第 3 章

◂ Python的内置类型 ▸

和其他程序语言一样，Python 也有各种各样的内置类型，包括数字类型、字符串、列表、字符串、元组、字典等，相对于静态语言（比如说 C++、Java），Python 的数据类型功能更全面和强大。

本章的主要内容是：

- Python 类型分类。
- 每种类型的功能和用法。
- 演练各种类型。

3.1 Python 的类型分类

Python 不同于其他语言的一个重要概念是：Python 中的一切均为对象，虽然很多面向对象语言也有这样的概念，但是 Python 面向对象的原理和其他语言不同，主要有两点：

- 所有数值都封装到特定的对象中，Python 不存在像 C 中的 int 这样的简单类型。
- 所有东西都是对象，包括代码本身也被封装到对象中。

Python 的解释器内建数个大类，共二十几种数据类型，一些类别包含最常见的对象类型，如数值、序列等，其他类型则较少使用。后面几节将详细描述这些最常用的类型。表 3-1 列出 Python 内建的常见类型。

表 3-1　Python 内置类型

分类	类型名称	描述
None	NoneType	空对象
数值	IntType	整数
	FloatType	浮点数
	ComplexType	复数
序列	StringType	字符串
	UnicodeType	Unicode 字符串
	ListType	列表
	TupleType	元组
	RangeType	range()函数返回的对象
	BufferType	buffer()函数返回的对象

（续表）

分类	类型名称	描述
映射	DictType	字典
集合	Set	集合类型
可调用类型	BuiltinFunctionType	内建函数
	BuiltinMethodType	内建方法
	ClassType	类
	FunctionType	用户定义函数
	InstanceType	类实例
	ModuleType	模块
类	ClassType	类定义
类实例	InstanceType	类实例
文件	FileType	文件对象
内部类型	CodeType	字节编译码
	FrameType	执行框架
	TracebackType	异常的堆栈跟踪
	SliceType	由扩展切片操作产生
	EllipsisType	在扩展切片中使用

在表 3.1 所罗列的类型中，None 和数值类型构造和使用较为简单，为简单类型。内部类型在解释器调用的时候使用，程序员一般较少使用。本章主要讨论的是简单类型以及序列类型、映射和集合类型。

3.2 简单类型

Python 有 6 个不同的简单类型，分别是：

- 布尔类型（bool 类型）：用于逻辑运算和比较的类型。
- 整数类型（int 类型）：类似于其他计算机语言的 int 类型。
- 浮点类型（float 类型）：用来表示小数的类型。
- 复数类型（complex 类型）：用来表示复数的类型，包括实数和虚数两部分。
- None 类型：空类型，用来表示空或者无返回值。

3.2.1 布尔类型

Python 中最简单的内置类型是 bool 类型，该类型包括的对象仅可能为 True 或 False。这个类型主要用于布尔表达式。Python 提供一整套布尔比较和逻辑运算。

布尔运算主要有：

- 小于，比如 i<100。

- 小于等于，比如 i<=100。
- 大于，比如 i>100。
- 大于等于，比如 i>=100。
- 相等，比如 i==100。
- 不等于，比如 i!=100。

逻辑运算符号包括：

- 逻辑非，比如 not b。
- 逻辑与，比如（b<100）and（b>50）。
- 逻辑或，比如（b<500）or（b>100）。

逻辑运算符的优先级低于单独的比较运算符，所以在运用的时候，需要用括号来特别说明运算的优先级。

实际上，在 Python 中，不只可以用布尔类型来表示真和假，也可以用来其他类型表示真和假，还可以参加逻辑运算。例子 3.1 是 Python 真假判断的例子。

例子 3.1　Python 的真假判断

```
>>> a=0.0
>>> print(not a)
True
>>> a=1.0
>>> print(not a)
False
>>> a=0
>>> print(not a)
True
>>> a=1
>>> print(not a)
False
>>> a=[]
>>> print(not a)
True
>>> a=[1]
>>> print(not a)
False
>>> a=0.0
>>> print(True and a)
0.0
>>> print(True or a)
True
```

```
>>> a=0
>>> print(False and a)
False
>>> a=1
>>> print(False and a)
False
>>> print(True and a)
1
```

在 Python 中，除了 False 表示假以外，[]、{}、()、””、0、0.0、None 这些均表示假。而其他均为真。例子 3.2 演示了用 0 和 0.0 等来和布尔类型来进行布尔运算的情况。在 Python 中，任何类型都可以类似于布尔类型进行布尔运算，只不过需要牢记在 Python 中，不只是 0 为假，还包括[]、{}、()等各种等同于假的情况。

在 C/C++等程序语言里 0 为假、其他为真。使用 Python 时需要在这一点上注意一下。

对于布尔运算表达式，Python 并不是将整个表达式逐步执行，例如类似于 expr1 and expr2 的运算，若 expr1 为假则直接返回 expr1，而不会去处理 expr2，也就是说如果 expr1 为 0、0.0、空列表等各种为假的情况，该表达式直接返回 expr1，而不会去处理 expr2，只有当 expr1 为真的时候，才会去处理 expr2，返回 expr2 的值。对于 or 运算（expr1 or expr2）来说，如果 expr1 为真，那么总是直接返回 expr1，而不会去处理 expr2，只有当 expr1 为假的时候，才会去处理 expr2，并返回 expr2 的值。

对于一个组合的表达式：condition and expr1 or expr2，会有怎样的结果呢？该表达式的结果有如下几种情况：

- condition 为真，expr1 为真，expr2 不做处理，直接返回 expr1。
- condition 为假，expr1 不做处理，直接返回 expr2。
- condition 为真，expr1 为假，直接返回 expr2。

从上面的分析可以看出，该组合表达式的返回值有 3 种可能，取决于 condition 和 expr1 这两个表达式的真假组合，只有当两个表示式全为真的时候，才返回 expr1，其他情况返回 expr2。

需要注意的是，该表达式还作为 Python 一个惯用的赋值语句，该惯用赋值语句假定 expr1 为真，那么 condition and expr1 or expr2 表达式的结果可以简化为：

- condition 为真，返回 expr1。
- condition 为假，返回 expr2。

在这种情况下，使用该语句就可以替换一些用 if 来判断赋值的简单语句。例子 3.2 给出了组合表达式的使用演示。

例子 3.2　组合表达式的使用

```
01  >>> a=3
02  >>> if a==3:
03  ...     str2="it is 3"
04  ... else:
05  ...     str2="it is 2"
06  ...
07  >>> print(str2)
08  it is 3
09  >>> str1= a==3 and "it is 3" or "it is 2"
10  >>> print(str1)
11  it is 3
12  >>>
```

上面代码第 2~6 行是一个 if…else 判断语句，用来判断如果 a 等于 3 就为 str2 赋值“it is 3”，否则赋为“it is 2”。在第 9 行，使用组合表达式替代 if...else 语句来达到同样的效果，而代码简洁很多。

这种用法完全是基于假定 expr1 为真的情况下使用的，所以当 expr1 为假的时候，是不能按照这种方法使用的。例子 3.3 给出了组合表达式惯用法不适用的情况。

例子 3.3　组合表达式惯用法的意外情况

```
01  >>> a=3
02  >>> if a==3:
03  ...     value=0
04  ... else:
05  ...     value=1
06  ...
07  >>> print(value)
08  0
09  >>> value= a==3 and 0 or 1
10  >>> print(value)
11  1
```

在例子 3.3 中，第 2~6 行是一个 if…else 语句，用来说明当 a 等于 3 的时候 value=0，否则 value=1，但是在第 9 行代码中，我们使用该表达式惯用法却得到和 if...else 语法不相同的结果，这是因为该惯用法的假设 expr2 为真的条件不存在了。

在 C 语言中，cond? true_expr: false_expr 的用法是当 cond 为真的时候直接返回 true_expr，否则返回 false_expr。Python 的 cond and true_expr or false_expr 的用法在形式上和该用法很相似，但是特别要注意，只有当 true_expr 为真的时候 Python 的该用法才和 C/C++中的语义等价。

3.2.2　整数类型

Python 用来存储整数的类型的就是整数类型。例如，Python 的数值类型都支持加、减、乘、除、幂、求模等各种运算，与 Python 2.X 略有不同的是 Python 3.X 的 int 类型合并了 Python 2.X 的 int 类型和 long 类型：

- 加：a=1+3
- 减：a=1-3
- 乘：a=1*3
- 除：a=1/3
- 幂：a=1**3
- 求模：a=1%3

上述数值运算中幂运算优先级最高，乘除和求模同一优先级，加减最后。需要调整优先级时可以通过加括号来实现，例如：

```
A=3+4*5**2/6
```

如果先要计算加，然后计算乘，可以使用如下方法：

```
A=((3+4)*5)**2/6
```

程序员一般在十进制系统中工作。有时其他进制的系统也相当有用，比如计算机就是基于二进制的。Python 可以提供对八进制和十六进制数字的支持。要通知 Python 应该按八进制数字常量处理数字，只需将一个零加上一个 o 附加在数字前面。将一个零加上一个 x 附加在数字的前面是告诉 Python 按十六进制数值常量处理数字，例如：

```
>>> print(13)
13
>>> print(0o13)
11
>>> print(0x13)
19
```

3.2.3　浮点数类型

所谓浮点数类型，就是小数。在 Python 中，带圆点符号的数值都会被认为是浮点数类型。例如：

```
>>> a=1.
>>> type(a)
<class 'float'>
```

3.2.4　复数类型

复数类型，在其他计算机语言中很少见。复数是数学里的一个概念，在科学计算中，有

着重要的用处，在形式上有实数和虚数两部分，在 Python 中由 float 类型表示，虚数是-1 的平方根的倍数，用字母 j 表示，例如：

```
>>> c=2+3j
>>> print(c)
(2+3j)
```

3.2.5 None 类型

None 类型是 Python 特殊的常量，表示空，等同于 C/C++中的 null，大部分时候用来判断函数或者对象方法的返回结果，无返回结果即为 None。

3.3 简单类型的运算

除了支持四则运算和求模、求幂等运算外，Python 还支持求补、左移、右移、按位和、按位异或、按位或等各种运算。例子 3.4 列举了各种位运算。

例子 3.4 Python 的位运算

```
01  >>> a=0x13
02  >>> ~a
03  -20
04  >>> a>>2
05  4
06  >>> a<<3
07  152
08  >>> a^3
09  16
10  >>> a&0x01
11  1
12  >>> a^0x01
13  18
14  >>> a|0x01
15  19
```

上面的代码演示了各种位运算。~符号代表求补运算。求补运算不考虑符号位，对它的原码各位取反，并在末位加 1 即可。>>符号代表向右位移，<<代表向左位移，^符号代表按位异或（两个整数根据二进制位进行异或操作），&符号代表求和操作（两个整数根据二进制位进行求和操作），| 符号代表按位或操作。

在 Python 中，有关数值的运算需要注意两点：

- 运算的优先级，幂运算最高，乘除位运算其次，加减最后。

- 在 Python 中，若不同数值类型在单个表达式中混合出现时，则会根据需要将表达式中的所有操作数转换为最复杂的操作数的类型，作为返回值的类型。复杂度的顺序是 int、float、complex。下面给出一个简单的示例:

```
>>> 1//3
0
>>> 1.0/3
0.3333333333333333
>>> 1.0//3
0.0
>>> 1%3
1
>>> 1.0%3
1.0
>>>
```

3.4 常量类型

简单类型均为常量类型，也就是说这些类型对象一旦被创建，其值就不能被更改。我们使用等于符号（=）进行新的赋值操作时，实际上，Python 解析器是创建了新的简单类型，并将该变量名和新创建的对象关联起来。使用 Python 的内置 id()函数（用来查看对象在内存中的编号的函数），就可以清楚地看到这一点。例子 3.5 将演示这些操作。

例子 3.5　整型类型的创建和更改

```
01  >>> a=3
02  >>> id(a)
03  1945071136
04  >>> a=4
05  >>> id(a)
06  1945071168
07  >>> b=3
08  >>> id(b)
09  1945071136
10  >>>a=b+2
11  >>>id(a)
12  1945071200
```

在例子 3.5 中，a 刚开始等于 3，它的 id 结果为 1945071136，更改 a=4 时，它的 id 结果为 1945071168，这表明 a 所指向的对象发生了变化，因为 a=4 这个操作并没有让原来的 3 更改成 4，而是新创建一个数值为 4 的对象并让 a 指向这个新对象。在例子 3.5 的第 10~12 行，b+2 的

结果为 5，a 又指向了这个新对象，而原来的数值对象 4 仍然存在内存中，并没有改变。

3.5 序列类型

在实际编写 Python 程序的时候，通常要处理复杂的逻辑，从而带来复杂的数据结构。如果使用简单类型来表达复杂的数据，就会存在大量的简单数据对象，存放和管理它们将成为大问题。容器类型就是用来存放和管理各种对象的类型，使用容器类型，就可以根据程序的需求把需要处理的复杂数据放到容器中。容器提供了一系列的方法，可以用来访问和管理这些数据。

容器类型可以分为两种：

- 序列容器，一般也被称为顺序容器，就是说该容器是将存放的数据按顺序放置在内存区中，如果一个新元素被插入或者已存元素被删除，其他在同一个内存块的元素就必须向上或者向下移动来为新元素提供空间，或者填充原来被删除的元素所占的空间，这种移动影响了效率。
- 关联容器，根据每个节点来存放元素，容器元素的插入或删除只影响指向节点的指向，而不是节点自己的内容，所以当有数据插入或删除时，元素值不需要移动。

在 Python 中，序列容器类型主要有 6 种，即 string、unicode、list（列表）、tuple（元组）、buffer、range；关联容器类型主要是字典类型和集合类型。

3.6 列表类型

list 类似于 C 中的数组、C++中的 vector，是用来顺序存储数据的容器，比如说，一周七天，可以表示为：

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

第 2 章曾简要介绍过 list，本节会详细介绍它的功能和方法。

3.6.1 创建 list

创建 list 的方法很简单，使用[]符号，中间的元素用逗号隔开。例如：

```
    A=[1,2,3]
```

3.6.2　list 的元素访问

可以通过下标来访问 list，下标从 0 开始，不同于其他语言的是增加了负下标的访问，-1 代表最后一个元素，-2 代表倒数第二个元素，以此类推。下面给出一个简单的例子：

```
>>> a=[1,2,3]
>>> a[0]
1
>>> a[-2]
2
```

下标不只可以访问 list 的单个元素，也能通过切片操作获得一个 list 的子 list，可以通过下标来指定范围，一个指定开始位置，一个指定结束位置，中间加冒号分隔。开始位置默认为 0，结束位置默认为-1。需要特别注意的是：指定的开始位置包括开始元素本身，而结束位置不包括结束位置本身。例子 3.6 使用切片操作获得列表的子列表。

例子 3.6　子列表的访问

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

3.6.3　列表运算

list 容器支持一系列的运算，包括加法、乘法、大小比较等运算。要查看 list 支持哪些运算，可以直接在 IDEL 里面输入 help(list)。help 是 Python 的自省功能之一，可以通过 help() 函数查看 Python 对象的一些帮助信息。例子 3.7 通过 help 自省功能，获得了列表的帮助信息。

例子 3.7　通过 help 查看 list 支持的运算

```
>>> help(list)
Help on class list in module __builtin__:

class list(object)
 |  list() -> new list
 |  list(iterable) -> new list initialized from sequence's items
 |
```

```
 |  Methods defined here:
 |
 |  __add__(...)
 |      x.__add__(y) <==> x+y
 |
 |  __contains__(...)
 |      x.__contains__(y) <==> y in x
 |
 |  __delitem__(...)
 |      x.__delitem__(y) <==> del x[y]
 |
 |  __delslice__(...)
 |      x.__delslice__(i, j) <==> del x[i:j]
 |
 |      Use of negative indices is not supported.
 |
 |  __eq__(...)
 |      x.__eq__(y) <==> x==y
 |
 |  __ge__(...)
 |      x.__ge__(y) <==> x>=y
 |
 |  __getattribute__(...)
 |      x.__getattribute__('name') <==> x.name
 |
 |  __getitem__(...)
 |      x.__getitem__(y) <==> x[y]
#省略部分结果
```

在 Help()函数打印出来的 list 对象信息中，类似于__xxx__这些函数都是 list 对象内建的一些类的特殊方法，其中大部分函数为 list 的运算操作符函数（类似于 C++中的 operation）。凡是支持特殊函数的对象，也就都支持对应的运算。从上面的帮助信息可以发现，list 主要支持以下运算操作：

- 实现了__add__函数，list 支持加法运算。
- 实现了__contains__函数，list 支持 in 操作。
- 实现了__eq__函数，list 支持==判断。
- 实现了__ge__函数，list 支持>=判断。
- 实现了__gt__函数，list 支持>判断。
- 实现了__iadd__函数，list 支持+=操作。
- 实现了__imul__函数，list 支持*=操作。
- 实现了__le__函数，list 支持<=操作。

- 实现了__lt__函数，list 支持<操作。
- 实现了__mul__函数，list 支持*操作。
- 实现了__ne__函数，list 支持!=操作。
- 实现__rmul__函数，list 支持被乘操作。

还有一些其他的方法，比如__getitem__，则是通过下标访问该对象的实现方法，在前面已经讨论了操作方法，这里不再重复叙述。

上面是从 help 的帮助信息中获得的有关 list 支持的运算信息，可以在 IDEL 里根据这些帮助信息对列表运算进行尝试。（例子 3.8 是自省出来的列表操作运算信息。）

例子 3.8　list 的操作运算

```
>>> a=[1,2,3]
>>> a+4
Traceback (most recent call last):
  File "<stdio>", line 1, in ?
TypeError: can only concatenate list (not "int") to list
>>> a=[1,2,3]
>>> a+[4,5,6]
[1, 2, 3, 4, 5, 6]
>>> a+3
Traceback (most recent call last):
  File "<stdio>", line 1, in ?
TypeError: can only concatenate list (not "int") to list
>>> 4 in a
False
>>> 1 in a
True
>>> a==[4,5,6]
False
>>> a==[1,2,3]
True
>>> a>[4,5,6]
False
>>> a>[1,2,1]
True
>>> a>=[1,2,1]
True
>>> a+=[4,5]
>>> print(a)
[1, 2, 3, 4, 5]
>>> a*2
```

```
31  [1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
32  >>> print(a)
33  [1, 2, 3, 4, 5]
34  >>> a*=2
35  >>> print(a)
36  [1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
37  >>> a!=[3,4,5]
38  True
39  >>> 2*a
40  [1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5]
41  >>> a*'2'
42  Traceback (most recent call last):
43    File "<stdio>", line 1, in ?
44  TypeError: can't multiply sequence by non-int
45  >>>
```

例子 3.8 调用了 Python 的各种运算操作。需要注意的是，各操作运算的对象类型有限制。例如，第 2~12 行，调用 list 的加法加一个整数就触发了以外，list 的加法对象也只能是 list。第 13~16 行，演示 list 的 in 操作。在 Python 的语法中，关键词 in 表示存在的意思。3 in a 表示判断 3 是否在 a 中。in 的反用法是 not in。第 30~35 行，演示 list 的乘法。list 的乘法是复制 list 中元素的快捷方法，需要复制几份就乘以几。

list 乘法的对象只能是整数类型。

3.6.4 列表的方法

除了运算操作外，列表还支持一些其他的操作方法，同样可以使用 help(list)来获得那些方法的信息。方法名称前面不带__符号的方法就是 list 的公用方法。例子 3.9 是摘录了 list 自省信息中有关列表的公用方法。

例子 3.9 查看 list 的公用方法

```
>>> help(list)
Help on class list in module __builtin__:

class list(object)
 |  list() -> new list
 |  list(iterable) -> new list initialized from sequence's items
 |
 |  Methods defined here:
 |  append(...)
 |      L.append(object) -- append object to end
```

```
     |
     |  count(...)
     |      L.count(value) -> integer -- return number of occurrences of value
     |
     |  extend(...)
     |      L.extend(iterable) -- extend list by appending elements from the
iterable
     |
     |  index(...)
     |      L.index(value, [start, [stop]]) -> integer -- return first index of
value
     |
     |  insert(...)
     |      L.insert(index, object) -- insert object before index
     |
     |  pop(...)
     |      L.pop([index]) -> item -- remove and return item at index (default
last)
     |
     |  remove(...)
     |      L.remove(value) -- remove first occurrence of value
     |
     |  reverse(...)
     |      L.reverse() -- reverse *IN PLACE*
     |
     |  sort(...)
     |      L.sort(cmp=None, key=None, reverse=False) -- stable sort *IN PLACE*;
     |      cmp(x, y) -> -1, 0, 1
     |
     |  ----------------------------------------------------------------------
     |  Data and other attributes defined here:
     |
     |  __new__ = <built-in method __new__ of type object>
     |      T.__new__(S, ...) -> a new object with type S, a subtype of T
```

从中我们可以看到，list 有 9 种不同用途的公用方法：

- append()方法，在列表后增加对象。
- count()方法，统计列表元素的个数。
- extend()方法，将一个序列对象转换成列表，并增加到该列表后面。
- index()方法，返回查找值的第一个下标，如果找不到查找值，则抛出错误。
- insert()方法，插入对象到指定的下标后面。
- pop()方法，弹出列表指定下标的元素。不指定下标时，弹出最后一个元素。

- remove()方法，删除列表指定的值，有多个指定的值在列表中时删除第一个。
- reverse()方法，将列表元素顺序倒置。
- sort()方法，对列表进行排序，排序的方法可以在 sort 的参数中指定，默认从小到大排序。

运用上面的公用方法，能够方便对 list 做各种操作，并且能够用 list 模拟其他的数据类型，比如堆栈（stack）、队列（queue），如例子 3.10 所示。

例子 3.10　用 list 模拟堆栈和队列

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
>>> queue = ["Eric", "John", "Michael"]
>>> queue.append("Terry")
>>> queue.append("Graham")
>>> queue.pop(0)
'Eric'
>>> queue.pop(0)
'John'
>>> queue
['Michael', 'Terry', 'Graham']
```

第 1~15 行模拟的是堆栈的 push 和 pop 操作。堆栈的特点是“先进后出”，使用 list 的 pop()方法，将列表的最后一个元素弹出，就可以模拟堆栈的操作。第 16~24 行模拟的是队列的操作。队列的特点是“先进先出”，通过 pop()方法弹出列表第一个元素（下标为 0），就可以模拟队列的“先进先出”。

3.6.5　列表的内置函数（range、filter、map）

列表除了可以使用运算操作、自带的公用函数外，还可以使用 Python 内置的 range()、filter()、map()、reduce 函数。这 4 个函数有着不同的用途。

（1）range()函数的作用是生成一个整型序列，有 3 个参数：开始数值（start）、结束数值（stop）、累进大小（step），开始数值默认为 0，累进大小默认为 1，例如：

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1,10,2))
[1, 3, 5, 7, 9]
>>>
```

（2）filter 过滤函数的作用是对列表进行过滤，只保留满足 filter()函数指定要求的元素，例如：

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
>>> list(filter(f, range(2, 25)))
[5, 7, 11, 13, 17, 19, 23]
```

（3）map 映射函数的作用是对列表的每一个元素映射到 map()函数指定的操作，例如：

```
>>> def cube(x): return x*x*x
>>> list(map(cube, range(1, 11)))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

map 同时可以处理多个列表，但是需要注意的是 map 处理多个列表时，这些列表的元素应该相同，否则就会抛出异常，例如：

```
>>> def add(a,b):
...     return a+b
...
>>> list(map(add,[1,2,3],[4,5,6]))
[5, 7, 9]
>>> list(map(ad,[1,2,3],[4,5]))
Traceback (most recent call last):
  File "<stdio>", line 1, in ?
NameError: name 'ad' is not defined
```

（4）reduce 函数是 reduce(function, sequence, starting_value)，它对 sequence 中的 item 顺序迭代调用 function，如果有 starting_value，还可以作为初始值调用，例如可以用来对 list 求和：

```
>>> def add(x,y): return x + y
>>> reduce(add, range(1, 11))
55
>>> reduce(add, range(1, 11), 20)
75
```

3.6.6 列表推导式

对于使用过滤和映射函数去生成特定要求的列表，Python 提供了一个更简洁的方法——列表推导式（List Comprehension）来完成这个工作，一个 List Comprehension 通常由一个表达式以及一个或者多个 for 语句和 if 语句组成，语法形式类似于[<expr1> for k in L if <expr2>]，for k in L 是对 L 列表的循环，if expr2 是用 expr2 对循环的元素 k 做过滤处理，expr1 则是返回表达式。例子 3.11 列出了列表推导式的具体用法。

例子 3.11 List Comprehension 的用法

```
>>> a=[1,2,3,4,5]
>>> [k*5 for k in a]
[5, 10, 15, 20, 25]
>>> [k*5 for k in a if a!=3]
[5, 10, 15, 20, 25]
>>> [k*5 for k in a if k!=3]
[5, 10, 20, 25]
>>> b=['A','b','cd','e']
>>> [k.upper()   for k in b]
['A', 'B', 'CD', 'E']
```

这种方法将过滤和映射操作合二为一，代码简洁很多，可读性更强。

3.7 元组类型

元组（tuple）类型通过一对括号“()”来表示，元组是常量的 list，使用 help(tuple)，可以获得 tuple 的自省信息。例子 3.13 就是使用该方法获得 tuple 的自省信息。

例子 3.12 tuple 的 help 信息

```
>>>help(tuple)
Help on class tuple in module __builtin__:

class tuple(object)
 |  tuple() -> an empty tuple
 |  tuple(iterable) -> tuple initialized from iterable's items
 |  If the argument is a tuple, the return value is the same object.
 |  Methods defined here:
 |  __add__(...)
 |      x.__add__(y) <==> x+y
 |  __contains__(...)
 |      x.__contains__(y) <==> y in x
 |  __eq__(...)
```

```
|      x.__eq__(y) <==> x==y
|  __ge__(...)
|      x.__ge__(y) <==> x>=y
|  __getattribute__(...)
|      x.__getattribute__('name') <==> x.name
|  __getitem__(...)
|      x.__getitem__(y) <==> x[y]
|  __getnewargs__(...)
|  __getslice__(...)
|      x.__getslice__(i, j) <==> x[i:j]
|      Use of negative indices is not supported.
|  __gt__(...)
|      x.__gt__(y) <==> x>y
|  __hash__(...)
|      x.__hash__() <==> hash(x)
|  __iter__(...)
|      x.__iter__() <==> iter(x)
|  __le__(...)
|      x.__le__(y) <==> x<=y
|  __len__(...)
|      x.__len__() <==> len(x)
|  __lt__(...)
|      x.__lt__(y) <==> x<y
|  __mul__(...)
|      x.__mul__(n) <==> x*n
|  __ne__(...)
|      x.__ne__(y) <==> x!=y
|  __repr__(...)
|      x.__repr__() <==> repr(x)
|  __rmul__(...)
|      x.__rmul__(n) <==> n*x
|  Data and other attributes defined here:
|  __new__ = <built-in method __new__ of type object>
|      T.__new__(S, ...) -> a new object with type S, a subtype of T
```

通过例子 3.12 的 help 信息，可以看出 tuple 和 list 的区别：

- tuple 和 list 一样，支持下标和切片操作（都实现__getitem__和__getslice__）。
- tuple 与 list 不同，不支持通过下标和切片操作更改元素和子列表（tuple 没有实现__setitem__和__setslice__）。
- tuple 支持和 list 一样的比较运算和加乘运算，但是不支持+=、*=操作（因为 tuple 为常量类型，这两个操作都是更改自身的操作）。
- tuple 不支持 list 的 9 种公用方法。

元组是常量类型的 list，和列表的区别在于，元组对 list 的那些更改自身元素的操作和方法都不支持，元组一旦在内存中创建，就不可被更改，这是它和 list 最大的区别，而其他使用方法则和列表一样，例如：

```
>>> a=(1,2,3,4,5)
>>> a[0]
1
>>> a[3]
4
>>> b=a+(7,8)
>>> print(b)
(1, 2, 3, 4, 5, 7, 8)
>>> b=a*2
>>> print(b)
(1, 2, 3, 4, 5, 1, 2, 3, 4, 5)
```

3.8 字符串类型

字符串可以看成特殊的元组，可以使用单引号或者双引号来表示字符串，例如：

```
>>> str1='word'
>>> str2="hello"
```

对于特殊字符，Python 使用转义字符反斜杠（\）来表示。表 3-2 列出了各种转义字符。

表 3-2　转义字符

字符	说明
\\	反斜杠
\’	单引号
\”	双引号
\a	响铃
\b	退格
\f	FF
\n	换行
\r	回车
\t	水平制表符
\ooo	ooo 表示八进制字符
\xhh	hh 表示十六进制字符

当需要输入一个很长的字符串时，可以分成多行，用反斜杠来连接。例如：

```
>>> str1="hello "\
```

```
...     "world"\
...     "hi"
```

如果需要输入一个非常长的字符串，可以使用连续的三个双引号，例如：

```
>>> str2="""
...     There is a way to remove an item from a list given its index
instead of its value: the del statement
...     """
```

字符串类型和元组一样是常量类型，支持下标访问、切片、比较运算、加乘法等运算。字符串类型的公用算法比元组多，主要分为以下几类。（使用 help(str)可以查看字符串类型所支持的公用算法。）

1. 大小写转换

大小写转换主要包括首字符大写（capitalize）、全部转换小写（lower）、全部转换大写（upper）、大小写互换（swapcase）、单词首字母大写其他小写（title）等方法。下面是应用的简单例子：

```
>>> "this is test".capitalize()
'This is test'
>>> "this is test".lower()
'this is test'
>>> "this is test".upper()
'THIS IS TEST'
>>> "This is Test".swapcase()
'tHIS IS tEST'
>>>"this is test".title()
'This Is Test'
```

2. 字符串的搜索

字符串搜索包括的方法有 find、index、rfind、rindex、count、replace 等函数。find 是从左向右查找，并返回找到第一个字母的下标。rfind 是从右向左查找。index 和 find 用法类似，不同之处在于，find()方法找不到时返回-1，index 则抛出一个异常。例如：

```
>>> str2="ni ok hello ok why"
>>> str2.find("ok")
3
>>> str2.rfind("ok")
12
>>> str2.find("ok2")
-1
>>> str2.index("ok2")
Traceback (most recent call last):
```

```
  File "<stdio>", line 1, in ?
ValueError: substring not found
```

3. 字符串的替换

字符串替换包括的方法有 replace（替换）、strip（去掉头尾指定的字符）、rstrip（从右边开始）、lstrip（左边）、expandtabs（用空格取代 tab 键）。例如：

```
>>> a="as1234567"
>>> a.strip("as")
'1234567'
>>> a.lstrip("as")
'1234567'
>>> a.rstrip("as")
'as1234567'
>>> a.replace("as","dd")
'dd1234567'
>>> b="ni   bb  ss"
>>> b.expandtabs(1)
'ni bb ss'
```

4. 字符的分隔

split()方法可以根据指定的字符，把一个字符串截断成列表。splitlines()可以将一个字符串，根据换行符截断列表，例如：

```
>>> a="1,2,3,4"
>>> a.split(",")
['1', '2', '3', '4']
>>> b="123\n456\n789"
>>> b.splitlines()
['123', '456', '789']
```

5. 字符判断功能

字符判断功能主要是以下几种：

- startwith（prefix，start[,end]），判断一个字符串是否以 prefix 开头。
- endwith（suffix[,start[,end]]），判断一个字符串是否以 suffix 结尾。
- isalnum()，判断是否由字母和数字组成，至少有一个字母。
- isdight()，判断是否全是数字。
- isalpha()，判断是否全是字母。
- isspace()，判断是否由空格字符组成。
- islower()，判断字符串中的字母是否全是小写。
- isupper()，判断字符串中的字母是否全是大写。

- istitle()，判断字符串是否为首字母大写。

3.9 字典类型

字典是 Python 中关联型的容器类型，字典的创建使用大括号{}的形式，字典中的每一个元素都是一对，每对包括 key 和 value 两部分，中间以冒号隔开。对于 key，需要注意以下两点。

（1）key 的类型只能是常量类型。

key 必须为常量类型（数值，不含有可变类型元素的元组、字符串等），不能用可变类型做 key 值，例如列表作为字典的 key，因为 key 必须保持不变，key 的用途是作为字典的索引值，Python 根据该值去存放数据。

（2）key 的值不能重复。

key 的数值在一个字典中是唯一的，不存在重复的 key 值。

可以通过自省功能来获得有关字典的帮助信息，通过 help(dict)就可以获得字典的详细信息。根据字典的自省信息，可以看到字典的使用细节，包括字典的创建方法、字典所支持的操作运算、字典所支持的公用操作方法等。

3.9.1 字典的创建

创建字典的语法为{key:value,key1:value1 ……}，例如：

```
>>> fruit={1:'apple',2:'orange',3:'banana',4:'tomato'}
```

从 dict 的帮助信息上可以发现，dict 还支持以下创建方法。

（1）dict()创建一个空字典，例如：

```
>>> fruit=dict()
>>> print(fruit)
{}
```

（2）通过一个映射类型的组对生成 dict：

```
>>> a={1:'one',2:'two',3:'three'}
>>> b=dict(a)
>>> print(b)
{1: 'one', 2: 'two', 3: 'three'}
```

（3）通过序列容器生成队列（序列容器的元素必须为两个元素的列表或者元组）：

```
>>> dict([(1,'one'),(2,'two'),(3,'three')])
{1: 'one', 2: 'two', 3: 'three'}
```

（4）通过输入方法参数（参数格式为 name=value）创建字典：

```
>>> dict(one=1,two=2,three=4,four=4)
{'four': 4, 'three': 4, 'two': 2, 'one': 1}
```

3.9.2 字典的操作

在 dict 的帮助信息中，可以查看 dict 所支持的操作方法和运算，主要有以下几类。

（1）通过 key 值作为下标来访问 value 值。

```
>>> a={1:'one',2:'two',3:'three'}
>>> a[2]
'two'
```

（2）各种比较运算（==、!=）。在 Python 3 中 dict 不支持大小比较：

```
>>> a={1:'one',2:'two',3:'three'}
>>> b={1:'one',2:'two',3:'tiree'}
>>> a==b
False
>>> a!=b
True
```

（3）清空字典（clear()方法）：

```
>>> a.clear()
>>> a
{}
```

（4）删除字典某一项（pop()、popitem()方法）：

```
>>> bdict={1:'a',2:'b',3:'c'}
>>> bdict.pop(1)
'a'
>>> bdict.popitem()
(2, 'b')
>>
```

（5）序列访问方法：提供序列访问字典的方法。items()方法返回一个列表，列表中是（key，value）的元组，iteritems、iterkeys、itervalues 返回迭代器对象，keys()方法返回一个以 key 为元素的列表。例如：

```
>>> bdict={1:'a',2:'b',3:'c',4:'d',5:'c'}
>>> bdict.items()
dict_items([(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'c')])
>>> for a in bdict.items():
...     print(a)
```

```
...
(1, 'a')
(2, 'b')
(3, 'c')
(4, 'd')
(5, 'c')
>>> for a in bdict.keys():
...     print(a)
...
1
2
3
4
5
>>> for a in bdict.values():
...     print(a)
...
a
b
c
d
c
>>> bdict.keys()
dict_keys([1, 2, 3, 4, 5])
```

3.10 集合类型

集合类型是在 Python 2.3 版本以后才新增加的，有可变的集合和不可变的集合两种类型。集合类型的作用可以用一句话来概括：无序并唯一地存放容器元素的类型。集合类型里面可以存放各种类型的对象（特点是无序存放并且不能重复存放）。

3.10.1 集合的创建

```
>>> a=set([1,2,3])
>>> b=frozenset([1,2,3])
>>> a,b
({1, 2, 3}, frozenset({1, 2, 3}))
```

set()方法用来创建可变集合。frozenset 用来创建不可变集合。

3.10.2 集合的方法和运算

集合方法的运算主要是并、交、差、补、判断子集。

（1）并是将两个集合的元素合并在一起，可以用 union()方法或者|运算。

（2）交是求两个集合都公有的元素，可以用 intersection()方法或者&运算。

（3）差是求一个集合比另一个集合多或者少的元素，可以用 difference()方法或者减法运算。

（4）补是求两个集合中不为交集的元素，可以用 symmetric_difference()方法或者运用^运算来判断。

（5）判断子集就是判断一个集合是否是另一个集合的子集。可以用 issubset()方法或者<=运算来判断。

集合类型的操作演示如例子 3.13 所示。

例子 3.13 集合类型的操作

```
>>> a=set([1,2,3,4])
>>> b=set([2,3,5,6])
>>> a.difference(b)
{1, 4}
>>> a-b
{1, 4}
>>> a|b
{1, 2, 3, 4, 5, 6}
>>> a&b
{2, 3}
>>> a^b
{1, 4, 5, 6}
>>> a>=b
False
>>>
```

3.11 开始编程：文本统计和比较

本节将使用前面所介绍的各种内建类型实现一个完整的例子：对两个英文文档进行统计，得到两个英文文档使用了多少单词、每个单词的使用频率，并且对两个文档进行比较，返回有差异的行号和内容。

【本节代码参考：C03\PyMerge.py】

3.11.1　需求说明

在日常工作中，经常需要对文本进行统计和比较，特别是在多人协作编写文档和代码的时候，要经常性地进行文本统计和比较，这样才能保证不会错误地覆盖文档和代码，通常会使用 Araxis Merge 软件来进行文本比较，本节将使用 Python 实现一个简单的 Merge 程序。

3.11.2　需求分析

本节要实现的程序取名为 PyMerge。它需要实现两个功能模块：统计和比较。

- 统计功能：主要是统计总词汇数和每个词汇的数目。
- 比较功能：主要是比较两个文本的差异，需要忽略空行和空格的影响，也就是因为多个空行或者空格产生的文本差异不应该列为文本差异。

3.11.3　整体思路

PyMerge 程序主要有两大功能：统计和比较。可以分开分析这两个功能的思路，统计功能的关键是如何存放词汇和词汇数，比较功能的关键是如何剔除空行和空格的影响。

- 统计功能的思路。

统计功能，包括两个功能：总词汇数统计和每个词汇的使用次数统计。可以将每个词汇作为 key 保存到字典中，对文本从开始到结束，循环处理每个词汇，并将词汇设置为一个字典的 key，并将其 value 设置为 1，如果已经存在该词汇的 key，说明该词汇已经使用过，就将 value 累加 1。

- 比较功能的思路。

比较功能是对两个文本逐行进行比较，在比较一行时忽略空格的影响。在实现这个功能的时候，首先要将文本分成一行一行的，对每一行进行处理，忽略空格的个数，将字符串里有效字符转换成列表，然后进行比较。

3.11.4　具体实现

3.11.3 节分析了功能实现的思路，统计功能是将词汇放到字典类型中，用字典的 key 来存放单词，用 value 来存放个数，而比较功能则是使用字符串分隔成列表的公用方法。

1. 统计功能的具体实现

统计功能就是将一个文本的每个字符作为 key 值，放入字典中。以下代码将实现文本词汇数的统计。

```
>>> readtxt="""
...     this is a test txt!
...     can you see this ?
```

```
04  ...     """
05  >>>
06  >>> readlist=readtxt.split()
07  >>> dict={}
08  >>> for every_word in readlist:
09  ...     if every_word in dict:
10  ...         dict[every_word]+=1
11  ...     else:
12  ...         dict[every_word]=1
13  ...
14  >>> print(dict)
15  {'a': 1, 'this': 2, 'is': 1, 'txt!': 1, 'see': 1, 'can': 1, 'test': 1,
'you': 1, '?': 1}
```

第 6 行代码，对文本字符串 readtxt 做 split 操作，就可以获得该文本字符串的所有词汇，每个词汇都作为列表的元素返回。第 8~12 行代码，循环处理词汇列表，将词汇作为 dict 的 key，如果 key 已经存在，就将 value 值累加 1，否则将 value 设置为 1。

上面实现的文本统计的代码需要封装起来给整个程序使用，可以使用函数封装（使用语法定义 def　functionname(): 函数就可以了）。以下代码对上述代码进行函数封装。

```
def wordcount(readtxt):
    dict={}
    readlist=readtxt.split()
    for every_word in readlist:
        if every_word in dict:
            dict[every_word]+=1
        else:
            dict[every_word]=1
return dict
```

2.文本比较功能

文本比较首先需要将文本字符串分成一行一行的，使用字符串 splitlines()方法，可以将一个字符串按行分成一个列表，删除列表中的空元素和空白字符元素，再将两个文本进行循环比较。以下代码将实现文本比较功能。

```
01 #!/usr/bin/env python3
02 def testcmp(sText, dText):
03     cmpList = []
04     sLineList = []
05     for line in sText.splitlines():
06         if not line.isspace() and line!="":
07             sLineList.append(line)
08     dLineList = []
```

```
09    for line in dText.splitlines():
10       if not line.isspace() and line!="":
11          dLineList.append(line)
12    sLen = len(sLineList)
13    dLen = len(dLineList)
14    for step in range(max(sLen, dLen)):
15       try:
16          sWordList = sLineList[step].split()
17       except IndexError as e:
18          print("sfile is end")
19          sLineList.append("")
20       try:
21          dWordList = dLineList[step].split()
22       except IndexError as e:
23          print("dFile is end")
24          dLineList.append("")
25       if sWordList != dWordList:
26          cmpList.append((step, sLineList[step], dLineList[step]))
```

上述代码的实现逻辑主要包括：

- 第 4~11 行代码，对两个文本按行划分。
- 第 16~21 行代码，将文本每一行转换成列表，转换过程中忽略空白字符。
- 第 25~26 行代码，比较结果，如果不相等，就写信息到列表 cmpList 中，以供函数返回信息。

3.11.5　文本读写

前面实现的两个函数分别用来进行文本统计和文本比较，但是还需要实现从文本文件中读取字符串的功能、读文件功能。Python 内置了 file 类型来实现读文件功能。读者可以先使用 help(file)来查看 file 类型的详细信息，里面说明了 file 类型的几个主要操作方法。

（1）file 类型的创建

file 类型使用 file()方法创建，包括 3 个参数：name、mode、buffering。name 是指文件名。mode 是读写模式——r 模式是只读，w 是可写，a 模式是接在文件的末尾写，b 是写二进制文件，+则表示可读也可写。buffering 设置文件读写缓存，0 表示不设置，1 表示设置，也可以自行指定缓存大小。

（2）文件读写

file 类型提供了 read()和 write()来读写文件。read()可以读取文本文件到字符串，write()则将字符串写到文件中。

在这个应用案例中，我们只需要读文件到字符串中，使用下面的代码就可以：

```
>>> open_file=open(filename)
>>> file_txt=open_file.read()
```

3.11.6 命令行参数

比较两个文本，需要将两个文本的名字传给程序。本程序是命令行程序，可以通过命令行参数传送文本名称。所谓命令行参数，就是运行命令后所带的参数。在 UNIX 系统中，程序大多带有命令行参数。在 Window 系统下，cmd 窗口中也可带命令行参数。这里演示一个在 UNIX/Linux 的 shell 下运行的命令：

```
ls –a  /dev
```

上面的 ls 命令类似于 Window 下的 dir 命令，用来参看某个目录或者文件的信息。在 ls–a/dev 中，ls 为程序名字，–a /dev 是命令行参数。

命令行参数包括以下两部分：

（1）命令行选项（option）

在 ls–a/dev 命令中，–a /dev 为命令行参数，-a 为命令行选项，一般程序用来标志出参数的作用。命令行选项一般习惯有两种：短选项和长选项。-a 是短选项，由一个减号和一个字母组成，等价于--all 长选项。长选项由两个减号和若干个字母组成。下面的两个命令行参数是等价的。

```
ls –all /dev
ls –a /dev
```

（2）选项参数（option augument）

命令行选项后面跟的是具体的参数。命令行选项用来说明是什么参数，选项参数说明参数的具体值。例如：

```
ls –a /dev
```

-a 是命令行选项，/dev 是选项参数，用来说明命令行选项的具体参数值。

在 Python 中，读取命令行参数很方便。Python 标准库 optparse 模块来读取命令行参数。该模块是一个强大、灵活、易用、易扩展的命令行解释器。使用 optparse 模块，只需要很好的代码，就可以给程序添加专业的命令行接口。

optparse 是 Python 标准库模块，而不是内置模块，使用前需要使用 import 导入该模块。例如：

```
>>> import optparse
```

optparse 用法分成三步：

（1）创建 OptionParser 对象。

```
parser = optparse.OptionParser()
```

（2）添加参数选项。

```
    parser.add_option("-f", "--file",action="store", type="string",
dest="filename",default='./')
```

为 optparse 模块添加参数选项，使用 add_option()方法，包括 6 个参数值，下面介绍其中的 5 个主要参数：

- 第一个是短选项，通常是一个减号加一个字母。
- 第二个是长选项，通常是两个减号加一个说明参数作用的代词。
- action 表示对命令行选项后的参数做的操作。action 的参数有三种：store、store_true、store_false。store 表示会将参数值保存到 dest 所指定的变量中。对于不需要参数的选项，可以将 store 改为 store_ture 或者 store_false。设置为 store_ture，命令行参数出现该命令选项时，dest 所指定的变量会被设置为 ture，否则为 false；设置为 store_false 时，当命令行参数中出现该命令选项时，dest 所指定的变量会被设置为 false，否则为 true。
- type 表示参数类型。
- default 设置参数默认值。如果没在 add_option 中设置 default 这一项，并且命令行参数中也没有发现该选项，那么对应的变量值就是 None，也可以在 default 中设置默认值。

（3）解释命令行。

```
    (options, args) = parser.parse_args()
```

parse_args 方法会将命令行参数解析以后放到 options 中，就有了命名为 add-option()方法中所指定的 dest 的值属性名。可以直接访问 options 来得到命令行参数的值。例如：

```
    >>> print(options.filename)
```

在本案例中，有两个参数，分别是两个需要互相比较的文件的目录和名字，我们可以使用下面的代码来实现命令行参数的解析：

```
    >>>parser.add_option("-f","--file1",action="store",type="string",dest="filename1")
    >>>parser.add_option("-d","—file2",action="store",type="string",dest="filename2")
```

3.11.7　程序入口

在其他计算机语言中，一般都存在一个 main()方法，作为程序的入口。在 Python 中，不存在这样的 main，当运行一个 Python 文件的时候，Python 解析器会从文件开头一步一步执行代码，直到文件结束。

为了代码风格规范一致，Python 也有和 main()方法类似的东西，例如：

```
    >>> if __name__=="__main__":
    ...     print("ok")
```

```
...
```

在编写 Python 程序的时候，一般都把程序的入口放在__name__=="__main__"代码之后。在前面小节，已经完成文本读写、命令行参数分析、文本统计和比较部分的功能，本节将把这些功能模块组合在一起，完成一个完整的程序。

整个程序流程是，首先解析命令行参数，然后读取文本，最后对文本做比较。下面是 PyMerge 主逻辑的功能实现：

```
01 import optparse
02 import sys
03
04 def wordcount(readtxt):
05     dict = {}
06     readlist = readtxt.split()
07     for every_word in readlist:
08         if every_word in dict:
09             dict[every_word] += 1
10         else:
11             dict[every_word] = 1
12     return dict
13
14 def testcmp(sText, dText):
15     cmpList = []
16     sLineList = []
17     for line in sText.splitlines():
18         if not line.isspace() and line != "":
19             sLineList.append(line)
20     dLineList = []
21     for line in dText.splitlines():
22         if not line.isspace() and line != "":
23             dLineList.append(line)
24     sLen = len(sLineList)
25     dLen = len(dLineList)
26     for step in range(max(sLen, dLen)):
27         try:
28             sWordList = sLineList[step].split()
29         except IndexError as e:
30             print("sfile is end")
31             sLineList.append("XXX")
32             # sWordList = sLineList[step]
33         try:
34             dWordList = dLineList[step].split()
35         except IndexError as e:
```

```
     36            print("dFile is end")
     37            dLineList.append("YYY")
     38            # dWordList = dLineList[step]
     39        if sWordList != dWordList:
     40            cmpList.append((step, sLineList[step], dLineList[step]))
     41
     42    return cmpList
     43
     44 if __name__ == '__main__':
     45    parser = optparse.OptionParser()
     46    parser.add_option("-s", "--sFile", action="store", type="string",
dest="sFileName")
     47    parser.add_option("-d", "--dFile", action="store", type="string",
dest="dFileName")
     48    (options, args) = parser.parse_args()
     49    with open(options.sFileName, 'r') as sFile, open(options.dFileName,
'r') as dFile:
     50        #开始统计文件
     51        sText = sFile.read()
     52        dText = dFile.read()
     53        print("文件 %s" %options.sFileName)
     54        print("词汇总数: %d" %len(wordcount(sText)))
     55        print("各词汇统计: %s" %wordcount(sText))
     56
     57        print("文件 %s" %options.dFileName)
     58        print("词汇总数: %d" %len(wordcount(dText)))
     59        print("各词汇统计: %s" %wordcount(dText))
     60
     61        #文本比较
     62        cmpList = testcmp(sText, dText)
     63        for diff in cmpList:
     64            print("%s %s: %s" %(options.sFileName, diff[0], diff[1]))
     65            print("%s %s: %s" %(options.dFileName, diff[0], diff[2]))
```

第 45~48 行的功能是负责解析命令行参数。

第 49~52 行负责将文本读到字符串中，使用 with…as…的方式打开文件，避免了文件名不存在、文件无法打开等问题。如果文件无法正常打开读取，程序直接退出并显示错误。

第 53~59 行完成对两个文本的词汇总数和各词汇数的统计，使用上面的 wordcount 来完成词汇统计，打印词汇统计信息。

第 62~65 行完成两个文本的比较，使用 testcmp()函数来进行文本比较，获得比较结果信息以后，将比较的信息打印出来。

3.11.8 运行效果

将 testcmp、wordcount 和程序入口的代码合并到一起以后，保存文件名到 PyMerge.py，完成该程序的开发。可以在 Window 的 cmd 窗口或者 UNIX/Linux 的 shell 下运行如下命令：

```
python  PyMerge.py -s a.txt -d b.txt
```

运行效果如图 3.1 所示。

```
$ python PyMerge.py -s a.txt -d b.txt
文件 a.txt
词汇总数: 4
各词汇统计:  {'aaa': 1, 'bbb': 1, 'ccc': 1, 'd': 1}
文件 b.txt
词汇总数: 4
各词汇统计:  {'aaa': 1, 'bbb': 1, 'ccc': 1, 'dd': 1}
a.txt 3: d
b.txt 3: dd
```

图 3.1 PyMerge 运行效果

执行命令时，需要保证 PyMerge.py、a.txt、b.txt 三个文件在当前目录下。

3.12 本章小结

本章主要学习了 Python 的简单类型和容器类型。读者通过本章的学习，应该熟练地掌握 Python 的简单类型和容器类型用法。在学习本章的过程中，需要注意以下几点：

- Python 的缩进要使用 4 个空格，不可用 Tab 和空格混用。
- 要多使用 help 和 id 这样的自省功能，Python 的内置模块和标准库都提供了详细的自省信息。查看这些自省功能，就可以了解各内置模块和标准库的详细用法。
- 字符串、列表、元组、字典是 Python 编程中很重要的 4 种数据类型，要熟练掌握它们的使用方法。
- 可变类型和常量类型的区别。

学习完本章以后，读者需要回答下列问题：

（1）元组和列表的区别是什么，何种情况下使用元组，何种情况下使用列表？

（2）字典的 key 可以是哪些类型，不可以为哪些类型？

（3）假设存在一个几百年的家族，族长需要用一个程序把几百年来的族谱都录入到计算机中，而计算机需要能够提供查询：查询家族某代某个人的个人情况和亲属血缘情况。使用 Python 来编写这样的程序时，使用何种类型来存储族谱信息，如何存储？

（4）3.11 节的应用案例是用来对英文文档进行比较的，如果要对中文文档进行比较，该如何处理？

第 4 章

◂ 流程控制和函数 ▸

上一章介绍了 Python 常用的内置类型，本章将会深入讨论 Python 的流程结构和函数。上一章在讨论内置类型的过程中，简单介绍了使用 for 循环控制结构和简单的函数封装功能模块，本章将讨论更多控制结构和更多的函数用法。

本章的主要内容是：

- 控制代码的执行顺序。
- 循环代码。
- 函数的使用。
- 嵌套函数。
- 八皇后算法。

4.1 流程控制

自一代大师 Dijkstra（第七届图灵奖得主）于 1968 年发表的著名文章《Go To Statement Considered Harmful》否定了 goto 的用法，使用条件选择结构和循环结构来控制程序的流程已经成为各种现代计算机语言的基础。Python 语言也不例外，不过 Python 的条件和循环结构又和其他语言略有不同。

4.1.1 选择结构

在程序执行的过程中，时常依据一些条件的变化改变程序的执行流程。改变程序流程的功能，主要由条件语句配合布尔表达式来完成。在 Python 中，使用 if 语句来实现这种流程选择的控制。例如：

```
>>>if x==0:
>>>    print("ok")
```

如果 x 等于 0 就会打印 ok，如果 x 不等于 0 就不会打印 ok。对于需要分别对应于满足和不满足条件来执行不同的流程程序，可以使用关键字 else 引出另一个程序流程。例如：

```
>>> if x==0:
...     print("x 等于 0")
... else:
...     print("x 不等于 0")
...
```

有时候，程序的分支可能是三个或更多。此时，就需要用 elif 语句引出更多的分支。elif 语句是“else if”的缩写，每一个 elif 语句均为程序引出一个分支。elif 语句的数量没有限制，例如：

```
>>> if x==1:
...     print("not 1")
... elif x==2:
...     print("not 2")
... elif x==3:
...     print("not 3")
... elif x==4:
...     print("not 4")
```

Python 会依次执行 if 语句下的程序按顺序检查条件表达式，当找到第一个满足要求的表达式后，执行此分支内的语句。剩下的条件，即使有满足要求的，也不做检查。需要注意的是，上面语句中的最后一个分支是 elif x==4，这样语法上虽然没有错误，但是为了代码更规范严谨，一般在编写这样的分支代码时，最后一个分支应该是 else，例如：

```
>>> if x==1:
...     print("not 1")
... elif x==2:
...     print("not 2")
... elif x==3:
...     print("not 3")
... elif x==4:
...     print("not 4")
... else:
...     print("not all")
```

最后 else 分支用于对于不满足上面所有其他分支的处理，这样不会漏过没有处理的选择情况，如果对于不满足上面所有其他分支的情况不做任何处理，可以使用 pass 语句来说明不需要做特定处理。例如：

```
>>> if x==1:
...     print("not 1")
... elif x==2:
...     print("not 2")
... elif x==3:
...     print("not 3")
```

```
... elif x==4:
...     print("not 4")
... else:
...     pass
```

4.1.2　for 循环结构

在上一章，我们大量使用 for…in 来访问序列类型和字典类型。Python 的 for 语句依据任意序列或者字典中的子项，按照它们在序列中的顺序来进行迭代。例如：

```
>>> ab=['a',1,3,4]
>>> for x in ab:
...     print(x)
...
a
1
3
4
```

需要注意的是，在循环过程中，修改循环的序列（当是可变序列类型时）是很不安全的，例如：

```
>>> a=['a','b','c','d']
>>> for x in a:
...     if x=='c':
...         bb=a.pop(0)
...     print(x)
...
c
```

对于这种情况，可以使用[:]对列表进行复制。当 b 是一个列表的时候，a=b[:]，就可以复制 b 的列表到 a。需要特别注意的是，a=b，并不是将 b 的列表复制到 a，只是让 a 和 b 指向同一个列表对象，只有使用 a=b[:]语句才是创建一个新列表对象复本，让 b 指向它。例子 4.1 说明两者之间的区别。

例子 4.1　列表的复制

```
01      >>> a=[1,2,3,4]
02      >>> b=a
03      >>> id(a)
04      31427880
05      >>> id(b)
06      31427880
07      >>> a.pop()
08      4
```

```
>>> print(a)
[1, 2, 3]
>>> print(b)
[1, 2, 3]
>>> a=[1,2,3,4]
>>> b=a[:]
>>> id(a)
31421424
>>> id(b)
31427560
>>> a.pop()
4
>>> print(a)
[1, 2, 3]
>>> print(b)
[1, 2, 3, 4]
```

从上面的代码可以看出，使用 b=a 时，实际上变量 a 和 b 所指向的列表是同一个列表（因为它们的对象 id 是一样的），所以 a 进行 pop 操作，b 的列表也一样被 pop 了；使用 b=a[:]时，b 的列表是复制 a 的对象（可以看到它们的对象 id 不一样），对 a 做 pop 操作，b 的列表并没有变化。

对于上面需要在循环中修改列表的情况，可以使用复制列表的技术来避免修改列表带来不安全循环的情况，例如：

```
>>> a=['a','b','c','d']
>>> for x in a[:]:
...     if x=='c':
...         bb=a.pop(0)
...     print(x)
```

4.1.3 while 循环结构

while 和 for 一样，也是一种循环结构。和 for 不同的是，while 循环的条件取决于 while 后面表达式的布尔值，例如：

```
>>> i=0
>>> while i<6:
...     i+=1
...     print(i)
```

当 while 后面的表达式为真时，执行 while 语句下的代码块，否则执行循环结束以后的代码。

在 while 循环中，需要强制退出循环的时候，可以使用 break 语句，例如：

```
>>> i=0
>>> while i<6:
...     i+=1
...     if i==4:
...        break;
...     print(i)
...
1
2
3
```

当使用 break 语句时，会直接退出循环，即不执行循环代码块下面的部分，也不继续执行循环处理，而是直接跳到循环结束后，执行循环结束后的代码。

continue 和 break 语句的区别是，continue 虽然也不执行循环代码下面的部分，但是 continue 会跳到循环结构的循环开始部分，继续下一次循环。例子 4.2 列出两个关键字的区别。

例子 4.2　break 和 continue 的区别

```
>>> for i in range(6):
...     if i==4:
...        break
...     print(i)
...
0
1
2
3
>>> for i in range(6):
...     if i==4:
...        continue
...     print(i)
...
0
1
2
3
5
>>> i=0
>>> while i<6:
...     print(i)
...     i+=1
...     if i==4:
...        break
```

```
26  ...
27  0
28  1
29  2
30  3
31  >>> i=0
32  >>> while i<6:
33  ...     print(i)
34  ...     i+=1
35  ...     if i==4:
36  ...        continue
37  ...
38  0
39  1
40  2
41  3
42  4
43  5
```

从例子 4.2 的第 3 行和第 12 行、第 25 行和第 36 行的比较情况可以看到，break 和 continue 的区别在于是否继续执行循环。这一点和其他计算机语言非常相似。

需要注意的是，Python 的循环也可以带 else 分支，这一语言特性是其他语言所没有的，用起来也非常方便，例如：

```
>>> for i in range(6):
...     print(i)
... else:
...     print("ok")
```

循环语句的 else 分支是可有可无的，若有，则表示如果循环语句是正常结束的（不是使用 break 强制结束的），就执行 else 分支里的代码块。例子 4.3 解释了循环语句中 else 的作用。

例子 4.3　循环语句的 else 用法

```
01  >>> for i in range(6):
02  ...     print(i)
03  ... else:
04  ...     print("ok")
05  ...
06  0
07  1
08  2
09  3
10  4
11  5
```

```
ok
>>>
>>>
>>>
>>> for i in range(6):
...     print(i)
... else:
...     print("ok")
...
0
1
2
3
4
5
ok
>>> for i in range(6):
...     if i==4:
...         continue
...     print(i)
... else:
...     print("ok")
...
0
1
2
3
5
ok
>>> for i in range(6):
...     if i==4:
...         break
...     print(i)
... else:
...     print("ok")
...
0
1
2
3
>>>
```

例子 4.3 列出了 3 种情况：正常循环，使用 continue 的循环，使用了 break 的循环。在这

3 种循环中，前两种正常执行循环到结束，所以都执行了 else 分支的代码块，而 break 强行结束了循环，直接跳到循环结束代码之后，没有执行 else 的代码块。

4.2 函数

对于需要重复使用的代码功能模块，一般都会将其封装成函数，以提高代码的可读性，使得程序结构整齐清晰。

4.2.1 函数的定义

在 Python 中，定义函数的语法形式如下：

```
def <function_name> ( <parameters_list> ):
   <code block>
```

其中，def 用来声明开始定义一个函数，function_name 是函数的名字，parameters_list 是函数输入的参数，code block 是函数的功能模块代码。例如，当需要将一个字符串中的字符 a 和 b 替换成 h 和 i 时，可以将该功能封装成函数 transchar：

```
>>> def transchar(para_str):
...     if type(para_str)==str:
...         str_1=para_str.replace('a','h')
...         str_2=str_1.replace('b','i')
...         return str_2
...     else:
...         return false
...
>>> transchar("abdbi")
'hidii'
```

4.2.2 函数的参数

Python 的函数参数使用方法比较灵活，既可以选择参数个数，也支持默认值，并且可以自行指定赋值顺序。总的来说，Python 函数的参数有如下特点：

- 参数有默认值，填写参数时个数可选。
- 参数的赋值，可以按照参数的名字来赋值，赋值的顺序可以改变。
- 支持不定个数参数，可以编写类似于 C 中 printf 那样的函数。

例如，需要编写一个用于计算长方体体积的函数，包括 3 个参数：长、宽、高。对于这样一个函数，可以定义如下：

```
>>> def getvolume(len,width,height):
...     return len*width*height
```

也可以设定这 3 个参数，这样没有指定参数值，会使用默认值进行调用，例如：

```
>>> def getvolume(len=0,width=0,height=0):
...     return len*width*height
...
>>> getvolume(12,13)
0
```

并且参数的赋值顺序也不一定是固定的，只要指定名字调用就没有问题，例如：

```
>>> getvolume(width=12, height=2,len=3)
72
```

如果使用过 C 语言的 printf()，就会对可变参数的意义比较了解。printf()函数只需要按照指定的格式，就可以输入任意个数的参数。Python 的 print()用法和 C 语言的 printf()颇为相似。例子 4.4 是 Python 的 print()方法和 C 语言的 printf()函数的比较。

例子 4.4　Python 的 print()方法和 C 语言的 printf()函数

```
>>>print("你好")
你好
>>> print("%s" % "你好")
你好
>>> print("%s,%s" %("你好","中国"))
你好,中国
>>> print("%s,%s,%d" %("你好","中国",2018))
你好,中国,2018

printf("你好")  #以下是 C 语言的语句
你好
printf("%s","你好")
你好
printf("%s,%s","你好","中国")
你好,中国
printf("%s,%s,%d","你好","中国",2018)
你好,中国,2018
```

在例子 4.4 中，第 1 行到第 8 行是 Python 的 print()方法，第 10 行到第 17 行是 C 语言的 printf()函数。从中可以看出，两者是非常相似的，不同的是，print()方法不是 Python 函数。下面使用 Python 的不定参数来定义一个和 C 语言 printf()函数一样的函数：

```
>>> def printf(format,*arg):
...     print(format%arg)
...
```

```
>>> printf("你好")
你好
>>> printf("%s,%s","你好","中国")
你好,中国
>>> printf("%s,%s,%d","你好","中国",2018)
你好,中国,2018
>>>
```

在 Python 中，不定参数使用*arg 来表示，而 arg 实际是一个元组（tuple）。它上面存放了输入的参数，例如：

```
>>> def getchange(*arg):
...     print(arg)
...
>>> getchange(1,2,3)
(1, 2, 3)
>>> getchange("a","b")
('a', 'b')
```

可以通过访问元组方法来访问可变参数。例如，编写一个累加所有参数的函数：

```
>>> def addall(*arg):
...     total=0
...     for arg_one in arg:
...         total+=arg_one
...     return total
...
>>> addall(1,2,3,4)
10
addall(1,2,3,4,8,9)
27
```

对于可变参数，除了使用*arg 来表示外，也可以使用**argv 来表示。不同的是，使用**argv 表示时，可变参数就会放到一个字典中，并且在输入参数时必须说明参数的名字；使用*arg 方法，在输入参数的时候，不能使用参数的名字。例如：

```
>>> getall(1)
(1,)
>>> getall(1,2)
(1, 2)
>>> getall(1,2,3)
(1, 2, 3)
>>> getall1(one=1)
{'one': 1}
>>> getall1(one=1,two=2)
{'two': 2, 'one': 1}
```

固定参数和可选参数、可变参数可以组合起来。Python 优先接受固定参数，然后是可选参数，最后是可变参数，所以*arg、**argv 只能够放到参数的最后，并且*arg 必须放到**argv 之前，可变参数只能放到固定参数后面。例如：

```
>>> def funexe(keyparam,chioce=1,*arg,**keywords):
...     print(keyparam,chioce,arg,keywords)
...
>>> funexe('a','b','c','e')
a b ('c', 'e') {}
>>> funexe('a','b','c','e',three=3)
a b ('c', 'e') {'three': 3}
```

4.2.3　函数调用和返回

可以使用函数名称来调用函数，例如：

```
>>> funexe('a','b','c','e',three=3)
a b ('c', 'e') {'three': 3}
```

函数名称本身也可以作为参数传递调用，例如：

```
>>> def addtwo(a,b):
...     return a+b
...
>>> addtwo(1,2)
3
>>> add1=addtwo
>>> add1(3,5)
8
```

也可以将函数名作为参数传给另一个函数做调用，例如：

```
>>> def test2(fun,a,b):
...     return fun(a,b)
...
>>> test2(add1,3,4)
7
```

对一个函数，需要有返回值时，可以使用 return 语句。若不使用 return 语句，则返回为 None 类型。例如：

```
>>> def addtwo(a,b):
...     return a+b
...
>>> print(addtwo(2,3))
5
>>> def addtwo(a,b):
```

```
...     a+b
...
>>> print(addtwo(2,3))
None
```

4.2.4 lambda 函数

lambda 函数是函数式编程中的一个概念。函数式编程是一种编程典范，不同于 C 语言这样的命令式语言，它将电脑运算视为函数（lambda 演算）计算的过程。最早的函数式编程语言是 LISP，现代的函数式编程语言有 Haskell、Erlang 等，函数式编程语言在人工智能、数据挖掘、算法设计等计算机领域有着重要的作用。

可以简单地将 lambda 函数理解为一种单行的匿名函数，例如：

```
>>> b=lambda a,b:a+b
>>> b(1,2)
3
```

需要注意的是，匿名函数只能有一行代码，可以有多个参数，包括可变参数，但是表达式只能为一个，并且只能为简单的操作。更本质地说，后面的表达式是能够返回一个值的，不能返回值的不能放在这里。例如：

```
>>> g = lambda x, y=0, z=0: x+y+z
>>> g(4,5,6)
15
>>> (lambda x, y=0, z=0: x+y+z)(1,2,3)
6
```

在 Python 中，lambda 函数可以通过一些技巧来实现 if 或者 for 的功能，代码量则少很多，比如可以使用 and 和 or 表达式来代替 if 语句，例如：

```
>>> def judage(a):
...     if a:
...         b=3
...     else:
...         b=2
...     return b
>>> f=lambda a:(a and 3 or 2)
```

集合这些 lambda 函数的特性，可以使用 lambda 函数以更短的代码实现一些需要多条循环选择语句实现的功能。例子 4.5 给出求质数的两种方法，从中可以看出使用 lambda 函数和普通函数求质数的区别。

例子 4.5　求质数的两种方法比较

```
01   >>> def isPrime(n):
02   ...     mid = int(pow(n,0.5)+1)
```

```
    03   ...     for i in range(2,mid):
    04   ...       if n % i == 0 : return False
    05   ...     return True
    06   ...
    07   >>> primes=[]
    08   >>> for i in range(2,100):
    09   ...     if isPrime(i): primes += [i]
    10   ...
    11  >>> print(primes)
    12   [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
67, 71, 73, 79, 83, 89, 97]
    13   >>> from functools import reduce
    14  >>> print(reduce(lambda l,y:not 0 in map(lambda x:y % x, l) and l+[y]
or l,range(2,100), [] ))
    15   [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61,
67, 71, 73, 79, 83, 89, 97]
    16  >>>
```

第 1 行到第 11 行使用普通的函数来求 100 内的所有质数。第 14 行使用了 lambda 函数方法。其中，not 0 in map(lambda x:y %x,l) 表示数 y 能否被 l 中的任何一个数整除，继而返回 l+[y]或者 l，这和第 4~5 行的代码是同一个作用。

对于习惯了命令式编程的人来说，第 1~11 代码虽然较长，但是比较清晰易懂，容易维护。对于熟悉函数式编程的人来说，第 14 行代码不但简洁，而且可读性更好。Python 同时提供了这两种编程方式和支持，具体使用哪种，要视个人情况而定。

4.2.5　嵌套函数

在 Python 中，可以在函数内部定义函数，例如：

```
>>> def getfun(x,y):
...     def addfun(a,b):
...         return a*b
...     return addfun(x,y)
...
>>> getfun(2,3)
6
```

对于嵌套函数，内层函数可以访问外层函数的变量，但是 Python 没有提供由内而外的绑定措施，所以在使用内层函数访问外层函数的时候要特别注意这一点，以免逻辑出错。可以参看下面的例子：

```
>>> def getfun(x,y):
...     a=3
...     def test2():
```

```
...       a=1
...     return a
...
>>> getfun(1,3)
3
```

在上面的代码中，getfun()函数的变量 a 在 test2 中无法被绑定，最后返回的结果 getfun()的变量 a 还是绑定原来的数值对象 3。

4.2.6 函数的作用域

在 Python 中查找变量，有一个所谓的 LGB 原则：L 是 local name space，局部命名空间的意思；G 是 global name space，全局命名空间的意思；B 是 buildin name space，内在命名空间的意思。LGB 原则是指，对于一个变量名称，先查找局部命名空间，再查找全局命名空间，最后查找内在命令空间。

例子 4.6 函数作用域

```
01  >>> var=[]
02  >>> def test2():
03  ...     var=[1]
04  …     var.append(1)
05  ...     return var
06  >>> test2()
07  [1,1]
08  >>> print(var)
09  []
10  >>> def test3():
11 ...     var.append(2)
12 ...     return var
13 ...
14 >>> test3()
15 [2]
16 >>> print(var)
17 [2]
```

例子 4.6 中的第 3 行代码定义一个变量为一个列表。该变量在局部命名空间下，所以优先级最高，所以在第 4 行进行 append 操作的时候，应该对局部变量 var 进行操作，而不是对全局变量 var 进行操作。第 11 行没有定义局部变量，所以 append()操作作用在全局变量 var 上。

4.3 开始编程：八皇后算法

在 4.1 和 4.2 节中讨论了 Python 的流程控制和函数的用法，本节将使用这些知识点来实现八皇后问题的算法。

【本节代码参考：C04\py_4.7.py】

4.3.1 八皇后问题

八皇后问题：在 8*8 国际象棋棋盘上，要求在每一行放置一个皇后，且能做到在竖方向、斜方向都没有冲突。国际象棋的棋盘如图 4.1 所示。

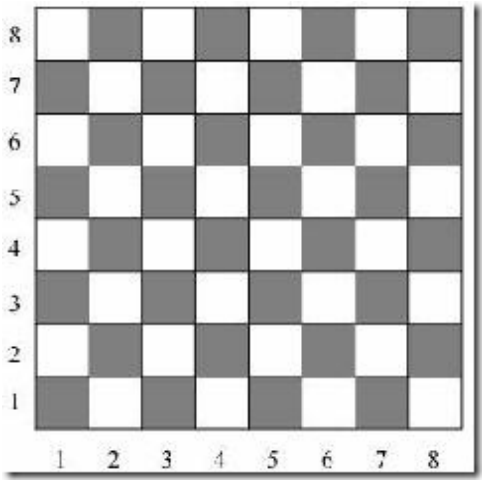

图 4.1　国际象棋棋盘

八皇后问题是一个古老而著名的问题，是 19 世纪著名的数学家高斯于 1850 年提出的：在 8*8 格的国际象棋上摆放八个皇后，使其不能互相攻击，即任意两个皇后都不能处于同一行、同一列或同一斜线上，问有多少种摆法。高斯认为有 76 种方案。1854 年在柏林的象棋杂志上不同的作者发表了 40 种不同的解，后来有人用图论的方法解出 92 种结果。计算机诞生以后，八皇后问题也成为计算机数据结构和算法的经典题目。

4.3.2 问题分析

对于这种较为复杂的算法问题，可以采用逐步试探的方法，能够继续前进，则更进一步，如果不能，就换个方向尝试，可称之为回溯法。

首先我们来分析一下国际象棋的规则。对于一个国际象棋的棋盘，每一个点，我们都用一个坐标来表示，这里采用图 4.1 一样的坐标，左下角为（1，1），右上角为（8，8），一个皇后（x,y）能否被另一个皇后（a,b）吃掉主要取决于下面四个方面：

（1）x=a，两个皇后在同一行上。

（2）y=b，两个皇后在同一列上。

（3）x+y=a+b，两个皇后在同一斜向正方向。

（4）x-y=a-b，两个皇后在同一斜向反方向。

有了上面的规则，可以先从第一个皇后开始分析，如果将第一个皇后放到（1，1）格中，那么根据规则：

（1）第二皇后可以放在（2，3）、（2，4）到（2，8）的任一个，现在假设放到（2，3）。

（2）第二皇后放到（2，3），那么第三个皇后只有（3，5）、（3，6）到（3，8）这四种可能可选择。现在假设放到（3，5）。

（3）第三个皇后放到（3，5），那么第四个皇后只有（4，2）、（4，7）、（4，8）这三种可能可选，假设放到（4，2）中。

（4）第四个皇后放到（4，2），那么第五个皇后只有（5，4）和（5，8）这两个地方可选，假设放到（5，4）中。

（5）第五个皇后放在（5，4），那么第六个皇后没有安全的位置可放。

在摆到第六个皇后时，就会无法再继续下去了，这时回到放第五个皇后的第二个选择（5，8），然后继续尝试第六个皇后，发现仍然没有安全的位置，只好再回到放第四个皇后，继续第四个皇后的其他可能。以此类推，不断尝试，一直到放最后一个皇后。

从第一步开始尝试，逐步尝试后，失败了就返回上一个步骤，尝试其他可能。根据上面的分析，用回溯的方法解决八皇后问题的步骤为：

（1）从第一列开始，为皇后找到安全位置，然后跳到下一列。

（2）如果在第 n 列出现死胡同，并且该列为第一列，那么棋局失败，否则后退到上一列，再进行回溯。

（3）如果在第 8 列上找到了安全位置，那么棋局成功。

4.3.3 程序设计

根据 4.3.2 小节对八皇后问题的分析，八皇后问题的步骤在于三步：找安全位置，继续下一列，如果下一列找不到安全位置，就进行回溯，直到八个皇后都找到安全位置为止。

对于程序设计来说，首先设计象棋棋盘的数据结构，然后编写安全位置的判断，最后撰写回溯的功能。

（1）象棋棋盘的数据结构

可以用列表来表示一个象棋棋盘，每个列表里有 8 个列表，每个列表有 8 个元素，例如：

```
    >>> chess=[[0 for x in range(8)] for x in range(8)]
    >>> print(chess)
    [[0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0,
0], [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0,
```

```
0], [0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0]]
    >>>
```

对于 chess 列表，初始元素值均为 0，元素值大于 0 为不安全、0 为安全。

（2）安全位置的判断

根据象棋棋盘数据结构的设计，凡是元素值为 0 的都是安全的，凡是元素值不为 0 的都是不安全的。可以使用下面的函数来实现这个功能：

```
>>> def judgedanger(chess,x,y):
...     if chess[x][y]==0:
...         return True
...     else:
...         return False
```

（3）回溯功能的实现

回溯的功能，需要先判断安全的位置，然后将皇后放到安全的位置，在将皇后放到安全的位置时，同时需要将该皇后的吃棋范围记录到 chess 列表中，这样下一步可以根据 chess 列表来判断安全的位置，同理，在该位置被认为无效，需要回溯的时候，同样需要将棋的范围位置信息清除，恢复到放皇后之前的状态。

实现记录吃棋范围信息的记录，可以使用如下代码：

```
>>> def setdanger(chess,x,y):
...     for col in range(len(chess)):
...         for row in range(len(chess[0])):
...             if col==x:
...                 chess[col][row]+=1
...             elif row==y:
...                 chess[col][row]+=1
...             elif col+row==x+y:
...                 chess[col][row]+=1
...             elif col-row==x-y:
...                 chess[col][row]+=1
...             else:
...                 pass
```

上面的代码根据皇后吃棋的四个判断规则对棋盘列表的每个位置做判断，如果皇后可以吃到，就将位置值加 1，表示该位置不再安全。

对于清除吃棋的范围位置信息，使用相反的逻辑思路。和记录吃棋范围信息相反，它将皇后可以吃到的位置信息减 1，减到 0 时表示该位置安全，可以放皇后。

```
>>> def erasedanger(chess,x,y):
...     for col in range(len(chess)):
...         for row in range(len(chess[0])):
...             if col==x:
```

```
...                chess[col][row]-=1
...            elif row==y:
...                chess[col][row]-=1
...            elif col+row==x+y:
...                chess[col][row]-=1
...            elif col-row==x-y:
...                chess[col][row]-=1
...            else:
...                pass
```

在上面实现了记录吃棋位置信息和清除吃棋位置信息的函数后，就可以将这两个函数用于回溯中的吃棋范围信息记录。

回溯过程中需要经常判断下一行是否有安全位置，所以先编写一个判断一行中有无安全位置的函数：

```
>>> def judgecol(chess,col):
...     for row in range(len(chess[col])):
...         if judgedanger(chess, col,row):
...             break
...     else:
...          return False
...     return True
```

在这些代码的基础上，可以使用回溯法。按照 4.3.2 小节对回溯规则的分析，回溯的步骤如下：

（1）将第 n 个皇后放到一个安全的位置。

（2）将 n 皇后的吃棋范围标出，尝试放置 n+1 皇后的安全位置。

（3）如果 n+1 皇后无安全位置，就回溯到 n 皇后，让 n 皇后清除吃棋范围，尝试下一个安全位置，重复第（2）步。

根据上面回溯步骤的分析，可以得到如下代码：

```
01   >>> def tryqueen(chess,col,flag,result):
02   ...     flag[0]=True
03   ...     if col==8:
04   ...         print("find")
05   ...     else:
06   ...         if   judgecol(chess,col):
07   ...             for row in range(len(chess[col])):
08   ...                 if judgedanger(chess,col,row):
09   ...                     #print "ok"+str(col)+":"+str(row)
10   ...                     setdanger(chess,col,row)
11   ...                     result.append((col,row))
12   ...                     tryqueen(chess,col+1,flag,result)
```

```
13  ...                    if flag[0]==False:
14  ...                        erasedanger(chess,col,row)
15  ...                        result.pop()
16  ...            else:
17  ...                 flag[0]=False
```

例子 4.7 是对上面回溯三段分析的实现。第 6 行代码判断该皇后是否还有安全位置，如果有，就开始尝试放置到第一个安全位置，在第 9 行成功放置到安全位置，第 10 行将该皇后的吃棋范围标出来，接着在第 12 行放置下一个皇后，如果下一个皇后没有安全位置（用 flag 标志来表示），就在第 14 行使用 erasedanger 清除吃棋范围信息，该皇后将尝试下一个安全位置，然后重复类似的步骤。一直到 8 个皇后全部放到安全的位置（代码第 4 行），求出八皇后问题的一个解。

4.3.4　问题深入

在 4.3.3 小节中，经过分析，已经可以使用函数求得八皇后问题的一个解，那么如何取得八皇后问题所有的 92 个解呢？

在上一小节的代码中，回溯结束的条件是：当第 8 个皇后可以放置到象棋棋盘中时，函数将是否有安全位置的标志设置为 True，这样尝试的过程就结束了。如果修改回溯结束的条件为：在第 8 个皇放置到象棋棋盘后打印出结果列表，并且将标志人为地设置为没有安全位置（将 flag[0]设置为 False），那么情况就如同没有找到解一样，函数会回溯上一次尝试的地方，尝试下一个可能，因为回溯结束条件中的标志被人为地设置为没有找到，这样函数就会尝试所有的可能，也就可以找出所有的解。

下面的代码是对八皇后所有解的求解。

```
01  >>> def tryqueen(chess,col,flag,result):
02   ...    flag[0]=True
03   ...    if col==8:
04   ...        print(result)
05   ...        flag[0]=False
06   ...    else:
07   ...       if  judgecol(chess,col):
08   ...          for row in range(len(chess[col])):
09   ...             if judgedanger(chess,col,row):
10  ...                 #print "ok"+str(col)+":"+str(row)
11  ...                 setdanger(chess,col,row)
12  ...                 result.append((col,row))
13  ...                 tryqueen(chess,col+1,flag,result)
14  ...                 if flag[0]==False:
15  ...                     erasedanger(chess,col,row)
16  ...                     result.pop()
17  ...           else:
```

```
18  ...                 flag[0]=False
```

修改部分主要是第 4~6 行，第 4 行开始打印结果列表，第 5 行将标志设置为 False，这样 tryqueen()函数就会一直尝试下去，直到尝试了所有可能，找出八皇后问题的某个解。

4.3.5 问题总结

八皇后问题是在计算机算法上的一个经典题目，解决的算法也很多，最简单的是穷举法。穷举法是对八皇后所有位置的可能进行一一判断（总共有 8 的 8 次方个可能），然后从中得到符合要求的 92 种可能。本小节使用的是较为复杂的算法：回溯法。相比穷举法，回溯法的算法性能更好一些，实现要复杂一些。例子 4.7 是用 Python 实现八皇后回溯算法的完整代码。

本例的算法有一些缺陷，并不能实现八皇后的所有解，这里只是用回溯法和简单的函数给读者演示一种求解的过程，等读者学完所有内容后，可以再利用一些高级内容实现更好更完善的算法。

例子 4.7　八皇后问题的 Python 回溯实现

```
def setdanger(chess,x,y):
    for col in range(len(chess)):
        for row in range(len(chess[0])):
            if col==x:
                chess[col][row]+=1
            elif row==y:
                chess[col][row]+=1
            elif col+row==x+y:
                chess[col][row]+=1
            elif col-row==x-y:
                chess[col][row]+=1
            else:
                pass

def erasedanger(chess,x,y):
    for col in range(len(chess)):
        for row in range(len(chess[0])):
            if col==x:
                chess[col][row]-=1
            elif row==y:
                chess[col][row]-=1
            elif col+row==x+y:
                chess[col][row]-=1
```

```
            elif col-row==x-y:
                chess[col][row]-=1
            else:
                pass

def judgedanger(chess,x,y):
    if chess[x][y]==0:
        return True
    else:
        return False

def judgecol(chess,col):
    for row in range(len(chess[col])):
        if judgedanger(chess,col,row):
            break
    else:
         return False
    return True

def tryqueen(chess,col,flag,result):
    flag=True
    if col==8:
         print(result)
         result=[]
         flag=False
    else:
        if judgecol(chess,col):
            for row in range(len(chess[col])):
                if judgedanger(chess,col,row):
                    print("安全"+str(col)+":"+str(row))
                    setdanger(chess,col,row)
                    result.append((col,row))
                    tryqueen(chess,col+1,flag,result)
                    if flag==False:
                        erasedanger(chess,col,row)
                        result.pop()
             else:
                    flag=False

if __name__=='__main__':
    chess=[[0 for x in range(8)] for x in range(8)]
    result=[]
    flag=True
```

```
        tryqueen(chess,0,flag,result)
```

4.4 本章小结

本章讨论了 Python 的流程控制和函数的相关知识点，在综合应用方面列举了八皇后问题的求解方法。在学习本章的过程中，需要注意的是：

- Python 的 for 用法和其他语句颇为不同。
- Python 的循环结构也可带 else 语句，这是 Python 语言的特性。
- 使用嵌套函数时，目前 Python 还不支持对外层变量的绑定。

学习完本章，读者可以思考如下问题：

（1）如何用穷举法求解八皇后问题？

（2）如果是在一个 m*n 的棋盘上放置 n 个皇后，该如何求解？