第3章 列表和元组

列表和元组是Python中最常用的两种序列结构，除此之外，Python中常用的序列结构还有字典、字符串、集合等。列表和元组的主要区别在于，列表可以修改，元组则不能。如果要根据要求来添加元素，那么更适合使用列表；而出于某些原因，序列不能修改的时候，使用元组则更为合适。一般来说，在几乎所有的情况下列表都可以代替元组。

3.1 序列简介

序列是程序设计中经常用到的数据存储方式，几乎每一种程序设计语言都提供了类似的数据结构，简单地说，序列是一块用来存放多个值的连续内存空间。一般而言，在实际开发中同一个序列中的元素通常是相关的。Python提供的序列类型可以说是所有程序设计语言类似数据结构中最灵活的，也是功能最强大的。

除了字典和集合属于无序序列之外，列表、元组和字符串等序列类型均支持双向索引，如果使用正向索引，第一个元素下标为0，第二个元素下标为1，以此类推；如果使用负向索引，则最后一个元素下标为－1，倒数第二个元素下标为－2，以此类推，如图3-1所示。可以使用负整数作为序列索引是Python语言的一大特色，熟练掌握和运用可以大幅度提高开发效率。

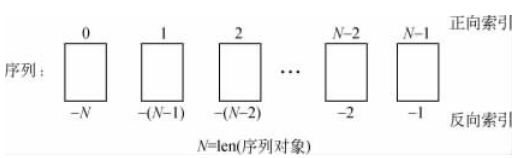

图3-1 序列的索引下标示意图

所有序列类型都可以进行以下操作：索引(indexing)、切片(slicing)、加(adding)、乘(multiplying)以及检查某个元素是否属于序列的成员(成员资格)。除此之外，Python还有计算序列长度、找出最大元素和最小元素的内置函数。

3.1.1 索引

序列对象定义了一个特殊方法__getitem__()，可通过整数下标访问序列的元素。

```
s[i]        #访问序列s在索引i处的元素
```

序列中的所有元素都是有编号的，这些元素可以分别通过编号访问，如下例所示：

```
>>> greeting = 'Hello'      # 字符串是一个由字符组成的序列
>>> greeting[0]             # 索引 0 指向第 1 个元素
'H'
```

所有序列都可以通过这种方式进行索引、获取元素，使用负向索引时，Python 会从右边，也就是从最后一个元素开始计数，而最后一个元素的位置编号是－1，例如：

```
>>> greeting[ - 1]
'o'
```

【例 3-1】 序列的索引访问示例。

```
>>> s = 'abcdef'   # 字符串序列
>>> s[0]
'a'
>>> s[2]
'c'
>>> s[ - 1]
'f'
>>> s[ - 3]
'd'
>>> t = ('a', 'e', 'i', 'o', 'u')   # 元组序列
>>> t[0]
'a'
>>> t[1]
'e'
>>> t[ - 1]
'u'
>>> t[ - 5]
'a'
>>> lst = [1, 2, 3, 4, 5]   # 列表序列
>>> lst[0]
1
>>> lst
[1, 2, 3, 4, 5]
>>> lst[2] = 'a'
>>> lst[ - 2] = 'b'
>>> lst
[1, 2, 'a', 'b', 5]
```

如果索引越界，则导致 IndexError；如果 i 不是整数，则导致 TypeError。例如：

```
>>> s = 'abc'
>>> s['a']                  # TypeError: string indices must be integers
>>> s[3]                    # IndexError: string index out of range
```

3.1.2 切片

切片使用 2 个冒号分隔的 3 个数字来完成：第一个数字表示切片开始位置(默认为 0)，第二个数字表示切片截止(但不包含)位置(默认为列表长度)，第三个数字表示切片的步长(默认为 1)，当步长省略时可以顺便省略最后一个冒号。可以使用切片来截取列表中的任何部分，得到一个新列表，也可以通过切片来修改和删除列表中的部分元素，甚至可以通过切片操作为列表对象增加元素。切片的基本形式为：

s[i:j]或者 **s[i:j:k]**

其中，i 为开始下标(包含 s[i])，j 为结束下标(不包含 s[j])，k 为步长。如果省略 i，则访问范围从下标 0 开始；如果省略 j，则访问范围直到结束为止；如果省略 k，则步长为 1。

切片操作对于提取序列的一部分是很有用的，而编号在这里显得尤为重要。第 1 个索引是需要提取部分的第 1 个元素的编号，而最后的索引则是切片之后剩下部分的第 1 个元素的编号，例如：

```
>>> numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> numbers[3:6]
```

```
[4, 5, 6]
>>> numbers[0:1]
[1]
```


简而言之，切片操作的实现需要提供两个索引作为边界，第 1 个索引的元素是包含在切片内的，但是第 2 个索引的元素不包含在切片中，如图 3-2 所示。

图 3-2　切片操作示例

1. 切片范围的控制

假设需要访问最后 3 个元素，那么可以进行如下操作：

```
>>> numbers[7:10]
[8, 9, 10]
```

现在，索引 10 指向的是第 11 个元素，而这个元素并不存在，同时也不包含在切片中。如果需要从列表的结尾开始计数，例如：

```
>>> numbers[-3:-1]
[8, 9]
```

最后一个元素因为不包含在切片中而没有被访问。

```
>>> numbers[-3:-5]
[ ]
```

如果切片中最左边的索引比它右边的晚出现在序列中，结果就是一个空的序列。如果切片所得部分包括序列结尾的元素，那么只需空置最后一个索引即可：

```
>>> numbers[-3:]
[8, 9, 10]
```

这种方法同样适用于包括序列开始的元素：

```
>>> numbers[:3]
[1, 2, 3]
```

如果需要复制整个序列，可以将两个索引都空置：

```
>>> numbers[:]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

2. 变化的步长

进行切片的时候，切片的开始和结束点需要进行指定(不管是直接还是间接)。而另外一个参数——步长，通常都是隐式设置的。在普通的切片中，步长是 1，切片操作按照这个步长逐个遍历序列的元素，然后返回开始和结束点之间的所有元素：

```
>>> numbers[0:10:1]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

在这个例子中，切片包含了另外一个数字，就是步长的显示设置。如果步长被设置为比1大的数，那么就会跳过某些元素。例如，步长为2的切片包括的是从开始到结束每隔1个的元素。

```
>>> numbers[0:10:2]
[1, 3, 5, 7, 9]
>>> numbers[3:6:3]
[4]
```

如果要将每4个元素中的第1个提取出来，那么只要将步长设置为4即可：

```
>>> numbers[::4]
[1, 5, 9]
```

当然，步长不能为0，但是步长可以是负数，即从右到左提取元素：

```
>>> numbers[8:3:-1]
[9, 8, 7, 6, 5]
>>> numbers[10:0:-2]
[10, 8, 6, 4, 2]
>>> numbers[::-2]
[10, 8, 6, 4, 2]
>>> numbers[7::-2]
[8, 6, 4, 2]
>>> numbers[:7:-2]
[10]
```

【注意】 开始点的元素（最左边元素）包含在结果之中，而结束点的元素（最后边的元素）则不在切片之内。当使用一个负数作为步长时，必须让开始点（开始索引）大于结束点。在没有明确指定开始点和结束点的时候，正负数的使用可能会带来一些混淆。不过在这种情况下，Python会进行正确的操作：对于一个正数步长，Python会从序列的头部开始向右提取元素，直到最后一个元素，而对于负数步长，则是从序列的尾部开始向左提取元素，直到第1个元素。

【例3-2】 序列的切片操作示例。

```
>>> s = 'abcdef'
>>> s[1:3]
'bc'
>>> s[3:10]
'def'
>>> s[8:2]
''
>>> s[:]
'abcdef'
>>> s[::2]
'ace'
>>> s[::-1]
'fedcba'
>>> t = ('a', 'e', 'i', 'o', 'u')
>>> t = [-2:-1]
('o',)
>>> t[-2:]
('o', 'u')
>>> t[-99:-3]
('a', 'e')
>>> t[::]
('a', 'e', 'i', 'o', 'u')
>>> t[1:-1]
('e', 'i', 'o')
>>> t[1::2]
('e', 'o')
>>> lst = [1, 2, 3, 4, 5]
>>> lst[:2]
[1, 2]
>>> 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'
```

3.2 序列相加

通过连接操作符＋,可以连接两个序列(s1 和 s2),形成一个新的序列对象:

```
s1 + s2
```

连接操作符也支持复合赋值运算,即＋＝。

【例 3-3】 序列的连接操作示例。

```
>>> lst1 = [1, 2]
>>> lst2 = ['a', 'b']
>>> lst1 + lst2
[1, 2, 'a', 'b']
>>> lst1 +  = lst2
>>> lst1
[1, 2, 'a', 'b']
>>> s1 = 'abc'
>>> s2 = 'xyz'
>>> s1 + s2
'abcxyz'
```

```
>>> s1 += s2
>>> s1
'abcxyz'
>>> t1 = (1, 2)
>>> t2 = ('a', 'b')
>>> t1 + t2
(1, 2, 'a', 'b')
>>> t1 += t2
>>> t1
(1, 2, 'a', 'b')
```

【注意】 两个相同类型的序列才能进行连接操作。

```
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
>>> 'Hello ' + 'world!'
'Hello world!'
>>> [1, 2, 3] + 'world!'
Traceback (innermost last):
  File "<pyshell#2>". line 1. In ?
[1, 2, 3] + 'world!'
TypeError: can only concatenate list (not "string") to list
```

正如错误信息所提示的,列表和字符串是无法连接在一起的,尽管它们都是序列。

3.2.1 序列重复

通过重复操作符＊,可以重复一个序列 *n* 次,基本形式为:

s ＊ n 或者 **n ＊ s**

重复操作符也支持复合赋值运算,即＊＝。

【例 3-4】 重复操作示例。

```
>>> lst1 = [1, 2]
>>> lst2 = ['a', 'b']
>>> 2 * lst1
[1, 2, 1, 2]
>>> lst2 *  = 2
>>> lst2
['a', 'b', 'a', 'b']
>>> s1 = 'abc'
>>> s2 = 'xyz'
>>> s1 * 3
'abcabcabc'
>>> s2 * = 2
'xyzxyz'
>>> t1 = (1, 2)
>>> t2 = ('a', 'b')
>>> t1 * 2
(1, 2, 1, 2)
>>> t2 * = 2
>>> t2
('a', 'b', 'a', 'b')
>>> 'python' * 3
'pythonpythonpython'
>>> [1] * 10
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

空列表可以简单地通过([ ])进行标识,如果想创建一个占用 10 个元素空间的列表,可以像前面这样使用[1] * 10;如果需要一个没有放置任何元素的列表,此时需要使用 None。None 是一个 Python 的内置值,它的确切含义是“这里什么都没有”。因此,如果想初始化一个长度为 10 的列表,可以按照下面的例子来实现:

```
>>> sequence = [None] * 10
>>> sequence
[None, None, None, None, None, None, None, None, None, None]
```

3.2.2 成员资格

检查一个元素是否存在于序列中即为成员资格判断,可以通过下列方式之一进行判断:

```
x in s
x not in s
s.count(x)
s.index(x[ , i[ , j]])
```

其中,指定范围[i, j),从下标 i(包括,默认为 0)开始,到下标 j 结束(不包括,默认为 len(s))。

对于 s. index(value, [start, [stop]])方法,如果找不到,则导致 ValueError。例如:

```
>>> 'To be or not to be, this is a question'. index('123')     # ValueError: substring not found
```

【例 3-5】 序列中元素的成员资格判断示例。

```
>>> lst = [1, 2, 3, 2]
>>> 1 in lst
True
>>> 2 not in lst
False
>>> lst.count(1)
1
>>> lst.index(2)
1
>>> lst.index(3)
2
>>> s = 'good'
>>> 'o' in s
True
>>> 'g' not in s
False
>>> s.index('o', 2)
2
>>> t = ('r', 'g', 'b')
>>> 'r' in t
True
>>> 'y' not in t
True
>>> t.count('r')
1
>>> t.index('g')
1
```

【例 3-6】 使用 in 运算符进行成员资格判断示例。

```
>>> greeting = 'Hello world!'
>>> 'w' in greeting
True
>>> 'a' in greeting
False
>>> users = ['Lucy', 'Sam', 'John']
>>> raw_input('Enter your user name: ') in users
Enter your user name: Lucy
True
>>> sentence = '$ $ $ Get up now!!! $ $ $'
'$ $ $' in sentence
True
```

本例中首先使用了成员资格测试分别来检查'w'和'a'是否出现在字符串 greeting 中,接下来则是检查所提供的用户名 Lucy 是否在用户列表中,最后检查字符串 sentence 是否包含字符串'$ $ $',这段程序可以作为垃圾邮件过滤器的一部分,执行某些安全策略。

3.2.3 序列比较

序列支持比较运算符(<、<=、==、!=、>=、>),比较运算按照顺序逐个元素进行比较,运算结果是 True 或 False,表 3-1 为序列比较运算符举例。

表 3-1 序列比较运算符举例

运 算	意义描述	运 算	意义描述
a < b	小于	a >= b	大于或等于
a <= b	小于或等于	a == b	等于
a > b	大于	a != b	不等于

【例 3-7】 序列的比较运算示例。

```
>>> 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
>>> 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
```

3.2.4 序列排序

通过内置函数 sorted(),可以返回序列的排序列表。

```
sorted(iterable, key = None, reverse = False)   #返回序列的排序列表
```

其中,key 是用于计算比较键值的函数(带 1 个参数),例如,key=str.lower; reverse 是排序规则,reverse=False 为升序(默认),reverse=True 则是降序。

【例 3-8】 序列的排序操作示例。

```
>>> s1 = 'acb'
>>> sorted(s1)
['a', 'b', 'c']
>>> s2 = (1, 5, 3)
>>> sorted(s2)
[1, 3, 5]
>>> sorted(s2, reverse = True)
[5, 3, 1]
>>> s3 = 'abAC'
>>> sorted(s3, key = str.lower)
['a', 'A', 'b', 'C']
```

3.2.5 长度、最小值和最大值

内置函数 len()、min()和 max()分别返回序列中所包含元素的数量、序列中最大和最小的元素,内置函数 sum()可获取列表或元组各元素之和;如果有非数值元素,则导致 TypeError;对于字符串(str)和字节数据(bytes),也将导致 TypeError。例如:

```
>>> s1 = (1, 2, 3, 4)
>>> sum(s)        #输出 10
>>> s2 = (1, 'a', 2)
>>> sum(s2)       #TypeError: unsupported operand type(s) for + : 'int' and 'str'
>>> s3 = '1234'
>>> sum(s3)       #TypeError: unsupported operand type(s) for + : 'int' and 'str'
```

【例 3-9】 序列的长度、最大值、最小值操作示例。

```
>>> s1 = [1, 2, 3]
>>> len(s1)
3
>>> max(s1)
3
>>> min(s1)
1
>>> s2 = ''
>>> len(s2)
0
>>> t1 = (10, 2, 3)
>>> len(t1)
3
>>> max(t1)
10
>>> min(t1)
2
>>> t2 = ()
>>> len(t2)
0
>>> lst1 = [1, 2, 9, 5, 4]
>>> len(lst1)
5
>>> max(lst1)
```

```
9
>>> min(lst1)
1
>>> lst2 = [ ]
>>> len(lst2)
0
>>> max(3, 30)
30
>>> min(5, 3, 7, 9)
3
```

3.3 列　　表

列表是 Python 的内置可变列表，是包含若干元素的有序连续内存空间。在形式上，列表的所有元素放在一对方括号“[”和“]”中，相邻元素之间使用逗号分隔开。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，从而保证元素之间没有缝隙。Python 列表内存的自动管理可以大幅度减少程序员的负担，但列表的这个特点会涉及列表中大量元素的移动，效率较低，并且对于某些操作可能会导致意外的错误结果。因此，尽量从列表尾部进行元素的增加或删除操作，这会大幅度提高列表处理速度。

列表(List)是一组有序存储的数据，例如，饭店点餐的菜单就是一种列表。列表具有如下特性：

(1) 与变量一样，每个列表都有一个唯一标识它的名称。

(2) 每个列表元素都有索引和值两个属性，索引是一个从 0 开始的整数，用于标识元素在列表中的位置，值就是对应位置的元素的值。

同一个列表中元素的类型可以不相同，可以同时包含整数、实数、字符串等基本类型，也可以是列表、元组、字典以及其他自定义类型的对象。例如

```
[1, 2, 3, 4, 5]
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
['spam', 1.0, 6, [10, 20]]
[['Tom', 10, 3], ['Mary', 8, 1]]
```

都是合法的列表对象。

对于 Python 序列而言，有很多方法是通用的，而不同类型的序列又有一些特有的方法。列表对象常用方法如表 3-2 所示，假设表中的示例基于 s＝[1，3，2]。除此之外，Python 的很多内置函数和命令也可以对列表和其他序列对象进行操作，后面将逐步进行介绍。

表 3-2　列表对象常用方法

方　　法	说　　明	示　　例
s.append(x)	将元素 x 添加至列表尾部	s.append('a') ＃s＝[1，3，2，'a'] s.append([1,2]) ＃s＝[1，3，2，[1,2]]
s.extend(t)	将列表 t 附加至列表 s 尾部	s.extend([4]) ＃s＝[1，3，2，4] s.extend(['a','b']) ＃s＝[1，3，2,'a'，'b']
s.insert(i，x)	在列表指定位置 i 处添加元素 x	s.insert(1,4) ＃s＝[1，4，3，2] s.insert(8,5) ＃s＝[1，4，3，2，5]

续表

方　法	说　明	示　例
s.remove(x)	在列表中删除首次出现的指定元素，若对象不存在，将导致 ValueError	s.remove(1) ＃s=[3，2] s.remove(0) ＃ ValueError：list.remove(x)：x not in list
s.pop([i])	删除并返回列表对象指定位置的元素，默认为最后一个元素	s.pop() ＃输出 2。s=[1，3] s.pop(0) ＃输出 1。s=[3，2]
s.index(x)	返回第一个值为 x 的元素的下标，若不存在值为 x 的元素，则抛出异常	s.index(1) ＃输出 0 s.index(5) ＃ ValueError：5 is not in list
s.count(x)	返回指定元素 x 在列表中的出现次数	s.count(1) ＃输出 1 s.count(0) ＃输出 0
s.reverse()	对列表元素进行原地翻转	s.reverse() ＃s=[2，3，1]
s.sort()	对列表元素进行原地排序	s.sort() ＃s=[1，2，3]

3.3.1　列表的创建与删除

1. 创建列表

列表采用方括号中用逗号分隔的项目定义。其基本形式如下：

```
[x1, [x2, …, xn] ]
```

如同其他类型的 Python 对象变量一样，使用赋值运算符“＝”直接将一个列表赋值给变量即可创建列表对象，例如：

```
>>> a_list = ['a', 'b', 'c', 'd']
>>> a_list = [ ]  ＃创建空列表
```

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

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

【例 3-10】 创建列表对象。

```
>>> [ ]
[ ]
>>> [1, 2, 3]
[1, 2, 3]
>>> list()
[ ]
>>> list((1, 2, 3))
[1, 2, 3]
>>> list(range(3))
[0, 1, 2]
>>> list('abc')
['a', 'b', 'c']
>>> list([1, 2, 3])
[1, 2, 3]
>>> a = ['x', 2]
>>> a
['x', 2]
```

上面的代码中用到了内置函数 range()，这是一个非常有用的函数，后面会多次用到，该函数语法为：

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

内置函数 range()接收三个参数，第一个参数表示起始值(默认为 0)，第二个参数表示终止值(结果中不包括这个值)，第三个参数表示步长(默认为 1)，该函数在 Python 2.x 中返回一个包含若干整数的列表。另外，Python 2.x 还提供了一个内置函数 xrange()，语法与 range()函数一样，但是返回 xrange 可迭代对象，而不是像 range()函数一样返回列表。例如：

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> xrange(5)
xrange(5)
>>> list(xrange(5))
[0, 1, 2, 3, 4]
```

使用 Python 2.x 处理大数据或较大循环范围时，建议使用 xrange()函数来控制循环次数或处理范围，以获得更高的效率。

2. 删除列表

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

```
>>> del a_list
>>> a_list
NameError: name 'a_list' is not defined
```

正如上面的代码所展示的一样，删除列表对象 a_list 之后，该对象就不存在了，再次访问时将抛出异常 NameError 提示所访问的对象名不存在。

【例 3-11】 列表的创建与删除操作示例。

```
>>> s = [1, 2, 3, 4, 5, 6]
>>> s[1] = 'a'
>>> s
[1, 'a', 3, 4, 5, 6]
>>> s[2] = [ ]
>>> s
[1, 'a', [ ], 4, 5, 6]
>>> del s[3]
>>> s
[1, 'a', [ ], 5, 6]
>>> s[:2]
[1, 'a']
```

```
>>> s[2:3] = [ ]
>>> s
[1, 'a', 5, 6]
>>> s[:1] = [ ]
>>> s
['a', 5, 6]
>>> s[:2] = 'b'
>>> s
['b', 6]
>>> del s[:1]
>>> s
[6]
```

3.3.2 列表元素的增加

列表元素的动态增加和删除是实际应用中经常遇到的操作，Python 列表提供了多种不同的方法来实现这一功能。

1. 运算符(+)

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

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

2. append()方法

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

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

3. extend()方法

使用列表对象的 extend()方法可以将另一个迭代对象的所有元素添加至该列表对象尾部。

```
>>> s.extend([11, 13])
>>> s
[3, 4, 5, 7, 9, 11, 13]
```

4. insert()方法

使用列表对象的 insert()方法将元素添加至列表的指定位置。

```
>>> s = [3, 4, 5]
>>> s.insert(2, 6)
>>> s
[3, 4, 6, 5]
```

列表的 insert()方法可以在列表的任意位置插入元素,但由于列表的自动内存管理功能,insert()方法会涉及插入位置之后所有元素的移动,这会影响处理速度,类似的还有后面介绍的 remove()方法以及使用 pop()函数弹出列表非尾部元素和使用 del 命令删除列表非尾部元素的情况。因此,除非必要,应尽量避免在列表中间位置插入和删除元素的操作,而是优先考虑使用前面介绍的 append()方法。

5. 乘法运算符(*)

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

```
>>> s = [3, 5, 7]
>>> t = s * 3
>>> t
[3, 5, 7, 3, 5, 7, 3, 5, 7]
```

该操作实际上是创建了一个新的列表，而不是真的扩展了原列表，该操作同样适用于字符串和元组，并具有相同的特点。

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

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

【例 3-12】 列表元素的增加操作示例。

```
>>> s = [1, 2, 3, 4, 5]
>>> t = ['a', 'b', 'c']
>>> s + t
[1, 2, 3, 4, 5, 'a', 'b', 'c']
>>> s.append(6)
>>> s
[1, 2, 3, 4, 5, 6]
>>> s.extend([8, 9])
>>> s
[1, 2, 3, 4, 5, 6, 8, 9]
>>> s.insert(6, 7)
>>> s
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> t * 2
['a', 'b', 'c', 'a', 'b', 'c']
```

3.3.3 列表元素的删除

1. del 命令

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

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

2. pop()方法

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

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

3. remove()方法

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

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

【例 3-13】 列表元素的删除操作示例。

```
>>> lst = [1,2,4,5,6]
>>> del lst[2]
>>> lst
[1, 2, 5, 6]
>>> lst.pop()
6
>>> lst
[1, 2, 5]
>>> lst.remove(5)
>>> lst
[1, 2]
```

3.3.4 列表元素的访问与计数

可以使用下标直接访问列表中的元素。如果指定下标不存在，则抛出异常提示下标越界，例如：

```
>>> s = [3, 4, 5, 6, 7, 9, 11, 13, 15, 17]
>>> s[3]
6
>>> s[3] = 5.5
>>> s
[3, 4, 5, 5.5, 7, 9, 11, 13, 15, 17]
>>> s[15]
IndexError: list index out of range
```

使用列表对象的 index()方法可以获取指定元素首次出现的下标，语法为 index(value, [start, [stop]])，其中 start 和 stop 用来指定搜索范围，start 默认为 0，stop 默认为列表长度。若列表对象中不存在指定元素，则抛出异常提示列表中不存在该值，例如：

```
>>> s
[3, 4, 5, 5.5, 7, 9, 11, 13, 15, 17]
>>> s.index(7)
4
>>> s.index(100)
ValueError: 100 is not in list
```

如果需要知道指定元素在列表中出现的次数，可以使用列表对象的 count()方法进行统计，例如：

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

该方法也可以用于元组、字符串以及 range 对象，例如：

```
>>> range(10).count(3)
1
>>> (3, 3, 4, 4).count(3)
2
>>> 'abcdefgabc'.count('abc')
2
```

3.3.5 成员资格判断

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

【例 3-14】 成员资格判断操作示例。

```
>>> s = [1, 2, 3]
>>> s
[1, 2, 3]
>>> 3 in s
True
>>> 18 in s
False
>>> t = [[1], [2], [3]]
>>> 3 in t
False
>>> 3 not in t
True
>>> [3] in t
True
>>> [5] in t
False
>>> s1 = [3, 5, 7, 9, 11]
>>> s2 = ['a', 'b', 'c', 'd']
>>> (3, 'a') in zip(s1, s2)
True
>>> for a, b in zip(s1, s2):
print (a, b)
(3,'a')
(5,'b')
(7,'c')
(9,'d')
```

关键字 in 和 not in 也可以用于其他可迭代对象，包括元组、字典、range 对象、字符串、集合等，常用在循环语句中对序列或其他可迭代对象中的元素进行遍历。使用这种方法来遍历序列或迭代对象，可以减少代码的输入量、简化程序员的工作，并且大幅度提高程序的可读性，建议熟练掌握和运用。

3.3.6 切片操作

切片是 Python 序列的重要操作之一，适用于列表、元组、字符串、range 对象等类型。与使用下标访问列表元素的方法不同，切片操作不会因为下标越界而抛出异常，而是简单地在列表尾部截断或者返回一个空列表，代码具有更强的健壮性。

【例 3-15】 列表的切片操作示例。

```
>>> s = [3, 5, 7, 9, 11]
>>> s[::]
[3, 5, 7, 9, 11]
>>> s[::-1]
[11, 9, 7, 5, 3]
>>> s[::2]
[3, 7]
>>> s[1::2]
[5, 9]
>>> s[3::]
[9, 11]
>>> s[3:6]
[9, 11]
>>> s[3:6:1]
```

```
[9, 11]
>>> s[0:100:1]
[3, 5, 7, 9, 11]
```

```
>>> s[100:]
[ ]
```

可以使用切片操作来快速实现很多目的，例如原地修改列表内容，列表元素的增、删、改、查以及元素替换等操作都可以通过切片来实现，并且不影响列表对象内存地址。

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

也可以结合使用 del 命令与切片操作来删除列表中的部分元素：

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

切片返回的是列表元素的浅复制，与列表对象的直接复制并不一样。

【例 3-16】 列表元素的浅复制与直接复制操作示例。

```
>>> s1 = [3, 5, 7]
>>> s2 = s1 #s1 和 s2 指向同一块内存
>>> s2
[3, 5, 7]
>>> s2[1] = 8
>>> s1
[3, 8, 7]
>>> s1 == s2
True
>>> s1 is s2
True
```

```
>>> s1 = [3, 5, 7]
>>> s2 = s1[::] #浅复制
>>> s1 == s2
True
>>> s1 is s2
False
>>> s2[1] = 8
>>> s2
[3, 8, 7]
>>> s1
[3, 5, 7]
>>> s1 == s2
False
>>> s1 is s2
False
```

3.3.7 列表排序

在实际应用中，经常需要对列表元素进行排序。

1. sort()方法

sort()方法用于在原位置对列表进行排序。在“原位置排序”意味着改变原来的列表，从而让其中的元素能按一定的顺序排列，而不是简单地返回一个已排序的列表副本，该方法支持多种不同的排序方式。

```
>>> s = [2, 4, 6, 1, 3, 5]
>>> s.sort()
>>> s
[1, 2, 3, 4, 5, 6]
```

当用户需要一个排好序的列表副本，同时又保留原有列表不变的时候，正确的方法是：首先把 s 的副本复制给 t，然后对 t 进行排序，例如：

```
>>> s = [2, 4, 6, 1, 3, 5]
>>> t = s[:]
>>> t.sort()
>>> s
[1, 2, 3, 4, 5, 6]
>>> t
[2, 4, 6, 1, 3, 5]
```

在此调用 s[：]得到的是包含了 s 所有元素的切片，这是一种很高效的复制整个列表的方法。如果只是简单地把 s 赋值给 t 是没用的，因为这样做就使得 s 和 t 都指向同一个列表了。

```
>>> t = s
>>> t.sort()
>>> s
[1, 2, 3, 4, 5, 6]
>>> t
[1, 2, 3, 4, 5, 6]
```

2. sorted()

内置函数 sorted()也可以对列表进行排序，与列表对象的 sort()方法不同，内置函数 sorted()返回新列表，并不对原列表进行任何修改。

```
>>> s = [2, 4, 6, 1, 3, 5]
>>> t = sorted(s)
>>> s
[2, 4, 6, 1, 3, 5]
>>> t
[1, 2, 3, 4, 5, 6]
```

这个函数实际上可以用于任何序列，却总是返回一个列表：

```
>>> sorted('Python')
['P', 'h', ',n', 'o', 't', 'y']
```

3. reversed()

在某些应用中可能需要将列表元素进行逆序排列，也就是所有元素位置翻转，第一个元素与最后一个元素交换位置，第二个元素与倒数第二个元素交换位置，以此类推。Python提供了内置函数reverse()支持对列表元素进行逆序排列，返回一个逆序排列后的迭代对象，例如：

```
>>> s = [3, 4, 5, 2, 1]
>>> t = reversed(s)
>>> t
<listreverseiterator at 0xa46af28>
>>> list(t)
[1, 2, 5, 4, 3]
```

【例 3-17】 列表元素的排序操作示例。

```
>>> s1 = [2, 4, 6, 1, 3, 5]
>>> s1.sort()
>>> s1
[1, 2, 3, 4, 5, 6]
s1 = [2, 4, 6, 1, 3, 5]
>>> s1.reverse()
>>> s1
[5, 3, 1, 6, 4, 2]
```

```
>>> s1.sort(reverse = True)
>>> s2 = [2, 4, 6, 1, 3, 5]
>>> sorted(s2)
[1, 2, 3, 4, 5, 6]
>>> s2 = [2, 4, 6, 1, 3, 5]
>>> s = reversed(s2)
>>> list(s)
[6, 5, 4, 3, 2, 1]
```

3.3.8 列表推导式

使用列表推导式，可以简单高效地处理一个可迭代对象，并生成结果列表。列表推导式的形式如下：

```
[expr for i1 in 序列 1 … for iN in 序列 N]                    #迭代序列里所有内容，并计算生成列表
[expr for i1 in 序列 1 … for iN in 序列 N if cond_expr] #按条件迭代，并计算生成列表
```

表达式expr使用每次迭代内容i1 … iN，计算生成一个列表。如果指定了条件表达式cond_expr，则只有满足条件的元素参与迭代。例如：

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

相当于

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

接下来再通过几个示例来进一步介绍列表推导式的功能。

(1) 使用列表推导式实现嵌套列表的平铺。

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

(2) 过滤不符合条件的元素。

在列表推导式中可以使用 if 子句来筛选,只在结果列表中保留符合条件的元素。例如,下面的代码用于从当前列表中选择符合条件的元素组成新的列表:

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

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

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

(4) 使用列表推导式实现矩阵变换。

```
>>> 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]]
```

【例 3-18】 列表推导式操作示例。

```
>>> [i ** 2 for i in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 91]
>>> [i for i in range(10) if i%2 == 0]
[0, 2, 4, 6, 8]
>>> [(x, y, x * y) for x in range(1, 4) for y in range(1, 4) if x>= y]
[(1, 1, 1), (2, 1, 2), (2, 2, 4), (3, 1, 3), (3, 2, 6), (3, 3, 9)]
```

3.4 元　　组

元组与列表类似,也是一种序列,但与列表不同的是,元组属于不可变序列,不能修改。元组一旦创建不可以修改其元素的值,也无法为元组增加或删除元素,如果确实需要修改,只能再创建一个新的元组。

3.4.1 元组的创建与删除

元组的定义形式和列表很相似,区别在于定义元组时所有元素放在一对圆括号“(”和“)”中,而不是方括号。圆括号可以省略:如果用逗号分隔了一些值,那么就自动创建了元组。

```
(x1, [x2, …, xn])
```

或者

```
x1, [x2, …, xn]
```

其中,x1, x2, …, xn 为任意对象。注意:如果元组中只有一个项目,后面的逗号不能省

略，这是因为 Python 解释器把(x1)解释为 x1，例如(1)解释为整数 1，(1,)则解释为元组。

元组也可以通过创建 tuple 对象来创建。其基本形式为：

```
tuple()              #创建一个空元组
tuple(iterable)      #创建一个元组，包含的项目可为枚举对象 iterable 中的元素
```

【例 3-19】 创建元组对象示例。

```
>>> 1, 2, 3
(1, 2, 3)
>>> (1, 2, 3)
(1, 2, 3)
>>> ()  #空元组
()
>>> 1,
(1, )
>>> (1)
1
>>> 'a', 'b', 'c'
('a', 'b', 'c')
>>> 'a',
('a', )
>>> tuple()
()
>>> tuple(range(3))
(0, 1, 2)
>>> tuple('abc')
('a', 'b', 'c')
>>> tuple([1, 2, 3])
(1, 2, 3)
```

tuple 函数的功能与 list 函数基本上是一样的：以一个序列作为参数并把它转换为元组。如果参数就是元组，那么该参数就会被原样返回。使用＝操作符将一个元组复制给变量，就可以创建一个元组变量。

```
>>> a_tuple = ('a', )
>>> b_tuple = ('a', 'b', 'c')
>>> c_tuple = ()
```

如同使用 list()函数将序列转换为列表一样，也可以使用 tuple()函数将其他类型序列转换为元组。

```
>>> print (tuple('abcdefg'))
('a', 'b', 'c', 'd', 'e', 'f', 'g')
>>> s = [1, 2, 3, 4]
>>> tuple(s)
(1, 2, 3, 4)
```

对于元组而言，只能使用 del 命令删除整个元组对象，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。

3.4.2 元组的基本操作

元组其实并不复杂，除了创建元组和访问元组元素之外，支持索引访问、切片操作、连接操作、重复操作、成员资格操作、比较运算符操作，以及求元组长度、最大值、最小值等。

【例 3-20】 元组的基本操作示例。

```
>>> s1 = (1, 2, 1)
>>> s2 = ('a', 'x', 'y', 'z')
>>> len(s1)
3
>>> len(s2)
4
>>> max(s1)
2
```

```
>>> min(s2)
'a'
>>> sum(s1)
4
>>> sum(s2)
Traceback (most recent call last):
  File " < ipython - input - 77 - bb5a6cd66c38 >", line 1, in < module >
    sum(s2)
TypeError: unsupported operand type(s) for + : 'int' and 'str'
>>> s1[0:2]
(1, 2)
```

元组的切片还是元组，就像列表的切片还是列表一样。

3.4.3 元组与列表的区别

列表属于可变序列，可以随意地修改列表中的元素值以及增加和删除列表元素，而元组属于不可变序列，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，即不能从元组中删除元素，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，而不支持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。

元组的访问和处理速度比列表更快，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非列表。可以认为元组对不需要修改的数据进行了“写保护”，从内在实现上不允许修改其元素值，而从使得代码更加安全。

另外，作为不可变序列，与整数、字符串一样，元组可用作字典的键，而列表则不能当作字典键使用，因为列表不是不可变的。

最后，虽然元组属于不可变列表，其元素的值是不可改变的，但是如果元组中包含序列，情况就略有不同，例如：

```
>>> s = ([1, 2], 3)
>>> s[0][0] = 5
>>> s
([5, 2], 3)
>>> s[0].append(8)
>>> s
([5, 2, 8], 3)
>>> s[0] = s[0] + [10]
Traceback (most recent call last):
  File "< ipython - input - 81 - 746f13e9d0fd >", line 1, in < module >
    s[0] = s[0] + [10]
TypeError: 'tuple' object does not support item assignment
>>> s
([5, 2, 8], 3)
```

3.4.4 生成器推导式

从形式上看，生成器推导式与列表推导式非常接近，只是生成器推导式使用圆括号而不是列表推导式所使用的方括号。与列表推导式不同的是，生成器推导式的结果是一个生成

器对象而不是列表，也不是元组。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()方法进行遍历，或者直接将其作为迭代器对象来使用。但是不管用哪种方法访问其元素，当所有元素访问结束后，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。

【例 3-21】 生成器推导式操作示例。

```
>>> s = [(i+2) ** 2 for i in range(10)]
s
[4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
>>> for i in g:
print i :
<generator object <genexpr> at
0x000000000A4901F8>
>>> list(s)
>>> s.next()
4
>>> s.next()
9
>>> s.next()
```

```
>>> tuple(s)
(4, 9, 16, 25, 36, 49, 64, 81, 100, 121)
>>> s = ((i+2) ** 2 for i in range(10))
>>> t=list(s)
>>> list(t)
>>> g=((i+2) * *2 for I in range(10))
16
>>> s.next()
>>> t = ( g = ((i+2) ** 2 for I in range(10)))
>>> for i in [(i+2) ** 2 for i in range(10)]
>>> print i,
[4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

3.5 本章小结

本章重点介绍了列表和元组，它们都是序列对象，列表是可变对象（可以进行修改），而元组是不可变对象（一旦创建了就是固定的）。通过切片操作可以访问序列的一部分，其中切片需要两个索引号来指出切片的起始和结束位置。要想改变列表，则要对相应的位置进行赋值，或者使用赋值语句重写整个切片。

列表、元组属于有序序列，支持双向索引，支持使用负整数作为下标来访问其中的元素，－1 表示最后一个元素位置，－2 表示倒数第二个元素位置，以此类推。在 Python 中，同一个列表元素的数据类型可以各不相同，并且支持复杂数据类型的嵌套。

将列表或元组对象与一个整数进行 * 运算符操作，表示将对象中的元素进行重复并返回一个新的同类类型。＋运算符可以连接两个列表对象，但并不是原地修改列表，而是返回一个新列表，不对原列表对象做任何修改。并且该运算符涉及大量的元素赋值操作，效率较低，建议优先考虑使用列表对象的 append()方法。

虽然列表支持在列表中间任意位置插入和删除元素，但建议尽量从列表的尾部进行元素的增加与删除，这样可以获得更高的速度。切片操作不仅可以用来返回列表、元组中的部分元素，还可以对列表中的元素值进行修改，以及增加或删除列表中的元素。

列表推导式可以使用简洁的形式来满足特定需要的列表。

3.6 上机实验

上机实验 1　Python 列表与集合

【实验目的】 掌握 Python 语言的列表与集合数据结构的方法。

【实验内容及步骤】 编写程序，实现删除列表 s1 = ['b', 'c', 'd', 'b', 'c', 'a', 'a'] 中的重复元素。

方法一：

创建包含重复元素的列表 s1。

```
>>> s1 = ['b', 'c', 'd', 'b', 'c', 'a', 'a']
```

set()操作将原列表转换为一个无序不重复元素集合，使用 list()将集合转换为列表。

```
>>> s2 = list(set(s1))
```

打印输出去除重复元素后的列表 s2。

```
>>> print s2
```

方法二：

创建包含重复元素的列表 s1。

```
>>> s1 = ['b', 'c', 'd', 'b', 'c', 'a', 'a']
```

创建空列表 s1

```
>>> s2 = []。
```

使用列表推导式，从列表 s1 中挑选尚未存在于列表 s2 中的元素，再将该元素增加到列表 s2 中。

```
>>> [s2.append(s) for s in s1 if not s in s2]
```

打印输出去除重复元素后的列表 s2。

```
>>> print s2
```

运行结果：

```
['b', 'c', 'd', 'a']
```

上机实验 2　序列内置函数

【实验目的】 掌握序列内置函数 len()、max()、min()和 sum()。

【实验内容及步骤】 编写程序，求列表 lst = [5, 9, 3, 1, 12, 33, 2, 10]中的元素个数、最大值、最小值、元素之和。

创建列表 lst，包含元素 5，9，3，1，12，33，2，10。

```
>>> lst = [5, 9, 3, 1, 12, 33, 2, 10]
```

内置函数 len()返回序列中所包含元素的数量。

```
>>> len(lst)
```

运行结果：8

内置函数 max()返回序列中最大元素。

```
>>> max(lst)
```

运行结果：33

序列内置函数 min()返回序列中最小的元素。

```
>>> min(lst)
```

运行结果：1

内置函数 sum()可获取列表或元组各元素之和。

```
>>> sum(lst)
```

运行结果：75

上机实验 3　偶数变换

【实验目的】 对数列的奇偶数分别进行操作。

【实验内容及步骤】 编写程序，将列表 s = [1, 2, 3, 4, 5, 6, 7, 8, 9]中的偶数变成它的平方，奇数变成它的立方。

创建列表 s。

```
>>> s = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

使用列表推导式，对列表 s 中的每个元素判断其是否为偶数，即除以 2 的余数是否为 0：i%2 == 0；如果是偶数则执行平方操作：i ** 2；新的元素放入列表 new_s 中存储。

```
>>> new_s = [i ** 2 for i in s if i % 2 == 0]
```

打印输出列表。

```
>>> print new_s
```

运行结果如下：

```
[4, 16, 36, 64]
```

上机实验 4　元组数据结构

【实验目的】 掌握元组数据结构，元组与列表的转换和列表的排序方法。

【实验内容及步骤】 对元组 fruits = ('banana', 'grapes', 'apple', 'strawberry')进行升序排序，输出排序后的元组。

创建元组，元组是不可变序列，无法直接进行排序操作。

```
>>> fruits = ('banana', 'grapes', 'apple', 'strawberry')
```

将元组转换为列表。

```
>>> fruits_lst = list(fruits)
```

使用 sort()对列表进行排序，默认为升序。

```
>>> fruits_lst.sort()
```

将排序后的列表再转换为元组。

```
>>> fruits_new = tuple(fruits_lst)
```

打印输出排序后的元组。


```
>>> print fruits_new
```

运行结果如下：

```
('apple', 'banana', 'grapes', 'strawberry')
```

习题 3

一、单项选择题

1. Python 语句 print(type([1, 2, 3, 4]))的输出结果是(　　)。

A. <class'tuple'>　　B. <class 'dict'>

C. <class 'set'>　　D. <class 'list'>

2. Python 语句 print(type((1, 2, 3, 4)))的输出结果是(　　)。

A. <class'tuple'>　　B. <class 'dict'>

C. <class 'set'>　　D. <class 'list'>

3. Python 语句 a=[1, 2, 3, None, (), [ ],]; print(len(a))的输出结果是(　　)。

A. 4　　B. 5　　C. 6　　D. 7

4. Python 语句 s1=[4, 5, 6]; s2=s1; s1[1]=0; print(s2)的运行结果是(　　)。

A. [4, 5, 6]　　B. [0, 5, 6]

C. [4, 0, 6]　　D. 以上都不对

5. Python 语句 s=[1, 2, 3, 4]; s.append([5, 6]); print(len(s))的运行结果是(　　)。

A. 4　　B. 5　　C. 6　　D. 7

6. Python 语句 s1=[1, 2, 3, 4]; s2=[5, 6, 7]; print (len(s1+s2))的运行结果是(　　)。

A. 4　　B. 5　　C. 6　　D. 7

7. Python 语句 print(tuple(range(2)))的运行结果是(　　)。

A. (0, 1)　　B. [0, 1]　　C. (1, 2)　　D. [1, 2]

8. Python 语句 list(range(2))的运行结果是(　　)。

A. (0, 1)　　B. [0, 1]　　C. (1, 2)　　D. [1, 2]

9. Python 列表推导式[i for i in range(5) if i%2 != 0]的运行结果是(　　)。

A. [0, 1, 2, 3, 4]　　B. [1, 3]

C. [0, 2, 4]　　D. [1, 2, 3, 4, 5]

10. 在 Python 中,设有 s=('a', 'b', 'c', 'd', 'e'),则有：

(1) s[2]的值为(　　)。

A. 'a'　　B. 'b'

C. 'c'　　D. 'd'

(2) s[2:4]的值为(　　)。

A. ('a', 'b')　　B. ('b', 'c')

C. ('c', 'd')　　D. ('d', 'e')

(3) s[:3]的值为(　　)。

A. ('a', 'b', 'c')　　B. ('b', 'c', 'd')

C. ('c', 'd', 'e')　　D. 以上都不对

(4) s[3:]的值为(　　)。

A. ('a', 'b')　　B. ('b', 'c')

C. ('c', 'd')　　D. ('d', 'e')

(5) s[1::2]的值为(　　)。

A. ('a', 'b')　　B. ('b', 'c')

C. ('b', 'd')　　D. ('a', 'c')

(6) s[-2]的值为(　　)。

A. 'a'　　B. 'b'

C. 'c'　　D. 'd'

(7) s[::-1]的值为(　　)。

A. ('a', 'b', 'c', 'd', 'e')　　B. ('e', 'd', 'c', 'b', 'a')

C. 'a'　　D. 'e'

(8) s[-99:-5]的值为(　　)。

A. 'a'　　B. 'e'

C. ()　　D. 以上都不对

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

A. append()　　B. extend()

C. remove()　　D. delete()

12. 假设列表对象 aList 的值为[3, 4, 5, 6, 7, 9, 11, 13, 15, 17],那么 aList[3:7]的运行结果是(　　)。

A. [5, 6, 7, 9]　　B. [5, 6, 7, 9, 11]

C. [6, 7, 9, 11]　　D. [6, 7, 9, 11, 13]

13. 假设有一个列表 a,现要求从列表 a 中每 3 个元素取 1 个,并且将取到的元素组成新的列表 b,可以使用语句(　　)。

A. b=a[1:3]　　B. b=a[::3]

C. b=a[1::3]　　D. 以上都不对

14. 使用列表推导式生成包含 10 个数字 5 的列表,语句可以写为(　　)。

A. [i for i in range(10)]　　B. [i for i in range(5)]

C. [5 for i in range(10)]　　D. 以上都不对

15. Python 语句 s=['a', 'b']; s.extend('34')的运行结果是(　　)。

A. ['a', 'b', '34']　　B. ['a', 'b', ['34']]

C. ['a', 'b', ('3', '4')]　　D. ['a', 'b', '3', '4']

16. Python 语句 s=[1, 2, 3]; s * 3 的运行结果是(　　)。

A. [1, 1, 1, 2, 2, 2, 3, 3, 3]　　B. [1, 2, 3, 1, 2, 3, 1, 2, 3]

C. [3, 6, 9]　　D. [[1, 2, 3], [1, 2, 3], [1, 2, 3]]

17. Python 语句 t=('a', 'b', 'c', 'd', 'a'); t.count('a')的运行结果是(　　)。

A. 0　　B. 1

C. 2　　D. 以上都不对

18. Python 语句 s=[i ** 2 for i in range(4)]的运行结果是(　　)。

A. [0, 2, 4, 6]　　B. [0, 1, 4, 9]

C. [0, 1, 2, 3, 0, 1, 2, 3]]　　D. [1, 4, 9, 16]

19. Python 语句 s=([1, 2], 3); s[0][0]=5; s 的运行结果是(　　)。

A. ([1, 5], 3)　　B. ([5, 2], 3)

C. ([1, 2], 5)　　D. ([0, 1, 2], 3)

20. Python 语句 s=([1, 2], 3); s[0].append(4)的运行结果是(　　)。

A. ([4, 1, 2], 3)　　B. ([1, 2, 4], 3)

C. (4, [1, 2], 3)　　D. ([1, 2], [4], 3)

21. Python 语句 list(range(1,10,3))的执行结果为(　　)。

A. [0, 3, 6, 9]　　B. [1, 4, 7]

C. [1, 4, 7, 10]　　D. 以上都不对

二、多项选择题

1. Python 的有序序列有(　　)。

A. 列表　　B. 元组　　C. 字符串　　D. 集合

2. Python 中具有查找功能的有(　　)。

A. in　　B. not in　　C. count　　D. index

3. 关于 a or b 的描述正确的是(　　)。

A. 如果 a = True,b = True,则 a or b 等于 True

B. 如果 a = True,b = False,则 a or b 等于 True

C. 如果 a = True,b = False,则 a or b 等于 False

D. 如果 a = False,b = False,则 a or b 等于 False

4. 以下可以对列表进行排序的有(　　)。

A. sort()　　B. ascend()　　C. sorted()　　D. reversed()

5. 列表可以执行的操作有(　　)。

A. 索引　　B. 切片　　C. 嵌套　　D. 排序

6. 列表 s= [2, 4, 6, 8, 10],能得到[8, 10]的切片操作有(　　)。

A. s[3::]　　B. s[3:6]　　C. s[3:6:1]　　D. s[-2:]

7. 列表 s= [1, 2, 3, 4, 5],能得到[1, 3, 5]的切片操作有(　　)。

A. s[0:4:2]　　B. s[0:5:2]　　C. s[::2]　　D. s[::-2]

8. 列表 s=[2, 7, 5, 1 ,3],能够实现将该列表升序排列的有(　　)。

A. sorted(s)　　B. s.sort(reverse=False)

C. s.sort(reverse=True)　　D. s.reverse()

9. 可以得到元组(1，3，5)的有(　　)。

A. 1，3，5　　B. (1，3，5)

C. tuple(1，3，5)　　D. tuple([1，3，5])

10. 以下列表中长度为 2 的有(　　)。

A. [1，[2，3]]　　B. [[1，3]]

C. ['hello','hi']　　D. ;d=[['x','y'],[1,2,[3]]]

三、判断题

1. 针对列表操作，函数 sorted()和方法 sort()的功能完全相同。(　　)

2. 使用下标访问元组中的元素时，如果指定下标不存在，则抛出异常。(　　)

3. 元组与列表一样，都包括多个成员对象，只是分别使用方括号、圆括号。(　　)

4. 即使元组的成员对象中有列表，元组也是不可变的。(　　)

5. 两个相同类型的序列才能进行连接操作。(　　)

6. 表达式“[3] in [1，2，3，4]”的值为 False。(　　)

7. 列表对象的 sort()方法用来对列表元素进行原地排序，该函数返回值为 None。(　　)

8. 可以使用 del 命令来删除元组中的部分元素。(　　)

9. 列表和元组的主要区别在于，列表可以修改，元组则不能。(　　)

10. 列表、元组和字符串等序列类型均支持双向索引。(　　)

11. 字符串属于 Python 有序序列，与列表、元组一样都支持双向索引。(　　)

12. 通过连接操作符＋，可以连接两个序列(s1 和 s2)，形成一个新的序列对象。(　　)

13. 当列表不再使用时，使用 del 命令删除整个列表。(　　)

14. 使用下标直接访问列表中的元素时，如果指定下标不存在，则抛出异常。(　　)

15. 使用列表推导式能够实现嵌套列表的平铺。(　　)

16. 定义元组时所有元素放在一对圆括号中，且圆括号不可以省略。(　　)

17. 元组的访问和处理速度比列表更快。(　　)

18. tuple()可以把序列转换成元组。(　　)

19. 元组中只包含一个元素时，需要在元素后面添加逗号来消除歧义。(　　)

20. Python 列表、元组和字符串 s 的最后一个元素的下标为 len(s)。(　　)

21. Python 列表中所有元素必须为相同类型的数据。(　　)

22. 对于列表而言，在尾部追加元素比在中间位置插入元素速度更快一些，尤其是对于包含大量元素的列表。(　　)

23. 删除列表中重复元素最简单的方法是将其转换为集合后再重新转换为列表。(　　)

24. 列表对象的 pop()方法默认删除并返回最后一个元素，如果列表已空则抛出异常。(　　)

25. 列表可以作为字典的“键”。

26. 元组可以作为字典的“键”。

27. 使用 Python 列表的方法 insert()为列表插入元素时会改变列表中插入位置之后元素的索引。 (　　)

28. 使用 del 命令或者列表对象的 remove()方法删除列表中元素时会影响列表中部分元素的索引。 (　　)