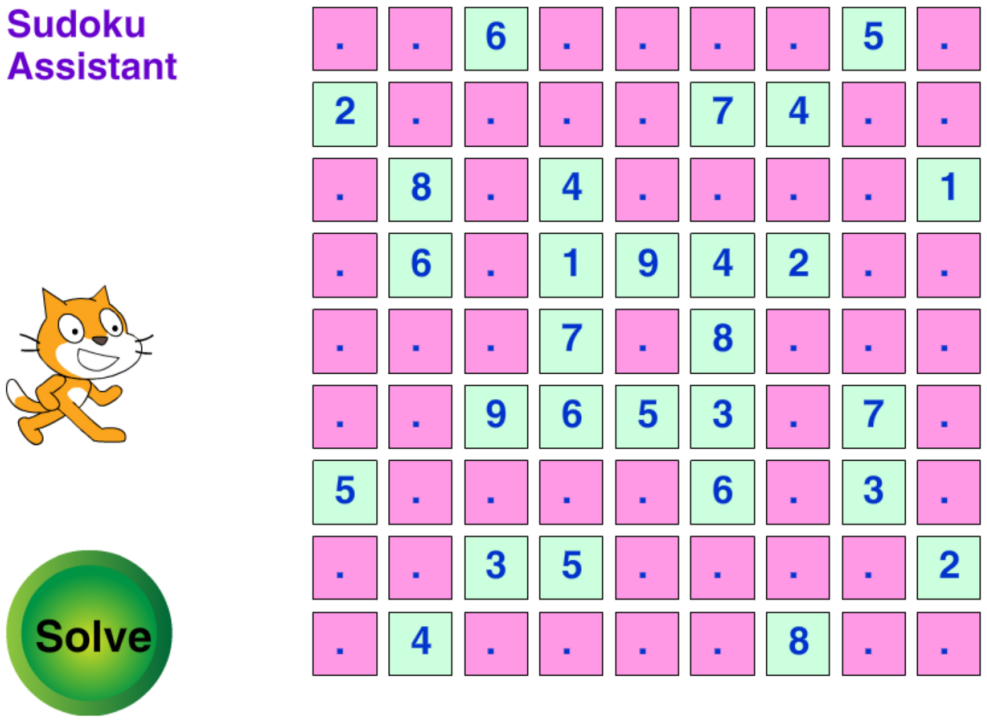
Sudoku Assistant

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.

This program provides assistance to solve any Sudoku puzzle. It lets you enter numbers manually by checking basic rules of the game. It also provides a "Solve" button that user can click anytime to have the computer try to fill in the remaining cells.

Here is how my program looks when a new puzzle is loaded:



Explore the game:

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

1. Click the "Green flag". Read instructions and click to continue.
2. Enter a Sudoku puzzle in the specified format.
3. Click blank cells to try to solve the puzzle yourself. Enter 0 to make a cell blank again.
4. Click "Solve" anytime and the program will attempt to fill in the remaining blanks.

Click here to download all program files. It includes a text file that contains a few sample puzzles in the format required for this program.

Scratch and CS Concepts Used

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

- Algorithms
  - Abstraction
  - Using algorithms
  - Designing new algorithms
  - Pseudo-code
- Arithmetic
  - Expressions
  - Basic operators (+, -, *, /)
  - Advanced operators: mod, floor, ceiling, etc.
- Concurrency
  - Synchronization using broadcasting
  - Synchronization using variables
- Conditional statements:
  - Conditions: YES/NO questions
  - Relational operators (=, <, >)
  - Conditionals (IF)
  - Conditionals (If-Else)
  - Conditionals (Wait until)
  - Conditionals (nested IF)
  - Boolean operators (and, or, not)
- Data structures – list
  - List operations
  - List traversal
- Data types – strings
  - String operations (join, letter, length of)
  - String traversal
- Divide and conquer (program design technique)
- Events
  - Events - coordinating multiple user events
- Looping (iteration)
  - Looping - simple (repeat, forever)
  - Looping - nested
  - Looping - conditional (repeat until)
- OOP
  - Clones
  - Clones differentiation: using private id
- Procedures
  - Built-in
  - User defined (custom)

    - Simple
    - With inputs
    - With return value
- Program output
    - Text
- Random numbers
- Recursion
- Sequence
- State
    - Transitions
- Stopping scripts
- User input
    - Text
    - Click buttons
    - Input validation
    - Keyboard events
    - Mouse events
- Variables
    - Simple
    - Properties (built-in)
    - Local/global scope
- XY Geometry

Special note:

We will write custom procedures that need to have local variables and return values, which Scratch does not support. We will use ideas proposed in "Script variables" (see Scratch/blog/Script-variables.pdf for details) to overcome this problem. Specifically, we will import the following custom blocks from that library:

Read the above document to understand how to use these custom blocks.

Checkpoint 1 High Level Design

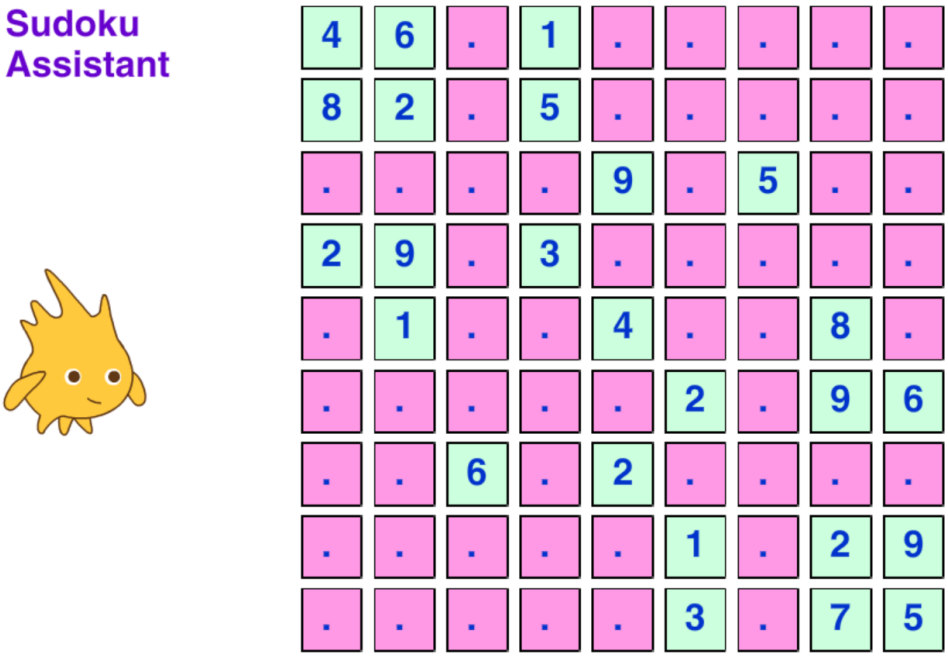
As usual, we will build our program step by step. In this checkpoint we will include the following features:

1. Display the Sudoku 9x9 board (matrix).
2. Ask the user to enter a new puzzle in a comma-separated string format.

(Ex: 460100000,820500000,000090500,290300000,010040080,000002096,006020000,000001029,000003075,)

3. Display the puzzle in the matrix.

Sample output:



Feature #1 9x9 puzzle matrix

Step 1: Draw the Sudoku matrix consisting of 81 cells arranged in a 9x9 format.

Design:

We will use the following "Number-table" program (that I wrote) as starter code for this feature:
Design description: http://www.abhayjoshi.net/spark/scratch/blog/number-table.pdf
Sample code: https://scratch.mit.edu/projects/318322805/

This program uses a single square sprite and a digit sprite to create a matrix of any dimension. It uses the STAMP command to create multiple instances of the square sprite and number sprite. But, in our Sudoku program we need these squares and numbers to be sensitive to mouse pointer click. So, we will use clones instead of STAMP.

Step 2: Ask the user to enter a new Sudoku puzzle and display it in the matrix.

Design:

This routine reads the initial puzzle and creates a list S of 81 elements. We will ask the user to enter the puzzle in format of 9 words each ending with a comma. For example:

```
000065130,000008900,100009002,025000709,000000000,607000240,800900004,001500000,032780000,
```

```
Algorithm Read Sudoku Puzzle:
Input: I - puzzle string 9 rows each ending with a comma, 0 for blank
Output: S - list of 81 item (each 0 to 9)

Steps:
Repeat 9 times
      Read 9 letters of I
      Validate and enter each letter as a digit into S
      Skip the 10th letter (since it's the comma)
End repeat
```

How do we "validate" each letter? We need verify that each letter is a digit between 0 to 9. We could use the fact that "abs" operator returns 0 for any non-digit letter (such as 'a' or 'z'). The following trick would do this:

```
If letter is not 0 AND absolute value of letter is 0
      Letter is not a digit
End if
```

Step 3: Show puzzle cells and user cells in different colors.

Design:

As shown at the top of this chapter, we would like to show puzzle cells in light green and user cells in purple color. This is simply a matter of having two costumes for the square sprite. When the user enters a new puzzle, we know that 0 indicates a user cell. The clone would then switch to the appropriate costume.

Later, we will also need a way to find out a cell's type: whether it's puzzle or user. Since every cell has a unique id (both the square and digit clones would have unique ids from 1 to 81), it can discover its type by looking the "costume #" property.

Here is the algorithm assuming 1st costume is light green and 2nd is purple.

Algorithm Get Type

```
Input: cell
Output: cell type
Procedure:
Every clone will compare "cell" with its "id"
If cell = id
      Look up costume #.
      If it is 1, type is "puzzle", else it is "user".
End if
```

Save as Program Version 1

Congratulations! You have completed all important features till this checkpoint. Compare your program with my program at the link below.

File: sudoku-assistant-1

How to run this program:

1. Click the "Green flag". Read instructions and click to continue.
2. Enter a Sudoku puzzle in the specified format.

Checkpoint 2 High Level Design

In this program version, we mainly want the user to be able to play the puzzle, i.e. be able to solve it themselves. Here are the required features:

1. User clicks a cell and enters a digit.
2. Program verifies that:
   - The cell is valid (not a puzzle cell)
   - The digit is valid (0 thru 9)
   - The digit meets Sudoku rule: not repeated in its row, column, 3x3 box
3. Program displays the digit in the matrix. Makes the cell blank if digit=0
4. Program checks if the board is solved.
5. User can click again.

Feature #2 Digit entry

Step 1: Allow the user to enter a digit for a particular cell of the Sudoku matrix.

Design:

In order to solve the puzzle, the user needs to fill the blank cells with appropriate digits (1 thru 9). One way is to repeatedly use the ASK command to get the cell # and the digit. But that would be tedious for the user. Is there a better way?

Can we allow the user to simply click a cell and press the digit key? For instance, if they wanted to enter "3" in a specific cell, they should be able to click that cell and then press the "3" key.

How can we implement this mechanism? There is no straightforward way to do this in Scratch.

Well, we can apply some ingenuity. Clicking a cell is of course straightforward. As soon as the user clicks a cell, we can find out that cell's "id", its "type, and its content. We can also have a series of "When key pressed" scripts to sense when user presses a number key.

But these scripts should be "activated" only after the cell click. How do we do that?

A variable would come to help! Let's call it "expecting numkey". All these scripts would look at this variable for their work. See an example below:

```
When "1" key pressed:
If "expecting numkey" = True
      Set "expecting numkey" = False
      Digit = 1
End if
```

Thus, "expecting numkey" would normally be False, thus deactivating all number keys. When a cell is clicked, it would be changed to True, thus activating the number keys. The number key scripts, in turn, would turn it off once a number key has been pressed.

Step 2: Validate the digit and enter it in the Sudoku matrix.

Design:

Once we have both the cell # and the digit, the rest is straightforward.

Here is the algorithm that is activated after clicking a cell.

Algorithm Cell clicked:

```
Input: cell
Procedure:
Verify the cell type is "user".
Prompt user to press a number key
Set "expecting numkey" to True (thus enabling the number key scripts mentioned above)
Set digit = -1
```

```
Wait until digit is not -1 (thus we wait here until user presses a num key)
If digit = 0
      Enter 0 in S (which will make the cell blank)
      Update the display
Else
      Validate digit by using the "check" row/column/box algorithms (defined below)
      Enter digit in S
      Update the display
End if
```

Feature #3 Digit validation

We need scripts to check if a given digit already exists in its own row, column, or 3x3 box.

Design:

The following algorithms take care of these needs. We will implement them as custom blocks. Note that in these algorithms, for the sake of convenience, we will count from 0. That is, first row is row 0, second row is row 1, and so on. Same for columns and boxes.

Step 1: Check if the given digit already exists in its own row.

Design:

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
```

Step 2: Check if the given digit already exists in its own column.

Design:

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
```

Step 3: Check if the given digit already exists in its own 3x3 box.

Design:

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 "Get box" algorithm (described later below) would allow us to discover in which box the given cell belongs.

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
      End repeat
```

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

Algorithm Get Box

Find out the 3x3 box in which the given cell resides. Sudoku grid has 9 boxes. We do this computation by first looking at the row and then at the column of the cell.

```
Input: cell (0 to 80)
Output: box (0 to 8)
Procedure:
Column = cell % 9     // remainder
Row = int (cell / 9)  // integer division
If (row < 3)
      box = 0
Else if (row < 6)
      box = 3
Else
      box = 6
If (column > 5)
      box += 2
Else if (column > 2)
      box += 1
Return box
```

Algorithm Check if Puzzle Solved

In a fully solved Sudoku puzzle, every row (and column and box) adds up to 45. So, the total 9x9 matrix would add up to 9*45=405. We just need to verify that the matrix has no blank cells and it adds up to 405.

Save as Program Version 2

Congratulations! You have completed all important features till this checkpoint. Compare your program with my program at the link below.

File: sudoku-assistant-2

How to run this program:

1. Click the “Green flag”. Read instructions and click to continue.
2. Enter a Sudoku puzzle in the specified format.
3. Click blank cells to try to solve the puzzle yourself. Enter 0 to make a cell blank again.

Checkpoint 3 High Level Design:

Provide a button to solve the puzzle. Program tries to fill in the remaining cells. This can be done using a brute-force method, also known as exhaustive search algorithm.

Feature #4 Solve the puzzle

Write scripts to attempt to solve the given Sudoku puzzle.

Design:

The program tries to fill in all the remaining empty cells. If the user has already entered some of the cells and if any of them is incorrect, the program would not be able to solve.

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 fill it with a digit that fits, and so on. If a cell does not accept any digit, we go back to the previous cell that we filled and try a different digit in it. This process of "trial and error" and "backtracking" 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 and solves most puzzles in a few seconds.

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
```

Save as Program Version Final

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

File: sudoku-assistant-final
MIT Scratch: https://scratch.mit.edu/projects/630922543/

How to run this program:

1. Click the "Green flag". Read instructions and click to continue.
2. Enter a Sudoku puzzle in the specified format.
3. Click blank cells to try to solve the puzzle yourself. Enter 0 to make a cell blank again.
4. Click "Solve" anytime and the program will attempt to fill in the remaining blanks.

Author: Abhay B. Joshi (abjoshi@yahoo.com)
Last updated: 19 January 2022