



# 《Python程序设计基础教程（微课版）》

<http://dblab.xmu.edu.cn/post/python>



## 第4章 序列





# 提纲

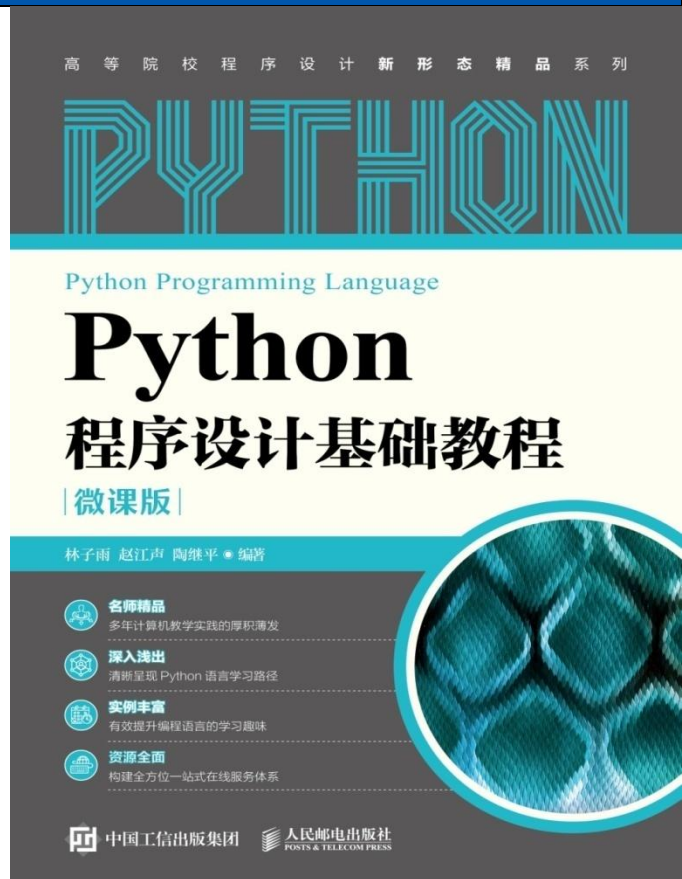
- 4.1 列表
- 4.2 元组
- 4.3 字典
- 4.4 集合

本PPT是如下教材的配套讲义：

《Python程序设计基础教程（微课版）》

厦门大学 林子雨,赵江声,陶继平 编著，人民邮电出版社

《Python程序设计基础教程（微课版）》教材官方网站：  
<http://dblab.xmu.edu.cn/post/python>





## 4.1 列表

列表是最常用的Python数据类型，列表的数据项不需要具有相同的类型。在形式上，只要把逗号分隔的不同的数据项使用方括号括起来，就可以构成一个列表，例如：

```
['hadoop', 'spark', 2021, 2010]
```

```
[1, 2, 3, 4, 5]
```

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

```
['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
```



# 4.1 列表

- 4.1.1 列表的创建与删除
- 4.1.2 访问列表元素
- 4.1.3 添加、删除、修改列表元素
- 4.1.4 对列表进行统计计算
- 4.1.5 对列表进行排序
- 4.1.6 成员资格判断
- 4.1.7 切片操作
- 4.1.8 列表推导式
- 4.1.9 二维列表

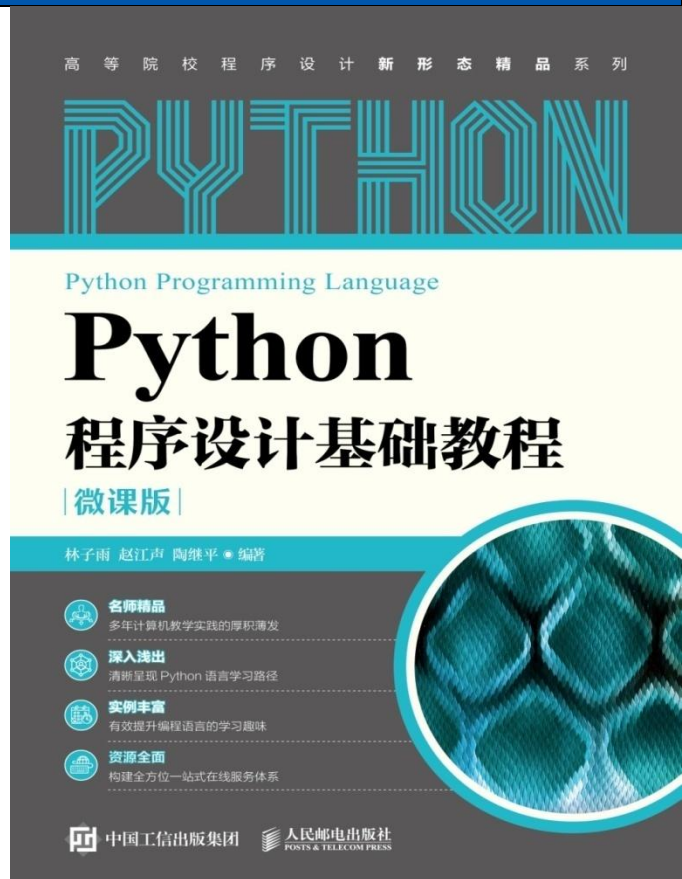
本PPT是如下教材的配套讲义：

《Python程序设计基础教程（微课版）》

厦门大学 林子雨,赵江声,陶继平 编著，人民邮电出版社

《Python程序设计基础教程（微课版）》教材官方网站：

<http://dblab.xmu.edu.cn/post/python>





## 4.1.1 列表的创建与删除

### 1. 使用赋值运算符直接创建列表

同其他类型的Python变量一样，在创建列表时，也可以直接使用赋值运算符“=”将一个列表赋值给变量。例如，以下都是合法的列表定义：

```
student = ['小明', '男', 2010, 10]
```

```
num = [1, 2, 3, 4, 5]
```

```
motto = ["自强不息", "止于至善"]
```

```
list = ['hadoop', '年度畅销书', [2020, 12000]]
```

可以看出，列表里面的元素仍然可以是列表。需要注意的是，尽管一个列表中可以放入不同类型的数据，但是，为了提高程序的可读性，一般建议在一个列表中只出现一种数据类型。



## 4.1.1 列表的创建与删除

### 2. 创建空列表

在Python中创建新的空列表的方法如下：

```
empty_list = []
```

这时生成的`empty_list`就是一个空列表，如果使用内置函数`len()`去获取它的长度，结果会返回0，如下所示：

```
>>> empty_list = []
```

```
>>> print(type(empty_list))
```

```
<class 'list'>
```

```
>>> print(len(empty_list))
```

```
0
```



## 4.1.1 列表的创建与删除

### 3.创建数值列表

Python中的数值列表很常用，用于存储数值集合。Python提供了`list()`函数，它可以将`range()`对象、字符串、元组或其他可迭代类型的数据转换为列表。例如，创建一个包含1到5的整数列表：

```
>>> num_list = list(range(1,6))
```

```
>>> print(num_list)
```

```
[1, 2, 3, 4, 5]
```

下面创建一个包含1到10中的奇数的列表：

```
>>> num_list = list(range(1,10,2))
```

```
>>> print(num_list)
```

```
[1, 3, 5, 7, 9]
```



## 4.1.1 列表的创建与删除

### 4.列表的删除

当列表不再使用时，可以使用`del`命令删除整个列表，如果列表对象所指向的值不再有其他对象指向，`Python`将同时删除该值。下面是一个具体实例：

```
>>> motto = ['自强不息','止于至善']
```

```
>>> motto
```

```
['自强不息','止于至善']
```

```
>>> del motto
```

```
>>> motto
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#43>", line 1, in <module>
```

```
    motto
```

```
NameError: name 'motto' is not defined
```

从上面代码的执行结果可以看出，删除列表对象`motto`以后，该对象就不存在了，再次访问时就会抛出异常。





## 4.1.2 访问列表元素

列表索引从 0 开始，第二个索引是 1，依此类推（如图4-1所示）。

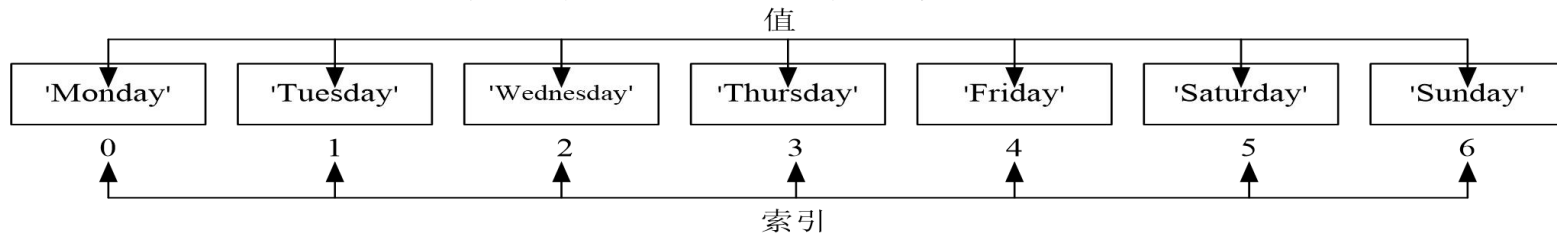


图4-1 列表的索引

下面是一段代码实例：

```
>>> list = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
>>> print(list[0])
Monday
>>> print(list[1])
Tuesday
>>> print(list[2])
Wednesday
```



## 4.1.2 访问列表元素

索引也可以从尾部开始，最后一个元素的索引为-1，往前一位为-2，依此类推（如图4-2所示）。

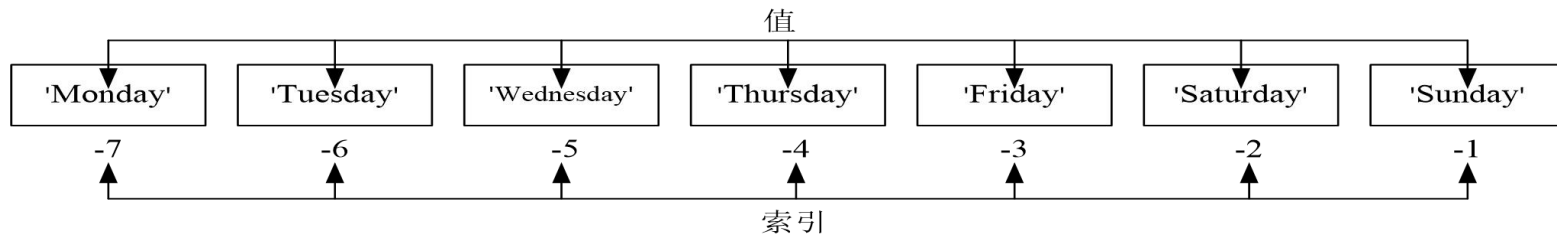


图4-2 列表的反向索引

下面是一段代码实例：

```
>>> list = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
>>> print(list[-1])
Sunday
>>> print(list[-2])
Saturday
>>> print(list[-3])
Friday
```



## 4.1.2 访问列表元素

可以使用for循环实现列表的遍历。

【例4-1】 使用for循环实现列表的遍历。

```
01      list1 = ["hadoop", "spark", "flink", "storm"]  
02      for element in list1:  
03          print(element)
```

该程序的执行结果如下：

```
hadoop  
spark  
flink  
storm
```



## 4.1.3 添加、删除、修改列表元素

### 1. 列表元素的添加

在实际应用中，我们经常需要向列表中动态增加元素。Python主要提供了五种添加列表元素的方法。

#### (1) **append()**

列表对象的**append()**方法用于在列表的末尾追加元素，举例如下：

```
>>> books = ["hadoop","spark"]
>>> len(books) #获取列表的元素个数
2
>>> books.append("flink")
>>> books.append("hbase")
>>> len(books) #获取列表的元素个数
4
```



## 4.1.3 添加、删除、修改列表元素

### (2) insert()

列表对象的insert()方法用于将元素添加至列表的指定位置，举例如下：

```
>>> num_list = [1,2,3,4,5]
>>> num_list
[1, 2, 3, 4, 5]
>>> num_list.insert(2,9)
>>> num_list
[1, 2, 9, 3, 4, 5]
```

在上面的代码中，`num_list.insert(2,9)`表示向列表的第2个元素后面添加一个新的元素9。这样会让插入位置后面所有的元素进行移动，如果列表元素较多，这样操作会影响处理速度。



## 4.1.3 添加、删除、修改列表元素

### (3) `extend()`

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，举例如下：

```
>>> num_list = [1,1,2]
```

```
>>> id(num_list)
```

```
47251200
```

```
>>> num_list.extend([3,4])
```

```
>>> num_list
```

```
[1, 1, 2, 3, 4]
```

```
>>> id(num_list)
```

```
47251200
```

可以看出，`num_list`调用`extend()`方法将目标列表的所有元素添加到本列表的尾部，属于原地操作（内存地址没有发生变化），不创建新的列表对象。



## 4.1.3 添加、删除、修改列表元素

### (4) “+” 运算符

可以使用“+”运算符来把元素添加到列表中。实例如下：

```
>>> num_list = [1,2,3]
>>> id(num_list)
46818176
>>> num_list = num_list + [4]
>>> num_list
[1, 2, 3, 4]
>>> id(num_list)
47251392
```

可以看出，`num_list`在添加新元素以后，内存地址发生了变化，这是因为，添加新元素时，创建了一个新的列表，并把原列表中的元素和新元素依次复制到新列表的内存空间。当列表中的元素较多时，该操作速度会比较慢。



## 4.1.3 添加、删除、修改列表元素

### (5) “\*” 运算符

Python提供了“\*”运算符来扩展列表对象。可以将列表与整数相乘，生成一个新列表，新列表是原列表中元素的重复。具体实例如下：

```
>>> num_list = [1,2,3]
>>> other_list = num_list
>>> id(num_list)
47170496
>>> id(other_list)
47170496
>>> num_list = num_list*3
>>> num_list
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> other_list
[1, 2, 3]
>>> id(num_list)
50204480
>>> id(other_list)
47170496
```





## 4.1.3 添加、删除、修改列表元素

### 2. 列表元素的删除

#### (1) 使用`del`语句删除指定位置的元素

在Python中可以使用`del`语句删除指定位置的列表元素，实例如下：

```
>>> demo_list = ['a','b','c','d']
>>> del demo_list[0]
>>> demo_list
['b', 'c', 'd']
```

#### (2) 使用`pop()`删除元素

`pop()`可删除列表末尾的元素，实例如下：

```
>>> demo_list = ['a','b','c','d']
>>> demo_list.pop()
'd'
>>> demo_list
['a', 'b', 'c']
```



## 4.1.3 添加、删除、修改列表元素

### (3) 使用**remove()**根据值删除元素

可以使用**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。实例如下：

```
>>> num_list = [1,2,3,4,5,6,7]
>>> num_list.remove(4)
>>> num_list
[1, 2, 3, 5, 6, 7]
```



## 4.1.3 添加、删除、修改列表元素

### 3. 列表元素的修改

修改列表元素的操作较为简单，实例如下：

```
>>> books = ["hadoop","spark","flink"]
```

```
>>> books
```

```
['hadoop', 'spark', 'flink']
```

```
>>> books[2] = "storm"
```

```
>>> books
```

```
['hadoop', 'spark', 'storm']
```



## 4.1.4 对列表进行统计计算

### 1. 获取指定元素出现的次数

可以使用列表对象的`count()`方法来获取指定元素在列表中的出现次数。实例如下：

```
>>> books = ["hadoop","spark","flink","spark"]
>>> num = books.count("spark")
>>> print(num)
2
```



## 4.1.4 对列表进行统计计算

### 2. 获取指定元素首次出现的下标

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，语法格式为：

`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。

如果列表对象中不存在指定元素，则会抛出异常。实例如下：

```
>>> books = ["hadoop","spark","flink","spark"]
```

```
>>> position = books.index("spark")
```

```
>>> print(position)
```

```
1
```



## 4.1.4 对列表进行统计计算

### 3.统计数值列表的元素和

Python提供了`sum()`函数用于统计数值列表中各个元素的和，语法格式如下：

```
sum(aList[,start])
```

其中，`aList`表示要统计的列表，`start`表示统计结果是从哪个数开始累加，如果没有指定，默认值为0。具体实例如下：

```
>>> score = [84,82,95,77,65]
>>> total = sum(score)  #从0开始累加
>>> print("总分数是：",total)
总分数是： 403
>>> totalplus = sum(score,100)  #从100开始累加
>>> print("增加100分后的总分数是：",totalplus)
增加100分后的总分数是： 503
```



## 4.1.5 对列表进行排序

### 1. 使用列表对象的`sort()`方法排序

可以使用列表对象的`sort()`方法对列表中的元素进行排序，排序后列表中的元素顺序将会发生改变。具体语法格式如下：

```
aList.sort(key=None,reverse=False)
```

其中，**aList**表示要排序的列表，**key**参数来指定一个函数，此函数将在每个元素比较前被调用，例如，可以设置“**key=str.lower**”来忽略字符串的大小写；**reverse**是一个可选参数，如果值为**True**，则表示降序排序，如果值为**False**，则表示升序排序，默认为升序排序。



## 4.1.5 对列表进行排序

具体实例如下：

```
>>> num_list = [1,2,3,4,5,6,7,8,9,10]
>>> import random
>>> random.shuffle(num_list)  #打乱排序
>>> num_list
[4, 9, 10, 6, 2, 8, 1, 3, 7, 5]
>>> num_list.sort()  #升序排序
>>> num_list
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> num_list.sort(reverse=True)  #降序
排序
>>> num_list
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```





## 4.1.5 对列表进行排序

当列表中的元素类型是字符串时，`sort()`函数排序的规则是，先对大写字母进行排序，然后再对小写字母进行排序。如果在排序时不考虑字母大小写，则需要设置“`key=str.lower`”。实例如下：

```
>>> books = ["hadoop","Hadoop","Spark","spark","flink","Flink"]
>>> books.sort()    #默认区分字母大小写
>>> books
['Flink', 'Hadoop', 'Spark', 'flink', 'hadoop', 'spark']
>>> books.sort(key=str.lower)  #不区分字母大小写
>>> books
['Flink', 'flink', 'Hadoop', 'hadoop', 'Spark', 'spark']
```



## 4.1.5 对列表进行排序

### 2.使用内置的sorted()函数排序

Python提供了一个内置的全局sorted()方法，可以用来对列表排序生成新的列表，原列表的元素顺序保持不变，语法格式如下：

```
sorted(aList,key=None,reverse=False)
```



## 4.1.5 对列表进行排序

其中，**aList**表示要排序的列表，**key**参数来指定一个函数，此函数将在每个元素比较前被调用，例如，可以设置“**key=str.lower**”来忽略字符串的大小写；**reverse**是一个可选参数，如果值为**True**，则表示降序排序，如果值为**False**，则表示升序排序，默认为升序排序。

具体实例如下：

```
>>> score = [84,82,95,77,65]
>>> score_asc = sorted(score)  #升序排序
>>> score_asc
[65, 77, 82, 84, 95]
>>> score    #原列表不变
[84, 82, 95, 77, 65]
>>> score_desc = sorted(score,reverse=True)  #降序排序
>>> score_desc
[95, 84, 82, 77, 65]
>>> score    #原列表不变
[84, 82, 95, 77, 65]
```



## 4.1.6 成员资格判断

如果需要判断列表中是否存在指定的值，可以采用四种不同的方式：`in`、`not in`、`count()`、`index()`。

可以使用`in`操作符判断一个值是否存在于列表中，实例如下：

```
>>> books = ["hadoop","spark","flink","spark"]
```

```
>>> "hadoop" in books
```

```
True
```

```
>>> "storm" in books
```

```
False
```



## 4.1.6 成员资格判断

可以使用`not in`操作符判断一个值是否不在列表中，实例如下：

```
>>> books = ["hadoop","spark","flink","spark"]
```

```
>>> "storm" not in books
```

```
True
```

```
>>> "hadoop" not in books
```

```
False
```

可以使用列表对象的`count()`方法，如果指定的值存在，则返回大于0的数，如果返回0，则表示不存在，实例如下：

```
>>> books = ["hadoop","spark","flink","spark"]
```

```
>>> books.count("spark")
```

```
2
```

```
>>> books.count("storm")
```

```
0
```



## 4.1.6 成员资格判断

可以使用`index()`方法查看指定值在列表中的位置，如果列表中存在指定值，则会返回该值第一次出现的位置，否则会抛出错误，实例如下：

```
>>> books = ["hadoop","spark","flink","spark"]
```

```
>>> books.index("spark")
```

```
1
```

```
>>> books.index("storm")
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#145>", line 1, in <module>
```

```
    books.index("storm")
```

```
ValueError: 'storm' is not in list
```



## 4.1.7 切片操作

切片操作是访问序列中元素的一种方法，切片操作不是列表特有的，Python中的有序序列（如字符串、元组）都支持切片操作。切片的返回结果类型和切片对象类型一致，返回的是切片对象的子序列，比如，对一个列表切片返回一个列表，对一个字符串切片返回字符串。

通过切片操作可以生成一个新的列表（不会改变原列表），切片操作的语法格式如下：

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

其中，**listname**表示列表名称；**start**是切片起点的索引，如果不指定，默认值为0；**end**是切片终点的索引（但是切片结果不包括终点索引的值），如果不指定，默认为列表的长度；**step**是步长，默认值是1（也就是一个一个依次遍历列表中的元素），当省略步长时，最后一个冒号也可以省略。



## 4.1.7 切片操作

下面是一些切片操作的具体实例：

```
>>> num_list = [13,54,38,93,28,74,59,92,85,66]
>>> num_list[1:3] #从索引1的位置开始取，取到索引3的位置（不含索引3）
[54, 38]
>>> num_list[:3] #从索引0的位置开始取，取到索引3的位置（不含索引3）
[13, 54, 38]
>>> num_list[1:] #从索引1的位置开始取，取到列表尾部，步长为1
[54, 38, 93, 28, 74, 59, 92, 85, 66]
>>> num_list[1::] #从索引1的位置开始取，取到列表尾部，步长为1
[54, 38, 93, 28, 74, 59, 92, 85, 66]
>>> num_list[:] #从头取到尾，步长为1
[13, 54, 38, 93, 28, 74, 59, 92, 85, 66]
>>> num_list[::] #从头取到尾，步长为1
[13, 54, 38, 93, 28, 74, 59, 92, 85, 66]
>>> num_list[::-1] #从尾取到头，逆向获取列表元素
[66, 85, 92, 59, 74, 28, 93, 38, 54, 13]
>>> num_list[::2] #从头取到尾，步长为2
[13, 38, 28, 59, 85]
```





## 4.1.7 切片操作

```
>>> num_list[2:6:2] #从索引2的位置开始取元素，取到索引6的位置（不含索引6）
[38, 28]
>>> num_list[0:100:1] #从索引0的位置开始取元素，取到索引100的位置
[13, 54, 38, 93, 28, 74, 59, 92, 85, 66]
>>> num_list[100:] #从索引100的位置开始取元素，不存在元素
[]
>>> num_list[8:2:-2] #从索引8的位置逆向取元素，取到索引2的位置（不含索引2）
[85, 59, 28]
>>> num_list[3:-1] #从索引3的位置取到倒数第1个元素（不包含倒数第1个元素）
[93, 28, 74, 59, 92, 85]
>>> num_list[-2] #取出倒数第2个元素
85
>>> num_list #原列表没有发生变化
[13, 54, 38, 93, 28, 74, 59, 92, 85, 66]
```



## 4.1.7 切片操作

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，实例如下：

```
>>> num_list =  
[13,54,38,93,28,74,59,92,85,66]  
>>> del num_list[:4]  
>>> num_list  
[28, 74, 59, 92, 85, 66]
```



## 4.1.8 列表推导式

列表推导式可以利用`range`对象、元组、列表、字典和集合等数据类型，快速生成一个满足指定需求的列表。

列表推导式的语法格式如下：

[表达式 for 迭代变量 in 可迭代对象 [if 条件表达式]]

其中，[if 条件表达式]不是必须的，可以使用，也可以省略。

例如，利用0到9的平方生成一个整数列表，代码如下：

```
>>> a_range = range(10)
>>> a_list = [x * x for x in a_range]
>>> a_list
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```



## 4.1.8 列表推导式

还可以在列表推导式中添加if条件语句，这样列表推导式将只迭代那些符合条件的元素，实例如下：

```
>>> b_list = [x * x for x in a_range if x % 2 == 0]
>>> b_list
[0, 4, 16, 36, 64]
```

上面的列表推导式都只包含一个循环，实际上可以使用多重循环，实例如下：

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



## 4.1.8 列表推导式

上面代码中，**x**是遍历**range(3)**的迭代变量（计数器），因此**x**可迭代3次；**y**是遍历 **range(2)**的计数器，因此**y**可迭代2次。因此，表达式(**x**, **y**)一共会迭代6次。

Python还支持类似于三层嵌套的**for**表达式，实例如下：

```
>>> d_list = [[x, y, z] for x in range(2) for y in range(2) for z in range(2)]
>>> d_list
[[0, 0, 0], [0, 0, 1], [0, 1, 0], [0, 1, 1], [1, 0, 0], [1, 0, 1], [1, 1, 0], [1, 1, 1]]
```



## 4.1.8 列表推导式

对于包含多个循环的for表达式，同样可以指定if条件。例如，要将两个列表中的数值按“能否整除”的关系配对在一起。比如列表list1中包含5，列表list2中包含20，20可以被5整除，那么就将20和5配对在一起。实现上述需求的代码如下：

```
>>> list1 = [3, 5, 7, 11]
```

```
>>> list2 = [20, 15, 33, 24, 27, 58, 46, 121, 49]
```

```
>>> result = [(x, y) for x in list1 for y in list2 if y % x == 0]
```

```
>>> result
```

```
[(3, 15), (3, 33), (3, 24), (3, 27), (5, 20), (5, 15), (7, 49), (11, 33), (11, 121)]
```



## 4.1.9 二维列表

所谓的“二维列表”，是指列表中的每个元素仍然是列表。比如，下面就是一个二维列表的实例：

```
['自','强','不','息'],  
['止','于','至','善']
```

可以通过直接赋值的方式来创建二维列表，代码如下：

```
>>> dim2_list = [['自','强','不','息'],['止','于','至','善']]  
>>> dim2_list  
[['自', '强', '不', '息'], ['止', '于', '至', '善']]
```



## 4.1.9 二维列表

还可以通过for循环来为二维列表赋值。

【例4-2】通过for循环来为二维列表赋值。

```
01      # create_list.py
02      dim2_list = []          #创建一个空列表
03      for i in range(3):
04          dim2_list.append([])  #为空列表添加的每个元素依然是空列表
05          for j in range(4):
06              dim2_list[i].append(j) #为内层列表添加元素
07      print(dim2_list)
```

该程序的执行结果如下：

```
[[0, 1, 2, 3], [0, 1, 2, 3], [0, 1, 2, 3]]
```





## 4.1.9 二维列表

此外，也可以使用列表推导式来创建二维列表，代码如下：

```
>>> dim2_list = [[j for j in range(4)] for i in range(3)]  
>>> dim2_list  
[[0, 1, 2, 3], [0, 1, 2, 3], [0, 1, 2, 3]]
```

访问二维数组时，可以使用下标来定位，实例如下：

```
>>> dim2_list = [['自','强','不','息'],['止','于','至','善']]  
>>> dim2_list[1][2]  
'至'
```



## 4.2 元组

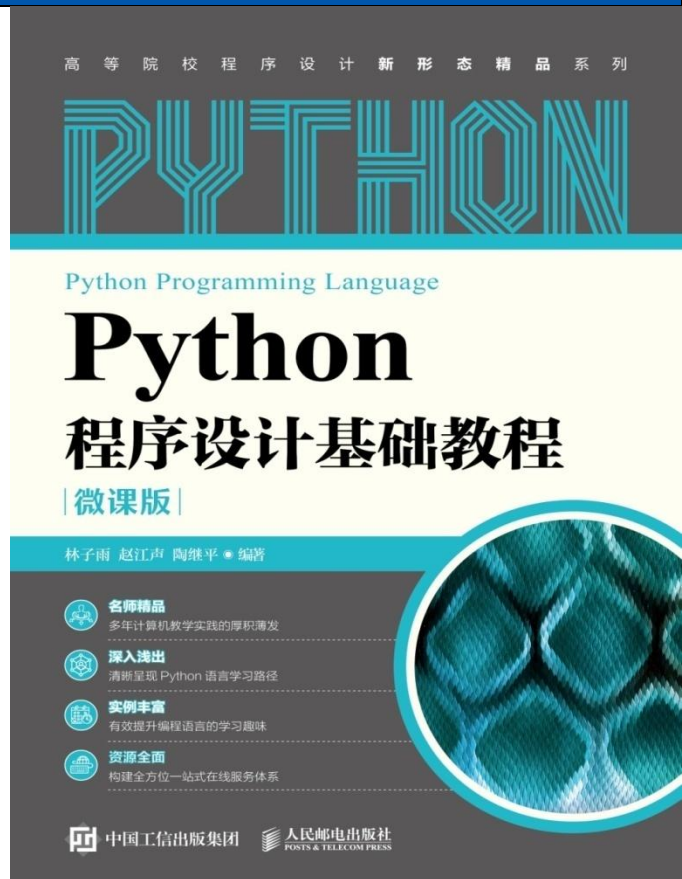
- 4.2.1 创建元组
- 4.2.2 访问元组
- 4.2.3 修改元组
- 4.2.4 删除元组
- 4.2.5 元组推导式
- 4.2.6 元组的常用内置函数
- 4.2.7 元组与列表的区别
- 4.2.8 序列封包和序列解包

本PPT是如下教材的配套讲义：

《Python程序设计基础教程（微课版）》

厦门大学 林子雨,赵江声,陶继平 编著，人民邮电出版社

《Python程序设计基础教程（微课版）》教材官方网站：  
<http://dblab.xmu.edu.cn/post/python>





## 4.2.1 创建元组

元组的创建和列表的创建很相似，不同之处在于，创建列表时使用的是方括号，而创建元组时则需要使用圆括号。元组的创建方法很简单，只需要在圆括号中添加元素，并使用逗号隔开即可，具体实例如下：

```
>>> tuple1 = ('hadoop','spark',2008,2009)
```

```
>>> tuple2 = (1,2,3,4,5)
```

```
>>> tuple3 = ('hadoop',2008,("大数据","分布式计算"),["spark","flink","storm"])
```

创建空元组的方法如下：

```
>>> tuple1 = ()
```



## 4.2.1 创建元组

需要注意的是，当元组中只包含一个元素时，需要在元素后面添加逗号，否则括号会被当作运算符使用，实例如下：

```
>>> tuple1 = (20)
```

```
>>> type(tuple1)
```

```
<class 'int'>
```

```
>>> tuple1 = (50,)
```

```
>>> type(tuple1)
```

```
<class 'tuple'>
```

也可以使用`tuple()`函数和`range()`函数来生成数值元组，实例如下：

```
>>> tuple1 = tuple(range(1,10,2))
```

```
>>> tuple1
```

```
(1, 3, 5, 7, 9)
```



## 4.2.2 访问元组

可以使用下标索引来访问元组中的元素，实例如下：

```
>>> tuple1 = ("hadoop", "spark", "flink", "storm")
>>> tuple1[0]
'hadoop'
>>> tuple1[1]
'spark'
```

对于元组而言，也可以象列表一样，采用切片的方式来获取指定的元素，实例如下：

```
>>> tuple1 = (1,2,3,4,5,6,7,8,9)
>>> tuple1[2:5]
(3, 4, 5)
```



## 4.2.2 访问元组

还可以使用for循环实现元组的遍历。

【例4-3】使用for循环实现元组的遍历。

```
01      # for_tuple.py
02      tuple1 = ("hadoop", "spark", "flink", "storm")
03      for element in tuple1:
04          print(element)
```

该程序的执行结果如下：

```
hadoop
spark
flink
storm
```



## 4.2.3 修改元组

元组中的元素值是不允许修改的，实例如下：

```
>>> tuple1 = ("hadoop", "spark", "flink")
```

```
>>> tuple1[0]
```

```
'hadoop'
```

```
>>> tuple1[0] = 'storm' #修改元组中的元素值，不允许，会报错
```

```
Traceback (most recent call last):
```

```
File "<pyshell#2>", line 1, in <module>
```

```
    tuple1[0] = 'storm'
```

```
TypeError: 'tuple' object does not support item assignment
```



## 4.2.3 修改元组

虽然元组中的元素值是不允许修改的，但是我们可以对元组进行连接组合，实例如下：

```
>>> tuple1 = ("hadoop", "spark", "flink")
>>> tuple2 = ("java","python","scala")
>>> tuple3 = tuple1 + tuple2
>>> tuple3
('hadoop', 'spark', 'flink', 'java', 'python', 'scala')
```

此外，也可以对元组进行重新赋值来改变元组的值，实例如下：

```
>>> tuple1 = (1,2,3)
>>> tuple1
(1, 2, 3)
>>> tuple1 = (4,5,6)
>>> tuple1
(4, 5, 6)
```





## 4.2.4 删除元组

元组属于不可变序列，无法删除元组中的部分元素，只能使用`del`命令删除整个元组对象，具体语法格式如下：

```
del tuplename
```

其中，`tuplename`表示要删除元组的名称。具体实例如下：

```
>>> tuple1 = ("hadoop", "spark", "flink", "storm")
```

```
>>> del tuple1
```

```
>>> tuple1
```

```
Traceback (most recent call last):
```

```
File "<pyshell#79>", line 1, in <module>
```

```
tuple1
```

```
NameError: name 'tuple1' is not defined
```

可以看出，当一个元组被删除以后，就不能再次引用，否则会抛出异常。



## 4.2.5 元组推导式

和生成列表一样，我们也可以使用元组推导式快速生成元组。元组推导式的语法格式如下：

(表达式 **for** 迭代变量 **in** 可迭代对象 [**if** 条件表达式] )

其中，[**if** 条件表达式]不是必须的，可以使用，也可以省略。

通过和列表推导式做对比可以发现，除了元组推导式是用圆括号将各部分括起来，而列表推导式用的是方括号，其它完全相同。不仅如此，元组推导式和列表推导式的用法也完全相同。例如，可以使用下面的代码生成一个包含数字1到9的元组：

```
>>> tuple1 = (x for x in range(1,10))
>>> tuple1
<generator object <genexpr> at 0x0000000002C7FC80>
```

可以看出，使用元组推导式生成的结果并不是一个元组，而是一个生成器对象，这一点和列表推导式是不同的。



## 4.2.5 元组推导式

如果我们想要使用元组推导式获得新元组或新元组中的元素，可以使用如下方式：

```
>>> tuple1 = (x for x in range(1,10))
>>> tuple(tuple1)
(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

或者也可以使用\_\_next\_\_()方法遍历生成器对象来获得各个元素，例如：

```
>>> tuple1 = (x for x in range(1,10))
>>> print(tuple1.__next__())
1
>>> print(tuple1.__next__())
2
>>> print(tuple1.__next__())
3
>>> tuple1 = tuple(tuple1)
>>> tuple1
( 4, 5, 6, 7, 8, 9)
```



## 4.2.6 元组的常用内置函数

元组的常用内置函数如下：

- `len(tuple)`: 计算元组大小，即元组中的元素个数；
- `max(tuple)`：返回元组中的元素最大值；
- `min(tuple)`：返回元组中元素最小值；
- `tuple(seq)`: 将序列转为元组。



## 4.2.6 元组的常用内置函数

【例4-4】常用的元组内置函数应用实例。

```
01      # tuple_function.py
02      tuple1 = ("hadoop", "spark", "flink", "storm")
03      #计算元组的大小
04      len_size = len(tuple1)
05      print("元组大小是: ",len_size)
06      # 返回元组元素最大值和最小值
07      tuple_number = (1,2,3,4,5)
08      max_number = max(tuple_number)
09      min_number = min(tuple_number)
10      print("元组最大值是: ",max_number)
11      print("元组最小值是: ",min_number)
12      # 将列表转为元组
13      list1 = ["hadoop", "spark", "flink", "storm"]
14      tuple2 = tuple(list1)
15      # 打印tuple2数据类型
16      print("tuple2的数据类型是: ",type(tuple2))
```



## 4.2.6 元组的常用内置函数

该程序的执行结果如下：

元组大小是： 4

元组最大值是： 5

元组最小值是： 1

tuple2的数据类型是： <class 'tuple'>



## 4.2.7 元组与列表的区别

元组和列表都属于序列，二者的区别主要体现在以下几个方面：

(1) 列表属于可变序列，可以随时修改或删除列表中的元素，比如使用 `append()`、`extend()`、`insert()` 向列表添加元素，使用 `del`、`remove()` 和 `pop()` 删除列表中的元素。元组属于不可变序列，没有 `append()`、`extend()` 和 `insert()` 等方法，不能修改其中的元素，也没有 `remove()` 和 `pop()` 方法，不能从元组中删除元素，更无法对元组元素进行 `del` 操作。

(2) 元组和列表都支持切片操作，但是，列表可以使用切片方式来修改其中的元素，而元组则不支持使用切片方式来修改其中的元素。

(3) 元组的访问和处理速度比列表快。如果只是对元素进行遍历，而不需要对元素进行任何修改，那么一般建议使用元组而非列表。

(4) 作为不可变序列，与整数、字符串一样，元组可以作为字典的键，而列表则不可以。



## 4.2.7 元组与列表的区别

在实际应用中，经常需要在元组和列表之间进行转换，具体方法如下：

(1) `tuple()`函数可以接受一个列表作为参数，返回同样元素的元组；

(2) `list()`函数可以接受一个元组作为参数，返回同样元素的列表。

下面是元组和列表互相转换的实例：

```
>>> list1 = ["hadoop", "spark", "flink", "storm"]
>>> tuple1 = tuple(list1)  #把列表转换成元组
>>> tuple1
('hadoop', 'spark', 'flink', 'storm')
>>> print("tuple1的数据类型是：",type(tuple1))
tuple1的数据类型是： <class 'tuple'>
>>> tuple2 = (1,2,3,4,5)
>>> list2 = list(tuple2)  #把元组转换成列表
>>> list2
[1, 2, 3, 4, 5]
>>> print("list2的数据类型是：",type(list2))
list2的数据类型是： <class 'list'>
```





## 4.2.8 序列封包和序列解包

程序把多个值赋给一个变量时，Python会自动将多个值封装成元组，这种功能被称为“序列封包”。下面是一个序列封包的实例：

```
>>> values = 1, 2, 3
```

```
>>> values
```

```
(1, 2, 3)
```

```
>>> type(values)
```

```
<class 'tuple'>
```

```
>>> values[1]
```

```
2
```



## 4.2.8 序列封包和序列解包

程序允许将序列（元组或列表等）直接赋值给多个变量，此时序列的各元素会被依次赋值给每个变量（要求序列的元素个数和变量的个数相等），这种功能被称为“序列解包”。可以使用序列解包功能对多个变量同时赋值，实例如下：

```
>>> a, b, c = 1, 2, 3
>>> print(a, b, c)
1 2 3
```

可以对range对象进行序列解包，实例如下：

```
>>> a, b, c = range(3)
>>> print(a, b, c)
0 1 2
```



## 4.2.8 序列封包和序列解包

可以将元组的各个元素依次赋值给多个变量，实例如下：

```
>>> a_tuple = tuple(range(1, 10, 2))
>>> print(a_tuple)
(1, 3, 5, 7, 9)
>>> a, b, c, d, e = a_tuple
>>> print(a, b, c, d, e)
1 3 5 7 9
```

下面是一个关于列表的序列解包的实例：

```
>>> a_list = [1, 2, 3]
>>> x, y, z = a_list
>>> print(x, y, z)
1 2 3
```



## 4.3 字典

字典也是Python提供的一种常用的数据结构，它用于存放具有映射关系的数据。比如有一份学生成绩表数据，语文67分，数学91分，英语78分，如果使用列表保存这些数据，则需要两个列表，即["语文","数学","英语"]和[67,91,78]。但是，使用两个列表来保存这组数据以后，就无法记录两组数据之间的关联关系。为了保存这种具有映射关系的数据，Python 提供了字典，字典相当于保存了两组数据，其中一组数据是关键数据，被称为“键”（key）；另一组数据可通过键来访问，被称为“值”（value）。



## 4.3 字典

字典具有如下特性：

- （1）字典的元素是“键值对”，由于字典中的键是非常关键的数据，而且程序需要通过键来访问值，因此字典中的键不允许重复，必须是唯一值，而且键必须不可变；
- （2）字典不支持索引和切片，但可以通过“键”查询“值”；
- （3）字典是无序的对象集合，列表是有序的对象集合，两者之间的区别在于，字典当中的元素是通过键来存取的，而不是通过偏移量存取；
- （4）字典是可变的，并且可以任意嵌套。



## 4.3 字典

### 4.3.1 字典的创建与删除

### 4.3.2 访问字典

### 4.3.3 添加、修改和删除字典元素

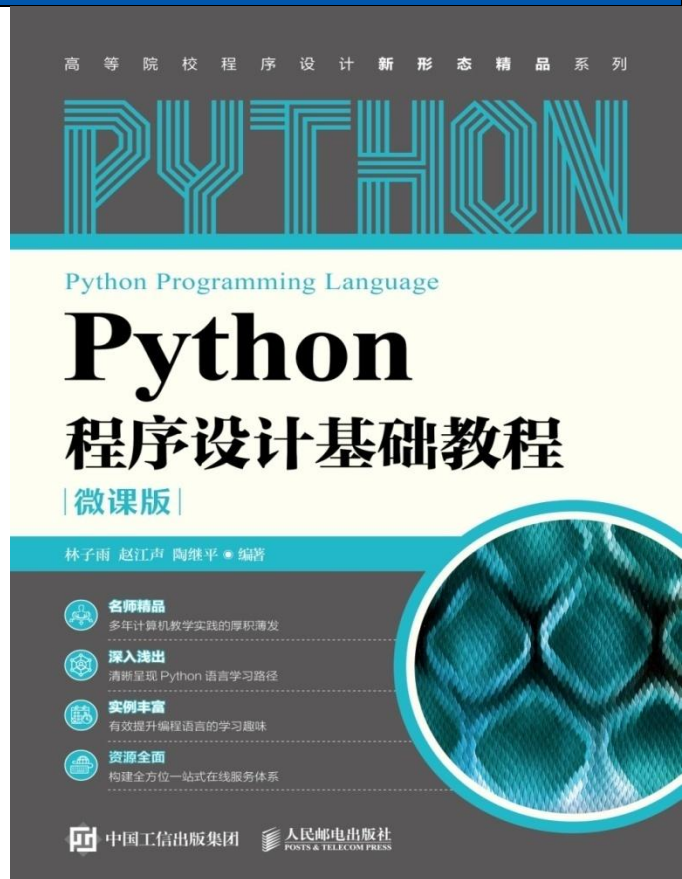
### 4.3.4 字典推导式

本PPT是如下教材的配套讲义：

《Python程序设计基础教程（微课版）》

厦门大学 林子雨,赵江声,陶继平 编著，人民邮电出版社

《Python程序设计基础教程（微课版）》教材官方网站：  
<http://dblab.xmu.edu.cn/post/python>





## 4.3.1字典的创建与删除

字典用大括号`{}`标识。在使用大括号语法创建字典时，大括号中应包含多个“键值对”，键与值之间用英文冒号隔开，多个键值对之间用英文逗号隔开。具体实例如下：

```
>>> grade = {"语文":67, "数学":91, "英语":78} #键是字符串
>>> grade
{'语文': 67, '数学': 91, '英语': 78}
>>> empty_dict = {} #创建一个空字典
>>> empty_dict
{}
>>> dict1 = {(1,2):"male",(1,3):"female"} #键是元组
>>> dict1
{(1, 2): 'male', (1, 3): 'female'}
```

需要指出的是，元组可以作为字典的键，但列表不能作为字典的键，因为字典要求键必须是不可变类型，但列表是可变类型，因此列表不能作为字典的键。



## 4.3.1字典的创建与删除

此外，Python还提供了内置函数dict()来创建字典，实例如下：

```
>>> books = [('hadoop', 132), ('spark', 563), ('flink', 211)]
>>> dict1 = dict(books)
>>> dict1
{'hadoop': 132, 'spark': 563, 'flink': 211}
>>> scores = [['计算机', 85], ['大数据', 88], ['Spark编程', 89]]
>>> dict2 = dict(scores)
>>> dict2
{'计算机': 85, '大数据': 88, 'Spark编程': 89}
>>> dict3 = dict(curriculum='计算机',grade=87) #通过指定参数创建字典
>>> dict3
{'curriculum': '计算机', 'grade': 87}
>>> keys = ["语文","数学","英语"]
>>> values = [67,91,78]
>>> dict4 = dict(zip(keys,values))
>>> dict4
{'语文': 67, '数学': 91, '英语': 78}
>>> dict5 = dict() #创建空字典
>>> dict5
{}

```





## 4.3.1字典的创建与删除

上面代码中，`zip()`函数用于将可迭代的对象作为参数，将对象中对应的元素打包成一个个元组，然后返回由这些元组组成的列表，例如：

```
>>> x = [1,2,3]
>>> y = ["a","b","c"]
>>> zipped = zip(x,y)
>>> zipped
<zip object at 0x0000000002CC9D40>
>>> list(zipped)
[(1, 'a'), (2, 'b'), (3, 'c')]
```



## 4.3.1字典的创建与删除

对于不再需要的字典，可以使用`del`命令删除，实例如下：

```
>>> grade = {"语文":67, "数学":91, "英语":78}  
>>> del grade
```

还可以使用字典对象的`clear()`方法清空字典中的所有元素，让字典变成一个空字典，实例如下：

```
>>> grade = {"语文":67, "数学":91, "英语":78}  
>>> grade.clear()  
>>> grade  
{}
```



## 4.3.2 访问字典

字典包含多个“键值对”，而键是字典的关键数据，因此对字典的操作都是基于键的，主要操作如下：

- 通过键访问值；
- 通过键添加键值对；
- 通过键删除键值对；
- 通过键修改键值对；
- 通过键判断指定键值对是否存在。



## 4.3.2 访问字典

与列表和元组一样，对于字典而言，通过键访问值时使用的也是方括号语法，只是此时在方括号中放的是键，而不是列表或元组中的索引，若指定的键不存在，则会抛出异常，实例如下：

```
>>> grade = {"语文":67, "数学":91, "英语":78}
```

```
>>> grade["语文"]
```

```
67
```

```
>>> grade["计算机"]
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#9>", line 1, in <module>
```

```
    grade["计算机"]
```

```
KeyError: '计算机'
```



## 4.3.2 访问字典

Python中推荐的方法是使用字典对象的`get()`方法获取指定键的值，其语法格式如下：

```
dictname.get(key[,default])
```

其中，`dictname`表示字典对象，`key`表示指定的键，`default`是可选项，用于当指定的键不存在时返回一个默认值，如果省略，则返回`None`，具体实例如下：

```
>>> grade = {"语文":67, "数学":91, "英语":78}
```

```
>>> grade.get("数学")
```

```
91
```

```
>>> grade.get("英语","不存在该课程")
```

```
78
```

```
>>> grade.get("计算机","不存在该课程")
```

```
'不存在该课程'
```

```
>>> grade.get("计算机")
```

```
>>> #执行结果返回None，屏幕上不可见
```



## 4.3.2 访问字典

另外，可以使用字典对象的`items()`方法获取“键值对”列表，使用字典对象的`keys()`方法获取“键”列表，使用字典对象的`values()`方法获取“值”列表，具体实例如下：

```
>>> grade = {"语文":67, "数学":91, "英语":78}
>>> items = grade.items()
>>> type(items)
<class 'dict_items'>
>>> items
dict_items([('语文', 67), ('数学', 91), ('英语', 78)])
>>> keys = grade.keys()
>>> type(keys)
<class 'dict_keys'>
>>> keys
dict_keys(['语文', '数学', '英语'])
>>> values = grade.values()
>>> type(values)
<class 'dict_values'>
>>> values
dict_values([67, 91, 78])
```



## 4.3.2 访问字典

可以看出，`items()`、`keys()`、`values()`三个方法依次返回 `dict_items`、`dict_keys` 和 `dict_values` 对象，Python不希望用户直接操作这几个方法，但可通过`list()`函数把它们转换成列表，实例如下：

```
>>> grade = {"语文":67, "数学":91, "英语":78}
```

```
>>> items = grade.items()
```

```
>>> list(items)
```

```
[('语文', 67), ('数学', 91), ('英语', 78)]
```

```
>>> keys = grade.keys()
```

```
>>> list(keys)
```

```
['语文', '数学', '英语']
```

```
>>> values = grade.values()
```

```
>>> list(values)
```

```
[67, 91, 78]
```



## 4.3.2 访问字典

还可以通过for循环对items()方法返回的结果进行遍历，实例如下：

```
>>> grade = {"语文":67, "数学":91, "英语":78}
```

```
>>> for item in grade.items():
```

```
    print(item)
```

```
('语文', 67)
```

```
('数学', 91)
```

```
('英语', 78)
```

```
>>> for key,value in grade.items():
```

```
    print(key,value)
```

```
语文 67
```

```
数学 91
```

```
英语 78
```





## 4.3.2 访问字典

此外，Python还提供了`pop()`方法用于获取指定键对应的值，并删除这个键值对，实例如下：

```
>>> grade = {"语文":67, "数学":91, "英语":78}
```

```
>>> grade.pop("英语")
```

```
78
```

```
>>> grade
```

```
{'语文': 67, '数学': 91}
```



## 4.3.3 添加、修改和删除字典元素

字典是可变序列，因此，可以对字典进行元素的添加、修改和删除操作。可以使用如下方式向列表中添加元素：

```
dictname[key] = value
```

其中，**dictname**表示字典对象的名称；**key**表示要添加的元素的键，可以是字符串、数字或者元组，但是键必须具有唯一性，并且是不可变的；**value**表示要添加的元素的值。具体实例如下：

```
>>> grade = {"语文":67, "数学":91, "英语":78}
```

```
>>> grade["计算机"] = 93
```

```
>>> grade
```

```
{'语文': 67, '数学': 91, '英语': 78, '计算机': 93}
```



## 4.3.3 添加、修改和删除字典元素

当需要修改字典对象某个元素的值时，可以直接为该元素赋予新值，新值会替换原来的旧值，实例如下：

```
>>> grade = {"语文":67, "数学":91, "英语":78}
```

```
>>> grade
```

```
{'语文': 67, '数学': 91, '英语': 78}
```

```
>>> grade["语文"] = 88
```

```
>>> grade
```

```
{'语文': 88, '数学': 91, '英语': 78}
```



## 4.3.3 添加、修改和删除字典元素

当不再需要字典中的某个元素时，可以使用`del`命令将其删除，实例如下：

```
>>> grade = {"语文":67, "数学":91, "英语":78}
```

```
>>> del grade["英语"]
```

```
>>> grade
```

```
{'语文': 67, '数学': 91}
```

另外，还可以使用字典对象的`update()`方法，用一个字典所包含的键值对来更新已有的字典。在执行`update()`方法时，如果被更新的字典中已包含对应的键值对，那么原值会被覆盖；如果被更新的字典中不包含对应的键值对，则该键值对被添加进去。具体实例如下：

```
>>> grade = {"语文":67, "数学":91, "英语":78}
```

```
>>> grade.update({"语文":59, "数学":91, "英语":78, "计算机":98})
```

```
>>> grade
```

```
{'语文': 59, '数学': 91, '英语': 78, '计算机': 98}
```



## 4.3.4 字典推导式

和列表推导式、元组推导式类似，也可以使用字典推导式快速生成一个符合需求的字典。字典推导式的语法格式如下：

`{表达式 for 迭代变量 in 可迭代对象 [if 条件表达式]}`

其中，用[]括起来的部分是可选项，可以省略。可以看到，和其它推导式的语法格式相比，唯一不同在于，字典推导式用的是大括号{}。具体实例如下：

```
>>> word_list = ["hadoop","spark","hdfs"]
>>> word_dict = {key:len(key) for key in word_list}
>>> word_dict
{'hadoop': 6, 'spark': 5, 'hdfs': 4}
```



## 4.3.4 字典推导式

还可以根据列表生成字典，实例如下：

```
>>> name = ["张三", "李四", "王五", "李六"] #名字列表
>>> title = ["教授", "副教授", "讲师", "助教"] #职称列表
>>> dict1 = {i : j for i, j in zip(name, title)} #字典推导式
>>> dict1
{'张三': '教授', '李四': '副教授', '王五': '讲师', '李六': '助教'}
```



## 4.3.4 字典推导式

下面给出一个实例，交换现有字典中各键值对的键和值：

```
>>> olddict = {"hadoop": 6, "spark": 5, "hdfs": 4}
>>> newdict = {v: k for k, v in olddict.items()}
>>> newdict
{6: 'hadoop', 5: 'spark', 4: 'hdfs'}
```

还可以在上面实例的基础上，使用if表达式筛选符合条件的键值对：

```
>>> olddict = {"hadoop": 6, "spark": 5, "hdfs": 4}
>>> newdict = {v: k for k, v in olddict.items() if v>5}
>>> newdict
{6: 'hadoop'}
```



## 4.4 集合

4.4.1 集合的创建与删除

4.4.2 集合元素的添加与删除

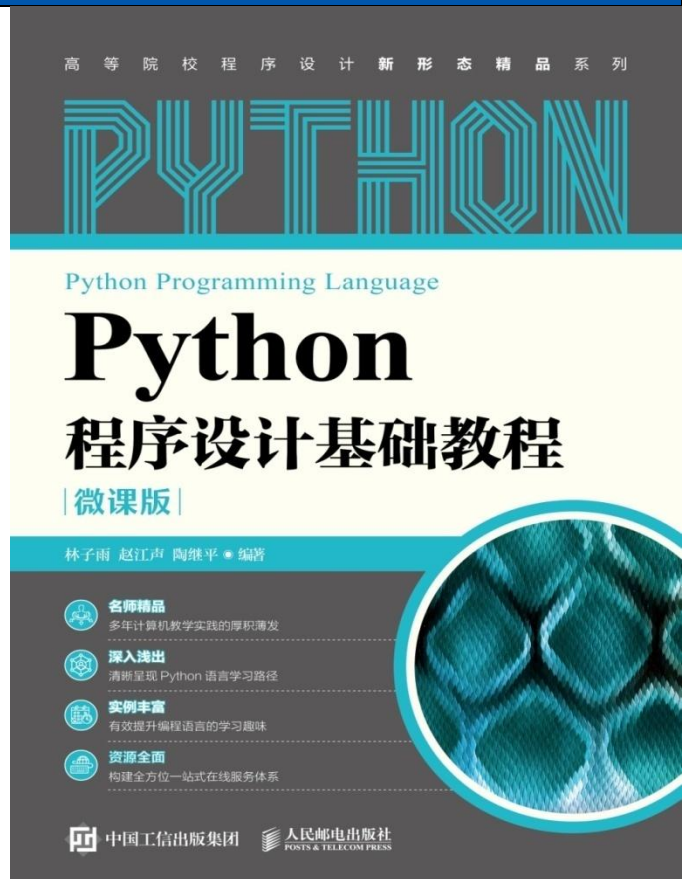
4.4.3 集合的并集、交集与差集操作

本PPT是如下教材的配套讲义：

《Python程序设计基础教程（微课版）》

厦门大学 林子雨,赵江声,陶继平 编著，人民邮电出版社

《Python程序设计基础教程（微课版）》教材官方网站：  
<http://dblab.xmu.edu.cn/post/python>







## 4.4.1 集合的创建与删除

集合（**set**）是一个无序的不重复元素序列。集合中的元素必须是不可变类型。在形式上，集合的所有元素都放在一对大括号“{}”中，两个相邻的元素之间使用逗号分隔。

可以直接使用大括号{}创建集合，实例如下：

```
>>> dayset = {'Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',  
'Saturday', 'Sunday'}  
>>> dayset  
{'Tuesday', 'Monday', 'Wednesday', 'Saturday', 'Thursday', 'Sunday',  
'Friday'}
```

在创建集合时，如果存在重复元素，Python只会自动保留一个，实例如下：

```
>>> numset = {2,5,7,8,5,9}  
>>> numset  
{2, 5, 7, 8, 9}
```



## 4.4.1 集合的创建与删除

与列表推导式类似，集合也支持集合推导式，实例如下：

```
>>> squared = {x**2 for x in [1, 2, 3]}  
>>> squared  
{1, 4, 9}
```



## 4.4.1 集合的创建与删除

也可以使用`set()`函数将列表、元组、`range`对象等其他可迭代对象转换为集合，语法格式如下：

```
setname = set(iteration)
```



## 4.4.1 集合的创建与删除

其中，**setname**表示集合名称，**iteration**表示列表、元组、**range**对象等可迭代对象，或者也可以是字符串，如果是字符串，返回的是包含全部不重复字符的集合。实例如下：

```
>>> set1 = set([1,2,3,4,5])  #从列表转换得到集合
```

```
>>> set1
```

```
{1, 2, 3, 4, 5}
```

```
>>> set2 = set((2,4,6,8,10))  #从元组转换得到集合
```

```
>>> set2
```

```
{2, 4, 6, 8, 10}
```

```
>>> set3 = set(range(1,5))  #从range对象转换得到集合
```

```
>>> set3
```

```
{1, 2, 3, 4}
```

```
>>> set4 = set("自强不息，止于至善")  #从字符串转换得到字符集合
```

```
>>> set4
```

```
{',', '自', '止', '不', '至', '强', '善', '息', '于'}
```



## 4.4.1 集合的创建与删除

需要注意的是，创建一个空集合必须用**set()**而不是**{}**，因为**{}**是用来创建一个空字典。实例如下：

```
>>> empty_set = set()
>>> empty_set
set()
```

当不再使用某个集合时，可以使用**del**命令删除整个集合。实例如下：

```
>>> numset = {1,2,3,4,5}
>>> del numset
```



## 4.4.2 集合元素的添加与删除

可以使用**add()**方法向集合中添加元素，被添加的元素只能是字符串、数字及布尔类型的**True**或者**False**等，不能是列表、元组等可迭代对象。如果被添加的元素已经在集合中存在，则不进行任何操作。实例如下：

```
>>> bookset = {"hadoop","spark"}
```

```
>>> bookset
```

```
{'spark', 'hadoop'}
```

```
>>> bookset.add("flink")
```

```
>>> bookset
```

```
{'flink', 'spark', 'hadoop'}
```

```
>>> bookset.add("spark")
```

```
>>> bookset
```

```
{'flink', 'spark', 'hadoop'}
```



## 4.4.2 集合元素的添加与删除

可以使用`pop()`、`remove()`方法删除集合中的一个元素，使用`clear()`方法清空集合中的所有元素，实例如下：

```
>>> numset = {1,2,3,4,5}
```

```
>>> numset.pop()
```

```
1
```

```
>>> numset
```

```
{2, 3, 4, 5}
```

```
>>> numset.remove(4)
```

```
>>> numset
```

```
{2, 3, 5}
```

```
>>> numset.clear()
```

```
>>> numset
```

```
set()
```



## 4.4.3 集合的并集、交集与差集操作

集合包括并集、交集、差集等操作。所谓并集是指把两个集合中的元素合并在一起，并且去除重复的元素。所谓交集是指取出两个集合中相同的元素。所谓差集是指，对于集合A和B，集合A中的元素在集合B中有重复时，去掉重复元素后集合A中剩余的元素就是A与B的差集。





## 4.4.3 集合的并集、交集与差集操作

Python集合支持常见的集合操作，包括并集、交集、差集等。具体实例如下：

```
>>> a = set('abc')
>>> b = set('cdef')
>>> a | b #并集
{'e', 'f', 'c', 'b', 'd', 'a'}
>>> a & b #交集
{'c'}
>>> a - b #差集
{'b', 'a'}
>>> a.intersection(b) #交集
{'c'}
>>> a.difference(b) #差集
{'b', 'a'}
```

The background is a solid blue gradient. It features several faint, light-blue silhouettes of people. In the top left, a group of four people is holding hands in a circle. In the top center, a group of seven people is holding hands in a line. On the right side, a person is shown in profile, looking towards the center. In the bottom left, there are two silhouettes of people, one appearing to be looking at the other. The text "Thank You!" is centered in the middle of the image in a white, bold, sans-serif font.

**Thank You!**