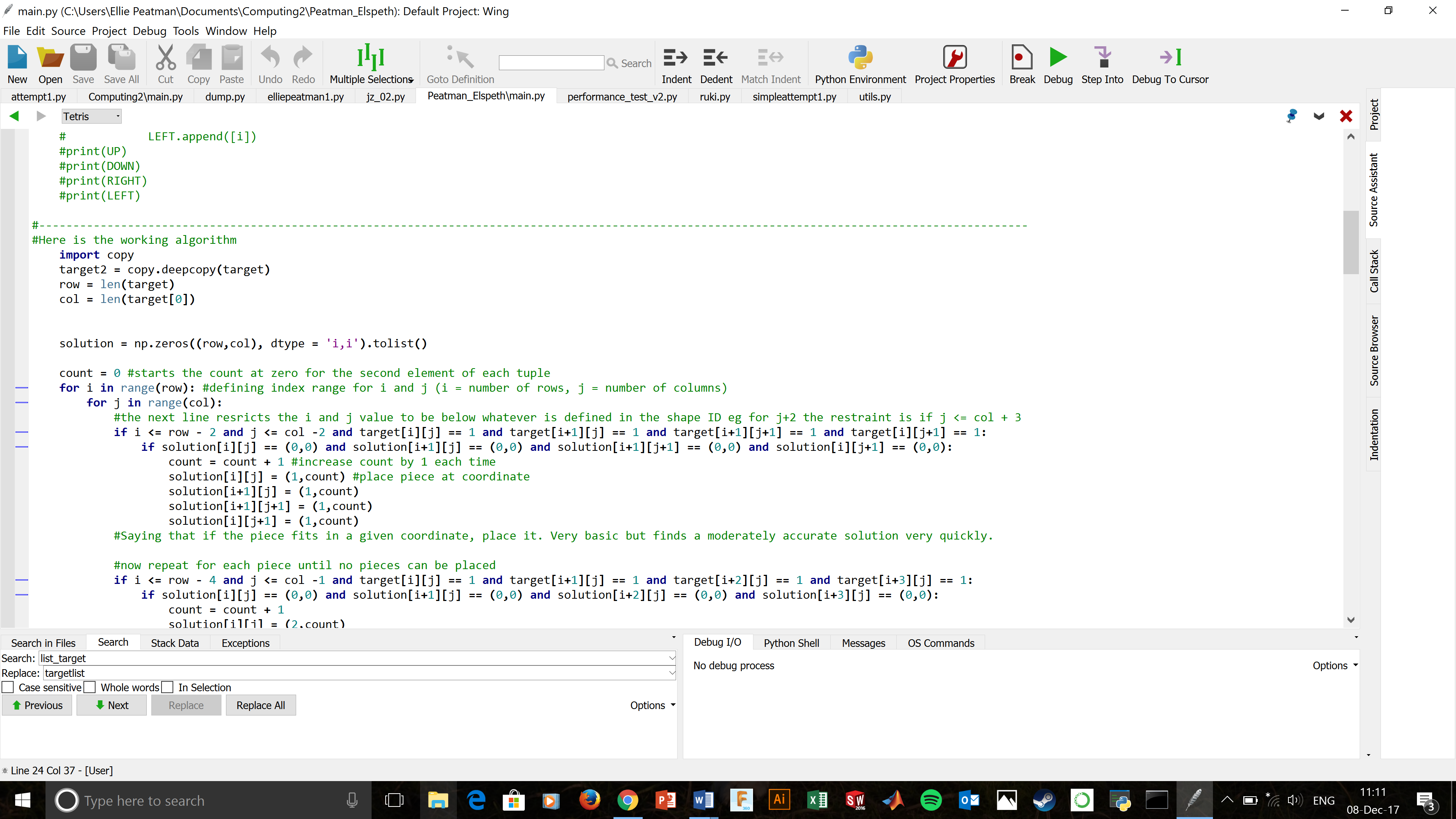
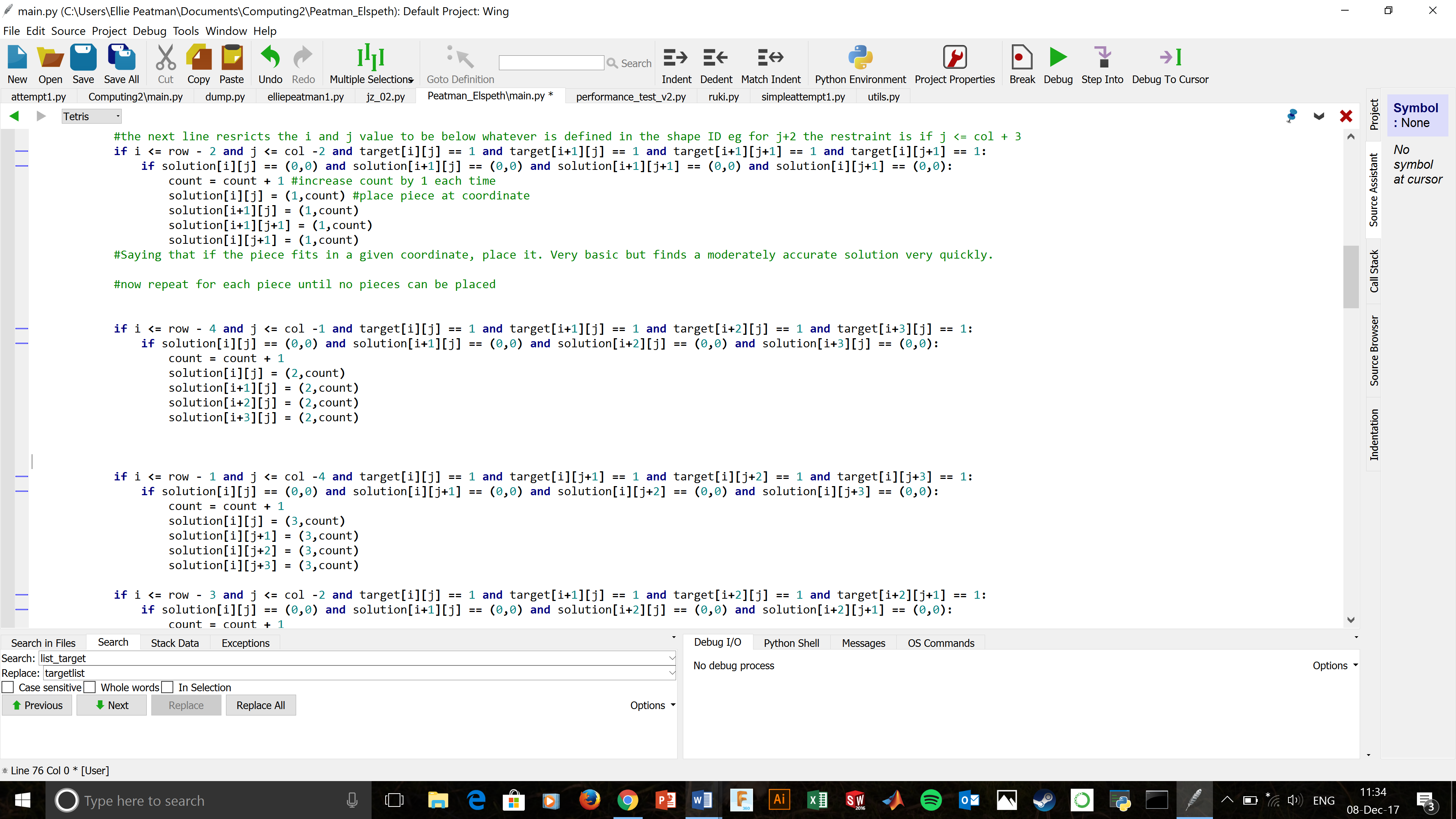
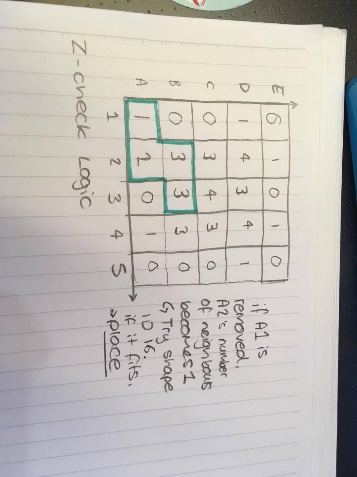
My submitted algorithm was very simple. I used the deepcopy function to clone the target matrix (to target2) so that if any changes were made to the target they wouldn’t affect the solution or the size of the target (len(target) wouldn’t change). This was before the deepcopy was added to performance test. I then made my solution - an embedded list the size of the target which would be filled with the tuples (shapeID,count).



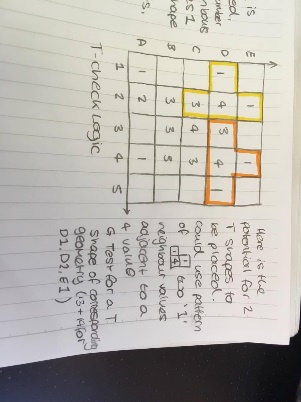
I used numpy to create an array filled with zeroes, with the size determined by the number of rows (len(target) which I made = row) and the length of the first row, or the number of columns (len(target[0]) which I made = col). The data type is integers in tuples (dtype = ‘i,i’). I then converted this array to a list by using the .tolist function.

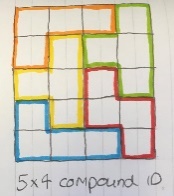
I then defined count = 0 as it would increase by 1 with each piece placed, followed by 2 for loops containing if statements for each shape ID. These for loops defined the range for i (number of rows or the negative y-axis) to be from zero to len(target) and for j (number of columns or positive x axis) to be from zero to len(target[0]). For each shape ID however, the coordinates of the other 3 blocks relative to [i][j] would vary between i-2 to i+3 for example, so to avoid an index error, for shape 1 (square) I set the condition that I had to be less than or equal to row – 2 as the coordinates went up to i+1. This prevented the algorithm trying to check if a coordinate that didn’t exist in the matrix had a value. I then in the same **if** statement made sure the coordinates of all the points in the shape ID had a value of 1 **and** the corresponding coordinates in the solution list didn’t already have a piece placed (value == (0,0)). If all these conditions were satisfied (if the piece fitted) then the count was increased by 1 and the piece was placed in the solution. Here is an example for shapeID 2:

After doing this for each shapeID I returned the solution. This algorithm is extremely fast and basic, returning between 60% and 80% accuracy for most targets, and was the best fit to the brief that I could code.

However, to improve on this algorithm, I would’ve like to first add functions that would check for outlying shapes. In many targets, there are exterior geometries that would only fit one shapeID in a 100% accuracy solution. To find these I would make 3 new function to check for T, L and Z shapes. For each of these, their number of axial (non-diagonal) neighbours should be counted and values put into an adjacency matrix with matching coordinates to the target. Then looking at the blocks with a value of 1

For Z shapes, if the block with value 1 is removed from target2 and neighbours was updated, the number of neighbours for the adjacent piece would then become 1, which is a unique pattern to outlying shapes of this geometry. If this pattern fits for 3 of the 4 pieces, then you can see that no other shape would fit there in a 100% accuracy solution.

For outlying T shapes, 2 or 3 blocks with one neighbour are almost always adjacent to a block with value 4 (in some instances this is a 3). Identifying this relationship using if statements, similar to my piece placing function would allow the algorithm to place these known pieces first.



Equally if there were 3 blocks that were outlying with only each other and the fourth block of the shape as neighbours (when removed each becomes 1 respectively) then an L shaped piece would be the only solution.

Once these outlying parts of the target that only have 1 shape ID in a perfect solution are found and removed, the next part of the algorithm comes into play. For a perfect solution, this should be at 100% accuracy so far, with a big unsolved group of blocks in the centre.

From here, I would’ve made a dictionary of compound shapes e.g. each combination and orientation of 3 shapes placed together and used a recursive function to test to see if one of them fitted, from largest to smallest. The largest would vary with len(target) so that the pieces being tested were in proportion to the size of the target. This may leave gaps but would be a quick way to place large pieces.

After this I would’ve run my submitted algorithm to test the rest of the remaining blocks.

Ellie Peatman

CID 01185117