```python
'''
File name      : hw5_prob2_TSP.py
Author         : Jared Hansen
Date created   : 10/22/2019
Python version : 3.7.3

DESCRIPTION: The purpose of this script is to solve the TSP (Traveling
             Salesman Problem) using integer programming. I have elected to use
             the pulp package in Python to do so (at least for the 6, 20, and
             40 city problems. It couldn't handle the 60 city problem.)

'''






#==============================================================================
#==============================================================================
#==== IMPORT STATEMENTS
#==============================================================================
#==============================================================================
import copy
import datetime as dt
#from gurobipy import *
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






#==============================================================================
#==============================================================================
#==== FUNCTION IMPLEMENTATIONS
#==============================================================================
#==============================================================================

def distance(pt_i, pt_j):
    ''' Function for calculating Euclidean distance between two points.
    '''
    dx2 = (pt_i[0] - pt_j[0])*(pt_i[0] - pt_j[0])
    dy2 = (pt_i[1] - pt_j[1])*(pt_i[1] - pt_j[1])
    return(math.sqrt(dx2 + dy2))

def subtour_remove(tsp_prob):
    ''' This function encodes the logic from Pataki's paper, equations 2.3 for
    removing sub-tours from our route.

    Parameters
    ----------
    tsp : pulp.pulp.LpProblem
        Our definition of the TSP problem as an LP using the PuLP library.

    Returns
    -------
    n/a : simply adds more constraints (eliminates subtours) to the tsp LP.
    '''
    # Use two nested loops to go over all pairs of cities. We will end up
    # ignoring tuples (city_i,city_j) where i == j.
    for i in cities:
```

```
        for j in cities:
            if((i != j) and                 # can't be same city
               ((i != '0') and (j != '0')) and  # can't be the origin city
               ((i,j) in bnry)):            # city combo that has 1 (connected)
                tsp_prob += (order[i] - order[j] <= n*(1-bnry[(i,j)]) - 1)

def city_visited(ind):
    ''' This function takes the index of a city visited, adds it to the
    "visited" list and removes it from the "remaining" list.
    '''
    visited.append(remaining.pop(remaining.index(ind)))





#=============================================================================
#=============================================================================
#==== PROCEDURAL CODE
#=============================================================================
#=============================================================================

# How many cities are we dealing with?
n = 20
# Create labeled cities
cities = []
for i in range(n):
    # Append chars '0', '1', '2', ..., 'n' to the cities list
    cities.append(str(i))
# Set seed for script, determine the coordinates for each city
random.seed(1776)
cities_x = np.random.randint(low=1, high=n, size=n)
cities_y = np.random.randint(low=1, high=n, size=n)


'''
#---------------------------------------------------------------------------
# This section is for doing part B where we are given the coordinates of
# the cities. UNCOMMENT TO RUN FOR PART B.
#---------------------------------------------------------------------------
cities_x = np.array([0,1,2,10,11,12])
cities_y = np.array([0,1,0.1,-0.1,1,0])
n = len(cities_x)
# Create labeled cities
cities = []
for i in range(n):
    # Append chars '0', '1', '2', ..., 'n' to the cities list
    cities.append(str(i))
#---------------------------------------------------------------------------
'''

# Define a dictionary of distance between each pair of points
pt_distances = {}
for i in range(n):
    for j in range(n):
        # Only store distances for cities that aren't the same (e.g. don't
        # store distance between city0 and city0 or city1 and city1, etc.)
        if(i != j):
            # The key for the dict is the tuple of city_i and city_j indices.
            # The values in the dict are the distance btwn city_i and city_j.
            pt_distances[(cities[i],cities[j])] = distance([cities_x[i], cities_y[j]],
                                                            [cities_x[j], cities_y[i]])

# 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):
```

```python
    tsp_img.annotate(cityNum, xy=(cities_x[i], cities_y[i]), xytext=(1,1),
                     textcoords='offset points',
                     fontsize=14)

# Define the problem, similar to using Problem-Based approach in Matlab
tsp_prob = LpProblem("TSP", LpMinimize)
# Define binary variables denoting if city_j is visited after city_i in tour
bnry = LpVariable.dicts('bnry', pt_distances, 0, 1, LpBinary)
# Define the objective function to be minimized.
# Our objective is to minimize the total distance traveled in order to visit
# each city exactly once, and end up back at the city in which we started.
obj = lpSum([bnry[(i,j)]*pt_distances[(i,j)] for (i,j) in pt_distances])
# Add our newly-defined objective function to the problem we've defined above
tsp_prob += obj

# Define our "assignment constraints" as given in the Pataki paper.
# Add them to the tsp_prob as we go.
for c in cities:
    # Every city is arrived at exactly once
    tsp_prob += (lpSum([bnry[(i,c)] for i in cities if (i,c) in bnry]) == 1)
    # Every city is departed from exactly once
    tsp_prob += (lpSum([bnry[(c,i)] for i in cities if (c,i) in bnry]) == 1)

# To get rid of subtours we have to store the order of cities visited.
order = LpVariable.dicts('order', cities, 0, (n-1), LpInteger)
# Call the function for removing subtours
subtour_remove(tsp_prob)

# Start timing the solver
start_time = dt.datetime.now()
# Call the "solve" method on our defined problem to get the solution
tsp_prob.solve()
# Stop timing the solver
timed = str(dt.datetime.now() - start_time)
print("The solver took " + timed + " to run for " + str(n) + " cities.")
print("(time is in hours : minutes : seconds.frac_of_second)")
# It returns a "1" to the console, so we know that it was successful.
# Furthermore, we can call the LpStatus method to see that 1 == "Optimal"
print(LpStatus[tsp_prob.status])

#---------------------------------------------------------------------------
# Determine the order in which we vist the cites
#---------------------------------------------------------------------------
# Make a deep copy of the list "cities" (otherwise it's basically just a
# reference to "cities" so we'd modify the original object).
remaining = copy.deepcopy(cities)
# Create an integer variable to hold the index of the current city
current = '0'
# Initialize a list with 0 since this is the origin city by definition. This
# list will keep track of the order of our visits
visited = []
city_visited(current)
# Loop until we've visited all of the cities (e.g. removed all of them from
# the "remaining" list)
while(len(remaining) > 0):
    for next_city in remaining:
        # If bnry for the tuple (current,next_city) is 1 this means that we
        # visit the next_city from the current one. So we can add current to
        # our visited list, remove it from remaining, and set current to be
        # the next city.
        if(bnry[(current,next_city)].varValue == 1):
            city_visited(next_city)
            current = next_city
            # Once we've found a new visit and modified lists correctly,
            # break out of the for loop.
            break
# Since we go back to the origin city at the end, add this to the visited list
```

```python
# now that we've stored the rest of the tour
visited.append('0')

# Store the distances between each city that we visited
visited_dists = [pt_distances[(visited[i-1], visited[i])] for i in
                 range(1,len(visited))]

print('\nPath travelled')
print('--------------')
# Display the order in which the cities were visited
for i in range(len(visited_dists)):
    print(' ', visited[i], ' to ', visited[i+1])
# How far was the total distance traveled?
total_dist = round(sum(visited_dists),4)
print('Total distance traveled is:', total_dist, 'units')

# Plot the solution 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(visited)-1):
    plt.plot([cities_x[cities.index(visited[i])],cities_x[cities.index(visited[i+1])]],
             [cities_y[cities.index(visited[i])],cities_y[cities.index(visited[i+1])]], 'c')
plt.title('Solution for ' + str(n) + ' Randomly Created Cities. Total distance: ' + str(total_dist) + ' 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()
```