I did my best to complete the code. However, due to my illness I am not able to perform the tasks. I could not update the chromosomes for some reason that I could not find. I just want to obtain as much as possible points from the code that I wrote. Therefore, I uploaded only the code. I have no intention to disrespect. Thank you for your consideration.

# -\*- coding: utf-8 -\*-  
*"""EE449\_HW2.ipynb  
  
Automatically generated by Colaboratory.  
  
Original file is located at  
 https://colab.research.google.com/drive/1cWjpemWtyFsWq5Od7wm5tT-8JO2etROM  
"""*from google.colab import drive  
drive.mount('/content/gdrive')  
  
#import necessary libraries and methods  
import numpy as np  
import cv2  
from random import randint,uniform,random  
from copy import deepcopy  
from cv2 import imwrite, addWeighted, circle, imread  
from google.colab.patches import cv2\_imshow  
import random  
  
#Read the image and get the size of the image.  
img = cv2.imread('/content/gdrive/MyDrive/EE449\_HW2/painting.png')  
img = img[:,:,0]  
print(img.shape)  
w = img.shape[0]  
h = img.shape[1]  
#image is 180x180  
  
class Gene:  
 def \_\_init\_\_(self, R=0,G=0,B=0,A=0,x=0,y=0,r=0):  
 self.R = R  
 self.G = G  
 self.B = B  
 self.A = A  
 self.x = x  
 self.y = y  
 self.r = r  
 def init\_genes(self):  
 #Initialize the genes with random parameters  
 R = randint(0,255)  
 G = randint(0,255)  
 B = randint(0,255)  
 A = uniform(0,1)  
 x = randint(0,180)  
 y = randint(0,180)  
 r = randint(1,30)  
 #Create new circle parameters until it is valid  
 while not self.check\_intersection(x,y,r):  
 x = randint(0,180)  
 y = randint(0,180)  
 r = randint(1,30)  
 return [R,G,B,A,x,y,r]  
 #Method to check if there is an intersection between image and circle   
 def check\_intersection(self,x,y,r):  
 x\_distance = abs(w/2 - x)  
 y\_distance = abs(h/2 - y)  
 #if both x and y distances are bigger than r, then   
 if(x\_distance > r and y\_distance > r):  
 return False  
 else:  
 return True  
 def mutation\_gene(self,mutation\_type):  
 self.mutation\_type = mutation\_type  
 if mutation\_type == 'unguided':  
 #Initialize the genes with random parameters  
 R = randint(0,255)  
 G = randint(0,255)  
 B = randint(0,255)  
 A = uniform(0,1)  
 x = randint(0,180)  
 y = randint(0,180)  
 r = randint(1,30)  
 #Create new circle parameters until it is valid  
 while not self.check\_intersection(x,y,r):  
 x = randint(0,180)  
 y = randint(0,180)  
 r = randint(1,30)  
 return [R,G,B,A,x,y,r]  
 else:  
 #Guided mutation. Parameters are reviewed according to the  
 #explanation in the homework manual.  
 #int and max(0,int()) functions are added.  
 #Otherwise I got "non-integer stop value" error for randint() function.  
 x = randint(max(0,int(self.x - w/4)) , int(self.x + w/4))  
 y = randint(max(0,int(self.y - h/4)) , int(self.y + h/4))  
 r = randint(max(0,int(self.r - 10)) , int(self.r + 10))  
 R = randint(max(0,int(self.R-64)) , int(min(255,self.R+64)))  
 G = randint(max(0,int(self.G-64)) , int(min(255,self.G+64)))  
 B = randint(max(0,int(self.B-64)) , int(min(255,self.B+64)))  
 A = randint(max(0,int(self.A-0.25)) , int(min(1,self.A+0.25)))   
 while not self.check\_intersection(x,y,r):  
 x = randint(max(0,int(self.x - w/4)) , int(self.x + w/4))  
 y = randint(max(0,int(self.y - h/4)) , int(self.y + h/4))  
 r = randint(max(0,int(self.r - 10)) , int(self.r + 10))  
 return [R,G,B,A,x,y,r]  
  
class Individual:   
 def \_\_init\_\_(self,ind\_id,chrom=[],num\_genes=50):  
 self.num\_genes = num\_genes  
 self.fitness = 0  
 self.ind\_id = ind\_id  
 self.chrom = chrom  
  
 def assign\_gene(self,num\_genes=50):  
 self.num\_genes = num\_genes  
 self.chrom = []  
 for i in range (0,num\_genes):  
 gene\_1 = Gene()   
 self.chrom.append(gene\_1.init\_genes())   
 def draw\_circle(self):   
 blank = np.zeros([180, 180], dtype= np.uint8)#Create an image (180,180)  
 blank.fill(255)#Make the image full white   
 overlay = blank.copy()#Copy the image to draw a circle   
 #Draw circle for each gene and combine the pictures.  
 for genes in self.chrom:  
 alpha = genes[3]  
 beta = 1-alpha  
 colour =(genes[2],genes[1],genes[0])  
 #colour = (255,0,0)  
 radius = genes[6]  
 center = (genes[4],genes[5])  
 #https://www.geeksforgeeks.org/numpy-ones-python/ is used as reference.   
 cv2.circle(overlay,center,radius, colour,-1)#Draw circle  
 #cv2.circle(overlay,(genes[4],genes[5]),genes[6], (150,120,100),-1)#Draw circle  
 #https://docs.opencv.org/3.4/d5/dc4/tutorial\_adding\_images.html is used as reference  
 image = addWeighted(overlay,alpha,blank, beta,0.0)#Sum the blank image and circle  
 return image  
  
 def mutation\_ind(self,mutation\_prob = 0.2,mutation\_type = 'guided'):  
 self.mutation\_prob = mutation\_prob  
 self.mutation\_type = mutation\_type  
 #I multiplied the probabilities with 100. Otherwise I get   
 #ValueError: non-integer stop for randrange() error  
 #Check mutation probability  
 prob\_checker = 1.1  
 while(prob\_checker < mutation\_prob):  
 #Update the value of probability checker randomly.  
 prob\_checker = random.uniform(0, 1)  
 #Choose a random gene to mutate in the chromosome  
 pick\_gene= randint(1,self.num\_genes)  
 #Guided mutation  
 if(mutation\_type =="guided"):  
 #Create a temp gene object to assign after mutation  
 temp\_gene = Gene()  
 temp\_gene = temp\_gene.mutation\_gene("guided")  
 self.chrom[pick\_gene] = temp\_gene  
 #Unguided mutation  
 else:  
 #Create a temp gene object to assign after mutation  
 temp\_gene = Gene()  
 temp\_gene = temp\_gene.mutation\_gene("unguided")  
 self.chrom[pick\_gene] = temp\_gene  
  
class Population:  
 def \_\_init\_\_(self,population = []):  
 self.population = []  
 def add\_to\_population(self, Individual):  
 #Add individuals to the population  
 self.population.append(Individual)  
 def evaluate(self):  
 #self.population = sorted(self.population, key=lambda x: x.fitness, reverse = True)  
 for individual in self.population:   
 #Create the drawing with respect to  
 #Genes of the individual  
 drawing = individual.draw\_circle()  
 #Evaluation according to the formula given in homework manual  
 fitness = np.sum(-1\*np.power(img-drawing,2))  
 individual.fitness = fitness  
 #print(fitness)  
 self.population = sorted(self.population, key=lambda x: x.fitness, reverse = True)  
 def sort\_by\_radius(self,pop):  
 self.pop = pop  
 #Initialize population and add the individuals to the population to return  
 pop\_sorted = Population()  
 pop\_sorted.population.clear()  
 #Sort individuals with respect to their fitness  
 pop\_sorted.population = sorted(pop.population, key=lambda x: x.chrom[6], reverse = True)  
 #Place the sorted population to return population  
 #for i in range(len(sorted\_by\_fitness)):  
 # pop\_sorted.population.append(deepcopy(sorted\_by\_fitness[i]))  
 return pop\_sorted  
  
 def sort\_by\_fitness(self,pop):  
 self.pop = pop  
 #Initialize population and add the individuals to the population to return  
 pop\_sorted = Population()  
 pop\_sorted.population.clear()  
 #Sort individuals with respect to their fitness  
 pop\_sorted.population = sorted(pop.population, key=lambda x: x.fitness, reverse = True)  
 #Place the sorted population to return population  
 #for i in range(len(sorted\_by\_fitness)):  
 # pop\_sorted.population.append(deepcopy(sorted\_by\_fitness[i]))  
 return pop\_sorted  
  
 def select(self,pop,frac\_elites = 0.2 ,num\_inds=20,tm\_size=5):  
 self.tm\_size = tm\_size  
 self.pop = pop  
 self.frac\_elites = frac\_elites  
 self.num\_inds = num\_inds  
 #Calculate the number of elites  
 num\_elites = int(frac\_elites\*num\_inds)  
 #Create a parent population  
 next\_gen = Population()  
 next\_gen.population.clear()  
 #In below for loop, elites are appended to the parents  
 #when the num\_elites are exceeded, remaining parts of the  
 #parent population is filled with tournament selection  
 for i in range(0,num\_inds):  
 if(i <= num\_elites):   
 temp = pop.population[i]  
 next\_gen.population.append(deepcopy(temp))  
 else:  
 winner, winner\_index = pop.tournament(pop,tm\_size)  
 temp = pop.population[winner\_index]  
 next\_gen.population.append(deepcopy(temp))  
   
 return next\_gen  
  
 def tournament(self,pop,tm\_size):  
 self.pop = pop  
 self.tm\_size = tm\_size  
 #Create a population for the attendants of the tournament  
 attend = Population()  
 for i in range(1,tm\_size+1):  
 #https://www.edureka.co/blog/arrays-in-python/#:~:text=Length%20of%20an%20array%20is,elements%20present%20in%20that%20array.&text=This%20returns%20a%20value%20of,the%20number%20of%20array%20elements.  
 #Above website is used as reference to find the number of individuals in population  
 t = randint(5,len(pop.population)-1)  
 #Create a population for tournament  
 attend.population.append(deepcopy(pop.population[t]))  
 #Sort attendants with respect to their fitnesses.   
 attend = attend.sort\_by\_fitness(attend)  
 #After sorting, winner will be in the index [0] of the population   
 winner = attend.population[0]   
 place\_of\_winner = 0  
 attend.population.clear()  
 for individual in pop.population:  
 if individual.ind\_id == winner.ind\_id:  
 #Find the original location of the winner  
 place\_of\_winner = place\_of\_winner  
 return winner,place\_of\_winner  
 else:  
 place\_of\_winner = place\_of\_winner + 1  
 def Crossover(self,parents,num\_genes):  
 self.parents = parents  
 children = Population()  
 children.population.clear()  
 self.num\_genes = num\_genes  
 par\_length = int(len(parents.population)/2)  
 for i in range(0,par\_length+1):  
 rand\_chooser = randint(0,len(self.parents.population)-1)  
 parent\_1 = self.parents.population[rand\_chooser]  
 rand\_chooser = randint(0,len(self.parents.population)-1)  
 parent\_2 = self.parents.population[rand\_chooser]  
 child\_1 = deepcopy(parent\_1)  
 child\_2 = deepcopy(parent\_2)  
   
 for i in range(0,num\_genes):  
 crossover\_checker = uniform(0,1)  
 if crossover\_checker < 0.5:  
 child\_1.chrom[i] = deepcopy(parent\_1.chrom[i])  
 child\_2.chrom[i] = deepcopy(parent\_2.chrom[i])  
 else:  
 child\_1.chrom[i] = deepcopy(parent\_2.chrom[i])  
 child\_2.chrom[i] = deepcopy(parent\_1.chrom[i])  
 children.population.append(deepcopy(child\_1))  
 children.population.append(deepcopy(child\_2))  
 return children  
  
#Create population with respect to the size.  
pop = Population()  
for i in range(1,51):  
 ind = Individual(i)#Add num\_genes parameter here  
 ind.assign\_gene()#Add num\_genes parameter here  
 pop.add\_to\_population(ind)  
#Create population with respect to the size.  
  
num\_of\_generations = 10000  
num\_inds = 50  
frac\_parents = 0.6 #adjust frac\_parents here.  
for i in range(0,10001):  
 print(i)  
 #Sort population with respect to their radius values.  
 sorted\_radius = pop.sort\_by\_radius(pop)  
  
 #Evaluate the sorted population.  
 sorted\_radius.evaluate()   
  
 #After drawing, sort the population with respect to their fitness values.  
 sorted\_fitness = sorted\_radius.sort\_by\_fitness(sorted\_radius)  
  
 #Select the parents of the next generation  
 parents = pop.select(pop, 0.2, num\_inds, 5) #adjust tm\_size and frac\_elites here.   
  
 #Create next generation population with respect to the fitness values  
 next\_gen = Population()  
 sum\_parents = int(frac\_parents \* num\_inds)  
 sorted\_parents = parents.sort\_by\_fitness(parents)  
 for j in range(0,sum\_parents):  
 next\_gen.population.append(deepcopy(sorted\_parents.population[j]))  
  
 #Create children from the parents selected  
 Children = parents.Crossover(parents,num\_inds)  
  
 #Combine the children and the first population created  
 combined = deepcopy(parents)  
 for i in range (0,len(Children.population)):  
 combined.population.append(Children.population[i])   
  
 #Apply mutation to the combined population  
 for individuals in combined.population:  
 individuals.mutation\_ind(0.2, 'guided') #adjust mutation\_prob and mutation\_type here.  
  
 #Sort the combined population with respect to the fitness after mutation  
 sorted\_fitness\_combined = combined.sort\_by\_fitness(combined)  
 sorted\_fitness\_combined.evaluate()   
  
 for i in range(0, num\_inds-sum\_parents):  
 next\_gen.population.append(deepcopy(sorted\_fitness\_combined.population[i]))  
   
 #Clear the intermediate populations  
 sorted\_radius.population.clear()  
 sorted\_fitness.population.clear()  
 parents.population.clear()  
 Children.population.clear()  
 combined.population.clear()  
 next\_gen\_sorted = next\_gen.sort\_by\_fitness(next\_gen)  
 pop = deepcopy(next\_gen\_sorted)  
 next\_gen.population.clear()  
 next\_gen\_sorted.population.clear()  
 for ind in pop.population:  
 print(ind.fitness)