1. 容器数据类型-collections

一. Counter 作为一个容器，可以跟踪相同的值增加了多少次.

1.1 三种初始化方式：

In [1]:
```
import collections

c1 = collections.Counter(['a', 'b', 'a', 'a', 'b'])
print(c1)
```

```
Counter({'a': 3, 'b': 2})
```

In [2]:
```
c2 = collections.Counter({"a": 4, "c": 5})
print(c2)
```

```
Counter({'c': 5, 'a': 4})
```

In [3]:
```
c3 = collections.Counter(a=5, b=3, z=2)
print(c3)
```

```
Counter({'a': 5, 'b': 3, 'z': 2})
```

1.2 如果不提供参数，可以创建一个空的Counter，然后通过update向其中添加参数.

In [4]:
```
c4 = collections.Counter()
c4.update('apple')
print(c4)
```

```
Counter({'p': 2, 'a': 1, 'l': 1, 'e': 1})
```

In [5]:
```
c4.update('abcde')
print(c4)
```

```
Counter({'a': 2, 'p': 2, 'e': 2, 'l': 1, 'b': 1, 'c': 1, 'd': 1})
```

1.3 获取Couter中的元素的个数

In [6]:
```
for j in c4:
    print("元素%s: 出现%s次!" %(j, c4[j]))
```

```
元素a: 出现2次!
元素p: 出现2次!
元素l: 出现1次!
元素e: 出现2次!
元素b: 出现1次!
元素c: 出现1次!
元素d: 出现1次!
```

1.4 通过elements生成一个迭代器.

In [7]:
```
c5 = c4.elements()
print(c5)
print(type(c5))
```

```
<itertools.chain object at 0x110d75860>
<class 'itertools.chain'>
```

1.5 most_common() 方法，可以生成一个序列，内容包括n个最常遇到的输入值及其相应的记数. 以下例子是统计字符串中某个字符出现的次数，生成一个频度分布，如果不向most_common()传参，则返回由所有元素及其出现次数构成的列表，并且按频度排序， 如果传参则返回出现次数最多到少的前n个元素

In [10]:
```
c6 = collections.Counter("apppllle")
c6.most_common()
```

Out[10]:
```
[('p', 3), ('l', 2), ('a', 1), ('e', 1)]
```

In [11]:
```
c6.most_common(2)
```

Out[11]:
```
[('p', 3), ('l', 2)]
```

1.6 Couter支持算数和集合操作来完成结果的聚集

In [13]:
```
c1 = collections.Counter("abcabb")
c2 = collections.Counter("alphabet")

print(c1)
print(c2)

print(c1 + c2)
```

```
Counter({'b': 3, 'a': 2, 'c': 1})
Counter({'a': 2, 'l': 1, 'p': 1, 'h': 1, 'b': 1, 'e': 1, 't': 1})
Counter({'a': 4, 'b': 4, 'c': 1, 'l': 1, 'p': 1, 'h': 1, 'e': 1, 't': 1})
```

In [14]:
```
print(c1 - c2)
```

```
Counter({'b': 2, 'c': 1})
```

In [15]:
```
print(c1 & c2)
```

```
Counter({'a': 2, 'b': 1})
```

In [16]:
```
print(c1 | c2)
```

```
Counter({'b': 3, 'a': 2, 'c': 1, 'l': 1, 'p': 1, 'h': 1, 'e': 1, 't': 1})
```

二. defaultdict

为字典的所有键赋予相同的初始值与核心数据类型字典中的from_keys方法类似

In [17]:
```
x = collections.defaultdict(lambda: "default value", foo = "bar")
print(x)
```

```
defaultdict(<function <lambda> at 0x110d76730>, {'foo': 'bar'})
```

In [18]:
```
print(x["foo"])
```

```
bar
```

In [19]:
```
print(x["bar"])
```

```
default value
```

三. OrderedDict

在Python核心数据类型的字典中的key是无序的，而OrderedDict可以记住字典中添加key时的顺序.

In [35]:
```
d = collections.OrderedDict()
d["banana"] = 10
d["orange"] = 10
d["apple"] = 20

print(d)
```

```
OrderedDict([('banana', 10), ('orange', 10), ('apple', 20)])
```

四. namedtuple
在使用核心数据类型的元组时，如果要使用其中的元素，需要记得元组的索引值，当元组中有大量的字段并且我们要获取的元素的索引相差很远的时候，可能发生类似取错元素这样的错误，namedtuple除了指定数值索引外，还指定了名字.

In [41]:
```
import collections

help(collections.namedtuple)
```

```
Help on function namedtuple in module collections:

namedtuple(typename, field_names, *, verbose=False, rename=False, module=None)
    Returns a new subclass of tuple with named fields.

    >>> Point = namedtuple('Point', ['x', 'y'])
    >>> Point.__doc__                   # docstring for the new class
    'Point(x, y)'
    >>> p = Point(11, y=22)             # instantiate with positional args or keywords
    >>> p[0] + p[1]                     # indexable like a plain tuple
    33
    >>> x, y = p                        # unpack like a regular tuple
    >>> x, y
    (11, 22)
    >>> p.x + p.y                       # fields also accessible by name
    33
    >>> d = p._asdict()                 # convert to a dictionary
    >>> d['x']
    11
    >>> Point(**d)                      # convert from a dictionary
    Point(x=11, y=22)
    >>> p._replace(x=100)               # _replace() is like str.replace() but targets named fields
    Point(x=100, y=22)
```

In [42]:
```
#示例代码1:

x = collections.namedtuple('Person', ["name", "age", "job"])
print(type(x))
```

```
<class 'type'>
```

In [50]:
```
lily = x(name="lily",age= 20, job = "it")
print(lily)
print(type(lily))
```

```
Person(name='lily', age=20, job='it')
<class '__main__.Person'>
```

In [52]:
```
#可以使用位置对元组内的元素进行获取
print(lily[1])
```

```
20
```

In [53]:
```
#可以使用名字访问namedtuple中的字段
print(lily.job)
```

```
it
```

五. deque(双端队列) 支持从任意一端增加或删除元素，更为常用的两种结构是队列和栈，他们的输入或输出只在序列的一端，序列(先进先出 类似火车过隧道)，栈(后进先出 类似弹夹)

5.1 定义双端队列

In [55]:
```
d = collections.deque("abcdefg")
print("length: %s" %len(d))
print("left: %s" %d[0])
print("right: %s" %d[-1])
```

```
length: 7
left: a
right: g
```

5.2 填充队列，可以从两边填充

In [56]:
```
#右侧填充
x = collections.deque()
x.extend("apple")
print(x)
```

```
deque(['a', 'p', 'p', 'l', 'e'])
```

In [61]:
```
x = collections.deque()
x.extendleft("apple")
print(x)


print(x.popleft())
print(x)
```

```
deque(['e', 'l', 'p', 'p', 'a'])
e
deque(['l', 'p', 'p', 'a'])
```

5.3 deque双端队列是线程安全的，可以在不同线程中从两侧获取双端队列中的内容

In [73]:
```
#示例代码:

import threading
x = collections.deque(range(10))

def f1(direct, method):
    current_thread = threading.current_thread().getName()
    print("Thread is : %s, direct is %s, value is %s" %(current_thread, direct, method()))
    if len(x) == 0:
        return

task = []
for _ in range(5):
    t1 = threading.Thread(target=f1, args=("left", x.popleft))
    t2 = threading.Thread(target=f1, args=("right", x.pop))
    task.append(t1)
    task.append(t2)

for j in task:
    j.start()

for j in task:
    j.join()
```

```
Thread is : Thread-74, direct is left, value is 0
Thread is : Thread-75, direct is right, value is 9
Thread is : Thread-76, direct is left, value is 1
Thread is : Thread-77, direct is right, value is 8
Thread is : Thread-78, direct is left, value is 2
Thread is : Thread-79, direct is right, value is 7
Thread is : Thread-80, direct is left, value is 3
Thread is : Thread-81, direct is right, value is 6
Thread is : Thread-82, direct is left, value is 4
Thread is : Thread-83, direct is right, value is 5
```

In [ ]: