04-字典、集合，你真的了解吗？

你好，我是景霄。

前面的课程，我们学习了Python中的列表和元组，了解了他们的基本操作和性能比较。这节课，我们再来学习两个同样很常见并且很有用的数据结构：字典（dict）和集合（set）。字典和集合在Python被广泛使用，并且性能进行了高度优化，其重要性不言而喻。

字典和集合基础

那究竟什么是字典，什么是集合呢？字典是一系列无序元素的组合，其长度大小可变，元素可以任意地删减和改变。不过要注意，这里的元素，是一对键（key）和值（value）的配对。

相比于列表和元组，字典的性能更优，特别是对于查找、添加和删除操作，字典都能在常数时间复杂度内完成。

而集合和字典基本相同，唯一的区别，就是集合没有键和值的配对，是一系列无序的、唯一的元素组合。

首先我们来看字典和集合的创建，通常有下面这几种方式：

```
d1 = {'name': 'jason', 'age': 20, 'gender': 'male'}
d2 = dict({'name': 'jason', 'age': 20, 'gender': 'male'})
d3 = dict([('name', 'jason'), ('age', 20), ('gender', 'male')])
d4 = dict(name='jason', age=20, gender='male')
d1 == d2 == d3 ==d4
True

s1 = {1, 2, 3}
s2 = Set([1, 2, 3])
s1 == s2
True
```

这里注意，Python中字典和集合，无论是键还是值，都可以是混合类型。比如下面这个例子，我创建了一个元素为1，'hello'，5.0的集合：

```
s = {1, 'hello', 5.0}
```

再来看元素访问的问题。字典访问可以直接索引键，如果不存在，就会抛出异常：

```
d = {'name': 'jason', 'age': 20}
d['name']
'jason'
d['location']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'location'
```

也可以使用get(key, default)函数来进行索引。如果键不存在，调用get()函数可以返回一个默认值。比如下面这个示例，返回了'null'。

```
d = {'name': 'jason', 'age': 20}
d.get('name')
'jason'
d.get('location', 'null')
'null'
```

说完了字典的访问，我们再来看集合。

首先我要强调的是，**集合并不支持索引操作，因为集合本质上是一个哈希表，和列表不一样。**所以，下面这样的操作是错误的，Python会抛出异常：

```
s = {1, 2, 3}
s[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object does not support indexing
```

想要判断一个元素在不在字典或集合内，我们可以用value in dict/set 来判断。

```
s = {1, 2, 3}
1 in s
True
10 in s
False

d = {'name': 'jason', 'age': 20}
'name' in d
True
'location' in d
False
```

当然，除了创建和访问，字典和集合也同样支持增加、删除、更新等操作。

```
d = {'name': 'jason', 'age': 20}
d['gender'] = 'male' # 增加元素对'gender': 'male'
d['dob'] = '1999-02-01' # 增加元素对'dob': '1999-02-01'
d
{'name': 'jason', 'age': 20, 'gender': 'male', 'dob': '1999-02-01'}
d['dob'] = '1998-01-01' # 更新键'dob'对应的值
d.pop('dob') # 删除键为'dob'的元素对
```

```
'1998-01-01'
d
{'name': 'jason', 'age': 20, 'gender': 'male'}

s = {1, 2, 3}
s.add(4) # 增加元素4到集合
s
{1, 2, 3, 4}
s.remove(4) # 从集合中删除元素4
s
{1, 2, 3}
```

不过要注意，集合的pop()操作是删除集合中最后一个元素，可是集合本身是无序的，你无法知道会删除哪个元素，因此这个操作得谨慎使用。

实际应用中，很多情况下，我们需要对字典或集合进行排序，比如，取出值最大的50对。

对于字典，我们通常会根据键或值，进行升序或降序排序：

```
d = {'b': 1, 'a': 2, 'c': 10}
d_sorted_by_key = sorted(d.items(), key=lambda x: x[0]) # 根据字典键的升序排序
d_sorted_by_value = sorted(d.items(), key=lambda x: x[1]) # 根据字典值的升序排序
d_sorted_by_key
[('a', 2), ('b', 1), ('c', 10)]
d_sorted_by_value
[('b', 1), ('a', 2), ('c', 10)]
```

当然，因为字典本身是无序的，所以这里返回了一个列表。列表中的每个元素，是由原字典的键和值组成的元组。

而对于集合，其排序和前面讲过的列表、元组很类似，直接调用sorted(set)即可，结果会返回一个排好序的列表。

```
s = {3, 4, 2, 1}
sorted(s) # 对集合的元素进行升序排序
[1, 2, 3, 4]
```

字典和集合性能

文章开头我就说到了，字典和集合是进行过性能高度优化的数据结构，特别是对于查找、添加和删除操作。那接下来，我们就来看看，它们在具体场景下的性能表现，以及与列表等其他数据结构的对比。

比如电商企业的后台，存储了每件产品的ID、名称和价格。现在的需求是，给定某件商品的ID，我们要找出其价格。

如果我们用列表来存储这些数据结构，并进行查找，相应的代码如下：

```
def find_product_price(products, product_id):
    for id, price in products:
        if id == product_id:
            return price
    return None

products = [
    (143121312, 100),
    (432314553, 30),
    (32421912367, 150)
]

print('The price of product 432314553 is {}'.format(find_product_price(products, 432314553)))

# 输出
The price of product 432314553 is 30
```

假设列表有n个元素，而查找的过程要遍历列表，那么时间复杂度就为O(n)。即使我们先对列表进行排序，然后使用二分查找，也会需要O(logn)的时间复杂度，更何况，列表的排序还需要O(nlogn)的时间。

但如果我们用字典来存储这些数据，那么查找就会非常便捷高效，只需O(1)的时间复杂度就可以完成。原因也很简单，刚刚提到过的，字典的内部组成是一张哈希表，你可以直接通过键的哈希值，找到其对应的值。

```
products = {
  143121312: 100,
  432314553: 30,
  32421912367: 150
}
print('The price of product 432314553 is {}'.format(products[432314553]))

# 输出
The price of product 432314553 is 30
```

类似的，现在需求变成，要找出这些商品有多少种不同的价格。我们还用同样的方法来比较一下。

如果还是选择使用列表，对应的代码如下，其中，A和B是两层循环。同样假设原始列表有n个元素，那么，在最差情况下，需要O(n^2)的时间复杂度。

```
# list version
def find_unique_price_using_list(products):
    unique_price_list = []
    for _, price in products: # A
        if price not in unique_price_list: #B
            unique_price_list.append(price)
    return len(unique_price_list)

products = [
    (143121312, 100),
    (432314553, 30),
    (32421912367, 150),
```

```
    (937153201, 30)
]
print('number of unique price is: {}'.format(find_unique_price_using_list(products)))

# 输出
number of unique price is: 3
```

但如果我们选择使用集合这个数据结构，由于集合是高度优化的哈希表，里面元素不能重复，并且其添加和查找操作只需O(1)的复杂度，那么，总的时间复杂度就只有O(1)。

```
# set version
def find_unique_price_using_set(products):
    unique_price_set = set()
    for _, price in products:
        unique_price_set.add(price)
    return len(unique_price_set)

products = [
    (143121312, 100),
    (432314553, 30),
    (32421912367, 150),
    (937153201, 30)
]
print('number of unique price is: {}'.format(find_unique_price_using_set(products)))

# 输出
number of unique price is: 3
```

可能你对这些时间复杂度没有直观的认识，我可以举一个实际工作场景中的例子，让你来感受一下。

下面的代码，初始化了含有100,000个元素的产品，并分别计算了使用列表和集合来统计产品价格数量的运行时间：

```
import time
id = [x for x in range(0, 100000)]
price = [x for x in range(200000, 300000)]
products = list(zip(id, price))

# 计算列表版本的时间
start_using_list = time.perf_counter()
find_unique_price_using_list(products)
end_using_list = time.perf_counter()
print("time elapse using list: {}".format(end_using_list - start_using_list))
## 输出
time elapse using list: 41.61519479751587

# 计算集合版本的时间
start_using_set = time.perf_counter()
find_unique_price_using_set(products)
end_using_set = time.perf_counter()
print("time elapse using set: {}".format(end_using_set - start_using_set))
# 输出
time elapse using set: 0.008238077163696289
```

你可以看到，仅仅十万的数据量，两者的速度差异就如此之大。事实上，大型企业的后台数据往往有上亿乃至十亿数量级，如果使用了不合适的数据结构，就很容易造成服务器的崩溃，不但影响用户体验，并且会给公司带来巨大的财产损失。

字典和集合的工作原理

我们通过举例以及与列表的对比，看到了字典和集合操作的高效性。不过，字典和集合为什么能够如此高效，特别是查找、插入和删除操作？

这当然和字典、集合内部的数据结构密不可分。不同于其他数据结构，字典和集合的内部结构都是一张哈希表。

- 对于字典而言，这张表存储了哈希值（hash）、键和值这3个元素。
- 而对集合来说，区别就是哈希表内没有键和值的配对，只有单一的元素了。

我们来看，老版本Python的哈希表结构如下所示：

```
--+-------------------------------+
  | 哈希值(hash)  键(key)  值(value)
--+-------------------------------+
0 |    hash0      key0    value0
--+-------------------------------+
1 |    hash1      key1    value1
--+-------------------------------+
2 |    hash2      key2    value2
--+-------------------------------+
. |           ...
__+_____________________________________+
```

不难想象，随着哈希表的扩张，它会变得越来越稀疏。举个例子，比如我有这样一个字典：

```
{'name': 'mike', 'dob': '1999-01-01', 'gender': 'male'}
```

那么它会存储为类似下面的形式：

```
entries = [
['--', '--', '--']
[-230273521, 'dob', '1999-01-01'],
['--', '--', '--'],
['--', '--', '--'],
[1231236123, 'name', 'mike'],
['--', '--', '--'],
[9371539127, 'gender', 'male']
```

```
]
```

这样的设计结构显然非常浪费存储空间。为了提高存储空间的利用率，现在的哈希表除了字典本身的结构，会把索引和哈希值、键、值单独分开，也就是下面这样新的结构：

```
Indices
-----------------------------------------------------
None | index | None | None | index | None | index ...
-----------------------------------------------------

Entries
--------------------
hash0   key0  value0
---------------------
hash1   key1  value1
---------------------
hash2   key2  value2
---------------------
        ...
---------------------
```

那么，刚刚的这个例子，在新的哈希表结构下的存储形式，就会变成下面这样：

```
indices = [None, 1, None, None, 0, None, 2]
entries = [
[1231236123, 'name', 'mike'],
[-230273521, 'dob', '1999-01-01'],
[9371539127, 'gender', 'male']
]
```

我们可以很清晰地看到，空间利用率得到很大的提高。

清楚了具体的设计结构，我们接着来看这几个操作的工作原理。

插入操作

每次向字典或集合插入一个元素时，Python会首先计算键的哈希值（hash(key)），再和 mask = PyDicMinSize - 1做与操作，计算这个元素应该插入哈希表的位置index = hash(key) & mask。如果哈希表中此位置是空的，那么这个元素就会被插入其中。

而如果此位置已被占用，Python便会比较两个元素的哈希值和键是否相等。

- 若两者都相等，则表明这个元素已经存在，如果值不同，则更新值。
- 若两者中有一个不相等，这种情况我们通常称为哈希冲突（hash collision），意思是两个元素的键不相等，但是哈希值相等。这种情况下，Python便会继续寻找表中空余的位置，直到找到位置为止。

值得一提的是，通常来说，遇到这种情况，最简单的方式是线性寻找，即从这个位置开始，挨个往后寻找空位。当然，Python内部对此进行了优化（这一点无需深入了解，你有兴趣可以查看源码，我就不再赘述），让这个步骤更加高效。

查找操作

和前面的插入操作类似，Python会根据哈希值，找到其应该处于的位置；然后，比较哈希表这个位置中元素的哈希值和键，与需要查找的元素是否相等。如果相等，则直接返回；如果不等，则继续查找，直到找到空位或者抛出异常为止。

删除操作

对于删除操作，Python会暂时对这个位置的元素，赋于一个特殊的值，等到重新调整哈希表的大小时，再将其删除。

不难理解，哈希冲突的发生，往往会降低字典和集合操作的速度。因此，为了保证其高效性，字典和集合内的哈希表，通常会保证其至少留有1/3的剩余空间。随着元素的不停插入，当剩余空间小于1/3时，Python会重新获取更大的内存空间，扩充哈希表。不过，这种情况下，表内所有的元素位置都会被重新排放。

虽然哈希冲突和哈希表大小的调整，都会导致速度减缓，但是这种情况发生的次数极少。所以，平均情况下，这仍能保证插入、查找和删除的时间复杂度为O(1)。

总结

这节课，我们一起学习了字典和集合的基本操作，并对它们的高性能和内部存储结构进行了讲解。

字典和集合都是无序的数据结构，其内部的哈希表存储结构，保证了其查找、插入、删除操作的高效性。所以，字典和集合通常运用在对元素的高效查找、去重等场景。

思考题

1. 下面初始化字典的方式，哪一种更高效？

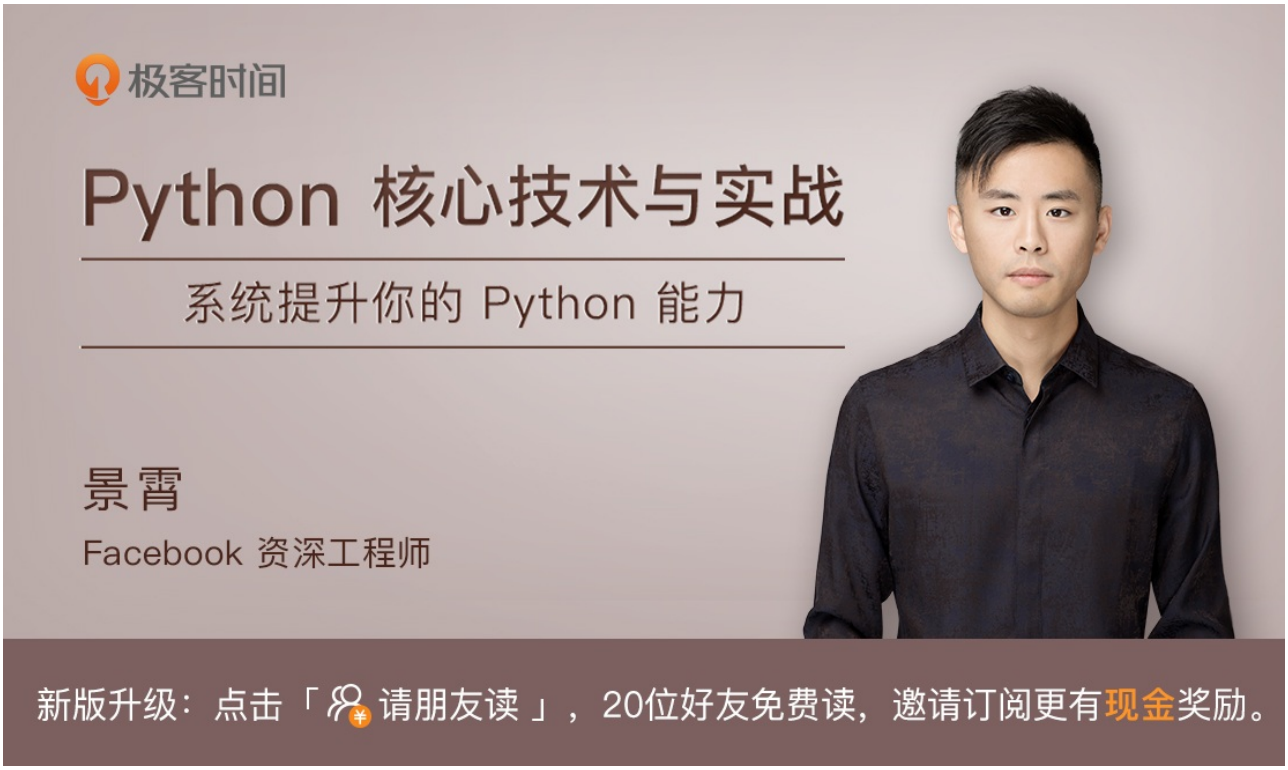
```
# Option A
d = {'name': 'jason', 'age': 20, 'gender': 'male'}

# Option B
d = dict({'name': 'jason', 'age': 20, 'gender': 'male'})
```

2. 字典的键可以是一个列表吗？下面这段代码中，字典的初始化是否正确呢？如果不正确，可以说出你的原因吗？

```
d = {'name': 'jason', ['education']: ['Tsinghua University', 'Stanford University']}
```

欢迎留言和我分享，也欢迎你把这篇文章分享给你的同事、朋友。



精选留言：

- 燕儿衔泥 2019-05-17 06:05:36
  1.直接 { } 的方式，更高效。可以使用dis分析其字节码
  2.字典的键值，需要不可变，而列表是动态的，可变的。可以改为元组 [3赞]

- 随风の 2019-05-17 08:18:00
  文中提到的新的哈希表结构有点不太明白 None 1 None None 0 None 2 是什么意思？ index是索引的话为什么中间会出现两个None [1赞]

- pyhhou 2019-05-17 05:57:22
  思考题 1：
  第一种方法更快，原因感觉上是和之前一样，就是不需要去调用相关的函数，而且像老师说的那样 {} 应该是关键字，内部会去直接调用底层C写好的代码

  思考题 2:
  用列表作为 Key 在这里是不被允许的，因为列表是一个动态变化的数据结构，字典当中的 key 要求是不可变的，原因也很好理解，key 首先是不重复的，如果 Key 是可以变化的话，那么随着 Key 的变化，这里就有可能就会有重复的 Key，那么这就和字典的定义相违背；如果把这里的列表换成之前我们讲过的元组是可以的，因为元组不可变
  [1赞]

- 夜行 2019-05-17 08:50:07
  旧的字典没有索引吗

- 无法言喻. 2019-05-17 08:46:39
  列表不可hash 所以不行

- ♀大力. 2019-05-17 08:16:32

1. OptionA 效率更高 ，因为调用dict函数等于运行时又增加了一层额外的操作，但达到的预期效果是一样的

2.python中，字典的键是不可变的，但列表是可变数据类型，如果使用列表当键，后期操作很可能出现键重复的问题

- 許敲敲 2019-05-17 08:06:44

  这里的源码，就是官网嘛？

  思考题:

  第一题 是和昨天的概念一样嘛？

  第二题 列表是可变的，不同列表对应 hash value不一样，所以不可以。解决方案，用元组

- Tango 2019-05-17 07:30:05

  第五天打卡。

- 大龄小学生 2019-05-17 03:54:21

  老师，字典和集合的空间复杂度是什么情况？

- farFlight 2019-05-17 03:45:35

  老师好，在王争老师的数据结构课程中提到哈希表常与链表一起使用，譬如用来解决哈希冲突。请问python底层对字典和集合的实现是否也是这样的呢？

- yshan 2019-05-17 01:27:12

  Option A更高效，应该也是与函数内部调用有关

  key不能为list，因为key不可变