Group Project: S U D O K U

FINAL PROJECT WRITE-UP

Group Members: Alex Malave and Lauren Kranis

GOAL

The goal of our project was to create a computer-generated Sudoku game. We expected to be able to learn the basics of a module named Tkinter so as to provide users with a neat interface through which to use our Sudoku code.We were able to implement it successfully. We also expected to be able to generate puzzles from pre-written files which are formatted a certain way (9 lines, each with 9 numbers), which we also successfully implemented.

CODE DESCRIPTIONS

The *sudoku.py* file contains three classes: **PuzzleBoard**, **SudokuInstance**, and **PuzzleGui**. The **PuzzleBoard** class creates the board list. The constructor reads a file line by line and appends the numbers to the board variable. We designed the files in a way such that each file contains nine lines, each containing nine numbers. The lines represent horizontal rows on the puzzle.

The **SudokuInstance** class checks the state of the board. It contains a constructor and six methods. This class also checks to see if the user has completed the puzzle correctly.

The method **checkEachSquare** creates a list of integers one through nine, and takes a list, *numList*, as a parameter. The method returns *True*, a boolean, if the list contains the numbers 1-9 without repeats, and *False* otherwise.

The **checkRow**, **checkColumn**, and **checkBox** methods call **checkEachSquare** in order to determine if the given row, column or box is correct, respectively. How they go about sending a list to **checkEachSquare** is a bit different, though.

checkRow is simple: its only parameter is *rowIdx*, which it uses to index the correct list within the puzzle list. That list is then passed on to **checkEachSquare** as is.

checkColumn also only takes one parameter, *columnIdx*, which it uses to index the corresponding value within the row lists (within the puzzle list). The method then creates a list out of the values it extracts, and passes that list on to **checkEachSquare**.

checkBox takes two parameters: *rowIdx* and *columnIdx*. Using the two parameters, the boxes within the puzzle (3x3 grids) are treated as indexes in their own right (0-2 columns, and 0-2 rows). Similarly to **checkColumn**, a list is created out of the values extracted from the 3x3 "grid," which is then passed on to **checkEachSquare**.

Finally, **checkIfSolved** determines if the puzzle is in a solved state or not. It checks all columns, rows and boxes (using **checkRow**, **checkColumn**, and **checkBox**, respectively) to see if each one contains the numbers one through nine with no repeats. If the methods all return *True* then the user has won.

The **PuzzleGui** class draws the puzzle, and binds keys in order to make user input possible. It contains nine methods, which are each responsible for different aspects of the user interface. The constructor

of the class and the method **__initUI()** set up instance variables, create the frame, create the canvas, draw the puzzle/grid, and bind events to key inputs.

The method **drawGrid** takes no parameters and draws a grid that's divided with purple lines into 3x3 squares. It first draws the vertical lines, and then horizontal lines, using the *marginSize* and *cellWidth* variables defined at the beginning of the file.

The method **drawPuzzle** takes no parameters and creates a cell for each of the numbers within the lists. This method is called every time a change is made to the *self.currentPuzzle* instance variable. If numbers were in the original puzzle (the unfinished puzzle), then they are colored black; otherwise, the numbers are colored gray. If the numbers are 0, the cells are left blank.

The method **clearAnswers** takes no parameters and essentially clears all user-inputted numbers on the board. If the user has already beaten the puzzle, this button also allows them to solve it again.

The method **cellClicked** takes one argument, *arg*, which is provided by the event that this method is triggered by. *arg* is expected to have x and y values, which represent the coordinates of the mouse at the time that the event was fired. The cell is highlighted using the method **drawCursor** if the coordinates of the mouse are within the bounds of the puzzle grid, and if the value of the cell is not from the original, unsolved puzzle.

The **drawCursor** method takes no arguments and draws a box outside of a cell, so that it's more clearly visible and so as to enable editing its value. If the cell is already selected, the cursor is removed.

The method **keyPressed** takes one argument, arg, which is provided by the event that this method is triggered by, similar to **cellClicked**. The event is expected to return the key pressed by the user. If the key value is a number and is in the range 0-9, the text of the selected cell is made to match inputted key. Choosing to write 0 as an input "clears" that selected cell.

Finally, the method **drawVictory** takes no arguments and simply draws a circle with the text "You win!" on the grid on the occasion that the user has won. The completion of the puzzle is checked for each time the user inputs a new number.

The **main** function asks the user for the difficulty level of the puzzle, and opens the file for read. What the user inputs as their difficulty level determines which file is read in, and the files consist of a number of zeros and integers in which the zeros represent black spaces for the user to fill in, and the integers are the initial numbers the puzzle starts with.

USAGE

To run the program, simply write: *python3 sudoku.py*. It takes no arguments. The puzzle text documents/files must use the .txt extension, and they must be formatted according to the format mentioned twice above. The tkinter module must be installed. Once the above code is run on the terminal, the user will be asked to select a difficulty (easy, medium, or hard). Any invalid inputs will result in the looping of a request for a valid choice. After the user chooses their difficulty, the gui created with tkinter will promptly show up on-screen, and the user will be able to fill out cells by clicking on them and typing a number on their keyboard (while a cell is selected).

TESTING & POSSIBLE MODIFICATIONS

Some modifications that could be implemented in the next version are: a method to randomly generate a puzzle (from a solved puzzle), and a way to highlight repeating numbers (in a row, column, or

box). One might think that beginning from a solved puzzle and then taking numbers out might be a way to go about randomly generating puzzles, but that's an incomplete thought. If done that way, there may be more than one solution, the puzzle may become impossible to solve without guessing (which is a nuisance), or the puzzle might be outright impossible to solve. In order to do this effectively, one must have a computer analyze the board and actually solve puzzles, dictating its moves and whether guessing was involved or not. Clearly, this capacity goes well beyond the parameters of our assignment, and is better fit for a Sudoku application, to name one example. Highlighting numbers that repeat in a row, column, or box presents a challenge that we're not sure our code was meant to tackle (in the way it was written). Deciding whether or not a number repeats in a series of numbers is not the problem, but rather determining where that repeating number, or repeating numbers, is/are on the board becomes the difficult part. Our code was not written to contain these kinds of indexes; but we acknowledge that it could be done, if one had the sufficient knowledge and patience: we simply deemed it was not necessary for this assignment, and that, after all,0it was more realistic to keep that possible feature out.

SOURCES

All sources below were used to learn how to use tkinter:

I. https://docs.python.org/2/library/tkinter.html
II. http://python-textbok.readthedocs.io/en/1.0/Introduction_to_GUI_Programming.html
III. http://usingpython.com/using-tkinter/