TechFlow2019

Python中heapq与优先队列【详细】

本文始发于个人公众号：**TechFlow**, 原创不易，求个关注

今天的文章来介绍Python当中一个蛮有用的库——**heapq**。

heapq的全写是heap queue，是堆队列的意思。这里的**堆和队列**都是数据结构，在后序的文章当中我们会详细介绍，今天只介绍heapq的用法，如果不了解heap和queue原理的同学可以忽略，我们并不会深入太多，会在之后的文章里详细阐述。

在介绍用法之前，我们需要先知道优先队列的定义。**队列**大家应该都不陌生，也是非常基础简单的数据结构。我们可以想象成队列里的所有元素排成一排，新的元素只能从队尾加入队列，元素要出队列只能通过队首，**不能中途从队列当中退出**。而优先队列呢，是给队列当中的元素每一个都设置了优先级，使得队伍当中的元素会**自动按照优先级排序**，优先级高的排在前面。

也就是说Python当中的heapq就是一个维护优先队列的library，我们通过调用它可以轻松实现优先队列的功能。

最大或最小的K个元素

我们来看一个实际的问题，假设我们当下有N个杂乱无章的元素，但是我们只关心其中最大的K个或者是最小的K个元素。我们想从整个数组当中将这部分抽取出来，应该怎么办呢？

这个问题在实际当中非常常见，随便就可以举出例子来。比如用户输入了搜索词，我们根据用户的搜索词找到了大量的内容。我们想要根据算法筛选出用户最有可能点击的文本来，机器学习的模型可以给每一个文本一个预测的分数。之后，我们就需要**选出分数最大的K个结果**。这种类似的场景还有很多，利用heapq库里的nlargest和nsmallest接口可以非常方便地做到这点。

我们一起来看一个例子：

```
import heapq

nums = [14, 20, 5, 28, 1, 21, 16, 22, 17, 28]
heapq.nlargest(3, nums)
# [28, 28, 22]
heapq.nsmallest(3, nums)
# [1, 5, 14]
```

heapq的nlargest和nsmallest接受两个参数，第一个参数是K，也就是返回的元素的数量，第二个参数是传入的数组，heapq返回的正是传入的数组当中的前K大或者是前K小。

这里有一个问题，如果我们数组当中的元素是一个对象呢？应该怎么办？

其实也很简单，有了解过Python自定义关键词排序的同学应该知道，和排序一样，我们可以通过**匿名函数**实现。

匿名函数

我们都知道，在Python当中通过def可以定义一个函数。通过def定义的函数都有函数名，所以称为**有名函数**。除了有名函数之外，Python还支持匿名函数。顾名思义，就是没有函数名的函数。也就是说它其他方面都和普通函数一样，只不过没有名字而已。

初学者可能会纳闷，**函数没有名字应该怎么调用呢**?

会有这个疑惑很正常，这是因为习惯了面向过程的编程，对面向对象理解不够深入导致的。在许多高级语言当中，**一切皆对象**，一个类，一个函数，一个int都是对象。既然函数也是对象，那么函数自然也可以用来传递，不仅可以用来传递，还可以用来返回。这是函数式编程的概念了，我们这里不多做深入。

当然，普通函数也一样可以传递，起到的效果一样。只不过在编程当中，有些函数我们只会使用一次，**没必要再单独定义一个函数**，使用匿名函数会非常方便。

举个例子，比方说我有一个这样的函数：

```
def operate(x, func):
  return func(x)
```

这个operate函数它接受两个参数，第一个参数是变量x，第二个**参数是一个函数。**它会在函数内部调用func，返回func调用的结果。我现在要做这样一件事情，我希望根据x这个整数对4取余的余数来判断应该用什么样的func。如果对4的余数为0，我希望求一次方，如果余数是2，我希望求平方，以此类推。如果按照正常的方法，我们需要实现4个方法，然后依次传递。

这当然是可以的，不过非常麻烦，如果使用匿名函数，就可以**大大简化**代码量：

```
def get_result(x):
  if x % 4 == 0:
    return operate(x, lambda x: x)
  elif x % 4 == 1:
    return operate(x, lambda x: x ** 2)
  elif x % 4 == 2:
    return operate(x, lambda x: x ** 3)
  else:
    return operate(x, lambda x: x ** 4)
```

在上面的代码当中，**我们通过lambda关键字定义了匿名函数**，避免了定义四种函数用来传递的情况。当然，这个问题还有更简单的写法，可以只用一个函数解决。

我们来看lambda定义匿名函数的语法，首先是lambda关键字，表示我们当下定义的是一个匿名函数。之后跟的是这个匿名函数的参数，我们只用到一个变量x，所以只需要写一个x。如果我们需要用到多个参数，通过逗号分隔，当然也可以不用参数。写完参数之后，我们用冒号分开，冒号后面写的是返回的结果。

我们也可以把匿名函数赋值给一个变量，之后我们就可以和调用普通函数一样来调用了：

```
square = lambda x: x ** 2

print(square(3))
print(operate(3, square))
```

自定义排序

回到之前的内容，如果我们想要heapq排序的是一个对象。那么heapq并不知道应该依据对象当中的哪个参数来作为排序的衡量标准，所以这个时候，需要我们自己**定义一个获取关键字的函数**，传递给heapq，这样才可以完成排序。

比如说，我们现在有一批电脑，我们希望heapq能够根据电脑的价格排序：

```
laptops = [
    {'name': 'ThinkPad', 'amount': 100, 'price': 91.1},
    {'name': 'Mac', 'amount': 50, 'price': 543.22},
    {'name': 'Surface', 'amount': 200, 'price': 21.09},
    {'name': 'Alienware', 'amount': 35, 'price': 31.75},
    {'name': 'Lenovo', 'amount': 45, 'price': 16.35},
    {'name': 'Huawei', 'amount': 75, 'price': 115.65}
]

cheap = heapq.nsmallest(3, portfolio, key=lambda s: s['price'])
expensive = heapq.nlargest(3, portfolio, key=lambda s: s['price'])
```

在调用nlargest和nsmallest的时候，我们额外传递了一个参数key，我们传入的是一个匿名函数，它返回的结果是这个对象的price，也就是说我们希望heapq根据对象的price来进行排序。

优先队列

heapq除了可以返回最大最小的K个数之外，还实现了优先队列的接口。我们可以直接调用heapq.heapify方法，输入一个数组，返回的结果是根据这个数组生成的堆（等价于优先队列）。

当然我们也可以从零开始，直接通过调用heapq的push和pop来维护这个堆。接下来，我们就通过heapq来自己动手实现一个优先队列，代码非常的简单，我想大家应该可以**瞬间学会**。

首先是实现优先队列的部分：

```
import heapq

class PriorityQueue:

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

  def push(self, item, priority):
    # 传入两个参数，一个是存放元素的数组，另一个是要存储的元素，这里是一个元组。
    # 由于heap内部默认有小到大排，所以对priority取负数
    heapq.heappush(self._queue, (-priority, self._index, item))
    self._index += 1

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

其次我们来实际看一下运用的情况：

```
q = PriorityQueue()

q.push('lenovo', 1)
q.push('Mac', 5)
q.push('ThinkPad', 2)
q.push('Surface', 3)

q.pop()
# Mac
q.pop()
# Surface
```

到这里，关于heapq的应用方面就算是介绍完了，但是还没有真正的结束。

我们需要分析一下heapq当中操作的复杂度，关于堆的部分我们暂时跳过，我们先来看nlargest和nsmallest。我在github当中找到了这个库的**源码**，在方法的注释上，作者写下了这个方法的复杂度，和**排序之后取前K个开销五五开**：

```
def nlargest(n, iterable, key=None):
    """Find the n largest elements in a dataset.

    Equivalent to:  sorted(iterable, key=key, reverse=True)[:n]
    """
```

我们都知道排序的复杂度的期望是$O(nlogn)$，如果你了解堆的话，会知道堆一次插入元素的复杂度是$logn$。如果我们限定堆的长度是K，我们插入n次之后也只能保留K个元素。每次插入的复杂度是$logK$，一共插入n次，所以整体的复杂度是$nlogK$。

如果K小一些，可能开销会比排序稍小，但是程度有限。那么有没有什么办法可以不用排序并且尽可能快地筛选出前K大或者是前K小的元素呢？

我这里先卖个关子，我们之后的文章当中再来讲解。

今天的文章就到这里，如果觉得有所收获，请顺手点个关注吧，你的举手之劳对我很重要。

> 参考资料
>
> Python CookBook Version3
>
> 维基百科

标签: 优先队列 , 数据结构 , python