Problem 1 - Sudoku

In order to solve this problem, I follow the following YouTube explanation: https://www.youtube.com/watch?v=G_UYXzGuqvM&ab_channel=Computerphile

My code consists of three functions: get_input, constrain, solve

get_input function contains script to get input from the user.

In CSP, we define variable, domain and constrain. For the sudoku problems, variable is the empty tiles, domain is the possible numbers (1 to 9), constrain is the rule. In my implementation, I define the constraint in the constrain function.

```
def constrain(y,x,d):
    for i in range(0,9):
        if grid[y][i] == d:
            return False
    for i in range(0,9):
        if grid[i][x] == d:
            return False
    x0 = (x//3)*3
    y0 = (y//3)*3
    for i in range(0,3):
        for j in range(0,3):
            if grid[y0+i][x0+j] == d:
                return False
    return True
```

```
def solve():
    global grid, solution
    for y in range(9):
        for x in range(9):
            if grid[y][x] == 0:
                for d in range(1,10):
                    if constrain(y,x,d):
                        grid[y][x] = d
                        solve()
                        grid[y][x] = 0
                return
    solution = np.copy(grid)
```

The solve function contains the main code to solve the sudoku. The idea is that if a number can be used in a given tile, we write it. If there is no number that we can assign in the given tile, we backtrack the process.

Problem 2 – Nonogram

The following blog explains how to solve nonogram programmatically:
https://towardsdatascience.com/solving-nonograms-with-120-lines-of-code-a7c6e0f627e4
I followed the main idea and made my own approach.

My code consists of four functions: get_input, generate_all_opts, overlap, update_opts.

get_input function contains code to get the user input.

In the beginning, my code generates all possibilities for each side and top's rules. I call the numbers that define how many and which blocks should be black for a row or a column, such as '7 0 0', a rule. I implemented this in the generate_all_opts function.

```
def generate_all_opts(places, groups):
    n_empty = places - (sum(groups) + (len(groups) - 1))
    opts = combinations(range(len(groups) + n_empty), len(groups))

    opts_matrix = []
    for opt in opts:
        opt_vec = [-1] * places
        start = 0
        for i in range(len(opt)):
            for j in range(groups[i]):
                opt_vec[start + opt[i] + j] = 1
            start = start + groups[i]
        opts_matrix.append(opt_vec)
    opts_matrix = np.array(opts_matrix)

    return opts_matrix
```

opts will have all the possibilities for a given rule as explained in the blog. After that I convert each combinations output to a vector of 1 and -1, with 1 being star and -1 being space, and keep all of them inside opts_matrix.

Once I get all the possibilities for all rules, I calculate the overlap. This is done in the overlap function. Overlap means, for a given block, all of the possibilities of a given rule agree that it should be either star or space. This is done by taking the summation of all the possibilities vectors. Since 1 represents star and -1 represents space, a vector element equals to the number of the possibilities means star agreement while a vector element equals to negative number of the possibilities means space agreement. Once we have the overlap indexes, we update the grid (rows and colums).

```python
def overlap(opts_matrix, place_proc):
    filtered_opts = update_opts(opts_matrix, place_proc)

    if len(filtered_opts) > 0:
        sum_filt_opts = np.sum(filtered_opts, axis=0)
        pos_overlap_idxs = np.where(sum_filt_opts == len(filtered_opts))
        neg_overlap_idxs = np.where(sum_filt_opts == -len(filtered_opts))

        return pos_overlap_idxs, neg_overlap_idxs, filtered_opts
    else:
        sys.exit('There is no solution')
```

The last function is update_opts function. In this function I delete all the possibilities that are obviously wrong. That is, if a given row or column says that a particular block is either star or space while a possibility says otherwise, then it will be deleted or, to be precise, ignored.

```python
def update_opts(opts_matrix, place_proc):
    filtered_opts = []
    for opt_vec in opts_matrix:
        opt_pass = True
        for i in range(len(place_proc)):
            if place_proc[i] == '*' and opt_vec[i] == -1:
                opt_pass = False
                break
            elif place_proc[i] == '-' and opt_vec[i] == 1:
                opt_pass = False
                break
        if opt_pass:
            filtered_opts.append(opt_vec)
    filtered_opts = np.array(filtered_opts)

    return filtered_opts
```

The code will keep on repeating overlap and update possibilities until all of the rules only have one possibility, or, in other words, the puzzle is solved.