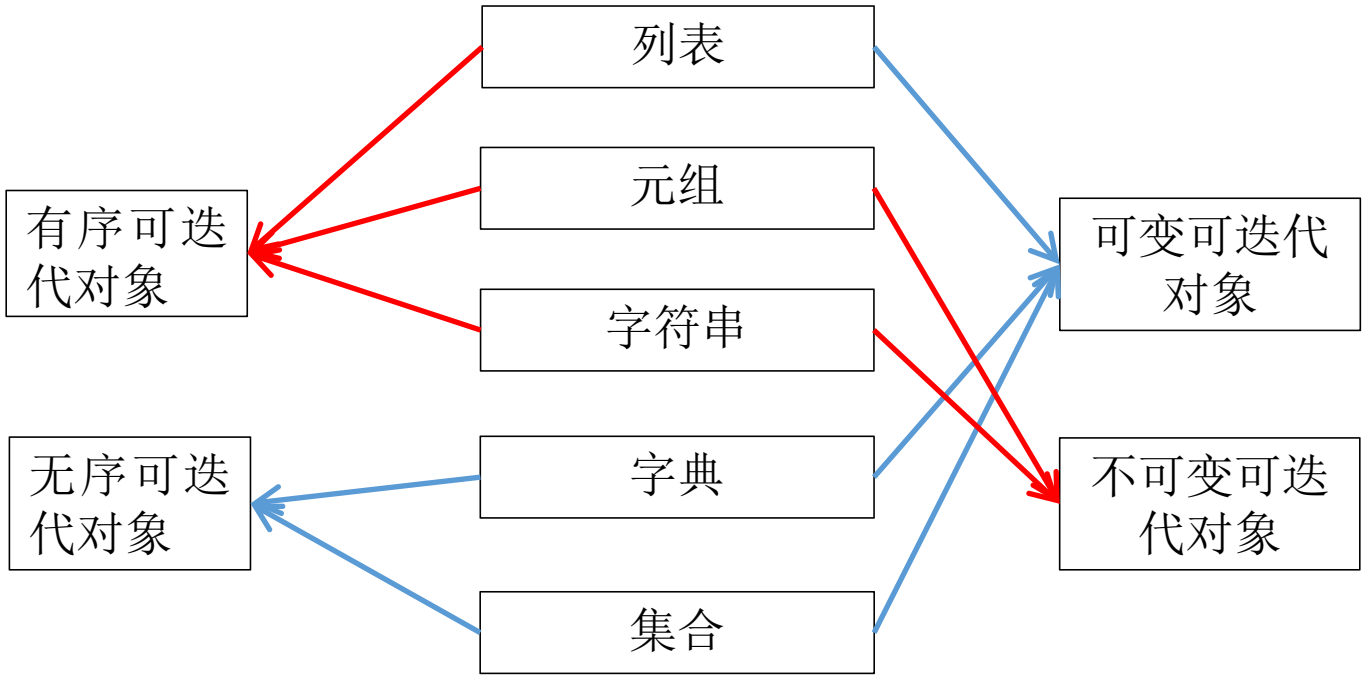
第3章 Python可迭代对象

第3章 Python可迭代对象

- 从面向对象程序设计的角度来讲，在Python中实现了特殊方法__iter__()的类的对象称作可迭代对象，同时实现了特殊方法__iter__()和__next__()的类的对象称作迭代器对象。
- 可迭代对象包括迭代器对象以及列表、元组、字典、集合、字符串和range对象。其中，列表、元组、字典、集合、字符串这几种类型的对象在任何时刻都在相应的内存中包含了全部的元素，称作容器对象。
- range对象比较特殊，不在内存中包含具体的元素但支持下标和切片运算，没有实现特殊方法__next__()所以又不属于迭代器对象。
- 迭代器对象具有惰性求值特点，只在必要时才会生成下一个值，每个值只会生成一次。迭代器对象不持有任何元素，不支持使用下标和切片访问指定位置上的元素，也不支持切片运算和内置函数len()，只能从前往后逐个生成值。

第3章 Python可迭代对象



3.1 列表

- 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

- 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]
['富强', '民主', '文明']
['和谐', 2.0, 5, [10, 20]]
[['自由', 200, 7], ['平等', 260, 9]]
[{3}, {5:6}, (1, 2, 3)]
```

3.1 列表

- 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“=”直接将一个列表字面值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、集合或其他有限长度的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

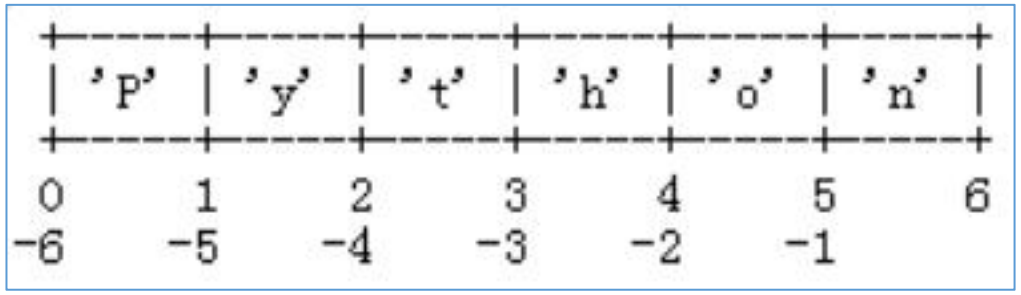
- 当一个变量不再使用时，可以使用del命令将其删除，这一点适用于所有类型的变量。

```
>>> x = [1, 2, 3]
>>> del x                          # 删除变量，解除变量与列表之间的关联
>>> x                              # 变量删除后无法再访问，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')                    # 创建列表对象
>>> 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	功能描述
append(object, /)	将任意类型的对象object追加至当前列表的尾部，不影响当前列表中已有的元素下标，也不影响当前列表的引用，没有返回值
clear()	删除当前列表中所有元素，不影响列表的引用，没有返回值
copy()	返回当前列表的浅复制，把当前列表中所有元素的引用复制到新列表中
count(value, /)	返回值为value的元素在当前列表中的出现次数，如果当前列表中没有值为value的元素则返回0，对当前列表没有任何影响
extend(iterable, /)	将可迭代对象iterable中所有元素追加至当前列表的尾部，不影响当前列表中已有的元素位置和列表的引用，没有返回值
insert(index, object, /)	在当前列表的index位置前面插入对象object，该位置及后面所有元素自动向后移动，索引加1，没有返回值
index(value, start=0, stop=9223372036854775807, /)	返回当前列表指定范围中第一个值为value的元素的索引，若不存在值为value的元素则抛出异常ValueError，可以使用参数start和stop指定要搜索的下标范围，start默认为0表示从头开始，stop默认值为最大允许的下标值
pop(index=-1, /)	删除并返回当前列表中下标为index的元素，该位置后面的所有元素自动向前移动，索引减1。index默认为-1，表示删除并返回列表中最后一个元素
remove(value, /)	在当前列表中删除第一个值为value的元素，被删除元素所在位置之后的所有元素自动向前移动，索引减1，不影响当前列表的引用，没有返回值
reverse()	对当前列表中的所有元素进行原地翻转，首尾交换，不影响当前列表的引用，没有返回值
sort(*, key=None, reverse=False)	对当前列表中的元素进行原地排序，是稳定排序（在指定规则下相等的元素保持原来的相对顺序）。参数key用来指定排序规则，可以为任意可调用对象；参数reverse为False表示升序，True表示降序。不影响当前列表的引用，没有返回值

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()用于将参数可迭代对象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]
>>> 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]
```

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]
```

3.1.3 列表常用方法

（5）copy()方法

copy()方法返回当前列表的浅复制，返回包含原列表中所有元素的引用的新列表。

```
>>> x = [1, 2, [3, 4]]
>>> y = x.copy()                # 返回列表x的浅复制
>>> y
[1, 2, [3, 4]]
>>> y[2].append(5)              # y[2]是子列表，append()不改变子列表引用，所以会影响x
>>> x[0] = 6                    # 这个操作不会对y造成任何影响
>>> y.append(6)                 # 这个操作不会影响x
>>> y
[1, 2, [3, 4, 5], 6]
>>> x
[6, 2, [3, 4, 5]]
```

3.1.3 列表常用方法

- 标准库copy中的deepcopy()函数可以实现深复制，对原列表中的元素的引用进行复制，如果元素是容器对象就递归对其中的元素进行复制。深复制得到的对象与原来的对象完全独立。

```
>>> import copy
>>> x = [1, 2, [3, 4]]
>>> y = copy.deepcopy(x)              # 得到完全独立的副本
>>> x[2].append(5)                    # 这个操作不影响y
>>> y.append(6)                       # 这个操作不影响x
>>> y
[1, 2, [3, 4], 6]
>>> x
[1, 2, [3, 4, 5]]
```

3.1.3 列表常用方法

- 把同一个列表赋值给两个变量时，这两个变量的引用不同，得到两个不同的列表，但有可能两个列表中的元素具有相同的引用。

```
>>> x = [1, 2, [3, 4]]
>>> y = [1, 2, [3, 4]]
>>> x == y                    # 两个列表中元素的值相同
True
>>> x is y                    # 两个列表的引用不同，是不同的两个列表，完全独立
False
>>> x[0] = 6                  # 这个操作不影响y
>>> x[2].append(5)            # 这个操作不影响y
>>> x
[6, 2, [3, 4, 5]]
>>> y
[1, 2, [3, 4]]
```

3.1.3 列表常用方法

- 直接把一个变量赋值给另一个变量时，两个变量引用同一个对象。

```
>>> x = [1, 2, [3, 4]]
>>> y = x
>>> x[0] = 7                # 这个操作影响y
>>> x[2].append(5)          # 这个操作影响y
>>> x.append(6)
>>> x
[7, 2, [3, 4, 5], 6]
>>> y
[7, 2, [3, 4, 5], 6]
```

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。+=用于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)
2461112034752
>>> x = x + [4]                    # 连接两个列表
>>> x
[1, 2, 3, 4]
>>> id(x)                          # 内存地址发生改变
2461112100224
>>> x += [5]                       # 为列表追加元素
>>> x
[1, 2, 3, 4, 5]
>>> id(x)                          # 内存地址不变
2461112100224
```

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)
2461112034304
>>> 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
                       # 注意，这里的数字和上一页相同，但不需要关心具体的值
                       # Python会尽量避免重复申请内存空间，尽量利用当前进程已经拥有的空间
2461112100224
>>> 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
2461112100224
```

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)]
>>> 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

- 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，但代码执行效率并不比循环结构高。

```
>>> 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

```
>>> 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())
```

- 也等价于下面两种写法，更推荐第二种，但实际开发时应避免把迭代器转换为列表之后再使用。

```
>>> aList = list(map(lambda s: s.strip(), freshfruit))
>>> aList = list(map(str.strip, freshfruit))
```

3.1.6 列表推导式语法与应用

（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，其中第一个循环相当于外循环，执行的慢；而第二个循环相当于内循环，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

- 使用标准库函数快速实现相同的功能。

```
>>> vec = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> from itertools import chain
>>> list(chain(*vec))                    # 更高效的方法
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> sum(vec, [])
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

3.1.7 切片操作的强大功能

- 切片用来访问有序序列中的部分元素。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；step为正整数时省略end表示一直切到最后一个元素，step为负整数时省略end表示一直切到第一个元素。

✓step为负整数时表示反向切片，这时start应该在end的右侧才行，否则返回结果为空。

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，其中包含原列表中部分元素的引用。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]                   # 从下标0开始，隔一个取一个，获取偶数位置的元素
[3, 5, 7, 11, 15]
>>> aList[1::2]                  # 从下标1开始，隔一个取一个，获取奇数位置的元素
[4, 6, 9, 13, 17]
>>> aList[3:6]                   # 返回下标[3,6)之间的元素
[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]                    # 起始位置小于左边界，在开始处截断
[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。切片在等号左侧时用来标记位置，并没有真的切出来。

```
>>> 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]                # step=1时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，切片步长不为1
                                         # 此时等号两边的元素数量必须相等
>>> aList
[0, 2, 0, 4, 0, 6]
>>> aList[::2] = ['a', 'b', 'c']         # 隔一个修改一个，等号两侧长度相等
>>> 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，等号右侧可以为可迭代对象
>>> 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
                                              # 否则代码会出错抛出异常
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，切片步长必须为1
>>> 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，此时切片可以不连续，不要求step必须为1。

```
>>> 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

- 列表的功能虽然很强大，但负担重、开销大，影响了程序运行效率。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。
- 定义后元组中元素的数量和引用不允许改变。

3.2.1 元组创建与元素访问

```
>>> x = (1, 2, 3)          # 直接把元组字面值赋值给一个变量
>>> 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
```

3.2.1 元组创建与元素访问

```
>>> 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,)
>>> x = 3,                     # 真正创建元组的是逗号，圆括号只是辅助
>>> x
(3,)
>>> 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或者可以生成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，一旦定义，其元素的数量和引用都是不可变的，不可以直接修改元组中元素的引用，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。
- 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，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，占用内存比列表略小，访问速度比列表略快。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引用，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 生成器表达式

- 生成器表达式（generator expression）的用法与列表推导式非常相似，只是在形式上生成器表达式使用圆括号（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，生成器表达式的结果是一个生成器对象。生成器对象属于迭代器对象，具有惰性求值的特点，只在需要时生成新元素，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，但一般不建议这样做。更推荐使用生成器对象的__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:
    print(item, end=' ')

4 9 16 25 36 49 64 81 100 121
```

3.3 字典

- 字典（dictionary）是包含若干“键:值”元素的无序可变容器对象，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对象作为字典的“键”。
- 字典中的“键”不允许重复，“值”可以重复。

3.3 字典

方法	功能描述
clear()	不接收参数，删除当前字典对象中的所有元素，没有返回值
copy()	不接收参数，返回当前字典的浅复制
fromkeys(iterable, value=None,/)	以参数iterable中的元素为“键”、以参数value为“值”创建并返回字典对象。字典中所有元素的“值”的引用都一样，要么是None，要么全部引用参数value指定的值
get(key, default=None,/)	返回当前字典对象中以参数key为“键”对应的元素的“值”，如果当前字典对象中没有以key为“键”的元素，返回参数default的值
items()	不接收参数，返回包含当前字典对象中所有元素的dict_items对象，其中每个元素形式为元组(key,value)
keys()	不接收参数，返回当前字典对象中所有的“键”，结果为dict_keys类型的可迭代对象，可以和集合进行并集、交集、差集等运算
pop(k[,d])	删除以k为“键”的元素，返回对应元素的“值”，如果当前字典中没有以k为“键”的元素，返回参数d，如果没有指定参数d，抛出KeyError异常
popitem()	不接收参数，按LIFO（Last In First Out，后进先出）顺序删除并返回一个元组(key, value)，如果当前字典为空则抛出KeyError异常
setdefault(key, default=None, /)	如果参数key是当前字典的“键”，返回对应的“值”；如果key不是当前字典的“键”，插入元素“key:default”并返回default的值
update([E,]**F)	使用E和F中的数据更新当前字典对象，**表示参数F只能接收字典或关键参数（见5.2.3节），没有返回值
values()	不接收参数，返回包含当前字典对象中所有的“值”的dict_values对象

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'}
>>> data = {'国家': ['富强', '民主', '文明', '和谐'],
            '社会': ['自由', '平等', '公正', '法治'],
            '公民': ['爱国', '敬业', '诚信', '友善']}
```

- 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]
>>> d = 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
```

3.3.1 字典创建与删除

```
>>> dict.fromkeys(['name', 'age', 'sex'])  # 以给定内容为“键”，创建“值”为空的字典
{'name': None, 'age': None, 'sex': None}
>>> dict.fromkeys([3, 5, 7], 666)          # 所有元素的“值”都是666
{3: 666, 5: 666, 7: 666}
>>> d = dict.fromkeys([3, 5, 7], [])       # 所有元素的“值”引用同一个空列表
>>> d
{3: [], 5: [], 7: []}
>>> d[3].append(666)                       # 所有元素的“值”都是一样的
>>> d
{3: [666], 5: [666], 7: [666]}
>>> d[3] = 999                             # 修改一个元素“值”的引用
                                           # 此后和其他的“值”再也没有关系了
>>> d
{3: 999, 5: [666], 7: [666]}
```

- 还可以使用字典推导式来创建字典。

```
>>> {num: chr(num) for num in range(97, 103)}
{97: 'a', 98: 'b', 99: 'c', 100: 'd', 101: 'e', 102: 'f'}
```

3.3.2 字典元素访问

- 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 字典元素访问

- 可以使用选择结构或异常处理结构来避免“键”不存在时代码崩溃抛出异常。

```
>>> aDict = {'age': 39, 'score': [98, 97], 'name': 'Dong', 'sex':
'male'}
>>> if 'Age' in aDict:
    print(aDict['Age'])
else:
    print('Not Exists.')

Not Exists.
>>> try:
    print(aDict['address'])
except:
    print('Not Exists.')

Not Exists.
```

3.3.2 字典元素访问

- 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”。

```
>>> 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，keys()方法可以返回字典的键，values()方法可以返回字典的值。

```
>>> aDict = {'age': 39, 'score': [98, 97], 'name': 'Dong', 'sex': 'male'}
>>> aDict.items()
dict_items([('age', 39), ('score', [98, 97]), ('name', 'Dong'), ('sex', 'male')])
>>> aDict.keys()
dict_keys(['age', 'score', 'name', 'sex'])
>>> aDict.values()
dict_values([39, [98, 97], 'Dong', 'male'])
```

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）若指定的“键”存在，表示修改该“键”对应的值；

2）若指定的“键”不存在，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, 'name': 'Dong', 'sex': 'male', 'address': 'SDIBT'}
```

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”元素一次性全部添加到当前字典对象中，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。

```
>>> 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
{'age': 39, 'score': [98, 97], 'name': 'Dong', 'sex': 'male', 'a': 97}
```

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], 'name': 'Dong', 'sex': 'male', 'a': 97}
```

- 也可以使用字典对象的pop()和popitem()方法删除元素，例如：

```
>>> aDict = {'age': 37, 'score': [98, 97], 'name': 'Dong', 'sex': 'male'}
>>> aDict.popitem()                           # 以LIFO顺序删除并返回一个元素
('sex', 'male')
>>> aDict.pop('age')                          # 删除指定的“键”，返回对应的“值”
37
>>> aDict
{'score': [98, 97], 'name': 'Dong'}
```

3.3.3 元素添加、修改与删除

- **应用：**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
from random import choices

x = string.ascii_letters + string.digits + string.punctuation
y = ''.join(choices(x, k=1000))
d = dict()                          # 使用字典保存每个字符出现次数
for ch in y:
    d[ch] = d.get(ch, 0) + 1
print(d)
```

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，如果确实需要一个可以记住并依赖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
from random import choices
from collections import defaultdict

x = string.ascii_letters + string.digits + string.punctuation
z = ''.join(choices(x, k=1000))

frequences = defaultdict(int)                # 所有值默认为0
for item in z:
    frequences[item] += 1                    # 修改每个字符的频次
print(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。

```
import string
from random import choices
from collections import Counter

x = string.ascii_letters + string.digits + string.punctuation
z = ''.join(choices(x, k=1000))

frequences = Counter(z)
print(frequences.items())
print(frequences.most_common(1))    # 出现次数最多的1个字符及其频率
print(frequences.most_common(3))    # 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无序可变容器对象，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，不能以任何形式包含列表、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

- 直接将集合字面值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对象或其他有限长度的可迭代对象转换为集合，并自动去除重复元素；如果原可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
```

3.4.1 集合对象的创建与删除

- 还可以使用集合推导式创建集合。

```
>>> {x.strip() for x in (' he  ', 'she ', '  I')}
{'he', 'she', 'I'}
>>> from random import randint
>>> x = {randint(1,500) for _ in range(50)}
>>> len(x)
47
>>> {str(x) for x in range(10)}
{'6', '5', '8', '1', '4', '0', '9', '2', '3', '7'}
```

3.4.2 集合操作与运算

方法	功能描述
add(...)	往当前集合中增加一个可哈希元素，如果集合中已经存在该元素，直接忽略该操作，如果参数不可哈希，抛出TypeError异常并提示参数不可哈希。该方法直接修改当前集合，没有返回值
clear()	删除当前集合对象中所有元素，没有返回值
difference(...)	接收一个或多个集合（或其他可迭代对象），返回当前集合对象与所有参数对象的差集，不对当前集合做任何修改，功能类似于差集运算符“-”
difference_update(...)	接收一个或多个集合（或其他可迭代对象），从当前集合中删除所有参数对象中的元素，对当前集合进行更新，该方法没有返回值，功能类似于运算符“-=”
discard(...)	接收一个可哈希对象作为参数，从当前集合中删除该元素，如果参数元素不在当前集合中则忽略该操作。该方法直接修改当前集合，没有返回值
intersection(...)	接收一个或多个集合对象（或其他可迭代对象），返回当前集合与所有参数对象的交集，不对当前集合做任何修改，功能类似于运算符“&”
intersection_update(...)	接收一个或多个集合（或其他可迭代对象），使用当前集合与所有参数对象的交集更新当前集合对象，没有返回值，功能类似于运算符“&=”
pop()	不接收参数，删除并返回当前集合中的任意一个元素，如果当前集合为空则抛出KeyError异常
remove(...)	从当前集合中删除参数指定的元素，如果参数指定的元素不在集合中就抛出KeyError异常，该方法直接修改当前集合，没有返回值
union(...)	接收一个或多个集合（或其他可迭代对象），返回当前集合与所有参数对象的并集，不对当前集合做任何修改，功能类似于并集运算符“|”
update(...)	接收一个或多个集合（或其他可迭代对象），把参数对象中所有元素添加到当前集合对象中，没有返回值，功能类似于运算符“|=”

3.4.2 集合操作与运算

（1）集合元素增加与删除

- 使用集合对象的add()方法可以增加新元素，如果该元素已存在则忽略该操作，不会抛出异常；
- update()方法用于合并另外一个或多个集合中的元素到当前集合中，并自动去除重复元素；

```
>>> s = {1, 2, 3}
>>> s.add(3)                    # 添加元素，重复元素自动忽略
>>> s
{1, 2, 3}
>>> s.update({3,4})             # 更新当前字典，自动忽略重复的元素
>>> s
{1, 2, 3, 4}
```

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
```

3.5 序列解包

- 使用序列解包可以对多个变量同时赋值。

```
>>> 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

```
>>> 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()
>>> b
('a', 1)
>>> b, c, d = s                          # 使用字典时不用太多考虑元素的顺序
>>> b
'a'
>>> b, c, d = s.values()
>>> print(b, c, d)
1 2 3
>>> 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

- 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

- 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
>>> for index, (first, second) in enumerate(zip('abc','ABC')):
       print(index, first, second)

0 a A
1 b B
2 c C
```

3.5 序列解包

- 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
```