THON

ython Priority Queue (Step By Step uide)

Mokhtar Ebrahim Published: June 11, 2021

A queue is a data structure that retrieves data items in an order called FIFO **(first in first out)**. In FIFO, the first inserted element will be popped out first from the queue.
The Priority Queue is an advanced version of the Queue data structure.

The element with the highest priority is placed on the very top of the Priority Queue and is the first one to be dequeued.

Sometimes, a queue contains items that have equal priorities; therefore, the items will be dequeued according to their order in the queue as in FIFO.

In Python, there are several options to implement Priority Queue. The **queue** standard library in Python supports Priority Queue.

Similarly, the **heapq** module in Python also implements Priority Queue. We can also use **list**, **dict**, and **tuple** modules to implement Priority Queue.

In this tutorial, you will learn how to create a Priority Queue and various other operations that can be performed on elements in a Priority Queue.

Table of Contents

13. Tie breaking

Why Priority Queue?

There are many applications of Priority Queue in the computer world. For example:

Operating systems use the Priority Queue to balance or distribute the load (set of tasks) among different computing units. This makes processing efficient hence introducing parallel computing.

Priority Queue is used for interrupt handling in operating systems.

- In artificial intelligence, Priority Queue implements the A* search algorithm. It keeps track of the unexplored routes and finds the shortest path between different vertices of the graph. The smaller the length of the path, the highest is its priority.
- When implementing Dijkstra's algorithm, Priority Queue finds the shortest path in a matrix or adjacency list graph efficiently.
- Priority Queue sorts heap. Heap is an implementation of Priority Queue

How to create a Priorit

An element in Priority Queue alway

Using list:

Implementing a Priority Queue using a list is pretty straightforward. Just create a list, append elements (key, value), and sort the list every time an element is appended.

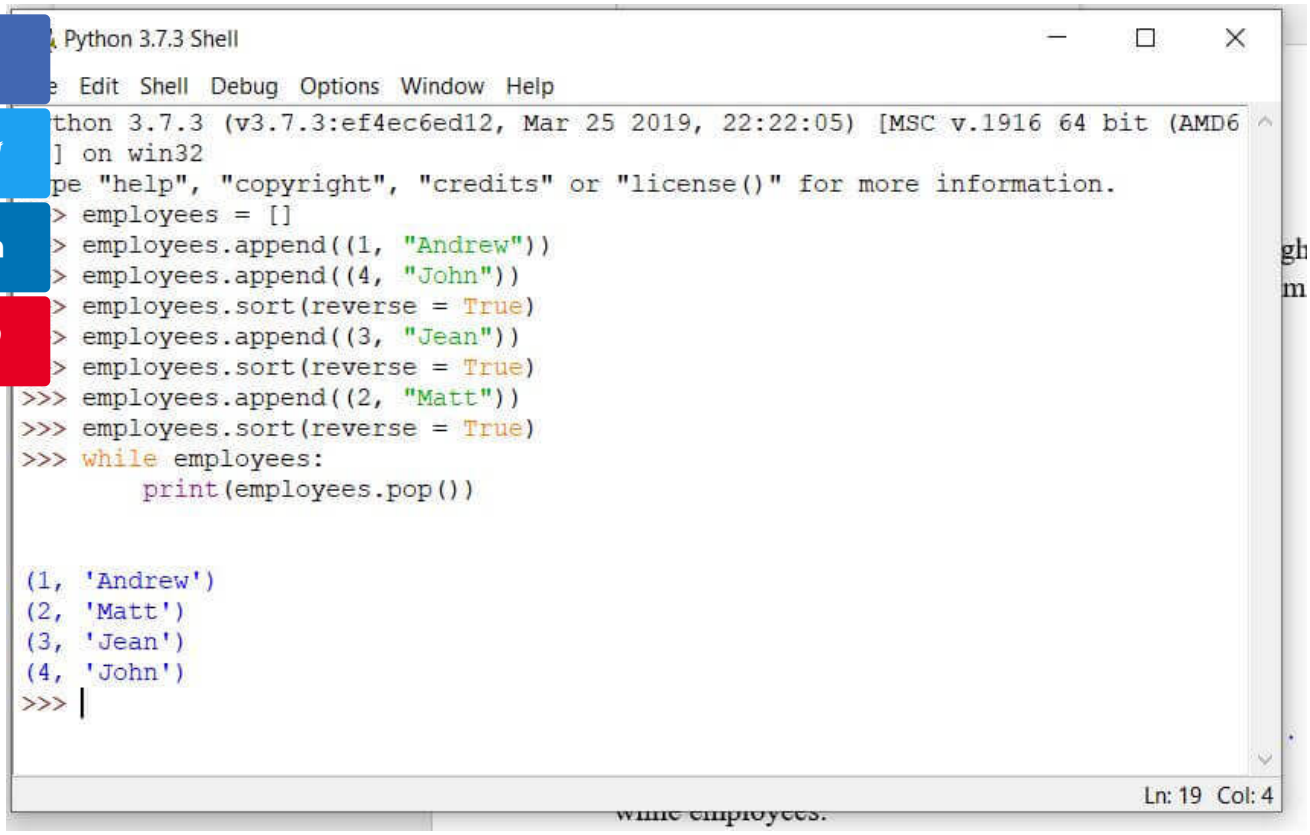
Code:


```
employees = []

employees.append((1, "Andrew"))

employees.append((4, "John"))

employees.sort(reverse = True)

employees.append((3, "Jean"))

employees.sort(reverse = Tru

employees.append((2, "Matt"))

employees.sort(reverse = Tru
```

```
        print(employees.pop())
```

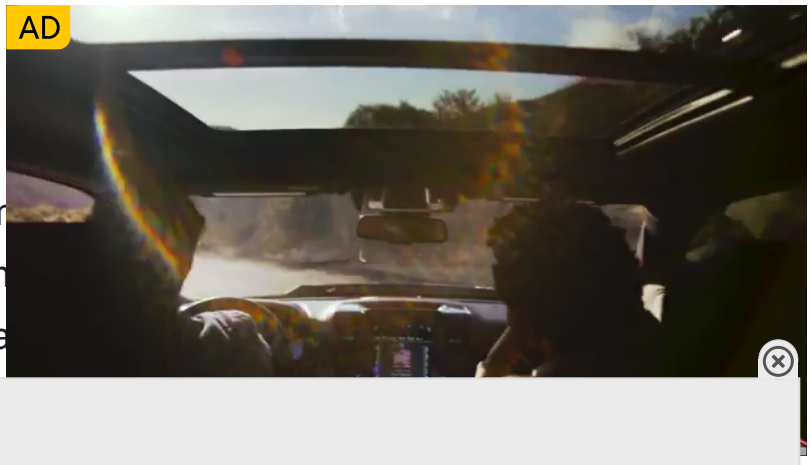
When the first element is appended to the list, there is no need to sort the list. List implementation of Priority Queue is not efficient as the list needs to be sorted after every new entry. Hence, it takes time to maintain the order of elements according to their priority.

Output:



Using tuples

Python tuples and lists are the sam
are ordered data structures of Pyth
elements of a list are changeable a



To implement Priority Queue with tuples, we will create a tuple first with elements of a priority queue and then we will sort the tuple.

Since you cannot change the elements in a tuple, tuples do not provide a regular sort function like lists. To sort out a tuple, we need to use the sorted function.

The difference between the sort and the sorted methods is that the sort method does not return anything and it makes changes to the actual sequence of the list.

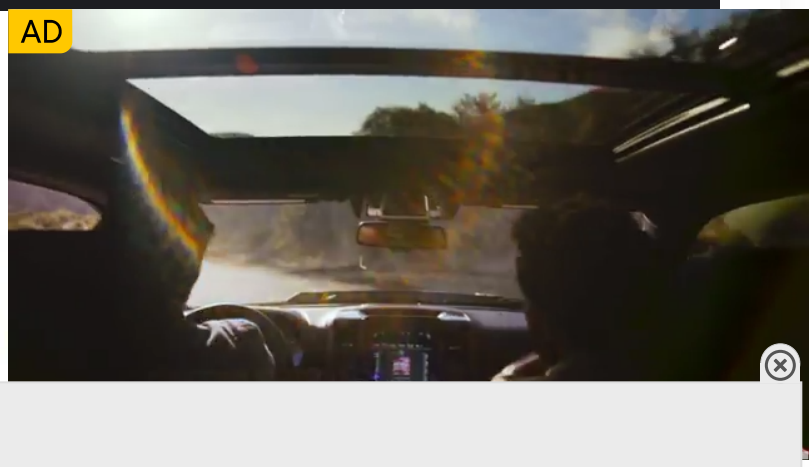
Whereas, the sorted function always returns the sorted sequence and does not disturb the actual sequence of a tuple.

In the following line of code, we will create a tuple and implement Priority Queue with a tuple:

```
mytuple = ((1, "bread"), (3, "pizza"), (2, "apple"))
```

Now let us sort the tuple using sorted() method:

```
sorted(mytuple)
```

Output:

```
>>> mytuple = ((1, "bread"),
>>> sorted(mytuple)
[(1, 'bread'), (2, 'apple'),
>>>
```

Using dictionary

In a Python dictionary, data is stored in pairs that are a key and a value. We will use the key as the priority number of the element and the value to be the value of the queue element.

is way, we can implement Priority Queue using the default Python dictionary.
Create a dictionary and add items (keys and values):

```
mydict = {2: "Asia", 4: "Europe", 3: "America", 1: "Africa"}
```

After creating the dictionary, you need to sort its items by key. We need to store the items of the dictionary int
method:

```
dict_items = mydict.items()
```


```
print(sorted(dict_items))
```

Output:

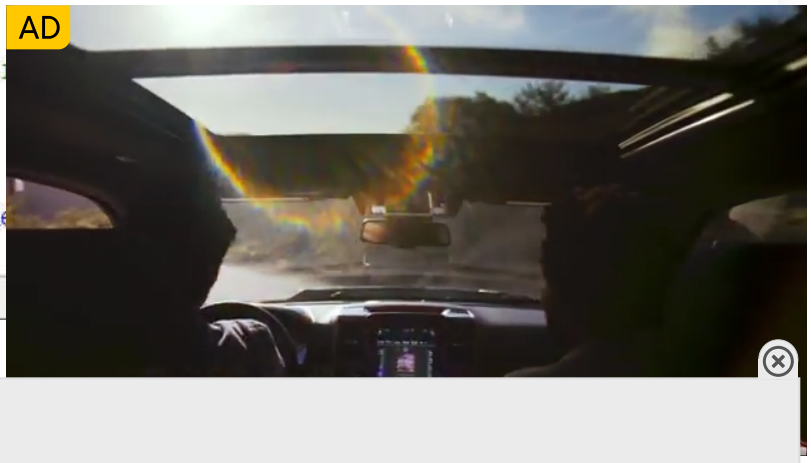
```
>>> mydict = {2: "Asia", 4: "Europe", 3: "America", 1: "Africa"}
>>> dict_items = mydict.items()
>>> print(sorted(dict_items))
[(1, 'Africa'), (2, 'Asia'), (3, 'America'), (4, 'Europe')]
>>>
```

pop items from the dictionary priority queue, you can use the **popitem()** ethod. The dictionary popitem() method will dequeue the element with the ghest priority:

```
mydict = {2: "Asia", 4: "Europe", 3: "America", 1: "Africa"}

mydict.popitem()

print(mydict)
```

Output:

```
>>> mydict = {2: "Asia", 4: "Eu
>>> mydict.popitem()
(1, 'Africa')
>>> print(mydict)
{2: 'Asia', 4: 'Europe', 3: 'Am
>>>
```

Let us create a Priority Queue using the built-in **queue** module in Python. Using the queue module is the simplest usage of Priority Queue.

Code:

```
import queue

p_queue = queue.PriorityQueue()

p_queue.put((2, "A"))

p_queue.put((1, "B"))

p_queue.put((3, "C"))
```

In this code, the constructor PriorityQueue() creates a priority queue and stores it in the variable p_queue. The put(priority_number, data) function of PriorityQueue class inserts an item in the queue.

The put(priority_number, data) function takes two arguments: the first argument is an integer to specify the priority number of the element in the queue, and the second argument is the element that is to be inserted in the queue.

To pop and return the items from t

```
print(p_queue.get())
```

As you can see all items are dequeued. To check if there exists any element in the queue, the empty() function is used. The empty() function returns a Boolean value. If it returns true, it means the queue is empty.

```
p_queue.empty()
```

sing heapdict

The **heapdict** module is similar to a regular dictionary in Python but in heapdict, you can pop the items and can also change the priority of them items in a Priority Queue.

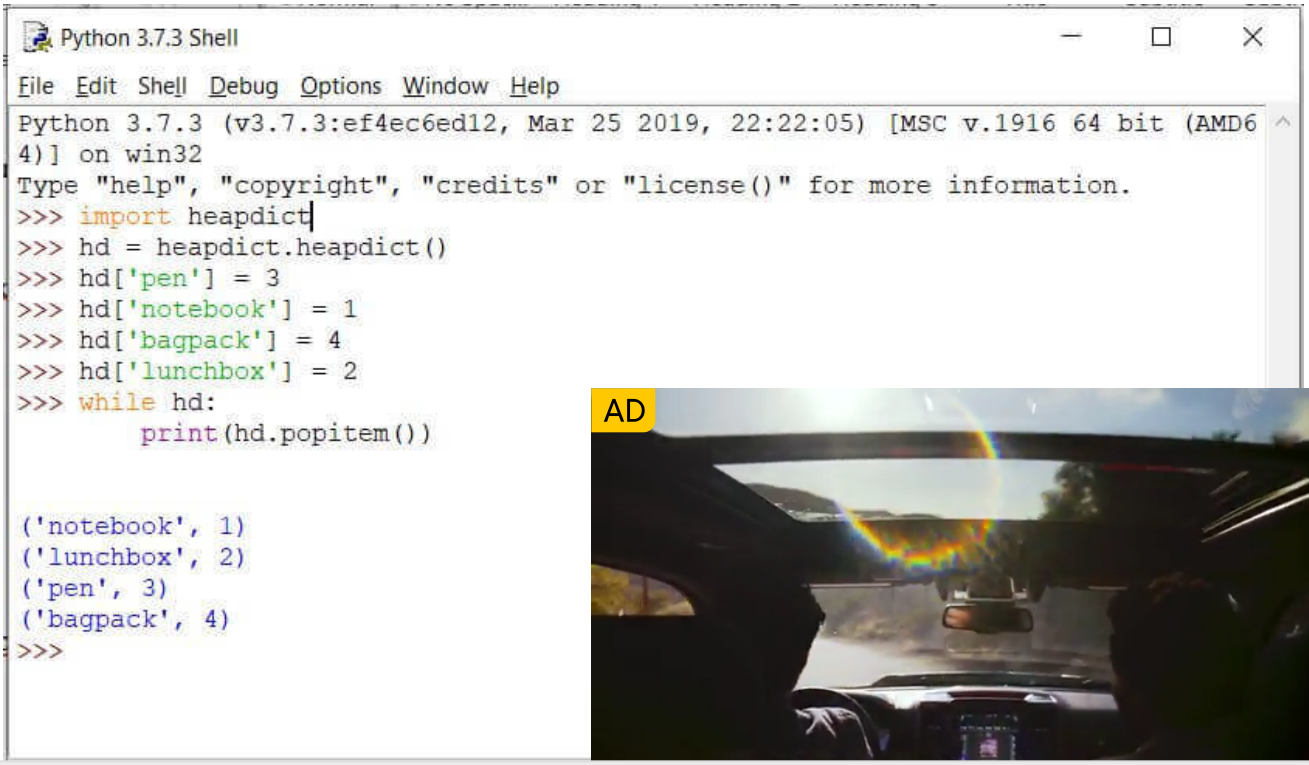
With heapdict, you can change the priority of items: that is, increase or decrease the key of the item.
The heapdict module is not installed by default. To install heapdict:

```
pip install heapdict
```

Now let's implement Priority Queue

```
import heapdict

hd = heapdict.heapdict()

hd['pen'] = 3

hd['notebook'] = 1

hd['bagpack'] = 4

hd['lunchbox'] = 2

while hd:

        print(hd.popitem())
```

Output:

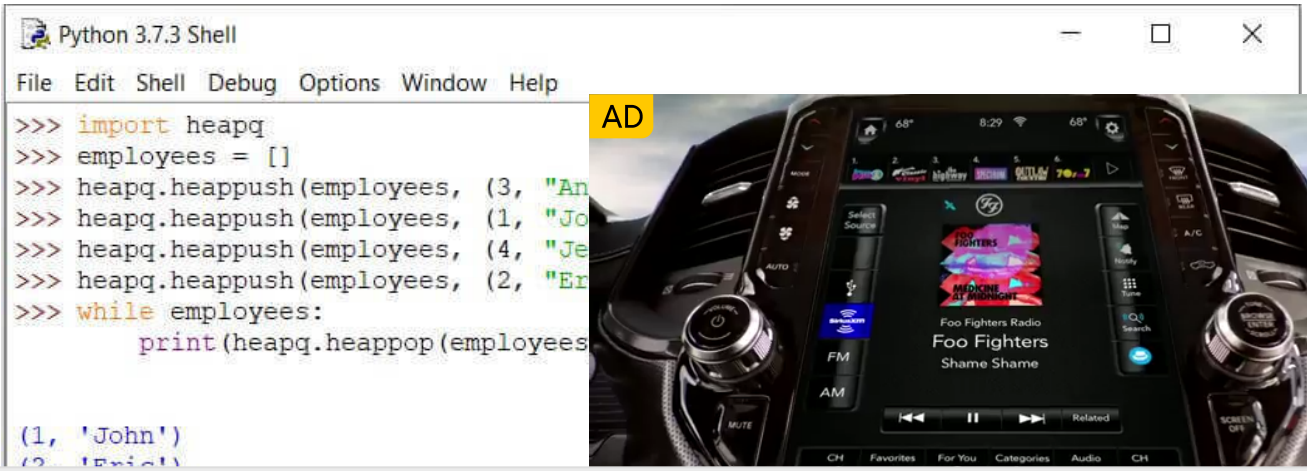


The heap data structure in the computer world is mainly aimed at implementing the priority queue. The heapq module in Python can be used to implement Priority Queue.

Code:

```
import heapq

employees = []

heapq.heappush(employees, (3, "Andrew"))

heapq.heappush(employees, (1, "John"))

heapq.heappush(employees, (4, "Jean"))

heapq.heappush(employees, (2, "Eric"))

while employees:

        print(heapq.heappop(employees))
```

Output:



In this code, a heap is created and the elements (priority key, value) are pushed into the heap.
The **heapq** module implements min-heap by default. The element with the smallest key is considered to have the highest priority in min-heap.

Therefore, the smallest element will be popped out first regardless of the order in which the elements were queued as shown in the output screen above.

The heapq module maintains the heap structure itself whenever an element is pushed or popped.
This tutorial will use heapq implementation of Priority Queue.

Priority Queue vs min

A Priority Queue is an implementat
implementation can be a max heap
Priority Queue is a max-heap, then

In a min-heap, the smallest node is the root of the binary tree.
Both priority queue and min heap are the same. The only difference is that in a priority queue the order of the elements depends on the priority number of the element.

Get a value at index

We can use heap implementation of Priority Queue to get value at an index. We create a heap first, then push items into the heap. An item in the Priority Queue will have a key and a value.

This key is not the index of the heap. This key quantifies the priority. The index is the location where the item (key, value) of the Priority Queue is stored.
Consider the example below:

Code:

```
import heapq

employees = []

heapq.heappush(employees, (3
heapq.heappush(employees, (1
heapq.heappush(employees, (4
```

```
print("Value at index 3: ", employees[3])
```

Output:

Delete an element

To delete an element from a Priority Queue you can just pop the element. The element with the highest priority will be dequeued and deleted from the queue.

Create a queue:

Code:

```
import heapq

hq = []

heapq.heappush(hq, (3, "Jean"))

heapq.heappush(hq, (2, "Eric"))

heapq.heappush(hq, (4, "Monica"))

heapq.heappush(hq, (1, "Joey"))

heapq.heappop(hq)
```

Output:

To update priority in Priority Queue, get the index of the element that you want to update the priority of and assign a new key to the element.

You can change the value of the element as well. Check out the code below:

Code:

```
import heapq

hq = []

heapq.heappush(hq, (3, "Jean"))

heapq.heappush(hq, (2, "Eric"))

heapq.heappush(hq, (4, "Monica"))

heapq.heappush(hq, (1, "Joey"))

print(hq)

hq[1] = (6, 'Eric')

print(hq)

heapq.heapify(hq)

print(hq)
```

Output:

ter updating the priority of an element, we need to heapify the heap to aintain the heap data structure. The **heapify()** method of heapq module nverts Python iterables into the heap data structure.

Replace an element

In the heap implementation of Priority Queue, you can pop the item with the highest priority and push the new item at the same time meaning that you are replacing the highest priority item with a new one.

This is done with the help of a **heapq** function called **heapreplace**:

```
heapq.heapreplace(heap, item)
```

You will pass the queue to pop an item from and pass the new item to add into the queue.

Code:

```
import heapq

hq = []

heapq.heappush(hq, (3, "Jean"))

heapq.heappush(hq, (2, "Eric"))

heapq.heappush(hq, (4, "Monica"))

heapq.heappush(hq, (1, "Joey"))

heapq.heapify(hq)

print(hq)

heapq.heapreplace(hq, (6, "Ross"))

print(hq)
```

Output:

The **heapreplace()** function dequeues the element with the highest priority and adds the new element in the queue. The priority of the new element is the lowest. Therefore, it is put to the last of the queue.

The **heapq** module also provides a method called **heappushpop(heap, item)**. The heappushpop(heap, item) combines the functionality of the heappop() and the heappush() methods.

The heappushpop() method increases the efficiency and takes less time than to push and pop an element using separate functions.

The difference between heapreplace() and heappushpop() is that heapreplace() pops the item first and then pushes the new item, which is the actual definition of replace.

Whereas, heappushpop() pushes a new item to the queue, and then pops the smallest item.

```
import heapq

heap = []

heapq.heappush(heap, (3, "Africa"))

heapq.heappush(heap, (2, "America"))

heapq.heappush(heap, (1, "Asia"))

heapq.heappush(heap, (4, "Europe"))

heapq.heappushpop(heap, (5, "Antarctica"))

while heap:

        heapq.heappop(heap)
```

Output:

Find top items without removing

To find the top items in a queue without popping them, **heapq** provides a function called **nlargest(n, heap)**.
This function returns n number of top items in the priority queue.

Code:

```
import heapq

heap = []

heapq.heappush(heap, (3, "eat"))

heapq.heappush(heap, (1, "study"))

heapq.heappush(heap, (2, "re

heapq.heappush(heap, (4, "sl

heapq.nlargest(3, heap)
```

Output:

can be seen in the output that the items at the top of the Priority Queue are urned when **nlargest()** function was used. Note that the function only urns the items and it does not dequeue the items as shown by the print mmand.

Find bottom items without removing

To find the items at the bottom in a Priority Queue without popping them, **heapq** provides a function called **nsmallest(n, heap)**. This function returns n number of items at the bottom in the priority queue.

Code:


```
import heapq

heap = []
```

```
heapq.heappush(heap, (1, "study"))

heapq.heappush(heap, (2, "rest"))

heapq.heappush(heap, (4, "sleep"))

heapq.nsmallest(3, heap)

print(heap)
```

Output:

It can be seen in the output that the items at the bottom of the Priority Queue are returned when **nsmallest()** function was used. Note that the function only returns the items and by the print command.

Python priority queue

A custom comparator is used to compare two user-defined iterable objects. In Python Priority Queue, a custom comparator can be used to sort the queue based on user-defined values.

For example, we create a Priority Queue using heapq. Then we sort the heapq using the sorted() method.

It will sort the elements in the queue according to the keys (priority number) of the elements. Consider the example below:

Code:

```
import heapq

heap = []

heapq.heappush(heap, (3, "ea

heapq.heappush(heap, (1, "st

heapq.heappush(heap, (2, "re
```

```
print(sorted(heap))
```

Output:

Now let us sort our queue based on our custom comparator. We want to arrange the elements in the queue in such a way that the values are in alphabetical order after sorting the queue.

For this, we will be using the lambda function. A lambda function is a small anonymous function that comprises of one expression with any number of arguments.

The lambda function or lambda expression returns a value that can be used anywhere in the program.

Code:

```
import heapq

heap = []

heapq.heappush(heap, (3, "ea
```

```
heapq.heappush(heap, (2, "rest"))

heapq.heappush(heap, (4, "sleep"))

print(sorted(heap, key=lambda heap: heap[1]))
```

Output:

this example, the lambda expression tells to sort the queue based on the ues (not keys) in alphabetical order. The sorted() method takes three guments:

- The **iterable**: sequence to be sorted
- **Key**: the key is optional. It is considered as a basis of sort comparison. Key is the user-defined comparator function.
- **Reverse**: Reverse is a Boolean. If it is set to true, it will reverse the sorted sequence. The reverse argument is false by default meaning that it will sort the sequence in ascending order. If reverse is set to true, the sequence will be in descending

Reverse priority queue

If the **reverse** argument is set to true, it will change the sequence in descending order as demonstrated in the example below:

Code:

```
import heapq

heap = []

heapq.heappush(heap, (3, "Africa"))

heapq.heappush(heap, (1, "America"))

heapq.heappush(heap, (2, "Asia"))

heapq.heappush(heap, (4, "Europe"))

print(sorted(heap, reverse=True))
```

Output:

If there are duplicate keys of elements in Priority Queue, it means the priority of those elements is the same. But the question is which element will be dequeued first?

Well, the element that is on the top of the queue will be dequeued first from the queue.

Code:

```
import heapq

heap = []

heapq.heappush(heap, (3, "Africa"))

heapq.heappush(heap, (2, "America"))

heapq.heappush(heap, (1, "Asia"))

heapq.heappush(heap, (1, "Europe"))

while heap:

        print(heap.pop())
```

Output:

ie breaking

Tie in priority queue occurs when there are elements with the same priority. When two elements are incomparable that is if the comparator returns 0 after comparing a and b elements of the queue.

In such a case, the Priority Queue has to decide which element will be dequeued first.

This is call tie-breaking.
We can implement FIFO (first in firs
priority queue if a tie occurs.

multiple clients around the world. He loves writing shell and Python scripts to automate his work.

LEAVE A REPLY

ur email address will not be published. Required fields are marked *

mment

Name *

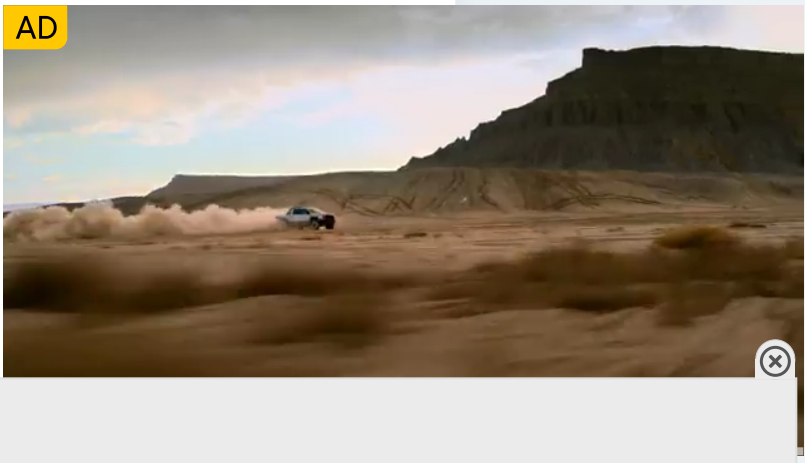
Email *

Replies to my comments Notify me subscribe without commenting.

Email

Subscribe

MY PUBLISHED BOOK



ADVERTISEMENTS

Slicing In Python (Comprehensive Tutotial)

Generate random numbers in Python

ADVERTISEMENTS

LATEST POSTS

CSV processing using Python

Remove punctuation using Python

JSON Processing using Python

Generate random numbers in Python

Python Priority Queue (Step By Step Guide)

Slicing In Python (Comprehensive Tutotial)

Remove punctuation using Python

JSON Processing using Python

Python math functions (Simple Examples)

NumPy array reshape (Shape transformation without data change)

Generate random numbers in Python

Slicing In Python (Comprehensive Tutotial)

Modulo Operator In Python (Simplified Examples)

ADVERTISEMENTS

LATEST COMMENTS

Mokhtar Ebrahim on **Best Linux distro that fits your needs**

themainliner on **Best Linux distro that fits your needs**

Mokhtar Ebrahim on **NLP Tutorial Using Python NLTK (Simple Examples)**

K. Nanda Kumar on **NLP Tutorial Using Python NLTK (Simple Examples)**

Mokhtar Ebrahim on **Dijkstra's algorithm in Python (Find Shortest & Longest Path)**

ADVERTISEMENTS

PICKED FOR YOU

Python GUI examples (Tkinter Tutorial)

30 Examples for Awk Command in Text Processing

Expect command and how to automate shell scripts like magic

NLP Tutorial Using Python NLTK (Simple Examples)

31+ Examples for sed Linux Command in Text Manipulation

PyQt5 tutorial – Python GUI programming examples

Kivy tutorial – Build desktop GUI apps using Python

Python web scraping tutorial (with examples)

ADVERTISEMENTS