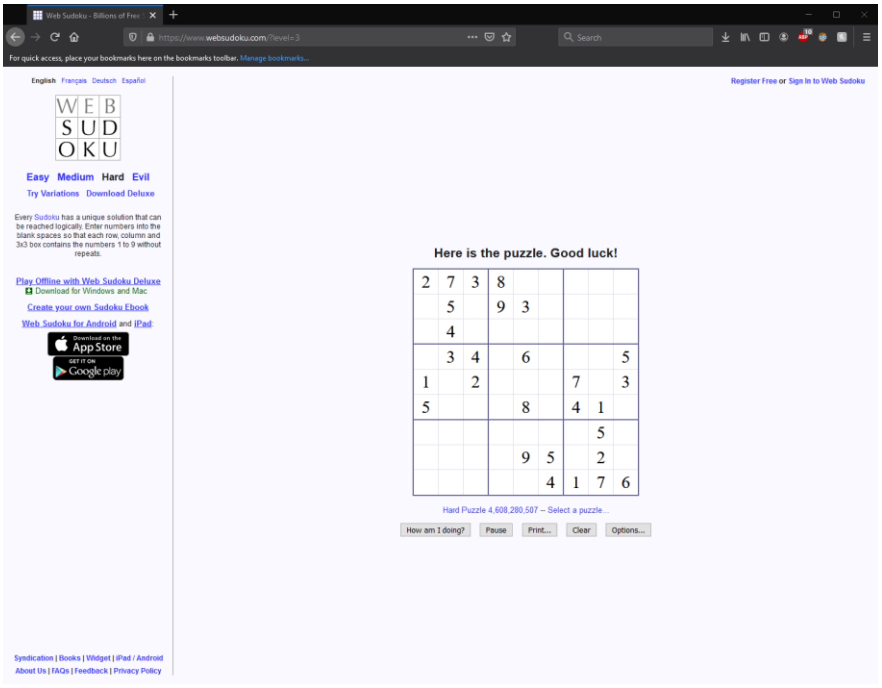
Simplified Guide to Computer Vision: Sudoku Solver

An Introduction to OpenCV with Python

By Z. Paul Weleschuk

Task: Solve Sudoku Puzzle using OpenCV

The task for this blog post is to use OpenCV to solve a sudoku puzzle. This program will not have any additional information or access to data other than what is visually displayed on the users screen. This program will solve sudoku puzzles on the website https://www.websudoku.com/.


Web Sudoku - Billions of Free
https://www.websudoku.com/?level=3
Search
For quick access, place your bookmarks here on the bookmarks toolbar. Manage bookmarks...
English Français Deutsch Español
WEB SUDOKU
Easy Medium Hard Evil
Try Variations Download Deluxe
Register Free or Sign In to Web Sudoku

Every Sudoku has a unique solution that can be reached logically. Enter numbers into the blank spaces so that each row, column and 3x3 box contains the numbers 1 to 9 without repeats.

Play Offline with Web Sudoku Deluxe
Download for Windows and Mac
Create your own Sudoku Ebook
Web Sudoku for Android and iPad
App Store
Google play

Here is the puzzle. Good luck!

2	7	3	8					
	5		9	3				
	4							
	3	4		6				5
1		2				7		3
5				8		4	1	
							5	
				9	5		2	
					4	1	7	6

Hard Puzzle 4,608,280,507 – Select a puzzle...

How am I doing?
Pause
Print...
Clear
Options...
Syndication | Books | Widget | iPad / Android
About Us | FAQs | Feedback | Privacy Policy

Rules of Sudoku

Sudoku is a puzzle in which a single digit can only occur once per row, column, and 3x3 sub grid.

Approach to the problem

The problem can roughly be broken down into three parts.

Part One: Identify the numbers and blank tiles The program will need to be able to identify the different numbers as well as the blank tiles on the sudoku board.

Part Two: Build the sudoku board
It will then need to determine where those values are position within the sudoku puzzle (ie: in a 2D array).

Part Three: Solve the sudoku
The program will then need to be able to figure out the solution to the puzzle and enter the correct values into the puzzle.

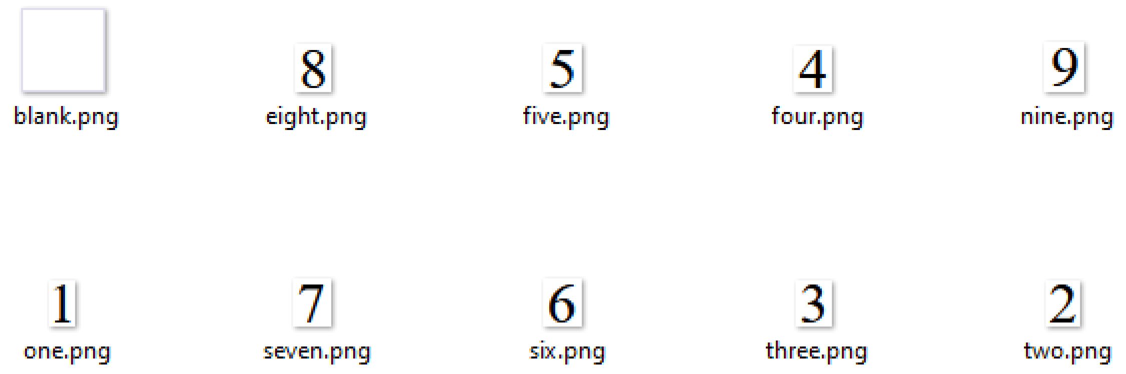
What will be needed

OpenCV
Numpy
Pillow
Pyautogui
Sample images

Take sample images

Sample images need to be taken from the sudoku board. There is no variation with the sample images (ie; all the ‘1’s look exactly the same for every occurrence on every board). Therefore template matching would be the best option for solving this problem.

Each number (1 - 9) and a blank tile are needed for template matching. This is done manually, but once done can be used on any of the 6.67×10^{21} possible puzzles on the website.



Images object

An object is created to store the attributes of each image.
The object stores the image, the value that image represents, the threshold value that will be used with the match template function, and a color (BGR). The threshold value must be determined through trial and error to get the most optimized output; this is where experience in OpenCV is a factor. The color is not required but used to better visually display how the program works.

```python
""" Image class that stores an image and its associated
attributes"""

class Image:
    def __init__(self, image, value, threshold,b,g,r):
        self.image = image
        self.value = value
        self.threshold = threshold
        self.b = b
        self.g = g
        self.r = r
```

Create image object list

Image objects are constructed and are put in a list.

```python
def createImageList():
    # returns object list of images and its parameters
    imageList = []
    blank = cv.imread('photos/blank.png', 0)
    imageBlank = Image(blank, 0, 0.85, 255, 255, 153)
    imageList.append(imageBlank)
    one = cv.imread('photos/one.png', 0)
    imageOne = Image(one, 1, 0.85, 0, 0, 255)
    imageList.append(imageOne)
    two = cv.imread('photos/two.png', 0)
    imageTwo = Image(two, 2, 0.87, 65, 153, 255)
    imageList.append(imageTwo)
    three = cv.imread('photos/three.png', 0)
    imageThree = Image(three, 3, 0.87, 51, 255, 255)
    imageList.append(imageThree)
    four = cv.imread('photos/four.png', 0)
    imageFour = Image(four, 4, 0.85, 0, 255, 128)
    imageList.append(imageFour)
    five = cv.imread('photos/five.png', 0)
    imageFive = Image(five, 5, 0.87, 153, 255, 51)
    imageList.append(imageFive)
    six = cv.imread('photos/six.png', 0)
    imageSix = Image(six, 6, 0.85,255, 153, 51 )
    imageList.append(imageSix)
    seven = cv.imread('photos/seven.png', 0)
    imageSeven = Image(seven, 7, 0.87, 255, 0, 127)
    imageList.append(imageSeven)
    eight = cv.imread('photos/eight.png', 0)
    imageEight = Image(eight, 8, 0.84, 255, 51, 255)
    imageList.append(imageEight)
    nine = cv.imread('photos/nine.png', 0)
    imageNine = Image(nine, 9, 0.86, 0, 0, 0)
    imageList.append(imageNine)
    return imageList
```

Take an image of the screen

An image of the screen, with the sudoku puzzle in it, will need to be captured in order for Open CV to have the information it needs to solve the puzzle.

In this example only a portion of the screen is captured (the top left portion of the screen, 800 pixels wide, 1000 tall). The image is then converted; one to RGB and the other to grey scale. The RGB image will be used to display information to the user and the grey scale image will be used with the template matching function. The grayscale image is then thresholded to binary color space (black or white). Both the thresheld image and the color image are returned.

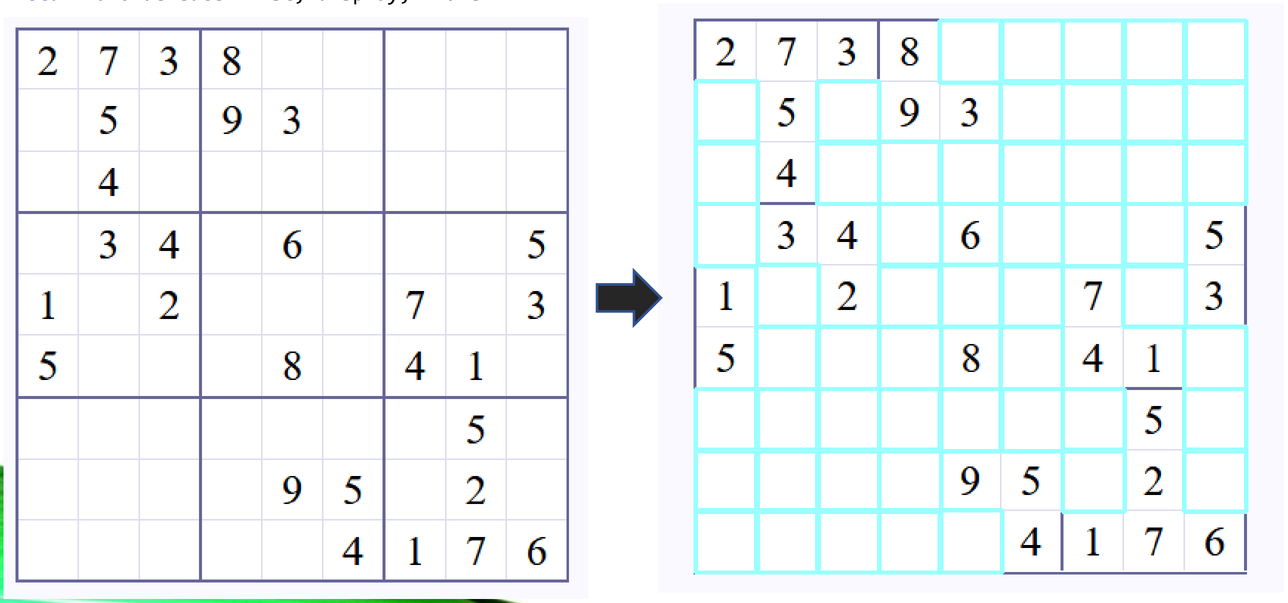
Note: one of the special things about python is that multiple values can be returned by a function, as opposed to java where only a single value can be returned by a method.

```
def getFrame():
    # returns images of screen
    topOfScreen = 0
    leftSideOfScreen = 0
    screenWidth = 800
    screenHeight = 1000
    image = ImageGrab.grab(bbox=(topOfScreen, leftSideOfScreen, screenWidth, screenHeight))
    img_np = np.array(image)
    display = cv.cvtColor(img_np, cv.COLOR_BGR2RGB)
    frame = cv.cvtColor(img_np, cv.COLOR_BGR2GRAY)
    _, frame = cv.threshold(frame, 240, 255, cv.THRESH_BINARY)
    return frame, display
```

Find each blank tile on the puzzle

Find the blank images using template matching. Oddly enough this is the hardest image to template match. Its hard to tell a computer to find a blank space, empty space, space with nothing in it. Because of this extra code is required to achieve the desired result. See slide 32 – 34 for explanation of the function [OpenCV Simplified Guide]. A cyan colored box was drawn around the blank tiles to show the user where blank tiles were identified by the program. The information of the found images are stored in an object Cell (explained later).

```python
def findBlankTiles(frame, display):
    # returns list of blank cell objects
    unorderedCellList = []
    _, blankThres = cv.threshold(imageList[0].image, 240, 255, cv.THRESH_BINARY)
    w, h = imageList[0].image.shape[::-1]
    blankResult = cv.matchTemplate(frame, blankThres, cv.TM_CCOEFF_NORMED)
    blankLocation = np.where(blankResult >= imageList[0].threshold)
    rectangles = []
    for loc in list(zip(*blankLocation[::-1])):
        rect = [int(loc[0]), int(loc[1]), w, h]
        rectangles.append(rect)  # Add every box to the list twice in order to retain single (non-overlapping) boxes
        rectangles.append(rect)
    rectangles, weights = cv.groupRectangles(rectangles, groupThreshold=1, eps=0.5)  # remove the duplicate results
    for i in range(len(rectangles)):
        cv.rectangle(display, (rectangles[i][0], rectangles[i][1]),
                     (rectangles[i][0] + rectangles[i][2], rectangles[i][1] + rectangles[i][3]),
                         (imageList[0].b, imageList[0].g, imageList[0].r), 2)
        cell = Cell(0, 0, rectangles[i][0], rectangles[i][1])
        unorderedCellList.append(cell)
    return unorderedCellList, display, frame
```

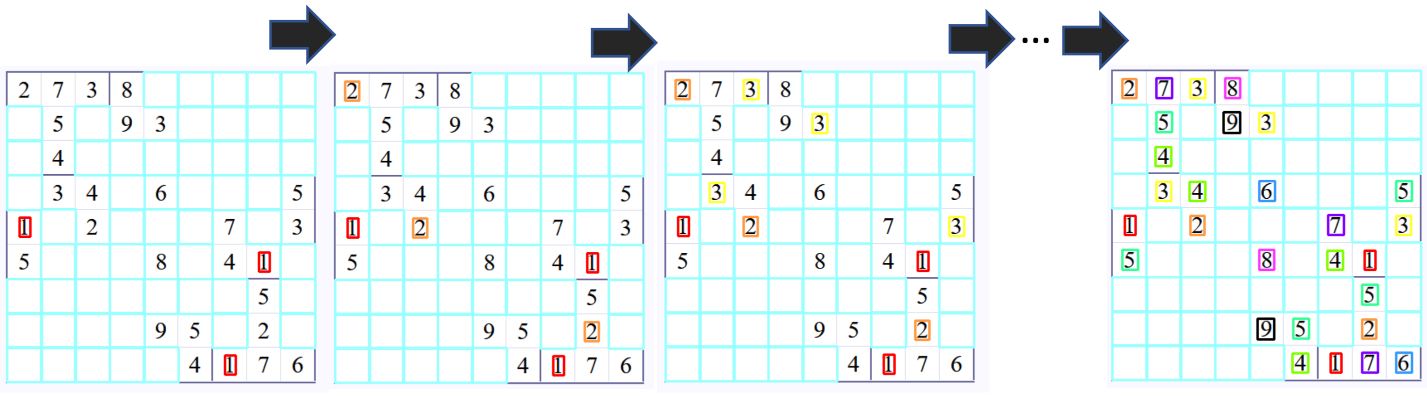


Find each number tile on the puzzle

Template matching is used to find all the original numbers on the puzzle. The numbers are found each digit/template image at a time. A colored box is drawn around each different number value to show the user where each image was identified. Cell objects are created from the found objects. The amount of each number found is displayed.

```python
def findNumber(image, value, threshold, frame, display, b, g, r, unorderedCellList):
    # returns list of the numbers found in the image and print results
    _, frame = cv.threshold(frame, 240, 255, cv.THRESH_BINARY)
    w, h = image.shape[::-1]
    _, imageThres = cv.threshold(image, 240, 255, cv.THRESH_BINARY)
    imageResult = cv.matchTemplate(frame, imageThres, cv.TM_CCOEFF_NORMED)
    imageLocation = np.where(imageResult >= threshold)
    count = 0
    for pt in zip(*imageLocation[::-1]):
        cv.rectangle(display, pt, (pt[0] + w, pt[1] + h), (b, g, r), 2)
        cell = Cell(value,value, pt[0], pt[1])
        unorderedCellList.append(cell)
        count += 1
    if value == 1:
        print("Numbers Found")
        print("-" * 13)
    print(value, "'s: ", count, sep="")
    return unorderedCellList
```

```
Numbers Found
-------------
1's: 3
2's: 3
3's: 4
4's: 4
5's: 5
6's: 2
7's: 3
8's: 2
9's: 2
```



Sudoku Cell Object

For each image found an object is created to store the attributes of the cells within a sudoku puzzle. A cell in a sudoku puzzle has an original value when the puzzle is started and if the cell is blank a new value can be entered into the cell. For this object each cell is located by its X and Y pixel value, not its position on the board (that will come later).

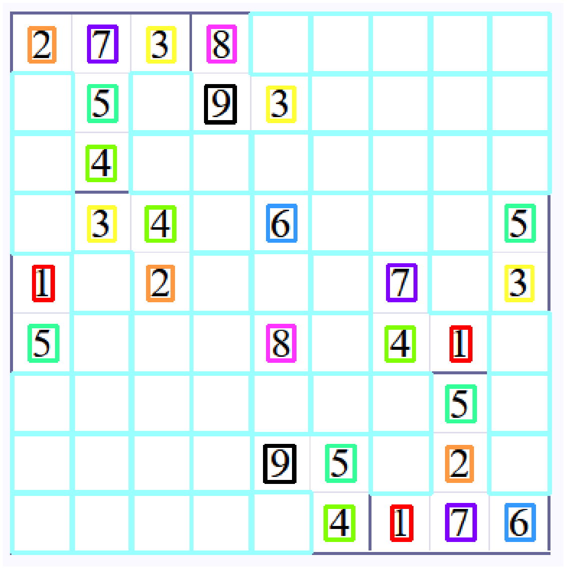
```
""" Cell class that represents each tile on a sudoku board"""

class Cell:
    def __init__(self, originalValue, newValue, xpixel, ypixel):
        self.originalValue = originalValue
        self.newValue = newValue
        self.xpixel = xpixel
        self.ypixel = ypixel
```

List of unorganized cell objects

For each image found, a cell object was created and appended to a list. See example image. Although the position of the found images is very intuitive to use as humans, computers have no concept of relative position and where they would fit in the context of a sudoku puzzle. To the computer, it is simply a list of images that have a corresponding value and position.

What we see



2	7	3	8					
	5		9	3				
	4							
	3	4		6				5
1		2				7		3
5				8		4	1	
							5	
				9	5		2	
					4	1	7	6

What the computer sees

```
Value: 0  X Position: 355  Y Position: 437
Value: 0  X Position: 397  Y Position: 437
Value: 0  X Position: 438  Y Position: 437
Value: 0  X Position: 479  Y Position: 437
Value: 0  X Position: 149  Y Position: 478
Value: 0  X Position: 231  Y Position: 478
Value: 0  X Position: 355  Y Position: 478
Value: 0  X Position: 397  Y Position: 478
Value: 0  X Position: 438  Y Position: 478
Value: 0  X Position: 479  Y Position: 478
Value: 0  X Position: 149  Y Position: 519
Value: 0  X Position: 231  Y Position: 519
Value: 0  X Position: 273  Y Position: 519
Value: 0  X Position: 314  Y Position: 519
Value: 0  X Position: 355  Y Position: 519
Value: 0  X Position: 397  Y Position: 519
Value: 0  X Position: 438  Y Position: 519
Value: 0  X Position: 479  Y Position: 519
Value: 0  X Position: 149  Y Position: 561
Value: 0  X Position: 273  Y Position: 561
Value: 0  X Position: 355  Y Position: 561
Value: 0  X Position: 397  Y Position: 561
Value: 0  X Position: 438  Y Position: 561
Value: 0  X Position: 190  Y Position: 602
Value: 0  X Position: 273  Y Position: 602
Value: 0  X Position: 314  Y Position: 602
Value: 0  X Position: 355  Y Position: 602
Value: 0  X Position: 438  Y Position: 602
Value: 0  X Position: 190  Y Position: 644
Value: 0  X Position: 231  Y Position: 644
Value: 0  X Position: 273  Y Position: 644
Value: 0  X Position: 355  Y Position: 644
Value: 0  X Position: 479  Y Position: 644
Value: 0  X Position: 149  Y Position: 686
Value: 0  X Position: 190  Y Position: 686
Value: 0  X Position: 231  Y Position: 686
Value: 0  X Position: 273  Y Position: 686
Value: 0  X Position: 314  Y Position: 686
Value: 0  X Position: 355  Y Position: 686
Value: 0  X Position: 397  Y Position: 686
Value: 0  X Position: 479  Y Position: 686
Value: 0  X Position: 149  Y Position: 727
Value: 0  X Position: 190  Y Position: 727
Value: 0  X Position: 231  Y Position: 727
Value: 0  X Position: 273  Y Position: 727
Value: 0  X Position: 397  Y Position: 727
Value: 0  X Position: 479  Y Position: 727
Value: 0  X Position: 149  Y Position: 768
Value: 0  X Position: 190  Y Position: 768
Value: 0  X Position: 231  Y Position: 768
Value: 0  X Position: 273  Y Position: 768
Value: 0  X Position: 314  Y Position: 768
Value: 1  X Position: 164  Y Position: 612
Value: 1  X Position: 453  Y Position: 654
Value: 1  X Position: 412  Y Position: 778
Value: 2  X Position: 161  Y Position: 447
Value: 2  X Position: 243  Y Position: 612
Value: 2  X Position: 450  Y Position: 737
Value: 3  X Position: 243  Y Position: 447
Value: 3  X Position: 326  Y Position: 488
Value: 3  X Position: 202  Y Position: 571
Value: 3  X Position: 491  Y Position: 612
Value: 4  X Position: 201  Y Position: 529
Value: 4  X Position: 242  Y Position: 571
Value: 4  X Position: 408  Y Position: 654
Value: 4  X Position: 366  Y Position: 778
Value: 5  X Position: 202  Y Position: 487
Value: 5  X Position: 491  Y Position: 570
Value: 5  X Position: 161  Y Position: 653
Value: 5  X Position: 450  Y Position: 695
Value: 5  X Position: 367  Y Position: 736
Value: 6  X Position: 326  Y Position: 570
Value: 6  X Position: 491  Y Position: 777
Value: 7  X Position: 202  Y Position: 446
Value: 7  X Position: 409  Y Position: 611
Value: 7  X Position: 450  Y Position: 777
Value: 8  X Position: 285  Y Position: 446
Value: 8  X Position: 326  Y Position: 653
Value: 9  X Position: 283  Y Position: 487
Value: 9  X Position: 324  Y Position: 736
```

Organize all found image locations into 2D array

To order the list of cell objects into a sudoku board, a 2 dimensional array that is 9 by 9 is needed. The list needs to be sorted and the lowest 9 Y pixel values are appended to a new list. These newly added values are then sorted by their X pixel values. This results in a row on the sudoku board that is in proper position. This loop is performed until the entire board is ordered into a 2D array.

```python
def OrganizeCells(unorderedCellList):
    # returns a 9x9 2D list with the values in the correct position
    sudokuMatrix = []
    temp = []
    for j in range(9):
        unorderedCellList = sorted(unorderedCellList, key= lambda Cell: Cell.ypixel)
        for i in range(9):
            temp.append(unorderedCellList[i])  # takes the nine cells with the lowest y pixel values
        for i in range(9):
            del unorderedCellList[0]
        sortedRow = sorted(temp, key=lambda Cell: Cell.xpixel)  # sorting the nine cells by x pixel value
        temp = []
        sudokuMatrix.append(sortedRow)
    return sudokuMatrix
```

```
Value: 0  X Position: 355  Y Position: 437
Value: 0  X Position: 397  Y Position: 437
Value: 0  X Position: 438  Y Position: 437
Value: 0  X Position: 479  Y Position: 437
Value: 0  X Position: 149  Y Position: 478
Value: 0  X Position: 231  Y Position: 478
Value: 0  X Position: 355  Y Position: 478
Value: 0  X Position: 397  Y Position: 478
Value: 0  X Position: 438  Y Position: 478
Value: 0  X Position: 479  Y Position: 478
Value: 0  X Position: 149  Y Position: 519
Value: 0  X Position: 231  Y Position: 519
Value: 0  X Position: 273  Y Position: 519
Value: 0  X Position: 314  Y Position: 519
Value: 0  X Position: 355  Y Position: 519
Value: 0  X Position: 397  Y Position: 519
Value: 0  X Position: 438  Y Position: 519
Value: 0  X Position: 479  Y Position: 519
Value: 0  X Position: 149  Y Position: 561
Value: 0  X Position: 273  Y Position: 561
Value: 0  X Position: 355  Y Position: 561
Value: 0  X Position: 397  Y Position: 561
Value: 0  X Position: 438  Y Position: 561
Value: 0  X Position: 190  Y Position: 602
Value: 0  X Position: 273  Y Position: 602
Value: 0  X Position: 314  Y Position: 602
Value: 0  X Position: 355  Y Position: 602
Value: 0  X Position: 438  Y Position: 602
Value: 0  X Position: 190  Y Position: 644
… continued
```

```
2  7  3  8  0  0  0  0  0
0  5  0  9  3  0  0  0  0
0  4  0  0  0  0  0  0  0
0  3  4  0  6  0  0  0  5
1  0  2  0  0  0  7  0  3
5  0  0  0  8  0  4  1  0
0  0  0  0  0  0  0  5  0
0  0  0  0  9  5  0  2  0
0  0  0  0  0  4  1  7  6
```

Starting Sudoku Grid

Solve the 2D array sudoku puzzle

A recursive backtracking algorithm is used on the 2D array. It uses brute force by symmetrically increments each cell, when a condition/rule of the sudoku puzzle is broken (more than a single occurrence of a digit in a row, column, or 3x3 sub grid), return to the previous recursive function and try the next value, continue until a result is achieved that does not break any rules.
See link for very good explanation on the function.
https://en.wikipedia.org/wiki/Sudoku_solving_algorithms

```python
def possibleValue(y, x, n):
    # return True if value is not found in the row, column, or sub-grid
    for i in range(9):
        if sudokuMatrix[y][i].originalValue == n:  # check rows
            return False
    for i in range(9):
        if sudokuMatrix[i][x].originalValue == n:  # check column
            return False
    xSubMatrix = (x // 3) * 3
    ysubMatrix = (y // 3) * 3
    for i in range(3):
        for j in range(3):
            if sudokuMatrix[ysubMatrix + i][xSubMatrix + j].originalValue == n:  # check 3x3 sub-grid
                return False
    return True

def solve():
    # brute force recursive function that will back track if it gets into a dead end
    for y in range(9):
        for x in range(9):
            if sudokuMatrix[y][x].originalValue == 0:
                for n in range(1, 10):
                    if possibleValue(y, x, n):
                        sudokuMatrix[y][x].originalValue = n
                        solve()
                        sudokuMatrix[y][x].originalValue = 0
                return
    #once the solution is found assign the new values
    for y in range(len(sudokuMatrix)):
        for x in range(len(sudokuMatrix)):
            sudokuMatrix[y][x].newValue = sudokuMatrix[y][x].originalValue
```

```
2 7 3 8 0 0 0 0 0
0 5 0 9 3 0 0 0 0
0 4 0 0 0 0 0 0 0
0 3 4 0 6 0 0 0 5
1 0 2 0 0 0 7 0 3
5 0 0 0 8 0 4 1 0
0 0 0 0 0 0 0 5 0
0 0 0 0 9 5 0 2 0
0 0 0 0 0 4 1 7 6
```
Starting Sudoku Grid

```
2 7 3 8 4 6 5 9 1
6 5 1 9 3 7 2 4 8
9 4 8 5 1 2 6 3 7
7 3 4 2 6 1 9 8 5
1 8 2 4 5 9 7 6 3
5 6 9 7 8 3 4 1 2
4 2 6 1 7 8 3 5 9
3 1 7 6 9 5 8 2 4
8 9 5 3 2 4 1 7 6
```
Solved Sudoku Grid

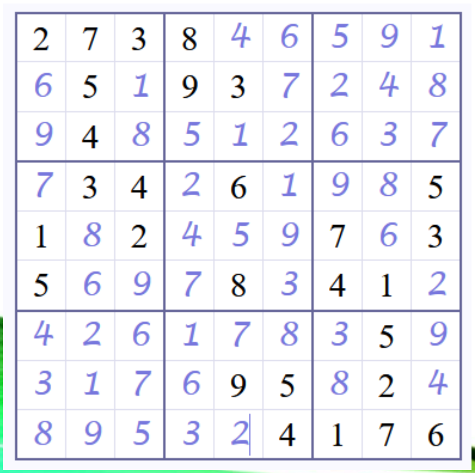
Use PyautoGUI to automate user inputs

Since the solution to the puzzle is now known, all that needs to be done now is enter the values. One could certainly enter the values by hand but since the process so far has been automated, why not also automate the input process. PyAutoGUI will be used to input the values into the puzzle.

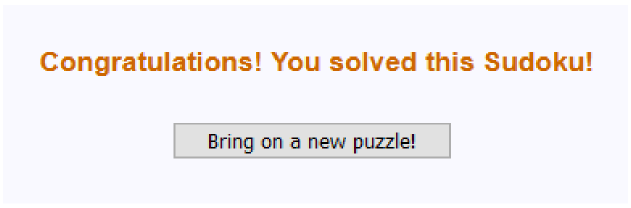
PyAutoGUI is a API that is certainly worthy its own presentation. It is used to automate keyboard and mouse inputs from python scripts. It was written by Al Sweigart who also wrote a book on it for beginner programmers.
Download PyAutoGUI: https://pyautogui.readthedocs.io/en/latest/#
Learn Python and PyAutoGUI: https://automatetheboringstuff.com/

Since the X, Y location for each tile on the screen are known, click on that location, then since correct value for each tile are known, enter that value with the keyboard. PyAutoGUI can easily do this for us.

```python
def completeTheSudoku(sudokuMatrix):
    # clicks on the tiles and enters the correct number
    for y in range(9):
        for x in range(9):
            if sudokuMatrix[y][x].originalValue == 0:
                pyautogui.click(sudokuMatrix[y][x].xpixel + 25, sudokuMatrix[y][x].ypixel + 25)
                #there is a bug that occurs when using both openCV and pyautogui in that the pixel locations are offset
                tempValue = str(sudokuMatrix[y][x].newValue)
                pyautogui.typewrite(tempValue)
```



2	7	3	8	4	6	5	9	1
6	5	1	9	3	7	2	4	8
9	4	8	5	1	2	6	3	7
7	3	4	2	6	1	9	8	5
1	8	2	4	5	9	7	6	3
5	6	9	7	8	3	4	1	2
4	2	6	1	7	8	3	5	9
3	1	7	6	9	5	8	2	4
8	9	5	3	2	4	1	7	6



Congratulations! You solved this Sudoku!