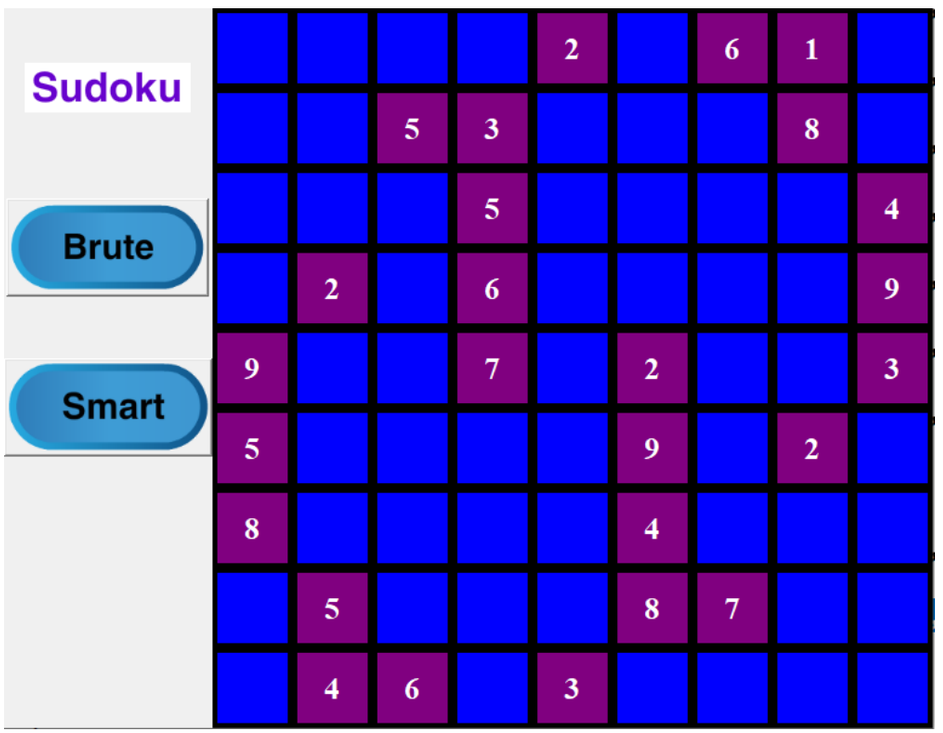
Sudoku Puzzle

Sudoku is a well-known number puzzle in which you are expected to fill a 9x9 matrix with digits 1 to 9 such that no digit is repeated in a row, column, or a 3x3 box.

Typically, the puzzle begins with a partially filled 9x9 matrix and you are expected to fill in the blanks.

Explore the game:

If you want to play with my final program to get a feel for this game, run the final version of the program given at the end of the article. Try not to peek at the code yet, since we want to design them ourselves below.



1. Run the Python program. Press h for help or ENTER to continue.
2. When prompted paste a puzzle in which you use 0s as blanks. See this example:

006000050,200007400,080400001,060194200,000708000,009653070,500006030,003500002,040000800

3. Click on the "brute" button to find the solution using the "brute force" method (described in Section 1 below).
4. Click on the "smart" button to find the solution using the "smart" method (described in Section 2 below).

Download the program files by clicking here.

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.

Concepts used in this program:

- Algorithms
  - Abstraction
  - Designing new algorithms
- Arithmetic
  - Expressions
  - Basic operators (+, -, *, /)
  - Advanced operators: %, //, etc.
- 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
  - Lists of lists (Python)
- Data types – basic
  - Integers
- Data types – strings
  - String operations
  - String traversal
- Data type conversion
- Events
- GUI (graphical user interface)
  - Basic widgets: labels, buttons
  - Synchronizing backend logic with GUI
- 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
- Recursion
- Sequence
- User input
    - Text
    - Click buttons
- Variables
    - Simple
    - Local/global scope

Section 1: The Brute Force Method

High Level Design:

Let's consider how the various features of this program can be separated out as distinct pieces.

As usual, we can consider the backend and the frontend separately. The backend will hold the board in some variable(s) and will have all the logic to manipulate the contents of the board as the game proceeds. The frontend will display the board in a 9x9 grid format and pass user input (the initial click to solve the puzzle) to the backend. The frontend will always show the current state of the board.

For the frontend, we will use a 9x9 grid of Tkinter labels to display the board.

The backend will internally represent the board as an 81-item list in which each item a number in the puzzle. The backend logic needs to solve the puzzle and send periodic updates to the frontend.

Objects:

We will distribute the code among the following objects:

- Frontend object will contain all logic related to the visible 9x9 grid.
- Backend logic will contain all code related to solving the puzzle.

Feature Idea # 1: The Sudoku board

Draw a 9x9 grid of clickable cells.

Design:

Step 1: The visible board

For this, we can simply borrow an older Python program called "matrix" which provides just this functionality. This uses a tkinter label widget in a 9x9 grid. In addition, we will provide a "solve" button which when clicked, will initiate the puzzle solving process.

Step 2: The board state

As discussed earlier, we will use an 81-item list to represent the 9x9 board.

And to ensure the visible board is in sync with the backend, the backend will send a "refresh" message every time the board state changes (while solving the puzzle).

Feature Idea # 2: Solve the puzzle

When the player clicks on the "solve" button, the program proceeds to solve the puzzle.

Design:

We will use what is called the "brute force" or "exhaustive search" approach. In this, we basically try all possible number combinations until the correct one is found. So, we start scanning the grid from top left and fill the first empty cell with a digit that fits (i.e. does not violate Sudoku rules) and then go to the next empty cell, and so on. If a cell does not accept any digit, we need to go back and try a different digit in the previous cell. This process of "trial and error" can go on for a very long time if done manually, but since computers are fast this "dumb" approach turns out to be quite practical.

The following recursive algorithm shows how to implement this approach.

Solve-Sudoku Recursive algorithm:

```
Input: cell id (0 thru 80)

Look for the first blank cell
If no blank cell, puzzle either solved or unsolvable, stop
Repeat digit=1 to 9:
      If digit fits (not used already in cell's row, column or box)
            Place it in current cell
            Recursive call to Solve-Sudoku (input: current cell + 1)
            If Success returned:
                  Return Success
            End if
      End if
      Try the next digit in repeat loop
End repeat
Since no valid digit found, reset cell to 0 and return FAIL
```

This algorithm will require a number of utility procedures which are listed below.

Utility procedures:

We need utility functions to check if a given digit already exists in a row, column, or 3x3 box. The following algorithms take care of these needs:

Algorithm Check Digit in Row

Check if the given digit is present in the given row. The 9 elements in a row appear one after the other.

```
Input: row, digit
Output: True if digit is present, False otherwise.

Procedure:
Count = 9*row
Repeat 9 times:
      If (List[count] == digit):
            Return True
      End if
      Count = Count + 1
End repeat
Return False
```

Algorithm Check Digit in Column

Check if the given digit is present in the given column. The 9 elements in a column appear one after the other but separated by 9 places. For example, column 0 elements would be at 0, 9, 18, and so on.

```
Input: column and digit
Output: True if digit is present, False otherwise.
Procedure:
Count = column
Repeat 9 times:
      If (List[count] == digit):
            Return True
      End if
      Count = Count + 9
End repeat
Return False
```

Algorithm Check Digit in Box

Check if the given digit is present in the given 3x3 box. For this algorithm, we need to hard-code the starting location of each box in the 9x9 grid:

Boxes = [0, 3, 6, 27, 30, 33, 54, 57, 60]

The 9 elements in a box appear in a 3x3 grid starting at the location given in the list above. For example, box 0 elements would be at 0, 1, 2, 9, 10, 11, 18, 19, 20.

```
Input: box and digit
Output: True if digit is present, False otherwise.
Procedure:
Row = 0
Repeat 3 times:
      Start = starting location of box + Row*9
      Count = 0
      Repeat 3 times:
            If (List[Start + count] == digit):
                  Return True
            End if
            Count = Count + 1
```

```
      Row = Row + 1
End repeat
Return False
```

Solution files:
With GUI: sudoku-gui-brute.py
Without GUI: Sudoku-console-brute.py

Section 2: The Smart Method

We already have a fully working version above, so, you may wonder why we need a so-called "smart" approach. Well, it is true that the brute-force method (used above) can solve any Sudoku puzzle in the world. But, it has some shortcomings. It can be terribly slow for some puzzles. For instance if you input the puzzle string given below, you will notice that your program takes a very, very long time (more than 5 minutes on my machine) to solve it.

900002005,862000000,005300600,000400070,009000000,000800510,070003001,000107800,604000002

That is because the brute-force method tries every possible number starting from the left corner of the puzzle. So the time to solve any puzzle depends on how many combinations the program has to cycle through before hitting the exact required combination.

The "Smart" approach:

The smart approach is not really smart in any mathematical sense, but, it tries to use some tricks to speed up the solution. Firstly, we insert the string "123456789" at each blank cell, implying that any one of those 9 digits could finally appear in that cell. The basic idea then is to continually prune the matrix such that all such multi-digit strings become single digit, which would obviously be the solution!

There are two parts to this pruning approach:

Prune Matrix:

It uses two pruning techniques to reduce this string in every cell:

1. If a particular digit appears alone in a cell, it gets removed from all other cells in its row, column, and 3x3 box.
2. If a digit DOES NOT appear alone in any cell in a row, column, or 3x3 box, AND it appears only once in another cell, the content of that cell gets replaced by this lone digit.

This pruning is performed on the matrix right at the beginning as well as later whenever something in the matrix has changed.

Prune a pair:

The "smart" approach uses another trick to achieve further opportunities for pruning. It is called "Prune a pair".

The idea is to prune pairs of digits into single digits by identifying the good digit. The program searches for a pair to play with. It discards each digit in the pair alternately and sees if subsequent pruning (using "Prune Matrix") leads to any error condition anywhere in the matrix. The one which does not lead to an error condition is the good digit, but only if the other one leads to a discrepancy. So, both digits cannot be good. Also, if both result in conflicts, there is internal error, or it's a bad Sudoku puzzle!

This 'playing with a pair' may have to be performed on a number of such pairs before the solution is reached.

The program tracks progress by counting the total number of digits in the matrix; when it reaches 81, that is likely to be the solution. The solution is verified by summing all rows, columns, and 3x3 boxes. The sum, in all cases, must be 45.

So, that's the basic approach. Below, we will see the detailed algorithms for the different procedures.

Algorithm Prune Matrix:

1. N1 = number of digits in matrix
2. Prune matrix using single digits (explained separately below).
3. Prune matrix lone digits (explained separately below).
4. N2 = number of digits in matrix
5. If N2 < N1 (which means pruning is working), repeat the pruning from step #1.

Algorithm Prune matrix using single digits:

1. Scan the entire matrix from top-left corner:
2. If a cell contains a single digit (e.g. '1') :
   - Eliminate that digit from all multi-digit cells in its row, column, and 3x3 box, (e.g. an existing string "12345" will not become "2345").
3. If the matrix changed in step 2 above, that means there may be more new single-digit cells, so, repeat the whole procedure from Step #1.

Algorithm Prune matrix using lone digits:

1. For each row R:
   - For digits 1 thru 9 (D):
     i. If D doesn't appear as a single digit in R, count how many multi-digit cells it appears in.
     ii. If the count is 1, replace that multi-digit cell with D.
   - (For example: if the digit "2" does not appear alone in a row, and it appears only once in the row, say in "2345", then it is safe to replace "2345" by 2, because by Sudoku rules "2" must appear alone once in this row.)
2. For each column C:
   - For digits 1 thru 9 (D):
     i. If D doesn't appear as a single digit in C, count how many multi-digit cells it appears in.
     ii. If the count is 1, replace that multi-digit cell with D.
3. For each 3x3 box B:
   - For digits 1 thru 9 (D):
     i. If D doesn't appear as a single digit in B, count how many multi-digit cells it appears in.
     ii. If the count is 1, replace that multi-digit cell with D.

Algorithm Prune a pair:

1. Scan the matrix from top-left corner and find a cell that contains 2 digits.
2. Use the 1st digit and check if it leads to conflict by calling "check conflict" (explained later)
3. Use the 2nd digit and check if it leads to conflict by calling "check conflict" (explained later)

4. If both result in non-conflicting states, we can't pick any. So, go back and look for the next pair.
5. If both result in conflict, there is internal error, or it's a bad Sudoku puzzle!
6. If one and only one leads to non-conflict, that is a good digit for that cell. Call "Prune Matrix".
7. Repeat these steps (go to Step 1) until there are no more pairs or a solution is reached.

Algorithm Check Conflict:

Conflict simply means violation of Sudoku rules. All we need to do is check if a row, column, or 3x3 box contains duplicates of the same single digit. If so, that's a conflict, otherwise not.

Solution files:

With GUI: sudoku-gui.py
Without GUI: sudoku-console.py

.

Author: Abhay B. Joshi (abjoshi@yahoo.com)
Last updated: 6 April 2020