第3章

组合数据类型

在大数据时代，数据获取方式更加多样，数据传递速度更加快捷，实际应用中人们面对的常常是大批量数据。如果将这些数据罗列起来，用一条或者多条Python语句进行批量化处理，势必可以简化操作，大大提高运行效率。这种能够表示多个数据的类型称为组合数据类型。Python组合数据类型是内置对象，可以实现复杂的数据处理任务。

3.1 组合数据类型的基本概念

Python提供的组合数据类型为多个相同或不同类型的数据提供了更为宽泛的单一表示，方便针对大量数据批量化高效处理。在Python中，组合数据类型主要有列表、元组、字典、集合、字符串等，按照数据的存放是否有先后次序可以将它们分为有序类型（列表、元组和字符串）和无序类型（字典和集合）两种；按照元素值是否允许改变可以分为可变类型（列表、字典和集合）和不可变类型（元组和字符串）；根据数据之间的关系可以分成三类，分别是序列类型（列表、元组和字符串）、集合类型（集合）和映射类型（字典）。

序列类型： Python中的序列类似数学中的数列，它是一串有序元素的向量，可以通过索引访问每一个元素。序列类型包括字符串、列表和元组，它们可以进行通用的序列操作，包括索引、切片、序列相加、乘法、计算长度、最小值、最大值等。序列类型在实际应用中使用频率颇高。

列表、元组、字符串等有序序列都支持双向索引。创建了一个有序序列，就可以通过索引访问其中的元素。有序序列的正向索引从0开始，以1为步长从左向右依次递增。索引为0的元素表示第1个元素，索引为1的元素表示第2个元素；负向索引从右向左开始，最右边一个元素的索引为－1，第二个元素的索引为－2，依次类推，如图3-1所示。

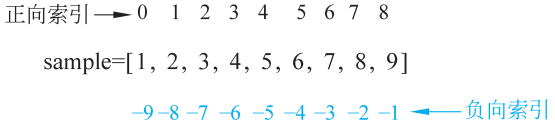

图3-1 列表的双向索引

Python 有序序列(例如列表、字符串、元组)都支持切片操作,其语法格式如下。

```
<序列名>[start: end: step]
```

其中,start 表示切片的起始位置,起始索引默认为 0;end 表示切片的截止位置,即结束索引;step 指切片的步长,当步长为正时,表示从左向右正向获取序列元素,步长为负时,表示从右向左负向获取元素。需要注意的是,切片操作的结果不包含结束索引 end 对应的元素。使用切片操作可以方便地在有序可变序列的任意位置实现多个元素的插入、删除、修改、替换等操作,而且不影响序列对象的内存地址。

集合类型:Python 中的集合类似数学中的集合概念。集合类型中的元素具有无序性,无法通过下标(索引)锁定集合中的每一个数据,相同元素在集合中唯一存在。集合中的元素只能是固定数据类型,例如整数、浮点数、字符串、元组等;由于列表、字典以及集合类型的可变性,它们不可以作为集合中的数据元素。集合类型与其他类型的最大不同是它不包含重复元素。当对数据进行去重处理时,一般使用集合完成。

映射类型:映射类型存储了对象之间的映射关系,是键值对的集合,也存在无序性。Python 中的字典正是映射类型的典型代表。在字典中,键代表属性,值表示这个属性对应的内容,通过元素的键可以获取对应元素的值。

3.2 列　　表

列表是 Python 的内置可变序列,是包含若干元素的有序连续内存空间,用于存储一组数据类型相同或不同的元素。在形式上,Python 列表的所有元素放在一对方括号“[]”中,相邻元素之间用逗号隔开。列表中的元素可以是各种类型的对象,无论是整数、浮点数、字符串,还是列表、字典、集合、元组或者其他自定义类型的对象,都可以作为列表元素。列表中的元素可以重复出现。例如:

```
[15, 20, 25, 30, 20, 10]
['Sunday', 'Monday', 'Tuesday']
[['Mary', 1992,'female',True,5], ['Tom', 1989,'male',False,8]]
[1, 2.0, 'a word', print(5), True, [3,8], (3,8), {'mykey': 'myvalue'}]
```

都是合法的列表对象。

3.2.1 列表的创建与删除

1. 用赋值语句创建列表

在已知列表元素个数及其值时,可以用赋值语句直接创建列表。

```
In [1]: a_list=[1,2.0,[3,4],'a word',"better"]
        a_list
```

```
Out[1]: [1, 2.0, [3, 4], 'a word', 'better']
In [2]: b_list=[]
        b_list
Out[2]: []
```

2. 调用 list()函数创建列表

list()函数可将一个可迭代对象转换为列表。

```
In [1]: a_list =list("hello") #将字符串转换为列表
        a_list
Out[1]: ['h', 'e', 'l', 'l', 'o']
In [2]: b_list=list() #创建一个空列表
        b_list
Out[2]: []
```

3. 列表的删除

用 del 命令删除列表,释放所占的存储空间。

```
In [1]: a_list
Out[1]: ['h', 'e', 'l', 'l', 'o']
In [2]: del a_list                              #删除列表 a_list
        a_list                                  #a_list 已经不存在了,抛出异常
Out[2]: --------------NameError    Traceback (most recent call last)
         <iPython-input-10-32d89a22c1f5>in <module>()
               1 del a_list                     #删除列表 a_list
         ---->2 a_list
         NameError: name 'a_list' is not defined
```

3.2.2　列表的基本操作

1. 列表元素的获取

方法一：在列表对象右边的方括号内输入对应的索引,获取列表中的某个元素。
格式：列表对象[索引]

```
In [1]: a_list=[1,2.0,"a word",[3,8],(3,8,9),{'key1': 128,'key2': 'myvalue',
        3: 'a number'}]
        a_list[-1]
Out[1]: {'key1': 128, 'key2': 'myvalue', 3: 'a number'}
In [2]: print(a_list[0],a_list[3],a_list[-3],a_list[-2])
Out[2]: 1 [3, 8] [3, 8] (3, 8, 9)
```

Python 可以直接访问一个序列元素,无须先将序列赋值给一个变量。

```
In [1]: print(["ABCD",'Hello','Python','University'][2])
Out[1]: Python
```

注意,当传入的索引超出列表正向索引或负向索引范围时,即索引取值小于第一个元素的负向索引或大于最后一个元素的正向索引时,Python 会抛出异常。

```
In [1]: a_list[-7]
Out[1]: ----------IndexError Traceback (most recent call last) <iPython-
        input-18-f9b61e944de4>in <module>
        ---->1 a_list[-7]
        IndexError: list index out of range
In [2]: a_list[6]
Out[2]: ----------IndexError Traceback (most recent call last) <iPython-
        input-19-6ac097c0a2a1>in <module>
        ---->1 a_list[6]
        IndexError: list index out of range
```

方法二:切片操作获取列表中的多个元素。

使用切片操作返回列表中部分元素组成的新列表。

```
In [1]: a=[1,2,3,4,5,6]
        b1=a[: ]          #省略全部,代表截取全部内容,可以将一个列表复制给另一个列表
        print(b1)
Out[1]: [1, 2, 3, 4, 5, 6]
In [2]: b=a[0: -1: 1]     #从索引 0 开始到结束,每次增加 1,截取内容,不包含结束索引
        print(b)
Out[2]: [1, 2, 3, 4, 5]
In [3]: c1=a[: 3]         #省略起始索引和步长。默认从索引 0 开始,默认步长为 1
        print(c1)         #结束索引为 3,切片操作结果不包含结束索引的值
Out[3]: [1, 2, 3]
In [4]: c=a[0: 5: 3]      #从第一个位置到第六个位置,每三个位置取一个值
        print(c)
Out[4]: [1, 4]
In [5]: d=a[5: 0: -1]     #负向取值
        print(d)
Out[5]: [6, 5, 4, 3, 2]
In [6]: d1=a[: : -1]
        print(d1)
Out[6]: [6, 5, 4, 3, 2, 1]
```

与使用索引访问列表元素不同,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,保证了代码的健壮性。

```
In [1]: print(a[0: 100])                    #切片结束位置大于列表长度,在列表末尾截断
Out[1]: [1, 2, 3, 4, 5, 6]
In [2]: print(a[100: ])                     #切片结束位置大于列表长度,返回空列表
Out[2]: []
```

2. 遍历列表元素

使用 for 循环遍历列表元素,常用以下两种方法。

方法一：隐藏列表长度,操作较为便利。

```
In [1]: a_list=range(10)
        for i in a_list:
          print(i,end=' ')
Out[1]: 0 1 2 3 4 5 6 7 8 9
```

方法二：使用 len()函数计算列表长度,使用 range()函数返回列表元素序列,通过索引遍历列表元素。

```
In [1]: for i in range(len(a_list)):
           print(a_list[i],end=' ')
Out[1]: 0 1 2 3 4 5 6 7 8 9
```

3. 列表元素的修改

方法一：利用索引修改列表元素。

列表是可变序列,可以直接指定列表索引的取值来修改列表元素。

```
In [1]: a_list =[0, 1, 2, 3, 4, 5]
        a_list[0]=100
        a_list
Out[1]: [100, 1, 2, 3, 4, 5]
```

方法二：使用切片操作批量修改列表元素。

```
In [1]: a_list =[0,1,2,3,4,5]
        a_list[0: 3] =[100,100,100]
        a_list
Out[1]: [100, 100, 100, 3, 4, 5]
```

4. 列表元素的添加

方法一：使用 append()方法向列表尾部添加一个新元素。

```
In [1]: a_list=list(range(5))
        a_list.append(True)
        a_list
Out[1]: [0, 1, 2, 3, 4, True]
```

方法二：使用 extend()方法向列表尾部添加一个可迭代对象中的所有元素。

```
In [1]: a_list=list(range(5))
        a_list.extend([True,'c',5])
        a_list
Out[1]: [0, 1, 2, 3, 4, True, 'c', 5]
```

方法三：使用 insert()方法向列表任意位置插入一个新元素。

该方法需要两个参数，第一个参数为插入位置，第二个参数为需要插入的元素。

```
In [1]: a_list=list(range(5))
        a_list.insert(3,'c')
        a_list
Out[1]: [0, 1, 2, 'c', 3, 4]
```

5. 列表元素随机排序

shuffle()函数用于打乱列表元素的顺序，将列表中的所有元素随机排序。shuffle()函数是不能直接访问的，需要首先导入 random 模块，然后通过 random 对象调用该函数。shuffle()函数不会生成新的列表，只是将原列表的次序打乱，因此不可以将 shuffle()函数的返回值赋值给另外一个列表，只能在原列表的基础上操作。

```
In [1]: import random
        a_list =[1,2,3,4,5]
        random.shuffle(a_list)
        print(a_list)
Out[1]: [5, 2, 1, 4, 3]
```

6. 列表元素的删除

方法一：使用 del 命令删除指定位置的列表元素。

```
In [1]: a_list =[0, 1, 2, 3, 4, 5]
        del a_list[0]                    #删除下标为 0 的元素
        a_list
Out[1]: [1, 2, 3, 4, 5]
```

方法二：使用 pop()方法删除并返回指定位置的列表元素，省略参数时弹出最后一

个元素。如果给定的下标超出了列表范围，则抛出异常。

```
In [1]: a_list =[0, 1, 2, 3, 4, 5]
        a_list.pop()
        a_list
Out[1]: [0, 1, 2, 3, 4]
In [2]: a_list.pop(6)
Out[2]: ------------IndexError  Traceback (most recent call last)
        <iPython-input-31-1d77944e9b78>in <module>()
        ---->1 a_list.pop(6)
        IndexError: pop index out of range
```

方法三：使用 remove()方法删除首次出现的指定元素。如果列表中不存在该元素，则抛出异常 ValueError。

```
In [1]: a_list =['x', 'y', 'z', 'a', 'b', 'z', 'c']
        a_list.remove('z')
        a_list
Out[1]: ['x', 'y', 'a', 'b', 'z', 'c']
```

方法四：利用切片操作删除多个列表元素。

```
In [1]: a_list=list(range(10))
        a_list[3: 8]=[]
        a_list
Out[1]: [0, 1, 2, 8, 9]
```

方法五：清空所有列表元素。

```
In [1]: a_list =list(range(10))
        a_list.clear()
        a_list
Out[1]: []
```

当增加或删除列表元素时，列表对象自动进行内存扩展或内存收缩，从而保证元素之间没有缝隙。Python 列表内存的自动管理功能在大幅减少程序员负担的同时，可能带来程序执行效率的降低，甚至可能导致意外的错误结果。为了避免可能涉及的大量列表元素移动，应尽量从列表尾部进行元素的增加或删除操作，这有助于大幅提高列表处理速度。

3.2.3　列表可用操作符

常用的列表操作符包括：系统提供的标准操作符和为序列提供的专门操作符。

1. 标准操作符

用于对象值比较的关系运算符：>、<、>=、<=、==和!=。
用于对象同一性测试的身份运算符：is 和 is not。
逻辑运算符：not、and、or。

```
In [1]: a_list=['ABC','Hello',1,2,3]
        b_list=['ABC','Hello']
        a_list>b_list                    #逐个比较列表元素,直到一方列表元素胜出
Out[1]: True
```

2. 序列专用操作符

为序列提供的专用操作包括连接、重复、切片和成员关系测试等。
连接操作将多个列表连接起来。连接操作符(+)两边的对象必须属于相同数据类型。

```
In [1]: a_list+b_list
Out[1]: ['ABC', 'Hello', 1, 2, 3, 'ABC', 'Hello']
In [2]: c_string='Python'
        a_list+c_string
Out[2]: ---------TypeError  Traceback (most recent call last)
        <iPython-input-46-669a4cfc2469>in <module>
        ---->1 a_list+c_string
        TypeError: can only concatenate list (not "str") to list
```

重复操作符(*)主要用于序列类型。

```
In [1]: b_list*3
Out[1]: ['ABC', 'Hello', 'ABC', 'Hello', 'ABC', 'Hello']
```

成员测试运算符用于检查一个对象是否是给定序列对象中的元素。

```
In [1]: a_list=['ABC','Hello',1,2,3]
        b_list=['ABC','Hello']
        b_list in a_list
Out[1]: False
In [2]: 'ABC'in a_list
Out[2]: True
In [3]: 4 in a_list
Out[3]: False
```

3.2.4 列表常用函数

Python 列表的常用函数基本上也适用于元组和字符串。

Python 中常用于列表对象的内置函数有：len()、max()、min()、sum()、list()、tuple()、enumerate()、zip()、sorted()、reversed()等。

1. len()函数

len()函数返回列表中元素的个数，同样适用于元组、字典、集合和字符串。

```
In [1]: a_list =[1,2,4,[4,5]]
        len(a_list)
Out[1]: 4
```

2. max()、min()、sum()函数

max()函数和 min()函数分别返回列表元素的最大值和最小值，参数为只包含字符串元素的列表或只有数字元素构成的列表，同样适用于元组、字典、集合或 range 对象。

sum()函数用于列表的求和操作，同样适用于元组求和或 range 对象求和。

```
In [1]: a_list=['a','A','F','fa','f']
        max(a_list)
Out[1]: 'fa'
In [2]: min(a_list)
Out[2]: 'A'
In [3]: sum(range(1,11))
Out[3]: 55
```

3. list()、tuple()函数

接受可迭代对象作为参数，list()函数将可迭代对象转换为列表，tuple()函数将可迭代对象转换为元组。

4. zip()函数

接受可迭代对象(如列表、元组或字符串)为参数，将其中的元素对应打包生成一个元组元素，然后返回由这些元组元素组成的可迭代 zip 对象。需要注意的是，当两个列表中的元素个数不相等时，舍弃其中多余的元素。

```
In [1]: a_list=['a','A','F','fa','f']
        b_list=[1,2,3]
        c_list=zip(a_list,b_list)
        c_list
Out[1]: <zip at 0x7f094cd73870>
In [2]: list(c_list)
Out[2]: [('a', 1), ('A', 2), ('F', 3)]
```

5. eunmerate()函数

eunmerate()函数将一个可迭代对象组合为一个数据对的序列,其中每个数据对元素表示为一个包含索引和数据值的元组。eunmerate()函数同样适用于元组和字符串。

```
In [1]: t=[2,4,5]
        for i in enumerate(t):
            print(i)
Out[1]: (0, 2)
        (1, 4)
        (2, 5)
```

6. sorted()函数

sorted()函数可以将列表元素按照正序或者逆序排列,返回一个排序之后的新列表,内置函数 sorted()并不改变原列表元素的排列次序。实际上,针对任何可迭代对象,sorted()函数的返回值都是一个排序之后的新列表。

```
In [1]: b_list=[4,3,2,8,1,6,9,5,3]
        sorted(b_list,reverse=True)
Out[1]: [9, 8, 6, 5, 4, 3, 3, 2, 1]
In [2]: b_list
Out[2]: [4, 3, 2, 8, 1, 6, 9, 5, 3]
In [3]: sorted(b_list)
Out[3]: [1, 2, 3, 3, 4, 5, 6, 8, 9]
```

7. reversed()函数

reversed()函数将列表元素逆序排列,返回一个迭代器对象,但并不修改原列表元素的位置。实际上,对于给定的可迭代对象,如列表、元组、字符串以及 range()函数等,内置函数 reversed()都返回一个逆序排列的 reversed 对象,可以利用此迭代器遍历其中的元素。

```
In [1]: b_list=[4,3,2,8,1,6,9,5,3]
        b_reverse=reversed(b_list)
        b_reverse
Out[1]: <list_reverseiterator at 0x7f54ac9ba890>
In [2]: list(b_reverse)
Out[2]: [3, 5, 9, 6, 1, 8, 2, 3, 4]
In [3]: b_list=[4,3,2,8,1,6,9,5,3]
        b_reverse=reversed(b_list)
        for i in b_reverse:
            print(i,end=" ")
```

```
Out[3]: 3 5 9 6 1 8 2 3 4
In [4]: list(b_reverse)
Out[4]: []
```

代码段最后的 list()函数没有输出任何内容，这是因为执行 for 循环的过程中，完成可迭代对象的遍历。如果需要再次查看迭代器的内容，需要重新创建该迭代器对象。

3.2.5　列表常用方法

列表作为一种序列类型，具有序列通用的操作方法，同时也具有列表特有的操作方法。使用列表对象常用方法的格式：[列表对象].<方法名>。除了列表元素的添加、删除方法，Python 还提供了其他常用方法，方便对列表元素的操作。

1. count()方法

统计指定元素在列表中出现的次数。

```
In [1]: a_list=list(range(5))+[3,1,3,5,3]
        a_list
Out[1]: [0, 1, 2, 3, 4, 3, 1, 3, 5, 3]
In [2]: a_list.count(3)
Out[2]: 4
```

2. index()方法

返回指定元素在列表中首次出现的索引。如果该元素不在列表中则抛出异常 ValueError。

```
In [1]: str_list=list("hello")
        str_list.index('l')
Out[1]: 2
```

3. sort()方法

按照指定规则原地排列列表中的元素，默认规则是将所有元素从小到大升序排列。

```
In [1]: a_list=[3,4,5,13,14,15]
        import random
        random.shuffle(a_list)                          #打乱顺序
        a_list
Out[1]: [4, 5, 15, 13, 3, 14]
In [2]: a_list.sort()                                   #默认规则是升序排列
        a_list
```

```
Out[2]: [3, 4, 5, 13, 14, 15]
In [3]: a_list.sort(reverse=True)                    #降序排列
        a_list
Out[3]: [15, 14, 13, 5, 4, 3]
In [4]: a_list.sort (key=lambda x: len(str(x)))      #自定义排序
        a_list
Out[4]: [5, 4, 3, 15, 14, 13]
```

若列表元素的数据类型不同,则不能比较大小,所以不能使用 sort()方法排序。

```
In [1]: a_list =['abc','bcd','cef',97,65,98,99]
        a_list.sort()
Out[1]: ------------------TypeError Traceback (most recent call last)
        <iPython-input-57-6abf35b10e50>in <module>()
               1 a_list=['abc','bcd','cef',97,65,98,99]
        ---->2 a_list.sort()
        TypeError: unorderable types: int() <str()
```

必要时,可以将列表元素转换成相同类型的数据,再使用 sort()方法排序。

```
In [1]: a_list.sort(key=str,reverse=True) #自定义排序
       a_list
Out[1]: ['cef', 'bcd', 'abc', 99, 98, 97, 65]
```

sort()方法和内置函数 sorted()都可以对列表元素排序,但 sort()方法是原地排序,同时修改原列表元素的顺序;而内置函数 sorted()返回新列表,并不会修改原列表元素的顺序。

```
In [1]: a_list=list(range(5))+[5,4,3,2,1]
        sorted(a_list)
Out[1]: [0, 1, 1, 2, 2, 3, 3, 4, 4, 5]
```

4. reverse()方法

原地逆序排列列表元素。

```
In [1]: a_list=[3,4,5,13,14,15]
        import random
        random.shuffle(a_list)                       #打乱顺序
        a_list
Out[1]: [15, 3, 14, 4, 13, 5]
In [2]: a_list.reverse()
       a_list
Out[2]: [5, 13, 4, 14, 3, 15]
```

reverse()方法用于原地逆序排列列表元素，同时修改原列表元素的顺序；而内置函数reversed()将列表逆序排列后的结果放在一个可迭代对象中，并不会修改原列表元素的顺序。

```
In [1]: new_list=reversed(a_list)
        new_list
Out[1]: <list_reverseiterator at 0x70cd81ef0>
In [2]: list(new_list)
Out[2]: [15, 3, 14, 4, 13, 5]
```

5. copy()方法

复制列表中的所有元素，生成一个新列表。

```
In [1]: a_list=[3,4,5,13,14,15]
        new_list=a_list.copy()          #copy()方法产生列表的副本
        new_list
Out[1]: [3, 4, 5, 13, 14, 15]
In [2]: a_list.clear()
        new_list
Out[2]: [3, 4, 5, 13, 14, 15]
```

对于基本数据类型(如整数或字符串)，可以使用等号进行对象赋值。对于列表类型，直接赋值方式只是为列表增加了一个别名，不能产生新列表，而使用 copy()方法才能复制列表，返回列表的副本。

```
In [1]: a_list=[3,4,5,13,14,15]
        b_list=a_list             #直接赋值只是为列表增加别名
        a_list.clear()
        b_list
Out[1]: []
```

例 3-1　计算基本统计值，输出一组数据的平均值、方差和众数。

分析：①为了便于功能重用，可以将每个功能用自定义函数实现；②计算一组数据的基本统计值，但是这组数据的元素个数不确定，因此使用可变序列中的列表来存储。

```
In [1]: def getNum():                          #获取用户输入的数据,数目不确定
            nums =[]
            iNumStr =input("请输入数字(回车退出)：")
            while iNumStr !="":
                  nums.append(eval(iNumStr))
                  iNumStr =input("请输入数字(回车退出)：")
                  return nums
        def mean(numbers):                     #计算平均值
            s =0.0
            for num in numbers:
```

```
            s =s +num
        return s / len(numbers)
    def dev(numbers, mean):                          #计算方差
        sdev =0.0
        for num in numbers:
            sdev =sdev +(num -mean) * * 2
        return pow(sdev / (len(numbers)-1), 0.5)
    def median(numbers):                             #计算中位数
        sorted(numbers)
        size =len(numbers)
        if size %2==0:
            med =(numbers[size//2-1] +numbers[size//2])/2
        else:
            med =numbers[size//2]
        return med
    n =getNum()
    m =mean(n)
    print("平均值:{}方差:{:.2}中位数:{}".format(m, dev(n,m),median
    (n)))
out[1]:请输入数字(回车退出):79
       请输入数字(回车退出):82
       请输入数字(回车退出):69
       请输入数字(回车退出):54
       请输入数字(回车退出):平均值:71.0方差:1.3e+01中位数:75.5
```

3.3 元　　组

元组是不可变有序序列。一旦创建了元组就不允许改变其中元素的值,也无法为元组增加或删除元素。元组中的元素放在一对圆括号“()”中,元素之间用逗号隔开,元素可以是任意数据类型。例如:

```
(15, 20, 25, 30, 20, 10)
('Sunday', 'Monday', 'Tuesday')
(['Mary', 1992,'female',True,5], ('Tom', 1989,'male',False,8))
(1, 2.0, 'a word', print(5), True, [3,8], (3,8), {'mykey': 'myvalue'})
```

都是合法的元组对象。

3.3.1 元组的创建与删除

元组的创建与删除与列表类似,主要采用以下方法。

1. 用赋值语句创建元组

将一个元组对象赋值给一个变量，就创建了一个元组变量。当一个元组只包含一个元素时，该元素后面也必须有逗号；当一个元组的圆括号内没有元素时，表示创建了一个空元组。

```
In [1]: d_tuple=(1, 2.0, 'a word', print(5), True, [3,8], (3,8), {'mykey': '
        myvalue'})
        d_tuple
out[1]: (1, 2.0, 'a word', None, True, [3, 8], (3, 8), {'mykey': 'myvalue'})
In [2]: type(d_tuple)
out[2]: tuple
In [3]: a_tuple =(1,)
        a_tuple                                  #创建只包含一个元素的元组
out[3]: (1,)
In [4]: b_tuple =()
        b_tuple                                  #创建一个空元组
out[4]: ()
```

实际上，在使用非空元组时，一对圆括号可以省略，Python 将一组用逗号分隔的数据自动默认为元组类型。

```
In [1]: c_tuple =1,2,3                           #用逗号分隔的多个数据
        c_tuple
out[1]: (1, 2, 3)
In [2]: d_tuple =1,                              #用逗号分隔的一个数据
        d_tuple
out[2]: (1,)
In [3]: type(d_tuple)                            #用逗号分隔的一个数据也是元组
out[3]: tuple
In [4]: t =1                                     #如不加逗号，则不是元组
        type(t)
out[4]: int
```

2. 使用 tuple()函数创建元组

tuple()函数可以将一个可迭代对象转换为元组。

```
In [1]: a_tuple=tuple([1,2,3,4])                 #将列表转换为元组
        print(a_tuple)
        b_tuple=tuple(range(5))                  #将 range 对象转换为元组
        print(b_tuple)
        c_tuple=tuple("hello")                   #将字符串转换为元组
        print(c_tuple)
```

```
out[1]: (1, 2, 3, 4)
        (0, 1, 2, 3, 4)
        ('h', 'e', 'l', 'l', 'o')
```

3. 元组的删除

对于元组而言,用 del 命令可以删除整个元组对象,而不能只删除元组中的部分元素,因为元组属于不可变序列。

```
In [1]: a_tuple = (2,3,4)                                    #创建一个元组
        a_tuple
out[1]: (2, 3, 4)
In [2]: del a_tuple[2]                                       #删除元组元素
out[2]: ----------------------TypeError  Traceback (most recent call last)
        <iPython-input-6-b012d3a47b80>in <module>()
        ---->1 del a_tuple[2]                                #删除元组
        TypeError: 'tuple' object doesn't support item deletion
In [3]: del a_tuple                                          #删除元组
        a_tuple                                              #此时元组已经不存在
out[3]: -------------------NameError   Traceback (most recent call last)
        <iPython-input-3-30499c72454a>in <module>()
        1 del a_tuple                                        #删除元组
        ---->2 a_tuple                                       #此时元组已经不存在
        NameError: name 'a_tuple' is not defined
```

3.3.2 元组与列表的区别

元组是类似于列表的一种数据结构,可以看作轻量级的列表。除了定义形式相似之外,二者还有许多相似之处,例如,都属于有序序列,支持双向索引和切片操作,支持运算符“+”“*”和“in”,对内置函数的支持也是大同小异。

二者的区别和联系如下。

(1) 列表是可变的,而元组是不可变的。列表支持针对部分元素的增加、删除操作,也可以修改列表中的元素值。相反,元组中的数据一旦定义就不允许通过任何方式更改,因此元组没有提供 append()、extend()和 insert()等方法,无法为元组增加元素;元组也没有 remove()和 pop()方法,不支持针对元组元素的 del 操作,不能删除元组中的部分元素,只能删除整个元组对象。可以通过切片操作访问元组中的元素,但是不支持使用切片操作对元组元素进行修改、增加和删除操作。

(2) 元组和列表可以相互转换。Python 内置函数 tuple()可以接收一个列表、字符串等可迭代对象作为参数,返回包含同样元素的元组;而 list()函数可以接收一个元组、字符串等可迭代对象作为参数,返回一个列表。从实现效果上看,tuple()函数可以理解为将列表“冻结”使之不可变,list()函数则是将元组“融化”使之可变。

（3）元组的访问和处理速度比列表更快，开销更小。如果定义了一系列常量，主要进行遍历等操作，而无须改变元素的取值，一般建议使用元组而非列表数据类型。

（4）元组可以使代码更安全。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的值，这时若使用列表则达不到需要的效果，甚至可以理解为元组对不需要修改的数据进行了“写保护”，在实现上不允许修改元素值，从而使代码更安全。实际上，Python 自定义函数中的 return 语句返回多个值时，这些值会形成一个元组数据类型，在一定程度上也起到保护数据、防止返回值被无意修改的作用。

```
In [1]: def multiply(x,y=10):
                    return x * y,x+y
        s=multiply(90,2)
        print('s=',s)                                    #返回元组数据类型
        a,b=multiply(90,2)
        print('a=',a)                                    #返回基本数据类型
        print('b=',b)                                    #返回基本数据类型
out[1]: s=(180, 92)
        a=180
        b=92
```

（5）二者与其他组合数据类型的关系。作为不可变序列，元组可以用作字典的键，也可以作为集合的元素。列表是可变序列，不可以用作字典的键，列表或包含列表的元组也不可以作为集合的元素。

元组是不可修改的，因此在列表中用于改变元素的方法和函数都不能用在元组上。元组也是一种序列类型，因此针对序列的通用操作都可以作用在元组上。需要注意的是，如果元组的元素是可变序列，那么该元素仍然可以修改。

```
In [1]: a_tuple=('a','b',1,[3,5],["A",'b','T'])
        a_tuple[4][1]=9
        a_tuple
out[1]: ('a', 'b', 1, [3, 5], ['A', 9, 'T'])
```

3.4　字　　典

作为一种有序序列，列表只接受基于位置的索引。如果数据量巨大的话，要记住每个元素及其在列表中的位置是不现实的。Python 提供了通过“键”数据查找“值”数据，表示映射关系的数据组织形式，这就是字典。

字典是一种典型的映射类型，由若干**键值对**元素构成的**无序可变序列**。字典中每个元素由“键(**key**)：值(**value**)”构成，通过键可以找到其对应的值。换一个角度讲，字典的“键”代表一个关键词，而字典的“值”代表这个关键词对应的内容。在使用字典时，只要查找字典前面的关键词就可找到该关键词对应的内容。

字典的键可以是任意不可变数据类型,如整数、浮点数、元组等。一个字典元素的键是唯一的,无法修改的;而值是可变的,可重复、可修改,可以是任何 Python 对象。

3.4.1 字典的创建和删除

1. 利用“{}”创建字典

字典中每个元素是一个键值对,其中,“键”和“值”用冒号隔开,相邻元素之间用逗号分隔,所有元素放在一对大括号“{}”中。

```
In [1]: a_dict={65: 'A',98: 'b',67: 'C','a': 97,'B': 66}
        a_dict
out[1]: {65: 'A', 98: 'b', 67: 'C', 'a': 97, 'B': 66}
In [2]: type(a_dict)
out[2]: dict
```

直接使用一对大括号可以创建一个空字典对象。

```
In [1]: a_dict={}
        print(a_dict,type(a_dict))
out[1]: {} <class 'dict'>
```

2. 使用 dict()函数创建字典

内置函数 dict()可以将一个可迭代对象转换为字典。

```
In [1]: keys=['a','b','c','d']
        values=[97,98,99,100]
        b_dict=dict(zip(keys,values))
        b_dict
out[1]: {'a': 97, 'b': 98, 'c': 99, 'd': 100}
```

dict()函数的参数也可以是形如“键=值”的数据对。

```
In [1]: c_dict=dict(name='Tom',age=18)
        c_dict
out[1]: {'age': 18, 'name': 'Tom'}
```

dict()函数的参数为空,表示创建一个空字典。

```
In [1]: d_dict=dict()
        d_dict
out[1]: {}
```

3. 使用 dict.fromkeys()方法创建字典

根据给定的键，使用 dict.fromkeys()方法可以创建一个具有相同值的字典。如果没有给出字典的值，则默认为空。

```
In [1]: e_dict=dict.fromkeys(['Tom','Mary'],'18')
        e_dict
out[1]: {'Mary': '18', 'Tom': '18'}
In [2]: e_dict=dict.fromkeys(['Tom','Mary'])
        e_dict
out[2]: {'Mary': None, 'Tom': None}
```

4. 字典的删除

当不需要时，可以用 del 命令删除字典，释放内存空间，与删除列表或元组类似。

3.4.2　字典的基本操作

与列表和元组的索引类似，可以使用字典中的“键”访问对应的字典元素。字典元素的键与值是一一对应的，不能脱离对方而存在。

1. 字典元素的获取

方法一：以键为索引获取元素的值。

字典中的元素是键值对，访问字典元素最常用的方法是以键为索引获取指定元素的值。

```
In [1]: a_dict={65: 'A',98: 'b',67: 'C','a': 97,'B': 66}
        a_dict[67]
out[1]: 'C'
```

使用“键”作为索引访问字典元素的“值”，若指定的“键”不存在则抛出异常。

```
In [1]: a_dict={65: 'A',98: 'b',67: 'C','a': 97,'B': 66}
        a_dict[77]
out[1]: -------------------KeyError  Traceback (most recent call last)
        <iPython-input-12-daf717106fe3>in <module>()
              1 a_dict={65: 'A',98: 'b',67: 'C','a': 97,'B': 66}
        ---->2 a_dict[77]
        KeyError: 77
```

方法二：使用 get()方法获取指定键的元素值。

使用字典对象的 get()方法可以获取指定“键”对应的“值”，并且在指定“键”不存在的情况下可以读取参数指定的值并返回，如果省略该参数，则不返回数据。

```
In [1]: a_dict={'B': 66, 65: 'A', 98: 'b', 67: 'C', 'a': 97}
        a_dict.get(65)                          #当键存在则返回相应的值
out[1]: 'A'
In [2]: a_dict.get(66,'NO')                     #当键不存在时返回指定的值
out[2]: 'NO'
In [3]: a_dict.get(66)                          #键不存在,未指定返回值,则不返回数据
```

2. 字典元素的遍历

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。

```
In [1]: dict1={1: 'A',2: 'B',3: 'C',4: 'D',5: ['E','e']}
        dict1.items
out[1]: <function dict.items>
In [2]: dict1.items()
out[2]: dict_items([(1, 'A'), (2, 'B'), (3, 'C'), (4, 'D'), (5, ['E', 'e'])])
In [3]: dict1.values()
out[3]: dict_values(['A', 'B', 'C', 'D', ['E', 'e']])
In [4]: dict1.keys()
out[4]: dict_keys([1, 2, 3, 4, 5])
```

使用 for 循环可以遍历字典中的元素。需要注意的是,迭代访问字典元素时,默认情况下遍历字典元素的“键”;如果需要遍历字典元素的“值”,可以使用 values()方法明确指定;如果需要遍历字典元素的“键值对”,可以使用 items()方法明确指定。

```
In [1]: d={65: 'A', 68: 'D', 67: 'C', 69: 'E', 66: 'B'}
        for i in d:                             #默认遍历的是字典的“键”
            print(i,end=" ")
out[1]: 65 66 67 68 69
In [2]: for i in d.values():                    #遍历字典的“值”
            print(i,end=" ")
out[2]: A B C D E
In [3]: for i in d.items():                     #遍历字典元素,即“键值对”
            print(i,end=" ")
out[3]: (65, 'A') (66, 'B') (67, 'C') (68, 'D') (69, 'E')
```

使用 len()、max()、min()、sum()、sorted()、enunerate()、map()、filter()等内置函数以及成员测试运算符 in 对字典对象操作时,也遵循同样的约定。