**class** Node: *#a dictionary of the nodes that are paired with the node:{...,neighbor name:edge weight,...)* **def** \_\_init\_\_(self,name):  
 self.name=name  
 self.adjacents={}  
  
 **def** \_\_str\_\_(self): *#returns a string with the node name, his neighbors name and their edges weight* neigbors\_details=**""  
 for** key **in** self.adjacents:  
 neigbors\_details+=**"%s with edge weight of %d\n"**%(key,self.adjacents[key])  
 **return "Node %s has %d neighbors:\n%s"**%(self.name,len(self.adjacents),neigbors\_details)  
  
 **def** \_\_eq\_\_(self, other): *#checks if two nodes have the same name* **return** self.name == other.name  
  
 **def** \_\_ne\_\_(self, other): *#checks if two nodes have different names* **return** self.name != other.name  
  
 **def** neighbors(self): *#returns a list of the names of the adjacent nodes (the node neighbors names)* **return** self.adjacents.keys()  
  
 **def** is\_neighbor(self, name): *#check if a node is a neighbor by his name* **return** name **in** self.neighbors()  
  
 **def** add\_neighbor(self, name, weight=1): *#adds a new neighbor to a node including his weight* **return** self.adjacents.update({name:weight})  
  
 **def** remove\_neighbor(self, name): *#removes one of the node neighbors by giving the neighbors name* **return** self.adjacents.pop(name)  
  
 **def** is\_isolated(self): *#checks if a node has no neighbors at all* **return** len(self.neighbors())==0  
  
*###################################################################################################################################***class** Graph(Node): *#Creates a dict of:{...,node\_name:node\_object,...)* **def** \_\_init\_\_(self,\*nodes):  
 self.nodes={}  
 **for** node **in** nodes:  
 self.nodes[node]=Node(node)  
  
 **def** \_\_str\_\_(self): *#returns a string explaining how many nodes are in the graph and their names* a=**""  
 for** node **in** self.nodes:  
 a+=**"%s, "**%node  
 **return "the graph has %d nodes:"**%(len(self.nodes))+a[:-2]+**"."  
  
 def** \_\_len\_\_(self): *#returns the no. of nodes in the graph* **return** len(self.nodes)  
  
 **def** \_\_contains\_\_(self, name): *#checks if a node is in the graph by his name* **return** name **in** self.nodes  
  
 **def** \_\_getitem\_\_(self, name): *#returns a node object by his name* **return** self.nodes[name]  
  
 **def** \_\_add\_\_(self, other): *# create a new graph from all the nodes of two other graphs* all\_names=list(self.names())+list(other.names())  
 a=Graph(\*all\_names)  
 **for** edge **in** self.edges():  
 a.add\_edge(edge[0],edge[1],self.get\_edge\_weight(edge[0],edge[1]))  
 **for** edge **in** other.edges():  
 a.add\_edge(edge[0],edge[1],other.get\_edge\_weight(edge[0],edge[1]))  
 **return** a  
  
 **def** names(self): *#returns a list of the names of all the nodes in the graph* **return** self.nodes.keys()  
  
 **def** edges(self): *# returns a list of all the edges(tuples) in the graph (without the opposites- if there is (a,b) and (b,a) only one of them will enter* list1 = []  
 list2 = []  
 **for** name **in** self.names():  
 list2.extend(self.\_\_getitem\_\_(name).neighbors())  
 a = 0  
 **while** a < len(self.\_\_getitem\_\_(name).neighbors()):  
 list1.append(name)  
 a += 1  
 list3=list(zip(list1, list2))  
 list4=[]  
 **for** i **in** range(len(list3)):  
 **if** (list3[i][1],list3[i][0]) **not in** list4:  
 list4.append((list3[i][0], list3[i][1]))  
 **return** list4  
  
 **def** is\_edge(self,frm\_name,to\_name):*#checks if there is such edge in the Graph* **return** self.\_\_getitem\_\_(frm\_name).is\_neighbor(to\_name) **or** self.\_\_getitem\_\_(to\_name).is\_neighbor(frm\_name)  
  
 **def** add\_edge(self, frm\_name, to\_name, weight=1): *#adds a new edge to the Graph* self.\_\_getitem\_\_(frm\_name).add\_neighbor(to\_name, weight)  
 self.\_\_getitem\_\_(to\_name).add\_neighbor(frm\_name, weight) *# no direction* **def** remove\_edge(self, frm\_name, to\_name): *#removes a specific edge from the Graph* self.\_\_getitem\_\_(frm\_name).remove\_neighbor(to\_name)  
 self.\_\_getitem\_\_(to\_name).remove\_neighbor(frm\_name) *# no direction* **def** get\_edge\_weight(self, frm\_name, to\_name): *#return the edge weight* **return** self.\_\_getitem\_\_(frm\_name).adjacents[to\_name]  
  
 **def** get\_path\_weight(self, path): *#return a path weight* t\_weight=0  
 **for** i **in** range(len(path)-1):  
 t\_weight+=self.get\_edge\_weight(path[i], path[i+1])  
 **return** t\_weight  
  
 **def** find\_path (self, frm\_name, to\_name): *#returns all possible paths between frm\_name and to\_name* possible\_paths=[[frm\_name,neighbor] **for** neighbor **in** self.\_\_getitem\_\_(frm\_name).neighbors()]  
 good\_paths=[]  
  
 **for** path **in** possible\_paths:  
 **if** path[-1] == to\_name:  
 good\_paths.append(path)  
 **elif** path[-1]!=frm\_name:  
 **for** neighbor **in** self.\_\_getitem\_\_(path[-1]).neighbors():  
 **if** neighbor **not in** path: *# no direction* possible\_paths.append(path+[neighbor])  
 **return** good\_paths  
  
 **def** find\_shortest\_path(self, frm\_name, to\_name): *#returns the possible paths between frm\_name and to\_name with the lowest weight* weights=[self.get\_path\_weight(path) **for** path **in** self.find\_path(frm\_name, to\_name)]  
 paths=self.find\_path (frm\_name, to\_name)  
 **return** paths[weights.index(min(weights))]  
  
*#Question4* **def** suggest\_friend(self, node\_name): *#return the name of the friend with the most common friends who is not friends of the node\_name* best\_suggestion = [**""**, 0]  
 **for** other\_node **in** self.names():  
 **if** node\_name **not in** self.\_\_getitem\_\_(other\_node).neighbors() **and** node\_name!=other\_node:  
 common\_friends = set(self.\_\_getitem\_\_(other\_node).neighbors()).intersection(set(self.\_\_getitem\_\_(node\_name).neighbors()))  
 **if** len(common\_friends)>best\_suggestion[1]:  
 best\_suggestion = [other\_node, len(common\_friends)]  
 **else**:  
 **continue  
 return** best\_suggestion[0]  
  
*###################################################################################################################################  
#Question1***with** open(**"social.txt"**,**"r"**) **as** f:  
 users=set([line.split()[0] **for** line **in** f])  
**with** open(**"social.txt"**, **"r"**) **as** f:  
 users2 =set([line.split()[3] **for** line **in** f])  
 users=users.union(users2)  
 users\_graph=Graph(\*users)  
*#def add\_edge(self, frm\_name, to\_name, weight=1)***with** open(**"social.txt"**, **"r"**) **as** f:  
 num\_friends=[]  
 **for** line **in** f:  
 frm\_name=line.split()[0]  
 to\_name = line.split()[2]  
 line\_num=1  
 **if** line.split()[-1]==**'friends.'**:  
 users\_graph.add\_edge(line.split()[0],line.split()[2])  
 *#print ("%s:%s"% (line.split()[0],users\_graph.\_\_getitem\_\_(line.split()[0]).neighbors()))* num\_friends.append(len(users\_graph.edges()))  
 **else**:  
 **if** users\_graph.is\_edge(line.split()[0],line.split()[2]):  
 users\_graph.remove\_edge(line.split()[0], line.split()[2])  
 num\_friends.append(len(users\_graph.edges()))  
 **if** users\_graph.is\_edge(line.split()[2],line.split()[0]):  
 users\_graph.remove\_edge(line.split()[2], line.split()[0])  
 num\_friends.append(len(users\_graph.edges()))  
 print (**"Question1 Answer is:"**,max(num\_friends))  
  
*#Benjamin and Levi became friends.  
#Ephraim and Benjamin cancelled their friendship.  
  
#Question2***with** open(**"social.txt"**,**"r"**) **as** f:  
 users=set([line.split()[0] **for** line **in** f])  
**with** open(**"social.txt"**, **"r"**) **as** f:  
 users2 =set([line.split()[3] **for** line **in** f])  
 users=users.union(users2)  
 users\_graph=Graph(\*users)  
  
**with** open(**"social.txt"**, **"r"**) **as** f:  
 num\_friends=[]  
 **for** line **in** f:  
 frm\_name=line.split()[0]  
 to\_name = line.split()[2]  
 line\_num=1  
 **if** line.split()[-1]==**'friends.' and 'Reuben' in** line.split():  
 users\_graph.add\_edge(line.split()[0],line.split()[2])  
 *#print "%s:%s"% (line.split()[0],users\_graph.\_\_getitem\_\_(line.split()[0]).neighbors())* num\_friends.append(len(users\_graph.edges()))  
 **elif 'Reuben' in** line.split():  
 **if** users\_graph.is\_edge(line.split()[0],line.split()[2]):  
 users\_graph.remove\_edge(line.split()[0], line.split()[2])  
 num\_friends.append(len(users\_graph.edges()))  
 **if** users\_graph.is\_edge(line.split()[2],line.split()[0]):  
 users\_graph.remove\_edge(line.split()[2], line.split()[0])  
 num\_friends.append(len(users\_graph.edges()))  
 print (**"Question2 Answer is:"**,max(num\_friends))  
  
*# Question3*paths=[]  
**for** frm\_name **in** users:  
 **for** to\_name **in** users:  
 paths.extend(users\_graph.find\_path(frm\_name, to\_name))  
longest\_path=paths[0]  
**for** path **in** paths:  
 **if** len(path)>len(longest\_path):  
 longest\_path=path  
print (**"Question3 Answer is:"**,longest\_path)  
  
*###################################################################################################################################  
#Part2-task1***class** DirectedGraph(Graph): *#Creates a dict of:{...,node\_name:node\_object,...)* **def** \_\_init\_\_(self,\*nodes):  
 self.nodes={}  
 **for** node **in** nodes:  
 self.nodes[node]=Node(node)  
  
 **def** \_\_add\_\_(self, other): *# create a new graph from all the nodes of two other graphs* all\_names = list(self.names()) + list(other.names())  
 a=DirectedGraph(\*all\_names)  
 **for** edge **in** self.edges():  
 a.add\_edge(edge[0],edge[1],self.get\_edge\_weight(edge[0],edge[1]))  
 **for** edge **in** other.edges():  
 a.add\_edge(edge[0],edge[1],other.get\_edge\_weight(edge[0],edge[1]))  
 **return** a  
  
 **def** edges(self): *# returns a list of all the edges(tuples) in the graph (without the opposites- if there is (a,b) and (b,a) only one of them will enter* list1 = []  
 list2 = []  
 **for** name **in** self.names():  
 list2.extend(self.\_\_getitem\_\_(name).neighbors())  
 a = 0  
 **while** a < len(self.\_\_getitem\_\_(name).neighbors()):  
 list1.append(name)  
 a += 1  
 list3=list(zip(list1, list2))  
 list4=[]  
 **for** i **in** range(len(list3)):  
 list4.append((list3[i][0], list3[i][1]))  
 **return** list4  
  
 **def** is\_edge(self,frm\_name,to\_name):  
 **return** self.\_\_getitem\_\_(frm\_name).is\_neighbor(to\_name)  
  
 **def** add\_edge(self, frm\_name, to\_name, weight=1):  
 self.\_\_getitem\_\_(frm\_name).add\_neighbor(to\_name, weight)  
  
 **def** remove\_edge(self, frm\_name, to\_name):  
 self.\_\_getitem\_\_(frm\_name).remove\_neighbor(to\_name)  
  
  
*#g=Graph('a','b','c','d','e')  
#d=DirectedGraph('a','b','c','d','e')  
  
###################################################################################################################################  
  
#Part2-task2: travelEW  
  
#weight= the average time all the travels took from one region to the other***with** open(**'travelsEW.csv'**) **as** ew: *#making a list out of the file* ew\_list=[line.replace(**","**,**" "**).strip(**"\n"**).split() **for** line **in** ew][1:]  
  
*#getting the nodes of travelsEW ready to be entered to a graph:  
 #I had to remove bad lines from the file. the code i used in order to track the bad lines::  
 #for line in ew\_list:  
 # if line[0]=='W' or line[3]=='Sou':  
 # print (ew\_list.index(line))*nodes\_ew\_frm=set([line[0] **for** line **in** ew\_list])  
nodes\_ew\_to=set([line[3] **for** line **in** ew\_list])  
nodes\_ew=nodes\_ew\_to.union(nodes\_ew\_frm)  
  
*#creating travelsEW graph:*travelsEW=DirectedGraph(\*nodes\_ew)  
  
**from** datetime **import** \*  
*#setting the travelsEW datetimes to match a known format:***for** line **in** ew\_list:  
 line[1]=line[1]+**" "**+line[2]  
 line[1].replace(**'h'**,**':'**).strip(**'m'**)  
 line[4]=line[4]+**" "**+line[5]  
 line.pop(2)  
 line.pop(-1)  
 line[1]=line[1].replace(**'h'**, **':'**).strip(**'m'**)  
 line[-1]=line[-1].replace(**'h'**, **':'**).strip(**'m'**)  
*# adding the time (seconds) each travel took to ew\_list:* st\_time=datetime.strptime(line[1], **'%d/%m/%Y %H:%M'**)  
 en\_time=datetime.strptime(line[-1], **'%d/%m/%Y %H:%M'**)  
 travel\_weight=en\_time-st\_time  
 line.append(float(travel\_weight.seconds))  
  
*# calculating the average weight for each track:*routes\_ew=[]  
**for** line **in** ew\_list:  
 **if** [line[0],line[2],0,0] **not in** routes\_ew:  
 routes\_ew.append([line[0], line[2],0,0]) *#[start,end,total\_time,num\_travels]***for** line **in** ew\_list:  
 **for** route **in** routes\_ew:  
 **if** line[0]==route[0] **and** line[2]==route[1]:  
 route[2]+=line[-1]  
 route[3]+=1  
  
**for** route **in** routes\_ew: *#[start,end,average\_seconds\_per\_travel]* route.append((route[2]/route[3]))  
 route.pop(2)  
 route.pop(2)  
  
**for** route **in** routes\_ew:  
 travelsEW.add\_edge(route[0],route[1],route[2])  
  
*###################################################################################################################################  
  
#Part2-task2: travelWE***with** open(**'travelsWE.csv'**) **as** we: *#making a list out of the file* we\_list=[line.replace(**" ; "**,**","**).replace(**","**,**"-"**).strip(**"\n"**).split(**"-"**) **for** line **in** we][1:]  
  
**for** line **in** we\_list: *#changing the we file list so that it the dates will be before the hours (to use with datetime later)  
#I had to remove bad lines from the file.* line.insert(3,line[1])  
 line.pop(1)  
 line.insert(6, line[4])  
 line.pop(4)  
print (**"after"**,we\_list)  
  
*#getting the nodes of travelsWE ready to be entered to a graph:*nodes\_we\_frm=set([line[0] **for** line **in** we\_list])  
nodes\_we\_to=set([line[3] **for** line **in** we\_list])  
nodes\_we=nodes\_we\_to.union(nodes\_we\_frm)  
  
*#creating travelsWE graph:*travelsWE=DirectedGraph(\*nodes\_we)  
print (travelsWE.names(),**"2"**)  
  
*#preparing for adding edges to travelsWE:  
  
#setting the travelsWE datetimes to match a known format:*print (we\_list)  
**for** line **in** we\_list:  
 line[1]=line[1]+**" "**+line[2]  
 line[4]=line[4]+**" "**+line[5]  
 line.pop(2)  
 line.pop(-1)  
*# adding the time (seconds) each travel took to we\_list:  
 # had to manualy fix:  
 # changed Jam to Jan> ValueError: time data 'Jam 25 16 05:55:00PM' does not match format '%b %d %y %I:%M:%S%p  
 # changed 00 to 28> ValueError: time data 'Feb 00 16 02:43:00AM' does not match format '%b %d %y %I:%M:%S%p'* st\_time=datetime.strptime(line[1], **'%b %d %y %I:%M:%S%p'**)  
 en\_time=datetime.strptime(line[-1], **'%b %d %y %I:%M:%S%p'**)  
 travel\_weight=en\_time-st\_time  
 line.append(float(travel\_weight.seconds))  
  
*# calculating the average weight for each track:*routes\_we=[]  
**for** line **in** we\_list:  
 **if** [line[0],line[2],0,0] **not in** routes\_we:  
 routes\_we.append([line[0], line[2],0,0]) *#[start,end,total\_time,num\_travels]***for** line **in** we\_list:  
 **for** route **in** routes\_we:  
 **if** line[0]==route[0] **and** line[2]==route[1]:  
 route[2]+=line[-1]  
 route[3]+=1  
  
**for** route **in** routes\_we: *#[start,end,average\_seconds\_per\_travel]* route.append((route[2]/route[3]))  
 route.pop(2)  
 route.pop(2)  
**for** route **in** routes\_we:  
 travelsWE.add\_edge(route[0],route[1],route[2])  
print (travelsWE.edges(),**"45"**)  
*#adding both graphs together*all\_travels=travelsEW.\_\_add\_\_(travelsWE)  
print (travelsWE.edges(),**"1"**)  
print (travelsEW.edges(),**"2"**)  
print (all\_travels.edges(),**"4"**)