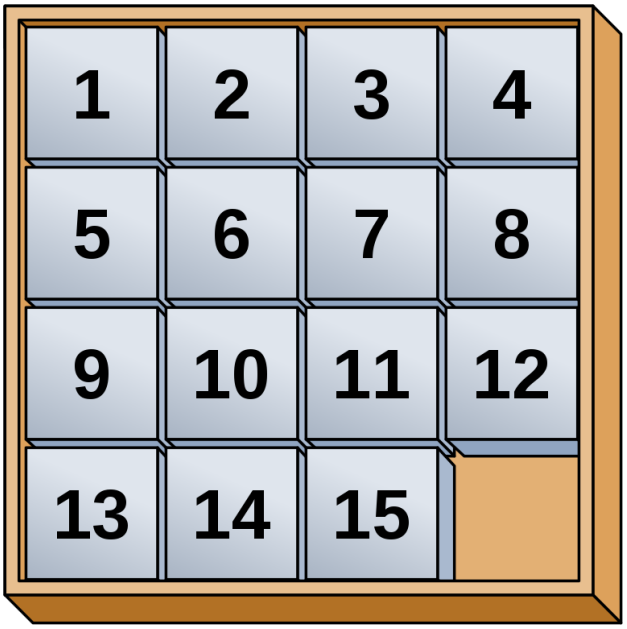
Sliding Number Puzzle

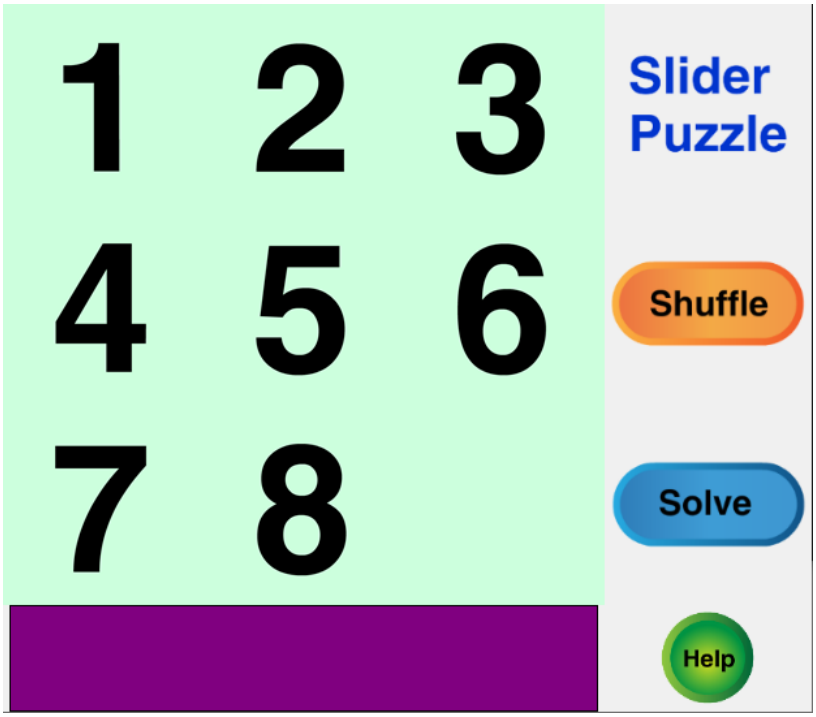
This is a game you play with a grid with numbered blocks. One of the cells is empty, so that neighboring blocks can slide to it. The goal of the game is to line up the numbers left-to-right and top-to-bottom. The following picture shows a 4x4 grid with numbers 1 thru 15.



We will write a program that implements a 3x3 grid containing numbers 1 thru 8.

Explore the game:

If you want to play with my final program to get a feel for this game, run the file mentioned at the end of the article. Try not to peek at the scripts yet, since we want to design them ourselves below. (Note: All intermediate versions of the program mentioned in the article below are available at this link.)



How to play the game:

1. Run the program: the GUI interface comes up.
2. Click "Shuffle" to get a shuffled state, i.e. the numbers won't be in order.
3. Enter any neighboring cell (of the blank square) to swap its position with the blank square. Solve the puzzle by arranging number 1 thru 8 in order.
4. Or, ask the program to solve the puzzle any time by clicking "solve".

Python and CS Concepts Used

When we design this program, we will make use of the following Python and CS concepts. Learn these concepts if you don't know them before proceeding further.

- Algorithms
    - Abstraction
    - Designing new algorithms
    - Pseudo-code
- Arithmetic

    - Expressions
    - Basic operators (+, -, *, /)
- Concurrency
    - Synchronization using threading
- Conditional statements:
    - Conditions: YES/NO questions
    - Relational operators
    - Conditionals (IF)
    - Conditionals (If-Else)
    - Conditionals (nested IF)
    - Boolean operators (and, or, not)
- Data structures – list
    - List operations
    - Using list as 2-D array
    - List traversal
- Data types – basic
    - Integers
- Data types – strings
    - String operations
- Divide and conquer (program design technique)
- Events
- Looping (iteration)
    - Looping - simple (for)
    - Looping - nested
    - Looping - conditional (while)
- Procedures
    - Built-in
    - User defined (custom)
    - Simple
    - With inputs
    - With return value
- Program output
    - Text
- Random numbers
- Sequence
- User input
    - Click buttons

- User interface elements
    - Button
    - Grid
    - Label
- Variables
    - Simple
    - Local/global scope

High Level Design: Console version

If you take a step back and think about how we can implement this program, the following ideas come to mind:

- We will need a way to track the 3x3 grid of numbers: both visually and in the backend.
- Each move will involve swapping of two cells: again both visually and in the backend.
- Shuffling can be thought be as a series of random swaps.
- For the "auto" mode we could keep a record of all moves made by "shuffle" and then replay those moves in the reverse order.

We will first design a purely text-based version.

Let us now consider how we can manage the various types of data required in this program.

Data structures:

We will use a 2-D list called "grid" to save the arrangement of the numbers in the puzzle. We will use "9" to denote the empty cell. Initially, the grid would contain numbers 1 thru 9 (1,2,3,4,5,6,7,8,9) since that is how the grid is displayed at the start. Every time there is a change, i.e. when cells move around, the grid will change accordingly. For example, if "6" moves in the place of the blank cell, the grid will have 1,2,3,4,5,9,7,8,6.

Each cell will need to know its position in this grid. Fortunately, the Tkinter grid does this job for us, when we get to the GUI version.

Global variables:

The above-mentioned lists would need to be global since the whole program will need to access them.

In addition, we will need global variables to give information about (location etc.) the blank cell and information about the most recent clicked cell. Why do we need these global variables? Well, because each cell movement is going to be a swap of cells, and so, we will need this information to implement the swap both physically (on the screen) and in the grid.

Note: We will keep all global variables in a separate "config" file and import them into the program file.

Initial Version

In the initial version of the program, we will work on the following feature ideas:

- Draw a 3x3 table:
    - Each cell should be numbered 1 thru 8 with the last one being blank.
    - Keep track of the arrangement of numbers.
- When a cell is selected (user enters row and column) it should swap places with the blank cell only if it is a valid cell (i.e. if it can slide into the blank position).

Feature Idea # 1: Square grid and swap cells

Step 1: Set up a 3x3 square grid having 9 numbered cells.

Design:

This is a matter of setting up the 2-D list (list of lists) "grid" such each sub-list represents a row. Internally, the blank cell would have the value 9, but while displaying we will show it as a blank cell.

Step 2: When a cell is selected it should swap places with the blank cell.

Design:

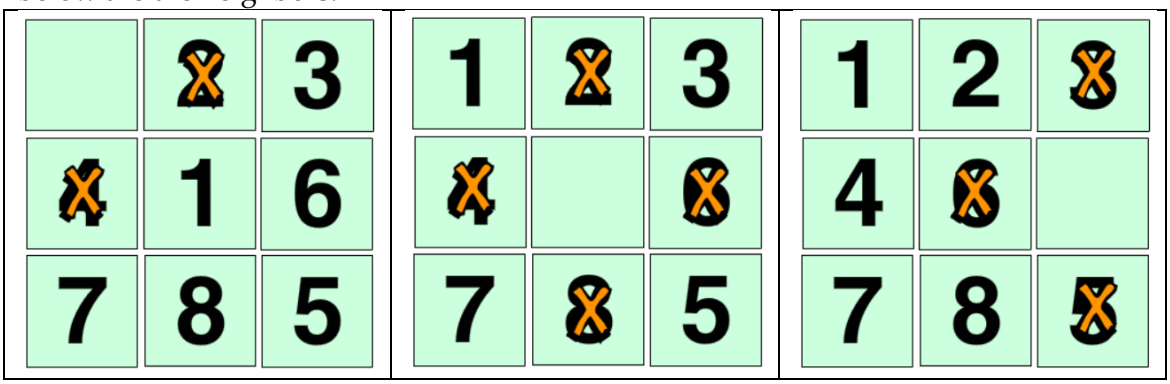
Swapping the selected cell with the blank cell will basically require the (row, column) location of each of them. We will always have the blank cell save its (row, column) in a pair of global variables, blankRow and blankCol.

Feature Idea # 2: Check if neighbor of blank cell

In the real game, the player is allowed to click only on the immediate neighbors (except those placed diagonally) of the blank square. Figure out a way to check if the given cell is a neighbor or not.

Design:

We will need to identify the 4 (max) neighbors of the blank square. The cells crossed below are the neighbors:



We can determine whether or not a cell is a neighbor of the blank cell by comparing its distance from the blank cell in "grid". Here is the algorithm to check if the given cell is a neighbor of the blank cell:

Algorithm Is Blank Neighbor:

Given:
Row, column of clicked cell: row1, col1: input parameters
Row, column of clicked cell: row2, col2: global variables

```
If row1 = row2 AND absolute(col1 - col2) = 1
        Return True
Else if col1 = col2 AND absolute(row1 - row2) = 1
        Return True
```

```
Else
        Return False
End if
```

Implement this function and call it from the script "when cell clicked".

Save program version 1:

Congratulations! You have completed all the features listed so far. As before, let's save this project before continuing to the advanced ideas. Compare your program with my program at the link below.

File: 8-slider-1.py

How to play the game:

1. Run the program: everything is reset to the original state.
2. Select any cell by entering its row and column, e.g. 2,3
3. It will swap its position with the blank square if it's a valid selection.

Next Set of Features/ideas:

The puzzle is not really a puzzle if the numbers aren't shuffled. Include the feature of shuffling the grid. Also include the advanced feature of "auto" mode in which the program can solve the puzzle. Design details are discussed in the features below.

Feature Idea # 3: Placement of cells

The real puzzle requires us to have a shuffled state. Shuffle the cells.

Design:

The actual puzzle requires a shuffled initial state. There are two ways to do this. One is to do random placement of the numbers. But it has been shown (source: Wikipedia) that not all initial number arrangements of this puzzle are solvable. So, we will use the second approach, in which we will initially present the solved puzzle in which numbers 1 thru 8 appear in order followed by the blank cell. We will then shuffle the numbers by picking cells randomly and actually sliding them. This way, it is guaranteed that the shuffled arrangement does have a solution.

It would be best to pick one of the “neighbor” cells of the blank cell because only neighbor cells can be moved. So, as a first step we will need to continuously track the neighbors of the blank cell. We will use the list variable "neighbors" for this purpose.

Data structures:

We will use a list called "neighbors" that will have a list of the current neighbors of the blank cell. If the blank cell moves, this list must be refreshed.

Step 1: Find neighbors of the blank cell and save them in a list.

Design:

This is a matter of scanning the entire grid and using the "Is Blank Neighbor" algorithm (see above) to check if a cell is next to the blank cell. If it is, add it to the "neighbors" list.

Step 2: Shuffle the cells.

Design:

Algorithm to shuffle:

We will simulate clicks on randomly picked neighbor cells for a large number of times.

```
Repeat 100
      pseudoClick = pick at random one of the neighbor cells
      Ask this cell to move by sending the "Simulated click" message
End-repeat
```

Feature Idea # 4: The "auto" mode

Can your program solve the puzzle?

Design:

So far, we expect the user to do all the clicking and solve the puzzle. It would be nice to include a "solve" button that will make the program solve the puzzle.

This sounds daunting but is really quite straightforward. Remember that we start the puzzle in a "solved" state and then through "shuffle" we ruin its order. If we save all the moves made by "shuffle" we just need to unroll those steps in the reverse order and

we will get the solution! To this main idea we will need to include a few minor tweaks to make it robust.

Data structures:

We will need a list called "solution" to list the numbers that need to be clicked to reach the solution of the puzzle.

See below the algorithmic outline of the auto feature:

The "auto" mode:

```
1. Save all moves during "shuffle". Save in list "solution".
2. When user clicks "solve":
   - Unwind this "solution" list bottom-up. Simulate a click
     for each move just like we do in "shuffle".
   - Empty "solution" list.
3. What if user plays manually:
   - Add these moves to "solution".
```

There is a small optimization we can make to this idea. If you inspect the "solution" list created by "shuffle" sometimes the list has meaningless moves (because it uses random operator which is not very intelligent at all). See the example below which shows a subset of this list.

```
5
6
6
5
7
8
8
```

Each item in the list indicates which cell was moved. Now, moves 2 and 3 are meaningless because a single piece was moved back and forth. Same thing with moves 6 and 7.

Such "null" moves should ideally be deleted from the list, right? In fact, this "optimization" may need to be done repeatedly, because, as in the example above, after the 1st pass the list would have 5 repeated at 1 and 2.

Here is the algorithm for the optimization.

```
Algorithm Optimize
Call RemoveNullMoves (below) repeatedly until it returns False

Algorithm Remove Null Moves
Purpose: If an item is repeated in "L", remove both instances
Input: list L
Output: True if duplicates were found, False otherwise
Found = False
I = 1
Size = length of L
Repeat until I >= Size
      Item1 = item at I
      Item2 = item at I+1
      If Item1 = Item2
            Delete Item1 and Item2 from L
            Size = Size - 2
            I = I - 1
            Found = True
      End if
      I = I + 1
End for
Return Found
```

Save as Program Version "Console"

Congratulations! You have completed all the main features of the console version. Compare your program with my program at the link below.

Solution: 8-slider-console.py

How to play the game:

1. Run the program: everything is reset to the original state.
2. Enter "Shuffle" to get a shuffled state, i.e. the numbers won't be in order.
3. Enter any neighboring cell (of the blank square) to swap its position with the blank square. Solve the puzzle by arranging number 1 thru 8 in order.
4. Or, ask the program to solve the puzzle any time by entering "solve".

Next Set of Features/ideas:

Now that we have all the game features in place, let us try to convert the console version to a proper graphical (GUI) version which will look as shown earlier in this document.

We will borrow the Tkinter GUI code from our earlier Tic-tac-toe program. Here are the main requirements from the graphical interface:

(1) Display: The following elements will be needed:
   a. A 3x3 grid of clickable labels (just like in Tic-tac-toe)
   b. Click buttons: Shuffle, Solve, and Help
   c. A message area to display error and other messages.

(2) Hooks to connect with the main logic:
   a. Every time a cell swap happens, we will need to swap the visible cells. The Tkinter grid manager already associates row/column with each cell, which will come handy to do the swapping.
   b. When 'shuffle' and 'solve' are clicked, we should simply call the "shuffle" and "solve" backend methods respectively. The only 'visual' information required is swapping of cells which is already taken care of in (a) above.
   c. Messages: Instead of calling the "print" method, we will use a "Show message" method which shows the message in the Tkinter message label.

Feature Idea # 5: Setup the GUI

Create the layout of all the required GUI elements.

Design:

Front-end components:

We have already listed above the GUI elements needed for this program. Let us now design the display to properly present all these difference pieces. We will use a grid format and allocate cells of the grid as described below.

GUI layout:

Row 1, 2, 3, Columns 1, 2, 3: the 3x3 grid
Row 4, Columns 1, 2, 3: Messages
Row 1 Col 4: Title

Row 2 Col 4: Solve button
Row 3 Col 4: Shuffle button
Row 4 Col 4: Help button

Total grid dimensions:

- Number of rows = 4
- Number of columns = 4

We will use the Tkinter library to design this grid and the pieces therein. We basically need these Tkinter widgets: Image labels, Text labels, and Buttons.

Images:

We will need a lot of images most of which will be of identical dimensions (150x150 to be precise). Instead of using image files, we will use the option of using images encoded in base64 format. Using an online tool, we will convert all these images into base64 strings and create string constants (variables whose value will not change) in our configuration file. For example, the image for the "Help" button has been converted to a base64 string and saved in the variable "helpIMG".

Global variables:

We will need most GUI elements (including the label images so that they are not de-allocated by Python's garbage collector) to be available to multiple functions, and hence we will save them in global variables. Just like other globals, we will list and initialize them in the "config" file.

Functions:

It is straightforward now to create the grid of the various GUI elements using standard Tkinter commands. We already have experience with Tkinter from our earlier projects such as Tic-Tac-Toe.

Save as Program Version "GUI"

Congratulations! You have completed all the features of the GUI version. Compare your program with my program at the link below.

Solution: 8-slider-gui.py

How to play the game:

1. Run the program: the GUI interface comes up.
2. Click "Shuffle" to get a shuffled state, i.e. the numbers won't be in order.
3. Enter any neighboring cell (of the blank square) to swap its position with the blank square. Solve the puzzle by arranging number 1 thru 8 in order.
4. Or, ask the program to solve the puzzle any time by clicking "solve".

Further challenge

See if you can extend this program for a 4x4 grid with numbers 1 thru 15 (as shown in the picture at the very top of this document).

Author: Abhay B. Joshi (abjoshi@yahoo.com)
Last updated: 5 June 2020