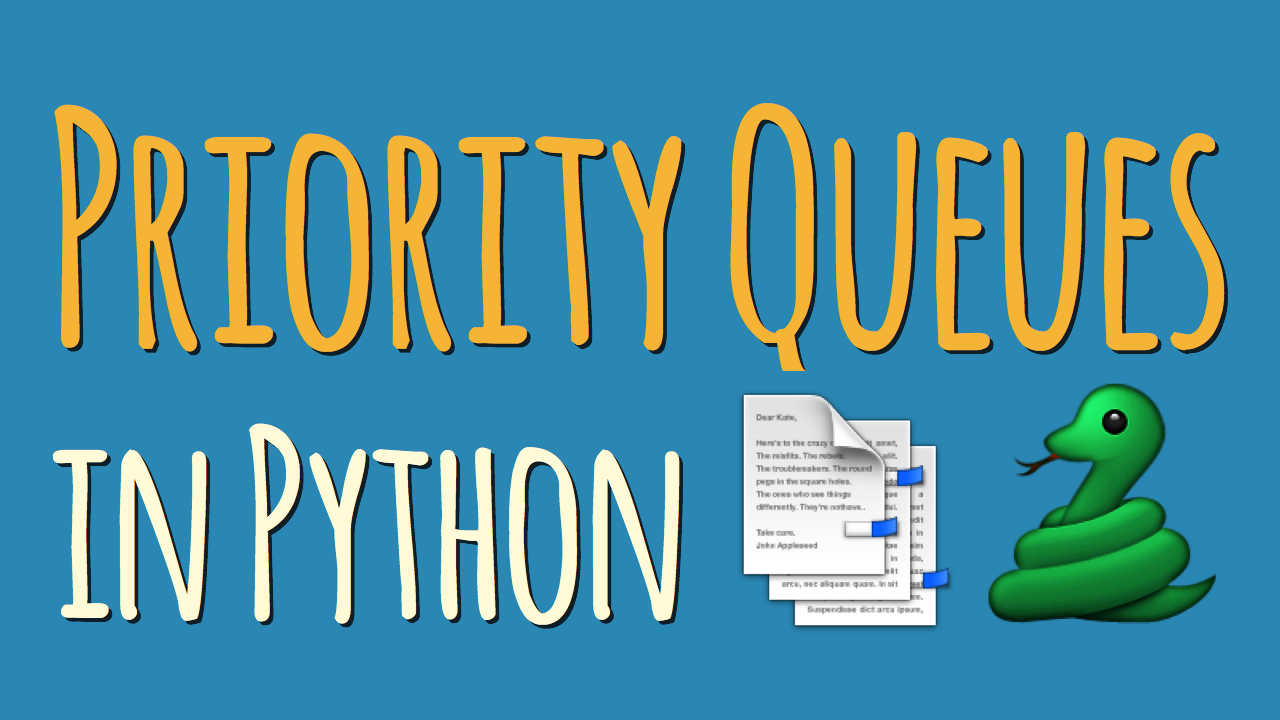
**Priority Queues in Python**

<https://dbader.org/blog/priority-queues-in-python>

By Dan Bader — Get free updates of new posts [here](https://www.getdrip.com/forms/80014959/submissions/new).

What are the various ways you can implement a priority queue in Python? Read on and find out what the Python standard library has to offer.



A priority queue is a container data structure that manages a set of records with [totally-ordered](https://en.wikipedia.org/wiki/Total_order) keys (for example, a numeric *weight* value) to provide quick access to the record with the *smallest* or *largest* key in the set.

You can think of a priority queue as a modified queue: instead of retrieving the next element by insertion time, it retrieves the *highest-priority* element. The priority of individual elements is decided by the ordering applied to their keys.

Priority queues are commonly used for dealing with scheduling problems. For example, to give precedence to tasks with higher urgency.

For example, let’s take an operating system task scheduler—ideally, high-priority tasks on the system (e.g. playing a real-time game) should take precedence over lower-priority tasks (e.g. downloading updates in the background). By organizing pending tasks in a priority queue that uses the task urgency as the key, the task scheduler can allow the highest-priority tasks to run first.

Let’s take a look at a few options for how you can implement Priority Queues in Python using built-in data structures or data structures that ship with Python’s standard library. They each have their up- and downsides, but in my mind there’s a clear winner for most common scenarios. But see for yourself:

**⛔ Keeping a Manually Sorted List**

You can use a sorted list to quickly identify and delete the smallest or largest element. The downside is that inserting new elements into a list is a slow *O(n)* operation.

While the insertion point can be found in *O(log n)* time using [bisect.insort](https://docs.python.org/3/library/bisect.html#bisect.insort) in the standard library, this is always dominated by the slow insertion step.

Maintaining the order by appending to the list and re-sorting also takes at least *O(n log n)* time.

Therefore sorted lists are **only suitable when there will be few insertions** into the priority queue.

q = []

q.append((2, 'code'))

q.append((1, 'eat'))

q.append((3, 'sleep'))

# NOTE: Remember to re-sort every time

# a new element is inserted, or use

# bisect.insort().

q.sort(reverse=True)

while q:

next\_item = q.pop()

print(next\_item)

# Result:

# (1, 'eat')

# (2, 'code')

# (3, 'sleep')

**✅ The** [**heapq**](https://docs.python.org/3/library/heapq.html) **Module**

This is a binary heap implementation usually backed by a plain list and it supports insertion and extraction of the smallest element in *O(log n)* time.

This module is a good choice for implementing priority queues in Python. Because heapq technically only provides a min-heap implementation, extra steps must be taken to ensure sort stability and other features [typically expected from a “practical” priority queue](https://docs.python.org/3/library/heapq.html#priority-queue-implementation-notes).

import heapq

q = []

heapq.heappush(q, (2, 'code'))

heapq.heappush(q, (1, 'eat'))

heapq.heappush(q, (3, 'sleep'))

while q:

next\_item = heapq.heappop(q)

print(next\_item)

# Result:

# (1, 'eat')

# (2, 'code')

# (3, 'sleep')

**✅ The** [**queue.PriorityQueue**](https://docs.python.org/3/library/queue.html#queue.PriorityQueue) **Class**

This priority queue implementation uses heapq internally and shares the same time and space complexities.

The difference is that PriorityQueue is synchronized and provides locking semantics to support multiple concurrent producers and consumers.

Depending on your use case this might be helpful, or just incur unneeded overhead. In any case you might prefer its class-based interface over using the function-based interface provided by heapq.

from queue import PriorityQueue

q = PriorityQueue()

q.put((2, 'code'))

q.put((1, 'eat'))

q.put((3, 'sleep'))

while not q.empty():

next\_item = q.get()

print(next\_item)

# Result:

# (1, 'eat')

# (2, 'code')

# (3, 'sleep')

**A good default choice: queue.PriorityQueue**

Now which priority queue implementation should you use in your Python programs? They each have slightly different use cases. But in my mind queue.PriorityQueue is a good default choice.

Sure, it might incur some unnecessary locking overhead—but it has a nice object-oriented interface and a name that states its intent clearly.

[*Read the full “Fundamental Data Structures in Python” article series here.*](https://dbader.org/blog/fundamental-data-structures-in-python) *This article is missing something or you found an error? Help a brother out and leave a comment below.*