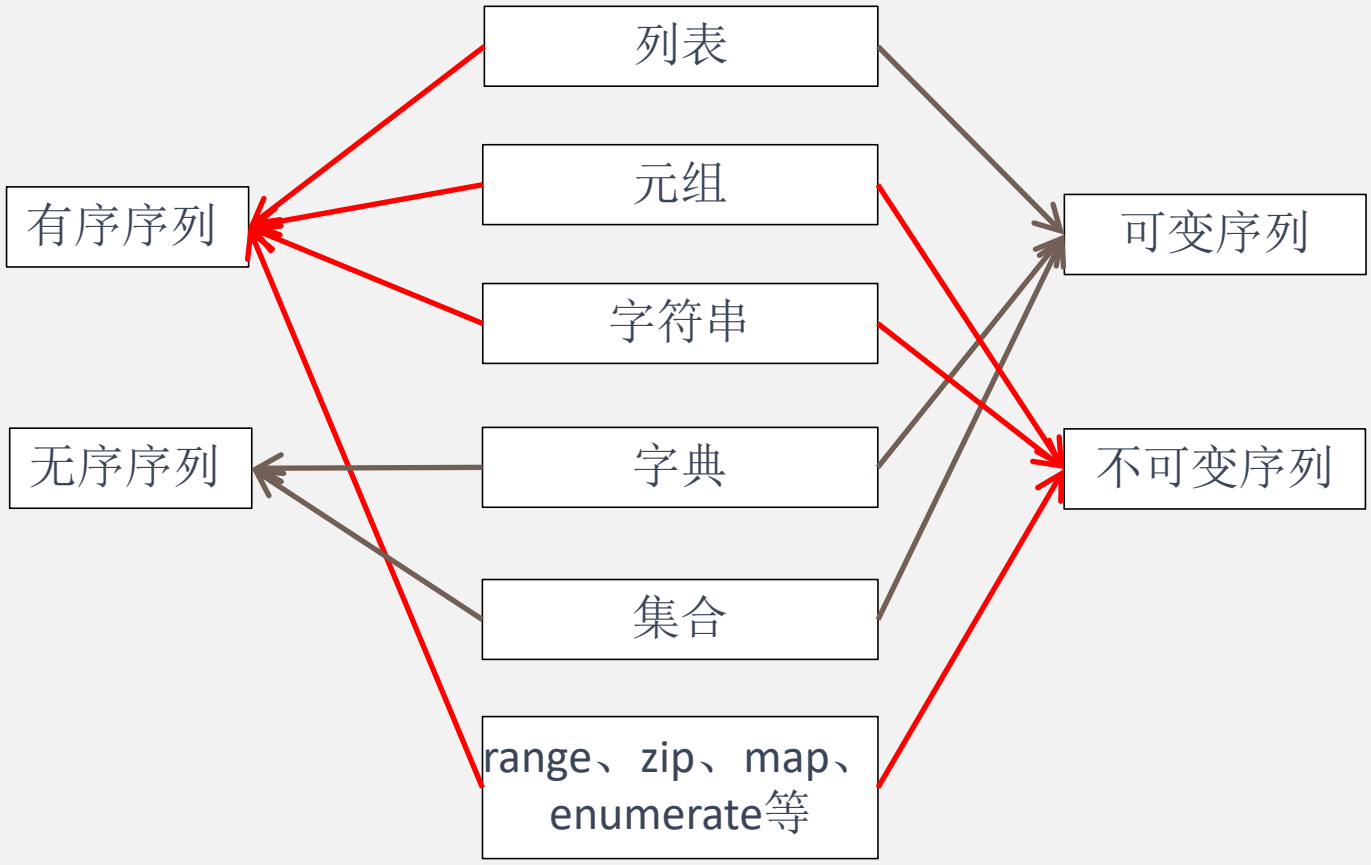
桂林理工大学
GUILIN UNIVERSITY OF TECHNOLOGY

第三章

详解Python序列结构

第三章 详解python序列结构



第三章 详解python序列结构

- 3.1 列表：打了激素的数组
- 3.2元组：轻量级列表
- 3.3字典：反映对应关系的映射类型
- 3.4 集合
- 3.5序列解包的多种形式和用法

3.1 列表：打了激素的数组

- 列表（list）是最重要的Python内置对象之一，是包含若干元素的有序连续内存空间。当列表增加或删除元素时，列表对象自动进行内存的扩展或收缩，从而保证相邻元素之间没有缝隙。Python列表的这个内存自动管理功能可以大幅度减少程序员的负担，但插入和删除非尾部元素时涉及到列表中大量元素的移动，会严重影响效率。
- 在非尾部位置插入和删除元素时会改变该位置后面的元素在列表中的索引，这对于某些操作可能会导致意外的错误结果。
- 除非确实有必要，否则应尽量从列表尾部进行元素的追加与删除操作。

3.1 列表：打了激素的数组

- 在形式上，列表的所有元素放在一对方括号[]中，相邻元素之间使用逗号分隔。
- 在Python中，同一个列表中元素的数据类型可以各不相同，可以同时包含整数、实数、字符串等基本类型的元素，也可以包含列表、元组、字典、集合、函数以及其他任意对象。
- 如果只有一对方括号而没有任何元素则表示空列表。

```
[10, 20, 30, 40]
['crunchy frog', 'ram bladder', 'lark vomit']
['spam', 2.0, 5, [10, 20]]
[['file1', 200,7], ['file2', 260,9]]
[{3}, {5:6}, (1, 2, 3)]
```

3.1 列表：打了激素的数组

- Python采用基于值的自动内存管理模式，变量并不直接存储值，而是存储值的引用或内存地址，这也是python中变量可以随时改变类型的重要原因。同理，Python列表中的元素也是值的引用，所以列表中各元素可以是不同类型的数据。
- 需要注意的是，列表的功能虽然非常强大，但是负担也比较重，开销较大，在实际开发中，最好根据实际的问题选择一种合适的数据类型，要尽量避免过多使用列表。

3.1.1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。

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

3.1.1列表创建与删除

- 也可以使用list()函数把元组、range对象、字符串、字典、集合或其他可迭代对象转换为列表。

```
>>> list((3,5,7,9,11))                          #将元组转换为列表
[3, 5, 7, 9, 11]
>>> list(range(1, 10, 2))                       #将range对象转换为列表
[1, 3, 5, 7, 9]
>>> list('hello world')                         #将字符串转换为列表
['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd']
>>> list({3,7,5})                               #将集合转换为列表
[3, 5, 7]
>>> list({'a':3, 'b':9, 'c':78})                #将字典的“键”转换为列表
['a', 'c', 'b']
>>> list({'a':3, 'b':9, 'c':78}.items())#将字典的“键:值”对转换为列表
[('b', 9), ('c', 78), ('a', 3)]
>>> x = list()                                    #创建空列表
```

3.1.1列表创建与删除

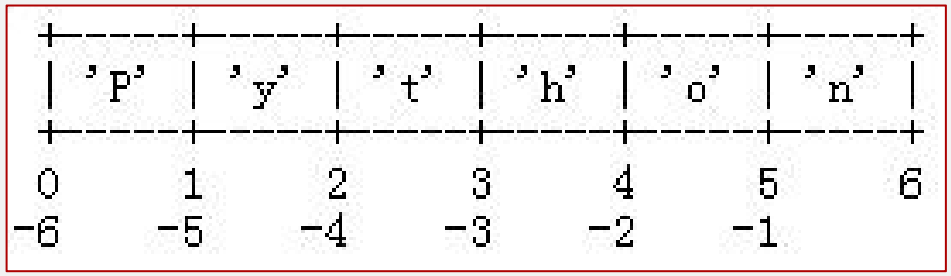
- 当一个列表不再使用时，可以使用del命令将其删除，这一点适用于所有类型的Python对象。

```
>>> x = [1, 2, 3]
>>> del x                          #删除列表对象
>>> x                              #对象删除后无法再访问，抛出异常
NameError: name 'x' is not defined
```

3.1.2 列表元素访问

- 创建列表之后，可以使用整数作为下标来访问其中的元素，其中0表示第1个元素，1表示第2个元素，2表示第3个元素，以此类推；列表还支持使用负整数作为下标，其中-1表示最后1个元素，-2表示倒数第2个元素，-3表示倒数第3个元素，以此类推。

```
>>> x = list('Python')                    #创建类别对象
>>> x
['P', 'y', 't', 'h', 'o', 'n']
>>> x[0]                                  #下标为0的元素，第一个元素
'P'
>>> x[-1]                                 #下标为-1的元素，最后一个元素
'n'
```



3.1.3列表常用方法

方法	说明
append(x)	将x追加至列表尾部
extend(L)	将列表L中所有元素追加至列表尾部
insert(index, x)	在列表index位置处插入x，该位置后面的所有元素后移并且在列表中的索引加1，如果index为正数且大于列表长度则在列表尾部追加x，如果index为负数且小于列表长度的相反数则在列表头部插入元素x
remove(x)	在列表中删除第一个值为x的元素，该元素之后所有元素前移并且索引减1，如果列表中不存在x则抛出异常
pop([index])	删除并返回列表中下标为index的元素，如果不指定index则默认为-1，弹出最后一个元素；如果弹出中间位置的元素则后面的元素索引减1；如果index不是[-L, L]区间上的整数则抛出异常
clear()	清空列表，删除列表中所有元素，保留列表对象
index(x)	返回列表中第一个值为x的元素的索引，若不存在值为x的元素则抛出异常
count(x)	返回x在列表中的出现次数
reverse()	对列表所有元素进行原地逆序，首尾交换
sort(key=None, reverse=False)	对列表中的元素进行原地排序，key用来指定排序规则，reverse为False表示升序，True表示降序
copy()	返回列表的浅复制

3.1.3列表常用方法

（1）append()、insert()、extend()

append()用于向列表尾部追加一个元素，insert()用于向列表任意指定位置插入一个元素，extend()用于将另一个列表中的所有元素追加至当前列表的尾部。这3个方法都属于原地操作，不影响列表对象在内存中的起始地址。

```
>>> x = [1, 2, 3]
>>> id(x)                                  #查看对象的内存地址
50159368
>>> x.append(4)                            #在尾部追加元素
>>> x.insert(0, 0)                         #在指定位置插入元素
>>> x.extend([5, 6, 7])                    #在尾部追加多个元素
>>> x
[0, 1, 2, 3, 4, 5, 6, 7]
>>> id(x)                                  #列表在内存中的地址不变
50159368
```

3.1.3列表常用方法

（2）pop()、remove()、clear()

pop()用于删除并返回指定位置（默认是最后一个）上的元素；remove()用于删除列表中第一个值与指定值相等的元素；clear()用于清空列表中的所有元素。这3个方法也属于原地操作。

另外，还可以使用del命令删除列表中指定位置的元素，同样也属于原地操作。

```
>>> x = [1, 2, 3, 4, 5, 6, 7]
>>> x.pop()                          #弹出并返回尾部元素
7
>>> x.pop(0)                         #弹出并返回指定位置的元素
1
>>> x.clear()                        #删除所有元素
>>> x
[]
>>> x = [1, 2, 1, 1, 2]
>>> x.remove(2)                      #删除首个值为2的元素
>>> del x[3]                         #删除指定位置上的元素
>>> x
[1, 1, 1]
```

3.1.3列表常用方法

（3）count()、index()

列表方法count()用于返回列表中指定元素出现的次数；index()用于返回指定元素在列表中首次出现的位置，如果该元素不在列表中则抛出异常。

```
>>> x = [1, 2, 2, 3, 3, 3, 4, 4, 4, 4]
>>> x.count(3)                    #元素3在列表x中的出现次数
3
>>> x.count(5)                    #不存在，返回0
0
>>> x.index(2)                    #元素2在列表x中首次出现的索引
1
>>> x.index(5)                    #列表x中没有5，抛出异常
ValueError: 5 is not in list
```

3.1.3列表常用方法

（4）sort()、reverse()

列表对象的sort()方法用于按照指定的规则对所有元素进行排序；reverse()方法用于将列表所有元素逆序或翻转。

```
>>> x = list(range(11))                          #包含11个整数的列表
>>> import random
>>> random.shuffle(x)                            #把列表x中的元素随机乱序
>>> x
[6, 0, 1, 7, 4, 3, 2, 8, 5, 10, 9]
>>> x.sort(key=lambda item:len(str(item)), reverse=True)  #按转换成字符串以后的长度，降序排列
>>> x
[10, 6, 0, 1, 7, 4, 3, 2, 8, 5, 9]
>>> x.sort(key=str)                              #按转换为字符串后的大小，升序排序
>>> x
[0, 1, 10, 2, 3, 4, 5, 6, 7, 8, 9]
>>> x.sort()                                     #按默认规则排序
>>> x
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> x.reverse()                                  #把所有元素翻转或逆序
>>> x
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

3.1.4 列表对象支持的运算符

- 加法运算符+也可以实现列表增加元素的目的，但不属于原地操作，而是返回新列表，涉及大量元素的复制，效率非常低。使用复合赋值运算符+=实现列表追加元素时属于原地操作，与append()方法一样高效。

```
>>> x = [1, 2, 3]
>>> id(x)
53868168
>>> x = x + [4]                    #连接两个列表
>>> x
[1, 2, 3, 4]
>>> id(x)                          #内存地址发生改变
53875720
>>> x += [5]                       #为列表追加元素
>>> x
[1, 2, 3, 4, 5]
>>> id(x)                          #内存地址不变
53875720
```

3.1.4 列表对象支持的运算符

- 乘法运算符*可以用于列表和整数相乘，表示序列重复，返回新列表。
  运算符*=也可以用于列表元素重复，属于原地操作。

```
>>> x = [1, 2, 3, 4]
>>> id(x)
54497224
>>> x = x * 2                                  #元素重复，返回新列表
>>> x
[1, 2, 3, 4, 1, 2, 3, 4]
>>> id(x)                                      #地址发生改变
54603912
>>> x *= 2                                     #元素重复，原地进行
>>> x
[1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4]
>>> id(x)                                      #地址不变
54603912
```

3.1.4 列表对象支持的运算符

- 成员测试运算符in可用于测试列表中是否包含某个元素，查询时间随着列表长度的增加而线性增加，而同样的操作对于集合而言则是常数级的。

```
>>> 3 in [1, 2, 3]
True
>>> 3 in [1, 2, '3']
False
```


3.1.5 内置函数对列表的操作

- max()、min()函数用于返回列表中所有元素的最大值和最小值，
- sum()函数用于返回列表中所有元素之和；
- len()函数用于返回列表中元素个数，zip()函数用于将多个列表中元素重新组合为元组并返回包含这些元组的zip对象；
- enumerate()函数返回包含若干下标和值的迭代对象；
- map()函数把函数映射到列表上的每个元素，filter()函数根据指定函数的返回值对列表元素进行过滤；
- all()函数用来测试列表中是否所有元素都等价于True，any()用来测试列表中是否有等价于True的元素。
- 标准库functools中的reduce()函数以及标准库itertools中的compress()、groupby()、dropwhile()等大量函数也可以对列表进行操作。

3.1.5 内置函数对列表的操作

```
>>> x = list(range(11))          #生成列表
>>> import random
>>> random.shuffle(x)            #打乱列表中元素顺序
>>> x
[0, 6, 10, 9, 8, 7, 4, 5, 2, 1, 3]
>>> all(x)                       #测试是否所有元素都等价于True
False
>>> any(x)                       #测试是否存在等价于True的元素
True
>>> max(x)                       #返回最大值
10
>>> max(x, key=str)              #按指定规则返回最大值
9
>>> min(x)
0
```

3.1.5 内置函数对列表的操作

```
>>> sum(x)                          #所有元素之和
55
>>> len(x)                          #列表元素个数
11
>>> list(zip(x, [1]*11))            #多列表元素重新组合
[(0, 1), (6, 1), (10, 1), (9, 1), (8, 1), (7, 1), (4, 1), (5, 1), (2,
1), (1, 1), (3, 1)]
>>> list(zip(range(1,4)))           #zip()函数也可以用于一个序列或迭代对象
[(1,), (2,), (3,)]
>>> list(zip(['a', 'b', 'c'], [1, 2]))  #如果两个列表不等长，以短的为准
[('a', 1), ('b', 2)]
>>> enumerate(x)                    #枚举列表元素，返回enumerate对象
<enumerate object at 0x0000000030A9120>
>>> list(enumerate(x))              #enumerate对象可以转换为列表、元组、集合
[(0, 0), (1, 6), (2, 10), (3, 9), (4, 8), (5, 7), (6, 4), (7, 5), (8,
2), (9, 1), (10, 3)]
```

3.1.6 列表推导式语法与应用案例

- 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。
- 列表推导式语法形式为：

```
[expression for expr1 in sequence1 if condition1
            for expr2 in sequence2 if condition2
            for expr3 in sequence3 if condition3
            ...
            for exprN in sequenceN if conditionN]
```


3.1.6 列表推导式语法与应用案例

- 列表推导式在逻辑上等价于一个循环语句，只是形式上更加简洁。例如：

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

相当于

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


3.1.6 列表推导式语法与应用案例

```
>>> freshfruit = [' banana', ' loganberry ', 'passion fruit ']
>>> aList = [w.strip() for w in freshfruit]
```

- 等价于下面的代码

```
>>> aList = []
>>> for item in freshfruit:
    aList.append(item.strip())
```

3.1.6 列表推导式语法与应用案例

- 阿凡提与国王比赛下棋，国王说要是自己输了的话阿凡提想要什么他都可以拿得出来。阿凡提说那就要点米吧，棋盘一共64个小格子，在第一个格子里放1粒米，第二个格子里放2粒米，第三个格子里放4粒米，第四个格子里放8粒米，以此类推，后面每个格子里的米都是前一个格子里的2倍，一直把64个格子都放满。需要多少粒米呢？

```
>>> sum([2**i for i in range(64)])
18446744073709551615
>>> int('1'*64, 2)
18446744073709551615
```


3.1.6 列表推导式语法与应用案例

（1）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]
```

在这个列表推导式中有2个循环，其中第一个循环可以看作是外循环，执行的慢；而第二个循环可以看作是内循环，执行的快。上面代码的执行过程等价于下面的写法：

```
>>> vec = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> result = []
>>> for elem in vec:
    for num in elem:
        result.append(num)
>>> result
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

3.1.6 列表推导式语法与应用案例

（2）过滤不符合条件的元素

在列表推导式中可以使用if子句对列表中的元素进行筛选，只在结果列表中保留符合条件的元素。下面的代码可以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', '.pyw'))]
```

下面的代码用于从列表中选择符合条件的元素组成新的列表：

```
>>> aList = [-1, -4, 6, 7.5, -2.3, 9, -11]
>>> [i for i in aList if i>0]                    #所有大于0的数字
[6, 7.5, 9]
```

3.1.6 列表推导式语法与应用案例

- **问题解决**：已知有一个包含一些同学成绩的字典，现在需要计算所有成绩的最高分、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())                    #最高分
>>> lowest = min(scores.values())                     #最低分
>>> average = sum(scores.values()) / len(scores)      #平均分
>>> highest, lowest, average
(99, 40, 72.33333333333333)
>>> highestPerson = [name for name, score in scores.items() if
score == highest]
>>> highestPerson
['Wu Shi']
```

3.1.6 列表推导式语法与应用案例

（3）同时遍历多个列表或可迭代对象

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

对于包含多个循环的列表推导式，一定要清楚多个循环的执行顺序或“嵌套关系”。例如，上面第一个列表推导式等价于

```
>>> result = []
>>> for x in [1, 2, 3]:
    for y in [3, 1, 4]:
        if x != y:
            result.append((x,y))
>>> result
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

3.1.6 列表推导式语法与应用案例

（4）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]]
```

上面列表推导式的执行过程等价于下面的代码

```
>>> matrix = [ [1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
>>> result = []
>>> for i in range(len(matrix[0])):
    result.append([row[i] for row in matrix])
>>> result
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

3.1.6 列表推导式语法与应用案例

（5）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

>>> [f(v) for v in [2, 3, 4, -1] if v>0]
[4, 4, 16]
>>> [v**2 if v%2 == 0 else v+1 for v in [2, 3, 4, -1] if v>0]
[4, 4, 16]
>>> x = list(range(10))
>>> [item>5 for item in x]
[False, False, False, False, False, False, True, True, True,
True]
```

3.1.7切片操作的强大功能

- 在形式上，切片使用2个冒号分隔的3个数字来完成。

```
[start:end:step]
```

- ✓ 第一个数字start表示切片开始位置，默认为0；
- ✓ 第二个数字end表示切片截止（但不包含）位置（默认为列表长度）；
- ✓ 第三个数字step表示切片的步长（默认为1）。
- ✓ 当start为0时可以省略，当end为列表长度时可以省略，当step为1时可以省略，省略步长时还可以同时省略最后一个冒号。
- ✓ 当step为负整数时，表示反向切片，这时start应该在end的右侧才行。

3.1.7切片操作的强大功能

（1）使用切片获取列表部分元素

使用切片可以返回列表中部分元素组成的新列表。与使用索引作为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。

3.1.7切片操作的强大功能

```
>>> aList = [3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> aList[::]                    #返回包含原列表中所有元素的新列表
[3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> aList[::-1]                  #返回包含原列表中所有元素的逆序列表
[17, 15, 13, 11, 9, 7, 6, 5, 4, 3]
>>> aList[::2]                   #隔一个取一个，获取偶数位置的元素
[3, 5, 7, 11, 15]
>>> aList[1::2]                  #隔一个取一个，获取奇数位置的元素
[4, 6, 9, 13, 17]
>>> aList[3:6]                   #指定切片的开始和结束位置
[6, 7, 9]
```

3.1.7切片操作的强大功能

```
>>> aList[0:100]          #切片结束位置大于列表长度时，从列表尾部截断
[3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> aList[100]            #抛出异常，不允许越界访问
IndexError: list index out of range
>>> aList[100:]           #切片开始位置大于列表长度时，返回空列表
[]
>>> aList[-15:3]          #进行必要的截断处理
[3, 4, 5]
>>> len(aList)
10
>>> aList[3:-10:-1]       #位置3在位置-10的右侧，-1表示反向切片
[6, 5, 4]
>>> aList[3:-5]           #位置3在位置-5的左侧，正向切片
[6, 7]
```

3.1.7切片操作的强大功能

（2）使用切片为列表增加元素

可以使用切片操作在列表任意位置插入新元素，不影响列表对象的内存地址，属于原地操作。

```
>>> aList = [3, 5, 7]
>>> aList[len(aList):]
[]
>>> aList[len(aList):] = [9]          #在列表尾部增加元素
>>> aList[:0] = [1, 2]                #在列表头部插入多个元素
>>> aList[3:3] = [4]                  #在列表中间位置插入元素
>>> aList
[1, 2, 3, 4, 5, 7, 9]
```

3.1.7切片操作的强大功能

（3）使用切片替换和修改列表中的元素

```
>>> aList = [3, 5, 7, 9]
>>> aList[:3] = [1, 2, 3]    #替换列表元素，等号两边的列表长度相等
>>> aList
[1, 2, 3, 9]
>>> aList[3:] = [4, 5, 6] #切片连续，等号两边的列表长度可以不相等
>>> aList
[1, 2, 3, 4, 5, 6]
>>> aList[::2] = [0]*3              #隔一个修改一个
>>> aList
[0, 2, 0, 4, 0, 6]
>>> aList[::2] = ['a', 'b', 'c']     #隔一个修改一个
>>> aList
['a', 2, 'b', 4, 'c', 6]
```

3.1.7切片操作的强大功能

```
>>> aList[1::2] = range(3)                    #序列解包的用法
>>> aList
['a', 0, 'b', 1, 'c', 2]
>>> aList[1::2] = map(lambda x: x!=5, range(3))
>>> aList
['a', True, 'b', True, 'c', True]
>>> aList[1::2] = zip('abc', range(3)) #map、filter、zip对象都支持这样的用法
>>> aList
['a', ('a', 0), 'b', ('b', 1), 'c', ('c', 2)]
>>> aList[::2] = [1]                          #切片不连续时等号两边列表长度必须相等
ValueError: attempt to assign sequence of size 1 to extended slice of size 3
```

3.1.7切片操作的强大功能

（4）使用切片删除列表中的元素

```
>>> aList = [3, 5, 7, 9]
>>> aList[:3] = []                          #删除列表中前3个元素
>>> aList
[9]
```

也可以结合使用del命令与切片结合来删除列表中的部分元素，并且切片元素可以不连续。

```
>>> aList = [3, 5, 7, 9, 11]
>>> del aList[:3]                           #切片元素连续
>>> aList
[9, 11]
>>> aList = [3, 5, 7, 9, 11]
>>> del aList[::2]                          #切片元素不连续，隔一个删一个
>>> aList
[5, 9]
```


3.2 元组：轻量级列表

- 列表的功能虽然很强大，但负担也很重，在很大程度上影响了运行效率。有时候我们并不需要那么多功能，很希望能有个轻量级的列表，元组（tuple）正是这样一种类型。
- 从形式上，元组的所有元素放在一对圆括号中，元素之间使用逗号分隔，如果元组中只有一个元素则必须在最后增加一个逗号。

3.2.1 元组创建与元素访问

```
>>> x = (1, 2, 3)          #直接把元组赋值给一个变量
>>> type(x)                #使用type()函数查看变量类型
<class 'tuple'>
>>> x[0]                   #元组支持使用下标访问特定位置的元素
1
>>> x[-1]                  #最后一个元素，元组也支持双向索引
3
>>> x[1] = 4               #元组是不可变的
TypeError: 'tuple' object does not support item assignment
>>> x = (3)                #这和x = 3是一样的
>>> x
3
>>> x = (3,)               #如果元组中只有一个元素，必须在后面多写一个逗号
>>> x
(3,)
```

3.2.1元组创建与元素访问

```
>>> x = ()                  #空元组
>>> x = tuple()             #空元组
>>> tuple(range(5))         #将其他迭代对象转换为元组
(0, 1, 2, 3, 4)
```

3.2.1 元组创建与元素访问

- 很多内置函数的返回值也是包含了若干元组的可迭代对象，例如enumerate()、zip()等等。

```
>>> list(enumerate(range(5)))
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)]
>>> list(zip(range(3), 'abcdefg'))
[(0, 'a'), (1, 'b'), (2, 'c')]
```

3.2.2 元组与列表的异同点

- 列表和元组都属于有序序列，都支持使用双向索引访问其中的元素，以及使用count()方法统计指定元素的出现次数和index()方法获取指定元素的索引，len()、map()、filter()等大量内置函数和+、+=、in等运算符也都可以作用于列表和元组。

3.2.2 元组与列表的异同点

- 元组属于不可变（immutable）序列，不可以直接修改元组中元素的值，也无法为元组增加或删除元素。
- 元组没有提供append()、extend()和insert()等方法，无法向元组中添加元素；同样，元组也没有remove()和pop()方法，也不支持对元组元素进行del操作，不能从元组中删除元素，而只能使用del命令删除整个元组。
- 元组也支持切片操作，但是只能通过切片来访问元组中的元素，而不允许使用切片来修改元组中元素的值，也不支持使用切片操作来为元组增加或删除元素。

3.2.2 元组与列表的异同点

- Python的内部实现对元组做了大量优化，访问速度比列表更快。如果定义了一系列常量值，主要用途仅是对它们进行遍历或其他类似用途，而不需要对其元素进行任何修改，那么一般建议使用元组而不用列表。
- 元组在内部实现上不允许修改其元素值，从而使得代码更加安全，例如调用函数时使用元组传递参数可以防止在函数中修改元组，而使用列表则很难保证这一点。

3.2.3 生成器推导式

- 生成器推导式（generator expression）的用法与列表推导式非常相似，在形式上生成器推导式使用圆括号（parentheses）作为定界符，而不是列表推导式所使用的方括号（square brackets）。
- 与列表推导式最大的不同是，生成器推导式的结果是一个生成器对象。生成器对象类似于迭代器对象，具有惰性求值的特点，只在需要时生成新元素，比列表推导式具有更高的效率，空间占用非常少，尤其适合大数据处理的场合。

3.2.3 生成器推导式

- 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__()方法或者内置函数next()进行遍历，或者直接使用for循环来遍历其中的元素。但是不管用哪种方法访问其元素，只能从前往后正向访问每个元素，没有任何方法可以再次访问已访问过的元素，也不支持使用下标访问其中的元素。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，enumerate、filter、map、zip等其他迭代器对象也具有同样的特点。

3.2.3 生成器推导式

- 使用生成器对象__next__()方法或内置函数next()进行遍历

```
>>> g = ((i+2)**2 for i in range(10))  #创建生成器对象
>>> g
<generator object <genexpr> at 0x0000000003095200>
>>> tuple(g)                                  #将生成器对象转换为元组
(4, 9, 16, 25, 36, 49, 64, 81, 100, 121)
>>> list(g)                  #生成器对象已遍历结束，没有元素了
[]
>>> g = ((i+2)**2 for i in range(10))  #重新创建生成器对象
>>> g.__next__()             #使用生成器对象的__next__()方法获取元素
4
>>> g.__next__()             #获取下一个元素
9
>>> next(g)                  #使用函数next()获取生成器对象中的元素
16
```

3.2.3 生成器推导式

- 使用for循环直接迭代生成器对象中的元素

```
>>> g = ((i+2)**2 for i in range(10))
>>> for item in g:                    #使用循环直接遍历生成器对象中的元素
    print(item, end=' ')
4 9 16 25 36 49 64 81 100 121
```

3.2.3 生成器推导式

- 访问过的元素不再存在

```
>>> x = filter(None, range(20))    #filter对象也具有类似的特点
>>> 5 in x
True
>>> 2 in x                         #不可再次访问已访问过的元素
False
>>> x = map(str, range(20))        #map对象也具有类似的特点
>>> '0' in x
True
>>> '0' in x                       #不可再次访问已访问过的元素
False
```

3.3 字典：反映对应关系的映射类型

- 字典（dictionary）是包含若干“键:值”元素的无序可变序列，字典中的每个元素包含用冒号分隔开的“键”和“值”两部分，表示一种映射或对应关系，也称关联数组。定义字典时，每个元素的“键”和“值”之间用冒号分隔，不同元素之间用逗号分隔，所有的元素放在一对大括号“{}”中。
- 字典中元素的“键”可以是Python中任意不可变数据，例如整数、实数、复数、字符串、元组等类型等可哈希数据，但不能使用列表、集合、字典或其他可变类型作为字典的“键”。另外，字典中的“键”不允许重复，而“值”是可以重复的。

3.3.1 字典创建与删除

- 使用赋值运算符“=”将一个字典赋值给一个变量即可创建一个字典变量。

```
>>> aDict = {'server': 'db.diveintopython3.org', 'database':
'mysql'}
```

- 也可以使用内置类dict以不同形式创建字典。

```
>>> x = dict()                                    #空字典
>>> type(x)                                       #查看对象类型
<class 'dict'>
>>> x = {}                                        #空字典
>>> keys = ['a', 'b', 'c', 'd']
>>> values = [1, 2, 3, 4]
>>> dictionary = dict(zip(keys, values))          #根据已有数据创建字典
>>> d = dict(name='Dong', age=39)                 #以关键参数的形式创建字典
>>> aDict = dict.fromkeys(['name', 'age', 'sex'])
                                  #以给定内容为“键”，创建“值”为空的字典
>>> aDict
{'age': None, 'name': None, 'sex': None}
```

3.3.2 字典元素的访问

- 字典中的每个元素表示一种映射关系或对应关系，根据提供的“键”作为下标就可以访问对应的“值”，如果字典中不存在这个“键”会抛出异常。

```
>>> aDict = {'age': 39, 'score': [98, 97], 'name':
'Dong', 'sex': 'male'}
>>> aDict['age']          #指定的“键”存在，返回对应的“值”
39
>>> aDict['address']      #指定的“键”不存在，抛出异常
KeyError: 'address'
```

3.3.2 字典元素的访问

- 字典对象提供了一个get()方法用来返回指定“键”对应的“值”，并且允许指定该键不存在时返回特定的“值”。例如：

```
>>> aDict.get('age')                          #如果字典中存在该“键”则返回对应的“值”
39
>>> aDict.get('address', 'Not Exists.') #指定的“键”不存在时返回指定的默认值
'Not Exists.'
```

- 使用字典对象的items()方法可以返回字典的键、值对。
- 使用字典对象的keys()方法可以返回字典的键。
- 使用字典对象的values()方法可以返回字典的值。

3.3.2 字典元素的访问

- **问题解决**：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
>>> y = [random.choice(x) for i in range(1000)]
>>> z = ''.join(y)
>>> d = dict()                          #使用字典保存每个字符出现次数
>>> for ch in z:
    d[ch] = d.get(ch, 0) + 1
```

3.3.3 元素添加、修改与删除

- 当以指定“键”为下标为字典元素赋值时，有两种含义：

1）若该“键”存在，则表示修改该“键”对应的值；

2）若不存在，则表示添加一个新的“键:值”对，也就是添加一个新元素。

```
>>> aDict = {'age': 35, 'name': 'Dong', 'sex': 'male'}
>>> aDict['age'] = 39                    #修改元素值
>>> aDict
{'age': 39, 'name': 'Dong', 'sex': 'male'}
>>> aDict['address'] = 'SDIBT'           #添加新元素
>>> aDict
{'age': 39, 'address': 'SDIBT', 'name': 'Dong', 'sex': 'male'}
```

3.3.3 元素添加、修改与删除

- 使用字典对象的update()方法可以将另一个字典的“键:值”一次性全部添加到当前字典对象，如果两个字典中存在相同的“键”，则以另一个字典中的“值”为准对当前字典进行更新。

```
>>> aDict = {'age': 37, 'score': [98, 97], 'name': 'Dong', 'sex': 'male'}
>>> aDict.update({'a':97, 'age':39})  #修改’age’键的值，同时添加新元素’a’:97
>>> aDict
{'score': [98, 97], 'sex': 'male', 'a': 97, 'age': 39, 'name': 'Dong'}
```

3.3.3 元素添加、修改与删除

- 如果需要删除字典中指定的元素，可以使用del命令。

```
>>> del aDict['age']          #删除字典元素
>>> aDict
{'score': [98, 97], 'sex': 'male', 'a': 97, 'name': 'Dong'}
```

- 也可以使用字典对象的pop()和popitem()方法弹出并删除指定的元素，例如：

```
>>> aDict = {'age': 37, 'score': [98, 97], 'name': 'Dong',
'sex': 'male'}
>>> aDict.popitem()           #弹出一个元素，对空字典会抛出异常
('age', 37)
>>> aDict.pop('sex')          #弹出指定键对应的元素
'male'
>>> aDict
{'score': [98, 97], 'name': 'Dong'}
```

3.3.4 标准库collections中与字典有关的类

（1）OrderedDict类

Python内置字典dict是无序的，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。

```
>>> 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)])
```

3.3.4 标准库collections中与字典有关的类

（2）defaultdict类

```
>>> 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)      #所有值默认为0
>>> frequences
defaultdict(<class 'int'>, {})
>>> for item in z:
    frequences[item] += 1               #修改每个字符的频次
>>> frequences.items()
```

3.3.4 标准库collections中与字典有关的类

（3）Counter类

对于频次统计的问题，使用collections模块的Counter类可以更加快速地实现这个功能，并且能够提供更多的功能，例如查找出现次数最多的元素。

```
>>> from collections import Counter
>>> frequences = Counter(z)          #这里的z还是前面代码中的字符串对象
>>> frequences.items()
>>> frequences.most_common(1)        #返回出现次数最多的1个字符及其频率
>>> frequences.most_common(3)        #返回出现次数最多的前3个字符及其频率
```


3.4 集合

- 集合（set）属于Python无序可变序列，使用一对大括号作为定界符，元素之间使用逗号分隔，同一个集合内的每个元素都是唯一的，元素之间不允许重复。
- 集合中只能包含数字、字符串、元组等不可变类型（或者说可哈希）的数据，而不能包含列表、字典、集合等可变类型的数据。

3.4.1 集合对象的创建与删除

- 直接将集合赋值给变量即可创建一个集合对象。

```
>>> a = {3, 5}                                    #创建集合对象
>>> type(a)                                       #查看对象类型
<class 'set'>
```

- 也可以使用函数set()函数将列表、元组、字符串、range对象等其他可迭代对象转换为集合，如果原来的数据中存在重复元素，则在转换为集合的时候只保留一个；如果原序列或迭代对象中有不可哈希的值，无法转换成为集合，抛出异常。

```
>>> a_set = set(range(8, 14))                     #把range对象转换为集合
>>> a_set
{8, 9, 10, 11, 12, 13}
>>> b_set = set([0, 1, 2, 3, 0, 1, 2, 3, 7, 8])   #转换时自动去掉重复元素
>>> b_set
{0, 1, 2, 3, 7, 8}
>>> x = set()                                     #空集合
```

3.4.2 集合操作与运算

- 直接将集合赋值给变量即可创建一个集合对象。

```
>>> a = {3, 5}                                    #创建集合对象
>>> type(a)                                       #查看对象类型
<class 'set'>
```

- 也可以使用函数set()函数将列表、元组、字符串、range对象等其他可迭代对象转换为集合，如果原来的数据中存在重复元素，则在转换为集合的时候只保留一个；如果原序列或迭代对象中有不可哈希的值，无法转换成为集合，抛出异常。

```
>>> a_set = set(range(8, 14))                     #把range对象转换为集合
>>> a_set
{8, 9, 10, 11, 12, 13}
>>> b_set = set([0, 1, 2, 3, 0, 1, 2, 3, 7, 8])    #转换时自动去掉重复元素
>>> b_set
{0, 1, 2, 3, 7, 8}
>>> x = set()                                     #空集合
```

3.4.2 集合操作与运算

- pop()方法用于随机删除并返回集合中的一个元素，如果集合为空则抛出异常；
- remove()方法用于删除集合中的元素，如果指定元素不存在则抛出异常；
- discard()用于从集合中删除一个特定元素，如果元素不在集合中则忽略该操作；
- clear()方法清空集合删除所有元素。

```
>>> s.discard(5)                    #删除元素，不存在则忽略该操作
>>> s
{1, 2, 3, 4}
>>> s.remove(5)                     #删除元素，不存在就抛出异常
KeyError: 5
>>> s.pop()                         #删除并返回一个元素
1
```

3.4.2 集合操作与运算

(2) 集合运算

```
>>> a_set = set([8, 9, 10, 11, 12, 13])
>>> b_set = {0, 1, 2, 3, 7, 8}
>>> a_set | b_set                          #并集
{0, 1, 2, 3, 7, 8, 9, 10, 11, 12, 13}
>>> a_set.union(b_set)                     #并集
{0, 1, 2, 3, 7, 8, 9, 10, 11, 12, 13}
>>> a_set & b_set                          #交集
{8}
>>> a_set.intersection(b_set)              #交集
{8}
>>> a_set.difference(b_set)                #差集
{9, 10, 11, 12, 13}
>>> a_set - b_set
{9, 10, 11, 12, 13}
>>> a_set.symmetric_difference(b_set) #对称差集
{0, 1, 2, 3, 7, 9, 10, 11, 12, 13}
>>> a_set ^ b_set
{0, 1, 2, 3, 7, 9, 10, 11, 12, 13}
```

3.4.2 集合操作与运算

```
>>> x = {1, 2, 3}
>>> y = {1, 2, 5}
>>> z = {1, 2, 3, 4}
>>> x < y                                  #比较集合大小/包含关系
False
>>> x < z                                  #真子集
True
>>> y < z
False
>>> {1, 2, 3} <= {1, 2, 3}                 #子集
True
```

3.4.3 集合应用案例

- 可以使用集合快速提取序列中单一元素，即提取出序列中所有不重复元素。如果使用传统方式的话，需要编写下面的代码：

```
>>> import random
#生成100个介于0到9999之间的随机数
>>> listRandom = [random.choice(range(10000)) for i in
range(100)]
>>> noRepeat = []
>>> for i in listRandom :
    if i not in noRepeat :
        noRepeat.append(i)
```

- 而如果使用集合的话，只需要下面这么一行代码就可以了，可以参考上面的代码对结果进行验证。

```
>>> newSet = set(listRandom)
```

3.4.3 集合应用案例

- **问题解决**：返回指定范围内一定数量的不重复数字。

```
import random

def randomNumbers(number, start, end):
    '''使用集合来生成number个介于start和end之间的不重复随机数'''
    data = set()
    while len(data)<number:
        element = random.randint(start, end)
        data.add(element)
    return data
```

3.4.3 集合应用案例

- **问题解决**：下面两段代码用来测试指定列表中是否包含非法数据，很明显第二段使用集合的代码更高效一些。

```
import random

lstColor = ('red', 'green', 'blue')
colors = [random.choice(lstColor) for i in range(10000)]

for item in colors:                    #遍历列表中的元素并逐个判断
    if item not in lstColor:
        print('error:', item)
        break

if (set(colors)-set(lstColor)): #转换为集合之后再比较
    print('error')
```

3.4.3 集合应用案例

- **问题解决**：假设已有若干用户名字及其喜欢的电影清单，现有某用户，已看过并喜欢一些电影，现在想找个新电影看看，又不知道看什么好。
- 思路：根据已有数据，查找与该用户爱好最相似的用户，也就是看过并喜欢的电影与该用户最接近，然后从那个用户喜欢的电影中选取一个当前用户还没看过的电影，进行推荐。

3.4.3 集合应用案例

```
from random import randrange

# 其他用户喜欢看的电影清单
data = {'user'+str(i):{'film'+str(randrange(1, 10))\
                       for j in range(randrange(15))}\
        for i in range(10)}

# 待测用户曾经看过并感觉不错的电影
user = {'film1', 'film2', 'film3'}
# 查找与待测用户最相似的用户和Ta喜欢看的电影
similarUser, films = max(data.items(), key=lambda item:
len(item[1]&user))
```

3.4.3 集合应用案例

```
print('历史数据：')
for u, f in data.items():
    print(u, f, sep=':')
print('和您最相似的用户是：', similarUser)
print('Ta最喜欢看的电影是：', films)
print('Ta看过的电影中您还没看过的有：', films-user)
```

3.5 序列解包的多种形式和用法

- 可以使用序列解包功能对多个变量同时进行赋值。

```
>>> x, y, z = 1, 2, 3                    #多个变量同时赋值
>>> v_tuple = (False, 3.5, 'exp')
>>> (x, y, z) = v_tuple
>>> x, y, z = v_tuple
>>> x, y = y, x                          #交换两个变量的值
>>> x, y, z = range(3)                   #可以对range对象进行序列解包
>>> x, y, z = iter([1, 2, 3])            #使用迭代器对象进行序列解包
>>> x, y, z = map(str, range(3))         #使用可迭代的map对象进行序列解包
```

3.5 序列解包的多种形式和用法

```
>>> a = [1, 2, 3]
>>> b, c, d = a                          #列表也支持序列解包的用法
>>> x, y, z = sorted([1, 3, 2])          #sorted()函数返回排序后的列表
>>> s = {'a':1, 'b':2, 'c':3}
>>> b, c, d = s.items()                  #这是Python 3.5之前的版本执行结果
                                         #Python 3.6之后的版本略有不同
>>> b
('c', 3)
>>> b, c, d = s                          #使用字典时不用太多考虑元素的顺序
>>> b
'c'
>>> b, c, d = s.values()
>>> print(b, c, d)
1 3 2
>>> a, b, c = 'ABC'                      #字符串也支持序列解包
>>> print(a, b, c)
A B C
```

3.5 序列解包的多种形式和用法

- 使用序列解包可以很方便地同时遍历多个序列。

```
>>> 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
```

3.5 序列解包的多种形式和用法

- 对内置函数enumerate()返回的迭代对象进行遍历：

```
>>> x = ['a', 'b', 'c']
>>> for i, v in enumerate(x):
    print('The value on position {0} is {1}'.format(i,v))
The value on position 0 is a
The value on position 1 is b
The value on position 2 is c
```

3.5 序列解包的多种形式和用法

- 使用序列解包遍历字典元素:

```
>>> s = {'a':1, 'b':2, 'c':3}
>>> for k, v in s.items(): #字典中每个元素包含“键”和“值”两部分
    print(k, v)
a 1
c 3
b 2
```