Thue-morse sequence:

The program starts off with function main where the user is asked to input a number which is held in the variable n. A try-except statement is then executed, with a custom exception being raised in thue\_morse if the number given is negative.

In thue\_morse, the variable t is created and given the value 0, the first character in the sequence. This variable is then used inside a loop that adds the two halfs of the equation tn-1 and inverse 𝒕𝒏−𝟏 together to create a thue-morse sequence, this loop occurs for the number of times given by n.

To get the inverse of the first half of the string, another auxiliary function is called named xor, as a exclusive or operation is acted upon the string. This works by selecting each character found in the given iteration of t from thue\_morse and comparing it against the value 1. Using conditional operators to create a xor operator, the conditional statement will add 1 to a new string called inverse if the slice in t is a 0 and a 0 if the t slice is a 1.

This inversed string is returned to thue\_morse where it is added to its complement to create a longer thue\_morse sequence, which then can be used in the next iteration to create a larger sequence until n iterations have been completed, the final sequence is then returned to function main to be printed into the terminal.

Square free

Print3Blocks

return spacedBlocks[:-1]  # Returns the formatted string, without the last whitespace as it is unneccessary - more elegant than using a conditional statement

Count squares

        # Due to slices being half-inclusive can't use signed (negative) integers in the slices as I won't be able to handle all cases the same

        # To select the last two characters I'll need to do something like [-2:], whilst other characters would be [-3:-1]

        # So instead just repeat the above but start off by +1

        # The number of iterations a search needs to go through and check an entire string will never exceed the length of the string

    for search\_group in range(1, (len(s)//2)): # Will end loop after reaching half of the length of the string, as no more matches are possible beyond this

                # the addition to i represents the i value in the next iteration or one after that

                # eg: the slices are like: is 1st iteration:2nd == 2nd:3rd?

                elif(int(s[(i\*search\_group) + offset:((i + 1)\*search\_group) + offset])

# Process:

# select each group of 2 from start and continue till the end

# Then do again but this time starting from the end

# Increase search group to 3 and repeat

# If search group greater than half of total length break as no more squares are possible

# len(s)//search\_group would give the number of possible iterations for a search\_group without going out of bounds