《Python统计计算》

（2021年秋季学期）

翟祥
北京林业大学

E-mail: zhaixbh@126.com

编程基础

第2章 Python数据结构

常用的数据结构

- Python中常用的数据结构有
  - 序列：列表、元组、字符串
  - 映射：字典
  - 集合
- 除字典和集合之外，列表、元组、字符串等序列均支持双向索引
  - 正向：第一个元素下标为0，第二个元素下标为1，以此类推，最后一个元素下标n-1；
  - 逆向：最后一个元素下标为-1，倒数第二个元素下标为-2，以此类推，第一个元素下标 -n。

序列

- 序列是程序设计中经常用到的数据存储方式，几乎每一种程序设计语言都提供了类似的数据结构，如C和Basic中的一维、多维数组等。
- Python提供的序列类型在所有程序设计语言中是最丰富，最灵活，也是功能最强大的。
- 序列是一系列连续值，它们通常是相关的，并且按一定顺序排列。
- Python中的序列包括：列表、元组、字符串

2.1 列表

- 列表是Python中内置可变序列，是若干元素的有序集合，列表中的每一个数据称为元素，列表的所有元素放在一对中括号“[”和“]”中，并使用逗号分隔开；
- 当列表元素增加或删除时，列表对象自动进行扩展或收缩内存，保证元素之间没有缝隙；
- 在Python中，一个列表中的数据类型可以各不相同，可以同时分别为整数、实数、字符串等基本类型，甚至是列表、元组、字典、集合以及其他自定义类型的对象。例如：

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
['spam', 2.0, 5, [10, 20]]
[['file1', 200,7], ['file2', 260,9]]
```

列表方法

方法	说明
list.append(x)	将元素x添加至列表尾部
list.extend(L)	将列表L中所有元素添加至列表尾部
list.insert(index, x)	在列表指定位置index处添加元素x
list.remove(x)	在列表中删除首次出现的指定元素
list.pop([index])	删除并返回列表对象指定位置的元素
list.clear()	删除列表中所有元素，但保留列表对象，该方法在Python2中没有
list.index(x)	返回值为x的首个元素的下标
list.count(x)	返回指定元素x在列表中的出现次数
list.reverse()	对列表元素进行原地逆序
list.sort()	对列表元素进行原地排序
list.copy()	返回列表对象的浅拷贝，该方法在Python2中没有

2.1.1 列表创建与删除

- 使用“=”直接将一个列表赋值给变量即可创建列表对象，例如：

```
>>> a_list = ['a', 'b', 'mpilgrim', 'z', 'example']
>>> a_list = [] #创建空列表
```

- 或者，也可以使用list()函数将元组、range对象、字符串或其他类型的可迭代对象类型的数据转换为列表。例如：

```
>>> a_list = list((3,5,7,9,11))
>>> a_list
[3, 5, 7, 9, 11]
>>> list(range(1,10,2))
[1, 3, 5, 7, 9]
>>> list('hello world')
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
>>> x = list() #创建空列表
```

2.1.1 列表创建与删除

- 内置函数range()该函数语法为

range([start,] stop[, step])

内置函数range()接收3个参数，第一个参数表示起始值（默认为0），第二个参数表示终止值（结果中不包括这个值），第三个参数表示步长（默认为1），该函数在Python 3.x中返回一个range可迭代对象，在Python 2.x中返回一个包含若干整数的列表。

- Python 2.x还提供了一个内置函数xrange()

Python 3.x中不提供该函数

语法与range()函数一样，返回xrange可迭代对象，类似于Python 3.x的range()函数，其特点为惰性求值，而不是像range()函数一样返回列表。例如下面的Python 2.7.11代码：

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> xrange(10)
xrange(10)
>>> list(xrange(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

2.1.1 列表创建与删除

- 使用Python 2.x处理大数据或较大循环范围时，建议使用xrange()函数来控制循环次数或处理范围，以获得更高的效率。例如下面的Python 2.7.11代码对range()和xrange()的运行效率进行了简单的对比。

```
import time
import math
start = time.time()
for j in range(100000000):
    1+1
print time.time()-start
start = time.time()
for j in xrange(100000000):
    1+1
print time.time()-start
```

上面的代码运行结果为

```
15.7339999676
11.7339999676
```

2.1.1 列表创建与删除

- 当不再使用时，使用del命令删除整个列表，如果列表对象所指向的值不再有其他对象指向，Python将同时删除该值。

```
>>> del a_list
>>> a_list
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    a_list
NameError: name 'a_list' is not defined
```

2.1.2 列表元素的增加

（1）可以使用“+”运算符来实现将元素添加到列表中的功能。虽然这种用法在形式上比较简单也容易理解，但严格意义上来讲，这并不是真的为列表添加元素，而是创建一个新列表，并将原列表中的元素和新元素依次复制到新列表的内存空间。由于涉及大量元素的复制，该操作速度较慢，在涉及大量元素添加时不建议使用该方法。

```
>>> aList = [3,4,5]
>>> aList = aList + [7]
>>> aList
[3, 4, 5, 7]
```

2.1.2 列表元素的增加

（2）使用列表对象的append()方法，原地修改列表，是真正意义上的在列表尾部添加元素，速度较快，也是推荐使用的方法。

```
>>> aList.append(9)
>>> aList
[3, 4, 5, 7, 9]
```

为了比较“+”和append()这两种方法的速度差异，请看以下代码：

```
import time
result = []
start = time.time()
for i in range(10000):
    result = result + [i]
print(len(result), ',', time.time()-start)
result = []
start = time.time()
for i in range(10000):
    result.append(i)
print(len(result), ',', time.time()-start)
```

2.1.2 列表元素的增加

在上面的代码中，分别重复执行10000次“+”运算和append()方法为列表插入元素并比较这两种方法的运行时间。在代码中，使用time模块的time()函数返回当前时间，然后运行代码之后计算时间差。由于各种运行时的原因，多次运行上面的代码得到的结果会有微小的差别，其中一次运行的结果如下。可以看出，这两个方法的速度相差还是非常大的，使用append()方法比使用“+”运算快约70倍，相差两个数量级。

10000 , 0.21801209449768066

10000 , 0.00300002098083496

2.1.2 列表元素的增加

- Python采用的是基于值的自动内存管理方式，当为对象修改值时，并不是真的直接修改变量的值，而是使变量指向新的值，这对于Python所有类型的变量都是一样的。

```
>>> a = [1,2,3]
>>> id(a)
20230752
>>> a = [1,2]
>>> id(a)
20338208
```

2.1.2 列表元素的增加

- 对于列表、集合、字典等可变序列类型而言，情况稍微复杂一些。以列表为例，列表中包含的是元素值的引用，而不是直接包含元素值。
  - 如果是直接修改序列变量的值，则与Python普通变量的情况是一样的
  - 如果是通过下标来修改序列中元素的值或通过可变序列对象自身提供的方法来增加和删除元素时，序列对象在内存中的起始地址是不变的，仅仅是被改变值的元素地址发生变化。

2.1.2 列表元素的增加

```
>>> a = [1,2,4]
>>> b = [1,2,3]
>>> a == b
False
>>> id(a) == id(b)
False
>>> id(a[0]) == id(b[0])
True
>>> a = [1,2,3]
>>> id(a)
25289752
>>> a.append(4)
>>> id(a)
25289752
>>> a.remove(3)
>>> a
[1, 2, 4]
>>> id(a)
25289752
>>> a[0] = 5
>>> a
[5, 2, 4]
>>> id(a)
25289752
```

2.1.2 列表元素的增加

（3）使用列表对象的extend()方法可以将另一个迭代对象的所有元素添加至该列表对象尾部。通过extend()方法来增加列表元素也不改变其内存首地址，属于原地操作。例如，继续上面的代码执行下面的代码。

```
>>> a.extend([7,8,9])
>>> a
[5, 2, 4, 7, 8, 9]
>>> id(a)
25289752
>>> aList.extend([11,13])
>>> aList
[3, 4, 5, 7, 9, 11, 13]
>>> aList.extend((15,17))
>>> aList
[3, 4, 5, 7, 9, 11, 13, 15, 17]
>>> id(a)
25289752
```

2.1.2 列表元素的增加

（4）使用列表对象的insert()方法将元素添加至列表的指定位置。

```
>>> aList.insert(3,6)
>>> aList
[3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
```

2.1.2 列表元素的增加

- 应尽量从列表尾部进行元素的增加与删除操作。列表的insert()可以在列表的任意位置插入元素，但由于列表的自动内存管理功能，insert()方法会涉及到插入位置之后所有元素的移动，这会影响处理速度，类似的还有后面介绍的remove()方法以及使用pop()函数弹出列表非尾部元素和使用del命令删除列表非尾部元素的情况。

2.1.2 列表元素的增加

```
import time
def Insert():
    a = []
    for i in range(10000):
        a.insert(0, i)
def Append():
    a = []
    for i in range(10000):
        a.append(i)
start = time.time()
for i in range(10):
    Insert()
print('Insert:', time.time()-start)
start = time.time()
for i in range(10):
    Append()
print('Append:', time.time()-start)
```

运行结果如下：

```
Insert: 0.578000068665
Append: 0.0309998989105
```

2.1.2 列表元素的增加

（5）使用乘法来扩展列表对象，将列表与整数相乘，生成一个新列表，新列表是原列表中元素的重复。

```
>>> aList = [3,5,7]
>>> bList = aList
>>> id(aList)
57091464
>>> id(bList)
57091464
>>> aList = aList*3
>>> aList
[3, 5, 7, 3, 5, 7, 3, 5, 7]
>>> bList
[3,5,7]
>>> id(aList)
57092680
>>> id(bList)
57091464
```

2.1.2 列表元素的增加

- 要注意的是，当使用“*”运算符将包含列表的列表重复并创建新列表时，并不创建元素的复制，而是创建已有对象的引用。因此，当修改其中一个值时，相应的引用也会被修改，例如下面的代码：

```
>>> x = [[None] * 2] * 3
>>> x
[[None, None], [None, None], [None, None]]
>>> x[0][0] = 5
>>> x
[[5, None], [5, None], [5, None]]
>>> x = [[1,2,3]]*3
>>> x[0][0] = 10
>>> x
[[10, 2, 3], [10, 2, 3], [10, 2, 3]]
```

2.1.3 列表元素的删除

（1）使用del命令删除列表中的指定位置上的元素。前面已经提到过，del命令也可以直接删除整个列表，这里不再赘述。

```
>>> a_list = [3,5,7,9,11]
>>> del a_list[1]
>>> a_list
[3, 7, 9, 11]
```

2.1.3 列表元素的删除

（2）使用列表的pop()方法删除并返回指定（默认为最后一个）位置上的元素，如果给定的索引超出了列表的范围则抛出异常。

```
>>> a_list = list((3,5,7,9,11))
>>> a_list.pop()
11
>>> a_list
[3, 5, 7, 9]
>>> a_list.pop(1)
5
>>> a_list
[3, 7, 9]
```

2.1.3 列表元素的删除

（3）使用列表对象的remove()方法删除首次出现的指定元素，如果列表中不存在要删除的元素，则抛出异常。

```
>>> a_list = [3,5,7,9,7,11]
>>> a_list.remove(7)
>>> a_list
[3, 5, 9, 7, 11]
```

2.1.3 列表元素的删除

- 有时候可能需要删除列表中指定元素的所有重复，大家会很自然地想到使用“循环+remove()”的方法，但是具体操作时很有可能会出现意料之外的错误，代码运行没有出现错误，但结果却是错的，或者代码不稳定，对某些数据处理结果是正确的，而对某些数据处理结果却是错误的。例如，下面的代码成功地删除了列表中的重复元素，执行结果是完全正确的。

```
>>> x = [1,2,1,2,1,2,1,2,1]
>>> for i in x:
        if i == 1:
            x.remove(i)
>>> x
[2, 2, 2, 2]
```

2.1.3 列表元素的删除

- 然而，上面这段代码的逻辑是错误的，尽管执行结果是正确的。例如下面的代码同样试图删除列表中所有的“1”，代码完全相同，仅仅是所处理的数据发生了一点变化，然而当循环结束后却发现并没有把所有的“1”都删除，只是删除了一部分。

```
>>> x = [1,2,1,2,1,1,1]
>>> for i in x:
        if i == 1:
            x.remove(i)
>>> x
[2, 2, 1]
```

2.1.3 列表元素的删除

- 两组数据的本质区别在于，第一组数据中没有连续的“1”，而第二组数据中存在连续的“1”。出现这个问题的原因是列表的自动内存管理功能。前面已经提到，在删除列表元素时，Python会自动对列表内存进行收缩并移动列表元素以保证所有元素之间没有空隙，增加列表元素时也会自动扩展内存并对元素进行移动以保证元素之间没有空隙。每当插入或删除一个元素之后，该元素位置后面所有元素的索引就都改变了。

2.1.3 列表元素的删除

- 正确的代码：

```
>>> x = [1,2,1,2,1,1,1]
>>> for i in x[::]:
        if i == 1:
            x.remove(i)
```

或者：

```
>>> x = [1,2,1,2,1,1,1]
>>> for i in range(len(x)-1,-1,-1):
        if x[i]==1:
            del x[i]
```

2.1.4 列表元素访问与计数

- 使用下标直接访问列表元素

```
>>> aList[3]
6
>>> aList[3] = 5.5
>>> aList
[3, 4, 5, 5.5, 7, 9, 11, 13, 15, 17]
```

- 如果指定下标不存在，则抛出异常

```
>>> aList[15]
Traceback (most recent call last):
  File "<pyshell#34>", line 1, in <module>
    aList[15]
IndexError: list index out of range
```

2.1.4 列表元素访问与计数

- 使用列表对象的index方法获取指定元素首次出现的下标

```
>>> aList
[3, 4, 5, 5.5, 7, 9, 11, 13, 15, 17]
>>> aList.index(7)
4
```

- 若列表对象中不存在指定元素，则抛出异常

```
>>> aList.index(100)
Traceback (most recent call last):
  File "<pyshell#36>", line 1, in <module>
    aList.index(100)
ValueError: 100 is not in list
```

2.1.4 列表元素访问与计数

- 使用列表对象的count方法统计指定元素在列表对象中出现的次数

```
>>> aList
[3, 4, 5, 5.5, 7, 9, 11, 13, 15, 17]
>>> aList.count(7)
1
>>> aList.count(0)
0
>>> aList.count(8)
0
```

2.1.5 成员资格判断

- 如果需要判断列表中是否存在指定的值，可以使用前面介绍的count()方法，如果存在则返回大于0的数，如果返回0则表示不存在。或者，使用更加简洁的“in”关键字来判断一个值是否存在于列表中，返回结果为“True”或“False”。

```
>>> aList
[3, 4, 5, 5.5, 7, 9, 11, 13, 15, 17]
>>> 3 in aList
True
>>> 18 in aList
False
>>> bList = [[1], [2], [3]]
>>> 3 in bList
False
>>> 3 not in bList
True
```

（续）

```
>>> [3] in bList
True
>>> aList = [3, 5, 7, 9, 11]
>>> bList = ['a', 'b', 'c', 'd']
>>> (3, 'a') in zip(aList, bList)
True
>>> for a, b in zip(aList, bList):
        print(a, b)
```

2.1.6 切片操作

- 切片是Python序列的重要操作之一，适用于列表、元组、字符串、range对象等类型。
- 切片使用2个冒号分隔的3个数字来完成，第一个数字表示切片开始位置（默认为0），第二个数字表示切片截止（但不包含）位置（默认为列表长度），第三个数字表示切片的步长（默认为1），当步长省略时可以顺便省略最后一个冒号。可以使用切片来截取列表中的任何部分，得到一个新列表，也可以通过切片来修改和删除列表中部分元素，甚至可以通过切片操作为列表对象增加元素。
- 与使用下标访问列表元素的方法不同，切片操作不会因为下标越界而抛出异常，而是简单地在列表尾部截断或者返回一个空列表，代码具有更强的健壮性。

2.1.6 切片操作

```
>>> aList = [3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> aList[::]
[3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> aList[::-1]
[17, 15, 13, 11, 9, 7, 6, 5, 4, 3]
>>> aList[::2]
[3, 5, 7, 11, 15]
>>> aList[1::2]
[4, 6, 9, 13, 17]
>>> aList[3::]
[6, 7, 9, 11, 13, 15, 17]
>>> aList[3:6]
[6, 7, 9]
>>> aList[3:6:1]
[6, 7, 9]
>>> aList[0:100:1]
[3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> a[100:]
[]
```

2.1.6 切片操作

- 可以使用切片来原地修改列表内容

```
>>> aList = [3, 5, 7]
>>> aList[len(aList):]
[]
>>> aList[len(aList):] = [9]
>>> aList
[3, 5, 7, 9]
>>> aList[:3] = [1, 2, 3]
>>> aList
[1, 2, 3, 9]
>>> aList[:3] = []
>>> aList
[9]
>>> aList = list(range(10))
>>> aList
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> aList[::2] = [0]*(len(aList)//2)
>>> aList
[0, 1, 0, 3, 0, 5, 0, 7, 0, 9]
```

2.1.6 切片操作

- 使用del与切片结合来删除列表元素

```
>>> aList = [3,5,7,9,11]
>>> del aList[:3]
>>> aList
[9, 11]
```

2.1.6 切片操作

- 切片返回的是列表元素的浅拷贝（浅copy与深copy）

```
>>> aList = [3, 5, 7]
>>> bList = aList #bList与aList指向同一个内存
>>> bList
[3, 5, 7]
>>> bList[1] = 8
>>> aList
[3, 8, 7]
>>> aList == bList
True
>>> aList is bList
True
>>> id(aList)
19061816
>>> id(bList)
19061816
```

2.1.6 切片操作

```
>>> aList = [3, 5, 7]
>>> bList = aList[::] #浅复制
>>> aList == bList
True
>>> aList is bList
False
>>> id(aList) == id(bList)
False
>>> bList[1] = 8
>>> bList
[3, 8, 7]
>>> aList
[3, 5, 7]
>>> aList == bList
False
>>> aList is bList
False
>>> id(aList)
19061816
>>> id(bList)
11656168
```

2.1.7 列表排序

- 使用列表对象的sort方法进行原地排序，支持多种不同的排序方法

```
>>> aList = [3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> import random
>>> random.shuffle(aList)  #将序列的所有元素随机排序。
>>> aList
[3, 4, 15, 11, 9, 17, 13, 6, 7, 5]
>>> aList.sort()
>>> aList.sort(reverse = True)
>>> aList
[17, 15, 13, 11, 9, 7, 6, 5, 4, 3]
>>> aList.sort(key = lambda x:len(str(x)))
>>> aList
[9, 7, 6, 5, 4, 3, 17, 15, 13, 11]
```

2.1.7 列表排序

- 使用内置函数sorted对列表进行排序并返回新列表

```
>>> aList
[9, 7, 6, 5, 4, 3, 17, 15, 13, 11]
>>> sorted(aList)
[3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> sorted(aList,reverse = True)
[17, 15, 13, 11, 9, 7, 6, 5, 4, 3]
```

2.1.7 列表排序

- 使用列表对象的reverse方法将元素原地逆序

```
>>> aList = [3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> aList.reverse()
>>> aList
[17, 15, 13, 11, 9, 7, 6, 5, 4, 3]
```

2.1.7 列表排序

- 使用内置函数reversed方法对列表元素进行逆序排列并返回迭代对象

```
>>> aList = [3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> newList = reversed(aList)
>>> newList
<listreverseiterator object at 0x0000000003624198>
>>> list(newList)
[17, 15, 13, 11, 9, 7, 6, 5, 4, 3]
>>> for i in newList:
        print(i, end=' ')
```

无输出内容，迭代对象已遍历结束，需要重新创建迭代对象

```
>>> newList = reversed(aList)
>>> for i in newList:
        print(i, end=' ')
17 15 13 11 9 7 6 5 4 3
```

2.1.8 用于序列操作的常用内置函数

- cmp(列表1，列表2)：对两个列表进行比较，若第一个列表大于第二个，则结果为1，相反则为-1，元素完全相同则结果为0，类似于==运算符，和is、is not不一样。

```
>>> (1, 2, 3) < (1, 2, 4)
True
>>> cmp((1, 2, 3) , (1, 2, 4))
-1
>>> [1, 2, 3] < [1, 2, 4]
True
>>> 'ABC' < 'C' < 'Pascal' < 'Python'
True
>>> cmp( 'Pascal', 'Python')
-1
```

2.1.8 用于序列操作的常用内置函数

```
>>> (1, 2, 3, 4) < (1, 2, 4)
True
>>> (1, 2) < (1, 2, -1)
True
>>> cmp((1, 2), (1, 2, -1))
-1
>>> (1, 2, 3) == (1.0, 2.0, 3.0)
True
>>> cmp((1, 2, 3), (1.0, 2.0, 3.0))
0
>>> (1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'), 4)
True
>>> cmp('a', 'A')
1
>>> 'a'>'A'
True
```

2.1.8 用于序列操作的常用内置函数

- 在Python 3.x中，不再支持cmp()函数，可以直接使用关系运算符来比较数值或序列的大小，也可以使用对象的“__le__()”及其相关方法，或者也可以使用其他写法来模拟Python 2.x的内置函数cmp()。

```
>>> a = [1, 2]
>>> b = [1, 2, 3]
>>> (a>b)-(a<b)
-1
>>> a.__le__(b)
True
>>> a.__gt__(b)
False
>>> a>b
False
>>> a<b
True
```

2.1.8 用于序列操作的常用内置函数

- len(列表)：返回列表中的元素个数，同样适用于元组、字典、字符串等等。
- max(列表)、 min(列表)：返回列表中的最大或最小元素，同样适用于元组、range。
- sum(列表)：对数值型列表的元素进行求和运算，对非数值型列表运算则出错，同样适用于元组、range。

2.1.8 用于序列操作的常用内置函数

- zip(列表1,列表2,...):将多个列表对应位置元素组合为元组，并返回包含这些元组的列表。

```
>>> aList = [1,2,3]
>>> bList = [4,5,6]
>>> cList = [7,8,9]
>>> dList = zip(aList, bList, cList)
>>> dList
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

- 而在Python 3.5.1中则需要这样使用：

```
>>> aList = [1, 2, 3]
>>> bList = [4, 5, 6]
>>> cList = zip(a, b)
>>> cList
<zip object at 0x0000000003728908>
>>> list(cList)
[(1, 4), (2, 5), (3, 6)]
```

2.1.8 用于序列操作的常用内置函数

- enumerate(列表):枚举列表元素，返回枚举对象，其每个元素为包含下标和值的元组。该函数对元组、字符串同样有效。

```
>>> for item in enumerate(dList):
        print(item)
(0, (1, 4, 7))
(1, (2, 5, 8))
(2, (3, 6, 9))
```

2.1.9 列表推导式

- 列表推导式使用非常简洁的方式来快速生成满足特定需求的列表，代码具有非常强的可读性。例如：

```
>>> aList = [x*x for x in range(10)]
```

相当于

```
>>> aList = []
>>> for x in range(10):
        aList.append(x*x)
```

2.1.9 列表推导式

- 使用列表推导式实现嵌套列表的平铺

```
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- 列出当前文件夹下所有Python源文件

```
>>> import os
>>> [filename for filename in os.listdir('.') if filename.endswith('.py')]
```

- 过滤不符合条件的元素

```
>>> aList = [-1,-4,6,7.5,-2.3,9,-11]
>>> [i for i in aList if i>0]
[6, 7.5, 9]
```

2.1.9 列表推导式

例如，已知有一个包含一些同学成绩的字典，计算成绩的最高分、最低分、平均分，并查找所有最高分同学，代码可以这样编写：

```
>>> scores = {"Zhang San": 45, "Li Si": 78, "Wang Wu": 40, "Zhou Liu": 96, "Zhao Qi": 65, "Sun Ba": 90, "Zheng Jiu": 78, "Wu Shi": 99, "Dong Shiyi": 60}
>>> highest = max(scores.values())
>>> lowest = min(scores.values())
>>> highest
99
>>> lowest
40
>>> average = sum(scores.values())*1.0/len(scores)
>>> average
72.33333333333333
>>> highestPerson = [name for name, score in scores.items() if score == highest]
>>> highestPerson
['Wu Shi']
```

2.1.9 列表推导式

- 在列表推导式中使用多个循环，实现多序列元素的任意组合，并且可以结合条件语句过滤特定元素

```
>>> [(x, y) for x in range(3) for y in range(3)]
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
>>> [(x, y) for x in [1, 2, 3] for y in [3, 1, 4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

2.1.9 列表推导式

- 使用列表推导式实现矩阵转置

```
>>>matrix = [ [1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

- 也可以使用内置函数来实现矩阵转置

```
>>>list(zip(*matrix)) #可变长度参数，序列解包为多个参数
 [(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

2.1.9 列表推导式

- 列表推导式中可以使用函数或复杂表达式

```
>>> def f(v):
        if v%2 == 0:
            v = v**2
        else:
            v = v+1
        return v
>>> print([f(v) for v in [2, 3, 4, -1] if v>0])
[4, 4, 16]
>>> print([v**2 if v%2 == 0 else v+1 for v in [2, 3, 4, -1] if v>0])
[4, 4, 16]
```

2.1.9 列表推导式

- 列表推导式支持文件对象迭代

```
>>> fp = open('C:\install.log', 'r')
>>> print([line for line in fp])
>>> fp.close()
```

2.1.9 列表推导式

- 使用列表推导式生成100以内的所有素数

```
>>> [ p for p in range(2, 100) if 0 not in \
        [ p% d for d in range(2, int(sqrt(p))+1)] ]
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47,
53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

2.2 元组

- 元组和列表类似，但属于不可变序列，元组一旦创建，用任何方法都不可以修改其元素。
- 元组的定义方式和列表相同，但定义时所有元素是放在一对圆括号“（”和“）”中，而不是方括号中。

2.2.1 元组创建与删除

- 使用“=”将一个元组赋值给变量

```
>>>a_tuple = ('a', )
>>> a_tuple
('a')
>>>a_tuple = ('a', 'b', 'mpilgrim', 'z', 'example')
>>> a_tuple
('a', 'b', 'mpilgrim', 'z', 'example')
>>> a=3
>>> a
3
>>> a=3,
>>> a
(3,)
>>> x = () #空元组
>>> x
()
```

2.2.1 元组创建与删除

- 使用tuple函数将其他序列转换为元组

```
>>> print tuple('abcdefg')
('a', 'b', 'c', 'd', 'e', 'f', 'g')
>>> aList
[-1, -4, 6, 7.5, -2.3, 9, -11]
>>> tuple(aList)
(-1, -4, 6, 7.5, -2.3, 9, -11)
>>> s = tuple() #空元组
>>> s
()
```

- 使用del删除元组对象，不能删除元组元素

2.2.2 元组与列表的区别

- 元组中的数据一旦定义就不允许更改。
- 元组没有append()、extend()和insert()等方法，无法向元组中添加元素；
- 元组没有remove()或pop()方法，也无法对元组元素进行del操作，不能从元组中删除元素。
- 内建的tuple( )函数接受一个列表参数，并返回一个包含同样元素的元组，而list( )函数接受一个元组参数并返回一个列表。从效果上看，tuple( )冻结列表，而list( )融化元组。

2.2.2 元组的优点

- 元组的速度比列表更快。如果定义了一系列常量值，而所需做的仅是对它进行遍历，那么一般使用元组而不用列表。
- 元组对不需要改变的数据进行“写保护”将使得代码更加安全。
- 一些元组可用作字典键（特别是包含字符串、数值和其它元组这样的不可变数据的元组）。列表永远不能当做字典键使用，因为列表不是不可变的。

2.2.3 序列解包

- 可以使用序列解包功能对多个变量同时赋值

```
v_tuple = (False, 3.5, 'exp')
>>> (x, y, z) = v_tuple
```

- 序列解包对于列表和字典同样有效

```
>>> a=[1,2,3]
>>> b,c,d=a
>>> s={'a':1,'b':2,'c':3}
>>> b,c,d=s
>>> b
'a'
>>> c
'c'
>>> d
'b'
```

2.2.3 序列解包

```
>>> keys=['a','b','c','d']
>>> values=[1,2,3,4]
>>> for k,v in zip(keys,values):
        print(k,v)

a 1
b 2
c 3
d 4
```

2.2.3 序列解包

```
>>> aList = [1,2,3]
>>> bList = [4,5,6]
>>> cList = [7,8,9]
>>> dList = zip(aList, bList, cList)
>>> for index, value in enumerate(dList):
        print(index, ':', value)
0 : (1, 4, 7)
1 : (2, 5, 8)
2 : (3, 6, 9)
```

2.2.4 生成器推导式

- 生成器推导式与列表推导式非常接近，只是生成器推导式使用圆括号而不是列表推导式所使用的方括号。
- 与列表推导式不同的是，生成器推导式的结果是一个生成器对象，而不是列表，也不是元组。
- 生成器对象类似于迭代器对象，具有惰性求值的特点，只在需要时返回元素，比列表推导式具有更高的效率，占用空间小。
- 使用生成器对象的元素时，可以根据需要将其转化为列表或元组，也可以使用生成器对象的next()方法（Python 2.x）或__next__()方法（Python 3.x）进行遍历，或者直接将其作为迭代器对象来使用。
- 不管用哪种方法访问其元素，当所有元素访问结束以后，如果需要重新访问其中的元素，必须重新创建该生成器对象。

2.2.4 生成器推导式

```
>>> g=((i+2)**2 for i in range(10))
>>> g
<generator object <genexpr> at 0x02B15C60>
>>>tuple(g)
(4, 9, 16, 25, 36, 49, 64, 81, 100, 121)
>>> tuple(g)
()
>>> g=((i+2)**2 for i in range(10))
>>> g.next()    #在python 3中可改为__next__()
4
>>> g.next()
9
>>> g.next()
16
>>> g.next()
25
```

2.3 字典

- 字典是键值对的无序可变集合。
- 定义字典时，每个元素的键和值用冒号分隔，元素之间用逗号分隔，所有的元素放在一对大括号“｛”和“｝”中。
- 字典中的每个元素包含两部分：键和值，向字典添加一个键的同时，必须为该键增添一个值。
- 字典中的键可以为任意不可变数据，比如整数、实数、复数、字符串、元组等等。
- 字典中的键不允许重复。

2.3.1 字典创建与删除

- 使用=将一个字典赋值给一个变量

```
>>> a_dict = {'server': 'db.diveintopython3.org',
    'database': 'mysql'}
>>> a_dict
{'database': 'mysql', 'server': 'db.diveintopython3.org'}
>>> x = {} #空字典
>>> x
{}
```

2.3.1 字典创建与删除

- 使用dict利用已有数据创建字典：

```
>>> keys=['a','b','c','d']
>>> values=[1,2,3,4]
>>> dictionary=dict(zip(keys,values))
>>> dictionary
{'a': 1, 'c': 3, 'b': 2, 'd': 4}
>>> x = dict() #空字典
>>> x
{}
```

- 使用dict根据给定的键、值创建字典

```
>>> d=dict(name='Dong',age=37)
>>> d
{'age': 37, 'name': 'Dong'}
```

2.3.1 字典创建与删除

- 以给定内容为键，创建值为空的字典

```
>>> adict=dict.fromkeys(['name','age','sex'])
>>> adict
{'age': None, 'name': None, 'sex': None}
```

- 使用del删除整个字典

2.3.2 字典元素的读取

- 以键作为下标可以读取字典元素，若键不存在则抛出异常

```
>>> aDict={'name':'Dong', 'sex':'male', 'age':37}
>>> aDict['name']
'Dong'
>>> aDict['tel']
Traceback (most recent call last):
  File "<pyshell#53>", line 1, in <module>
    aDict['tel']
KeyError: 'tel'
```

2.3.2 字典元素的读取

- 使用字典对象的get方法获取指定键对应的值，并且可以在键不存在的时候返回指定值。

```
>>> print(aDict.get('address'))
None
>>> print(aDict.get('address', 'SDIBT'))
SDIBT
>>> aDict['score'] = aDict.get('score',[])
>>> aDict['score'].append(98)
>>> aDict['score'].append(97)
>>> aDict
{'age': 37, 'score': [98, 97], 'name': 'Dong', 'sex': 'male'}
```

2.3.2 字典元素的读取

- 使用字典对象的items方法可以返回字典的键、值对列表
- 使用字典对象的keys方法可以返回字典的键列表
- 使用字典对象的values方法可以返回字典的值列表

2.3.2 字典元素的读取

```
>>> aDict={'name':'Dong', 'sex':'male', 'age':37}
>>> for item in aDict.items():
        print(item)
('age', 37)
('name', 'Dong')
('sex', 'male')
>>> for key in aDict:
        print(key)
age
name
sex
>>> for key, value in aDict.items():
        print(key, value)
age 37
name Dong
sex male
>>> aDict.keys()
['age', 'name', 'sex']
>>> aDict.values()
[37, 'Dong', 'male']
```

2.3.3 字典元素的添加与修改

- 当以指定键为下标为字典赋值时，若键存在，则可以修改该键的值；若不存在，则表示添加一个键、值对。

```
>>> aDict['age'] = 38
>>> aDict
{'age': 38, 'name': 'Dong', 'sex': 'male'}
>>> aDict['address'] = 'SDIBT'
>>> aDict
{'age': 38, 'address': 'SDIBT', 'name': 'Dong', 'sex': 'male'}
```

2.3.3 字典元素的添加与修改

- 使用字典对象的update方法将另一个字典的键、值对添加到当前字典对象

```
>>> aDict
{'age': 37, 'score': [98, 97], 'name': 'Dong', 'sex': 'male'}
>>> aDict.items()
[('age', 37), ('score', [98, 97]), ('name', 'Dong'), ('sex', 'male')]
>>> aDict.update({'a':'a','b':'b'})
>>> aDict
{'a': 'a', 'score': [98, 97], 'name': 'Dong', 'age': 37, 'b': 'b',
    'sex': 'male'}
```

2.3.3 字典元素的添加与修改

- 使用del删除字典中指定键的元素
- 使用字典对象的clear方法来删除字典中所有元素
- 使用字典对象的pop 方法删除并返回指定键的元素
- 使用字典对象的popitem方法删除并返回字典中的一个元素

2.3.4 字典应用案例

- 下面的代码首先生成包含1000个随机字符的字符串，然后统计每个字符的出现次数。

```
>>> import string
>>> import random
>>> x = string.ascii_letters + string.digits + string.punctuation
>>> x
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789!"#
    $%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
>>> y = [random.choice(x) for i in range(1000)]
>>> z = ''.join(y)
>>> d = dict()
>>> for ch in z:
     d[ch] = d.get(ch, 0) + 1
```

2.3.4 字典应用案例

- 也可以使用collections模块的defaultdict类来实现该功能。

```
>>> import string
>>> import random
>>> x = string.ascii_letters + string.digits + string.punctuation
>>> y = [random.choice(x) for i in range(1000)]
>>> z = ''.join(y)
>>> from collections import defaultdict
>>> frequences = defaultdict(int)
>>> frequences
defaultdict(<type 'int'>, {})
>>> for item in z:
        frequences[item] += 1
>>> frequences.items()
```

2.3.4 字典应用案例

- 使用collections模块的Counter类可以快速实现这个功能，并且能够满足其他需要，例如查找出现次数最多的元素。下面的代码演示了Counter类的用法：

```
>>> from collections import Counter
>>> frequences = Counter(z)
>>> frequences.items()
>>> frequences.most_common(1)
[('A', 22)]
>>> frequences.most_common(3)
[('A', 22), (';', 18), ('``', 17)]
```

2.3.4 字典应用案例

- Counter对象用法示例

```
>>> # Tally occurrences of words in a list
>>> cnt = Counter()
>>> for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:
        cnt[word] += 1
>>> cnt
Counter({'blue': 3, 'red': 2, 'green': 1})
>>> # Find the ten most common words in Hamlet
>>> import re
>>> words = re.findall(r'\w+', open('hamlet.txt').read().lower())
>>> Counter(words).most_common(10)
[('the', 1143), ('and', 966), ('to', 762), ('of', 669), ('i', 631),
('you', 554),  ('a', 546), ('my', 514), ('hamlet', 471), ('in', 451)]
```

2.3.5 有序字典

- Python内置字典是无序的，前面的示例很好地说明了这个问题。如果需要一个可以记住元素插入顺序的字典，可以使用collections.OrderedDict。例如下面的代码：

```
>>> x = dict() #无序字典
>>> x['a'] = 3
>>> x['b'] = 5
>>> x['c'] = 8
>>> x
{'b': 5, 'c': 8, 'a': 3}
>>> import collections
>>> x = collections.OrderedDict() #有序字典
>>> x['a'] = 3
>>> x['b'] = 5
>>> x['c'] = 8
>>> x
OrderedDict([('a', 3), ('b', 5), ('c', 8)])
```

2.3.6 字典推导式

```
>>> s = {x:x.strip() for x in ('  he  ', 'she    ', '    I')}
>>> s
{'  he  ': 'he', '    I': 'I', 'she    ': 'she'}
>>> for k, v in s.items():
        print(k, ':', v)

  he   : he
    I : I
she     : she
```

- globals()返回包含当前作用域内所有全局变量和值的字典
- locals()返回包含当前作用域内所有局部变量和值的字典

2.4 集合

- 集合是无序可变对象，使用一对大括号界定，元素不可重复。

2.4.1 集合的创建与删除

- 直接将集合赋值给变量

```
>>> a={3,5}
>>> a.add(7)
>>> a
set([3, 5, 7])
```

- 使用set将其他类型数据转换为集合

```
>>> a_set=set(range(8,14))
>>> a_set
set([8, 9, 10, 11, 12, 13])
>>> b_set=set([0,1,2,3,0,1,2,3,7,8])
>>> b_set
set([0, 1, 2, 3, 7, 8])
>>> c_set = set() #空集合
>>> c_set
set()
```

- 使用del删除整个集合

2.4.1 集合的创建与删除

- 当不再使用某个集合时，可以使用del命令删除整个集合；
- 也可以使用集合对象的pop()方法弹出并删除其中一个元素；
- 或者使用集合对象的remove()方法直接删除指定元素；
- 以及使用集合对象的clear()方法清空集合删除所有元素。

```
>>> a={1, 4, 2, 3}
>>> a.pop()
1
>>> a
set([2, 3, 4])
>>> a.pop()
2
>>> a
set([3, 4])
>>> a.add(2)
>>> a
set([2, 3, 4])
>>> a.remove(3) #删除指定元素
>>> a
set([2, 4])
>>> a.pop(2) #pop()方法不接收参数
TypeError: pop() takes no arguments (1 given)
```

2.4.2 集合操作

- Python集合支持交集、并集、差集等运算

```
>>> a_set.union(b_set)
set([0, 1, 2, 3, 7, 8, 9, 10, 11, 12, 13])
>>> a_set
set([8, 9, 10, 11, 12, 13])
>>> b_set
set([0, 1, 2, 3, 7, 8])
>>> a_set&b_set
set([8])
>>> a_set.intersection(b_set)
set([8])
>>> a_set.difference(b_set)
set([9, 10, 11, 12, 13])
>>> a_set.symmetric_difference(b_set)
set([0, 1, 2, 3, 7, 9, 10, 11, 12, 13])
>>> a_set^b_set
set([0, 1, 2, 3, 7, 9, 10, 11, 12, 13])
```

2.4.2 集合操作

- 使用集合快速提取序列中单一元素

```
>>> import random
>>> listRandom = [random.choice(range(10000)) for i in range(100)]
>>> noRepeat = []
>>> for i in listRandom :
        if i not in noRepeat :
                noRepeat.append(i)
>>> len(listRandom)
>>> len(noRepeat)
#以下使用集合的方法
>>>newSet = set(listRandom)
```

2.4.3 集合推导式

```
>>> s = {x.strip() for x in ('  he  ', 'she    ', '    I')}
>>> s
{'I', 'she', 'he'}
```

2.5 再谈内置方法sorted()

- 列表对象提供了sort()方法支持原地排序，而内置函数sorted()返回新的列表，并不对原列表进行任何修改。除此之外，sorted()方法还可以对元组、字典进行排序，并且借助于其key和cmp参数（Python 3.x的sorted()方法没有cmp参数）可以实现更加复杂的排序。
- 需要注意的是，Python 2.x中内置方法sorted()的cmp参数会被处理多次，而key参数只会被处理一次，具有更高的速度。

2.5 再谈内置方法sorted()

```
>>> persons = [{'name':'Dong', 'age':37}, \
              {'name':'Zhang', 'age':40}, \
              {'name':'Li', 'age':50}, \
              {'name':'Dong', 'age':43}]
>>> print(persons)
[{'age': 37, 'name': 'Dong'}, {'age': 40, 'name': 'Zhang'},
{'age': 50, 'name': 'Li'}, {'age': 43, 'name': 'Dong'}]
#使用key来指定排序依据，先按姓名升序排序，姓名相同的按年龄降序排序
>>> print(sorted(persons, key=lambda x:(x['name'], -x['age'])))
[{'age': 43, 'name': 'Dong'}, {'age': 37, 'name': 'Dong'},
{'age': 50, 'name': 'Li'}, {'age': 40, 'name': 'Zhang'}]
```

2.5 再谈内置方法sorted()

```
>>> from timeit import Timer
#在Python 2.7.11中比较sorted()方法的key参数与cmp参数对排序速度的影响
>>> Timer(stmt='sorted(xs, key=lambda x: x[1])',  \
            setup='xs=range(100); xs=zip(xs, xs); ').timeit(100000)
1.930681312803035
>>> Timer(stmt='sorted(xs, cmp=lambda a, b: cmp(a[1], b[1]))', \
            setup='xs=range(100);xs=zip(xs, xs); ').timeit(100000)
3.056278670527241 6
```

timeit.timeit(stmt='pass', setup='pass', timer=<defaulttimer>, number=1000000)
返回：
　返回执行stmt这段代码number遍所用的时间，单位为秒，float型
参数：
　stmt：要执行的那段代码
　setup：执行代码的准备工作,初始化代码或构建环境导入语句,不计入时间，一般是import之类的语句
　timer：这个在win32下是time.clock()，linux下是time.time()，默认的，不用管
　number：要执行stmt多少遍

Timer类，Timer类里面的函数跟上面的函数是一模一样的
class timeit.Timer(stmt='pass', setup='pass',timer=<timer function>)
Timer.timeit(number=1000000)

2.5 再谈内置方法sorted()

```
>>> phonebook = {'Linda':'7750', 'Bob':'9345', 'Carol':'5834'}
>>> from operator import itemgetter
>>> sorted(phonebook.items(), key=itemgetter(1)) #按字典中元素值进行排序
[('Carol', '5834'), ('Linda', '7750'), ('Bob', '9345')]
>>> sorted(phonebook.items(), key=itemgetter(0)) #按字典中元素的键进行排序
[('Bob', '9345'), ('Carol', '5834'), ('Linda', '7750')]
```

2.5 再谈内置方法sorted()

```
>>> from operator import itemgetter
>>> gameresult = [['Bob', 95.0, 'A'], ['Alan', 86.0, 'C'], \
                  ['Mandy', 83.5, 'A'], ['Rob', 89.3, 'E']]
>>> sorted(gameresult, key=itemgetter(0, 1)) #按姓名升序，姓名相同按分数升序排序
[['Alan', 86.0, 'C'], ['Bob', 95.0, 'A'], ['Mandy', 83.5, 'A'], ['Rob', 89.3, 'E']]
>>> sorted(gameresult, key=itemgetter(1, 0)) #按分数升序，分数相同的按姓名升序排序
[['Mandy', 83.5, 'A'], ['Alan', 86.0, 'C'], ['Rob', 89.3, 'E'], ['Bob', 95.0, 'A']]
>>> sorted(gameresult, key=itemgetter(2, 0)) #按等级升序，等级相同的按姓名升序排序
[['Bob', 95.0, 'A'], ['Mandy', 83.5, 'A'], ['Alan', 86.0, 'C'], ['Rob', 89.3, 'E']]
```

2.5 再谈内置方法sorted()

```
>>> from operator import itemgetter
>>> gameresult = [{'name':'Bob', 'wins':10, 'losses':3, 'rating':75.0}, \
        {'name':'David', 'wins':3, 'losses':5, 'rating':57.0}, \
        {'name':'Carol', 'wins':4, 'losses':5, 'rating':57.0}, \
        {'name':'Patty', 'wins':9, 'losses':3, 'rating':72.8}]
>>> sorted(gameresult, key=itemgetter('wins', 'name'))
#按’wins’升序，该值相同的按’name’升序排序
[{'wins': 3, 'rating': 57.0, 'name': 'David', 'losses': 5}, {'wins': 4, 'rating': 57.0, 'name': 'Carol', 'losses': 5}, {'wins': 9, 'rating': 72.8, 'name': 'Patty', 'losses': 3}, {'wins': 10, 'rating': 75.0, 'name': 'Bob', 'losses': 3}]
```

2.5 再谈内置方法sorted()

- 以下代码演示如何根据另外一个列表的值来对当前列表元素进行排序

```
>>> list1 = ["what", "I'm", "sorting", "by"]
>>> list2 = ["something", "else", "to", "sort"]
>>> pairs = zip(list1, list2)
>>> pairs = sorted(pairs)
>>> pairs
[("I'm", 'else'), ('by', 'sort'), ('sorting', 'to'), ('what', 'something')]
>>> result = [x[1] for x in pairs]
>>> result
['else', 'sort', 'to', 'something']
```

2.6 其他数据结构

- 在Python中还有其他一些常用的数据结构，如堆、栈、队列、树、图等等。
- 有些结构Python已经提供，而有些则需要自己利用基本数据结构来实现。

2.6.1 堆

```
>>> import heapq
>>> import random
>>> data=range(10)
>>> data
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> random.choice(data)
9
>>> random.choice(data)
1
>>> random.choice(data)
>>> random.shuffle(data)
>>> data
[6, 1, 3, 4, 9, 0, 5, 2, 8, 7]
>>> heap=[]
>>> for n in data:
    heapq.heappush(heap,n)
>>> heap
[0, 2, 1, 4, 7, 3, 5, 6, 8, 9]
```

2.6.1 堆

```
>>> heapq.heappush(heap,0.5)
>>> heap
[0, 0.5, 1, 4, 2, 3, 5, 6, 8, 9, 7]
>>> heapq.heappop(heap)
0
>>> heapq.heappop(heap)
0.5
>>> heapq.heappop(heap)
1
>>> myheap=[1,2,3,5,7,8,9,4,10,333]
>>> heapq.heapify(myheap)  #将list类型转化为heap,在线性时间内,重新排列列表。
>>> myheap
[1, 2, 3, 4, 7, 8, 9, 5, 10, 333]
>>> heapq.heapreplace(myheap,6)  #删除现有元素并将其替换为一个新值。
1
>>> myheap
[2, 4, 3, 5, 7, 8, 9, 6, 10, 333]
>>> heapq.nlargest(3, myheap) #返回前3个最大的元素
[333, 10, 9]
>>> heapq.nsmallest(3, myheap) #返回前3个最小的元素
[2, 3, 4]
```

2.6.2 队列

```
>>> import Queue  #queue in python3
>>> q=Queue.Queue()
>>> q.put(0)
>>> q.put(1)
>>> q.put(2)
>>> q.queue
deque([0, 1, 2])
>>> q.get()
0
>>> q.queue
deque([1, 2])
>>> q.get()
1
>>> q.queue
deque([2])
```

2.6.2 队列

另外，Queue和queue模块还提供了“后进先出”队列和优先级队列，例如下面的Python 3.5.1代码所演示：

```
>>> import queue
>>> LiFoQueue = queue.LifoQueue(5)  #“后进先出”队列
>>> LiFoQueue.put(1)
>>> LiFoQueue.put(2)
>>> LiFoQueue.put(3)
>>> LiFoQueue.queue
[1, 2, 3]
>>> LiFoQueue.get()
3
>>> LiFoQueue.get()
2
>>> LiFoQueue.get()
1
```

2.6.2 队列

```
>>> import queue
>>> PriQueue = queue.PriorityQueue(5) #优先级队列
>>> PriQueue.put(3)
>>> PriQueue.queue
[3]
>>> PriQueue.put(5)
>>> PriQueue.queue
[3, 5]
>>> PriQueue.put(1)
>>> PriQueue.queue
[1, 5, 3]
>>> PriQueue.put(8)
>>> PriQueue.queue
[1, 5, 3, 8]
>>> PriQueue.get()
1
>>> PriQueue.get()
3
>>> PriQueue.get()
5
>>> PriQueue.get()
8
```

2.6.2 队列

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")
>>> queue.append("Graham")
 >>> queue.popleft()
'Eric'
>>> queue.popleft()
 'John'
>>> queue
deque(['Michael', 'Terry', 'Graham'])
```

2.6.2 队列

- 也可以使用列表自行定义队列

myQueue.py

2.6.3 栈

- 栈是一种“后进先出（LIFO）”或“先进后出（FILO）”的数据结构，Python列表本身就可以实现栈结构的基本操作。例如，列表对象的append()方法是在列表尾部追加元素，类似于入栈操作；pop()方法默认是弹出并返回列表的最后一个元素，类似于出栈操作。但是直接使用Python列表对象模拟栈操作并不是很方便，例如当列表为空时再执行pop()出栈操作时则会抛出一个不很友好的异常；另外，也无法限制栈的大小。

2.6.3 栈

- 直接使用列表来实现栈结构

```
>>> myStack = []
>>> myStack.append(3)
>>> myStack.append(5)
>>> myStack.append(7)
>>> myStack
[3, 5, 7]
>>> myStack.pop()
7
>>> myStack.pop()
5
>>> myStack.pop()
3
>>> myStack.pop()
出错
```

2.6.3 栈

- 使用列表实现栈结构

完整代码见Stack.py

2.6.3 栈

```
>>> import Stack
>>> x = Stack.Stack()
>>> x.push(1)
>>> x.push(2)
>>> x.show()
[1, 2]
>>> x.pop()
2
>>> x.show()
[1]
>>> x.showRemainderSpace()
Stack can still PUSH  9  elements.
>>> x.isEmpty()
False
>>> x.isFull()
False
```

2.6.4 链表

- 可直接使用列表来实现：

```
>>> linkTable = []
>>> linkTable.append(3)
>>> linkTable.append(5)
>>> linkTable
[3, 5]
>>> linkTable.insert(1,4)
>>> linkTable
[3, 4, 5]
>>> linkTable.remove(linkTable[1])
>>> linkTable
[3, 5]
```

2.6.5 二叉树

- 使用代码中的类BinaryTree创建的对象不仅支持二叉树的创建以及前序遍历、中序遍历与后序遍历等三种常用的二叉树节点遍历方式，还支持二叉树中任意“子树”的遍历。

BinaryTree.py

2.6.5 二叉树

```
>>> import BinaryTree
>>> root = BinaryTree.BinaryTree('root')
>>> firstRight = root.insertRightChild('B')
>>> firstLeft = root.insertLeftChild('A')
>>> secondLeft = firstLeft.insertLeftChild('C')
>>> thridRight = secondLeft.insertRightChild('D')
>>> root.postOrder() #后序遍历
D C A B root
>>> root.preOrder() #前序遍历
root A C D B
>>> root.inOrder() #中序遍历
C D A root B
>>> firstLeft.inOrder() #遍历“子树”
C D A
>>> secondLeft.removeRightChild() #删除二叉树中的节点
>>> root.preOrder()
root A C B
```

2.6.6 有向图

DirectedGraph.py

谢谢Q/A