**LRU Cache**

Design and implement a data structure for [Least Recently Used (LRU) cache](https://en.wikipedia.org/wiki/Cache_replacement_policies#LRU). It should support the following operations: get and put.

get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.  
put(key, value) - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.

**Follow up:**  
Could you do both operations in **O(1)** time complexity?

**Example:**

LRUCache cache = new LRUCache( 2 /\* capacity \*/ );

cache.put(1, 1);

cache.put(2, 2);

cache.get(1); // returns 1

cache.put(3, 3); // evicts key 2

cache.get(2); // returns -1 (not found)

cache.put(4, 4); // evicts key 1

cache.get(1); // returns -1 (not found)

cache.get(3); // returns 3

cache.get(4); // returns 4

from collections import OrderedDict

class LRUCache:

def \_\_init\_\_(self, capacity: int):

self.capacity = capacity

self.dict = OrderedDict()

def get(self, key: int) -> int:

if(key in self.dict):

self.dict.move\_to\_end(key)

return self.dict[key]

else:

return -1

def put(self, key: int, value: int) -> None:

if (key in self.dict):

self.dict.move\_to\_end(key)

self.dict[key] = value

if(len(self.dict) > self.capacity):

self.dict.popitem(last=False)