16 最佳实践：灵活运用内建容器，提高开发效率-慕课专栏

本节以若干场景抛砖引玉，介绍如何选择内建容器对象，高效解决问题。

列表排序

我们经常需要对数据进行 **排序** ，例如制作一个积分榜。假设用户数据由 *list* 对象保存，数据项字段如下：

- *name* ，名字；
- *sex* ，性别；
- *area* ，区域；
- *score*，积分；

```
users = [{
    'name': 'jim',
    'sex': MALE,
    'city': 'guangzhou',
    'score': 28000,
}, {
    'name': 'lily',
    'sex': FEMALE,
    'city': 'shenzhen',
    'score': 25000,
}, ...]
```

为得到积分榜，我们需要对列表数据进行排序。这在 *Python* 中只是小菜一碟，*list* 对象 *sort* 方法即可胜任：

```
users.sort(key=lambda user: user['score'])
```

由于数据项类型 *dict* 不是 **可比较** 的，我们需要提供 *key* 函数，为每个数据项生成可比较的排序 *key* 。排序 *key* 可以是单字段，也可以复合字段，还可以是数据项的某种运算结果。

这样一来，*sort* 函数便按照 *score* 字段对数据进行排序，积分榜也就诞生了！

可比较对象

此外，我们可以用一个自定义类来抽象用户信息，并实现 *__eq__* 系列比较方法。借助 *total_ordering* 装饰器，我们只需要实现 *__eq__* 和 *__lt__* 两个比较方法，其他诸如 *__gt__* 等均由 *total_ordering* 自动生成：

```
from functools import total_ordering

@total_ordering
class User:

    def __init__(self, name, sex, area, score):
        self.name = name
        self.sex = sex
        self.area = area
        self.score = score

    def __eq__(self, other):
        return self.score == other.score

    def __lt__(self, other):
        return self.score < other.score
```

这样一来，按积分排序 *User* 列表时，便不再需要提供 *key* 函数了：

```
users = [
    User(name=a1, age=a2, area=a3, score=a4),

    User(name=z1, age=z2, area=z3, score=z4),
]

users.sort()
```

比较函数只能实现一种排序标准，我们的 *User* 类默认只能按积分，即 *score* 字段排序。如果想按 *name* 字段进行排序，则可以提供 key 辅助函数：

```
users.sort(key=lambda user: user.name)
```

由此可见，虽然 *key* 辅助函数应用起来要麻烦些，但 *灵活性* 更胜一筹。

排行榜

如果我们想将排名前 *100* 位的用户制作成积分榜，可以先按积分 *降序排序* ，再取出前 100 个：

```
users.sort(reverse=True)
top100 = users[:100]
```

注意到，指定 *reverse* 参数为 *True* ，*sort* 方法便按降序排序。

sort 方法排序的时间复杂度为 O(NlogN)O(NlogN)O(NlogN)，如果基数 NNN 很大，计算开销肯定也不小。由于积分榜只关心前 100 位用户，似乎没有必要对整个列表进行排序。这个场景非常典型，可以用最小堆来解决：

```
from heapq import heappush, heappop


top100 = []


for user in users:

    if len(top100) < 100:
        heappush(top100, user)
        continue


    if user > top100[0]:
        heappop(top100)
        heappush(top100, user)
```

引入最小堆后，制作排行榜的时间复杂度降为 O(Nlog100)O(Nlog100)O(Nlog100)。**由于** O(Nlog100)O(Nlog100)O(Nlog100)**是个常数，因此时间复杂度等价于**O(N)O(N)O(N)**更一般地，假设为**NNN**位用户制作长度为**KKK**的排行榜，两个方案的的时间复杂度分别是：**

- **全量排序：**O(NlogN)O(NlogN)O(NlogN)**；**
- **最小堆：**O(NlogK)O(NlogK)O(NlogK)**；**

由于一般情况下，KKK **远小于，**NNN**因此采用最小堆性能更理想。**

列表推导

假设我们需要将一个用 *dict* **表示的用户信息列表转换成一个用** *User* **类表示的新列表，可以这样做：**

```
new_users = []

for user in users:
    new_users.append(User(**user))
```

1. **第** *1* **行，先创建一个新列表；**
2. **第** *3-4* **行，遍历原列表每一项，将其转换成** *User* **对象并加入新列表；**

思路平白无奇，但我们可以用 列表推导 语法对代码进行优化：

```
new_users = [User(**user) for user in users]
```

这段代码比上一段代码更简洁，也更易读。新列表由原列表 *users* **生成而来：遍历** *users* **每个数据项，以原数据项为参数实例化** *User* **对象作为新列表的数据项。**

列表过滤

列表推导支持通过 *if* 关键字过滤部分符合指定条件的数据项。例如，从用户列表中过滤出所有男性：

```
new_users = [
    user
    for user in users
    if user.sex == MALE
]
```

这个列表推导遍历原列表 *users* 中每个数据项，如果性别为男性则填充到新列表 *new_users* 。

此外，我们还可以借助 *filter* 内建函数进行过滤：

```
new_users = list(
    filter(lambda user: user.sex == MALE, users),
)
```

filter 函数接收两个参数：

- *function* ，判定函数，以数据项为参数，返回过滤结果，*True* 或者 *False* ；
- *iterable* ，可迭代对象；

抽象运算

我们刚刚提到了 *filter* 内建函数，它只是众多 **抽象运算** 操作中的一员。

数学是一门美丽的科学，为刻画复杂的现实世界，提供高度抽象的思维和工具。例如，很多大数据处理框架，如 *Hadoop* 等，将计算任务归纳为 *map* 和 *reduce* 两种基本操作的组合。

接下来，我们以一个简单例子，讲解以下几个操作的含义及应用：

- *filter* ，过滤操作；
- *map* ，映射操作；
- *reduce* ，聚合操作；

我们的研究对象是一个整数列表：

```
>>> nums = list(range(1,10))
>>> nums
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

filter 函数将符合条件的对象从对象集中过滤出来，条件以 **判定函数** 的形式指定。判定函数以具体对象为参数，返回一个真值，以表明该对象是否符合条件。

假设我们需要找出整数中的偶数，判定函数可以这么写：

```
>>> list(filter(lambda x: x%2 == 0, nums))
[2, 4, 6, 8]
```

对于整数 *x* ，如果 *x* 除以 *2* 余数为 *0* (偶数)，则返回 *True* 表示符合条件。

map 函数对集合中每个对象进行加工，将其映射成一个新对象。加工方法由操作函数指定，操作函数以待加工对象为参数，计算并返回加工结果。例如，计算列表每个整数的平方值：

```
>>> list(map(lambda x: x**2, nums))
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

reduce 函数对多个对象进行汇总，并生成结果。汇聚方法由操作函数指定，操作函数接收两个对象，并计算并返回这两个对象的合并结果。例如，对所有整数进行求和：

```
>>> from functools import reduce
>>> reduce(lambda x, y: x+y, nums)
45
```

对于一个长度为 NNN 的整数列表，操作函数需要调用 O(N−1)O(N-1)O(N−1) 次。

你可能会说，整数求和用 *sum* 内建函数即可，无须大动干戈。的确如此。但 reduce 可以胜任更复杂的汇总操作，比如乘积计算：

```
>>> reduce(lambda x, y: x*y, nums)
362880
```

字典默认值

如果现在需要根据城市对用户进行分组，我们可以用一个 *dict* 对象来维护城市与对应用户列表的映射关系：

```
city2users = {}

for user in users:
    city = user.city

    if city not in city2users:
        city2users[city] = []

    city2users[city].append(user)
```

这段代码毫无难度，但还有优化空间。第 *10-11* 行，需要先判断城市是否已经在映射表内，并在必要时为其创建一个空的 *list* 对象，用以保存该城市下所有用户。借助 *setdefault* 方法，我们可以将代码写得更简洁、漂亮：

```
city2users = {}

for user in users:
    city = user.city
    city2users.setdefault(city, []).append(user)
```

setdefault 方法先检查给定键是否已存在，未存在则以第二个参数进行初始化，最后返回与键关联的值。

defaultdict

标准库 *collections* 模块中的 *defaultdict* ，完美解决字典默认值问题。*defaultdict* 接收一个 *default_factory* 参数，当访问到不存在的键时，*defaultdict* 调用 *default_factory* 为其提供一个初始值。

```
>>> from collections import defaultdict


>>> d = defaultdict(list)
>>> d
defaultdict(<class 'list'>, {})



>>> d['foo']
[]
>>> d
defaultdict(<class 'list'>, {'foo': []})
```

这样一来，按城市分组用户的功能可以这么来写：

```
city2users = defaultdict(list)

for user in users:
    city2users[user.city].append(user)
```

树结构

借助 *defaultdict* ，我们只需一行代码便可实现一个树形存储容器：

```
>>> Tree = lambda: defaultdict(Tree)
```

这行代码的巧妙之处在于 **递归** 。*Tree* 函数初始化一棵树，树的实际结构是一个 *defaultdict* 对象。当我们访问一个不存在的树分支时，*defaultdict* 再次调用 *Tree* 函数完成子树的初始化！

我们接着看看 *Tree* 如何应用：

```
>>> tree = Tree()


>>> tree['fruits']['apple'] = 10
>>> tree['fruits']['pear'] = 20
>>> tree['pets']['cat'] = 3
>>> tree['pets']['dog'] = 1


>>> tree.keys()
dict_keys(['fruits', 'pets'])


>>> tree['fruits'].items()
dict_items([('apple', 10), ('pear', 20)])
```

Counter

如果我们只需统计每个城市的用户数，可以使用 *int* 对象作为 *defaultdict* 的默认值：

```
city2total = defaultdict(int)

for user in users:
    city2total[user.city] += 1
```

数量统计是一个非常常见的场景，为此 *collections* 模块提供了一个更趁手的解决方案——*Counter* 类：

```
from collections import Counter

city2total = Counter()

for user in users:
    city2total[user.city] += 1
```

***Counter* 还有很多其他高级特性，由于篇幅关系就不再深入讲解了，有兴趣的同学请自行查阅 *Python* 文档。**