Python程序设计

第五讲 批量数据类型与操作
序列数据（一）

张 华
WHU

序列数据：元组和列表

- 序列数据及其基本操作
- 元组
- 列表

序列数据

Python内置的序列数据类型

- 元组（**tuple**）
  - 也称为定值表，用于存储值固定不变的数据表。
- 列表（**list**）
  - 用于存储值可变的数据表。
- 字符串（**str**）
  - 字符串是包括若干字符的序列数据，支持序列数据的基本操作。
- 字节数据（**bytes**和**bytearray**）
  - 字节序列。

序列数据的基本操作

用内置函数操作序列数据

- len()、max()、min()，获取系列的长度、系列中元素最大值、系列中元素最小值。

```
>>> s='abcdefg'
>>> len(s)
7
>>> max(s)
'g'
>>> min(s)
'a'
>>> s2=''
>>> len(s2)
0
```

```
>>> t=(10,2,3)
>>> len(t)
3
>>> max(t)
10
>>> min(t)
2
>>> t2=()
>>> len(t2)
0
```

```
>>> lst=[1,2,9,5,4]
>>> len(lst)
5
>>> max(lst)
9
>>> min(lst)
1
>>> lst2=[ ]
>>> len(lst2)
0
```

```
>>> b=b'ABCD'
>>> len(b)
4
>>> max(b)
68
>>> min(b)
65
>>> b2=b''
>>> len(b2)
0
```

序列数据的基本操作

用内置函数操作序列数据

- sum()获取列表或元组中各元素之和。

```
>>> t1=(1,2,3,4)
>>> sum(t1)     #输出：10
>>> t2=(1,'a',2)
>>> sum(t2)  #TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>> s='1234'
>>> sum(s)  #TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

序列数据的基本操作

通过索引访问序列中的数据项

s[i] #访问序列s在索引i处的数据项

s[-5]	s[-4]	s[-3]	s[-2]	s[-1]
'bonus'	-228	'purple'	'100'	19.84
s[0]	s[1]	s[2]	s[3]	s[4]

```
>>> s='abcdef'
>>> s[0]
'a'
>>> s[2]
'c'
>>> s[-1]
'f'
>>> s[-3]
'd'
```

```
>>> t=('a','e','i','o','u')
>>> t[0]
'a'
>>> t[1]
'e'
>>> t[-1]
'u'
>>> t[-5]
'a'
```

```
>>> lst=[1,2,3,4,5]
>>> lst[0]
1
>>> lst
[1, 2, 3, 4, 5]
>>> lst[2]='a'
>>> lst[-2]='b'
>>> lst
[1, 2, 'a', 'b', 5]
```

```
>>> b=b'ABCDEF'
>>> b[0]
65
>>> b[1]
66
>>> b[-1]
70
>>> b[-2]
69
```

序列数据的基本操作

切片操作（取序列的一部分元素，即子序列）

s[i:j]　或　s[i:j:k]

- 其中，i为子序列的开始下标，j为子序列结束下标的后一个下标，k为步长。
  - 如果省略i，则从下标0开始。
  - 如果省略j，则直到序列结束为止。
  - 如果省略k，则步长为1。
- 注意：下标可以为负数。如果截取范围内没有数据，则返回空序列；否则超过下标范围，不报错。

```
>>> s='abcdef'
>>> s[1:3]
'bc'
```

```
>>> t=('a','e','i','o','u')
>>> t[-2:-1]
('o',)
```

```
>>> lst=[1,2,3,4,5]
>>> lst[:2]
[1, 2]
```

```
>>> b=b'ABCDEF'
>>> b[2:2]
b''
```

序列数据的基本操作

切片操作 s[i:j] 或 s[i:j:k]

```
>>> s[3:10]
'def'
>>> s[8:2]
''
>>> s[:]
'abcdef'
>>> s[:2]
'ab'
>>> s[::2]
'ace'
>>> s[::-1]
'fedcba'
```

```
>>> t[-2:]
('o', 'u')
>>> t[-99:-5]
()
>>> t[-99:-3]
('a', 'e')
>>> t[::]
('a', 'e', 'i', 'o', 'u')
>>> t[1:-1]
('e', 'i', 'o')
>>> t[1::2]
('e', 'o')
```

```
>>> lst[:1]=[]
>>> lst
[2, 3, 4, 5]
>>> lst[:2]
[2, 3]
>>> lst[:2]='a'
>>> lst[1:]='b'
>>> lst
['a', 'b']
>>> del lst[:1]
>>> lst
['b']
```

```
>>> b[0:1]
b'A'
>>> b[1:2]
b'B'
>>> b[2:2]
b''
>>> b[-1:]
b'F'
>>> b[-2:-1]
b'E'
>>> b[0:len(b)]
b'ABCDEF'
```

序列数据的基本操作

连接和重复操作

s1 + s2　或者　s*n　或者　n*s

```
>>> s1='abc'
>>> s2='xyz'
>>> s1+s2
'abcxyz'
>>> s1*3
'abcabcabc'
>>> s1 += s2
>>> s1
'abcxyz'
>>> s2 *= 2
>>> s2
'xyzxyz'
```

```
>>> t1=(1,2)
>>> t2=('a','b')
>>> t1+t2
(1, 2, 'a', 'b')
>>> t1*2
(1, 2, 1, 2)
>>> t1 += t2
>>> t1
(1, 2, 'a', 'b')
>>> t2 *= 2
>>> t2
('a', 'b', 'a', 'b')
```

```
>>> lst1=[1,2]
>>> lst2=['a','b']
>>> lst1+ lst2
[1, 2, 'a', 'b']
>>> 2 * lst2
['a', 'b', 'a', 'b']
>>> lst1 += lst2
>>> lst1
[1, 2, 'a', 'b']
>>> lst2 *=2
>>> lst2
['a', 'b', 'a', 'b']
```

```
>>> b1=b'ABC'
>>> b2=b'XYZ'
>>> b1+b2
b'ABCXYZ'
>>> b1*3
b'ABCABCABC'
>>> b1+=b2
>>> b1
b'ABCXYZ'
>>> b2*=2
>>> b2
b'XYZXYZ'
```

序列数据的基本操作

成员关系操作与序列对象的方法

```
x in s              #检测x是否在s中
x not in s          #检测x是否不在s中
s.count(x)          #统计x在s中出现的次数
s.index(x[,i[,j]]) #返回x在s的位置i~j-1间第1次出现的位置
```

```
>>> s='Good, better, best!'
>>> 'o' in s
True
>>> 'g' not in s
True
>>> s.count('e')
3
>>> s.index('e', 10)
10
```

```
>>> t=('r', 'g', 'b')
>>> 'r' in t
True
>>> 'y' not in t
True
>>> t.count('r')
1
>>> t.index('g')
1
```

```
>>> lst=[1,2,3,2,1]
>>> 1 in lst
True
>>> 2 not in lst
False
>>> lst.count(1)
2
>>> lst.index(3)
2
```

```
>>> b=b'Oh, Jesus!'
>>> b'O' in b
True
>>> b'o' not in b
True
>>> b.count(b's')
2
>>> b.index(b's')
6
```

序列数据的基本操作

比较运算

```
>>> s1='abc'
>>> s2='abc'
>>> s3='abcd'
>>> s4='cba'
>>> s1 > s4
False
>>> s2 <= s3
True
>>> s1 == s2
True
>>> s1 != s3
True
>>> 'a' > 'A'
True
>>> 'a' >= ''
True
```

```
>>> t1=(1,2)
>>> t2=(1,2)
>>> t3=(1,2,3)
>>> t4=(2,1)
>>> t1<t4
True
>>> t1 <= t2
True
>>> t1 == t3
False
>>> t1 != t2
False
>>> t1 >= t3
False
>>> t4 > t3
True
```

```
>>> s1=['a','b']
>>> s2=['a','b']
>>> s3=['a','b','c']
>>> s4=['c','b','a']
>>> s1<s2
False
>>> s1<=s2
True
>>> s1==s2
True
>>> s1!=s3
True
>>> s1>=s3
False
>>> s4>s3
True
```

```
>>> b1=b'abc'
>>> b2=b'abc'
>>> b3=b'abcd'
>>> b4=b'ABCD'
>>> b1<b2
False
>>> b1<=b2
True
>>> b1==b2
True
>>> b1>=b3
False
>>> b3!=b4
True
>>> b4>b3
False
```

序列数据的基本操作

用内置函数对序列进行排序

sorted(iterable, key=None, reverse=False)

- 对原可迭代对象的元素排序，添加到一个新列表（对原对象没有影响），这个新列表是函数的返回值。
- 其中，key适用于计算比较键值的函数（带一个参数）。
- 如果reverse=True，则反向排序。

```
>>> s1='axd'
>>> sorted(s1)
['a', 'd', 'x']
>>> s2=(1,4,2)
>>> sorted(s2)
[1, 2, 4]
>>> sorted(s2,reverse=True)
[4, 2, 1]
>>> s3='abAC'
>>> sorted(s3, key=str.lower)
['a', 'A', 'b', 'C']
```

序列数据的基本操作

序列数据的拆封（或称为序列解包）

- （1）变量个数和序列长度相等

  变量1, 变量2, ... , 变量n = 序列或可迭代对象

```
>>> a,b,c = [12,'abc',34.56]
```

- （2）变量个数和序列长度不相等

  使用 *列表变量，将多个值作为列表赋值给列表变量

```
>>> x,*y,z = (1,2,3,4,5,6)
>>> y
[2, 3, 4, 5]
>>> type(y)
<class 'list'>
```

- （3）使用临时变量

```
>>> _,x,_=1,2,3
```

元组

元组和列表

- **Python中的元组和列表是任意数量的一组相关数据形成的一个整体。**
  - 其中的每一项可以是任意类型的数据项。
  - 各数据项之间按索引号排列并允许访问。
- **元组和列表的区别为：**
  - 元组是固定的，创建之后就不能改变其数据项。
  - 列表创建后允许修改或删除其中的数据项、插入新数据项。

元组

元组的基本操作

- （1）元组的字面表示
- （2）元组的类型构造器
- （3）元组元素的访问

元组的基本操作

元组的字面表示

- 元组一般使用圆括号来表示，数组项之间用逗号分隔
- 例如，用字面表示方式创建元组

```
>>> t=1,2,3
>>> t
(1, 2, 3)
>>> t=(1,2,3)
>>> t
(1, 2, 3)
```

```
>>> a=(1)
>>> a
1
>>> b=(1,)  #只包含一个元素的元组
>>> b
(1,)
```

元组的基本操作

元组的字面表示

- 例如，用字面表示方式创建元组
  - 数据项可以是相同类型的，也可以是不同类型的：

```
>>> t2="east","south","west","north"
>>> t2
('east', 'south', 'west', 'north')
>>> t3="0010110","张山","men",18
>>> t3
('0010110', '张山', 'men', 18)
```

元组的基本操作

元组的字面表示

- 例如，用字面表示方式创建元组
  - 可以定义空的元组，也可以定义嵌套的元组：

```
>>> t4=()
>>> t4
()
>>> t5=23,(5,8,6),18,6
>>> t5
(23, (5, 8, 6), 18, 6)
>>> t6=t1,t2,t3,t4,t5
>>> t6
((1, 2, 3), ('east', 'south', 'west', 'north'), ('0010110', '张山', 'men', 18), (), (23, (5, 8, 6), 18, 6))
```

元组的基本操作

元组的类型构造器

- 容器类型对象可以通过它们的类型构造器相互转化。
- 元组的类型构造器tuple()可以生成一个空的元组，也可以将字符串、列表、集合等转化为元组。
- 例如，生成一个空的元组

```
>>> t7=tuple()
>>> t7
()
```

- 例如，将一个字符串转化为元组

```
>>> t7=tuple('Python')
>>> t7
('P', 'y', 't', 'h', 'o', 'n')
```

元组的基本操作

元组元素的访问

- 可以通过下标访问元组中的某一项，称为元组元素。
- 下标从0开始。
- 举例：

```
>>> t6
((1, 2, 3), ('east', 'south', 'west', 'north'), ('0010110','张山', 'men', 18), (23, (5, 8, 6), 18, 6), ())
>>> t6[2]
('0010110', '张山', 'men', 18)
>>> t6[2][0]
'0010110'
```

元组的基本操作

元组元素的访问

- 元组的元数不能被修改。
- 举例：

```
>>> t6[2]=t2
Trace  back (most recent call last):
  File "<pyshell#60>", line 1, in <module>
    t6[2]=t2
TypeError: 'tuple' object does not support item assignment
```

元组的基本操作

内置函数操作元组

- 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')]
```

元组对象的方法

元组对象的方法

方法	描述
T.count(value)	计算value值在元组中出现的次数
T.index(value)	计算value值在元组中第一次出现的位置

列表

列表

列表的基本操作

- （1）列表的字面表示
- （2）列表的类型构造器
- （3）列表元素的访问

列表的基本操作

列表的字面表示

- 列表的创建与元组的区别在于需要使用方括号。
- 其它与元组类似
  - 数据项之间以逗号分隔
  - 可以嵌套定义
  - 可以是不同的数据类型
  - 可以是空列表

列表的基本操作

列表的字面表示

- 举例：字面表示方式创建列表

```
>>> L1=["one","two","three","four","five"]  #由5个字符串构成的列表
>>> L2=[[1,2],[3,4],[5,6]]  #由3个列表构成的嵌套列表
>>> L3=["zhang",True,185,"li",False,165,"wang",True,176]  #混合类型的列表
>>> L4=[[ ],[ ],[ ]]  #嵌套空列表的列表
>>>L5=[ ]  #生成空的列表
```

列表的基本操作

列表的类型构造器

- 列表的类型构造器list()可以生成一个空的列表，也可以将字符串、元组、集合等转化为列表。
- 举例：
  - 生成空的列表，与L5=[ ]功能相同

```
>>> L5=list()
```

  - 将一个字符串对象转化为列表

```
>>> L6=list('Pyhton')
>>> L6
['P', 'y', 'h', 't', 'o', 'n']
```

  - 将一个元组转化为列表

```
>>> L7=list(('he','her','here'))
 >>> L7
['he', 'her', 'here']
```

列表的基本操作

列表元素的访问

- 列表支持索引访问
  - 访问特定列表元素
  - 访问子列表
  - 修改列表元素
- 举例：

```
>>> L1[1]
'two'
>>> L2[2][1]
6
```

列表的基本操作

修改列表元素

- 列表和元组根本区别在于可以改变列表中的元素。
- 举例：

```
>>> L2[2]=5
>>> L2
[[1, 2], [3, 4], 5]
>>> L2[0][0]=L2[0][1]*10
>>> L2
[[20, 2], [3, 4], 5]
```

列表的基本操作

连接列表

- 举例：

```
>>> L2=L2+[7,8]  # L2本身“加长”了，其元素增多了
>>> L2
[[20, 2], [3, 4], 5, 7, 8]
>>> L2=L2+[[9,10]]
>>> L2
[[20, 2], [3, 4], 5, 7, 8, [9, 10]]
```

列表的基本操作

插入列表元素

- 举例：通过切片插入元素

```
>>> L2[3:3]=[6]    #3:3表示下标为3的前面位置
>>> L2
[[20, 2], [3, 4], 5, 6, 7, 8, [9, 10]]
```

列表的基本操作

删除列表元素

- 举例：通过切片删除元素

```
>>> L2[2:6]=[ ]      #2:6表示下标从2到5
>>> L2
[[20, 2], [3, 4], [9, 10]]
```

列表的基本操作

- 在指定位置插入嵌套的列表数据项
  - 举例：

```
>>> L2[2:2]=[[5,6],[7,8]]
>>> L2
[[20, 2], [3, 4], [5, 6], [7, 8], [9, 10]]
```

列表的基本操作

计算列表的长度

- 举例：

```
>>> len(L2)
5
>>> len(L3)
9
```

列表的基本操作

成员测试

- 举例

```
>>> 'wang' in L3
True
>>> t2 in t6
True
>>> 7 in L2
False
>>> [7,8] in L2
True
```

列表操作

内置函数对列表的操作

- max()、min()、sum()、len()函数
- 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。

列表操作

- 标准库中大量函数也可以操作列表
  - functools中的reduce()函数；
  - itertools中的compress()、groupby()、dropwhile()等。

列表对象的方法

方法	描述
L.append(object)	在列表L尾部追加对象。
L.clear()	移除列表L中的所有对象。
L.count(value)	计算value在列表L中出现的次数。
L.copy()	返回L的备份的新对象。
L.extend(Lb)	将Lb的表项扩充到L中。
L.index(value,[start,[stop]])	计算value在列表L指定区间第一次出现的下标值。
L.insert(index, object)	在列表L的下标为index的表项前插入对象。
L.pop([index])	返回并移除下标为index的表项，默认最后一个。
L.remove(value)	移除第一个值为value的表项。
L.reverse()	倒置列表L。
L.sort()	对列表中的数值按从低到高的顺序排序。

列表对象的方法

列表对象的方法应用举例

- 对某居民家庭一年的用电情况进行维护和分析。
  - （1）初始化列表

```
>>> t=[ ]
```

  - （2）增加1月份的用电量

```
>>> t.append(271)
```

  - （3）批量增加2到12月份的用电量

```
>>> t.extend ([151,78,92,83,134,357,421,210,88,92,135])
>>> t
[271, 151, 78, 92, 83, 134, 357, 421, 210, 88, 92, 135]
```

列表对象的方法

列表对象的方法应用举例

- 对某居民家庭一年的用电情况进行维护和分析。
  - （4）修改8月份的用电量421为425

```
>>> t.remove(421)
>>> t.insert(7,425)
>>> t
[271, 151, 78, 92, 83, 134, 357, 425, 210, 88, 92, 135]
```

  - （5）求用电量最高的月份maxm

```
>>> maxm=t.index(max(t))+1
>>> maxm
8
```

列表对象的方法

列表对象的方法应用举例

- 对某居民家庭一年的用电情况进行维护和分析。
  - （6）按用电量从低到高排序

```
>>> s=t.copy()  #原始数据保留在 t 中
>>> s.sort()
>>> s
[78, 83, 88, 92, 92, 134, 135, 151, 210, 271, 357, 425]
```

  - （7）找用电量最高的三个月

```
>>> s.reverse()
>>> m1,m2,m3=t.index(s[0])+1,t.index(s[1])+1,t.index(s[2])+1
>>> print(m1,m2,m3)
8 7 1
```

列表应用举例

问题1

- 输入某班10位学生5次考试的成绩，保存到列表，计算每位学生的平均分，找到最高平均分和最低平均分。

```
MAXNUM = 5
SCORENUM = 3
scores = []

for i in range(MAXNUM):
    print("input one student's 3 scores (one score per line):")
    oneline = []
    for j in range(SCORENUM):
        score = float(input())
        oneline.append(score)
    scores.append(oneline)
```

列表应用举例

问题1：成绩计算

续

```
averages = []

for onestudent in scores:
    average = sum(onestudent)/len(onestudent)
    averages.append(average)

highest = max(averages)
lowest = min(averages)

print("highest average: {0:.1f}".format(highest))
print("lowest average: {0:.1f}".format(lowest))
```

列表应用举例

问题2

- 保存一个静态迷宫，然后显示迷宫。

1	1	1	1	1	1	1	1
2	0	1	0	0	0	0	1
1	0	0	0	1	1	1	1
1	0	1	0	0	1	0	0
1	0	1	0	1	0	0	1
1	0	1	0	1	0	1	1
1	0	0	0	0	0	0	1
1	1	1	1	1	1	1	1

迷宫的数据

```
########
@ #      #
#    ####
# #   #
# # #   #
# # # ##
#       #
########
```

迷宫

列表应用举例

问题2：显示迷宫

代码

```
MAXROWS = 8
MAXCOLS = 8

maze = [
     [1,1,1,1,1,1,1,1],
     [2,0,1,0,0,0,0,1],
     [1,0,0,0,1,1,1,1],
     [1,0,1,0,0,1,0,0],
     [1,0,1,0,1,0,0,1],
     [1,0,1,0,1,0,1,1],
     [1,0,0,0,0,0,0,1],
     [1,1,1,1,1,1,1,1]
     ]
```

```
for row in maze:
     for cell in row:
          if cell == 0:
               print(' ', end='')
          elif cell == 1:
               print('#', end='')
          else:
               print('@', end='')
     print()
```

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。
- 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]
```

列表推导式

- 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)
```

列表推导式应用举例

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？
- 解：

```
>>> sum( [2**i for i in range(64)] )
18446744073709551615
```

列表推导式应用举例

应用1：实现嵌套列表的平铺

举例

```
>>> 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)
```

列表推导式应用举例

应用2：过滤不符合条件的元素

- 举例：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]
```

列表推导式应用举例

应用3：同时遍历多个列表或可迭代对象

举例

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

列表推导式应用举例

应用4：使用列表推导式实现矩阵转置

举例

```
>>> 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])
```

列表推导式应用举例

- 应用5：列表推导式中可以使用函数或复杂表达式
  - 举例

```
>>> 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]
```

小结

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。

小结

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。

小结

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。