**TECHNICAL TEST**

**NOTE: All the .py files are enclosed on git**

Question A

Your goal for this question is to write a program that accepts two lines (x1,x2) and (x3,x4) on the x-axis and returns whether they overlap. As an example, (1,5) and (2,6) overlaps but not (1,5) and (6,8).

*#!/usr/bin/env python*

*'''  
Strategy:  
Read one input from user containing 2 values, store in ons string variable  
Read another input from user containing 2 other values  
Normalize input data  
Build 2 lists, representing a line each one  
Check if they overlap  
Output the result  
'''*

**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
  
 line1 = []  
 line2 = []  
 overlaps = **False** *# read user input* **print**(**"Enter first line (x1,x2): "**)  
 user1 = str(input())  
 **print**(**"enter second line (x3,x4): "**)  
 user2 = str(input())  
  
 *# data pre-processing* user1 = user1.strip(**"("**)  
 user1 = user1.strip(**")"**)  
 user2 = user2.strip(**"("**)  
 user2 = user2.strip(**")"**)  
 values1 = user1.split(**","**)  
 values2 = user2.split(**","**)  
  
 **if** len(values1) != 2:  
 **print** (**"Bad user input for line 1."**)  
 exit()  
 **if** len(values2) != 2:  
 **print** (**"Bad user input for line 2."**)  
 exit()  
  
 **for** element **in** values1:  
 line1.append(int(element.strip(**" "**)))  
 **for** element **in** values2:  
 line2.append(int(element.strip(**" "**)))  
  
 **if** line1[0] > line1[1]:  
 temp = line1[0]  
 line1[0] = line1[1]  
 line1[1] = temp  
 **if** line2[0] > line2[1]:  
 temp = line2[0]  
 line2[0] = line2[1]  
 line2[1] = temp  
  
 *# build the lines* range1 = range(line1[0], line1[1]+1, 1)  
 range2 = range(line2[0], line2[1]+1, 1)  
  
 *# overlapping detection* **for** element **in** range1:  
 **if** element **in** range2:  
 overlaps = **True** *# result output* **if** overlaps:  
 **print**(**"Overlap detected."**)  
 **else**:  
 **print** (**"No overlap detected."**)

Question B

The goal of this question is to write a software library that accepts 2 version string as input and returns whether one is greater than, equal, or less than the other. As an example: “1.2” is greater than “1.1”. Please provide all test cases you could think of.

*#!/usr/bin/env python*

*'''  
Strategy:  
Buid a class taht may accept 2 type of inputs  
Instance an object of this class and use the compare method in both given types  
Run test cases  
'''*

**class** Detection:  
 **def** verifying(self, a, b):  
 *# version 1, receiving two strings* **if** isinstance(a, str) **and** isinstance(b, str):  
 **if** a > b:  
 **return "GT"  
 if** a < b:  
 **return "LT"  
 if** a == b:  
 **return "EQ"  
  
 else**:  
 *# version 2, receiving two floats* **if** isinstance(a, float) **and** isinstance(b, float):  
 **if** a > b:  
 **return "GT"  
 if** a < b:  
 **return "LT"  
 if** a == b:  
 **return "EQ"  
 else**:  
  
 **return "Bad params. Use 2 strings(e.g. \"1\") or 2 floats (e.g. 1.33)"  
  
if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 detection = Detection()  
 *# test cases* **print** (detection.verifying(**""**, **""**)) *# two empty strings -> pass* **print** (detection.verifying(**"pass"**, **"passcode"**)) *# two word strings -> pass* **print** (detection.verifying(**"a"**, **"b"**)) *# two alpha strings -> pass* **print** (detection.verifying(**"2"**, **"1"**)) *# two numeric strings -> pass* **print** (detection.verifying(**"C"**, **"C"**)) *# two equal strings -> pass* **print** (detection.verifying(1.1, 1.3)) *# two floats -> pass* **print** (detection.verifying(2.1, 1.3)) *# two floats -> pass* **print** (detection.verifying(1.0, 1.0)) *# two floats -> pass* **print** (detection.verifying(1, 1.3)) *# one int and one float -> error message* **print** (detection.verifying(**"A"**, 1.3)) *# one string and one float -> error message* **print** (detection.verifying(**"A"**, 1)) *# one string and one int -> error message* **print** (detection.verifying(1.3, 1)) *# one float and one int -> error message* **print** (detection.verifying(1.3, **"A"**)) *# one float and one string -> error message* **print** (detection.verifying(1, **"A"**)) *# one int and one string -> error message***'''  
--OUTPUT:  
EQ  
LT  
LT  
GT  
EQ  
LT  
GT  
EQ  
Bad params. Use 2 strings(e.g. "1") or 2 floats (e.g. 1.33)  
Bad params. Use 2 strings(e.g. "1") or 2 floats (e.g. 1.33)  
Bad params. Use 2 strings(e.g. "1") or 2 floats (e.g. 1.33)  
Bad params. Use 2 strings(e.g. "1") or 2 floats (e.g. 1.33)  
Bad params. Use 2 strings(e.g. "1") or 2 floats (e.g. 1.33)  
Bad params. Use 2 strings(e.g. "1") or 2 floats (e.g. 1.33)  
'''**

Question C

At Ormuco, we want to optimize every bits of software we write. Your goal is to write a new library that can be integrated to the Ormuco stack. Dealing with network issues everyday, latency is our biggest problem. Thus, your challenge is to write a new Geo Distributed LRU (Least Recently Used) cache with time expiration. This library will be used extensively by many of our services so it needs to meet the following criteria:

    1 - Simplicity. Integration needs to be dead simple.

    2 - Resilient to network failures or crashes.

    3 - Near real time replication of data across Geolocation. Writes need to be in real time.

    4 - Data consistency across regions

    5 - Locality of reference, data should almost always be available from the closest region

    6 - Flexible Schema

    7 - Cache can expire

*#!/usr/bin/env python*

*'''  
Strategy:  
Build a class for the GeoItem, containing it's properties  
Build a class for the Cache, containing it's rules  
Provide the interfaces PUT and GET, to deadly simple use.  
  
Instantiate the object GeoItem  
Instantiate the object LRU\_cache  
  
Put GeoItems in cache  
Manipulate GeoItems in cache  
  
To Do: Separate Public and Private sections, granting public access only to the Put and Get methods  
'''*

**class** GeoItem(object):  
 **def** \_\_init\_\_(self, key, value, region, time):  
 self.key = key  
 self.value = value  
 self.region = region  
 self.time = time  
 self.saved = **False** self.consistency = **False** self.previous\_e = **None** self.next\_e = **None  
  
 def** \_\_str\_\_(self):  
 **return "(%s, %s, %s, %s, %s, %s)"** % (self.key, self.value, self.region, self.time, self.saved, self.consistency)  
  
  
**class** LRU(object):  
 **def** \_\_init\_\_(self, limit):  
 self.error = **False** self.check = **False** *# check consistency* self.success = **False** *# Write on disk Success* self.new\_key = 0  
 self.new\_value = **""** self.region = **""** self.time = 0  
  
 **if** limit <= 0:  
 **raise** ValueError(**"Limit must be grater than 0"**)  
  
 self.limit = limit  
 self.actual\_size = 0  
 self.first\_e = **None** self.last\_e = **None** self.lru\_cash\_map = {}  
  
 **def** get\_value(self, s\_key):  
 *# return the element and moves it to the first [0] position of the LRU  
 # if the element has expired or being discarded returns null (None)* **if** s\_key **not in** self.lru\_cash\_map:  
 **return None** element = self.lru\_cash\_map[s\_key]  
 **if** self.first\_e == element:  
 **return** element.value  
 self.remove\_element(element)  
 self.insert\_element(element)  
 self.time\_tick()  
 self.write\_to\_disk() *# To do* self.check\_consistency(**"A"**) *# To do: pass self.region* **return** element.value  
  
 **def** put\_value(self, new\_element):  
 *# insert new value in the first position [0] of the LRU list* self.new\_key = new\_element.key  
 self.new\_value = new\_element.value  
 self.region = new\_element.region  
 self.time = new\_element.time  
  
 self.time\_tick() *# decreases lifetime for everyone before inserting the new item  
 # so if any item expires opens a new place in the cache* **if** self.new\_key **in** self.lru\_cash\_map:  
 element = self.lru\_cash\_map[self.new\_key]  
 element.value = self.new\_value  
 **if** self.first\_e != element:  
 self.remove\_element(element)  
 self.insert\_element(element)  
 **else**:  
 insert\_element = GeoItem(self.new\_key, self.new\_value, self.region, self.time)  
 **if** self.actual\_size == self.limit:  
 **del** self.lru\_cash\_map[self.last\_e.key]  
 self.remove\_element(self.last\_e)  
 self.insert\_element(insert\_element)  
 self.lru\_cash\_map[self.new\_key] = insert\_element  
  
 self.write\_to\_disk()  
  
 **def** remove\_element(self, element):  
 **if not** self.first\_e:  
 **return** *# remove one element in the middle, not first, not last one* **if** element.previous\_e:  
 element.previous\_e.next\_e = element.next\_e  
 **if** element.next\_e:  
 element.next\_e.previous\_e = element.previous\_e  
  
 *# Only one element on lru cache* **if not** element.next\_e **and not** element.previous\_e:  
 self.first\_e = **None** self.last\_e = **None** *# if the element is the last* **if** self.last\_e == element:  
 self.last\_e = element.next\_e  
 self.last\_e.prev\_e = **None** self.actual\_size -= 1  
 **return** element  
  
 **def** insert\_element(self, element):  
 *# insert element at cache first position (newest)* **if not** self.first\_e:  
 self.first\_e = element  
 self.last\_e = element  
 **else**:  
 element.previous\_e = self.first\_e  
 self.first\_e.next\_e = element  
 self.first\_e = element  
 self.actual\_size += 1 *# limit is verified in put\_elemet()* **def** check\_consistency(self, region):  
 *# TO DO: Call close regions to check data consistency* self.check = region  
 *# to do: verify neighbors, compare data, check consistency* self.error = **False  
 if** self.error:  
 **return "Error "**+self.check  
 **else**:  
 **return** self.check+**" OK"  
  
 def** write\_to\_disk(self):  
 *# TO DO: write the file to disk* self.error = **False** self.success = **not** self.error  
 **return** self.success  
  
 **def** time\_tick(self):  
 *# every manipulation on the LRU decreases the time to live of each object on list  
 # when the time falls to 0 the item is discarded (expires)* element = self.first\_e  
 *# run through all elements* **while** element:  
 element.time -=1  
 **if** element.time <= 0:  
 self.remove\_element(element)  
 element = element.previous\_e  
  
 **def** print\_values(self):  
 element = self.first\_e  
 *# run through all elements* **while** element:  
 **print** element *# using builtin \_\_str\_\_ defined in class* element = element.previous\_e  
 **print  
  
if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 cache = LRU(5) *# create the LRU cache with 5 elements limit  
 # All items starting with same Time To Live (TTL = 10)* item = GeoItem(1, **"Quick"**, **"A"**, 05) *# create an item key = 1, value = Quick, Region = A, time to live = 5* cache.put\_value(item) *# Put the item on the cache* item = GeoItem(2, **"Brown"**, **"B"**, 05) *# create an item key = 2, value = Brown, Region = B, time to live = 6* cache.put\_value(item) *# Put the item on the cache* item = GeoItem(3, **"Fox"**, **"A"**, 05) *# create an item key = 3, value = Fox, Region = A, time to live = 2* cache.put\_value(item) *# Put the item on the cache* item = GeoItem(4, **"Jumps"**, **"C"**, 05) *# create an item key = 4, value = Jumps, Region = C, time to live = 1* cache.put\_value(item) *# Put the item on the cache* item = GeoItem(5, **"Over"**, **"C"**, 05) *# create an item key = 5, value = Over, Region = C, time to live = 8* cache.put\_value(item) *# Put the item on the cache* item = GeoItem(6, **"Lazy"**, **"D"**, 05) *# create an item key = 6, value = Lazy, Region = D, time to live = 9* cache.put\_value(item) *# Put the item on the cache - Oldest item (1 - quick) is discarded  
 # each move made on the cache decreases TTL for existing elements* cache.print\_values()  
  
 cache.get\_value(3) *# get the element 3,  
 # move it to the first position [0].  
 # Decreases lifetime for all elements on LRU.  
 # item 2 (Brown) expires.* cache.print\_values()  
  
**'''   
--OUTPUT:  
  
(6, Lazy, D, 5, False, False)  
(5, Over, C, 4, False, False)  
(4, Jumps, C, 3, False, False)  
(3, Fox, A, 2, False, False)  
(2, Brown, B, 1, False, False)  
  
(3, Fox, A, 1, False, False)  
(6, Lazy, D, 4, False, False)  
(5, Over, C, 3, False, False)  
(4, Jumps, C, 2, False, False)  
'''**

As a hint, we are not looking for quantity, but rather quality, maintainability, scalability, testability and a code that you can be proud of.

When submitting your code add the necessary documentation to explain your overall design and missing functionalities.  Do it to the best of your knowledge.

Good Luck, **you can write it in the language of your choice**, name the test using the convention firstname\_lastname\_test and provide a link in your personal github so we can review the work.