML file. Upload the HTML file to Canvas and upload the IPYNB file to CoCalc, placing the file into the folder **Projects/Project 03**.