20/20

1A) Correct.

1B) Correct.

1C) Correct. Cool example.

2A) Correct

2B) Correct.

2C) Correct.

Exercise Sheet 1

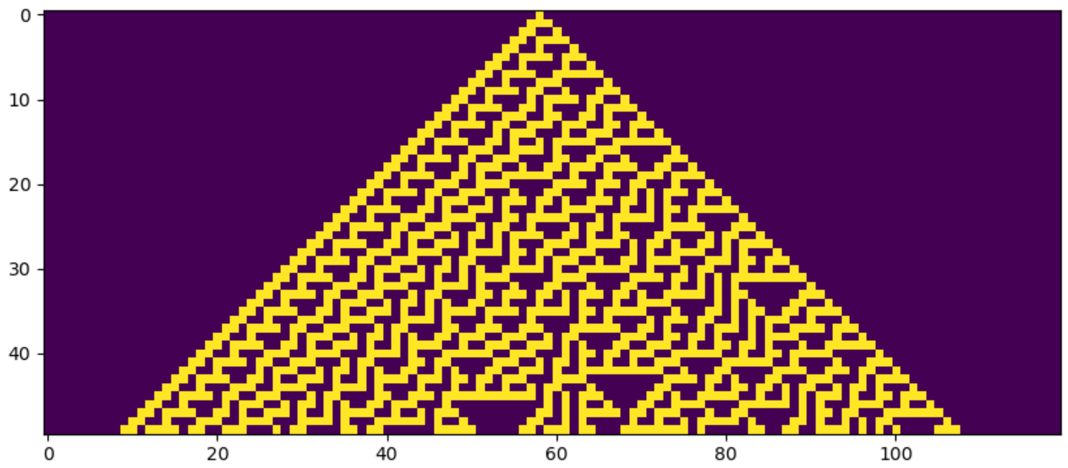
Pulkit Kukreja, Xiongxiao Wang

This Exercise has been done with Python 3.6

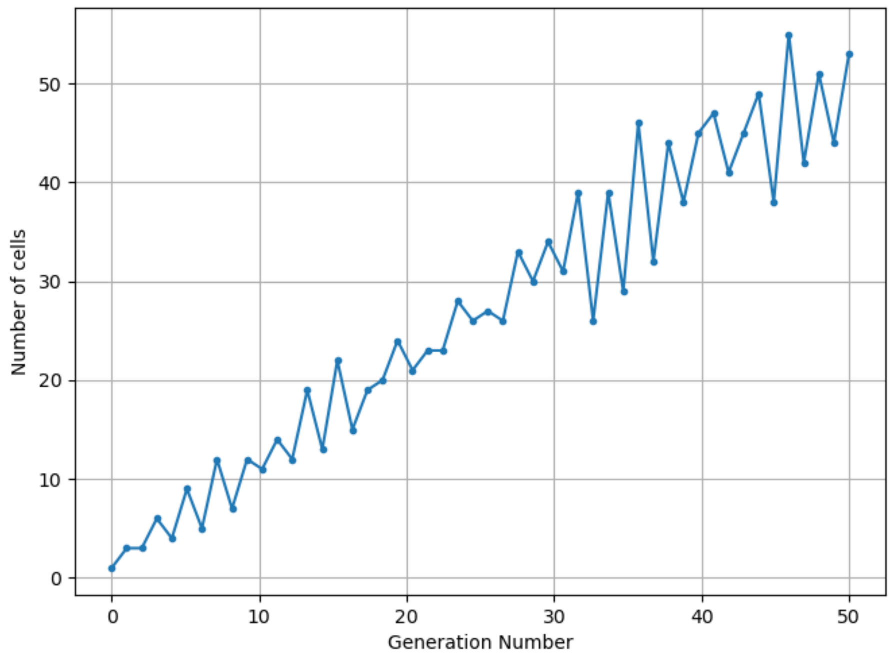
1. Rule N

A) The main loops of the code:

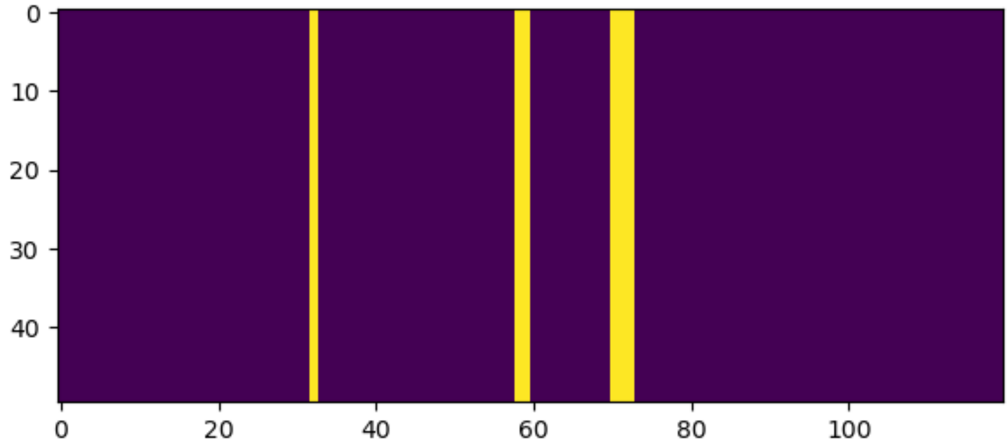
```
n_gen = 50
new = ini_state
new_mat = []
i = 1
old_new = new
#Update loops
#number of generations loop
while i <= n_gen:
    j = 2
    #The loop that goes over cells in each generation
    #Also ensures that j doesn't move out of the boundary
    #(could have used boundary conditions here)
    while j>=(num_par // 2) and j< (N_cells - (num_par //
    2)):
        numb = ''
        #The loop that concatenates the previous state
        for k in range(-1*(num_par//2), (num_par//2)+1):
            #old_new has the data of the prev generation
            numb = numb + str(old_new[j+k])

        new[j] = f_rule[int(numb, 2)]
        j += 1
    old_new = new
    #new_mat has data of all the generations
    new_mat.append(new)
    #Make the values of new back to zero
    new = np.zeros(N_cells, dtype=int)
    i += 1
```

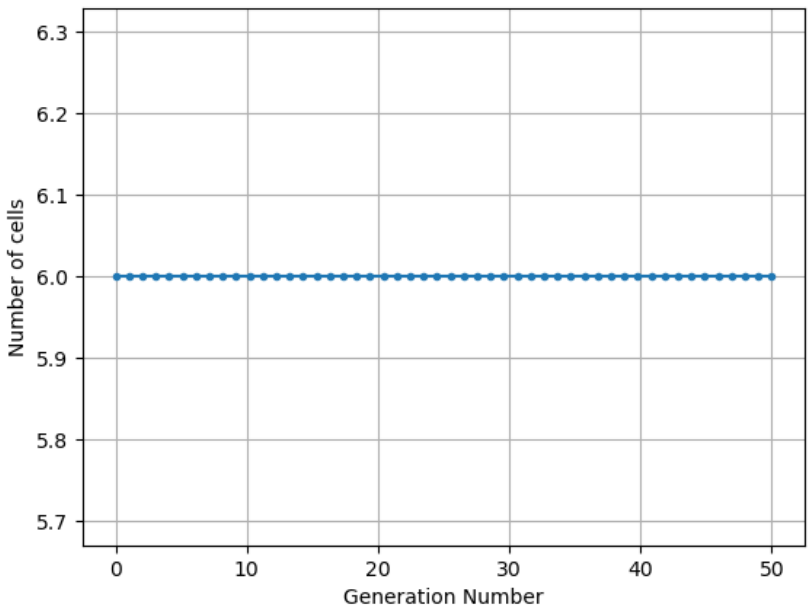


B) Time dependence of the number of cells:



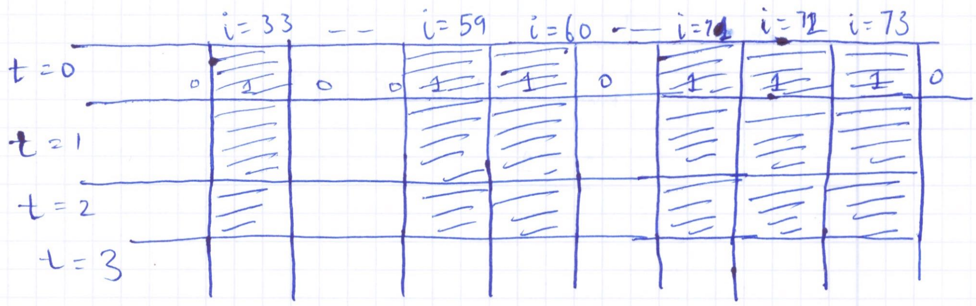
C) Rule 204 produces exactly the same configuration as the initial Configuration.
For e.g. In the initial configuration, there are 6 cells present at i = 33, 59, 60, 71,72, and 73. The image below shows evolution of the system.





	7	6	5	4	3	2	1	0
(a,b,c)	111	110	101	100	011	010	001	000
f(a,b,c)	1	1	0	0	1	1	0	0

$$[1\,1\,0\,0\,1\,1\,0\,0]_2 = [204]_{10}$$

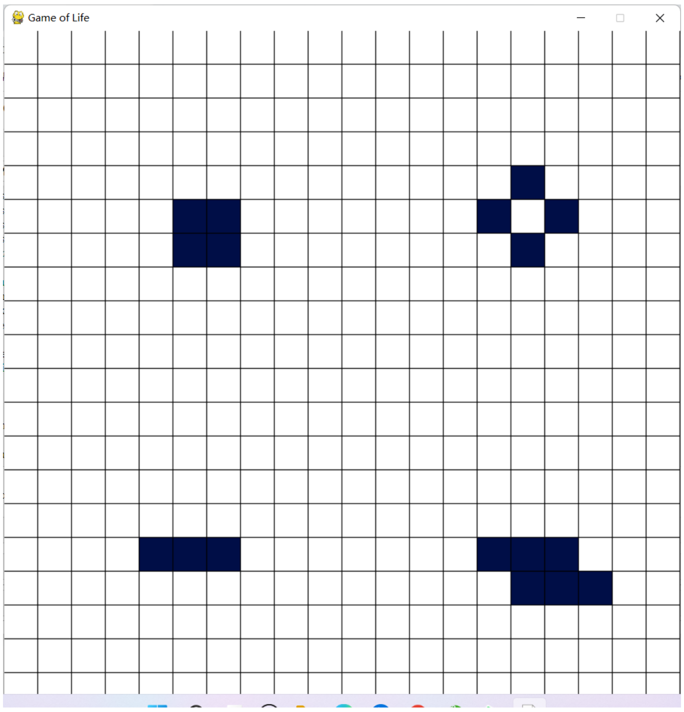


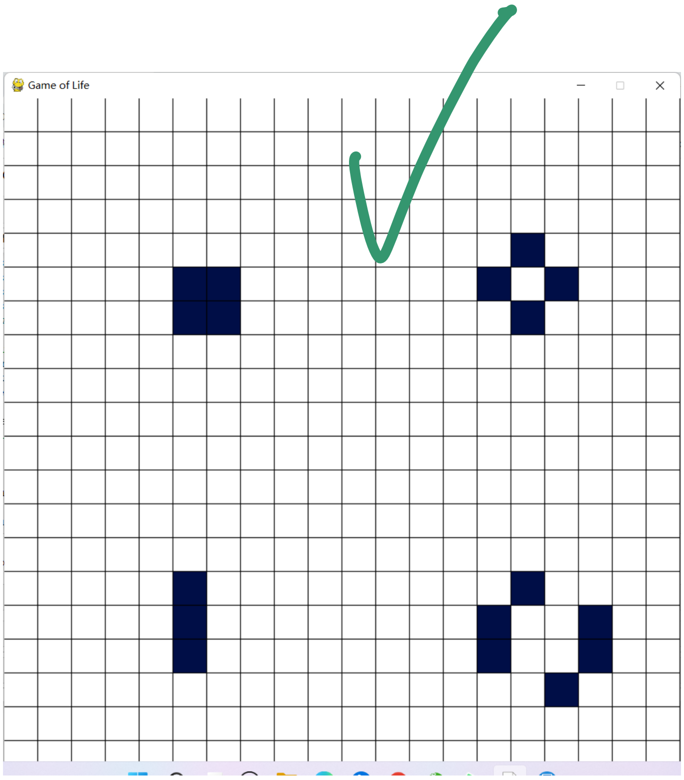
2. Game of Life

A) The main part of the code: (grid.py)

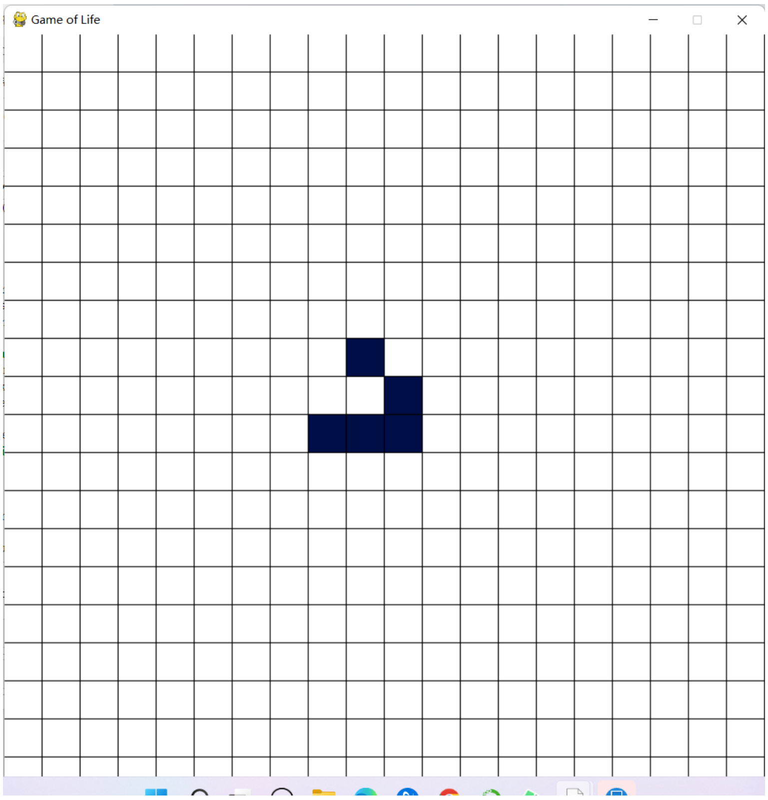
```
def Conway(self, off_color, on_color, surface): # on_color is
#black here ,off_color is white
        for x in range(self.rows):  # the rule of Conway
        #game of life
            for y in range(self.columns):
                state = self.grid_array[x][y]
                neighbours = self.get_neighbours(x,y)
                if state == 0 and neighbours == 3: # if the
                #cell is dead and the number of neighbours
                #are 3, the cell will alive
                    next[x][y] = 1
                elif state == 1 and (neighbours<2 or
                neighbours >3):# if the cell is living and is
                #underpopulated or overpopulated, turns to
                #dead
                    next[x][y] = 0
                else:
                    next[x][y] = state
        self.grid_array = next
def get_neighbours(self,x,y): # get the number of
#neighbours in period boundary condition
        total = 0
        for n in range(-1,2):
            for m in range(-1,2):
                #Periodic Boundary conditions
                x_edge = (x+n+self.rows)%self.rows
                y_edge = (y+m+self.columns)%self.columns
                total += self.grid_array[x_edge,y_edge]

        total -= self.grid_array[x][y]
        return total
```

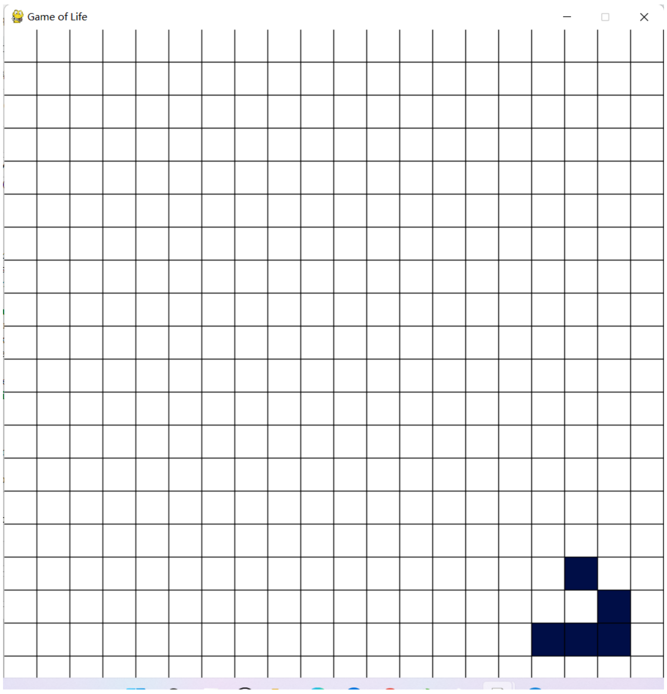
See the two snapshots of the configurations. (The bottom two configurations only change in time)

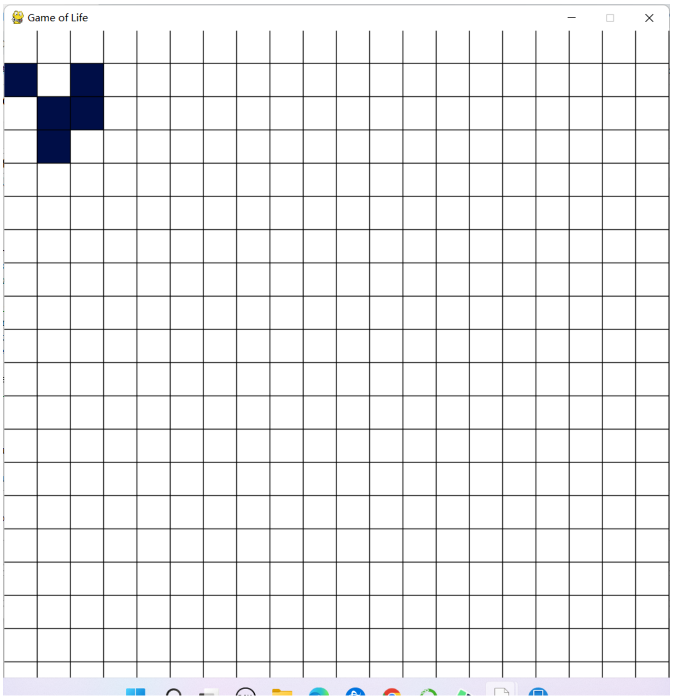




B) Some snapshots of the motion of the glider has been taken. (See the .gif files in the additional folder)







C) This is best shown in the animation attached with this file. From the variation shown below, it's evident that the number of live cells fall to zero after 130 generations.