CIS 210, Fall 2016
Introduction to Computer Science

Sudoku Helper

This assignment is due at 5pm on Friday, November 18. Use Canvas to turn sdktactics.py in electronically.

Purpose

This is practice in object-oriented programming, particularly illustrating how the same object can "belong to" (be referenced from) different containers. It provides a little more practice indexing lists as arrays (which you'll need to finish the "groups" list) and using boolean values in variables.

This program also continues our trend toward programs with more moving parts. You can't keep this whole program in your head. You will read more code than you will write. As a consequence, a big part of the work is reading what you need, organizing the knowledge, and keeping just the *right* details in your head to make the changes you need to make.

This program also illustrates callback functions (listeners) for the "model view controller" (MVC) architectural design pattern (covered in lecture on Monday), to cleanly factor a view component (sdkdisplay.py and the graphics modules it uses) from the logic of a model component (sdkboard.py, which can be used with or without the graphical display). Listeners are also used to signal progress when filling in open spaces in a Sudoku puzzle. I have provided this code for you, but you should read it and try to understand how it works.

Pair Assignment

I encourage you to use *pair programming* to complete this assignment. Work together with one classmate. (Switch up; Don't forget the maximum of 3 projects with the same partner.)

Before writing code at the computer, you should work together and independently on the design. Each of you should be able to clearly explain how the program or a part of the program will work. When you are convinced that you both understand how the code will work, then and only then are you ready to write the code.

Sudoku helper

Sudoku is a popular logic-based number placement puzzle.For an example and a bit of history, see this wikipedia article on Sudoku. One of the interesting bits of history is the role of of a Sudoku puzzle generating program in popularizing Sudoku. Creating good puzzles is much harder than solving them!

Your program will read a Sudoku board that may be partially completed. A board file contains 9 lines of 9 symbols, each of which is either a digit 0-9 or the full-stop symbol '.' (also called "period" or "dot") to indicate a tile that has not been filled. Your program will first check for violations of Sudoku rules. If the board is valid, then your program will apply two simple tactics to fill in an as many open tiles as possible.

A valid Sudoku solution has the following properties:

- In each of the 9 rows, each digit 1..9 appears exactly once. (No duplicates, and no missing digits.)
- In each of the 9 columns, each digit 1..9 appears exactly once.
- The board can be divided into 9 subregion blocks, each 3x3. In each of these blocks, each digit 1..9 appears exactly once.

When a board contains the full-stop symbol ".", we check for duplicates but not for missing digits.

Requirements

Your program will read a file in which each line contains a sequence of digits and '.' characters. The file name is given on the command line, like this:
python3 sudoko.py myboard.txt

It may be useful (though slower) to see a graphical depiction of the board, with bad (duplicated) tiles highlighted. You can give the command like this:
python3 sudoku.py --display myboard.txt

Input board descriptions look like this:

```
...26.7.1
68..7..9.
19...45..
82.1...4.
..46.29..
.5...3.28
..93...74
.4..5..36
7.3.18...
```

If there are no duplicated entries in the board (and regardless of whether it is complete, with digits only, or has '.' characters indicating tiles yet to be filled), your program will proceed to the next step. If there are duplicated elements, your program will report them. For example, suppose the input board contained this:

```
435269781
682571493
197834562
826195347
```

```
374682915
951743628
519326874
248957136
963418257
```

Then the interaction would look like this:

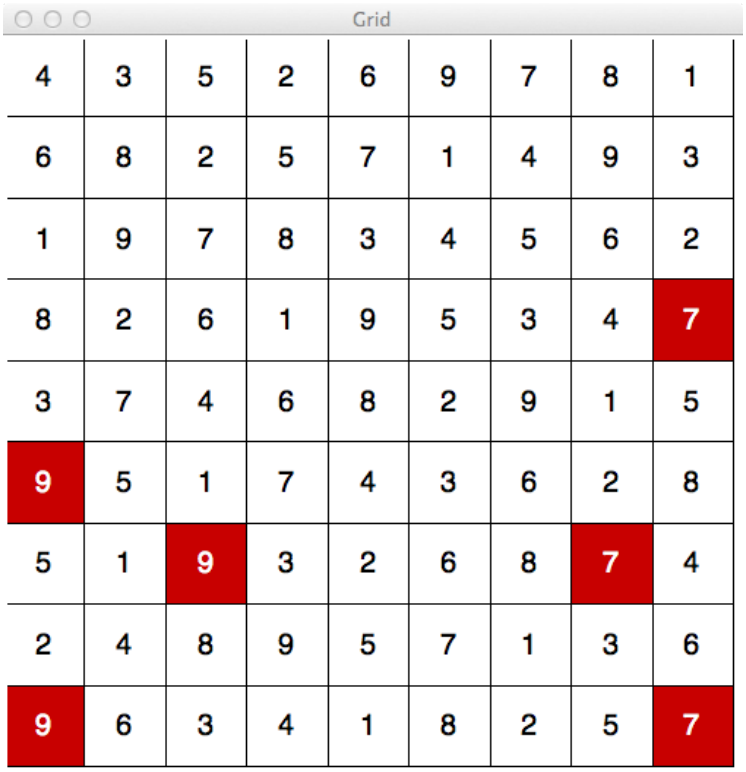
```
$ python3 sudoku.py board1.txt
*** Tile at  5 0  is a duplicate of  9
*** Tile at  8 0  is a duplicate of  9
*** Tile at  3 8  is a duplicate of  7
*** Tile at  8 8  is a duplicate of  7
*** Tile at  6 2  is a duplicate of  9
*** Tile at  6 7  is a duplicate of  7
Sudoku FAIL
```

Note that when a duplicate is found, we report *all* instances of the duplicate. For example, both 9's in the first column are reported as duplicates.

If you used the display option like this:

```
$ python3 sudoku.py --display ../board1.txt
```

Then instead of the textual output, the program will display the board with duplicate items marked:



How to check

Because of the pigeonhole principle of mathematics, if the board contains only the digits 1..9, the following two statements are equivalent:

- None of the 9 digits 1..9 appears more than once in a row.
- Each of the 9 digits 1..9 appear at least once in a row.

That is why we are only checking for duplicates. Checking for missing entries is similar but slightly more complicated, particularly since we are allowing "open" tiles with the dot (".") symbol.

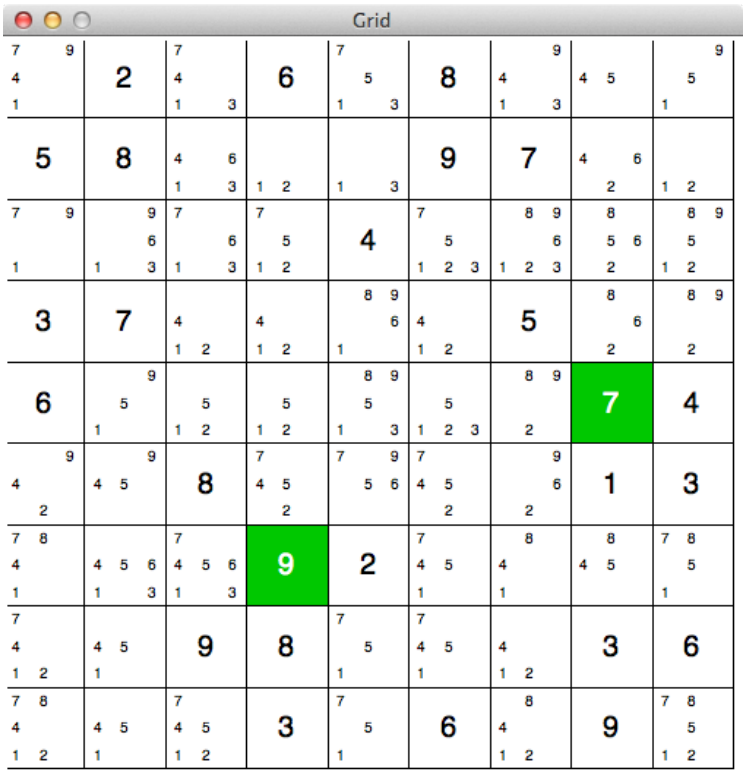
Python makes it fairly easy to check for duplicates, using the built-in “set” data type, although the logic will require careful design and coding. See the lecture from Friday November 13 for the basic idea. The other challenge for you in checking a board for validity is to finish filling in the lists of all the groups (blocks, rows, and columns).

If the board is valid, then (and only then) your program will apply two simple tactics to fill some of the empty tiles, then print the resulting board. (Note that this has no effect on a board that is already complete.) For example:

```
$ more board-sandiway-intermediate.txt
.2.6.8...
58...97..
....4....
37....5..
6.......4
..8....13
....2....
..98...36
...3.6.9.
$ python3 sudoku.py board-sandiway-intermediate.txt
.2.6.8...
58...97..
....4....
```

```
37....5..
6......74
..8....13
...92....
..98...36
...3.6.9.
```

In the example above, only a few tiles have been filled in, because only simple tactics have been used. If you use the --display option, you can see progress in filling in tiles, including elimination of some candidates:



For some easy problems, the simple tactics may be enough:

```
$ more board-incomplete-easy1.txt
...26.7.1
68..7..9.
19...45..
82.1...4.
..46.29..
.5...3.28
..93...74
.4..5..36
7.3.18...
$ python3 sudoku.py board-incomplete-easy1.txt
435269781
682571493
197834562
826195347
374682915
951743628
519326874
248957136
763418259
```

What you must solve

We will consider the solving part of your program correct if it can solve all Sudoku puzzles that can be solved using only the "naked single" (also called "singleton" or "sole candidate") and "hidden single" (also called "unique candidate") tactics described at the SadMan Software site.

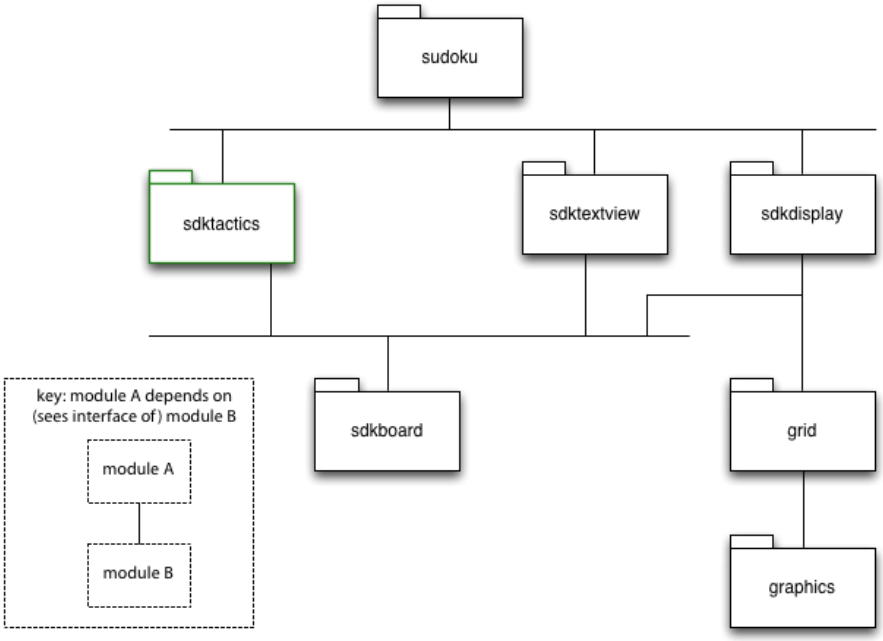
Notes and hints on the Sudoku helper

It is difficult to avoid using the numbers 3 and 9 as magic numbers in a Sudoku program, and it is probably not worth trying. Since they have no other meaning aside from the size of the board and blocks within the board, using symbolic constants doesn't help.

This code features something you may not have seen before: Functions as objects. Look at the way a Tile object allows functions to be registered as event handlers, and how that is used so that the display of an sdkboard.Tile object can be highlighted even though the sdkboard module contains no references to the sdkdisplay module at all. (The text reporting of duplicates is handled the same way, for consistency and to make it easy to suppress the text output when the graphical display is used.) This is a slightly simplified version of an important design pattern that you will see and use a lot in the future. It's called "model-view-controller" because it distinguishes a "model" object (the sdkboard.Board and its component Tile objects) from a a "view" object (the display managed by sdkdisplay, or the textual "display" of messages). If we were taking interactive input from a mouse, that would be the "controller" functionality of the "model-view-controller" pattern. We'll talk about this in lecture.

Starter code and boards to test with

There are a lot of moving parts! However, it is organized so that you never have to think about many parts at once:



You only need to change one file (sdktactics.py), but you will need to read and understand at least one more (sdkboard.py) and probably also the main program (sudoku.py, which does very little itself).

- sdktactics.py This is where you need to provide logic for checking validity, and also for the naked single and hidden single solving tactics.
- sdkboard.py The Tile and Board classes. While you won't need to change these, you will need to read them carefully and refer to them while writing your tactics module.
- sudoku.py Main program ... it just reads the command line and wires together the parts. This includes choosing either the graphical or textual display.
- sdkdisplay.py Uses the grid module to display a Sudoku board. Note the way it registers listeners to highlight duplicate items, display newly selected values in tiles, and display choices for tiles that haven't been filled in yet.
- sdktextview.py Default textual output, which registers listeners just like the graphical display. Note how the main program wires up either the graphical display or this text display. It could just as easily wire up both.
- boards.zip Bad boards that you should recognize as invalid, easy (and valid) boards that you can solve with naked single and hidden single, and hard boards that you will not be able to fully solve with those tactics.
- grid.py A support module that provides a display for grids, using graphics.py.
- graphics.py An open source graphics module providing simple graphics from Python. Not to be confused with turtle graphics!

Grading rubric

Functional correctness				**50**
	Exactly meets input/output spec	5	3=producing extra output (e.g., debugging), 0-3= messed up input/output behavior in other ways.	
	Duplicate detection	15	Accepts correct boards (complete or incomplete) and rejects incorrect boards (those with duplicates in a row, column, or block). Reports all duplicates. 12 = Correctly classifies board, but may not report all duplicate tiles (e.g., reports only one of the 9s in a row containing two 9s). 7 = Correctly classifies most boards, but fails for some (e.g., does not detect duplicates in columns, or reports open squares as duplicates).	
	Naked single	15	Applies the "naked single" tactic successfully. Can solve boards for which that tactic is sufficient. 8 = Correctly applies tactic to most boards, but fails for some.	
	Hidden single	15	Applies the "hidden single" tactic successfully. Can solve boards for which that tactic is sufficient. 8 = Correctly applies tactic to most boards, but fails for some.	
Other requirements				35
	Header docstring	5	5 = as specified, 0 = didn't update with author name, 0-4 other issues	
	Function/method header docstrings	8	8 = complete docstrings consistent with code, 6 = minor problems, 4 = incorrect or multiple missing, 0 = docstrings not provided	
			7 = modules and classes used correctly, no violation of	

	Modularity	7	abstraction, 4 = decomposition has some issues, such as misuse of global variables, use of magic numbers where symbolic constants are more appropriate, 0 = severe violations of modularity		
	Program style and readability	15	15 = Clear logic, appropriate comments, good variable names, indentation, etc, including understandable logic. 12 = minor issues, such as inconsistent indentation or too much repetition in code, or some unclear logic that should have been either simplified or better documented, 5 = major issues that interfere with readability of code, 0 = unreadable mess		
Total					85

I can't anticipate all issues that may be encountered in grading, so points may be deducted for other issues not listed in the rubric. A program that does not compile and run (e.g., because of a syntax error) starts with 0 points for functional correctness, but the grader at his or her discretion may award some partial credit.