第三章 字典和集合

1. 字典

1.1 常见的创建字典

```
a = dict(one=1,two=2,three=3)
b = {"one":1,"two":2,"three":3}
c = dict(zip(['one','two', 'three'],[1,2,3]))
d = dict([('one',1),('two',2),('three',3)])
e = dict({‘one’:1,"two":2,"three":3})
```

1.2 字典推导式

以下是通过list的元组属性，实现字典充值。

```
DIAL_CODES = [ (86, 'China'), (91, 'India'),(1, 'United States'),(62, 'Indonesia'),(55, 'Brazil'),(92, 'Pakistan'), (880, 'Bangladesh'), (234, 'Nigeria'), (7, 'Russia'),(81, 'Japan')]

country_code = {country: code for code, country in DIAL_CODES} # 国家名为key，区域码为value
print(country_code)
country_code_2  = {code: country for country, code in country_code.items() if code <66}
print(country_code_2)
```

1.3 其他方法

1.3.1 enumrate

enumerate(可迭代可遍历的对象,1) # 可以指定索引的起始值

```
for id, (key, value) in dict.items():
    print(id,(key,value)) # id:1,2,3,4.....即从1开始
```

1.3.2 setdefault

setdefault方法用于设置key的默认值。该方法接收两个参数，第1个参数表示key，第2个参数表示默认值。该方法会返回这个默认值。如果未指定第2个参数，那么key的默认值是None。

如果key在字典中不存在，那么setdefault方法会向字典中添加这个key，并用第2个参数作为key的值。

如果key在字典中已经存在，setdefault不会修改key原来的值，而且该方法会返回key原来的值。

```
#定义一个空字典
dict = {}
print(dict.setdefault('name','Bill'))
#向字典中添加一个名为name的key，默认值是Bill，输出结果：Bill
print(dict)
#输出结果：{'name': 'Bill'}

print(dict.setdefault('name','Mike'))
```

```
#并没有改变name的值，输出结果：Bill
print(dict)
#输出结果：{'name': 'Bill'}

#向字典中添加一个名为age的key，默认值是None，输出结果：None
print(dict.setdefault('age'))
print(dict)
#输出结果：{'name': 'Bill', 'age': None}
```

其实如果 `key` 在字典中不存在，`setdefault(key,value)` 方法与 `dict[key] = value` 形式是完全一样的，区别就是当key在字典中存在的情况下。`setdefault(key,value)` 并不会改变原值，而 `dict[key] = value` 是会改变原值的。所以 `setdefault` 方法主要用于向字典中添加一个 `key-value` 对，而不是修改key对应的值。

```
遍历字典的key是否存在，如果不存在就添加并赋value为list，如果存在就value值累加
my_dict.setdefault(key,[]).append(new_value)
等价于：
if key not in my_dict:
    my_dict[key] = []
my_dict[key].append(new_value
```

1.3.3 collections.defaultdict

defaultdict的作用是在于，当**字典里的key不存在但被查找时，返回的不是keyError而是一个默认值**

```
dict =defaultdict( factory_function)
```

这个factory_function可以是list、set、str等等，作用是当key不存在时，返回的是工厂函数的默认值，比如list对应[ ]，str对应的是空字符串，set对应set( )，int对应0，如下举例：

```
from collections import defaultdict

dict1 = defaultdict(int)
dict2 = defaultdict(set)
dict3 = defaultdict(str)
dict4 = defaultdict(list)
dict1[2] ='two'

print(dict1[1]) # 0
print(dict2[1]) # set()
print(dict3[1]) # 空格
print(dict4[1]) # []
--------------------------------------------------------------------------------
import collections
bag = ['apple', 'orange', 'cherry', 'apple','apple', 'cherry', 'blueberry']
count = collections.defaultdict(int)
for fruit in bag:
    count[fruit] += 1

print(count) # defaultdict(<class 'int'>, {'apple': 3, 'orange': 1,
'cherry': 2, 'blueberry': 1})
```

1.3.4 collections.OrderedDict

1. python中的字典是无序的，因为它是按照hash来存储的，但是python中有个模块collections(英文，收集、集合)，里面自带了一个子类OrderedDict，实现了对字典对象中元素的排序。**换言之，也是创建一个字典，只是这个字典添加进去的值是按照添加进去的顺利存储的。**
2. OrderedDict对象的字典对象，如果其顺序不同那么Python也会把他们当做是两个不同的对象

```
import collections
d1 = collections.OrderedDict()
d1['a'] = 'A'
d1['b'] = 'B'
d1['c'] = 'C'
d1['1'] = '1'
d1['2'] = '2'
for k, v in d1.items():
    print(k, v)
"""
    a A
    b B
    c C
    1 1
    2 2
"""
第二个知识点：创建的OrderedDict对象，如果添加的key，value顺利不能完全一直，则表示是两个不同的字典。因为在常规的字典中，只要key，value相同，不管顺序，则是一个相同的字典。
具体详细案例：https://www.cnblogs.com/gide/p/6370082.html
```

1.3.5 collections.ChainMap

功能1：在两个dict中查找需要的元素，先在第一个dict中查找，如果找到就返回结束，如果在第一个字典中没有找到就开始在第二个dict中查找；找到就返回，还是未找到就返回None。

```
from collections import ChainMap
a = {'x': 1, 'z': 3}
b = {'y': 2, 'z': 4}
c = ChainMap(a, b)
print(c.get('z')) # 3
```

功能2：在chainMap()可以传入n个dict，然后针对这n个dict，可以有增删改查的功能，并且还会以n个dict作为list元素，返回list结构的数据格式。其中涉及到的函数有：`combined.parents：ChainMap formed after removing the first dict, cobined.maps: return of list type`

```
from collections import ChainMap

combined = ChainMap({"a": "A"}, {"b": "B"}, {"c": "C"}, {"d": "D"})
# 访问元素
element_a = combined["a"] # A

# 增删，修改只对第一个dict有用
# 增加元素
combined["big_a"] = "big_A"  # combined = ChainMap({'a': 'A', 'big_a': 'big_A'}, {'b': 'B'}, {'c': 'C'}, {'d': 'D'})
combined["b"] = "big_B"  # ChainMap({'a': 'A', 'big_a': 'big_A', 'b': 'big_B'}, {'b': 'B'}, {'c': 'C'}, {'d': 'D'})
print(combined)
# 删除元素
```

```
del combined["b"]  # ChainMap({'a': 'A', 'big_a': 'big_A'}, {'b': 'B'}, {'c': 'C'}, {'d': 'D'})
print(combined)

# new_child()，用一个空dict插到第一个dict前面后构成的ChainMap
to_child = combined.new_child()  # to_child = ChainMap({}, {'a': 'A', 'big_a': 'big_A'}, {'b': 'B'}, {'c': 'C'}, {'d': 'D'})
print(to_child)
# parents，除去第一个dict后构成的ChainMap
to_parents = combined.parents  # to_parents = ChainMap({'b': 'B'}, {'c': 'C'}, {'d': 'D'})
print(to_parents)

# maps，得到成员dict的列表
to_maps = combined.maps  # to_maps = [{'a': 'A', 'big_a': 'big_A'}, {'b': 'B'}, {'c': 'C'}, {'d': 'D'}]
print(to_maps)

# 有重复keys时访问元素，在前面的才是最终会被访问的
com_repeat = ChainMap({"a": "A", "d": "good_D"}, {"b": "B"}, {"c": "C"}, {"d": "D"})
element_d = com_repeat["d"]  # element_d = "good_D"
print(element_d)

# 注意，构成ChainMap是使用dict的引用，因而原来的dict改变，会导致ChainMap对象跟着改变
dict1, dict2, dict3, dict4 = {"a": "A", "d": "good_D"}, {"b": "B"}, {"c": "C"}, {"d": "D"}
com_new = ChainMap(dict1, dict2, dict3, dict4)
print(com_new)
dict4["d"] = "big_D"  # com_new = ChainMap({'a': 'A', 'd': 'good_D'}, {'b': 'B'}, {'c': 'C'}, {'d': 'big_D'})
print(com_new)
```

1.3.6 collections.Counter

Function 1：统计词频，以dict类型返回

```
from collections import Counter
colors = ['red', 'blue', 'red', 'green', 'blue', 'blue']
c = Counter(colors)
print(c) # Counter({'blue': 3, 'red': 2, 'green': 1}) OR dict(c):
```

Function 2：可以传入迭代器（字符串、字典、元组等）然后实现各种操作

```
c = Counter('gallahad')                 # 传进字符串
c = Counter({'red': 4, 'blue': 2})      # 传进字典
c = Counter(cats=4, dogs=8)             # 传进元组
```

Function 3：判断是否包含某元素，可以转化为dict然后通过dict判断，Counter也带有函数可以判断：

```
c = Counter(['eggs', 'ham'])
c['bacon']# 不存在就返回0
```

Function 4：删除元素：

```
c['sausage'] = 0  # counter entry with a zero count
del c['sausage']
```

Function 5：获得所有元素

```
c = Counter(a=4, b=2, c=0, d=-2)
list(c.elements())
#['a', 'a', 'a', 'a', 'b', 'b']
```

Function 6：查看最常见出现的k个元素：most_common(n)

```
Counter('abracadabra').most_common(3)
#[('a', 5), ('r', 2), ('b', 2)]
```

Function 7：Counter更新

```
c = Counter(a=3, b=1)
d = Counter(a=1, b=2)
c + d                       # 相加
#Counter({'a': 4, 'b': 3})
c - d                       # 相减，如果小于等于0，删去
#Counter({'a': 2})
c & d                       # 求最小
#Counter({'a': 1, 'b': 1})
c | d                       # 求最大
#Counter({'a': 3, 'b': 2})
```

1.3.7 MappingProxyType

简而言之，MappingProxyType就是对原始字典创建一个映射试图，然后进行只读操作。

```
from types import MappingProxyType
# 创建一个集合
index_a = {'a': 'b'}
# 创建index_a的映射视图
a_proxy = MappingProxyType(index_a)
print(a_proxy) # {'a': 'b'}

a_proxy['a']
# #不能对a_proxy视图进行修改
# a_proxy['b'] = 'bb'
# 但是可以对原映射进行修改
index_a['b'] = 'bb'
print(a_proxy)
```

2. 集合

2.1 Function 1： 去重

```
l = ['spam', 'spam', 'eggs', 'spam']
ll = set(l)
lll = list(set(l))
print(ll,lll) # {'eggs', 'spam'} ['eggs', 'spam']
```

2.2 Function 2：合、差、交集、比较运算符

集合运算符

给定两个集合`a`和`b`，`a|b`返回的是它们的合集，`a & b`得到的是交集，而`a - b`得到的是差集，`a^b`为对称差集。

```
# 方式一：集合a的元素在集合b中出现的次数，其中a，b都是set类型
found = len(a&b)
# 方式二：常规方式
found = 0
for n in a:
    if n in b:
        found+=1
```

对于不是set类型的可迭代对象`a`和`b`，还有一种实现方式：

```
found = len(set(a) & set(haystack))
# 另一种写法：
found = len(set(needles).intersection(haystack))
```

集合比较运算符

数学符号	Python 运算符	方法	描述
		s.isdisjoint(z)	查看 s 和 z 是否不相交（没有共同元素）
e ∈ S	e in s	s.__contains__(e)	元素 e 是否属于 s
S ⊆ Z	s <= z	s.__le__(z)	s 是否为 z 的子集
		s.issubset(it)	把可迭代的 it 转化为集合，然后查看 s 是否为它的子集
S ⊂ Z	s < z	s.__lt__(z)	s 是否为 z 的真子集
S ⊇ Z	s >= z	s.__ge__(z)	s 是否为 z 的父集
		s.issuperset(it)	把可迭代的 it 转化为集合，然后查看 s 是否为它的父集
S ⊃ Z	s > z	s.__gt__(z)	s 是否为 z 的真父集

集合常见的方法

	set	frozenset	
s.add(e)	•		把元素 e 添加到 s 中
s.clear()	•		移除掉 s 中的所有元素
s.copy()	•	•	对 s 浅复制
s.discard(e)	•		如果 s 里有 e 这个元素的话，把它移除
s.__iter__()	•	•	返回 s 的迭代器
s.__len__()	•	•	len(s)
s.pop()	•		从 s 中移除一个元素并返回它的值，若 s 为空，则抛出 KeyError 异常
s.remove(e)	•		从 s 中移除 e 元素，若 e 元素不存在，则抛出 KeyError 异常

2.3 Function 3：集合创建

```
a = set()
b = {1,2,3} # 必须需要初始化，否则是一个字典
```

像 `{1, 2, 3}` 这种字面量句法相比于构造方法（`set([1, 2, 3])`）要更快且更易读。后者的速度要慢一些，因为 Python 必须先从set这个名字来查询构造方法，然后新建一个列表，最后再把这个列表传入到构造方法里。

```
from dis import dis # 查看字节码
zjm_01  = dis('{1}')
zjm_02 = dis('set([1])')
```

```
{'spam', 'eggs'} ['spam', 'eggs']
  1           0 LOAD_CONST               0 (1)
              2 BUILD_SET                1
              4 RETURN_VALUE
  1           0 LOAD_NAME                0 (set)
              2 LOAD_CONST               0 (1)
              4 BUILD_LIST               1
              6 CALL_FUNCTION            1
              8 RETURN_VALUE
```

2.4 Functions 4：集合推导

```python
set_a = {value for value in "有人云淡风轻，有人负重前行"}
print(set_a) # {'云', '轻', '风', '前', '重', '人', '负', ',', '行', '有', '淡'}
"""@note
1. 集合是无序且不重复的,所以会自动去掉重复的元素,并且每次运行显示的顺序不一样
2. 集合推导式就是将列表推导式的[]换成{},字典推导式就是推导出两个值并构建成键值对的样子
3. 不管是字典推导式还是集合推导式,后面都可以像列表推导式一样接if条件语句,嵌套循环等
"""
```

2.5 dict、set、list

总而言之，三者的处理数据的速度：dict>set>list.

具体测试就不举例了，这是通过书中测试得到的结论。

3 注意事项

由于字典使用了散列表，而散列表又必须是稀疏的，这导致它在空间上的效率低下。举例而言，如果你需要存放数量巨大的记录，那么放在由**元组或是具名元组构成的列表**中会是比较好的选择；最好不要根据 JSON 的风格，用由字典组成的列表来存放这些记录。用元组取代字典就能节省空间的原因有两个：其一是避免了散列表所耗费的空间，其二是无需把记录中字段的名字在每个元素里都存一遍。

以上是空间优化方案。

时间上，字典是典型的空间换时间，查询上千万数据，时间上毫不吃力。

不要对字典同时进行迭代和修改。如果想扫描并修改一个字典，最好分成两步来进行：首先对字典迭代，以得出需要添加的内容，把这些内容放在一个新字典里；迭代结束之后再对原有字典进行更新。

集合里的元素必须是可散列的。

集合很消耗内存。

可以很高效地判断元素是否存在于某个集合。

元素的次序取决于被添加到集合里的次序。

往集合里添加元素，可能会改变集合里已有元素的次序