```python
'''
File name      : hw7_prob7_TSP_ga.py
Author         : Jared Hansen
Date created   : 11/23/2019
Python version : 3.7.3

DESCRIPTION: The purpose of this script is to solve the TSP (Traveling
             Salesman Problem) using a genetic algorithm.

'''



#==============================================================================
#==============================================================================
#==== IMPORT STATEMENTS
#==============================================================================
#==============================================================================
import copy
import datetime as dt
import math
import matplotlib.pyplot as plt
import numpy as np
import random
import pylab as pl
from matplotlib import collections as mc
from pulp import *
from datetime import datetime
from deap import base, creator, tools
import string
from deap import base, creator, tools


#==============================================================================
#==============================================================================
#==== Runner CLASS : uses the DEAP toolbox to set up the GA, track performance
#====                metrics, and output the final population (population is
#====                composed of "individuals", e.g. paths around the cities).
#==============================================================================
#==============================================================================
class Runner:

    def __init__(self, toolbox):
        # Use the default toolbox (from DEAP).
        self.toolbox = toolbox
        # Set defaults for the parameters (modified in the method below).
        self.set_params(10, 5, 2)

    def set_params(self, pop_size, iters, num_matings):
        # Specifying population size, number of iterations, and number of
        # matings for our genetic algorithm.
        self.iters = iters
        self.pop_size = pop_size
        self.num_matings = num_matings

    def set_fitness(self, population):
        # Create a list of fitnesses: evaluate each individual in the
        # population based on the calc_path_len function below (a fitter
        # individual has a shorter path length).
        fitnesses = [
            (individual, self.toolbox.calc_path_len(individual))
            for individual in population
        ]
        for individual, fitness in fitnesses:
            individual.fitness.values = (fitness,)

    def get_offspring(self, population):
        # After an iteration of the GA has completed, use the old population
```

```python
        # to create the new population (offspring).
        n = len(population)
        for _ in range(self.num_matings):
            i1, i2 = np.random.choice(range(n), size=2, replace=False)
            offspring1, offspring2 = \
                self.toolbox.mate(population[i1], population[i2])
            yield self.toolbox.mutate(offspring1)[0]
            yield self.toolbox.mutate(offspring2)[0]

    @staticmethod
    def return_stats(population, iteration=1):
        # Returns a dictionary that specifies performance metrics (mean,
        # std dev, max, min) for the passed-in population of individual paths.
        fitnesses = [ individual.fitness.values[0] for individual in population ]
        return {
            'i': iteration,
            'mu': np.mean(fitnesses),
            'std': np.std(fitnesses),
            'max': np.max(fitnesses),
            'min': np.min(fitnesses)
        }

    def Run(self):
        # Runs the GA. Sets the population, calculates performance metrics
        # ("stats" list below) for each iteration, creates offspring, and so on
        # until we've reached the specified number of iterations (num_iters
        # below).
        population = self.toolbox.population(n=self.pop_size)
        self.set_fitness(population)
        stats = []
        # Iteratively creates parent population, evaluates, uses to create
        # offspring population which becomes the next parent population, and so on.
        for iteration in list(range(1, self.iters + 1)):
            current_population = list(map(self.toolbox.clone, population))
            offspring = list(self.get_offspring(current_population))
            for child in offspring:
                current_population.append(child)
            self.set_fitness(current_population)
            population[:] = self.toolbox.select(current_population, len(population))
            stats.append(
                Runner.return_stats(population, iteration))
        return stats, population


#==========================================================================
#==========================================================================
#==== SPECIFYING AND RUNNING THE GA, PLOTTING FINAL RESULTS
#==========================================================================
#==========================================================================
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", list, fitness=creator.FitnessMin)
random.seed(11);
np.random.seed(121);
INDIVIDUAL_SIZE = NUMBER_OF_CITIES = n = 25
pop_size = 200
num_iters = 1000
num_matings = 50
'''
pop_size = 400
num_iters = 2000
num_matings = 100
'''

# Create labeled cities
cities = []
for i in range(n):
    # Append chars '0', '1', '2', ..., 'n' to the cities list
```

```python
    cities.append(str(i))

# Create coordinates for each city
random.seed(1)
points = [(random.randint(0,100),random.randint(0,100)) for i in range(n)]
cities_x = np.array(points)[:,0]
cities_y = np.array(points)[:,1]

# Plot the points that we've generated, with cities labeled by number
fig, tsp_img = plt.subplots()
tsp_img.scatter(cities_x, cities_y)
for i, cityNum in enumerate(cities):
    tsp_img.annotate(cityNum, xy=(cities_x[i], cities_y[i]), xytext=(1,1),
                    textcoords='offset points',
                    fontsize=9)
# Function for calculating the distance between two cities.
def calc_dist(city, to_city):
    dist = math.sqrt((points[city][0] - points[to_city][0])**2 +
                    (points[city][1] - points[to_city][1])**2)
    return(dist)

# Calculate the distances between each city.
distances = np.zeros((n, n))
for city in range(n):
    for to_city in [i for i in range(n) if not i == city]:
        distances[to_city][city] = distances[city][to_city] = calc_dist(city, to_city)

# Set additional GA specifications: toolbox, permutation regimen, and population
# setup.
toolbox = base.Toolbox()
toolbox.register("indices", random.sample, range(INDIVIDUAL_SIZE), INDIVIDUAL_SIZE)
toolbox.register("individual", tools.initIterate, creator.Individual, toolbox.indices)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)

# Define the calc_path_len function to calculate the length of the individual path
# e.g. how long to travel along the specified cities path.
def calc_path_len(individual):
    summation = 0
    start = individual[0]
    for i in range(1, len(individual)):
        end = individual[i]
        summation += distances[start][end]
        start = end
    return summation
# Specifying some additional genetic algorithm settings.
toolbox.register("calc_path_len", calc_path_len)
toolbox.register("mate", tools.cxOrdered)
toolbox.register("mutate", tools.mutShuffleIndexes, indpb=0.01)
toolbox.register("select", tools.selTournament, tournsize=10)

# Actually run the genetic algorithm to get an answer.
a = Runner(toolbox)
a.set_params(pop_size, num_iters, num_matings)
stats, population = a.Run()

# Plot the "learning" results of the algorithm.
plt.figure(figsize=(15,5))
plt.subplot(1,2,1)
_ = plt.scatter([ s['min'] for s in stats ], [ s['max'] for s in stats ], marker='.', s=[ (s['std'] + 1) / 20 for s in stats ])
_ = plt.title('min by max')
_ = plt.xlabel('min')
_ = plt.ylabel('max')
_ = plt.plot(stats[0]['min'], stats[0]['max'], marker='.', color='yellow')
_ = plt.plot(stats[-1]['min'], stats[-1]['max'], marker='.', color='red')
plt.subplot(1,2,2)
_ = plt.scatter([ s['i'] for s in stats ], [ s['mu'] for s in stats ], marker='.', s=[ (s['std'] + 1) / 20 for s in stats ])
_ = plt.title('average by iteration')
```

```python
_ = plt.xlabel('iteration')
_ = plt.ylabel('average')
_ = plt.plot(stats[0]['i'], stats[0]['mu'], marker='.', color='yellow')
_ = plt.plot(stats[-1]['i'], stats[-1]['mu'], marker='.', color='red')
plt.tight_layout()
plt.show()

# See how long the best route took.
fitnesses = sorted([
    (i, toolbox.calc_path_len(individual))
    for i, individual in enumerate(population)
], key=lambda x: x[1])
best_fit = np.round(fitnesses[:1][0][1], 3)
calc_path_len(population[0])
best_path = list(population[0])

# Plot the best path over top of the points
fig, tsp_img = plt.subplots()
tsp_img.scatter(cities_x, cities_y)
for i, cityNum in enumerate(cities):
    tsp_img.annotate(cityNum, xy=(cities_x[i], cities_y[i]), xytext=(1,1),
                     textcoords='offset points',
                     fontsize=12)
for i in range(len(best_path)):
    if(i < len(best_path)-1):
        plt.plot([cities_x[best_path[i]], cities_x[best_path[i+1]]],
                 [cities_y[best_path[i]], cities_y[best_path[i+1]]])
    else:
        plt.plot([cities_x[best_path[-1]], cities_x[best_path[0]]],
                 [cities_y[best_path[-1]], cities_y[best_path[0]]])

plt.title('Solution for ' + str(NUMBER_OF_CITIES) + ' Randomly Created Cities. Total distance: ' + str(best_fit) + ' units')
#plt.title('Solution for ' + str(n) + ' part-B Cities. Total distance: ' + str(total_dist) + ' units')
plt.xlabel('x coordinate of the city')
plt.ylabel('y coordinate of the city')
plt.show()
```