```python
import numpy as np
from evolution import selection, fitness
from functions import *

class EvoPhase:
    #Initiate parameters
    def __init__(self, population_tot, crossover_rate, mutation_rate, generations, N, d, target):
        self.population_tot = population_tot
        self.crossover_rate = crossover_rate
        self.mutation_rate = mutation_rate
        self.generations = generations
        self.N = N
        self.target = target

        # Spacing between antennas
        self.pop_dis = np.ndarray(shape=(self.population_tot, self.N-1))

        # Phase Shift of the signal of each antenna element
        self.pop_pha = np.random.uniform(size = (self.population_tot, self.N), low = 0, high = 2*np.pi) # randians

        # Amplitude of the signal of each antenna element.
        self.pop_amp = np.ndarray(shape=(self.population_tot, self.N))
        # Uinform amplitude
        amp = np.array([1, 1, 1, 1, 1, 1])

        # Population with same spacing and amplitude -> Only optimizing phase
        for i in range(self.population_tot):
            self.pop_dis[i, :] = d
            self.pop_amp[i, :] = amp

        # Fitness scores
        self.pop_fit = np.zeros(self.population_tot, dtype = np.float)

        self.evolve()

    def results(self):
        # Return the best solution
        q = np.argmin(self.pop_fit)
        return self.pop_pha[q]

    def evolve(self):
        # probabilities for crossover. To avoid calculationg random numbers every itteration.
        crossovers = np.random.uniform(size=(self.generations), low = 0, high=1)
        mutations = np.random.uniform(size=(self.generations), low = 0, high=1)

        for i in range(self.generations):
            tot = 0
            # Calculate the fitness score of each solution
            for dna in range(self.population_tot):
                score = fitness(self.pop_dis[dna], self.pop_pha[dna], self.pop_amp[dna], self.target)
                self.pop_fit[dna] = score;
                tot += score
            # Index of fittest solution
            index = np.argmin(self.pop_fit)
            # Copy fitteset solution over to next genertation (first and last index of array)
            self.pop_pha[0] = self.pop_pha[index]
            self.pop_pha[-1] = self.pop_pha[index]

            # Stop simulating when the generation matches the maximun no of generations.
            if i == self.generations-1:
                print('DONE')
                break

            # Itterations per generations
            for _ in range(1):
                if crossovers[i] < self.crossover_rate:
                    # crossover occurs (Two parent solutions)
                    s_1 = selection(self.pop_fit)
                    s_2 = selection(self.pop_fit)
                    # Ofspring
                    self.pop_pha[s_1] = np.divide(self.pop_pha[s_1] + self.pop_pha[s_2],2)

                if mutations[i] < self.mutation_rate:
                    # mutation occurs
                    s_1 = selection(self.pop_fit)
                    self.pop_pha[s_1][np.random.randint(low=0, high=self.N)] =  np.random.uniform(low = 0, high=2*np.pi)
```