###### Python系列之 - Collections系列

2016年01月12日 10:29:30

阅读数：6710

collections的常用类型有：

计数器(Counter)

双向队列(deque)

默认字典(defaultdict)

有序字典(OrderedDict)

可命名元组(namedtuple)

使用以上类型时需要导入模块 from collections import \*

### 1. Counter

Counter 作为字典(dict)的一个子类用来进行hashtable计数，将元素进行数量统计、计数后返回一个字典,键值为元素：值为元素个数

**[python]** [view plain](https://blog.csdn.net/songfreeman/article/details/50502194) [copy](https://blog.csdn.net/songfreeman/article/details/50502194)

1. s = 'abcbcaccbbad'
2. l = ['a','b','c','c','a','b','b']
3. d = {'2': 3, '3': 2, '17': 2}
4. # Counter 获取各元素的个数,返回字典
5. **print**(Counter(s))   # Counter({'c': 4, 'b': 4, 'a': 3})
6. **print**(Counter(l))   # Counter({'b': 3, 'a': 2, 'c': 2})
7. **print**(Counter(d))   # Counter({3: 3, 2: 2, 17: 1})

#### most\_common

**[python]** [view plain](https://blog.csdn.net/songfreeman/article/details/50502194) [copy](https://blog.csdn.net/songfreeman/article/details/50502194)

1. # most\_common(int) 按照元素出现的次数进行从高到低的排序,返回前int个元素的字典
2. m1 = Counter(s)
3. **print**(m1)                 # Counter({'c': 4, 'b': 4, 'a': 3, 'd': 1})
4. **print**(m1.most\_common(3))  # [('c', 4), ('b', 4), ('a', 3)]

#### elements

**[python]** [view plain](https://blog.csdn.net/songfreeman/article/details/50502194) [copy](https://blog.csdn.net/songfreeman/article/details/50502194)

1. # elements 返回经过计数器Counter后的元素,返回的是一个迭代器
2. e1 = Counter(s)
3. **print**(''.join(sorted(e1.elements())))  # aaabbbbcccc
4. e2 = Counter(d)
5. **print**(sorted(e2.elements()))  # ['17', '17', '2', '2', '2', '3', '3'] 字典返回value个key

#### update

**[python]** [view plain](https://blog.csdn.net/songfreeman/article/details/50502194) [copy](https://blog.csdn.net/songfreeman/article/details/50502194)

1. # update 和set集合的update一样,对集合进行并集更新
2. u1 = Counter(s)
3. u1.update('123a')
4. **print**(u1)  # Counter({'a': 4, 'c': 4, 'b': 4, '1': 1, '3': 1, '2': 1})

#### substract

**[python]** [view plain](https://blog.csdn.net/songfreeman/article/details/50502194) [copy](https://blog.csdn.net/songfreeman/article/details/50502194)

1. # substract 和update类似，只是update是做加法，substract做减法,从另一个集合中减去本集合的元素，
2. sub1 = 'which'
3. sub2 = 'whatw'
4. subset = Counter(sub1)
5. **print**(subset)   # Counter({'h': 2, 'i': 1, 'c': 1, 'w': 1})
6. subset.subtract(Counter(sub2))
7. **print**(subset)   # Counter({'c': 1, 'i': 1, 'h': 1, 'a': -1, 't': -1, 'w': -1}) sub1中的h变为2，sub2中h为1,减完以后为1

#### iteritems

与字典dict的items类似，返回由Counter生成的字典的所有item,只是在Counter中此方法返回的是一个迭代器,而不是列表

#### iterkeys

与字典dict的keys方法类似，返回由Counter生成的字典的所有key,只是在Counter中此方法返回的是一个迭代器,而不是列表

#### itervalues

与字典dict的values方法类似，返回由Counter生成的字典的所有value,只是在Counter中此方法返回的是一个迭代器,而不是列表

### 2.deque

deque 包含在文件\_collections.py中,属于高性能的数据结构(High performance data structures)之一.可以从两端添加和删除元素,常用的结构是它的简化版。

deque常用方法：

#### deque

**[python]** [view plain](https://blog.csdn.net/songfreeman/article/details/50502194) [copy](https://blog.csdn.net/songfreeman/article/details/50502194)

1. str1 = 'abc123cd'
2. dq = deque(str1)
3. **print**(dq)        # deque(['a', 'b', 'c', '1', '2', '3', 'c', 'd'])

#### append

队列右边添加元素

#### appendleft

队列左边添加元素

**[python]** [view plain](https://blog.csdn.net/songfreeman/article/details/50502194) [copy](https://blog.csdn.net/songfreeman/article/details/50502194)

1. dq = deque('abc123')
2. dq.append('right')
3. dq.appendleft('left')
4. **print**(dq) # deque(['left', 'a', 'b', 'c', '1', '2', '3', 'right'])

#### clear

clear 清空队列中的所有元素

#### count

count(value)  返回队列中包含value的个数,结果类型为 integer

#### extend

extend 队列右边扩展,可以是列表、元组或字典，如果是字典则将字典的key加入到deque

#### extendleft

extendleft  同extend, 在左边扩展

**[python]** [view plain](https://blog.csdn.net/songfreeman/article/details/50502194) [copy](https://blog.csdn.net/songfreeman/article/details/50502194)

1. dq = deque('abc123')
2. dq.extend({1:10,2:20})
3. dq.extendleft('L')
4. **print**(dq) # deque(['L', 'a', 'b', 'c', '1', '2', '3', 1, 2])

#### pop

pop  移除并且返回队列右边的元素

#### popleft

popleft 移除并且返回队列左边的元素

#### remove

remove(value) 移除队列第一个出现的元素（从左往右开始的第一次出现的元素value)

#### reverse

reverse  队列的所有元素进行反转

#### rotate

rotate(n) 对队列的数进行移动,若n<0，则往左移动即将左边的第一个移动到最后,移动n次，n>0 往右移动

**[python]** [view plain](https://blog.csdn.net/songfreeman/article/details/50502194) [copy](https://blog.csdn.net/songfreeman/article/details/50502194)

1. dq = deque([1,2,3,4,5])
2. dq.rotate(-1) # 左移,1往左移动一位到5后面
3. **print**(dq)

### 3.defaultdict

默认字典,是字典的一个子类,继承有字典的方法和属性,默认字典在进行定义初始化的时候可以指定字典值得默认类型：

**[python]** [view plain](https://blog.csdn.net/songfreeman/article/details/50502194) [copy](https://blog.csdn.net/songfreeman/article/details/50502194)

1. dic = collections.defaultdict(dict)
2. dic['k1'].update({'k2':'aaa'})
3. **print**(dic)

我们看上面的例子,字典dic在定义的时候就定义好了值为字典类型,虽然现在字典中还没有键值 k1，但仍然可以执行字典的update方法. 这种操作方式在传统的字典类型中是无法实现的,必须赋值以后才能进行值得更新操作，否则会报错。  
我看看一下传统的字典类型

**[python]** [view plain](https://blog.csdn.net/songfreeman/article/details/50502194) [copy](https://blog.csdn.net/songfreeman/article/details/50502194)

1. b = dict()
2. b['k1'].append('2')
3. # TypeError: 'type' object is not iterable

### 4.OrderedDict

OrderDict 叫做有序字典,也是字典类型(dict)的一个子类,是对字典的一个补充。 前面我们说过,字典类型是一个无序的集合,如果要想将一个传统的字典类型进行排序一般会怎么做了，我们可能会将字典的键值取出来做排序后在根据键值来进行有序的输出,我们看下面的一个例子：

**[python]** [view plain](https://blog.csdn.net/songfreeman/article/details/50502194) [copy](https://blog.csdn.net/songfreeman/article/details/50502194)

1. # 定义传统字典
2. dic1 = dict()
3. # 按顺序添加字典内容
4. dic1['a'] = '123'
5. dic1['b'] = 'jjj'
6. dic1['c'] = '394'
7. dic1['d'] = '999'
8. **print**(dic1)    # 结果： {'a': '123', 'c': '394', 'b': 'jjj', 'd': '999'}
9. # 排序
10. dic1\_key\_list = []
11. **for** k **in** dic1.keys():
12. dic1\_key\_list.append(k)
13. dic1\_key\_list.sort()
14. **for** key **in** dic1\_key\_list:
15. **print**('dic1字典排序结果 %s:%s' %(key,dic1[key]))

以上为定义传统字典类型时的一个简单排序过程。 如果我们定义一个有序字典时,将不用再如此麻烦, 字典顺序将按照录入顺序进行排序且不会改变。

**[python]** [view plain](https://blog.csdn.net/songfreeman/article/details/50502194) [copy](https://blog.csdn.net/songfreeman/article/details/50502194)

1. # 定义有序字典
2. dic2 = OrderedDict()
3. dic2['a'] = '123'
4. dic2['b'] = 'jjj'
5. dic2['c'] = 'abc'
6. dic2['d'] = '999'
7. **for** k, v **in** dic2.iteritems():
8. **print**('有序字典：%s:%s' %(k,v))

### 5.nametuple

标准的tuple类型使用数字索引来访问元素,

**[python]** [view plain](https://blog.csdn.net/songfreeman/article/details/50502194) [copy](https://blog.csdn.net/songfreeman/article/details/50502194)

1. bob = ('Bob', 30, 'male')
2. **print**('Representation:', bob)
4. jane = ('Jane', 29, 'female')
5. **print**('\nField by index:', jane[0])
7. **print**('\nFields by index:')
8. **for** p **in** [bob, jane]:
9. **print**('%s is a %d year old %s' % p)

这种对于标准的元组访问,我们需要知道元素对应下标索引值,但当元组的元素很多时,我们可能无法知道每个元素的具体索引值,这个时候就是可命名元组登场的时候了。

nametuple 的创建是由自己的类工厂nametuple()进行创建,而不是由标准的元组来进行实例化，通过nametuple()创建类的参数包括类名称和一个包含元素名称的字符串

**[python]** [view plain](https://blog.csdn.net/songfreeman/article/details/50502194) [copy](https://blog.csdn.net/songfreeman/article/details/50502194)

1. **from** collections **import** namedtuple
3. #创建一个nametuplede 类,类名称为Person，并赋给变量P
4. P = namedtuple('Person', 'name,age,gender')
5. **print**('Type of Person:', type(P))  # Type of Person: <class 'type'>
7. #通过Person类实例化一个对象bob
8. bob = P(name='Bob', age=30, gender='male')
9. **print**('\nRepresentation:', bob)  # Representation: Person(name='Bob', age=30, gender='male')
11. #通过Person类实例化一个对象jane
12. jane = P(name='Jane', age=29, gender='female')
13. **print**('\nField by name:', jane.name)  # Field by name: Jane
15. **print**('\nFields by index:')
16. **for** p **in** [bob, jane]:
17. **print**('%s is a %d year old %s' % p)
18. # Fields by index:
19. # Bob is a 30 year old male
20. # Jane is a 29 year old female

通过上面的实例可以看出,我们通过nametuple()创建了一个Person的类,并复制给P变量,Person的类成员包括name,age,gender,并且顺序已经定了,在实例化zhangsan这个对象的时候,对张三的属性进行了定义。这样我们在访问zhangsan这个元组的时候就可以通过张三的属性来复制(zhangsan.name、zhangsan.age等)。这样就算这个元组有1000个元素我们都能通过元素的名称来访问而不用考虑元素的下标索引值。

非法的参数值

使用nametuple()来创建类的时候,传递的成员属性参数名称不能非法（不能为系统参数名称），且参数名称不能重复,否则会报值错误

**[python]** [view plain](https://blog.csdn.net/songfreeman/article/details/50502194) [copy](https://blog.csdn.net/songfreeman/article/details/50502194)

1. # 参数字段的名称非法,包含系统名称class
2. **try**:
3. p = namedtuple('Person','age,name,class,gender')
4. **print**(p.\_fields)
5. **except** ValueError as err:
6. **print**(err)
8. # Type names and field names cannot be a keyword: 'class'
10. # 类成员字段参数名称重复 age
11. **try**:
12. p1 = namedtuple('Person','age,gender,name,age')
13. **print**(p1.\_fields)
14. **except** ValueError as err:
15. **print**(err)
16. # Encountered duplicate field name: 'age'

但是也有时候我们是无法控制的,如果参数的名称来自外部，比如是通过读取数据库中的内容来传递的参数,此时我们无法手工的修改参数名称,那该如何是好呢! 别担心,只需要增加一个属性就OK了，它就是rename

**[python]** [view plain](https://blog.csdn.net/songfreeman/article/details/50502194) [copy](https://blog.csdn.net/songfreeman/article/details/50502194)

1. # 参数字段的名称非法,包含系统名称class
2. **try**:
3. p = namedtuple('Person','age,name,class,gender',rename=True)
4. **print**(p.\_fields)
5. **except** ValueError as err:
6. **print**(err)
8. # ('age', 'name', '\_2', 'gender')
10. # 类成员字段参数名称重复 age
11. **try**:
12. p1 = namedtuple('Person','age,gender,name,age',rename=True)
13. **print**(p1.\_fields)
14. **except** ValueError as err:
15. **print**(err)
16. # ('age', 'gender', 'name', '\_3')

从以上的实例我们看出，当有参数错误的时候，系统自动将错误的参数通过增加 "下划线+参数索引" 的方式自动将参数名称替换了。

**collections defaultdict（默认字典）**

2017年03月02日 00:42:31

阅读数：699

**class collections.default([default\_factory[, ...]])**   
返回一个类字典对象。**defaultdict**是内置类型**dict**的子类。他重写了父的一个方法并且增加了一个可以的实例变量。余下的功能与字典的一样，在这里就不写文档了。   
第一个参数为**default\_factory**属性提供初始值；**default\_factory**的默认值为**None**.余下的参数被视为**dict**构造器的参数包括关键字参数

**defaultdict**对象除了支持标准字典操作外，还支持下面的方法   
**\_\_missing\_\_(key)**

* 如果**default\_factory**属性为**None**，以*key*作为参数引发一个**KeyError**异常
* 如果**default\_factory**不是**None**,这个方法会被调用为*key*提供一个默认值，这个值会被赋值给*key*,并且返回
* 当调用**default\_factory**引发一个异常时，这个异常会被原封不动的传播
* 当调用**dict**的\_\_getitme\_\_()而需要的*key*没有被找到时，这个方法无论返回或引发什么都会被\_\_getitem\_\_()原封不动的返回
* 注意这个\_\_missing\_\_()不会被任何操作调用除了\_\_getitem()。这代表着**get()**方法，会像正常的字典方法一样，返回默认的**None**而不是调用**default\_factory**

**defaultdict**对象提供下面的实例变量：   
这个属性用于**\_\_missing\_\_()**方法。他初始化由第一个参数，当作构造器，如果第一个参数存在，如果不存在，会是None

**defaultdict 的例子**

使用**list**作为**default\_factory**,他很容易的将一个以键值形式表现的序列分组成一个字典列表

>>> s = [('yellow', 1), ('blue', 2), ('yellow', 3), ('blue', 4), ('red', 1)]

>>> d = defaultdict(list)

>>> for k, v in s:

... d[k].append(v)

...

>>> d.items()

[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]

当每个**key**第一次被访问时，他们肯定没有在映射中：一个入口自动的被创建了使用**default\_factory**函数返回一个空的列表。然后**list.append()**操作将值放入新的列表中。当key再次被访问时，查找工作正常（返回这个key的列表）然后**list.append()**操作为列表添加别外的值。这个技巧比等价的技艺（使用**dict.setdefault()**)更简节，更快

>>> d = {}

>>> for k, v in s:

... d.setdefault(k, []).append(v)

...

>>> d.items()

[('blue', [2, 4]), ('red', [1]), ('yellow', [1, 3])]

将**default\_factory**设置成**int**使**defaultdict**对于计数非常有用

>>> s = 'mississippi'

>>> d = defaultdict(int)

>>> for k in s:

... d[k] += 1

...

>>> d.items()

[('i', 4), ('p', 2), ('s', 4), ('m', 1)]

当一个字每第一次被访问时，这个字母在映射中不存在，因此**default\_factory**函数调用**int()**提供一个默认的0作为count。然后自增操作计算每个字母出现的次数   
**int()**函数总是返回0，因为他是常量函数的一个特殊的例子。一个更快更灵活的方式是创建一个常量函数（**itertools.repeat()** –可以提供任何常量，不仅仅是0)

>>> def constant\_factory(value):

... return itertools.repeat(value).next

>>> d = defaultdict(constant\_factory('<missing>'))

>>> d.update(name='John', action='ran')

>>> '%(name)s %(action)s to %(object)s' % d

'John ran to <missing>'

设置**default\_factory**为**set**使**defaultdict**构建集合字典非常有用

>>> s = [('red', 1), ('blue', 2), ('red', 3), ('blue', 4), ('red', 1), ('blue', 4)]

>>> d = defaultdict(set)

>>> for k, v in s:

... d[k].add(v)

...

>>> d.items()

[('blue', set([2, 4])), ('red', set([1, 3]))]