**import** random  
**class** Net:  
 *""" Net as the graph model """* **def** \_\_init\_\_(self):  
 *# network model time* self.time = 0  
 *# duration of the network simulation [min]* self.duration = 0  
 *# network geography* self.nodes = []  
 self.links = []  
 self.lines = []  
 *# transport demand* self.demand = []  
 *# resulting characteristics* self.total\_wait\_time = 0  
 self.sum\_vehicles\_time = 0  
 self.num\_serviced\_passengers = 0  
 **def** contains\_node(self, node\_code):  
 *"""" Determines if the network contains a node with the specified code """* **for** n **in** self.nodes:  
 **if** n.code == node\_code:  
 **return** True  
 **return** False  
 **def** get\_node(self, code):  
 *"""" Returns the first found node with the specified code """* **for** n **in** self.nodes:  
 **if** n.code == code:  
 **return** n  
 **return** None  
 **def** contains\_link(self, out\_node, in\_node):  
 *""" Checks if the net contains a link """* **for** l **in** self.links:  
 **if** l.out\_node **is** out\_node **and** l.in\_node **is** in\_node:  
 **return** True  
 **return** False  
 **def** get\_link(self, out\_node, in\_node):  
 *"""" Returns the first found link with the specified out and in nodes """* **for** l **in** self.links:  
 **if** l.out\_node **is** out\_node **and** l.in\_node **is** in\_node:  
 **return** l  
 **return** None  
 **def** add\_link(self, out\_code, in\_code, weight=0, directed=False):  
 *"""" Adds a link with the specified characteristics """* **if** self.contains\_node(out\_code):  
 *# out-node is already in the net* out\_node = self.get\_node(out\_code)  
 **if** self.contains\_node(in\_code):  
 *# in-node is already in the net* in\_node = self.get\_node(in\_code)  
 **if** self.contains\_link(out\_node, in\_node):  
 *# out-node and in-node are already linked: change the link weight* self.get\_link(out\_node, in\_node).weight = weight  
 **else**:  
 *# there is no such a link in the net: add a new one* new\_link = link.Link(out\_node, in\_node, weight)  
 out\_node.out\_links.append(new\_link)  
 in\_node.in\_links.append(new\_link)  
 self.links.append(new\_link)  
 **else**:  
 *# the net contains the specified out-node, but there is no in-node* *# with the specified code* in\_node = node.Node(in\_code)  
 new\_link = link.Link(out\_node, in\_node, weight)  
 out\_node.out\_links.append(new\_link)  
 in\_node.in\_links.append(new\_link)  
 self.nodes.append(in\_node)  
 self.links.append(new\_link)  
 **else**:  
 *# the net does not contain the specified out-node* out\_node = node.Node(out\_code)  
 **if** self.contains\_node(in\_code):  
 *# in-node is already in the net* in\_node = self.get\_node(in\_code)  
 **else**:  
 *# there are no in-node and out-node with the specified codes* in\_node = node.Node(in\_code)  
 *# create new link* new\_link = link.Link(out\_node, in\_node, weight)  
 out\_node.out\_links.append(new\_link)  
 in\_node.in\_links.append(new\_link)  
 self.nodes.append(in\_node)  
 self.nodes.append(out\_node)  
 self.links.append(new\_link)  
 *# add the reverse link* **if not** directed:  
 self.add\_link(in\_code, out\_code, weight, True)  
 *# sort the nodes (is useful for calculating the short distances matrix)  
 # self.nodes.sort()* **def** generate(self, nodes\_num, links\_num, s\_weight):  
 *"""  
 nodes\_num - number of nodes in the net  
 links\_num - number of links in the net  
 s\_weight - stochastic variable of the links weight  
 """  
 # limit lower bound for the number of nodes* **if** nodes\_num < 2:  
 nodes\_num = 2  
 *# limit lower bound for the number of links* **if** links\_num < 1:  
 links\_num = 1  
 *# limit upper bound for the number of links* **if** links\_num > nodes\_num\*(nodes\_num - 1):  
 links\_num = nodes\_num \* (nodes\_num - 1)  
 *# define a set of the network nodes* **for** i **in** range(1, nodes\_num + 1):  
 self.nodes.append(node.Node(i))  
 *# generate random set of the network links  
 # ! some nodes in the network could not be linked* l\_num = 0 *# counter for the links number* **while** l\_num < links\_num:  
 out\_node = random.choice(self.nodes)  
 in\_node = random.choice(self.nodes)  
 **while** out\_node **is** in\_node:  
 in\_node = random.choice(self.nodes)  
 **if not** self.contains\_link(out\_node, in\_node):  
 self.add\_link(out\_node.code, in\_node.code, s\_weight.get\_value(), True)  
 l\_num += 1  
 **def** gen\_lines(self, lines\_num, s\_stops\_num):  
 *"""  
 Generates specified number of lines which contain the random number of stops  
 lines\_num - number of lines, s\_stop\_num - stochastic variable of the stops number  
 """  
 # line could contain more than 1 stop in the same node* **for** idx\_line **in** range(lines\_num):  
 stops\_num = int(s\_stops\_num.get\_value())  
 **if** stops\_num < 2:  
 stops\_num = 2  
 stops = []  
 stop = random.choice(self.nodes) *# begin stop* **while** len(stop.out\_links) == 0:  
 stop = random.choice(self.nodes)  
 stops.append(stop.code)  
 **for** idx\_stop **in** range(stops\_num - 1):  
 next\_stop = (random.choice(stop.out\_links)).in\_node  
 **while** len(next\_stop.out\_links) == 0 **or** next\_stop.code **in** stops:  
 next\_stop = (random.choice(stop.out\_links)).in\_node  
 stop = next\_stop  
 stops.append(stop.code)  
 self.lines.append(line.Line(self, stops))

**def** gen\_demand(self, duration):  
 *""""  
 Generates demand for trips in the network  
 duration - duration of the simulation period, hrs  
 """* self.demand = []  
 **for** nd **in** self.nodes:  
 time = 0  
 **while** time <= duration:  
 interval = round(nd.s\_interval.get\_value(), 1)  
 time += interval  
 *# generating a new passenger* new\_passenger = passenger.Passenger()  
 new\_passenger.m\_appearance = time  
 new\_passenger.origin\_node = nd  
 *# defining the destination node - random choice rule* *# (can't be the same as origin node)* destination\_node = random.choice(self.nodes)  
 **while** destination\_node == nd:  
 destination\_node = random.choice(self.nodes)  
 new\_passenger.destination\_node = destination\_node  
 *# adding the passenger to the origin node collection* nd.pass\_out.append(new\_passenger)  
 self.demand.append(new\_passenger)  
  
 **def** simulate(self, duration=8\*60, time\_step=1):  
 *""" Simulation of the transport network """* self.duration = duration  
 *# demand generation* self.gen\_demand(self.duration)  
 **for** ln **in** self.lines:  
 *# define schedules* ln.define\_schedule()  
 **for** v **in** ln.vehicles:  
 *# put zero values to the vehicle characteristics* v.servicing = {}  
 v.passengers = []  
 v.serviced\_passengers = []  
 *# correct the simulation duration* **if** self.duration < v.schedule[-1][0]:  
 self.duration = v.schedule[-1][0]  
 *# run the lines simulation* self.time = 0  
 **while** self.time <= self.duration:  
 **for** ln **in** self.lines:  
 ln.run()  
 self.time += time\_step  
 *# printing out simulation results* self.total\_wait\_time = 0  
 self.num\_serviced\_passengers = 0  
 self.sum\_vehicles\_time = 0  
 **for** ln **in** self.lines:  
 **for** v **in** ln.vehicles:  
 self.sum\_vehicles\_time += v.schedule[-1][0] - v.schedule[0][0]  
 self.num\_serviced\_passengers += len(v.serviced\_passengers)  
 *# calculate sum of waiting time* **for** ps **in** v.serviced\_passengers:  
 self.total\_wait\_time += ps.wait\_time  
*# estimate total wait time of unserved passengers**# (under condition that they wait till the end of simulation)* up\_wait\_time = 0  
 upn = 0  
 **for** ps **in** self.demand:  
 **if** ps.used\_vehicle **is** None:  
 up\_wait\_time += self.time - ps.m\_appearance  
 upn += 1  
 self.total\_wait\_time += up\_wait\_time

**class** Node:  
 *""" Node of the transport net """* **def** \_\_init\_\_(self, code=0, name=**""**):  
 *# type: (int, str) -> Node* self.code = code  
 **if** name **is ""**:  
 self.name = **"Node"** + str(code)  
 **else**:  
 self.name = name  
 *# graph features* self.out\_links = []  
 self.in\_links = []  
 *# demand features* self.s\_interval = None  
 self.pass\_out = []  
 self.pass\_in = []

**class** Link:  
 *""" Link between the net nodes """* **def** \_\_init\_\_(self, out\_node, in\_node, weight=0):  
 *# type: (Node, Node, float) -> Link* self.out\_node = out\_node  
 self.in\_node = in\_node  
 self.lines\_number = 1  
 *# link length [km]* self.weight = weight  
 self.capacity = 0  
 self.load = 0

**class** Passenger:  
 *"""" Passenger traveling in the network """* **def** \_\_init\_\_(self):  
 *# event moments* self.m\_appearance = 0  
 self.m\_boarding = 0  
 self.m\_disembarkation = 0  
 *# travel features* self.origin\_node = None  
 self.destination\_node = None  
 self.used\_vehicle = None  
  
 @property  
 **def** travel\_time(self):  
 **return** self.m\_disembarkation - self.m\_appearance  
  
 @property  
 **def** wait\_time(self):  
 **return** self.m\_boarding - self.m\_appearance  
  
 @property  
 **def** transportation\_time(self):  
 **return** self.m\_disembarkation - self.m\_boarding

**class** Line:  
 *""" Public transport line """* **def** \_\_init\_\_(self, net, codes):  
 *# route configuration* self.net = net  
 self.nodes = []  
 *# checking the route correctness* **if** len(codes) < 2:  
 **raise** Exception(**"Line should contain at least 2 nodes."**)  
 **for** code **in** codes:  
 **if** net.contains\_node(code):  
 self.nodes.append(net.get\_node(code))  
 **else**:  
 **raise** Exception(**"No such a node in the net: "**, code)  
 **for** idx **in** range(len(self.nodes)-1):  
 out\_node = self.nodes[idx]  
 in\_node = self.nodes[idx + 1]  
 **if not** net.contains\_link(out\_node, in\_node):  
 **raise** Exception(**"No such a link in the net: "**,  
 str(out\_node.code) + **" - "** + str(in\_node.code))  
 *# collection of vehicles operating at the line* self.vehicles = []  
 *# schedule parameters [min]* self.nodes\_sequence = []  
 self.end\_stop\_duration = 5  
 self.intermediate\_stop\_duration = 1  
 *# mean velocity of buses at the line [km/h]* self.velocity = 40  
 @property  
 **def** line\_length(self):  
 *"""" Returns the line length [km] """* length = 0  
 **for** idx **in** range(len(self.nodes) - 1):  
 out\_node = self.nodes[idx]  
 in\_node = self.nodes[idx + 1]  
 **if** self.net.contains\_link(out\_node, in\_node):  
 length += self.net.get\_link(out\_node, in\_node).weight  
 **return** length  
 @property  
 **def** trace\_string(self):  
 *"""" Returns the line trace as a sequence of nodes: a string containing*

*the codes of the nodes """* t\_str = str(self.nodes[0].code) + **" - "  
 if** len(self.nodes) > 2:  
 **for** idx **in** range(1, len(self.nodes) - 1):  
 t\_str += str(self.nodes[idx].code) + **" - "** t\_str += str(self.nodes[-1].code)  
 **return** t\_str  
 @property  
 **def** line\_end\_stops(self):  
 *"""" Returns a list containig the codes of end stops """* **return** [self.nodes[0], self.nodes[-1]]  
 @property  
 **def** turnaround\_time(self):  
 *"""" Returns the line turnaround duration [min] """* **return** 2 \* (60 \* self.line\_length / self.velocity + self.end\_stop\_duration +  
 self.intermediate\_stop\_duration \* (len(self.nodes) - 2))  
 @property  
 **def** turns\_number(self):  
 *"""" Returns the possible number of turns during the simulation period """* **return** int(self.net.duration / self.turnaround\_time) + 1  
 @property  
 **def** nodes\_reversed(self):  
 *"""" Returns the list of the line stops (their codes) in the reversed order """* nds = []  
 **for** i **in** range(len(self.nodes) - 1, -1, -1):  
 nds.append(self.nodes[i])  
 **return** nds

**def** define\_sequence(self, turns):  
 nds = []  
 nds.extend(self.nodes[:-1])  
 **for** tn **in** range(self.turns\_number - 1):  
 nds.extend(self.nodes\_reversed)  
 nds.extend(self.nodes[1:-1])  
 nds.extend(self.nodes\_reversed)  
 self.nodes\_sequence = nds  
  
 **def** define\_schedule(self):  
 *"""" Defines the line schedule """* interval = round(self.turnaround\_time / len(self.vehicles), 0)  
 self.define\_sequence(self.turns\_number)  
 t0 = 0  
 **for** vhcl **in** self.vehicles:  
 *# defining start parameters of the vehicle's schedule* vhcl.schedule = []  
 vhcl.last\_move = None  
 *# adding start point to the schedule* st = t0  
 vhcl.schedule.append((st, self.nodes\_sequence[0]))  
 *# time shift for the first stop* st -= self.end\_stop\_duration  
 *# calculating the moments of arrivals to the line stops* **for** idx **in** range(1, len(self.nodes\_sequence)):  
 **if** self.nodes\_sequence[idx-1] **in** self.line\_end\_stops:  
 st += self.end\_stop\_duration  
 **else**:  
 st += self.intermediate\_stop\_duration  
 st += round(60 \* self.net.get\_link(self.nodes\_sequence[idx-1],  
 self.nodes\_sequence[idx]).weight\  
 / self.velocity, 0)  
 vhcl.schedule.append((st, self.nodes\_sequence[idx]))  
 *# time shift for the next vehicle* t0 += interval  
  
 **def** add\_vehicles(self, vehicles):  
 *"""" Adds vehicles to run on the line """* **for** vehicle **in** vehicles:  
 vehicle.line = self  
 self.vehicles.append(vehicle)  
  
 **def** run(self):  
 *"""" Runs the line simulation """* **for** vhcl **in** self.vehicles:  
 vhcl.move()

**class** Vehicle:  
 *"""" Vehicle servicing the line """* **def** \_\_init\_\_(self, capacity=40):  
 *# line where vehicle operates* self.line = None  
 *# schedule (list of tuples <t, node>)* self.schedule = []  
 self.last\_move = None  
 self.servicing = {}  
 self.passengers = []  
 self.serviced\_passengers = []  
 *# vehicle capacity* self.capacity = capacity  
  
 @property  
 **def** occupancy(self):  
 *"""" Returns current occupancy """* **return** len(self.passengers)  
  
 @property  
 **def** model\_time(self):  
 *"""" Returns current model time """* **return** self.line.net.time  
  
 @property  
 **def** current\_position(self):  
 *"""" Returns the current position at the schedule """* **if** self.last\_move **is not** None:  
 **return** self.schedule.index(self.last\_move)  
 **else**:  
 **return** -1  
  
 @property  
 **def** moves\_number(self):  
 *"""" Returns the number of moves according to the schedule """* **return** len(self.schedule)  
  
 @property  
 **def** stops\_left(self):  
 *"""  
 Returns nodes left till the next end stop:  
 is used in order to determine, if the passenger should use the vehicle  
 """* stops = []  
 pos = 0  
 **if** self.last\_move **is not** None:  
 pos = self.current\_position + 1  
 **if** pos < self.moves\_number:  
 next\_stop = self.schedule[pos][1]  
 stops.append(next\_stop)  
 **while** next\_stop **not in** self.line.line\_end\_stops:  
 pos += 1  
 next\_stop = self.schedule[pos][1]  
 stops.append(next\_stop)  
 **return** stops  
  
 **def** set\_passengers(self):  
 *"""" Set passengers at the current stop """* **if** self.last\_move **is not** None: *# excess condition?* passengers = self.passengers[:]  
 **for** psngr **in** self.passengers:  
 **if** psngr.destination\_node **is** self.last\_move[1]:  
 passengers.remove(psngr)  
 psngr.m\_disembarkation = self.line.net.time  
 psngr.destination\_node.pass\_in.append(psngr)  
 self.serviced\_passengers.append(psngr)  
 self.passengers = passengers

**def** get\_passengers(self):  
 *"""" Get passengers at the current stop """* **if** self.last\_move **is not** None: *# excess condition?* passengers = self.last\_move[1].pass\_out[:]  
 **for** passenger **in** self.last\_move[1].pass\_out:  
 **if** passenger.m\_appearance <= self.model\_time **and** passenger.destination\_node **in** self.stops\_left\  
 **and** self.occupancy < self.capacity:  
 passengers.remove(passenger)  
 passenger.m\_boarding = self.line.net.time  
 passenger.used\_vehicle = self  
 self.passengers.append(passenger)  
 self.last\_move[1].pass\_out = passengers  
  
 **def** move(self):  
 *"""" Simulate the vehicle's move to the next stop """* **if** self.last\_move **is not** None:  
 *# the schedule is not covered yet* **if** self.current\_position < self.moves\_number - 1:  
 *# intermediate stops of the route* **if** self.schedule[self.current\_position + 1][0] <= self.model\_time:  
 *# moving to the next stop* self.last\_move = self.schedule[self.current\_position + 1]  
 self.set\_passengers()  
 self.get\_passengers()  
 **else**:  
 *# route ending: self.current\_position == self.moves\_number* **if** self.last\_move[0] <= self.model\_time **and** self.occupancy > 0:  
 self.set\_passengers()  
 **else**:  
 *# starting case* **if** self.schedule[0][0] <= self.model\_time:  
 self.last\_move = self.schedule[0]  
 self.get\_passengers()  
 *# passengers serviced at the current move* self.servicing[self.last\_move] = self.passengers[:]

**import** math  
**import** random  
  
  
**class** Stochastic:  
 *"""" Stochastic variable """* **def** \_\_init\_\_(self, law=0, location=0, scale=1, shape=0):  
 *# distribution law* self.law = law  
 *# location parameter* self.location = location  
 *# scale parameter* self.scale = scale  
 *# form parameter* self.shape = shape  
  
 **def** get\_value(self):  
 *"""" Returns the generated value """* **if** self.law == 0:  
 *# rectangular distribution* **return** random.uniform(self.location, self.location + self.scale)  
 **elif** self.law == 1:  
 *# normal distribution* **return** random.gauss(self.location, self.scale)  
 **elif** self.law == 2:  
 *# exponential distribution* r = random.random()  
 **while** r == 0 **or** r == 1:  
 r = random.random()  
 **return** -self.scale\*math.log(r)  
 **else**:  
 *# rectangular distribution by default* **return** random.uniform(self.location, self.location + self.scale)