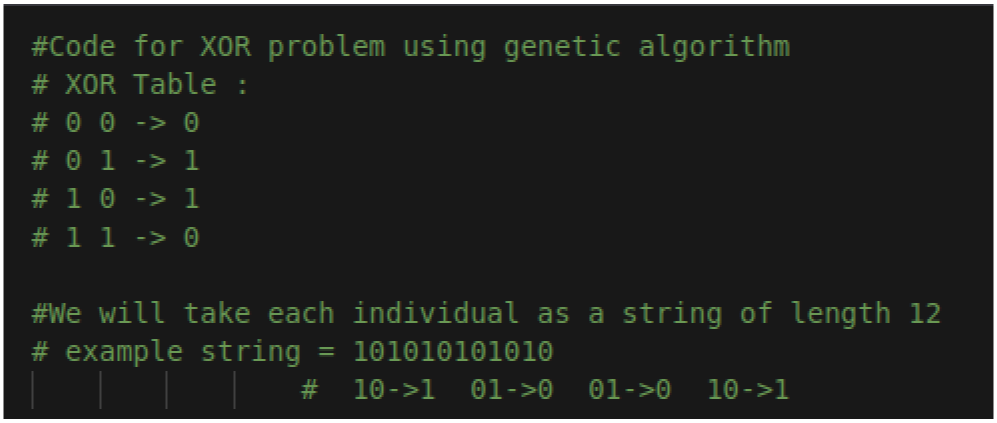
ML LAB ASSIGNMENT 5

Mohammed Junaid Anwar Qader
21CSB0B36

Q. Implement a genetic algorithm program to successfully classify examples in the 2 input EX-OR problem using Gabil's method.

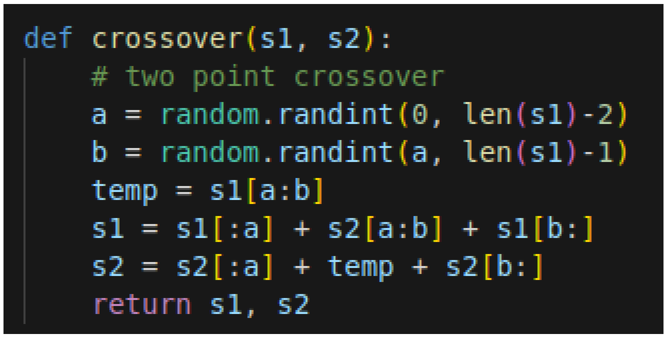


```
#Code for XOR problem using genetic algorithm
# XOR Table :
# 0 0 -> 0
# 0 1 -> 1
# 1 0 -> 1
# 1 1 -> 0

#We will take each individual as a string of length 12
# example string = 101010101010
                #  10->1  01->0  01->0  10->1
```

The string we will be using is a 12 length one, in which its a part of 3x4 length strings
Each 3 length string is of the form ABO where A, B are inputs and O is the output.
So if a string is 101010101010
So it means:

A	B	O
1	0	1
0	1	0
1	0	1
0	1	0

So based on this we will decide the way we proceed.

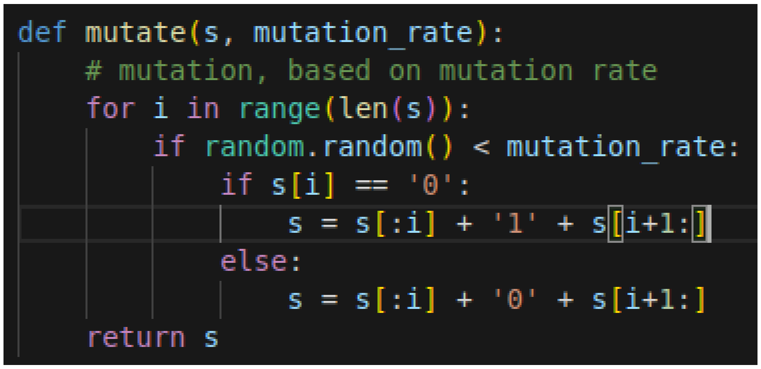


```python
def crossover(s1, s2):
    # two point crossover
    a = random.randint(0, len(s1)-2)
    b = random.randint(a, len(s1)-1)
    temp = s1[a:b]
    s1 = s1[:a] + s2[a:b] + s1[b:]
    s2 = s2[:a] + temp + s2[b:]
    return s1, s2
```

Crossover function->

Here we do a 2 point crossover, that is 2 cut points and we swap from them.

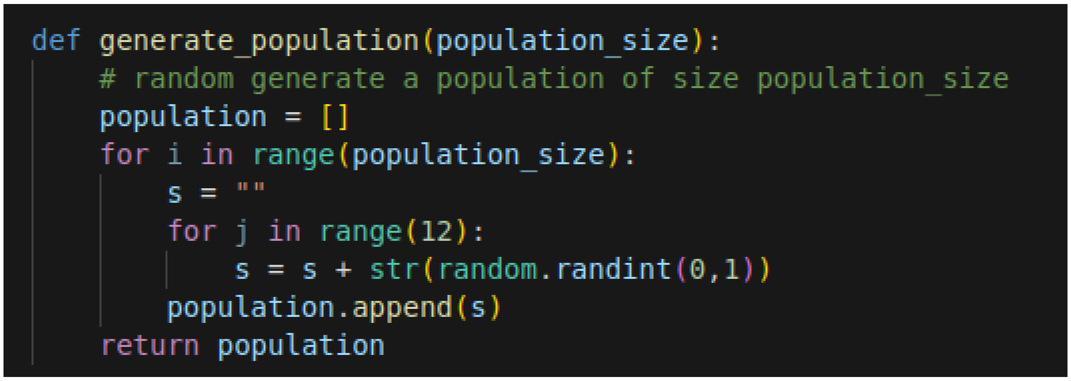
Basically just swapping middle part of two parents, based on the random crossover ponts.



```python
def mutate(s, mutation_rate):
    # mutation, based on mutation rate
    for i in range(len(s)):
        if random.random() < mutation_rate:
            if s[i] == '0':
                s = s[:i] + '1' + s[i+1:]
            else:
                s = s[:i] + '0' + s[i+1:]
    return s
```

Mutation function->

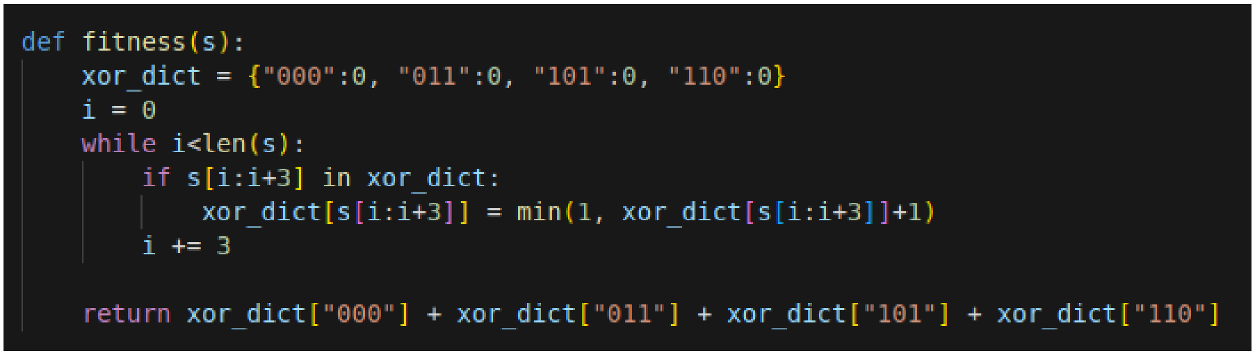
Here, just taking a string, and finding a random number and if its less than mutation rate then we do mutation by flipping the bit.



```python
def generate_population(population_size):
    # random generate a population of size population_size
    population = []
    for i in range(population_size):
        s = ""
        for j in range(12):
            s = s + str(random.randint(0,1))
        population.append(s)
    return population
```

Generate population->

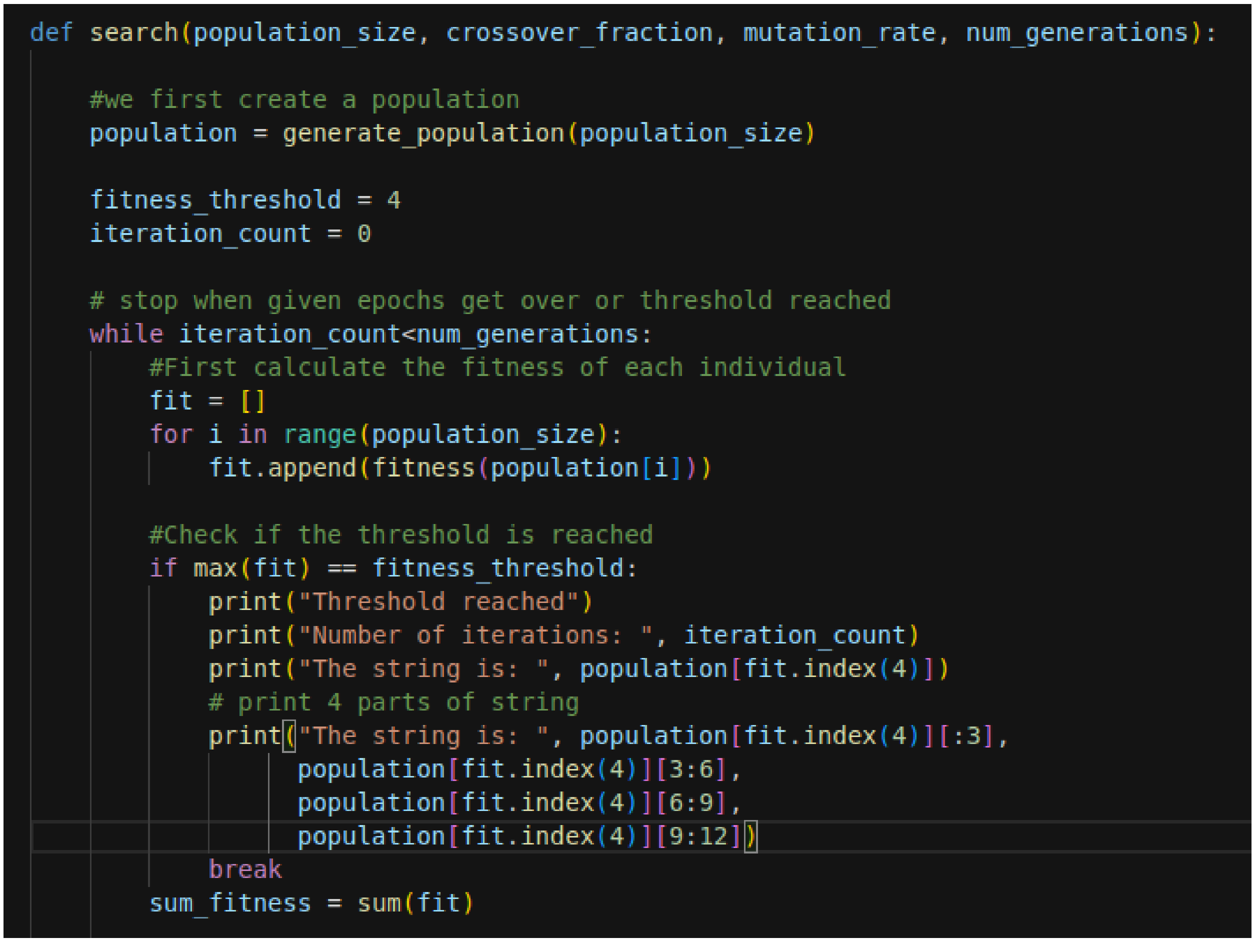
Just taking a population size, and generating a 12 size string in count of population size.



```python
def fitness(s):
    xor_dict = {"000":0, "011":0, "101":0, "110":0}
    i = 0
    while i<len(s):
        if s[i:i+3] in xor_dict:
            xor_dict[s[i:i+3]] = min(1, xor_dict[s[i:i+3]]+1)
        i += 3

    return xor_dict["000"] + xor_dict["011"] + xor_dict["101"] + xor_dict["110"]
```

Fitness function->

Here we just check the fitness of each string, based on the type of 3 length strings it has, if it has all 4 unique types that solve a xor problem then we return a fitness of 4

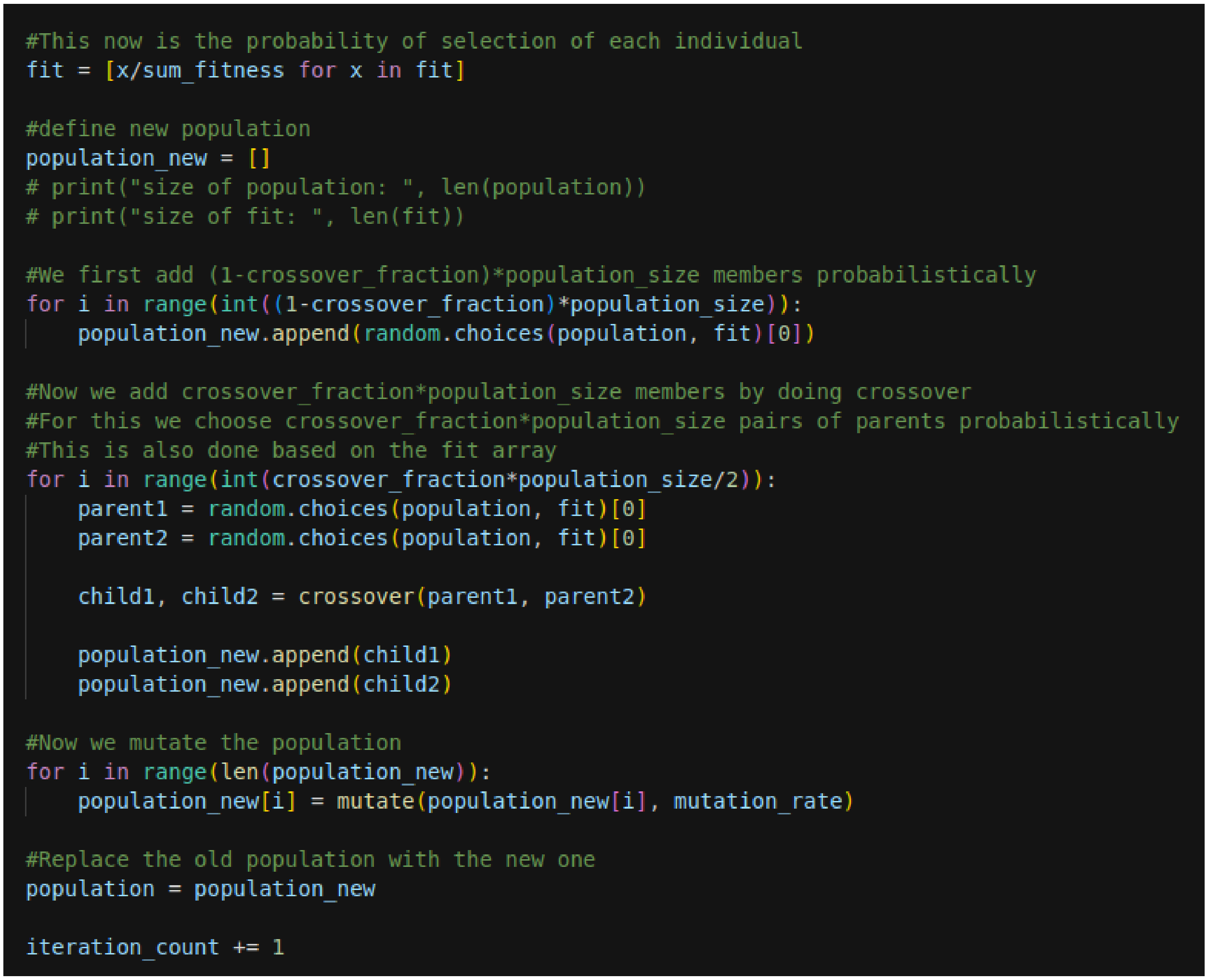


```python
def search(population_size, crossover_fraction, mutation_rate, num_generations):

    #we first create a population
    population = generate_population(population_size)

    fitness_threshold = 4
    iteration_count = 0

    # stop when given epochs get over or threshold reached
    while iteration_count<num_generations:
        #First calculate the fitness of each individual
        fit = []
        for i in range(population_size):
            fit.append(fitness(population[i]))

        #Check if the threshold is reached
        if max(fit) == fitness_threshold:
            print("Threshold reached")
            print("Number of iterations: ", iteration_count)
            print("The string is: ", population[fit.index(4)])
            # print 4 parts of string
            print("The string is: ", population[fit.index(4)][:3],
                  population[fit.index(4)][3:6],
                  population[fit.index(4)][6:9],
                  population[fit.index(4)][9:12])
            break
        sum_fitness = sum(fit)
```

Searching function->

We first generate a population, and iterate till we either reach number of epochs or we get a threshold of 4 (fitness function)

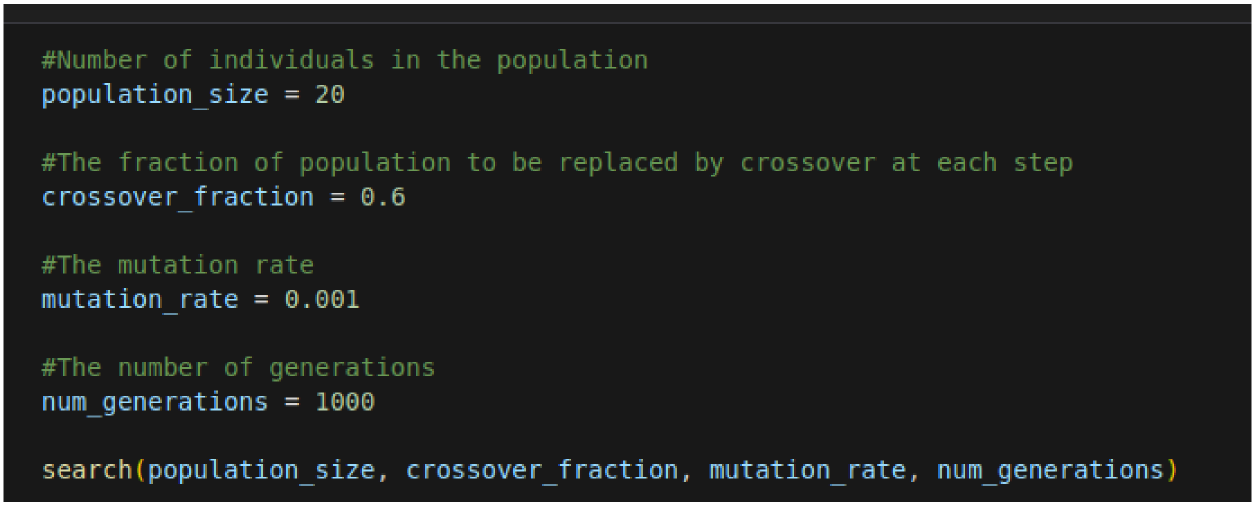
If we get a fit individual, then we stop and print that individual.



```python
#This now is the probability of selection of each individual
fit = [x/sum_fitness for x in fit]

#define new population
population_new = []
# print("size of population: ", len(population))
# print("size of fit: ", len(fit))

#We first add (1-crossover_fraction)*population_size members probabilistically
for i in range(int((1-crossover_fraction)*population_size)):
    population_new.append(random.choices(population, fit)[0])

#Now we add crossover_fraction*population_size members by doing crossover
#For this we choose crossover_fraction*population_size pairs of parents probabilistically
#This is also done based on the fit array
for i in range(int(crossover_fraction*population_size/2)):
    parent1 = random.choices(population, fit)[0]
    parent2 = random.choices(population, fit)[0]

    child1, child2 = crossover(parent1, parent2)

    population_new.append(child1)
    population_new.append(child2)

#Now we mutate the population
for i in range(len(population_new)):
    population_new[i] = mutate(population_new[i], mutation_rate)

#Replace the old population with the new one
population = population_new

iteration_count += 1
```

-> continuation of the above function, here we generate a new population from the previous population.

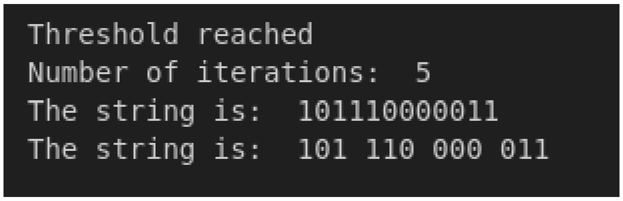
First we select randomly parents and perform crossover then perform mutation and keep the most fit for the next generation.



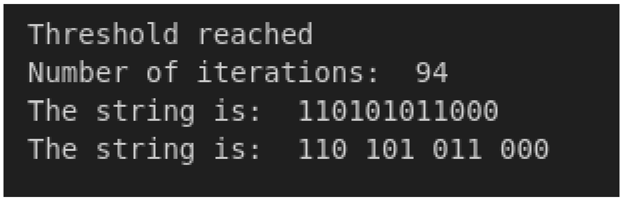
```python
#Number of individuals in the population
population_size = 20

#The fraction of population to be replaced by crossover at each step
crossover_fraction = 0.6

#The mutation rate
mutation_rate = 0.001

#The number of generations
num_generations = 1000

search(population_size, crossover_fraction, mutation_rate, num_generations)
```

Now we call this function first by initializing the factors, like rates and the population size etc.

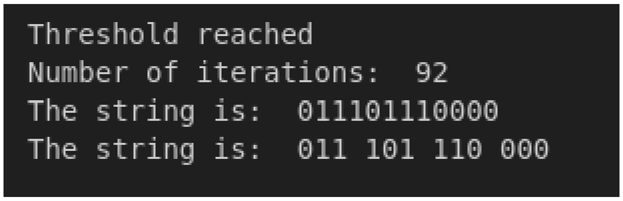
OUTPUT->



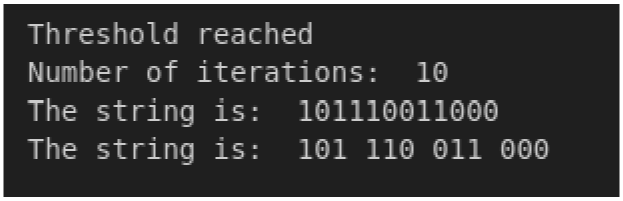
```
Threshold reached
Number of iterations:  5
The string is:  101110000011
The string is:  101 110 000 011
```



```
Threshold reached
Number of iterations:  94
The string is:  110101011000
The string is:  110 101 011 000
```

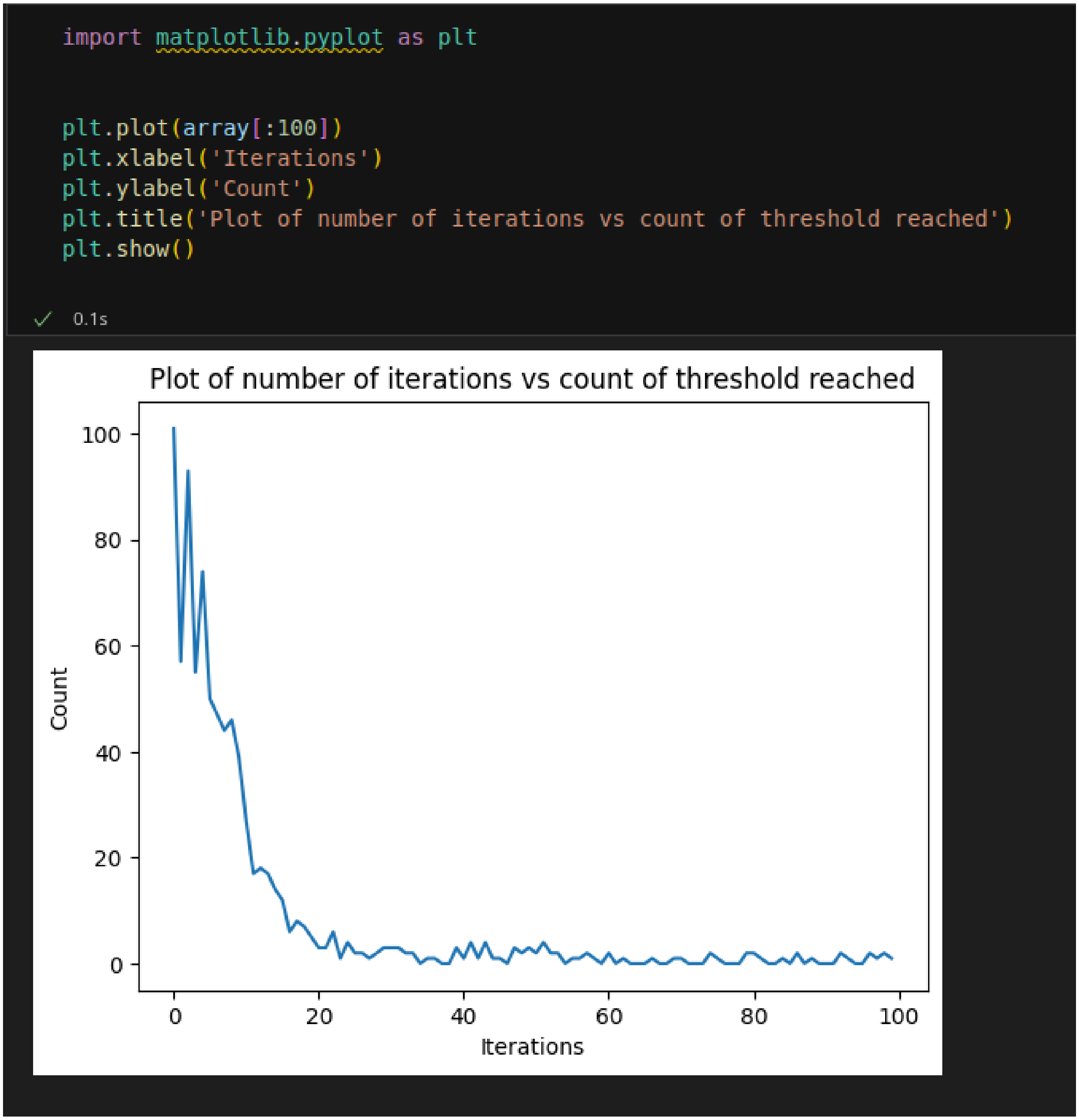


```
Threshold reached
Number of iterations:  92
The string is:  011101110000
The string is:  011 101 110 000
```



```
Threshold reached
Number of iterations:  10
The string is:  101110011000
The string is:  101 110 011 000
```

-> every time i run, i get a different output, but it solves the solution perfectly



```
import matplotlib.pyplot as plt


plt.plot(array[:100])
plt.xlabel('Iterations')
plt.ylabel('Count')
plt.title('Plot of number of iterations vs count of threshold reached')
plt.show()
```

✓ 0.1s

I also ran this a 1000 times to see count of iterations where we reach the correct solution, as we can see form the graph, it reached solution maximum time in the range 0-25 ie it takes approximately 25 iterations to reach the correct solution on average.