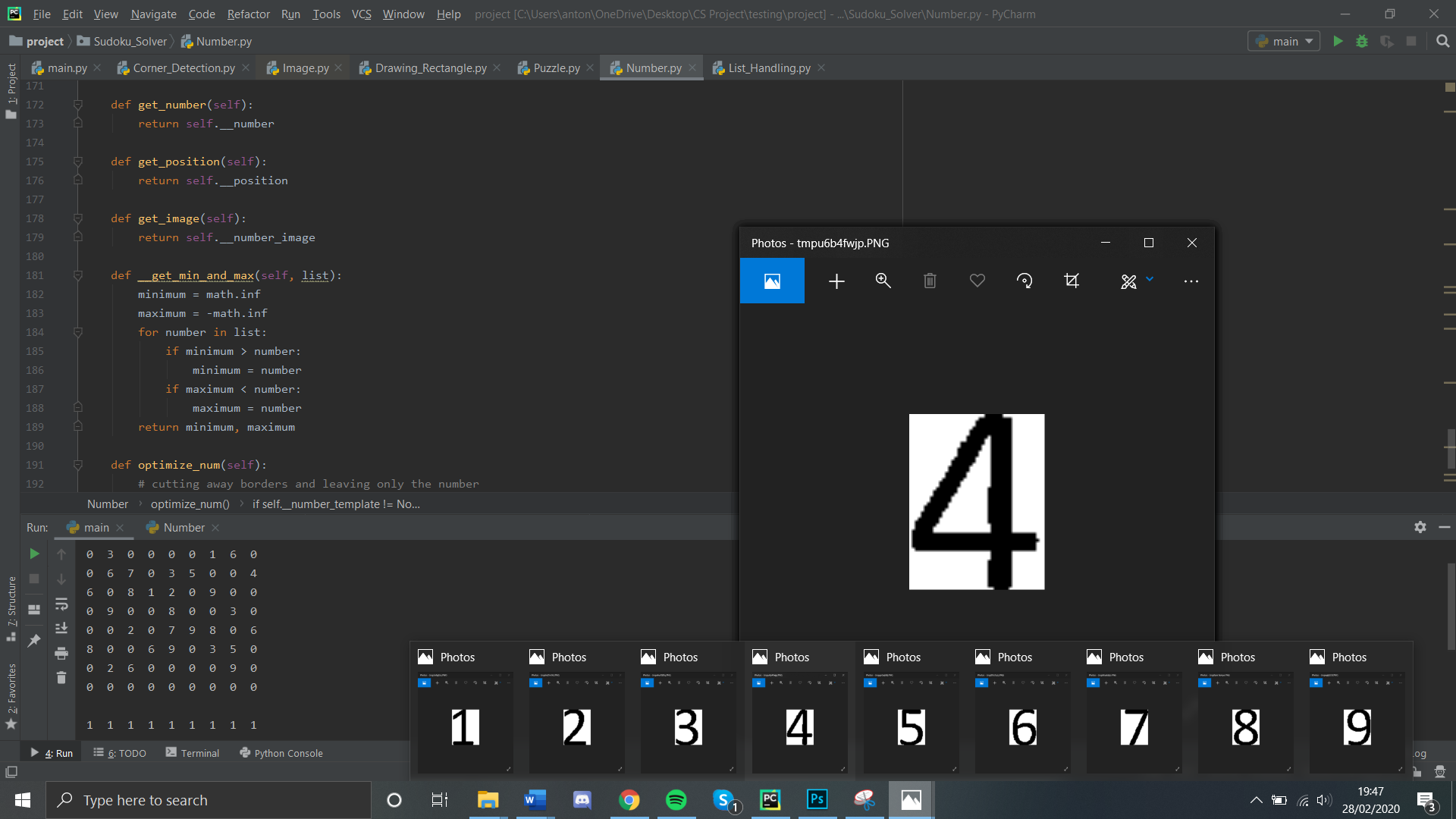
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Test** | **Input** | **Expected Output** | **Received Output** | **Passed?** |
|  | Getting a square from an array | [[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9],  [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],  [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],  [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],  [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],  [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],  [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],  [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],  [80, 81, 82, 83, 84, 85, 86, 87, 88, 89],  [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]]  Size -> 4x4  Top left: (5,5) | [[55, 56, 57, 58, 59],  [65, 66, 67, 68, 69],  [75, 76, 77, 78, 79],  [85, 86, 87, 88, 89],  [95, 96, 97, 98, 99]] | |  |  |  | | --- | --- | --- | | Bound\_y | Bound\_X | Row | | 9 | 9 |  | |  |  | 44 45 46 47 48 | |  |  | 54 55 56 57 58 | |  |  | 64 65 66 67 68 | |  |  | 74 75 76 77 78 | |  |  | 84 85 86 87 88 | | Passed |
|  | Getting a rectangle from an array | [[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9],  [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],  [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],  [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],  [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],  [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],  [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],  [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],  [80, 81, 82, 83, 84, 85, 86, 87, 88, 89],  [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]]  Size -> 7x6  Top left: (5,3) | [[67 68 69]  [77 78 79]  [87 88 89]] | |  |  |  | | --- | --- | --- | | Bound\_y | Bound\_X | Row | | 9 | 10 |  | |  |  | 67 68 69 | |  |  | 77 78 79 | |  |  | 87 88 89 | |  |  | 97 98 99 | | Passed |
| 1 | Compressing with a 3x3 square | [[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9],  [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],  [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],  [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],  [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],  [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],  [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],  [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],  [80, 81, 82, 83, 84, 85, 86, 87, 88, 89],  [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]] | [[11, 14, 17, 19],  [41, 44, 47, 49],  [71, 74, 77, 79],  [91, 94, 97, 99]] | [[11, 14, 17, 19],  [41, 44, 47, 49],  [71, 74, 77, 79],  [91, 94, 97, 99]] | Passed |
| 2 | Testing whether the compression algorithm works |  | An image with no noise, and just the grid with numbers |  | Not Passed  (later achieved as the result on the expected output) |
| 3 | Getting dynamic threshold list  With a dynamic size of 3 | [[ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9],  [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],  [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],  [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],  [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],  [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],  [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],  [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],  [80, 81, 82, 83, 84, 85, 86, 87, 88, 89],  [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]] | [[16, 20, 23],  [56, 60, 63],  [86, 90, 93]]  Aim of this test is to make sure that there is no remainder. If traditional divison was to take place with a 10 wide row and a dynamic size of 3, the output would be 4, and the 1 wide column at the end would corrupt results, appearing darker than the rest. | [[16, 20, 23],  [56, 60, 63],  [86, 90, 93]] | Passed |
| 4 | Detecting the first black pixel and set the starting pixel to the closest pixel to the top left | [  [w, w, w, w, w, w, w, w, w, w, w, w],  [w, w, w, w, w, w, w, w, w, w, w, w],  [w, w, w, w, w, b, b, b, b, b, b, b],  [w, w, w, w, w, b, w, w, w, w, w, w],  [w, w, w, w, w, b, w, w, w, w, w, w],  [w, w, w, w, w, b, w, w, w, w, w, w]  ] | |  |  |  | | --- | --- | --- | | Filter size | Return list | Starting coordinate | | 1 | Empty |  | | 2 | Empty |  | | 3 | Empty |  | | 4 | Empty |  | | 5 | Empty |  | | 6 | (5, 2), (5, 3), (5, 4), (5, 5) | (5,2) | | 1 (False, [])  2 (False, [])  3 (False, [])  4 (False, [])  5 (False, [])  6 (True, [(5, 2), (5, 3), (5, 4), (5, 5)])  Starting coordinate = (5,2) | Passed |
| 5 | Get hits algorithm changes direction correctly | [(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0), (7, 0)],  [(0, 1), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1), (7, 1)],  [(0, 2), (1, 2), (2, 2), (3, 2), (4, 2), (5, 2), (6, 2), (7, 2)],  [(0, 3), (1, 3), (2, 3), (3, 3), (4, 3), (5, 3), (6, 3), (7, 3)],  [(0, 4), (1, 4), (2, 4), (3, 4), (4, 4), (5, 4), (6, 4), (7, 4)],  [(0, 5), (1, 5), (2, 5), (3, 5), (4, 5), (5, 5), (6, 5), (7, 5)] | When centre is (3,2)  Order of pixels as they’re read:  (1, 0), (2, 0), (3, 0), (4, 0), (5, 0)  (1, 1) (5, 1)  (1, 2) ((3,2)) (5, 2)  (1, 3) (5, 3)  (1, 4), (2, 4), (3, 4), (4, 4), (5, 4) | (2, 0), (3, 0), (4, 0), (5, 0), (5, 1), (5, 2)  (5, 3), (5, 4), (4, 4), (3, 4), (2, 4), (1, 4)  (1, 3), (1, 2), (1, 1), (1, 0), (2, 0)  The last (2,0) is printed because the next pixel is caluclated at the end of the loop, so it printed but didn’t actually get checked again | pass |
| 6 | (console representation)  Whether the filter identifies the surrounding black pixels correctly | [1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 0, 1, 1, 1, 1, 1], [1, 1, 0, 1, X, 1, 0, 1], [1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 0, 1, 0, 1, 1]  Where X is the centre pixel, and 0 indicates black | Returns  Top : []  Right : (6, 3)  Bottom : (5, 5), (3, 5)  Left : (2, 3), (2, 2) | [[], [(6, 3)], [(5, 5), (3, 5)], [(2, 3), (2, 2)]]  Where lists are returned in the order top, right, bottom, left. | Passed |
| 7 | (graphical representation)  Whether the filter identifies the surrounding black pixels correctly |  | When filter size is 7:  Where the green is the gap, and the red is the pixels that are highlighted | The white pixel represents the centre of the filter and the blue represent the square that surrounds it. | Passed |
| 8 | Finding next centre |  | With current vector being (0 , 1)  Hits on the right:  [(234, 225), (234, 226), (234, 227), (234, 228), (234, 229), (234, 230), (234, 231), (234, 232), (234, 233), (234, 234), (234, 235)]  Therefore the average of these values will be the next centre;  Next centre will be (234, 230) | \*Outlined in red | Passed |
| 9 | Changing vector |  |  |  | Failed |
| 10 | Changing vector |  | Where the green area is where the change in vector will happen | the below vector change happened in the below arrangement of hits | Passed |
| 11 | Without careful engineering of the filters, it might run into a dead-end |  | Where the red is the followed path of the centres of the filter  Green is the point where the vector changes |  | Passed |
| 12 | Is cycle complete |  |  |  | Passed |
| 13 | Skipping noise |  | When the filter tracer completed a cycle it shoult return true for noise, and completely ignore that black pixel.  (circled) |  | Passed |
| 14 | No corners | Passing in a blank image: | Exception: no corners found | “Exception: no corners were found in the image”  A handler that I built in throws an exception if there are no corners detected | Passed |
| 15 | Identifying corners using longest distance |  | 0, 281  46, 293  88, 283  138, 296 | 0, 281  46, 293  88, 283  138, 296 | Passed |
| 16 | Identifying corners using equations of lines |  | (from centres, these are not the actual corners) | These are the corners, as they’re computed from all the centres | Passed |
| 17 | Normalising corners | [(375, 635), (798, 530), (699, 96), (268, 217)] | [(268, 217), (699, 96), (798, 530), (375, 635)] | [(268, 217), (699, 96), (798, 530), (375, 635)] | Passed |
| 18 | Rotation and crop |  |  |  | Passed |
| 19 | Noise handling, corner finding and rotation with crop |  |  |  | Passed |
| 20 | Identifying numbers |  |  |  | Passed |
| 21 | Taking out the numbers out of the template |  | Numbers separated from eachother | (below) | Passed |



State transition table for cut\_out\_number:

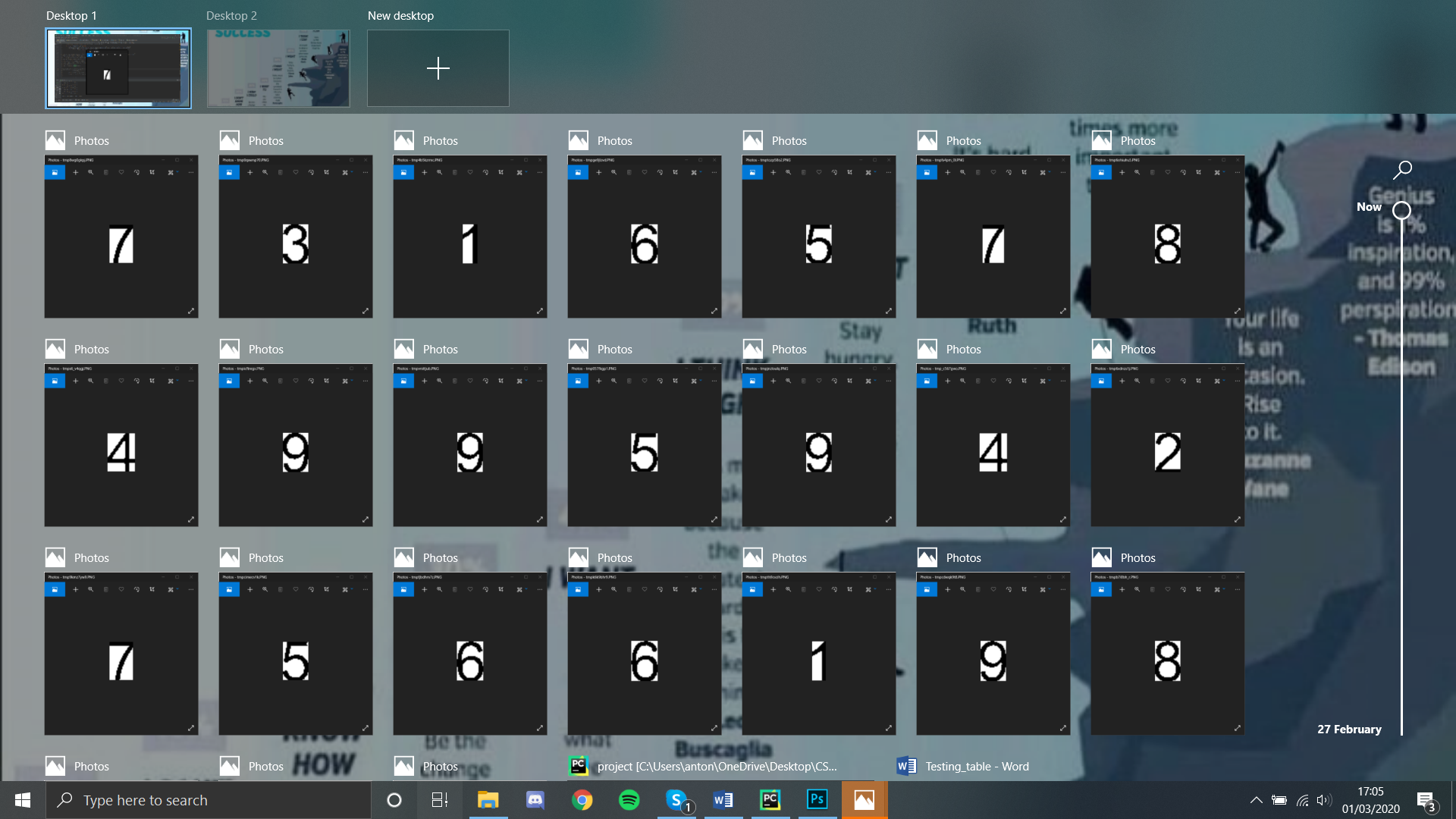
Input column: Expected Output:

[1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1], [[0, 0, 0, 0,], [0, 1, 1, 0,]]  
[1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1]]

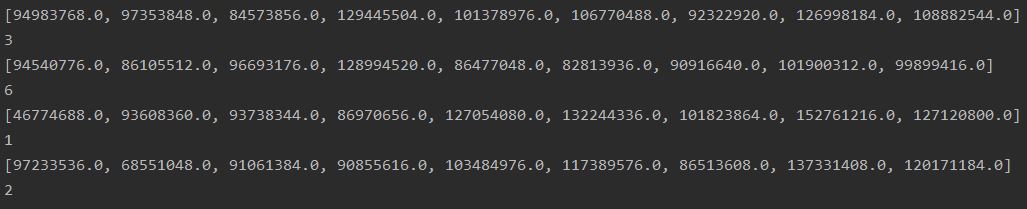
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Column | Left | Right | Step\_L | Step\_R | State\_before |
|  | 3 | 9 | -1 | 1 | (3,9) |
| [0, 0] | 2 |  |  |  | (3,9) |
| [1, 1] |  | 8 |  | -1 |  |
| [1, 1] | 3 |  | 1 |  | (2,8) |
| [1, 1] |  | 7 |  |  |  |
| [0, 0] |  |  |  |  | (3,7) |
| [1, 1] |  | 6 |  |  |  |
| [0, 0] |  |  |  |  | (3,6) |
| [0, 0] |  |  |  |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 23 | Cutting | matrix = [  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],  [1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1],  [1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 1],  [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]] | [[0, 0, 0, 0,],  [0, 1, 1, 0,]] | [[0, 0, 0, 0,],  [0, 1, 1, 0,]] | passed |
| 24 | Cropping out numbers |  | An eight on its own |  | Passed |
| 25 | Extreme number crop |  | An eight on its own |  | Passed |
| 26 |  |  | 1 should be cropped out of the image |  | Fail |
| 27 |  | Just by recursivelly calling the same function with the coordinates it finished with | 1 should be cropped out of the image |  | passed |

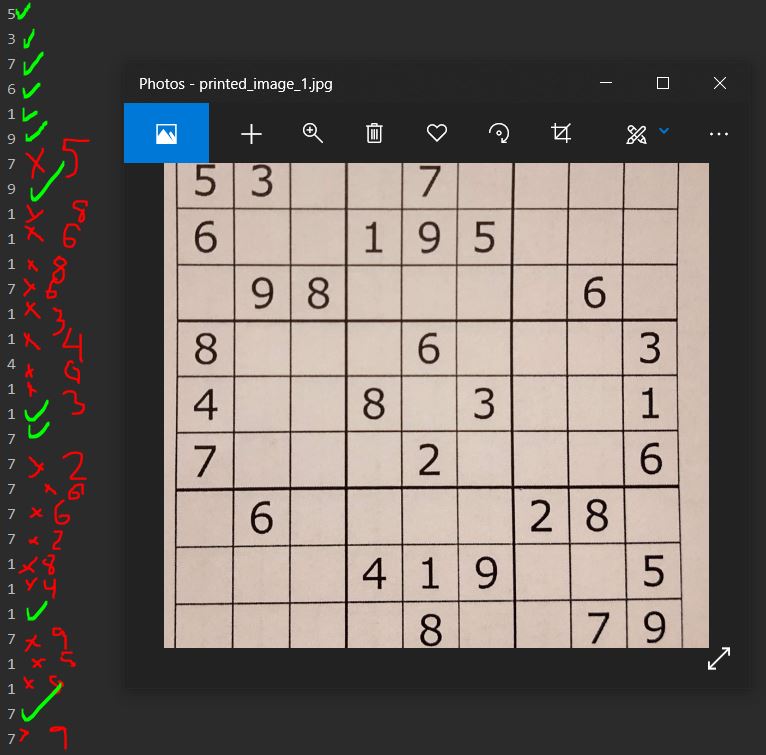
**Successful** cropping of images with no noise

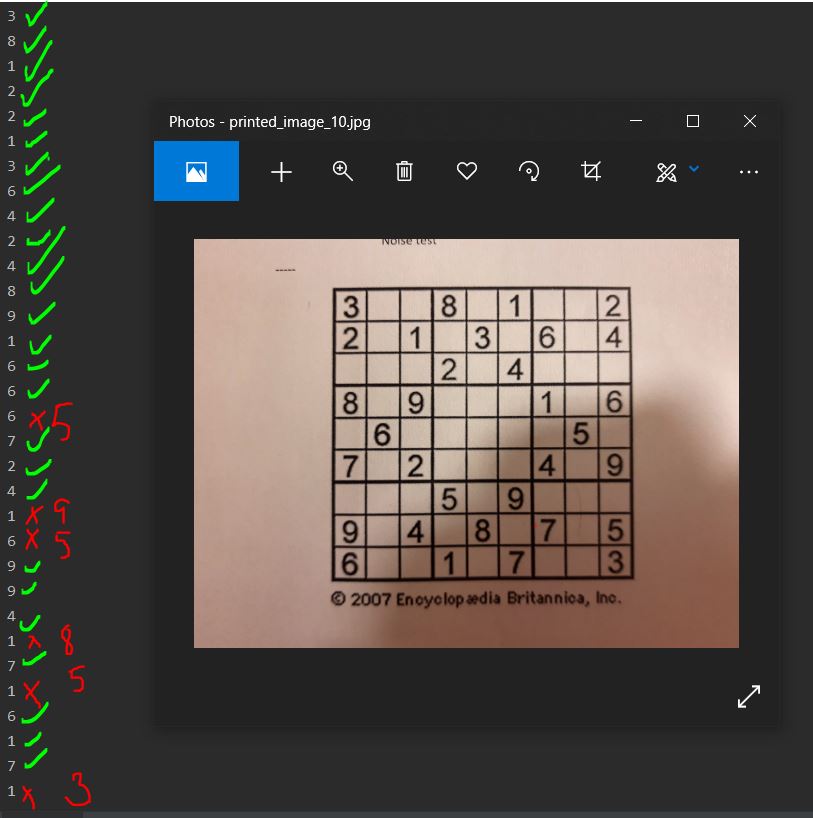


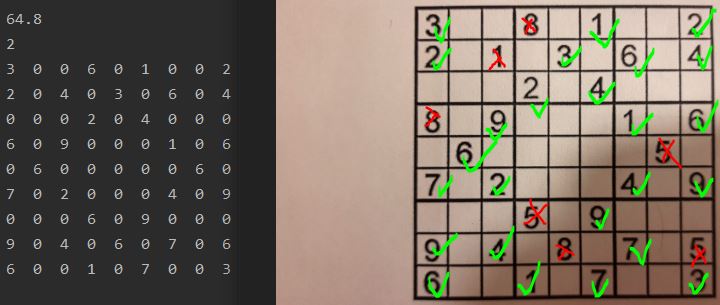
Because of the different fonts used or some other anomalies in the numbers, some templates might match better than others. In a test to identify the number 3, 8, 1, 2, the following scores were observed, and, depending on the method used, the minimum was taken.

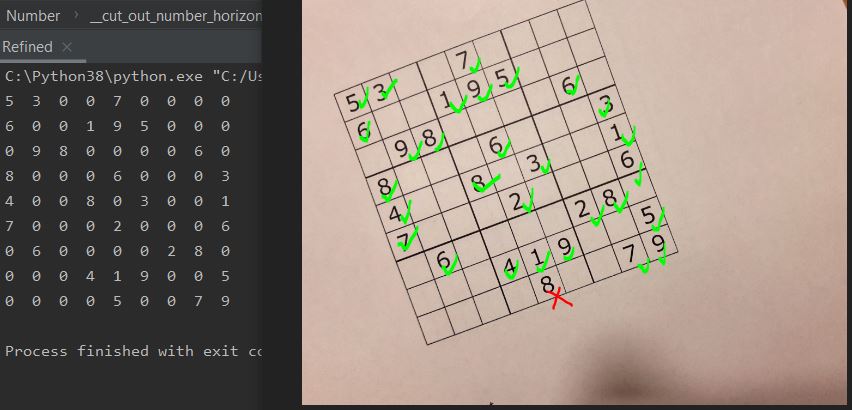
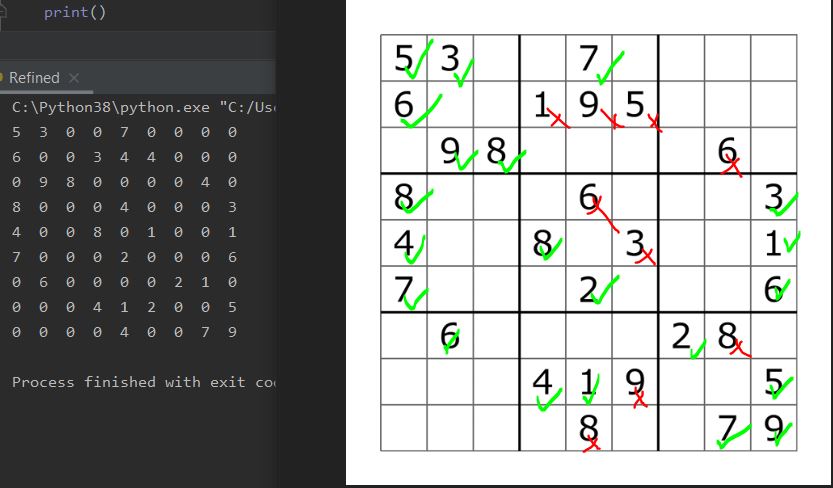
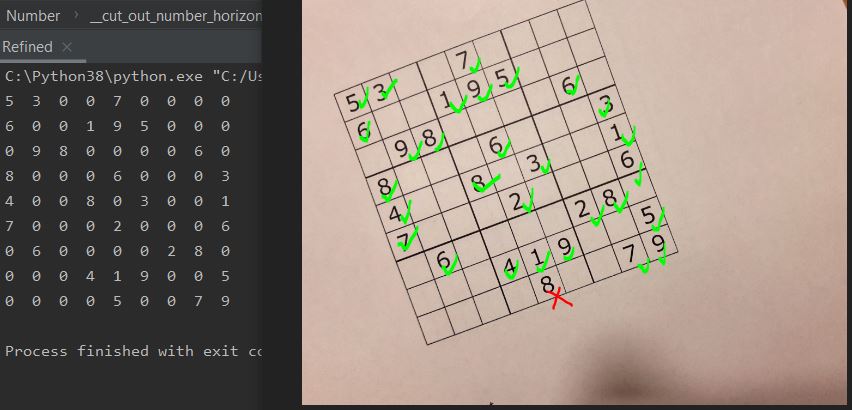
****

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **No** | **Input** | **Expected** | **Output** | **Passed?** |
| **1** |  | **The expected result for all these, is that the main quadrilateral puzzle should be cropped out the original image**  **The square cut out of its surroundings**  **The square cut out of its surroundings** |  | **pass** |
| **2** |  |  | **pass** |
| **3** |  |  | **pass** |
| **5** |  |  | **pass** |
| **6** |  |  | **Fail** |
| **7** |  |  | **Passed** |
| **8** | **Note that this image is the same, only quite blurred** | 0 0 0 0 8 4 4 0 0  0 0 0 7 0 0 0 0 0  8 0 8 0 0 4 6 0 0  7 0 4 0 0 0 0 8 0  8 0 0 0 1 0 0 0 4  0 8 0 0 0 0 8 0 4  0 0 6 8 0 0 8 0 7  0 0 0 0 7 8 0 0 0  0 0 7 8 4 0 0 0 0 | **It passed identifying corners, but classifying numbers was dreadful** |

****

****

****

****

These tests were a success because they demonstrated that I can predict with 80% success-rate the numbers in the grids

One thing that came up from this is that the thickness of numbers matter, but this can be attributed to the fonts and the divisor in the dynamic exaggerator

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 28 | Adding numbers to the image |  | And so on, 1s throughout the grid |  | Passed |
| 29 | Adding numbers to a warped image |  | Numbers added INSIDE the squares |  | Passed only along the x axis, and not the y, because it requires more than a linear approach |
| 30 | Adding numbers to the grid |  | Add a row of ones along the top row:  5 3 1 1 7 1 1 1 1  6 0 0 1 9 5 0 0 0  0 9 8 0 0 0 0 6 0  8 0 0 0 6 0 0 0 3  4 0 0 8 0 3 0 0 1  7 0 0 0 2 0 0 0 6  0 6 0 0 0 0 2 8 0  0 0 0 4 1 9 0 0 5  0 0 0 0 8 0 0 7 9 |  | passed |
| 31 | Reading from a website | https://cdn.britannica.com/42/97142-131-E3E24AA5/sudoku-puzzle-games.jpg |  |  | Passed |