1.5 实现一个优先级队列¶

问题¶

怎样实现一个按优先级排序的队列？ 并且在这个队列上面每次 pop 操作总是返回优先级最高的那个元素

解决方案¶

下面的类利用 `heapq` 模块实现了一个简单的优先级队列：

```
import heapq

class PriorityQueue:
    def __init__(self):
        self._queue = []
        self._index = 0

    def push(self, item, priority):
        heapq.heappush(self._queue, (-priority, self._index, item))
        self._index += 1

    def pop(self):
        return heapq.heappop(self._queue)[-1]
```

下面是它的使用方式：

```
>>> class Item:
...     def __init__(self, name):
...         self.name = name
...     def __repr__(self):
...         return 'Item({!r})'.format(self.name)
...
>>> q = PriorityQueue()
>>> q.push(Item('foo'), 1)
>>> q.push(Item('bar'), 5)
>>> q.push(Item('spam'), 4)
>>> q.push(Item('grok'), 1)
>>> q.pop()
Item('bar')
>>> q.pop()
Item('spam')
>>> q.pop()
Item('foo')
>>> q.pop()
Item('grok')
>>>
```

仔细观察可以发现，第一个 `pop()` 操作返回优先级最高的元素。 另外注意到如果两个有着相同优先级的元素（ `foo` 和 `grok` ），pop 操作按照它们被插入到队列的顺序返回的。

讨论¶

这一小节我们主要关注 `heapq` 模块的使用。 函数 `heapq.heappush()` 和 `heapq.heappop()` 分别在队列 `_queue` 上插入和

删除第一个元素， 并且队列 `_queue` 保证第一个元素拥有最高优先级（ 1.4 节已经讨论过这个问题）。 `heappop()` 函数总是返回”最小的”的元素，这就是保证队列pop操作返回正确元素的关键。 另外，由于 push 和 pop 操作时间复杂度为 O(log N)，其中 N 是堆的大小，因此就算是 N 很大的时候它们运行速度也依旧很快。

在上面代码中，队列包含了一个 `(-priority, index, item)` 的元组。 优先级为负数的目的是使得元素按照优先级从高到低排序。 这个跟普通的按优先级从低到高排序的堆排序恰巧相反。

`index` 变量的作用是保证同等优先级元素的正确排序。 通过保存一个不断增加的 `index` 下标变量，可以确保元素按照它们插入的顺序排序。 而且， `index` 变量也在相同优先级元素比较的时候起到重要作用。

为了阐明这些，先假定 `Item` 实例是不支持排序的：

```
>>> a = Item('foo')
>>> b = Item('bar')
>>> a < b
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unorderable types: Item() < Item()
>>>
```

如果你使用元组 `(priority, item)` ，只要两个元素的优先级不同就能比较。 但是如果两个元素优先级一样的话，那么比较操作就会跟之前一样出错：

```
>>> a = (1, Item('foo'))
>>> b = (5, Item('bar'))
>>> a < b
True
>>> c = (1, Item('grok'))
>>> a < c
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unorderable types: Item() < Item()
>>>
```

通过引入另外的 `index` 变量组成三元组 `(priority, index, item)` ，就能很好的避免上面的错误， 因为不可能有两个元素有相同的 `index` 值。Python 在做元组比较时候，如果前面的比较已经可以确定结果了， 后面的比较操作就不会发生了：

```
>>> a = (1, 0, Item('foo'))
>>> b = (5, 1, Item('bar'))
>>> c = (1, 2, Item('grok'))
>>> a < b
True
>>> a < c
True
>>>
```

如果你想在多个线程中使用同一个队列，那么你需要增加适当的锁和信号量机制。 可以查看 12.3 小节的例子演示是怎样做的。

`heapq` 模块的官方文档有更详细的例子程序以及对于堆理论及其实现的详细说明。