

[第一章 快速改造：基础知识 4](#_Toc25371)

[1.4 数字和表达式 4](#_Toc28876)

[除法运算 4](#_Toc16809)

[幂（乘方）运算符 4](#_Toc15861)

[1.4.2 十六进制和八进制 4](#_Toc9819)

[1.7 获取用户输入 5](#_Toc6009)

[input函数 5](#_Toc16031)

[1.8 函数 5](#_Toc11610)

[pow 函数 5](#_Toc31123)

[abs函数 5](#_Toc15576)

[round函数 5](#_Toc26536)

[1.9 模块 6](#_Toc1035)

[命令import用来导入模块 6](#_Toc10330)

[使用变量来引用函数 6](#_Toc7994)

[floor函数 （类型对象） 6](#_Toc24226)

[int函数 （类型对象） 6](#_Toc1536)

[1.9.1 cmath和复数 6](#_Toc10773)

[cmath 模块（即complex math, 复数） 6](#_Toc20092)

[Sqrt 函数 7](#_Toc20932)

[1.10 保存并执行文件 7](#_Toc20256)

[1.10.3 注释 7](#_Toc30559)

[井号（#） 7](#_Toc15324)

[1.11 字符串 7](#_Toc7207)

[1.11.1 单引号字符串和转义引号 7](#_Toc436)

[转义引号 7](#_Toc20143)

[1.11.3 字符串表示，str和repr 7](#_Toc31754)

[Str函数 （类型） 7](#_Toc20357)

[repr函数 8](#_Toc14620)

[1.11.4 input和raw\_input的比较 8](#_Toc11316)

[raw\_input函数 （Python3里面没有这个函数） 8](#_Toc28547)

[1.11.5 长字符串、原始字符串和Unicode 8](#_Toc2880)

[长字符串 8](#_Toc5312)

[转义换行符 8](#_Toc30980)

[/n换行符 9](#_Toc17690)

[原始字符串 r’ ’ 9](#_Toc17574)

[第二章 列表和元组 9](#_Toc28490)

[2.1 序列概览 9](#_Toc13686)

[六种内建的序列 9](#_Toc2775)

[数据结构容器 9](#_Toc25715)

[列表 10](#_Toc5651)

[2.2通用序列操作 10](#_Toc8908)

[2.2.1索引 10](#_Toc16645)

[2.2.2 分片 11](#_Toc23959)

[步长 12](#_Toc5338)

[2.2.3序列相加 12](#_Toc12310)

[2.2.4 乘法 12](#_Toc9095)

[None、空列表和初始化 13](#_Toc8506)

[2.2.5成员资格 13](#_Toc27068)

[in 运算符 （in 操作符） 13](#_Toc11859)

[2.2.6 长度、最小值和最大值 14](#_Toc7531)

[内建函数len、min和max 14](#_Toc5118)

[2.3 列表：Python的“苦力” 14](#_Toc13664)

[2.3.1 list函数 （list其实是一个类型） 14](#_Toc4056)

[2.3.2 基本的列表操作 15](#_Toc6850)

[元素赋值 15](#_Toc26756)

[删除元素 del语句 15](#_Toc15219)

[分片赋值 15](#_Toc14070)

[2.3.3列表方法 16](#_Toc11438)

[append方法 17](#_Toc817)

[count方法 17](#_Toc18260)

[extend 方法 17](#_Toc6103)

[index方法 18](#_Toc13244)

[insert方法 18](#_Toc19250)

[pop 方法 18](#_Toc18704)

[remove方法 19](#_Toc12713)

[reverse方法 19](#_Toc1140)

[sort方法 19](#_Toc3261)

[高级排序 20](#_Toc16016)

[2.4 元组：不可变序列 21](#_Toc8958)

[元组 21](#_Toc14895)

[2.4.1 tuple函数 （其实是一种类型） 21](#_Toc20634)

[2.4.2 基本元组操作 22](#_Toc15633)

[2.4.3元组意义 22](#_Toc23923)

[第三章 使用字符串 22](#_Toc11305)

[3.1 基本字符串操作 22](#_Toc19683)

[3.2 字符串格式化：精简版 23](#_Toc11989)

[模板字符串 substitute模板方法 24](#_Toc15504)

[3.3 字符串格式化：完整版 24](#_Toc14551)

[3.3.2 字段宽度和精度 25](#_Toc21256)

[3.3.3 符号、对其和用0填充 26](#_Toc6424)

[代码清单3-1 26](#_Toc1008)

[3.4 字符串方法 27](#_Toc23419)

[3.4.1 find 方法 27](#_Toc24341)

[3.4.2 join方法 28](#_Toc32326)

[3.4.3 lower方法 28](#_Toc2210)

[3.4.4 replace方法 29](#_Toc16373)

[3.4.5 split方法 29](#_Toc11320)

[3.4.6 strip方法 29](#_Toc12395)

[3.4.7 translate 方法 30](#_Toc2510)

# 第一章 快速改造：基础知识

## 1.4 数字和表达式

#### 除法运算

##### \_future\_

>>> from \_future\_ import div ision

让Python执行普通的除法

书本第8 - 9页

python 3.x里面，// 是地板除，/ 不管两边是不是整数得到的都是小数。

四舍五入请用

>>> round(5/3,2)

1.67

#### 幂（乘方）运算符

>>> 2 \*\* 3

8

>>> -3 \*\* 2

-9

>>> (-3) \*\* 2

9

幂运算符比取反（一元减运算符）的优先级要高，所以-3\*\*2等同于-(3\*\*2)。如果想要计算(-3)\*\*2,就需要显式说明

书本第9页

### 1.4.2 十六进制和八进制

在python中，十六进制书写方式：

>>> 0xAF

175

八进制是：

>>> 010

8

书本第10页

## 1.7 获取用户输入

#### input函数

获取控制台的输入

input() 在对待纯数字输入时具有自己的特性，它返回所输入的数字的类型（ int, float ）。

Python3里input()默认收到的是str类型

<http://www.runoob.com/python/python-func-input.html>

书本第12页

## 1.8 函数

#### pow 函数

把pow等标准函数称之为内建函数

计算乘方

格式：

>>> pow(2,3)

8

<http://www.runoob.com/python/func-number-pow.html>

书本第13页

#### abs函数

可以得到数的绝对值

>>> abs(-10)

10

书本第13页

#### round函数

把浮点数四舍五入到最为接近的整数值

>>> round(1.0/2.0)

1.0

书本第13页

## 1.9 模块

#### 命令import用来导入模块

模块的使用方式：

>>> import math #使用import导入模块math

>>> math.floor(32.9) #按照“模块 . 函数”的格式使用这个模块的函数

32.0

使用**from模块import函数**这种形式的import命令后，就可以直接使用函数，而不用模块名作为前缀

书本第14页

#### 使用变量来引用函数

可以使用变量来引用函数（或者Python中大多数的对象）。比如，通过foo=math.sqrt进行赋值，然后就可以用foo来计算平方根了：foo(4)的结果为2.0。

书本第14页

#### floor函数 （类型对象）

返回数字的下舍整数

>>> import math

>>> math.floor(32.9)

32.0

书本第14页

#### int函数 （类型对象）

把参数转化为整数，会自动向下取整

>>> int(math.floor(32.9))

32

书本第14页

### 1.9.1 cmath和复数

#### cmath 模块（即complex math, 复数）

用来处理虚数（以及复数，即实数和虚数之和）

为了防止命名冲突应该尽量使用普通的import

书本第15页

#### Sqrt 函数

>>> import cmath

>>> cmath.sprt(-1)

1j

1j是个虚数，虚数均以j（或者J）结尾

书本第15页

## 1.10 保存并执行文件

### 1.10.3 注释

#### 井号（#）

#右侧的一切内容都会被忽略

书本第18页

## 1.11 字符串

### 1.11.1 单引号字符串和转义引号

#### 转义引号

\’ \” 转义引号

书本第19 - 20页

### 1.11.3 字符串表示，str和repr

#### Str函数 （类型）

会把值转换成合理形式的字符串，以便用户理解

>>> print(str(“Hello, word!”))

Hello, word!

>>> print(str(10000L))

10000

书本第21页

#### repr函数

是将Python值转换为合法的Python表达式

>>> print(repr(“Hello, word!”))

‘Hello, word!’

>>> print(repr(10000L))

10000L

书本第21页

### 1.11.4 input和raw\_input的比较

#### raw\_input函数 （Python3里面没有这个函数）

此函数会把所有的输入当作原始数据（raw data），然后将其放入字符串

除非对input有特别的需要，否则就应该尽可能的使用raw\_input函数

**Python3里面没有这个函数**

书本第22页

### 1.11.5 长字符串、原始字符串和Unicode

#### 长字符串

使用三个引号（或三个双引号）代替普通引号可以跨越多行来写字符串

print(‘’’This is a very long string.

It continues here.

And it’s not over yet.

“Hello, word!”

Still here.)

#### 转义换行符

普通字符串的最后一个如果是\（反斜线），那么，换行符本身就被转义了，会被忽略

print(“Hello, \

word!”)

这个用法也适用于表达式和语句：

>>> 1 + 2 + \

. . . 4 + 5

12

>>> print\

. . . (‘Hello, world’)

Hello, world

书本第22页

#### /n换行符

书本第23页

#### 原始字符串 r’ ’

原始字符串以r开头。原始字符串不会把反斜线（\）当做特殊字符。在原始字符串中输入的每个字符都会与书写的方式保持一致。原始字符串的结尾不能是 反斜线（\） 。如果一定要以 反斜线（\） 结尾，就讲反斜线作为一个独立的字符串来处理。

>>> print(r’C:\Program Files\foo\bar’ ‘\\’)

C:\Program Files\foo\bar\

书本第23 - 24页

# 列表和元组

## 2.1 序列概览

#### 六种内建的序列

六种内建的序列：列表、元组、字符串、Unicode字符串、buffer对象和xrange对象。

列表和元组的主要区别在于，列表可以更改，而元组不能。

书本第26页

#### 数据结构容器

书本第27页

#### 列表

格式：

>>> a = [‘a’, ‘abc’, 1]

>>> b = [‘b’, ‘bcd’, 2]

>>> c = [a, b, ‘a’, 1]

>>> c

[[‘a’, ‘abc’, 1], [‘b’, ‘bcd’, 2], ‘a’, 1]

书本第26 - 27页

## 2.2通用序列操作

### 2.2.1索引

从0开始递增, 使用负数索引时，Python会从右边开始计数

访问方式：

>>> a = ‘hello’

>>> a[0]

‘h’

>>> a[-1]

‘o’

字符串字面值（就此而言，其他序列字面量亦可）可以直接使用索引，而不需要一个变量引用它们。

>>> ‘hello’[1]

E

如果一个函数调用返回一个序列，那么可以直接对返回结果进行索引操作。

>>> a = input(‘year: ’)[3]

year: 2005

>> a

‘5’

示例程序，要求输入年、月（1~12的数字）、日（1~31），然后打印出相应日期的月份名称。

months = [

'January',

'February',

'March',

'April',

'May',

'June',

'July',

'August',

'Septemper',

'October',

'November',

'December'

]

#以1~31的数字作为结尾的列表

endings = ['st','nd','rd'] + 17 \* ['th']\

+ ['st','nd','rd'] + 7 \* ['th']\

+ ['st']

year = raw\_input('Year: ')

month = raw\_input('Month(1-12): ')

day = raw\_input('Day(1-31): ')

month\_number = int(month)

day\_number = int(day)

#记得要将月份和天数减1，以获得正确的索引

month\_name = months[month\_number-1]

ordinal = day + endings[day\_number-1]

print month\_name + ' '+ ordinal + '.' + year

程序执行的一部分结果

Year: 2015

Month(1-12): 9

Day(1-31): 30

Septemper 30th.2015

书本第27 - 28页

### 2.2.2 分片

以两个索引为边界，第一个包括在分片里面，第二个不包括

格式：

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

>>> numbers[3:6]

[4, 5, 6]

>>> numbers[-3:-1]

[8, 9]

>>> numbers[1:-3]

[2, 3, 4, 5, 6, 7]

>>> numbers[-3:]

[8, 9, 10]

>>> numbers[:3]

[1, 2, 3]

>>> numbers[:]

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

书本第29页

#### 步长

步长不能为0

格式：

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

>>> numbers[0:3:1]

[1, 2, 3]

>>> numbers[0:3:2]

[1, 3]

>>> numbers[3:0:-1]

[3, 2, 1]

>>> numbers[3::-1]

[3, 2, 1]

>>> numbers[:3:2]

[1, 3]

>>> numbers[::2]

[1, 3, 5, 7, 9]

书本第29 - 31页

### 2.2.3序列相加

使用加运算符可以进行序列的连接操作：

>>> [1, 2, 3] + [4, 5, 6]

[1, 2, 3, 4, 5, 6]

>>> ‘Hello. ’ + ‘world!’

‘Hello, world!’

列表和字符串无法连接在一起。两种同类型的序列才能进行连接操作。

### 2.2.4 乘法

用数字x乘以一个序列会生成新的序列，在新的序列中，原来的序列将被重复x遍。

>>> ‘python’ \* 5

‘pythonpythonpythonpythonpython’

>>> [42] \* 10

[42, 42, 42, 42, 42, 42, 42, 42, 42, 42]

序列（字符串）乘法示例：

sentence = input("Sentence: ")

screen\_width = 80

text\_width = len(sentence)

box\_width = text\_width + 6

left\_margin = (screen\_width - box\_width) // 2

print()

print(' ' \* left\_margin + '+' + '-' \* (box\_width - 2) + '+')

print(' ' \* (left\_margin + 2) + '|' + ' ' \* text\_width + '|')

print(' ' \* (left\_margin + 2) + '|' + sentence + '|')

print(' ' \* (left\_margin + 2) + '|' + ' ' \* text\_width + '|')

print(' ' \* left\_margin + '+' + '-' \* (box\_width - 2) + '+')

print()

运行效果：

Sentence: 123456789

+-------------+

| |

|123456789|

| |

+-------------+

#### None、空列表和初始化

空列表可以用两个中括号进行表示（[]）——里面什么都没有。

None是一个Python的內键值，它的确切含义是“这里什么都没有”。

如果要初始化一个长度为10的列表：

>>> Sequence = [None] \* 10

>>> Sequence

[None, None, None, None, None, None, None, None, None, None]

书本第31 - 32页

### 2.2.5成员资格

为了检查一个值是否在序列中，可以使用in运算符。

#### in 运算符 （in 操作符）

格式：

>>> permissions = ‘rw’

>>> ‘w’ in permissions

True

>>> ‘xx’ in permissions

False

>>> users = [‘mlh’, ‘foo’, ‘bar’]

>>> input(‘Enter your user name: ’) in users

Enter your user name: mlh

True

书本第32 - 33页

### 2.2.6 长度、最小值和最大值

#### 内建函数len、min和max

Len返回序列中所包含的元素数量

min 返回序列中最小的元素

max 返回序列中最大的元素

格式：

>>> numbers = [100, 34, 678]

>>> len(numbers)

3

>>> min(numbers)

34

>>> max(numbers)

678

>>> max(2,3)

3

>>> min(9, 3, 2, 5)

2

书本第33 - 34页

## 2.3 列表：Python的“苦力”

### 2.3.1 list函数 （list其实是一个类型）

将序列转换成列表

格式：

>>> list(‘hello’)

[‘h’, ‘e’, ‘l’, ‘l’, ‘o’]

书本第34页

### 2.3.2 基本的列表操作

#### 元素赋值

格式：

>>> x= [1, 1, 1]

>>> x[1] = 2

>>> x

[1, 2, 1]

书本第34 - 35页

#### 删除元素 del语句

del语句，删除元素后，列表的长度也相应的缩短。出了删除列表中的元素，del还可用于删除其他元素。

格式：

>>> names = [‘Alice’, ‘Beth’, ‘Cecil’, ‘Dee-Dee’, ‘Earl’]

>>> del names[2]

>>> names

[‘Alice’, ‘Beth’, ‘Dee-Dee’, ‘Earl’]

书本第35页

#### 分片赋值

格式：

>>> name=list("Prel")  
>>> name  
['P', 'r', 'e', 'l']  
>>> name[2:]=list("ar")  
>>> name  
['P', 'r', 'a', 'r']

使用分片赋值的时候，可以使用与源序列不等长的序列将分片替换：  
>>> name=list("perl")  
>>> name  
['p', 'e', 'r', 'l']  
>>> name[1:]=list("ython")  
>>> name  
['p', 'y', 't', 'h', 'o', 'n']

分片赋值语句可以在不需要替换任何原有元素的情况下插入新的元素

>>> numbers=[1,5]  
>>> numbers[1:1]=range(2, 3, 4)  
>>> numbers  
[1, 2, 3, 4, 5]

这里看起来是“替换”了一个空的分片，但是实际上是插入了一个新的序列。因此，通过分片赋值来删除元素也是可行的。

>>> number=[1, 2, 3, 4, 5]

>>> number

[1, 2, 3, 4, 5]

>>> number[1:4]=[]

>>> number

[1,5]

如果说步长为1以外的数字，那么分区内要替换几个数值等号后面就要写几个数值。

>>> numbers = list('1111111111')

>>> numbers[::2] = [‘2’]\*5

>>> print(numbers)

['2', '1', '2', '1', '2', '1', '2', '1', '2', '1']

书本第34 - 36页

### 2.3.3列表方法

方法是一个与某些对象有紧密联系的函数，对象可能是列表、数字，也可能是字符串或者其他类型的对象。

格式：

对象.方法(参数)

*网络资料：*

*Python列表操作的函数和方法*

*列表操作包含以下函数:*

*1、cmp(list1, list2)：比较两个列表的元素*

*2、len(list)：列表元素个数*

*3、max(list)：返回列表元素最大值*

*4、min(list)：返回列表元素最小值*

*5、list(seq)：将元组转换为列表*

*列表操作包含以下方法:*

*1、list.append(obj)：在列表末尾添加新的对象*

*2、list.count(obj)：统计某个元素在列表中出现的次数*

*3、list.extend(seq)：在列表末尾一次性追加另一个序列中的多个值（用新列表扩展原来的列表）*

*4、list.index(obj)：从列表中找出某个值第一个匹配项的索引位置*

*5、list.insert(index, obj)：将对象插入列表*

*6、list.pop(obj=list[-1])：移除列表中的一个元素（默认最后一个元素），并且返回该元素的值*

*7、list.remove(obj)：移除列表中某个值的第一个匹配项*

*8、list.reverse()：反向列表中元素*

*9、list.sort([func])：对原列表进行排序*

书本第36页

#### append方法

用于在列表末尾追加新的对象

>>>1st = [1, 2, 3]

>>>1st.append(4)

>>>1st

[1, 2, 3, 4]

只能一次添加一个元素

他不是简单地返回一个修改过的新列表，而是直接修改原来的列表。

书本第36页

#### count方法

统计某个元素在列表中出现的次数：

>>> [‘to’, ‘be’, ‘or’, ‘not’, ‘to’, ‘be’].count(‘to’)

2

>>> x = [[1, 2], 1, 1, [2, 1, [1, 2]]]

>>> x.count(1)

2

>>> x.count([1, 2])

1

书本第36页

#### extend 方法

在列表的末尾追加另一个序列的中的多个值。换句话说，可以用新的列表拓展原有列表。

>>> a = [1, 2, 3]

>>> b = [4, 5, 6]

>>> a.extend(b)

>>> a

[1, 2, 3, 4, 5, 6]

extend方法修改了被扩展的序列，而原始的连接操作（+）则不然，它会返回一个全新的列表

可以使用分片赋值来实现相同的效果

>>> a = [1, 2, 3]

>>> b = [4, 5, 6]

>>> a[len(a):] = b

>>> a

[1, 2, 3, 4, 5, 6]

虽然这么做是可行的，但是代码的可读性没有使用extend好。

书本第36 - 37页

#### index方法

从列表中找出某个值第一个（注意是第一个）匹配项的索引位置

>>> a=['one', 'two']

>>> a.index('one')

0

>>> a.index(‘three’)

File "<stdin>", line 1

a.index(‘three’)

^

SyntaxError: invalid character in identifier

当搜索three时，就会发生一个异常，因为这个单词没有被找到。

书本第37页

#### insert方法

将对象插入到列表中。

>>> a=[1, 2]

>>> a.insert(1, 'insert')

>>> a

[1, 'insert', 2]   
可以使用分片赋值来实现相同的效果

>>> a=[1, 2]

>>> a[1:1] = [‘insert’]

>>> a

[1, 'insert', 2]

虽然这么做是可行的，但是代码的可读性没有使用insert好。

书本第37 - 38页

#### pop 方法

移除列表中的一个元素（默认是最后一个）。**是列表操作中唯一一个，既能修改列表，又能返回元素值的列表方法。**

>>> x=[1, 2, 3]

>>> x.pop()

3

>>> x

[1, 2]

>>> x.pop(0)

1

>>> x

[2]

使用pop方法可以实现一个常见的数据结构——栈。可以使用append方法来入栈，使用pop方法来出栈。

书本第38页

#### remove方法

移除列表中某个值的第一个匹配项。它没有返回值。

>>> x=[1, 2, 3]

>>> x.remove(2)

>>> x

[1, 3]

当搜索一个不存在的值时，就会发生一个异常，因为这个单词没有被找到。

书本第38 - 39页

#### reverse方法

将列表中的元素反向存放。它没有返回值。它返回的是一个迭代器（iterator）对象。可以用list函数把返回的对象转化为列表。

>>> x=[1, 2, 3]

>>> x.reverse()

>>> x

[3, 2, 1]

书本第39页

#### sort方法

在改变原有列表的情况下，对列表进行排序。

>>> x = [4, 6, 2, 1, 7, 9]

>>> x.sort()

>>> x

[1, 2, 4, 6, 7, 9]

当用户需要一个排列好序的列表副本，又要保留原有列表不变的时候，需要先把x的副本赋值给y，然后在对y进行排序。

>>> x = [4, 6, 2, 1, 7, 9]

>>> y = x[:]

>>> y.sort()

>>> x

[4, 6, 2, 1, 7, 9]

>>> y

[1, 2, 4, 6, 7, 9]

**调用x[:]得到的是包含了x所有元素的分片，这是一种很有效率的赋值整个列表的方法。只是简单的把x赋值给y是没有用的，因为这样就让x和y都指向同一个列表了**

>>> y = x

>>> y.sort()

>>> x

[1, 2, 4, 6, 7, 9]

>>> y

[1, 2, 4, 6, 7, 9]

还有一种获取已排序的列表副本的方法是，使用sorted函数：

>>> x = [4, 6, 2, 1, 7, 9]

>>> y = sorted(x)

>>> x

[4, 6, 2, 1, 7, 9]

>>> y

[1, 2, 4, 6, 7, 9]

sorted函数可以用于任何序列，却总是返回一个列表。

>>> sorted(‘Python’)

[‘h’, ‘n’, ‘o’, ‘P’, ‘t’, ‘y’]

书本第39 - 40页

#### 高级排序

sort 方法可以改变排序方式

内建函数cmp可以提供给sort方法作为参数

>>> cmp(42, 32)

1

>>> cmp(99, 100)

-1

>>> cmp(10, 10)

0

>>> numbers = [5, 2, 9, 7]

>>> numbers.sort(cmp)

>>> numbers

[2, 5, 7, 9]

sort方法有另外两个可选参数——key和reverse。如果要使用它，就要通过名字来指定（这叫关键字参数）参数key与参数cmp类似--必须提供一个在排序过程中使用的函数。然而，该函数并不是直接用来确定对象的大小，而是为每个元素创建一个键，然后所有元素根据键来排序。因此，如果要根据元素的长度进行排序，那么可以使用len作为键函数：

>>> x = ['aardvark', 'abalone', 'acme', 'add', 'aerate']

>>> x.sort(key=len)

>>> x

['add', 'acme', 'aerate', 'abalone', 'aardvark']

另一个关键字参数reverse是简单的布尔值（True或者是False），用来指明列表是否要进行反向排序。

>>> x = [4, 6, 2, 1, 7, 9]

>>> x.sort(reverse=True)

>>> x

[9, 7, 6, 4, 2, 1]

cmp、key、reverse参数都可用于sorted函数。在多数情况下，为cmp或key提供自定义函数是非常有用的。

书本第40 - 41页

## 2.4 元组：不可变序列

#### 元组

元组和列表一样，也是一种序列。区别就在于元组不能修改（字符串也是如此）。创建元组的方法很简单：只要用逗号分隔一些值，就自动创建了元组。

>>> 1, 2, 3

(1, 2, 3)

元组也是（大部分时候是）通过圆括号括起来的

>>> (1, 2, 3)

(1, 2, 3)

空元组可以用没有包含内容的两个圆括号来表示

>>> ()

()

创建包含一个值的元组。必须加个逗号，即使只有有一个值：

>>> 42

42

>>> 42,

(42,)

>>> (42,)

(42,)

最后两个生成了一个长度为1的元组，而第一个例子根本不是元组。

>>> 3\*(40+2)

126

>>> 3\*(40+2,)

(42, 42, 42)

书本第41 - 42页

### 2.4.1 tuple函数 （其实是一种类型）

以一个序列为参数并把它转换为元组。如果原来就是元组，就原样返回。

>>> tuple([1, 2, 3])

(1, 2, 3)

>>> tuple(‘abc’)

(‘a’, ‘b’, ‘c’)

>>> tuple((1, 2, 3))

(1, 2, 3)

书本第42页

### 2.4.2 基本元组操作

元组其实并不复杂——除了创建元组和访问元组元素之外，也没有太多其他操作，可以参照其他类型的序列来实现：

>>> x = 1, 3, 5

>>> x[1]

3

>>> x[0:2]

(1, 3)

>>> x[0:1]

(1,)

元组的分片还是元组，就像列表的分片还是列表一样。

书本第42页

### 2.4.3元组意义

由于元组不能改变其中的内容，那么当不能改变元素内容的时候是不是只能用列表?大部分情况是这样的。但是，在某些情况下，元组是不可替代的：

* 元组可以在映射(和集合的成员)中当作键使用——而列表则不行;
* 元组作为很多内建函数和方法的返回值存在，也就是说你必须对元组进行处理。只要不尝试修改元组，那么,"处理"元组在绝大多数情况下就是把它们当作列表来进行操作(除非需要使用一些元组没有的方法，如index和count)。

# 使用字符串

## 3.1 基本字符串操作

所有标准的序列操作(索引、分片、乘法、求长度、最小值、最大值和判断成员资格等)对于字符串同样适用。但是，有一点非常重要的是，字符串都是不可改变的，比如：

>>> str = "hello world!"

>>> str[-6:] = "guest!"

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'str' object does not support item assignment

## 3.2 字符串格式化：精简版

在%的左侧放置一个字符串（格式化字符串），而右侧则放置希望格式化的值。可以使用一个值，如一个字符串或者数字，也可以使用多个值的元组或者字典。一般情况下使用元组:

>>> format = “hello, %s . %s enough for ya?”

>>> values = (‘world’, ’hot’)

>>> print(format % values)

hello, world . hot enough for ya?

如果使用列表或其他序列代替元组，那么序列将会被解释为一个值。只有元组和字典可以格式化一个以上的值：

>>> format = "hello,%s. %s are brave!"

>>> values = [123]

>>> print(format % values)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: not enough arguments for format string

>>> format = "hello,%s. are brave!"

>>> print(format % values)

hello,[123]. are brave!

"%"是Python风格的字符串格式化操作符，非常类似C语言里的printf()函数的字符串格式化（C语言中也是使用%）。

下面整理了一下Python中字符串格式化符合：

|  |  |
| --- | --- |
| 格式化符号 | 说明 |
| %c | 转换成字符（ASCII 码值，或者长度为一的字符串） |
| %r | 优先用repr()函数进行字符串转换 |
| %s | 优先用str()函数进行字符串转换 |
| %d / %i | 转成有符号十进制数 |
| %u | 转成无符号十进制数 |
| %o | 转成无符号八进制数 |
| %x / %X | 转成无符号十六进制数（x / X 代表转换后的十六进制字符的大小写） |
| %e / %E | 转成科学计数法（e / E控制输出e / E） |
| %f / %F | 转成浮点数（小数部分自然截断），还可以提供精度：一个句点加上希望保留的小数位数，如：%.3f |
| %g / %G | 如果指数大于-4或小于精度值则和%e / %E相同，其他情况同%f / %F |
| %% | 输出% （格式化字符串里面包括百分号，那么必须使用%%） |

#### 模板字符串 substitute模板方法

string模块提供另外一种格式化值得方法：模板字符串。

如下所示，substitute模板方法会用传递进来的关键字参数foo替换字符串中的$foo。

方法safe\_substitute不会因缺少值或者不正确使用%字符而出错

>>> from string import Template

>>> s = Template('$x, glorious $x!')

>>> s.substitute(x='slurm')

'slurm, glorious slurm!'

>>> s.substitute(x="xxx")

'xxx, glorious xxx!'

# 如果只是替换单词一部分，那个参数名就必须用大括号括起来

>>> s = Template("It's ${x}tastic!")

>>> s.substitute(x="slurm")

"It's slurmtastic!"

>>> s.substitute(x="xxx")

"It's xxxtastic!"

# 使用$$插入美元符号

>>> s = Template("Make $$ selling $x!")

>>> s.substitute(x="slurm")

'Make $ selling slurm!'

>>> s.substitute(x="xxx")

'Make $ selling xxx!'

# 除关键字参数外，还可以使用字典变量提供值/名称对

>>> s = Template('A $thing must never $action.')

>>> d = {}

>>> d['thing'] = 'gentleman'

>>> d['action'] = 'show his socks'

>>> s.substitute(d)

'A gentleman must never show his socks.'

书本第44 - 45页

## 3.3 字符串格式化：完整版

格式化操作符的右操作数可以是任意类型如果是元组或者映射类型（如字典），那么字符串格式化将会有所不同。

如果右操作符是元组的话，其中每一个元素都会被单独格式化，每一个值都需要一个对应的转换说明符

如果需要转换的元组作为转换表达式的一部分存在，那么必须将它用圆括号括起来避免出错

>>> '%s plus %s equals %s' % (1,2,3)

'1 plus 2 equals 3'

基本的转换说明符包括以下部分。注意，这些项的顺序是至关紧要的：

1）**%字符：**标记转换说明符的开始。

2）**转换标志（可选）**：-表示左对齐；+表示在转换值之前要加上正负号；“”（空白字符）表示正数之前保留空格；0表示转换值若位数不够则用0填补。

3）**最小字段宽度（可选）**：转换后的字符串至少应该具有该值指定的宽度。如果是\*，则宽度会从值元组中读出。

4）**点（.）后跟精度值（可选）**：如果转换的是实数，精度值就表示出现在小数点后的位数。如果转换的是字符串，那么该数字就表示最大字段宽度。如果是\*，那么精度将会从元组中读出。

5）转换类型

|  |  |
| --- | --- |
| 字符串格式化转换类型 | |
| 转换类型 | 含义 |
| d, i | 带符号的十进制整数 |
| o | 不带符号的八进制 |
| u | 不带符号的十进制 |
| x | 不带符号的十六进制（小写） |
| X | 不带符号的十六进制（大写） |
| e | 科学计数法表示的浮点数（小写） |
| E | 科学计数法表示的浮点数（大写） |
| f, F | 十进制浮点数 |
| g | 如果指数大于-4或者小于精度值则和e相同 其它情况与f相同 |
| G | 如果指数大于-4或者小于精度值则和E相同 其它情况与F相同 |
| C | 单字符（接受整数或者单字符字符串） |
| r | 字符串（使用repr转换任意Python对象） |
| s | 字符串（使用str转换任意Python对象） |

书本第46页

### 3.3.2 字段宽度和精度

转换说明符可以包括字段宽度和精度。字段宽度是转换后的值所保留的最小字符个数，精度则是结果中应该包含的小数位数。

两个参数都是整数（字段宽度.精度），通过点号（.）分隔。虽然两个都是可选的参数，但如果只给出精度，就必须包含点号：

>>> import math

>>> '%10f' % math.pi #字段宽10

' 3.141593'

>>> '%10.2f' % math.pi #字段宽10，精度2

' 3.14'

>>> '%.3f' % math.pi #精度3

'3.142'

>>> '%.5s' % 'abcdefghijk' #最大字段宽度

'abcde'

>>> '%.\*s' % (5, 'abcdefghijk') #使用\*号在元组参数中读出

'abcde'

书本第47页

### 3.3.3 符号、对其和用0填充

在字段宽度和精度值之前还可以放置一个“标志”，该标志可以是零、加号、减号或空格。

零表示数字将会用0进行填充：

>>> from math import pi

>>> ‘%010.2f’% pi #字段宽度为10，并使用0进行填充

‘0000003.14’

注意，在010中开头的那个0不意味着字段宽度说明符为八进制数，他就是一个普通的Python数值。

使用减号（-）用来左对齐数值：

>>> ‘%-10.2f’% pi

‘3.14 ’

空格（“ ”）意味着在正数前加上空格。这在需要对齐正负数时会很有用：

>>> print(‘% 5d’% 10) + ‘\n’ + (‘% 5d’ % -10)

10

- 10

使用加号（+）表示不管是正数还是负数都标出符号（有对齐的作用）：

>>> print(‘%+5d’% 10) + ‘\n’ + (‘%+5d’ % -10)

+10

- 10

书本第47 - 48页

#### 代码清单3-1

字符串格式化示例：

#使用给定的宽度打印格式化之后的价格列表

width = int(input('Please enter width: '))

price\_width = 10

item\_width = width - price\_width

header\_format = '%-\*s%\*s'

format = '%-\*s%\*.2f'

print ('=' \* width)

print (header\_format % (item\_width, 'Item', price\_width, 'Prce'))

print ('-' \* width)

print (format % (item\_width, 'Apples',price\_width,0.4))

print (format % (item\_width, 'Pears',price\_width,0.5))

print (format % (item\_width, 'Cantaloupes',price\_width,1.92))

print (format % (item\_width, 'Dried Apricots(16 oz.)',price\_width,8))

print (format % (item\_width, 'Prunes(4 lbs.)',price\_width,12))

书本第48 - 49页

## 3.4 字符串方法

字符串方法完全来源于string模块，但是这个模块还包括一些不能作为字符串方法使用的常量和函数。下面是一些有用的字符串常量。

string.digits：包含数字0~9的字符串

string.ascii\_letters：包含所有字母（大写或小写）的字符串

string.lowercase：包含所有小写字母的字符串

string.printable：包含所有可打印字符的字符串

string.punctuation：包含所有标点的字符串

string.uppercase：包含所有大写字母的字符串

书本第49页

#### 3.4.1 find 方法

find方法可以在一个较长的字符串中查找子串。它返回子串所在位置的最左端索引。如果没有找到则返回-1。

>>> ‘With a moo-moo here, and a moo-moo there’.find(‘moo’)

7

>>> title = "Monty Python's Flying Circus"

>>> title.find('Monty')

0

>>> title.find('Python')

6

>>> title.find('Flying')

15

>>> title.find('Zirquss')

-1

注：字符串的find方法并不返回布尔值，如果返回的是0，则证明在索引0位置找到了子串。

这个方法还有可选的第二第三参数。第二和第三个参数可以指定起始点和结束点。

>>> subject = '$$$ Get rich now!!! $$$'

>>> subject.find('$$$')

0

>>> subject.find('$$$', 1) #只提供起点

20

>>> subject.find('!!!')

16

>>> subject.find('!!!', 0, 16) #提供起始点和结束点

-1

注意，由起始和终止值指定的范围（第二个和第三个参数）包含第一个索引，但不包含第二个索引。这在Python中是个惯例。

**类似的方法还有：rfind,index,rindex,count,startwith,endwith。**

书本第49 - 50页

#### 3.4.2 join方法

join方法是非常重要的字符串方法，他是split方法的逆方法，用来连接序列中的元素。

>>> seq = ['1','2','3','4','5']

>>> sep = '+'

>>> sep.join(seq)

'1+2+3+4+5'

>>> dirs = '','usr','bin','env'

>>> '/'.join(dirs)

'/usr/bin/env'

>>> print('C:' + '\\'.join(dirs))

C:\usr\bin\env

需要连接的序列元素都必须是字符串。最后两个例子中使用了目录的列表，而在格式化时，根据UNIX和DOS/Windows的约定，使用了不同的分隔符号。

书本第50 - 51页

#### 3.4.3 lower方法

lower方法返回字符串的小写字母版。

>>> "ABcdEf".lower()

'abcdef'

>>> "FFff".lower()

'ffff'

当判断某一个字符串，想要有不区分大小写的功能，那么就可以使用lower这个函数方法。

**相关方法：islower、capitalize、swapcase、title、istitle、upper、isupper**

书本第51页

##### title方法

title方法将字符串转换为标题--也就是所有的首字母大写。但是它使用的单词划分可能会得到并不自然的结果：

>>> "that's all folks".title()

That'S All Folks

所以再介绍另外一个string模块的capwords函数

##### string模块的capwords函数

将首字母大写

>>> import string

>>> string.capwords("that's all, folks")

That's All, Folks

#### 3.4.4 replace方法

replace方法返回某字符串的所有匹配项均被替换之后得到的字符串

replace方法有两个参数，第一个参数是要被替换的字符串，第二个参数代表用该字符串取代原字符串中匹配的字符串：

>>> 'This is not issue!'.replace('is','cover')

'Thcover cover not coversue!'

>>> 'hello word!'.replace('word','world')

'hello world!'

相关方法：expandtabs（把字符串中的 tab 符号('\t')转为空格）

书本第51 - 52页

#### 3.4.5 split方法

split方法是非常重要的字符串方法，他是join方法的逆方法，用来将字符串分隔成序列。

split()通过指定分隔符对字符串进行切片，如果参数num 有指定值，则仅分隔 num 个子字符串

格式：

str.split(str="", num=string.count(str)).

参数：

str -- 分隔符，默认为所有的空字符，包括空格、换行(\n)、制表符(\t)等。

num -- 分割次数。

返回值

返回分割后的字符串列表。

实例

str = "Line1-abcdef \nLine2-abc \nLine4-abcd"

print (str.split( ))

print (str.split(' ', 1 ))

以上实例输出结果如下：

['Line1-abcdef', 'Line2-abc', 'Line4-abcd']

['Line1-abcdef', '\nLine2-abc \nLine4-abcd']

如果不提供任何分隔符，程序会将所有空格作为分隔符（空格、制表、换行等）。

**相关方法：rsplit（**同split，但是计数时是从右往左**）,splitlines（**按照行('\r', '\r\n', \n')分隔，返回一个包含各行作为元素的列表**）**

书本第52页

#### 3.4.6 strip方法

strip方法返回去除字符串两侧(不包括内部)空格的字符串

>>> " hehe,my mood is good! ".strip()

'hehe,my mood is good!'

>>> " front input is useless !! ".strip()

'front input is useless !!'

它和lower方法一起使用的话就可以很方便的对比输入的和存储的值。

也可以使用strip方法指定需要去除的字符串，将它们列为参数即可（只要字符串上的字符在参数里的，就删除掉）。比如：

>>> '\*\*\*\*\* spam comment for blog!\*\*\*!'.strip(' \*!')

'spam comment for blog'

**相关方法：lstrip(**将开头处位于参数的字符删除**)、rstrip(**将结尾处位于参数的字符删除**)**

书本第52页

#### 3.4.7 translate 方法

网络资料：<https://coderschool.cn/1548.html>

translate方法和replace方法一样，可以替换字符串中的某些部分，但是和前者不同的是，translate方法只处理单个字符。它的优势在于可以同时进行多个替换，有些时候比replace效率高得多。

translate() 方法根据参数table给出的表(包含 ASCII字符集中256 个字符 )转换字符串的字符, 要过滤掉的字符在table参数中指定。

格式：

str.translate(table);

参数

table -- 转换表，转换表是通过**[maketrans方法](#_maketrans() 方法)**转换而来。

以下实例展示了 translate()函数的使用方法：

>>> map = str.maketrans('fff','ddd','12')

>>> s = "fffdd1234"

>>> s.translate(map)

'ddddd34'

书本第52 - 54页

##### maketrans() 方法

maketrans() 方法用于创建字符映射的转换表，对于接受两个参数的最简单的调用方式，第一个参数是字符串，表示需要转换的字符，第二个参数也是字符串表示转换的目标，第三个参数为**translate**要删除的字符。

两个字符串的长度必须相同，为一一对应的关系。

注：Python3.4已经没有string.maketrans()了，取而代之的是内建函数: bytearray.maketrans()、bytes.maketrans()、str.maketrans()

格式：

str.maketrans(x[, y[, z]])

参数：

x -- 字符串中要替代的字符组成的字符串。

y -- 相应的映射字符的字符串。

z -- **translate**要删除的字符

如果只有一个参数就一定得是字典

在python2.X下

>>> import string

>>> map = string.maketrans('123', 'abc')

>>> s = '54321123789'

#除了转换，还要删除字符串中的字符'7','8'

>>> s.translate(map, '78')

#转换后的字符串没有字符'7','8'

'54cbaabc9'

在python3.X下

>>> map = str.maketrans('fff','ddd','12')

>>> s = "fffdd1234"

>>> s.translate(map)

'ddddd34'

##### 非英语字符串的问题

书本第53 - 54页

# 字典：当索引不好用时

## 4.1 字典的使用

字典可以快速找到某个特定的词语（键），从而找到他的定义（值）。

某些情况下，字典比列表更加适用，比如：

* 表征游戏棋盘的状态，每个键都是由坐标值组成的元组。
* 存储文件修改次数，用文件名作为建。
* 数字电话/地址薄。

书本第55页

## 4.2 创建和使用字典

键和值之间用冒号（:）隔开，项之间用逗号（,）隔开，整个字典用一对大括号括起来。

**字典的键是唯一的（其他类型的映射也是这样），而值并不唯一。**

格式：

>>> phonebook={'Jack':'1234','Peter':'123','Rose':'4321'}

名字为键，电话号码为值

空字典由两个大括号组成{};

书本第56页

### 4.2.1 dict函数 （类型）

除了前面讲的，可以用大括号创建字典外，还可以用dict函数，通过其它映射(比如其它字典)或者(键，值)这样的序列对建立字典。dict函数不是真正的函数。它是个类型，像list函数、tuple函数和str一样。

>>> items = [('name','gumby'),('age',24)]

>>> d = dict(items)

>>> d

{'name': 'gumby', 'age': 24}

>>> d['name']

'gumby'

>>> d['age']

24

dict函数也可以通过关键字参数赋值的形式来创建字典，如下所示：

>>> d = dict(name="psz",age=25)

>>> d

{'name': 'psz', 'age': 25}

>>> d['age']

25

创建字典是dict函数最有用的功能了，但是还能以映射作为dict函数的参数，以建立其项与映射相同的字典。

>>> d

{'name': 'psz', 'age': 25}

>>> fd = dict(d)

>>> fd

{'name': 'psz', 'age': 25}

>>> fd['age']

25

>>> d['age']

25

如果不带任何参数，则dict函数返回一个新的空字典，就像list、tuple等函数一样。比如：

>>> dd = dict()

>>> dd

{}

>>> items = []

>>> ff = dict(items)

>>> ff

{}

书本第56 - 57页

### 4.2.2 基本字典操作

字典的基本行为在很多方面与序列(sequence)相似，下面是字典几个常用的操作方法,其中d代表某个字典：

* len(d)返回d中项(键-值对)的数量;
* d[k]返回关联到键k上的值;
* d[k] = v将值v关联到键k上;
* del d[k]删除键为k的项;
* k in d检查d中是否含有键为k的项。返回布尔型。

下面给出一些实例：

>>> d = dict(name="psz",age=25)

>>> d

{'name': 'psz', 'age': 25}

>>> len(d)

2

>>> d['age']

25

>>> del d['name']

>>> d

{'age': 25}

>>> d['name'] = 'peter'

>>> d

{'name': 'peter', 'age': 25}

>>> 'age' in d

True

尽管字典和列表有很多特性相同，但也有下面一些重要的区别。

* **键类型**：字典的键不一定为整型数据(但也可能是)，也可能是其他不可变类型，比如浮点型(实型)、字符串或者元组。
* **自动添加**:即使某个键一开始在字典中并不存在，但也可以为它分配一个值，这样字典就会建立新的项。而(在不使用append方法或者其他类似操作的情况下)不能将值关联到列表范围之外的索引上。
* **成员资格**：表达式k in d（d为字典）查找的是键，而不是值。表达式v in l（l为列表）则用来查找值，而不是索引。如果字典含有指定的键，查找相应的值就很容易。

在字典中检查键的成员资格比在列表中检查值的成员资格更高效，数据结构的规模越大，两者的效率差距越明显。

下面看下自动添加字典和列表的区别：

>>> x = []

>>> x[42] = 'Foobar'

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

IndexError: list assignment index out of range

>>> x = {}

>>> x[42] = 'Foobar'

>>> x

{42: 'psz'}

从上面可以看到，程序试图将字符串'psz'关联到一个空列表的42号位置上——这显然是不对的。因为列表这个位置不存在。如果想将其变为可能，我们可以用[None]\*43或者其他方式初始化x，而不能仅仅用[]。而我们可以将'Foobar'直接关联到空字典的键42上。这样新的项便添加到字典中。

书本第57页

#### 代码清单4-1 字典示例

# 一个简单的数据库

# 字典使用人名作为键。每个人用另一个字典来表示，其键"phone"和"addr"分别表示他们的电话号码和地址。

people = {

"Alice": {

"phone": "2341",

"addr": "Foo drive 23"

},

"Beth": {

"phone": "9102",

"addr": "Bar street 42"

},

"Cecil": {

"phone": "3158",

"addr": "Baz avenue 90"

}

}

# 针对电话号码和地址使用的描述性标签，会在打印输出的时候用到

labels = {

"phone": "phone number",

"addr": "address"

}

name = input("Name: ")

# 查找电话号码还是地址

request = input("Phone number (p) or address (a)? ")

# 使用正确的键

if request == "p": key = "phone"

if request == "a": key = "addr"

# 如果名字是字典中的有效键才打印信息

if name in people:

print ("%s's %s is %s." % (name, labels[key], people[name][key]))

下面是程序的运行示例：

Name: Beth

Phone number (p) or address (a)? a

Beth's address is Bar street 42.

书本第58页

### 4.2.3 字典的格式化字符串

可以使用字符串格式化功能来格式化元组中的所有值，同样也可以使用字典来完成同样的工作。字典是通过格式化键名，进而得到相对应的值。

字典格式化字符串操作是通过在每个转换说明符（conversion specifier）中的%字符后面，然后添加上（用圆括号括起来）键名，后面再跟上其他说明元素。比如：

>>> phonebook = {'psz':'0592111','peter':'0591111'}

>>> "psz's phone number is %(psz)s." % phonebook

"psz's phone number is 0592111."

从上面可以看到，在被格式化的字符串后面，要加上%和所包含键名的字典。

除了增加的字符串键之外，转换说明符还是像以前一样工作。当以这种方式使用字典的时候，只要所给出的键都能在字典中找到，就可以获得任意数量的转换说明符。这类字符串格式化在模板系统中非常有用。（本例中使用HTML）比如：

>>> template = '''<html>

... <head><title>%(title)s</title></head>

... <body>

... <h1>%(title)s</h1>

... <p>%(text)s</p>

... </body>

... </head>

... </html>'''

>>> data = {'title':'Test page','text':'welcome'}

>>> print(template % data)

<html>

<head><title>Test page</title></head>

<body>

<h1>Test page</h1>

<p>welcome</p>

</body>

</head>

</html>

书本第58 - 59页

### 4.2.4 字典方法

#### clear方法

clear方法清除字典中所有的项。这是个原地操作（类似list.sort）,所以无返回值（或者说返回None）。

>>> con = {}

>>> con['name'] = 'peter'

>>> con['age'] = 24

>>> con

{'name' : 'peter', 'age' : 24}

>>> return\_value = con.clear()

>>> con

{}

>>> print(return\_value)

None

clear方法就是清空字典中的所有项。那么为什么这个方法有用？考虑下面两种情况。情况1：

>>> x = {}

>>> y = x

>>> x['key'] = 'value'

>>> y

{'key': 'value'}

>>> x = {}

>>> y

{'key': 'value'}

>>> x

{}

然后是情况2：

>>> x = {}

>>> y = x

>>> x['key'] = 'value'

>>> y

{'key': 'value'}

>>> x

{'key': 'value'}

>>> x.clear()

>>> x

{}

>>> y

{}

上面两种情况中，x和y最初对应同一个字典。情况1中将x关联到一个新的空字典来"清空"它，这对y一点影响也没有，它还是关联到原先的字典。但是如果想清空原始字典中所有的元素，必须使用clear方法。像情况2中那样，y随后也被清空了。

书本第59 - 60页

#### copy方法

##### copy模块的deepcopy函数

copy方法返回一个具有相同键-值对的新字典，这个方法实现的是浅复制(shallow copy)。因为用copy方法进行复制后的值本身就是是相同的，而不是副本。比如：

>>> x = {'username':'psz','machines':['foo','bar','baz']}

>>> y = x.copy()

>>> y['username'] = 'admin'

>>> y

{'machines': ['foo', 'bar', 'baz'], 'username': 'admin'}

>>> x

{'machines': ['foo', 'bar', 'baz'], 'username': 'psz'}

>>> y['machines'].remove('bar')

>>> y

{'machines': ['foo', 'baz'], 'username': 'admin'}

>>> x

{'machines': ['foo', 'baz'], 'username': 'psz'}

可以看到，使用copy方法，当在副本中替换值的时候，原字典不受影响。但是，如果修改了某个值（原地修改，而不是替换）,原始的字典也会被改变，因为同样的值也存储在原字典中。就像上面例子中'machines'列表一样。

避免这个问题的一种方法是使用深复制（deep copy），复制其包含所有的值。可以使用copy模块的deepcopy函数来完成操作：

>>> from copy import deepcopy

>>> d = {}

>>> d['names'] = ['psz','admin']

>>> c = d.copy()

>>> dc = deepcopy(d)

>>> d['names'].append('peter')

>>> c

{'names': ['psz', 'admin', 'peter']}

>>> d

{'names': ['psz', 'admin', 'peter']}

>>> dc

{'names': ['psz', 'admin']}

书本第60页

#### fromkeys方法

fromkeys方法基本语法是：

dict.fromkeys(seq[, value]))

参数：

seq -- 这是将用于字典的键准备的值的列表。

value -- 这是可选的，如果提供的话则值将被设置为这个值

fromkeys方法使用给定的键建立新的字典，每个键默认对应的值为None。比如：

>>> dic = {}

>>> dic.fromkeys(['name','age'])

{'age': None, 'name': None}

>>> dic

{}

>>> new\_dic = dic.fromkeys(['name','age'])

>>> new\_dic

{'age': None, 'name': None}

>>> dic

{}

上面的例子中，首先构造一个空字典，然后调用它的fromkeys方法，建立另一个新字典。也可以直接在所有字典的类型dict上面调用方法，比如：

>>> dict.fromkeys(['name','age'])

{'age': None, 'name': None}

如果不想使用None作为默认值，也可以自己提供默认值。如：

>>> dict.fromkeys(['name','age'],'(unknown)')

{'age': '(unknown)', 'name': '(unknown)'}

书本第60 - 61页

#### get方法

get()方法返回指定键的值，如果值不在字典中返回默认值。

get()方法语法：

dict.get(key, default=None)

参数

key -- 字典中要查找的键。

default -- 如果指定键的值不存在时，返回该默认值。

get方法是个更宽松的访问字典项的方法。一般来说，如果试图访问字典中不存在的项时会出错。比如：

>>> d = {}

>>> print(d['name'])

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

KeyError: 'name'

而使用get方法就不会：

>>> print(d.get('name'))

None

可以看到，当使用get方法访问一个不存在的键时，没有任何异常，而得到了None值。还可以自定义"默认"值，来替换None：

>>> d.get('name','psz')

'psz'

上面的例子中当键不存在时，如果有指定默认值，则返回默认值，否则返回None。

如果键存在，则get方法用起来就像普通的字典查询一样。比如：

>>> d

{}

>>> d['name'] = 'psz'

>>> d.get('name')

'psz'

书本第61页

#### 代码清单4-2 字典方法示例

# 使用get的简单数据库

# 这里添加代码清单4-1中插入数据库的代码

people = {

"Alice": {

"phone": "2341",

"addr": "Foo drive 23"

},

"Beth": {

"phone": "9102",

"addr": "Bar street 42"

},

"Cecil": {

"phone": "3158",

"addr": "Baz avenue 90"

}

}

labels = {

"phone": "phone number",

"addr": "address"

}

name = input("Name: ")

# 查找电话号码还是地址

request = input("Phone number (p) or address (a)? ")

# 使用正确的键

key = request # 如果请求既不是‘p’也不是‘a’

if request == "p": key = "phone"

if request == "a": key = "addr"

# 使用get()提供默认值

person = people.get(name, {})

label = labels.get(key, key)

result = person.get(key, 'not available')

print("%s's %s is %s." % (name, label, result))

下面是程序的运行示例：

Name: Gumby

Phone number (p) or address (a)? batting average

Gumby's batting average is not available.

get 方法带来的灵活性使得程序在用户输入我们并未准备的值是也能做出合理的反应。

书本第61 - 62页

#### has\_key方法 （python3.0没有这个函数）

has\_key方法可以检查字典中是否含有给出的键。表达式d.has\_key(k)相当于表达式k in d。使用哪个方式很大程度上取决于个人的喜欢。Python 3.0中不包括这个函数。

描述：

Python 字典(Dictionary) has\_key() 函数用于判断键是否存在于字典中，如果键在字典dict里返回true，否则返回false。

语法：

dict.has\_key(key)

参数：

key -- 要在字典中查找的键。

示例：

>>> d = {}

>>> d.has\_key('name')

False

>>> d['name'] = 'psz'

>>> d.has\_key('name')

True

书本第62页

#### items和iteritems方法（在Python3 iteritems方法已经被去掉了。）

**Items方法**

描述

items() 方法以列表返回可遍历的(键, 值) 元组数组。

语法

dict.items()

items方法将所有的字典项以列表方式返回，这些列表项中的每一项都来自于(键，值)。但是项在返回时并没有特殊的顺序。

>>> d = {'url':'coderschool.cn','title':'psz'}

>>> d.items()

dict\_items([('title', 'psz'), ('url', 'coderschool.cn')])

**iteritems()方法**

**在Python比较高的版本这个方法已经被去掉了。**

语法：

dict.iteritems()

描述

iteritems() 方法返回以迭代器对象而不是列表。

iteritems方法的作用和items方法大致相同，但是iteritems会返回一个迭代器对象而不是列表。

具体实例如下：

>>> d = {'title':'psz','age':18}

>>> f = d.iteritems()

>>> f

<dictionary-itemiterator object at ....>

>>> list(f) #迭代器转换为一个列表

[('title','psz'),('age',18)]

书本第62页

#### keys和iterkeys方法（在Python3 iterkeys方法已经被废除。）

**keys()方法**

描述

keys() 方法以列表返回一个字典所有的键。

语法

dict.keys()

keys()方法不用带任何参数。具体实例：

>>> d = {'url':'coderschool.cn','title':'psz'}

>>> d.keys()

dict\_keys(['url', 'title'])

keys()方法将全部键以列表的形式返回。

**iterkeys()方法**

**在3.0以上版本，iterkeys方法已经被废除**

语法：

dict.iterkeys()

描述：

iterkeys()方法返回针对键的迭代器而不是列表。

具体实例：

aDict = {'key1':'value1', 'key2':'value2', 'key3':'value3'}

for k in aDict.iterkeys():

print '%s:%s' % (k, aDict[k])

结果：

key3:value3

key2:value2

key1:value1

书本第62页

#### pop方法

描述

pop()方法用来获得对应给定键的值，然后将这个键-值对从字典中移除。

语法

pop(key[,default])

参数

key: 要删除的键-值对

default: 如果没有 key，返回 default 值

返回值

返回被删除的值。

具体实例：

>>> d = {'url':'coderschool.cn','title':'psz'}

>>> d

{'title': 'psz', 'url': 'coderschool.cn'}

>>> d.pop()

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: pop expected at least 1 arguments, got 0

>>> d.pop('title')

'psz'

>>> d

{'url': 'coderschool.cn'}

书本第62 - 63页

#### popitem方法

描述

popitem() 方法随机返回并删除字典中的一对键和值。

如果字典已经为空，却调用了此方法，就报出KeyError异常。

语法

dict.popitem()

返回值

返回一个键值对(key,value)形式。

popitem方法类似于list.pop,后者会弹出列表的最后一个元素。但不同的是，popitem弹出随机的项，因为字典并没有"最后的元素"或者其他有关顺序的概念。若想一个接一个地移除并处理项，这个方法就非常有效（因为不用首先获取键的列表）。

具体实例：

>>> d = {'name':'psz','age':18}

>>> d.popitem()

('age', 18)

>>> d

{'name': 'psz'}

>>> d['age'] = 18

>>> d['sex'] = "male"

>>> d

{'age': 18, 'name': 'psz', 'sex': 'male'}

>>> d.popitem()

('age', 18)

尽管popitem方法和列表的pop方法很类似，但字典中没有与append等价的方法。因为字典是无序的，类似于append的方法是没有任何意义的。

书本第63页

#### setdefault方法

描述

setdefault() 函数和get() 方法类似, 如果键不存在于字典中，将会添加键并将值设为默认值。

语法

dict.setdefault(key, default=None)

参数

key -- 查找的键值。

default -- 键不存在时，设置的默认键值。

返回值

如果字典中包含有给定键，则返回该键对应的值，否则返回为该键设置的值。

setdefault方法在某种程度上类似于get方法，就是能够获得与给定键相关联的值，除此之外，setdefault还能在字典中不含有给定键的情况下设定相应的键值。

>>> d = {}

>>> d.setdefault('name',None)

>>> d

{'name': None}

>>> d['name'] = 'psz'

>>> d.setdefault('name','peter')

'psz' #输出d['name']的值，如果d['name']没值，则输出'peter'

>>> d

{'name': 'psz'}

>>> d.setdefault('age',18)

18 #d字典中并没有age这个键，便默认返回18

可以看到，当键不存在的时候，setdefault返回默认值并且相应地更新字典。如果键存在，那么就返回与其对应的值，但不改变字典。默认值是可选的，这点和get一样。如果不设定，会默认使用None。如：

>>> d = {}

>>> print(d.setdefault('name'))

None

>>> d

{'name': None}

书本第63页

#### update方法

描述

update() 函数把字典dict2的键-值对更新到dict里。

语法

dict.update(dict2)

参数

dict2 -- 添加到指定字典dict里的字典。

update方法可以利用一个字典项更新另外一个字典，具体实例：

>>> d = {'name':'psz','age':18}

>>> x = {'sex':'girl'}

>>> d.update(x) #用x字典更新d字典

>>> d

{'age': 18, 'name': 'psz', 'sex': 'girl'}

update方法提供的字典中的项会被添加到旧的字典中，若有相同的键则会进行覆盖。如：

>>> d = {'name':'psz','sex':'boy','age':18}

>>> d

{'age': 18, 'name': 'psz', 'sex': 'girl'}

>>> x = {'name':'psz','age':19}

>>> d.update(x)

>>> d

{'age': 19, 'name': 'psz', 'sex': 'girl'}

update方法可以和映射、拥有（键、值）对的队列（或者其他可迭代的对象）以及关键字参数一起调用。

书本第63 - 64页

#### values和itervalues方法（python3.0没有itervalues函数）

描述

values() 函数以列表返回字典中的所有值。

语法

dict.values()

返回值

以列表的形式返回字典中的所有值。

values方法以列表的形式返回字典中的值（itervalues返回值的迭代器）。与返回键的列表不同的是，返回值的列表中可以包含重复的元素，实例：

>>> d = {}

>>> d[1] = 1

>>> d[2] = 2

>>> d[3] = 2

>>> d[4] = 1

>>> d.values()

>>> dict\_values([1, 2, 2, 1])

>>> dd = d.values()

>>> dd

dict\_values([1, 2, 2, 1])

itervalues()方法语法：

dict.itervalues()

描述

Python 2 中d.itervalues() 返回针对字典d里所有值的迭代器对象，Python 3无此语法。

参数

无任何参数

旧版中用list(d.itervalues())就可以得到字典中的值以列表形式返回。

书本第64页

# 第五章 条件、循环和其他语句

## 5.1 print和import的更多信息

### 5.1.1使用逗号打印多个值 （加号分隔符）

使用print打印多个表达式（或值）是非常容易的，只要将它们用逗号隔开就好了。比如：

>>> print('sex:','male')

sex: male

可以看到，每个参数之间都插人了一个空格。

要注意的是，print的参数并不能像我们预期那样构成一个元组：

>>> 1,2,3

(1, 2, 3)

>>> print(1,2,3)

1 2 3

>>> print((1,2,3))

(1, 2, 3)

如果想要同时输出文本和变量值，却又不希望使用字符串格式化的话，那这个特性就非常有用了：

>>> name = "psz"

>>> print("My name is ",name)

My name is psz

如果要两个参数之间没有空格，就要用到加号分隔符了：

>>> print("My name is" + "," + name )

My name is,psz

print() 函数的语法:

print(\*objects, sep=' ', end='\n', file=sys.stdout)

参数

objects -- 复数，表示可以一次输出多个对象。输出多个对象时，需要用 , 分隔。

sep -- 用来间隔多个对象，默认值是一个空格。

end -- 用来设定以什么结尾。默认值是换行符 \n，我们可以换成其他字符串。

file -- 要写入的文件对象。

书本第65 - 66页

### 5.1.2从模块导入函数

从模块导入函数，即把某件事作为另一件事导入，从模块导入函数的时候，可以使用：

import somemodule

或者

from somemodule import somefunction

或者

from somemodule import somefunction , anotherfunction ,yetanotherfunction

或者

#表示导入module的所有相关函数

from somemodule import \*

只要确定自己想要从给定的模块导入所有功能时，才应该使用最后一个版本。但是如果两个模块都有open函数，那又该怎么办?只需要使用第一种方式导入，然后像下面这样使用函数：

module1.open(......)

module2.open(......)

但还有另外的选择：可以在语句末尾增加一个as子句，在该子句后给出名字。

为整个模块提供别名：

>>> import math as math\_alias

>>> math\_alias.sqrt(4)

2.0

为函数提供别名：

>>> from math import sqrt as math\_alias

>>> math\_alias(4)

2.0

对于open函数，可以像下面这样使用：

from module1 import open as open1

from module2 import open as open2

有些模块，列如os.path是分层次安排的（一个模块在另一个模块的内部）

书本第66 - 67页

## 5.2 赋值魔法

### 5.2.1序列解包

赋值语句的例子有很多，其中包括对变量和数据结构成员的（比如列表中的位置和分片以及字典中的槽）赋值。但赋值的方法还不止这些。比如，用序列解包的方法给多个变量同时进行赋值操作：

>>> x,y,z = 1,2,3

>>> print(x,y,z)

1 2 3

序列解包的方法非常有用，还可以用它交换两个（或更多个）变量也是没问题的，如:

>>> x,y,z = 1,2,3

>>> print(x,y,z)

1 2 3

>>> x,y = y,x

>>> print(x,y,z)

2 1 3

事实上，这里所做的事情叫做序列解包(sequence unpacking)或递归解包——将多个值的序列解开，然后放到变量的序列中。更形象一点的表示出来就是：

>>> values = 1,2,3

>>> values

(1, 2, 3)

>>> x,y,z = values

>>> x

1

当函数或者方法返回元组（或者其他序列或可迭代对象）时，这个特性尤其有用。假设需要获取（和删除）字典中任意的键一值对，可以使用popitem方法，这个方法将键-值作为元组返回。那么这个元组就可以直接赋值到两个变量中：

>>> person = {'name':'psz','age':18}

>>> key,value = person.popitem()

>>> key

'age'

>>> value

18

它允许函数返回一个以上的值并且打包成元组，然后通过一个赋值语句很容易进行访问。所解包的序列中的元素数量必须和放置在赋值符号(=)左边的变量数量完全一致，否则Python会在赋值时引发异常，比如：

>>> a,b,c = 1,2

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

ValueError: need more than 2 values to unpack

**Python 3.0中有另外一个解包的特性：可以像在函数的参数列表中一样使用星号运算符。例如，a, b, \*rest = [1, 2, 3, 4]最终会在 a 和 b 都被赋值之后将所有其他的参数都是收集到 rest 中。本例中，rest的结果将会是[3, 4]。使用星号的变量也可以放在第一个位置，这样它就总会包含一个列表。右侧的赋值语句可以是可迭代对象。**

书本第67 - 68页

### 5.2.2 链式赋值

链式赋值（chained assignment）是将同一个值赋给多个变量的捷径。它看起来像并行赋值，不过这里只处理一个值，基本方式：

x = y = somefunction

等价于

y = somefunction

x = y

注意，上面的语句和下面的语句不一定等价：

x = somefunction()

y = somefunction()

对于链式赋值的实例：

>>> x = y = 4

>>> x

4

>>> y

4

书本第68页

### 5.2.3增量赋值

最常用的Python赋值方法就是比如x=x+y这种类型的赋值，不过这里要介绍的赋值方法是将表达式运算符(本例是+)放置在赋值运算符=的左边，比如写成x+=1。这种写法叫做增量赋值（augmented assignment），对于\*、/、%等标准运算符都适用：

>>> y = 3

>>> y+=1

>>> y

4

>>> y\*=2

>>> y

8

对于其他数据类型也适用（只要二元运算符本身适用于这些数据类型即可）：

>>> str = 'hello'

>>> str += ' world!'

>>> str

'hello world!'

>>> str\*=2

>>> str

'hello world!hello world!'

增量赋值可以使得代码更加紧凑和简练。

书本第68页

## 5.3 语句块：缩排的乐趣

语句块并非一种语句， 语句块是在条件为真（条件语句）时执行或者执行多次（循环语句）的一组语句。在代码前放置空格来缩进语句即可创建语句块。

注意，使用tab字符也可以缩进语句块。Python将一个tab字符解释为到下一个tab字符位置的移动，而一个tab字符位置为8个空格，但是标准且推荐的方式是只用空格，尤其是在每个缩进需要4个空格的时候。

块中的每行都应该缩进同样的量。下面的伪代码（并非真正Python代码）展示了缩进的工作方式：

this is a line

this is another 1ine：

this is another block

continuing the same block

the last 1ine of this block

phew.there we escaped the inner block

很多语言使用特殊单词或者字符（比如begin或{）来表示一个语句块的开始，用另外的的单词或者字符（比如end或者}）表示语句块的结束。在Python中，冒号（:）用来标识语句块的开始，块中的每一个语句都是缩进的（缩进量相同）。当回退到和已经闭合的块一样的缩进量时，就表示当前块已经结束了（很多程序编辑器和IDE都知道如何缩进语句块，可以帮助用户轻松把握缩进）。

书本第69页

## 5.4 条件和条件语句

### 5.4.1布尔变量

真值（也叫做布尔值，这个名字根据在真值上做过大量研究的George Boole命名的）是接下来内容的主角。

下面的值在作为布尔表达式的时候，会被解释器看作假(false):

False None 0 "" () [] {}

换句话说，也就是标准值False和None、所有类型的数字0（包括浮点型、长整型和其他类型）、空序列（比如空字符串、元组和列表）与及空的字典都为假。其他的一切都被解释为真，包括特殊值True。

事实上，True和False只不过是1和0的另一种写法，看起来不同，但作用相同。

>>> True

True

>>> False

False

>>> True == 1

True

>>> False == 0

True

>>> True + False + 42

43

那么，如果某个逻辑表达式返回1或0，那么它实际的意思是返回Ture或False。

布尔值True和False属于布尔类型，bool函数可以用来（和list、str与及tuple一样）转换其他值。比如：

>>> bool('Hello World!')

True

>>> bool(12)

True

>>> bool(0)

False

>>> bool("")

False

>>> bool([])

False

>>> bool(())

False

因为所有值都可以用作布尔值，所以几乎不需要对他们进行显示转换。

尽管[]和“”都是假值，也就是说bool([])==bool(“”)==False,但是他们本身却并不相等(也就是说[]!=””),对于其他不同类型的假值对象也是如此（例如()!=False）

书本第69 - 70页

### 5.4.2条件执行和if语句

if语句，它可以实现条件执行。即如果条件(在if和冒号之间的表达式)判定为真，那么后面的语句块就会被执行。如果条件为假，

语句块就不会被执行。实例：

>>> if 1:

... print(1)

...

1

注意，if语句在最后有一个冒号，紧随其后的是python语句块。语句块要记得要缩进，可以用tab键进行缩进。如果没有缩进就会报错，比如：

>>> if 1:

... print(123)

  File "<stdin>", line 2

    print(123)       ^

IndentationError: expected an indented block

python是强行要求缩进代码块，因为这样有助于提高代码的可读性。同时要十分注意代码块的作用域，如：

if a :

if b:

# 这里是if b的作用区间

#这里是if a的作用区间

#这里不在if 区间

书本第70 - 71页

### 5.4.3 else子句

else字句可以和if语句配合使用，增加一种选择。当if后面的表达式为假时，就执行else字句后面的语句块。else之所以叫做字句是因为它不是独立的语句，而只能作为if语句的一部分。下面看个实例：

>>> a = 1

>>> b = 2

>>> if a<b :

...     print("a<b")

... else:

...     print("a>=b")

...

a<b

如果if第一个语句块没有被执行（因为条件被判定为假），那么就会转入第2个语句块。

书本第71页

### 5.4.4 elif语句

如果需要检查多个条件，就可以使用elif字句，它是"else if"的简写，也是if和else字句的联合使用——也就是具备条件的else字句。实例：

>>> number = 1

>>> if number > 0 :

...     print("number > 0")

... elif number < 0 :

...     print("number < 0")

... else :

...     print("number == 0")

...

number > 0

注意：

1、else、elif为子块，不能独立使用

2、一个if语句中可以包含多个elif语句，但结尾只能有一个else语句

书本第71页

### 5.4.5嵌套代码块

书本第71 - 72页

### 5.4.6更复杂的条件

#### 比较运算符

用在条件中，最基本的运算符就是比较运算符了，它们用来比较其他对象。比较运算符整理如下：

表达式     描述

x == y       x等于y

x < y     x小于y

x > y       x大于y

x >= y       x大于等于y

x <= y       x小于等于y

x != y       x不等于y

x is y       x和y是同一个对象

x is not y     x和y不是同一个对象

x in y   x是y容器（例如，序列）的成员

x not in y     x不是y容器（例如，序列）的成员

理论上，对于相对大小的任意两个对象和y都是可以使用比较运算符(例如，<和<=）比较的,并且都会得到一个布尔结果。但是只有在x和y是相向或者近似类型的对象时，比较才有意义（例如，两个整形数或者一个整形数和一个浮点形数进行比较）。

正如将一个整形数添加到一个字符串中是没有意义的，检查一个整形是否比一个字符串小，看起来也是毫无意义的。但奇怪的是，在Python3.0之前的版本中这却是可以的。对于此类行为应该避免出现，因为结果完全不可靠，在每次程序执行的时候得到的结果都可能不同。在Python3.0中，比较不兼容类型的对象已经不再可行。

**有时候你会见到x<>这样的表达式，它的意思其实就是x!=y。不建议使用<>运算符。**

在Python中比较运算符和赋值运算一样是可以连接的，几个运算符可以连在一起使用，比如：

>>> x = 1

>>> 1 < x < 2

False

>>> 0 < x < 2

True

书本第72 - 73页

#### 相等运算符

如果想要知道两个东西是否相等，应该使用相等运算符，即两个等号"==":

>>> 1 == 2

False

>>> "ss" == "ss"

True

几乎所有的编程语言判断是否相等都是用两个等号，这和我们传统数学里面的判断是不一样的。在Python下，单个相等运算符是赋值运算符，是用来改变值的，而不能用来比较。比如：

>>> x = 3

>>> x

3

>>> "foo" = "foo"

  File "<stdin>", line 1

SyntaxError: can't assign to literal

书本第73页

#### is:同一性运算符

is这个运算符比较有趣，它看起来和"=="一样，事实上却不同，比如：

>>> x = y = [1,2,3]

>>> z = [1,2,3]

>>> x == y

True

>>> x == z

True

>>> x is y

True

>>> x is z

False

从上面的例子可以看到，x和z相等却不等同，为什么?因为is运算符是判定同一性而不是相等性的。变量x和y都绑定到同一个列表上，而变量z被绑定在另外一个具有相同数值和顺序的列表上。它们的值可能相等，但是却不是同一个对象。

再看个实例：

>>> x = [1,2,3]

>>> y = [2,4]

>>> x is not y

True

>>> del x[2]

>>> y[1] = 1

>>> y.reverse()

>>> y

[1, 2]

>>> x

[1, 2]

>>> x == y

True

>>> x is y

False

本例中，首先包括两个不同的列表x和y。可以看到x not y与(x is y相反），这个已经知道了。之后我改动了y列表，尽管它们的值相等了，但是还是两个不同的列表。

显然，两个列表值相等但是不等同。

总结一下：使用==运算符来判定两个对象是否相等，使用is判定两者是否等同（同一个对象）。

**避免将is运算符用于比较类似数值和字符串这类不可变值。由于Python内部操作这些对象的方式的原因，使用运算符的结果是不可预测的。**

书本第73 - 74页

#### in:成员资格运算符

In运算符它可以像其他比较运算符一样在条件语句中使用

name= input('What is your name? ')

If 's' in name:

print 'Your name contains the letter"s".'

else:

print 'Your name does not contain theletter "s".'

书本第74页

#### 字符串和序列比较 （ord函数和chr函数）

字符串可以按照字母顺序排列进行比较。如：

>>> "alpha" < "beta"

True

如果字符串内包括大写字母，那么结果就会有点乱（实际上，字符是按照本身的顺序值排列的。一个字母的顺序值可以用ord函数查到，ord函数与chr函数功能相反）。如果要忽略大小写字母的区别，可以使用字符串方法upper和lower。

>>> "abc".upper() == "ABC"

True

其他的序列也可以使用同样的方式进行比较，不过比较的不是字符而是元素的其它类型。

>>> [1,2] < [2,1]

True

>>> [1,2] < [1,2,3]

True

如果一个序列中包括其他序列元素，比较规则也同样适用于序列元素。如：

>>> [2,[1,3]] < [2,[1,5]]

True

书本第74页

#### 布尔运算符

返回真值的对象有许多（事实上，所有值都可以解释为真值，所有的表达式也都返回真值）。但有时想要检查一个以上的条件。例如，如果需要编写读取数字并且判断该数字是否位于1到10之间（包括10）的程序，可以像下面这样做：

number = int(input('Enter a number between 1 and 10: '))

if number <= 10 :

if number >= 1 :

print("Great!")

else :

print("wrong!")

else:

print("wrong!")

上面的程序判断有点复杂和麻烦，可以通过使用布尔运算符简化程序，比如：

number = int(input('Enter a number between 1 and 10: '))

if number >= 1 and number <= 10:

print("Great!")

else:

print("wrong!")

上面实例还有更简单的方法，即使用连接比较：1 <= number <= 10。

and运算符就是所谓的布尔运算符。它连接两个布尔值，并且在两者都为真时返回真，否则返回假。与它同类的还有两个运算符，or和not，使用这3个运算符就可以随意结合真值。下面具体介绍这三个布尔运算符的使用方法：

* 表达式： x and y ，需要两个变量都为真时才为真，所以如果x为假，表达式就会立刻返回false，而不管y的值。实际上，如果x为假，表达式会返回x的值——否则它就返回y的值。
* 表达式：x or y中，至少一个变量为真才返回真，否则返回加。也就是说，x为真时，它就直接返回真(不会再去判断y的值)，否则返回y值。
* 表达式：not x中，当x变量为假时表达式为真，当x变量为真时，表达式为假

书本第74 - 75页

##### 短路逻辑和条件表达式

布尔运算符有个有趣的特性：只有在需要求值时才会进行求值。举例来说，表达式x and y需要两个变量都为真时才为真，所以如果x为假，表达式就会立刻返回false，而不管y的值。如果x为假，表达式会返回x的值--否则它就返回y的值。这种行为被称为短路逻辑或惰性求值：布尔运算符通常被称为逻辑运算符。这种行为对于or来说也同样适用，在表达式x or y中，x为真时，它直接返回x值，否则返回y值。注意，这意味着在布尔运算符之后的所有代码都不会执行。

这主要是避免了无用的执行代码，可以作为一种技巧使用，假设用户应该输入一个名字，但也可以什么都不输入，这时候可以使用默认值'<unknown>'。可以使用if语句，但是可以用一个很简洁的方式：

name = input('Please enter your name: ') or '<unknown>'

这里如果input语句的返回值为真（不是空字符串），那么它的值就会赋给name，否则将默认的'<unknown>'赋值给name。

这类短路逻辑可以用来实现C和java中所谓的三元运算符（或条件运算符）。

书本第75页

### 5.4.7断言

if语句有个非常有用的“近亲”，就是assert断言，它的工作方式多少有点像下面这样（伪代码）：

if not condition ：

crash program

其表达的意思就是，程序在我的假设条件下，能够正常良好的运作，其实就相当于一个if语句：

if 假设成立 :

程序正常运行；

else :

报错&&终止程序！（避免由程序运行引起更大的错误）

assert断言是声明其布尔值必须为真的判定，如果发生异常就说明表达示为假。可以用来测试表示式，其返回值为假，就会触发异常。

assert语法格式如下：

assert expression [, arguments]

assert 表达式 [, 参数]

为什么会需要这样的代码呢？就是因为与其让程序在晚些时候崩溃，不如在错误条件出现时直接让它崩溃。一般说来，你可以要求某些条件必须为真（例如，在检查函数参数的属性时，或者作为初期测试和调试过程中的辅助条件）。语句中使用的关键字是assert。如：

>>> year = 2015

>>> assert 0 < year < 2016

>>> assert year >= 2016

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AssertionError

如果需要确保程序中的某个条件一定为真才能让程序正常工作的话，assert语句就非常有用了，它可以在程序中置入检查点。

条件后可以添加字符串，用来解释断言：

>>> age = -1

>>> assert 0 < age < 100 , "age must be realistic"

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AssertionError: age must be realistic

书本第76页

## 5.5循环

### 5.5.1 while循环（内有isspace()和strip()）

while语句用于循环执行程序，即在某条件下，循环执行某段程序，以处理需要重复处理的相同任务。其基本形式为：

while 判断条件：

执行语句...

执行语句可以是单个语句或语句块。判断条件可以是任何表达式，任何非零、或非空（null）的值均为true。当判断条件假false时，循环结束。

一个循环就可以确保用户输入了名字：

name=''

while not name:

name = input('please enter your name ')

print 'Hello,%s!' %name

运行该程序，如果直接输入一个空格，程序会接受这个名字，因为包括一个空格的字符并不是空的，我们可以把while not name改为while not name or name.isspace()或者改为while not name.[strip()](#_3.4.6 strip方法)

while 语句时还有另外两个重要的命令 continue，break 来跳过循环，continue 用于跳过该次循环，break 则是用于退出循环，此外"判断条件"还可以是个常值，表示循环必定成立，具体用法如下：

>>> x = 10

>>> while 1: #判定条件永远为真

... x -= 1

... if x <= 5: #当x<=5无条件终止循环

... break;

... print(x)

...

9

8

7

6

和

>>> x = 10

>>> while x >= 6:

... x -= 1

... if x == 8: #当x=8时跳出该次循环，不执行后面语句块

... continue

... print(x)

...

10

9

7

6

**无限循环**

如果条件判断语句永远为 true，循环将会无限的执行下去，如下实例：

>>> while 1:

... print(1)

...

可以按ctril+c来中断循环。

在程序中要避免无限循环的出现，以免造成不必要的错误。当在编写循环程序时，一定要十分注意循环条件能否使循环正常终止。

书本第77页

### 5.5.2 for循环 （内含range函数）

while语句非常灵活。它可以用来在任何条件为真的情况下重复执行一个代码块。一般情况下这样就够用了，但有时候比如要为一个集合(序列和其他可迭代对象)的每个元素都执行一个代码块，这时候就可以使用for循环。

for循环可以遍历任何序列的项目，如一个列表或者一个字符串。

语法：

for iterating\_var in sequence:

statements(s)

for循环实例，循环输出列表中的每个值：

>>> words = ['I','am','a','coder']

>>> for word in words :

... print(word)

...

I

am

a

coder

循环输出字符串中的每个字符：

>>> letters = "good day!"

>>> for letter in letters:

... print(letter)

...

g

o

o

d

d

a

y

!

因为在for循环下，迭代某种范围的数字是很常见的，所以有时候还可以通过巧用内建函数range来使得循环输出更灵活：

>>> range(0,5)

range(0, 5) #range(0,5)即[0,1,2,3,4]

range函数的工作方式类似于分片。它包含下限(上面例子中的0)，但不包含上限(上面例子中的5)。如果希望下限为0，可以只提供上限：

>>> range(10)

range(0, 10)

range函数的第三个参数表示步长，如果是负数则表示反向迭代。

下面通过range打印1到5之间的数字：

>>> for number in range(1,6):

... print(number)

...

1

2

3

4

5

实例，通过序列索引迭代：

>>> words = ['I','am','good!']

>>> for index in range(len(words)):

... print(words[index])

...

I

am

good!

上面中len(words)的值为3，所以for index in range(len(words))等价于for index in range(3)。

可见，for循环比while循环更简洁。如果能使用for循环，就尽量不用while循环。

书本第77 - 78页

### 5.5.3循环便利字典元素

一个简单的for语句就能循环字典的所有键，就像遍历访问序列一样：

>>> d = {'a':1,'b':2,'c':3}

>>> for index in d:

... print(d[index])

...

1

3

2

>>> d = {'age':18,'name':'psz'}

>>> for key in d :

... print(key + ":" , d[key])

...

name: psz

age: 18

在python2.2之前，还只能用[beys](#_keys和iterkeys方法（在Python3 iterkeys方法已经被废除。）)等字典方法来获取键（因为不允许直接迭代字典）。如果只需要值，可以使用d.[values](#_values和itervalues方法（python3.0没有itervalues函数）)替代d.[beys](#_keys和iterkeys方法（在Python3 iterkeys方法已经被废除。）)。d.[items](#_items和iteritems方法（在Python3 iteritems方法已经被去掉了。）)方法会将键值对作为元组返回，for循环的一大好处就是可以循环中使用序列解包。

>>> d = {'x':1, 'y':2, 'z':3}

>>> for key, value in d.items():

... print key, 'corresponds to', value

...

y corresponds to 2

x corresponds to 1

z corresponds to 3

字典元素的顺序通常是没有定义的。换句话说，迭代的时候，字典中的键和值都能保证被处理，但是处理顺序不确定。如果顺序很重要的话，可以将键值保存在单独的列表中，例如在迭代前进行排序。

书本第78页

### 5.5.4 一些迭代工具

#### 并行迭代

可以同时迭代两个序列。比如有下面两个列表：

names = ['jack','peter','psz']

ages = [12,15,17]

如果想要打印名字和对应的年龄，可以像下面这样做：

>>> names = ['jack','peter','psz']

>>> ages = [12,15,17]

>>> for i in range(len(names)):

... print(names[i],'is',ages[i],'years old')

...

jack is 12 years old

peter is 15 years old

psz is 17 years old

这里**i是循环索引的标准变量名**。

还可以使用内建的zip函数用来进行并行迭代，可以把两个序列“压缩”在一起，然后返回一个元组的列表：

>>> for (name,age) in zip(names,ages):

... print(name,'is',age,'years old')

...

jack is 12 years old

peter is 15 years old

psz is 17 years old

在上面的实例中：

>>> zip(names,ages)

#zip(names,ages)等价于[('jack',12),('peter',15),('psz',17)]

<zip object at 0x02420260>

>>> last(zip(names, ages))

[('jack',12),('peter',15),('psz',17)]

zip函数也可以作用于任意多个序列，这一点在某些情况下非常好用，比如：

>>> a = [1,2,3]

>>> b = [1,2,3]

>>> c = [1,2,3]

>>> for (x,y,z) in zip(a,b,c):

... print(x,y,z)

...

1 1 1

2 2 2

3 3 3

上面的zip(a,b,c)最后等价于[(1,1,1),(2,2,2),(3,3,3)]。

关于很重要的一点是，zip还可以应付不等长的序列，当最短的序列“用完”的时候就会停止，比如：

>>> a = [1,2,3]

>>> b = [1,2,3,4]

>>> c = [1,2,3,4,5]

>>> for (x,y,z) in zip(a,b,c):

... print(x,y,z)

...

1 1 1

2 2 2

3 3 3

书本第79页

#### 按索引迭代 （enumerate函数）

有些时候想要迭代序列中的对象，同时还要获取当前对象的索引。例如，在一个字符串列表中替换索引包含“abc”的字符串。实现的方法有很多，比如可以用下面这种方法：

>>> list = ['abc','abcde','ff','abc']

>>> for string in list:

... if 'abc' in string:

... index = list.index(string)

... list[index] = 'ccc'

...

>>> list

['ccc', 'ccc', 'ff', 'ccc']

上面的例子中，不好的地方在于在替换字符串前还要搜索给定的字符串。还有另一种比较好的方法，如：

>>> list = ['abc','abcde','ff','abc']

>>> index = 0

>>> for string in list:

... if "abc" in string:

... list[index] = "ccc"

... index += 1

...

>>> list

['ccc', 'ccc', 'ff', 'ccc']

另一种方法是使用内建的enumerate函数：

>>> list = ['abc','abcde','ff','abc']

>>> for (index,string) in enumerate(list):

... if "abc" in string:

... list[index] = "ccc"

...

>>> list

['ccc', 'ccc', 'ff', 'ccc']

enumerate函数可以在提供索引的地方迭代索引-值(键值)对。

序列索引迭代：

>>> words = ['I','am','good!']

>>> for index in range(len(words)):

... print(words[index])

...

I

am

good!

上面中len(words)的值为3，所以for index in range(len(words))等价于for index in range(3)。

书本第79 - 80页

#### 翻转和排序迭代

两个有用的函数：reversed和sorted。他们同列表的[reverse](#_reverse方法)和[sort](#_sort方法)（sorted和sort使用同样的参数）方法类似，打作用域任何序列或可迭代对象上，不是原地修改对象，而是返回翻转后的版本：

>>> sorted([4, 3, 6, 8, 3])

[3, 3, 4, 6, 8]

>>> sorted('Hello, word!')

[' ', '!', ',', 'H', 'd', 'e', 'l', 'l', 'o', 'o', 'r', 'w']

>>> list(reversed('Hello, word!'))

['!', 'd', 'r', 'o', 'w', ' ', ',', 'o', 'l', 'l', 'e', 'H']

>>> ''.join(reversed('Hello, word!'))

'!drow ,olleH'

注意，虽然sorted方法返回列表，reversed方法却返回一个更加不可思议的可迭代对象。它们具体的含义不用过多关注，大可在for循环以及join方法中使用，而不会有任何问题。不过却不能直接对它使用索引、分片以及调用list方法，如果希望进行上述处理，那么就可以使用list类型进行转换返回的对象，上述例子已经给出具体的做法。

书本第80页

### 5.5.5 跳出循环

#### break

break语句用来结束（跳出）循环

假设需要寻找100以内的最大平方数，那么程序可以从100往下迭代到0.当找到一个平方数时就不需要继续循环了，所以可以跳出循环：

from math import sqrt

for n in range(99,0,-1):

root=sqrt(n)

if root==(int)root:

print n

Break

如果执行这个程序的话，会打印出81，然后程序停止。如果不加break的话程序就会执行完，打印出81， 64，49，36，25，16，9，4，1

range 的第三个参数-表示步长，如果是负数则表示反向迭代。它也可以用来跳过数字：

>>> range(0,10,2) #步长为2

[0, 2, 4, 6, 8]

>>> range(10,0,-2) #步长为-2,反向迭代

[10, 8, 6, 4, 2]

书本第80页

#### Continue

continue语句比break语句用得要少得多。它会让当前的迭代结束，跳到下一轮循环的开始。它的基本的意思是“跳过剩余的循环体，但是不结束循环”，当循环体很大而且很复杂的时候这会很有用。

书本第80 - 81页

#### while true/break习语

如果需要当用户在提示符下输入单词时做一些事情，并且在用户不输入单词后结束循环。可使用下面的方法：

word = "dummy"

while word:

word = input("Please enter a word: ")

# 处理word：

print ("The word was " + word)

# 下面是一个会话示例:

Please enter a word: first

The word was first

Please enter a word: second

The word was second

Please enter a word:

代码按要求的方式工作(大概还能做些比直接打印出单词更有用的工作)。但是代码有些丑。在进入循环之前需要给word赋一个哑值(未使用的)。使用哑值(dummy value)就是工作没有尽善尽美的标志。让我们试着避免它：

word = raw\_input("Please enter a word: ")

while word:

# 处理word

print "The word was " + word

word = raw\_input("Please enter a word: ")

哑值没有了。但是有重复的代码(这样也不好):要用一样的赋值语句在两个地方两次调用raw\_input。能否不这么做呢？可以使用while True/break语句:

while True:

word = raw\_input("Please enter a word: ")

if not word: break

# 处理word

print "The word was " + word

while True的部分实现了一个永远不会自己停止的循环。但是在循环内部的if语句中加入条件也是可以的，在条件满足时使用break语句。这样一来就可以在循环内部任何地方而不是只在开头(像普通的while循环一样)终止循环。if/break语句自然地将循环分为两部分：第一部分负责初始化(在普通的while循环中，这部分需要重复)，第二部分则在循环条件为真的情况下使用第一部分內初始化好的数据。

尽管应该避免在代码中频繁使用break语句(因为这可能会让循环的可读性降低，尤其是在一个循环中使用多个break语句的时候)，但这个特殊的技术用得非常普遍，大多数Python程序员(包括你自己)都能理解你的意思。

书本第81 - 82页

### 5.5.6循环中的else语句

在循环中添加一个else子句，这样子可以在没有调用break时执行

from math import sqrt

for n in range(99, 81, -1):

root = sqrt(n)

if root == int(root):

print n

break

else:

print "Didn't find it!"

注意我将下限改为81(不包括81)以测试else子句。如果执行程序的话，它会打印出"Didn't find it!"，因为(就像在break那节看到的一样)100以内最大的平方数是81。for和while循环中都可以使用continue、break语句和else子句。

书本第82页

## 5.6列表推导式——轻量级循环（利用其他列表创建新列表）

列表推导式(list comprehension)是利用其他列表创建新列表(类似于数学术语中的集合推导式)的一种方法。它的工作方式类似于for循环，也很简单：

>>> [x\*x for x in range(10)]

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

列表有range(10)中每个x的平方组成。太容易了？如果只想打印出那些能被3整除的平方数呢？那么可以使用模除运算符——y%3，当数字可以被3整除时返回0(注意，x能被3整除时，x的平方必然也可以被3整除)。这个语句可以通过增加一个if部分添加到列表推导式中：

>>> [x\*x for x in range(10) if x % 3 == 0]

[0, 9, 36, 81]

也可以增加更多for语句的部分:

>>> [(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)]

作为对比，下面的代码使用两个for语句创建了相同的列表:

result = []

for x in range(3):

for y in range(3):

result.append((x, y))

print result

也可以和if子句联合使用，像以前一样:

>>> girls = ["alice", "bernice", "clarice"]

>>> boys = ["chris", "arnold", "bob"]

>>> [b+"+"+g for b in boys for g in girls if b[0] == g[0]]

['chris+clarice', 'arnold+alice', 'bob+bernice']

这样就得到了那些名字首字母相同的男孩和女孩。

书本第82 - 83页

##### 更优秀的方案

男孩/女孩名字对的例子其实效率不高，因为它会检查每个可能的配对。Python有很多解决这个问题的方法，下面的方法是Alex Martelli推荐的：

girls = ["alice", "bernice", "clarice"]

boys = ["chris", "arnold", "bob"]

letterGirls = {}

for girl in girls:

letterGirls.setdefault(girl[0], []).append(girl)

print [b+"+"+g for b in boys for g in letterGirls[b[0]]]

这个程序创建了一个叫做letterGirls的字典，其中每一项都把单字母作为键，以女孩名字组成的列表作为值。(setdefault字典方法在前一章中已经介绍过)在字典建立后，列表推导式循环整个男孩集合，并且查找那些和当前男孩名字首字母相同的女孩集合。这样列表推导式就不用尝试所有的男孩女孩的组合，检查首字母是否匹配。

书本第83页

## 5.7三人行 （pass、del、exec）

### 5.7.1什么都没发生（pass）

需要程序什么都不做的时候，就可以使用pass：

>>> pass

>>>

把pass放在代码中起到了占位符的作用。比如程序需要一个if语句，但是缺少一个语句块的代码：

if name == "Ralph Auldus Melish":

print "Welcome!"

elif name == "End":

# 还没完······

elif name == "Bill Gates":

print "Access Denied"

代码不会执行，因为Python中空代码块是非法的。解决方案就是在语句块中加上一个pass语句:

if name == "Ralph Auldus Melish":

print "Welcome!"

elif name == "End":

# 还没完······

pass

elif name == "Bill Gates":

print "Access Denied"

注：注释和pass语句联合的代替方案是插入字符串。对于那些没有完成的函数(参见第六章)和类(参见第七章)来说这个方法尤其有用，因为它们会扮演文档字符串(docstring)的角色(第六章中会有解释)。

书本第84页

### 5.7.2 使用del删除

一般来说，Python会删除那些不再使用的对象(因为使用者不会再通过任何变量或数据结构引用它们)：

>>> scoundrel = {"age": 42, "first name": "Robin", "last name": "of Locksley"}

>>> robin = scoundrel

>>> scoundrel

{'last name': 'of Locksley', 'first name': 'Robin', 'age': 42}

>>> robin

{'last name': 'of Locksley', 'first name': 'Robin', 'age': 42}

>>> scoundrel = None

>>> robin = None

首先，robin和scoundrel都被绑定到同一个字典上。所以当设置scoundrel为None的时候，字典通过robin还是可用的。但是当我把robin也设置为None的时候，字典就“漂”在内存里了，没有任何名字绑定到它上面。没有办法获取和使用它，所以Python解释器(以其无穷的智慧)直接删除了那个字典(这种行为被称为垃圾收集)。注意，也可以使用None之外的其他值。字典同样会“消失不见”。

另外一个方法就是使用del语句(我们在第二章和第四章里面用来删除序列和字典元素的语句)，它不仅会移除一个对象的引用，也会移除那个名字本身。

>>> x = 1

>>> del x

>>> x

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'x' is not defined

看起来很简单，但有时理解起来有些难度。例如，下面的例子中，x和y都指向同一个列表:

>>> x = ["Hello", "world"]

>>> y = x

>>> y[1] = "Python"

>>> x

['Hello', 'Python']

会有人认为删除x后，y也就随之消失了，但并非如此:

>>> del x

>>> y

['Hello', 'Python']

为什么会这样？x和y都指向同一个列表。但是删除x并不会影响y。原因就是删除的只是名称，而不是列表本身(值)。事实上，在Python中是没有办法删除值的(也不需要过多考虑删除值的问题，因为在某个值不再使用的时候，Python解释器会负责内存的回收)。

书本第84 - 85页

### 5.7.3使用exec和eval执行和求值字符串

有些时候可能会需要动态地创造Python代码，然后将其作为语句执行或作为表达式计算。下面详细介绍如何执行存储在字符串中的代码。

#### evec

exec 执行储存在字符串或文件中的 Python 语句，相比于 eval，exec可以执行更复杂的 Python 代码。

exec语法：

exec(str)

其中：str可以是：

代码字符串

文件对象

代码对象

tuple

注意：python2时exec 是一个语法声明，不是一个函数。也就是说和if,for一样。但在Python3.0中，exec是一个函数而不是语句。

使用exec执行一个字符串： （内外分号要不同）

>>> exec('print("Hello,world!")')

Hello,world!

>>> a = 2

>>> exec("a=1")

>>> a

1

但是，比如下面这个例子就会报错：

>>> from math import sqrt

>>> exec("sqrt = 1")

>>> sqrt(4)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'int' object is not callable

exec语句最有用的地方在于可以动态地创建代码字符串。如果字符串是从其他地方获得——很有可能是用户——那么几乎不能确定其中到底包含什么代码。所以为了安全起见，可以增加一个字典，起到命名空间的作用。

命名空间，可以浅显的理解为放置变量的地方。可以用字典起到放置代码字符串命名空间的作用。实例：

**网络代码：**

>>> a = 2

>>> b = 3

>>> g = {"a":6,"b":3}

>>> a

2

>>> g['a']

6

>>> from math import sqrt

>>> d = {"sqrt":1}

>>> sqrt(4)

2.0

>>> d['sqrt']

1

exec还可以带多个参数，比如：

exec(expr, globals) 其中globals必须是字典

通过带两个参数的形式也可以解决命名冲突，实例：

>>> from math import sqrt

>>> ff = {}

>>> exec("sqrt=1",ff)

>>> sqrt(4)

2.0

>>> ff['sqrt']

1

注意，如果需要将scope打印出来的话，会看到其中包含很多东西，因为內建的\_\_builtins\_\_字典自动包含所有的內建函数和值：

>>> len(ff)

2

>>> ff.keys()

['\_\_builtins\_\_', 'sqrt']

书本第85 - 86页

#### eval

eval() 函数用来执行一个字符串表达式，并返回表达式的值。

eval 语法：

eval(expression[, globals[, locals]])

eval会计算Python表达式（以字符串形式书写），并返回结果。

有三个参数，表达式字符串；globals变量作用域，全局命名空间（必须是字典）；locals变量作用域，局部命名空间（可以是任何形式的映射）。 其中第二个和第三个参数是可选的。

如果忽略后面两个参数，则eval在当前作用域执行。

不带参数实例，执行操作，并返回结果：

>>> a= 1

>>> eval("a+1")

2

如果指定globals参数

>>> f = 1

>>> dict = {"f":10}

>>> eval("f+1",dict)

11

>>> f

1

>>> dict['f']

10

如果指定locals参数

>>> a=10

>>> b=10

>>> c=20

>>> g={"a":6,"b":8}

>>> l={"b":9,"c":10}

>>> eval("a+b+c",g,l)

25

>>> a

10

>>> b

10

>>> c

20

书本第86 - 87页

##### 初探作用域

给exec或者eval语句提供命名空间时，还可以在真正使用命名空间前放置一些值进去：

>>> scope = {}

>>> scope["x"] = 2

>>> scope["y"] = 3

>>> eval("x \* y", scope)

6

同理，exec或者eval调用的作用域也能在另一个上面使用:

>>> scope = {}

>>> exec "x = 2" in scope

>>> eval("x \* x", scope)

4

事实上，exec语句和eval语句并不常用，但是它们可以作为后兜里的得力工具(当然，这仅仅是比喻而已)。

书本第87页