`!heapq` --- Heap queue algorithm

- Basic Examples
- Other Applications
- Priority Queue Implementation Notes
- Theory

heapq

Source code: `Lib/heapq.py`

This module provides an implementation of the heap queue algorithm, also known as the priority queue algorithm.

Min-heaps are binary trees for which every parent node has a value less than or equal to any of its children. We refer to this condition as the heap invariant.

For min-heaps, this implementation uses lists for which `heap[k] <= heap[2*k+1]` and `heap[k] <= heap[2*k+2]` for all *k* for which the compared elements exist. Elements are counted from zero. The interesting property of a min-heap is that its smallest element is always the root, `heap[0]`.

Max-heaps satisfy the reverse invariant: every parent node has a value *greater* than any of its children. These are implemented as lists for which `maxheap[2*k+1] <= maxheap[k]` and `maxheap[2*k+2] <= maxheap[k]` for all *k* for which the compared elements exist. The root, `maxheap[0]`, contains the *largest* element; `heap.sort(reverse=True)` maintains the max-heap invariant.

The `!heapq` API differs from textbook heap algorithms in two aspects: (a) We use zero-based indexing. This makes the relationship between the index for a node and the indexes for its children slightly less obvious, but is more suitable since Python uses zero-based indexing. (b) Textbooks often focus on max-heaps, due to their suitability for in-place sorting. Our implementation favors min-heaps as they better correspond to Python `lists <list>`.

These two aspects make it possible to view the heap as a regular Python list without surprises: `heap[0]` is the smallest item, and `heap.sort()` maintains the heap invariant!

Like `list.sort`, this implementation uses only the `<` operator for comparisons, for both min-heaps and max-heaps.

In the API below, and in this documentation, the unqualified term *heap* generally refers to a min-heap. The API for max-heaps is named using a `_max` suffix.

To create a heap, use a list initialized as `[]`, or transform an existing list into a min-heap or max-heap using the `heapify` or `heapify_max` functions, respectively.

The following functions are provided for min-heaps:

heapify(x)

Transform list *x* into a min-heap, in-place, in linear time.

heappush(heap, item)

Push the value *item* onto the *heap*, maintaining the min-heap invariant.

heappop(heap)

Pop and return the smallest item from the *heap*, maintaining the min-heap invariant. If the heap is empty, `IndexError` is raised. To access the smallest item without popping it, use `heap[0]`.

heappushpop(heap, item)

Push *item* on the heap, then pop and return the smallest item from the *heap*. The combined action runs more efficiently than `heappush` followed by a separate call to `heappop`.

heapreplace(heap, item)

Pop and return the smallest item from the *heap*, and also push the new *item*. The heap size doesn't change. If the heap is empty, `IndexError` is raised.

This one step operation is more efficient than a `heappop` followed by `heappush` and can be more appropriate when using a fixed-size heap. The pop/push combination always returns an element from the heap and replaces it with *item*.

The value returned may be larger than the *item* added. If that isn't desired, consider using `heappushpop` instead. Its push/pop combination returns the smaller of the two values, leaving the larger value on the heap.

For max-heaps, the following functions are provided:

heapify_max(x)

Transform list *x* into a max-heap, in-place, in linear time.

3.14

heappush_max(heap, item)

Push the value *item* onto the max-heap *heap*, maintaining the max-heap invariant.

3.14

heappop_max(heap)

Pop and return the largest item from the max-heap *heap*, maintaining the max-heap invariant. If the max-heap is empty, `IndexError` is raised. To access the largest item without popping it, use `maxheap[0]`.

3.14

heappushpop_max(heap, item)

Push *item* on the max-heap *heap*, then pop and return the largest item from *heap*. The combined action runs more efficiently than `heappush_max` followed by a separate call to `heappop_max`.

3.14

heapreplace_max(heap, item)

Pop and return the largest item from the max-heap *heap* and also push the new *item*. The max-heap size doesn't change. If the max-heap is empty, `IndexError` is raised.

The value returned may be smaller than the *item* added. Refer to the analogous function `heapreplace` for detailed usage notes.

3.14

The module also offers three general purpose functions based on heaps.

merge(*iterables, key=None, reverse=False)

Merge multiple sorted inputs into a single sorted output (for example, merge timestamped entries from multiple log files). Returns an `iterator` over the sorted values.

Similar to `sorted(itertools.chain(*iterables))` but returns an iterable, does not pull the data into memory all at once, and assumes that each of the input streams is already sorted (smallest to largest).

Has two optional arguments which must be specified as keyword arguments.

key specifies a `key function` of one argument that is used to extract a comparison key from each input element. The default value is `None` (compare the elements directly).

reverse is a boolean value. If set to `True`, then the input elements are merged as if each comparison were reversed. To achieve behavior similar to `sorted(itertools.chain(*iterables), reverse=True)`, all iterables must be sorted from largest to smallest.

3.5 Added the optional *key* and *reverse* parameters.

nlargest(n, iterable, key=None)

Return a list with the *n* largest elements from the dataset defined by *iterable*. *key*, if provided, specifies a function of one argument that is used to extract a comparison key from each element in *iterable* (for example, `key=str.lower`). Equivalent to: `sorted(iterable, key=key, reverse=True)[:n]`.

nsmallest(n, iterable, key=None)

Return a list with the *n* smallest elements from the dataset defined by *iterable*. *key*, if provided, specifies a function of one argument that is used to extract a comparison key from each element in *iterable* (for example, `key=str.lower`). Equivalent to: `sorted(iterable, key=key)[:n]`.

The latter two functions perform best for smaller values of *n*. For larger values, it is more efficient to use the `sorted` function. Also, when `n==1`, it is more efficient to use the built-in `min` and `max` functions. If repeated usage of these functions is required, consider turning the iterable into an actual heap.

Basic Examples

A heapsort can be implemented by pushing all values onto a heap and then popping off the smallest values one at a time:

```
>>> def heapsort(iterable):
...     h = []
...     for value in iterable:
...         heappush(h, value)
...     return [heappop(h) for i in range(len(h))]
...
>>> heapsort([1, 3, 5, 7, 9, 2, 4, 6, 8, 0])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

This is similar to `sorted(iterable)`, but unlike `sorted`, this implementation is not stable.

Heap elements can be tuples. This is useful for assigning comparison values (such as task priorities) alongside the main record being tracked:

```
>>> h = []
>>> heappush(h, (5, 'write code'))
>>> heappush(h, (7, 'release product'))
>>> heappush(h, (1, 'write spec'))
>>> heappush(h, (3, 'create tests'))
>>> heappop(h)
(1, 'write spec')
```

Other Applications

Medians are a measure of central tendency for a set of numbers. In distributions skewed by outliers, the median provides a more stable estimate than an average (arithmetic mean). A running median is an online algorithm that updates continuously as new data arrives.

A running median can be efficiently implemented by balancing two heaps, a max-heap for values at or below the midpoint and a min-heap for values above the midpoint. When the two heaps have the same size, the new median is the average of the tops of the two heaps; otherwise, the median is at the top of the larger heap:

```
def running_median(iterable):
    "Yields the cumulative median of values seen so far."

    lo = []  # max-heap
    hi = []  # min-heap (same size as or one smaller than lo)

    for x in iterable:
        if len(lo) == len(hi):
            heappush_max(lo, heappushpop(hi, x))
            yield lo[0]
        else:
            heappush(hi, heappushpop_max(lo, x))
            yield (lo[0] + hi[0]) / 2
```

For example:

```
>>> list(running_median([5.0, 9.0, 4.0, 12.0, 8.0, 9.0]))
[5.0, 7.0, 5.0, 7.0, 8.0, 8.5]
```

Priority Queue Implementation Notes

A priority queue is common use for a heap, and it presents several implementation challenges:

- Sort stability: how do you get two tasks with equal priorities to be returned in the order they were originally added?
- Tuple comparison breaks for (priority, task) pairs if the priorities are equal and the tasks do not have a default comparison order.
- If the priority of a task changes, how do you move it to a new position in the heap?
- Or if a pending task needs to be deleted, how do you find it and remove it from the queue?

A solution to the first two challenges is to store entries as 3-element list including the priority, an entry count, and the task. The entry count serves as a tie-breaker so that two tasks with the same priority are returned in the order they were added. And since no two entry counts are the same, the tuple comparison will never attempt to directly compare two tasks.

Another solution to the problem of non-comparable tasks is to create a wrapper class that ignores the task item and only compares the priority field:

```
from dataclasses import dataclass, field
from typing import Any

@dataclass(order=True)
class PrioritizedItem:
    priority: int
    item: Any=field(compare=False)
```

The remaining challenges revolve around finding a pending task and making changes to its priority or removing it entirely. Finding a task can be done with a dictionary pointing to an entry in the queue.

Removing the entry or changing its priority is more difficult because it would break the heap structure invariants. So, a possible solution is to mark the entry as removed and add a new entry with the revised priority:

```
pq = []                         # list of entries arranged in a heap
entry_finder = {}               # mapping of tasks to entries
REMOVED = '<removed-task>'      # placeholder for a removed task
counter = itertools.count()     # unique sequence count

def add_task(task, priority=0):
    'Add a new task or update the priority of an existing task'
    if task in entry_finder:
        remove_task(task)
    count = next(counter)
    entry = [priority, count, task]
    entry_finder[task] = entry
    heappush(pq, entry)

def remove_task(task):
    'Mark an existing task as REMOVED.  Raise KeyError if not found.'
    entry = entry_finder.pop(task)
    entry[-1] = REMOVED

def pop_task():
    'Remove and return the lowest priority task. Raise KeyError if empty.'
    while pq:
        priority, count, task = heappop(pq)
        if task is not REMOVED:
            del entry_finder[task]
            return task
    raise KeyError('pop from an empty priority queue')
```

Theory

Heaps are arrays for which `a[k] <= a[2*k+1]` and `a[k] <= a[2*k+2]` for all *k*, counting elements from 0. For the sake of comparison, non-existing elements are considered to be infinite. The interesting property of a heap is that `a[0]` is always its smallest element.

The strange invariant above is meant to be an efficient memory representation for a tournament. The numbers below are *k*, not `a[k]`:

Example (min-heap) binary tree.

In the tree above, each cell *k* is topping `2*k+1` and `2*k+2`. In a usual binary tournament we see in sports, each cell is the winner over the two cells it tops, and we can trace the winner down the tree to see all opponents s/he had. However, in many computer applications of such tournaments, we do not need to trace the history of a winner. To be more memory efficient, when a winner is promoted, we try to replace it by something else at a lower level, and the rule becomes that a cell and the two cells it tops contain three different items, but the top cell "wins" over the two topped cells.

If this heap invariant is protected at all time, index 0 is clearly the overall winner. The simplest algorithmic way to remove it and find the "next" winner is to move some loser (let's say cell 30 in the diagram above) into the 0 position, and then percolate this new 0 down the tree, exchanging values, until the invariant is re-established. This is clearly logarithmic on the total number of items in the tree. By iterating over all items, you get an $O(n \log n)$ sort.

A nice feature of this sort is that you can efficiently insert new items while the sort is going on, provided that the inserted items are not "better" than the last 0'th element you extracted. This is especially useful in simulation contexts, where the tree holds all incoming events, and the "win" condition means the smallest scheduled time. When an event schedules other events for execution, they are scheduled into the future, so they can easily go into the heap. So, a heap is a good structure for implementing schedulers (this is what I used for my MIDI sequencer :-).

Various structures for implementing schedulers have been extensively studied, and heaps are good for this, as they are reasonably speedy, the speed is almost constant, and the worst case is not much different than the average case. However, there are other representations which are more efficient overall, yet the worst cases might be terrible.

Heaps are also very useful in big disk sorts. You most probably all know that a big sort implies producing "runs" (which are pre-sorted sequences, whose size is usually related to the amount of CPU memory), followed by a merging passes for these runs, which merging is often very cleverly organised[^1]. It is very important that the initial sort produces the longest runs possible. Tournaments are a good way to achieve that. If, using all the memory available to hold a tournament, you replace and percolate items that happen to fit the current run, you'll produce runs which are twice the size of the memory for random input, and much better for input fuzzily ordered.

Moreover, if you output the 0'th item on disk and get an input which may not fit in the current tournament (because the value "wins" over the last output value), it cannot fit in the heap, so the size of the heap decreases. The freed memory could be cleverly reused immediately for progressively building a second heap, which grows at exactly the same rate the first heap is melting. When the first heap completely vanishes, you switch heaps and start a new run. Clever and quite effective!

In a word, heaps are useful memory structures to know. I use them in a few applications, and I think it is good to keep a 'heap' module around. :-)

Footnotes

[^1]: The disk balancing algorithms which are current, nowadays, are more annoying than clever, and this is a consequence of the seeking capabilities of the disks. On devices which cannot seek, like big tape drives, the story was quite different, and one had to be very clever to ensure (far in advance) that each tape movement will be the most effective possible (that is, will best participate at "progressing" the merge). Some tapes were even able to read backwards, and this was also used to avoid the rewinding time. Believe me, real good tape sorts were quite spectacular to watch! From all times, sorting has always been a Great Art! :-)

□□□ #2
□ □□□ □□□ 2026-10-31 17:11:34 UTC
□ □□□ □□□ 2026-10-31 17:11:36 UTC