

REVIEWED

By Chris Yan at 4:01 pm, Jul 29, 2007

• WeiZhong/2006-01-17

Python 精要参考(第二版) [Python Essential Reference, Second Edition](#) 译文

原著: David M Beazley 出版商: New Riders Publishing

初译: Feather andelf@gmail.com 修正补充: WeiZhong weizhong2004@gmail.com

1. 第一章 Python快速入门

本章是Python的快速入门,在这一章并不涉及python的特殊规则和细节,目标是通过示例使你快速了解Python语言的特点。本章简要介绍了变量,表达式,控制流,函数以及输入/输出的基本概念,在这一章不涉及Python语言的高级特性。尽管如此,有经验的程序员还是能够通过阅读本章的材料创建高级程序。我们鼓励新手通过运行示例,亲身体验一把Python。

1.1. 运行Python

Python 程序通过解释器执行。如果你的机器已经装好了python,简单的在命令行键入python即可运行python解释器。在解释器运行的时,会有一个命令提示符 >>>,在提示符后键入你的程序语句,键入的语句将会立即执行。在下边的例子中,我们在>>>提示符后边键入最常见的显示"Hello World"的命令:

```
Python 2.4.2 (#67, Sep 28 2005, 12:41:11) [MSC v.1310 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> print "Hello World"
Hello World
>>>
```

程序也可以像下面一样放置在一个文件中

```
# helloworld.py
print "Hello World"
```

Python源代码文件使用.py后缀.'#'表示注释(到行末结束)

执行文件helloworld.py

```
% python helloworld.py
Hello World
%
```

目录

1. 第一章 Python快速入门
 1. 运行Python
 2. 变量和表达式
 3. 条件语句
 4. 文件输入/输出
 5. 字符串
 6. 列表和元组(Lists & Tuples)
 7. 循环
 8. 字典
 9. 函数
 10. 类
 11. 异常
 12. 模块

在Windows下,只需双击一个.py文件就能执行这个python程序。windows会自动调用python解释程序,然后启动一个终端窗口(类DOS窗口)来执行它。在这种情况下,

终端窗口会在程序执行完毕后立即关闭(经常是在你看到它的输出之前)。为避免这个问题,你可以使用python集成开发环境,例如IDLE或Pythonwin。另一个可行的方法是建立一个bat文件,在文件写入这样一行语句,如 `python -i helloworld.py`。运行这个批处理,程序在执行完成后会自动进入python解释器。

在解释器中,也可以通过函数`execfile()`来运行一个保存在磁盘上的程序,如下例:

```
>>> execfile("helloworld.py")
Hello World
```

在UNIX下,你可以在程序的首行写入 `#!` 魔法字符串 来自动调用python解释器执行你的脚本。

```
#!/usr/local/bin/python
print "Hello World"
```

解释器会一直运行直到文件结束。如果在交互模式下,键入 EOF字符退出解释器。在UNIX下,EOF字符是Ctrl+ D;在Windows下,EOF字符是Ctrl+Z。也可以在程序中使用`sys.exit()`函数或者通过引发`SystemExit`异常来退出程序:

```
>>> import sys
>>> sys.exit()
```

或者

```
>>> raise SystemExit
```

1.2. 变量和表达式

通过Listing 1.1所示的程序示例变量和表达式的用法

Listing 1.1 复利计算器(Simple Compound-Interest Calculation)

```
Toggle line numbers
1 principal = 1000           # Initial amount (本金)
2 rate = 0.05                # Interest rate (利率)
3 numyears = 5               # Number of years (期数,年)
4 year = 1
5 while year <= numyears:
6     principal = principal*(1+rate)
7     print year, principal
8     year += 1
```

程序输出:

```
1 1050.0
2 1102.5
3 1157.625
4 1215.50625
5 1276.2815625
```

Python 是一种动态语言,在程序运行过程中,同一变量名可以(在程序运行的不同阶

段) 代表不同形式的值(整型,浮点,列表,元组...)。事实上, 程序中使用的变量名只是各种数据及对象的引用。这与C语言不同,C语言中变量名代表的是用来存放结果的一个固定位置及长度的内存片段。从例子Listing 1.1中的变量principal可以看出Python语言的动态特性.最初,它被赋值为一个整数,但是稍后程序将它再次赋值:

```
principal = principal*(1+rate)
```

这个语句计算表达式的值, 然后将计算结果赋给 principal 变量做为它的新值。当赋值动作发生时,principal最初绑定的值1000被丢弃。赋值结束, 不但 principal 绑定的值发生了变化, 它的类型也随着赋值动作发生了相应的变化。在这个例子中, 由于rate是一个浮点数,所以在赋值完成后,principal也变成一个浮点数。

Python中每个语句以换行结束,当然你也可以在一行中写多个语句, 这时语句之间必须使用分号分隔, 就象下面这样:

```
principal = 1000; rate = 0.05; numyears = 5;
```

(建议这样的写法仅仅用于调试语句, 因为可以很方便的只删一行就删掉全部调试语句)

while 语句首先检查在它后边的循环条件,若条件表达式为真,它就执行冒号后面的语句块, 然后再次测试循环条件, 直至为假。冒号后面的缩进语句块为循环体。注意, Python语言使用缩进块来表示程序逻辑(其它大多数语言使用大括号等)。在Listing 1.1中while语句后的三条语句为循环体, 在每次循环中均执行。Python并未指定缩进的空白(空格和制表符)数目, 唯一的要求是同一层次的语句必须有相同的缩进空白。(注意, 要么都是空格, 要是么都制表符, 千万别混用)

Listing 1.1中的程序美中不足的就是输出不是很好看, 为了让它美观一点,可以用格式字符串将计算结果只保留小数点后两位:

```
print "%3d %0.2f" % (year, principal)
```

这样,程序的输出就变为:

```
1 1050.00
2 1102.50
3 1157.63
4 1215.51
5 1276.28
```

格式字符串包含普通文本及格式化字符序列(例如"%d", "%s", 和 "%f"),这些序列决定特定类型的数据(如整型,字符串,浮点数)的输出格式.'%3d'将一个整数在宽度为3个字符的栏中右对齐,%0.2f将一个浮点数的小数点后部分转换为2位。格式字符串的作用和C语言中的sprintf()函数基本相同。详细内容请参阅第四章--操作符及表达式。

1.3. 条件语句

if和else语句用来进行简单的测试, 如:

Toggle line numbers

```
1 # Compute the maximum (z) of a and b (得到a与b中较大的一个)
2 if a < b:
3     z = b
4 else:
5     z = a
```

if和else的语句块用缩近来表示，else从句在某些情况下可以省略。如果if或else语句块只有一个语句，也可以不使用缩进。也就是说：

Toggle line numbers

```
1 if a<b: z=a
2 else: z=b
```

这样的写法也是合法的，但这不是推荐的作法。一直使用缩进可以让你方便的在语句体中添加一个语句，而且读起来更清晰。若某个子句不需任何操作,就使用pass语句，如：

Toggle line numbers

```
1 if a < b:
2     pass      # Do nothing
3 else:
4     z = a
```

通过使用 or,and 和 not 关键字你可以建立任意的条件表达式：

Toggle line numbers

```
1 if b >= a and b <= c:
2     print "b is between a and c"
3 if not (b < a or b > c):
4     print "b is still between a and c"
```

用 elif 语句可以检验多重条件(用于代替其它语言中的switch语句)：

Toggle line numbers

```
1 if a == '+':
2     op = PLUS
3 elif a == '-':
4     op = MINUS
5 elif a == '*':
6     op = MULTIPLY
7 else:
8     raise RuntimeError, "Unknown operator"
```

1.4. 文件输入/输出

下面的程序打开一个文件,然后一行行地读出并显示文件内容：

Toggle line numbers

```
1 f = open("foo.txt")          # Returns a file object
2 line = f.readline()         # Invokes readline() method on file
3 while line:
4     print line,              # trailing ',' omits newline character
5     line = f.readline()
6 f.close()
```

open()函数返回一个新文件对象(file object)。通过调用此对象的不同方法可以对文件进行不同的操作。readline()方法读取文件的一行(包括换行符'\n')。如果读到文件末尾，就返回一个空字符串。要将程序的输出内容由屏幕重定向到文件中，可以使用'>>'运算符，如下例：

Toggle line numbers

```
1 f = open("out","w")      # Open file for writing
2 while year <= numyears:
3     principal = principal*(1+rate)
4     print >>f,"%3d %0.2f" %% (year,principal) #将格式文本输出到
文件对象 f
5     year += 1
6 f.close()
```

当然,文件对象也拥有write()方法,通过它可以向文件对象写入新的数据。例如上边例子中的print的语句也可以写成这样:

```
f.write("%3d %0.2f\n" % (year,principal))
```

1.5. 字符串

要创建一个字符串,你使用单引号,双引号或三引号将其引起来,如下例:

Toggle line numbers

```
1 a = 'Hello World'
2 b = "Python is groovy"
3 c = """What is footnote 5?"""
```

一个字符串用什么引号开头,就必须用什么引号结尾。两上三引号之间的一切都为字符串的内容,对应的单引号与双引号却只能创建单行字符串。如下例:

Toggle line numbers

```
1 print '''Content-type: text/html
2
3 <h1> Hello World </h1>
4 Click <a href="http://www.python.org">here</a>.
5 '''
```

字符串是一个以0开始,整数索引的字符序列,要获得字符串s中的第i+1个字符(别忘了0是第一个),使用索引操作符s[i]:

Toggle line numbers

```
1 a = "Hello World"
2 b = a[4]          # b = 'o'
```

要获得一个子串,使用切片运算符s[i:j]。它返回字符串s中从索引i(包括i)到j(不包括j)之间的子串。若i被省略,python就认为i=0,若j被省略,python就认为j=len(s)-1:

Toggle line numbers

```
1 c = a[0:5]      # c = "Hello"
2 d = a[6:]      # d = "World"
3 e = a[3:8]     # e = "lo Wo"
```

可以用加(+)运算符来连结字符串:

```
g = a + " This is a test"
```

通过使用str()函数,repr()函数或向后的引号()可以将其他类型的数据转换为字符串:

Toggle line numbers

```
1 s = "The value of x is " + str(x)
2 s = "The value of y is " + repr(y)
3 s = "The value of y is " + `y`
```

`repr()`函数用来取得对象的规范字符串表示, 向后的引号(```)是`repr()`函数的快捷版。在大多情况下`str()`和`repr()`函数会返回同一个结果,但是它们之间有很微妙的差别,后边的章节对此将有详细描述。

1.6. 列表和元组 (Lists & Tuples)

就如同字符串是字符的序列,列表和元组则是任意对象的序列。象下面这样就可以创建一个列表:

```
names = [ "Dave", "Mark", "Ann", "Phil" ]
```

列表和元组都是以整数0来开始索引的序列,你可以用索引操作符来读取或者修改列表中特定元素的值:

```
a = names[2]           # Returns the third item of the list, "Ann"
names[0] = "Jeff"     # Changes the first item to "Jeff"

用len()函数得到列表的长度:

print len(names)      # prints 4

append()方法可以把一个新元素插入列表的末尾:

names.append("Kate")

aList.insert(index,aMember)方法可以把新元素 aMember 插入到列表 aList[index]
元素之前:

names.insert(2, "Sydney")

用切片操作符可以取出一个子列表或者对子列表重新赋值:

b = names[0:2]         # Returns [ "Jeff", "Mark" ]
c = names[2:]         # Returns [ "Sydney", "Ann",
"Phil", "Kate" ]
names[1] = 'Jeff'     # Replace the 2nd item in names
with "Jeff"
names[0:2] = ['Dave','Mark','Jeff'] # 用右边的 list 替换 names 列表中的前两
个元素

加(+)运算符可以连结列表:

a = [1,2,3] + [4,5]   # Result is [1,2,3,4,5]

列表元素可以是任意的 Python 对象,当然也包括列表:

a = [1,"Dave",3.14, ["Mark", 7, 9, [100,101]], 10]
```

子列表的元素用下面的方式调用：

```
a[1]          # Returns "Dave"  
a[3][2]      # Returns 9  
a[3][3][1]   # Returns 101
```

Listing 1.2中代码从一个文件中读取一系列数字，然后输出其中的最大值和最小值。通过这个示例我们可以了解到列表的一些高级特性：

Listing 1.2 列表的高级特性

```
Toggle line numbers  
1 import sys          # Load the sys module (导入sys模  
块)  
2 f = open(sys.argv[1]) # Filename on the command line  
(从命令行读取文件名)  
3 svalues = f.readlines() # Read all lines into a list (读出  
所有行到一个列表)  
4 f.close()  
5  
6 # Convert all of the input values from strings to floats (把输入的值  
转换为浮点数)  
7 fvalues = [float(s) for s in svalues]  
8  
9 # Print min and max values (输出最大值和最小值)  
10 print "The minimum value is ", min(fvalues)  
11 print "The maximum value is ", max(fvalues)
```

程序第一行用import语句从Python library中导入sys模块。

你需要在命令行提供一个文件名给上面的程序，该文件名参数保存在sys.argv 列表中，open方法通过读取sys.argv[1]得到这个文件名参数。

readlines()方法读取文件中的所有行到一个列表中。

表达式 [float(s) for s in svalues] 通过循环列表svalues中的所有字符串并对每个元素运行函数float()来建立一个新的列表,这种特殊的建立列表的方法叫做列表包含(list comprehension)。在列表中所有的字符串都转换为浮点数之后,内建函数min()和max()计算出列表中的最大值及最小值。

元组(tuple)类型和列表关系很密切,通过用圆括号中将一系列逗号分割的值括起来可以得到一个元组:

```
Toggle line numbers  
1 a = (1,4,5,-9,10)  
2 b = (7,)          # 一个元素的元组 (注意一定要  
加一个额外的逗号! )  
3 person = (first_name, last_name, phone)
```

在某些时候，即使没有圆括号，Python仍然可以根据上下文认出这是一个元组，如：(为了写出更清晰可读的程序，建议你不要依赖 Python 的智能)

```
a = 1,4,5,-9,10  
b = 7,  
person = first_name, last_name, phone
```

元组支持大多数列表的操作,比如索引,切片和连结。一个关键的不同是你不能在一个tuple创建之后修改它的内容。也就是说,你不能修改其中的元素,也不能给tuple添加新的元素。

1.7. 循环

通过使用while语句,我们在前面已经简单介绍了 while 循环。在Python中另一种循环结构是 for 循环,它通过迭代一个序列(例如字符串,列表,或者tuple等)中的每个元素来建立循环。下边是一个例子:

```
for i in range(1,10):
    print "2 to the %d power is %d" % (i, 2**i)
```

range(i,j)函数建立一个整数序列,这个序列从第 i 数开始(包括 i)到第 j 数为止(不包括 j)。若第一个数被省略,它将被认为是0。该函数还可以有第三个参数,步进值,见下面的例子:

```
a = range(5)           # a = [0,1,2,3,4]
b = range(1,8)        # b = [1,2,3,4,5,6,7]
c = range(0,14,3)     # c = [0,3,6,9,12]
d = range(8,1,-1)     # d = [8,7,6,5,4,3,2]
```

for语句可以迭代任何类型的序列:

```
a = "Hello World"
# Print out the characters in a
for c in a:
    print c
b = ["Dave","Mark","Ann","Phil"]
# Print out the members of a list
for name in b:
    print name
```

range()函数根据起始值,终止值及步进值三个参数在内存中建立一个列表,当需要一个很大的列表时,这个既占内存又费时间。为了克服它的缺点,Python提供了 xrange()函数:

```
for i in xrange(1,10):
    print "2 to the %d power is %d" % (i, 2**i)

a = xrange(100000000)    # a = [0,1,2, ..., 99999999]
b = xrange(0,100000000,5) # b = [0,5,10, ...,99999995]
```

xrange()函数只有在需要值时才临时通过计算提供值,这大大节省了内存。

1.8. 字典

字典就是一个关联数组(或称为哈希表)。它是一个通过关键字索引的对象的集合。使用大括号{}来创建一个字典,如下例:


```
a = {
    "username" : "beazley",
    "home" : "/home/beazley",
    "uid" : 500
}
```

用关键字索引操作符可以访问字典的某个特定值:

```
u = a["username"]
d = a["home"]
```

用下面的方式插入或者修改对象:

```
a["username"] = "pxl"
a["home"] = "/home/pxl"
a["shell"] = "/usr/bin/tcsh"
```

尽管字符串是最常见的 关键字(key) 类型, 你还是可以使用很多其它的 python 对象做为字典的关键字, 比如 数字 和 **tuple**, 只要是不可修改对象, 都可以用来做字典的key。有些对象,例如列表和字典,不可以用来做字典的key,因为他们的内容是允许更改的。

我们可以使用 `has_key()` 方法来检验一个键/值对是否存在(或者`in`操作符):

```
if a.has_key("username"):
    username = a["username"]
else:
    username = "unknown user"
```

上边的操作还可以用更简单的方法完成:

```
username = a.get("username", "unknown user")
```

字典的`keys()` 方法返回由所有关键字组成的列表:

```
k = a.keys()          # k = ["username", "home", "uid", "shell"]
```

`del`语句可以删除字典中的特定元素:

```
del a["username"]
```

1.9. 函数

在Python中, 使用`def`语句来创建函数, 如下例:

```
def remainder(a,b):
    q = a/b
    r = a - q*b
    return r
```

要调用一个函数，只要使用函数名加上用括号括起来的参数就可以了。比如`result = remainder(37,15)`，如果你打算让函数返回多个值，就让它返回一个元组好了。（当然，只要你愿意，让它返回一个列表我们也不会介意）

```
def divide(a,b):
    q = a/b          # If a and b are integers, q is an integer
    r = a - q*b
    return (q,r)
```

当返回一个 `tuple` 时，你会发现象下面这样调用函数会很有用：

```
quotient, remainder = divide(1456,33)
```

你也可以象下面这样给函数的参数指定一个默认值：

```
def connect(hostname,port,timeout=300):
    # Function body
```

若在函数定义的时候提供了默认参数，那么在调用函数时就允许省略这个参数：

```
connect('www.python.org', 80)
```

你也可以使用关键字参数来调用函数，这样你的参数就可以使用任意顺序：

```
connect(port=80,hostname="www.python.org")
```

函数内部定义的变量为局部变量，要想在一个函数内部改变一个全局变量的值，在函数中使用`global`语句：

```
Toggle line numbers
1 a = 4.5
2 ...
3 def foo():
4     global a
5     a = 8.8          # 改变全局变量 a
```

1.10. 类

Python支持面向对象编程，在面向对象编程中，`class`语句用于定义新类型的对象。例如，下面这个类定义了一个简单的堆栈：

```
Toggle line numbers
1 class Stack(object):
2     def __init__(self):          # 初始化栈
3         self.stack = [ ]
4     def push(self,object):
5         self.stack.append(object)
6     def pop(self):
7         return self.stack.pop()
8     def length(self):
9         return len(self.stack)
```

在类定义中，方法用 `def` 语句定义。类中每个方法的第一个参数总是引用类实例对象本身，大家习惯上使用 `self` 这个名字代表这个参数。不过这仅仅是个习惯而已，如果你愿意也可以用任意的别的名字。不过为了别人容易看懂你的程序，最好还是跟

随大家的习惯。类的方法中若需要调用实例对象的属性则必须显式使用self变量(如上所示)。方法名中若前后均有两个下划线,则表示这是一个特殊方法,比如init方法被用来初始化一个对象(实例)。

象下面这样来使用一个类:

Toggle line numbers

```
1 s = Stack()           # Create a stack (创建)
2 s.push("Dave")       # Push some things onto it (写入)
3 s.push(42)
4 s.push([3,4,5])
5 x = s.pop()          # x gets [3,4,5] (读取)
6 y = s.pop()          # y gets 42
7 del s                # Destroy s (删除)
```

1.11. 异常

如果在你的程序发生了一个错误,就会引发异常(exception),你会看到类似下面的错误信息:

```
Traceback (most recent call last):
  File "<interactive input>", line 42, in foo.py
NameError: a
```

错误信息指出了发生的错误类型及出错位置,通常情况下,错误会导致程序终止。不过你可以使用 try 和 except 语句来捕获并处理异常:

```
try:
    f = open("file.txt","r")
except IOError, e:
    print e
```

上面的语句表示:如果有 IOError 发生,造成错误的详细原因将会被放置在对象 e 中,然后运行 except 代码块。若发生其他类型的异常,系统就会将控制权转到处理该异常的 except 代码块,如果没有找到该代码块,程序将运行终止。若没有异常发生,except 代码块就被忽略掉。

raise 语句用来有意引发异常,你可以使用内建异常来引发异常,如下例:

```
raise RuntimeError, "Unrecoverable error"
```

当然,你也可以建立你自己的异常,这将在第五章--控制流中的定义新的异常中详细讲述。

1.12. 模块

当你的程序变得越来越大,为了便于修改和维护,你可能需要把它们分割成多个相关文件。Python 允许你把函数定义或公共部分放入一个文件,然后在其他程序或者脚本中将该文件作为一个模块导入。要创建一个模块,把相应的语句和定义放入一个文件,这个文件名就是模块名。(注意:该文件必须有.py 后缀):

Toggle line numbers

```
1 # file : div.py
2 def divide(a,b):
3     q = a/b          # If a and b are integers, q is an integer
4     r = a - q*b
5     return (q,r)
```

要在其它的程序中使用这个模块，使用import语句:

```
import div
a, b = div.divide(2305, 29)
```

import语句创建一个新的名字空间，该空间包含模块中所有定义对象的名称。要访问这个名字空间，把模块名作为一个前缀来使用这个模块内的对象，就像上边例子中那样:div.divide()

如果你希望使用一个不同的模块名字访问这个模块，给import语句加上一个 as 模块名 部分就可以了:

```
import div as foo
a,b = foo.divide(2305,29)
```

如果你只想导入指定的对象到当前的名称空间,使用 from 语句:

```
from div import divide
a,b = divide(2305,29)      # No longer need the div prefix (不再需要div前缀)
```

导入一个模块中的所有内容到当前的名称空间:

```
from div import *
```

最后，内建函数dir()可以列出一个模块中的所有可访问内容。当你在python交互环境中测试一个模块的功能时，这会是一个很有用的工具，因为它可以提供一个包含可用函数及变量的列表:

```
>>> import string
>>> dir(string)
['_ _builtins_ _', '_ _doc_ _', '_ _file_ _', '_ _name_ _', '_idmap',
'_idmapL', '_lower', '_swapcase', '_upper', 'atof', 'atof_error',
'atoi', 'atoi_error', 'atol', 'atol_error', 'capitalize',
'capwords', 'center', 'count', 'digits', 'expandtabs', 'find',
...
>>>
```

• WeiZhong/2006-01-18

Python精要参考第二章

Python 精要参考(第二版) Python Essential Reference, Second Edition 译文

原著:David M Beazley 出版商: New Riders Publishing

初译: Feather andelf@gmail, com 修正补充: WeiZhong weizhong2004@gmail, com

1. 第二章 语法及代码约定

目录

1. 第二章 语法及代码约定
 1. 行结构/缩进
 2. 标识符及保留字
 3. 数字/文字
 4. 运算符、分隔符及特殊符号
 5. 文档字符串

本章讲述了Python程序的语法和代码约定。本章的主题有行结构, 语句分组, 保留字, 字符串, 运算符, token等等, 另外对如何使用Unicode字符串也做了详细的描述。

1.1. 行结构/缩进

程序中的每个语句都以换行符结束。特别长的语句可以使用续行符(\)来分成几个短小的行, 如下例:

```
import math
a = math.cos(3*(x-n)) + \
    math.sin(3*(y-n))
```

当你定义一个三引号字符串、列表、tuple 或者字典的时候不需要续行符来分割语句。也就是说, 在程序中, 凡是圆括号(, , ,), 方括号[, , ,], 花括号{, , , } 及三引号字符串内的部分均不需要使用续行符。

缩进被用来指示不同的代码块, 比如函数的主体代码块, 条件执行代码块, 循环体代码块及类定义代码块。缩进的空格(制表符)数目可以是任意的, 但是在整个块中的缩进必须一致:

Toggle line numbers

```
1 if a:
2     statement1      # 缩进一致, 正确!
3     statement2
4 else:
5     statement3
6     statement4     #缩进不一致, 错误!
```

如果块中只有很少的语句, 那么你也可以把它们放置在同一行:

```
if a: statement1
else: statement2
```

要表示一个空的块或是空的主体, 使用 pass语句:

```
if a:
```

```
    pass
else:
    statements
```

尽管允许用制表符指示缩进，我还是要说这是一个不好的习惯。坚决不要混合使用制表符和空格来缩进，这会给你带来意想不到的麻烦。建议你在每个缩进层次中使用单个制表符或两个或四个空格。运行 Python 的时候使用 `-t` 参数，如果 `python` 发现存在制表符和空格混用，它就显示警告信息，若使用 `-tt` 参数 `python` 则会在遇到混用情况时引发 `TabError` 异常。

分号(;)可以把多个语句放在同一行中，只有一个语句的行也可以用分号来结束。

`#`指示这是一个延长至行末的注释，但是包在字符串内的`#`没有这个功能。

最后要说明的，解释器会忽略所有的空白行(非交互模式下)。

1.2. 标识符及保留字

标识符是用于识别变量、函数、类、模块以及其他对象的名字，标识符可以包含字母、数字及下划线(`_`)，但是必须以一个非数字字符开始。字母仅仅包括ISO-Latin字符集中的A-Z和a-z。标识符是大小写敏感的，因此 `FOO`和`foo`是两个不同的对象。特殊符号，如`$`、`%`、`@`等，不能用在标识符中。另外，如 `if`，`else`，`for` 等单词是保留字，也不能将其用作标识符。下面的表列出了所有的保留字符：

<code>and</code>	<code>elif</code>	<code>global</code>	<code>or</code>
<code>assert</code>	<code>else</code>	<code>if</code>	<code>pass</code>
<code>break</code>	<code>except</code>	<code>import</code>	<code>print</code>
<code>class</code>	<code>exec</code>	<code>in</code>	<code>raise</code>
<code>continue</code>	<code>finally</code>	<code>is</code>	<code>return</code>
<code>def</code>	<code>for</code>	<code>lambda</code>	<code>try</code>
<code>del</code>	<code>from</code>	<code>not</code>	<code>while</code>

以下划线开始或者结束的标识符通常有特殊意义。例如以一个下划线开始的标识符(如 `_foo`)不能用 `from module import *` 语句导入。前后均有两个下划线的标识符，如 `__init__`，被特殊方法保留。前边有两个下划线的标识符，如 `__bar`，被用来实现类私有属性，这个将在第七章--类与面向对象编程中讲到。通常情况下，应该避免使用相似的标识符。

1.3. 数字/文字

Python中有四种内建的数值类型:整数、长整数、浮点数和复数。

象1234这样的数被解析为一个十进制的整数。要指定一个八进制或者十六进制的整数，在一个合法的八进制数前加上 `0` 或者在一个合法的16进制数前加上 `0x` 就可以了。(如 `0644` 和 `0x100fea8`)。在一个整数后面加上字母 `l` 或 `L` 系统就认为这是一个长整数(如 `1234567890L`)。与受机器字长限制整数类型不同，长整数可以是任何长度(只受内存大小限制)。象 `123.34`和`1.2334e+02`这样的数被解析为浮点数。一个整数或者浮点数加上后缀 `J` 或者 `j` 就构成了一个复数的虚部，你可以用一个实数加上一个虚部创建一个复数，比如 `1.2 + 12.34J`。

Python目前支持两种类型的字符串:

8位字符数据 (ASCII)

16位宽字符数据 (Unicode)

最常用的是ASCII字符串，因为这个字符集刚好只用一个字节就可以字符集中的任意一个字符。通常情况下，ASCII串用单引号(')，双引号(")，或者三引号(''' 或 """)来定义。字符串前后的引号类型必须一致。反斜杠(\)用来转义特殊字符，比如换行符、反斜杠本身、引号以及其他非打印字符。Table 2.1中列出了公认的特殊字符的表示方法，无法识别的转义字符串将被原样保留(包括前边的反斜杠)。此外，字符串可以包含嵌入的空字节和二进制数据。三引号字符串中可以包含不必转义的换行符和引号。

Table 2.1 Standard Character Escape Codes

标准特殊字符	
字符	描述
\	续行符
\\	反斜杠
\'	单引号
\"	双引号
\a	Bell(音箱发出吡的一声)
\b	退格符
\e	Escape
\0	Null(空值)
\n	换行符，等价于\x0a和\cJ
\v	垂直制表符，等价于\x0b和\cK
\t	水平制表符，等价于\x09和\cI
\r	回车符，等价于\x0d和\cM
\f	换页符，等价于\x0c和\cL
\OOO	八进制值(000-377)
\xhh	十六进制值(x00-xff)
\un	Unicode字符值，n是四个十六进制数字表示的Unicode字符

Unicode 字符串用来表示多字节国际字符集，它包括65, 536个字符。Unicode字符使用u或者U前缀来定义，例如`a = u"hello"`。在Unicode字符集中，每一个字符用一个16位整数来表示。Unicode字符使用 U+xxxx 这种形式来表示，xxxx是一个由 4 个十六进制数字组成的16进制数。(注意：这种记法只是一个表示Unicode字符的习惯，并不是Python的语法)。例如U+0068是Unicode字符字母h(在Latin-1字符集中，你可以发现Unicode字符集的前256个字符与Latin-1的对应字符编码完全相同)。当Unicode字符串被赋值时普通字符和特殊字符都直接转换成Unicode字符序数(在[U+0000, U+00FF]中)。例如，字符串"hello\n"映射为ASCII时是:0x68, 0x65, 0x6c, 0x6c, 0x6f, 0x0a, 当使用u"hello\n"转换为Unicode字符串时是:U+0068, U+0065, U+006C, U+006C, U+006F, U+000A.任意Unicode字符都可以使用\uxxxx来定义，\uxxxx必须位于一个Unicode字符串中，例如：

```
s = u"\u0068\u0065\u006c\u006c\u006f\u000a"
```

在Python 的较老版本中, \xXXXX字节序列被用来定义Unicode字符(这与系统识别Unicode字符方式有关)。虽然现在仍然允许这样做, 我仍然建议你最好采用新的表示方法。(因为旧的表示方法随时可能废止。)另外, 八进制代码\ooo也可以用来定义在[U+0000, U+01FF]中的Unicode字符。

Unicode字符不能通过使用 UTF-8或者UTF-16编码中的原始字节序列来定义。例如, UTF-8编码的字符串 u'M\303\274ller' 建立的七个字符, 用Unicode表示为+004D, U+00C3, U+00BC, U+006C, U+006C, U+0065, U+0072, 这并不是你想要的结果。这是因为在UTF-8中, 多字节序列\303\274用来代表U+00FC, 而不是U+00C3, U+00BC。更多关于Unicode编码细节你可以阅读第三章--"类型和对象", 第四章--"运算符和表达式", 第九章--"输入和输出".

你可以给一个字符串加上前缀r或者R, 例如 `r'\n\'`, 这些字符串被称为原始字符串, 因为里边几乎所有的特殊字符都会原封不动地留下。不过原始字符串并不能以一个单独的反斜杠结尾(例如 r"\"). 如果原始字符串使用ur或者UR前缀来定义的话, \uXXXX仍然会被解析为Unicode字符。如果你不想这样, 你可以在它前边再加一个反斜杠, 例如ur"\u1234", 它定义了一个含有7个字符的字符串。需要注意的是, 定义原始Unicode字符串时, r必须在u之后。

邻近的字符串(被空格或者续行符分割), 例如 "hello" 'world' 会被Python自动连结为一个字符串 "helloworld". 无论是普通字符串, Unicode字符串, 还是自然字符串, 都会自动连结。当然, 只要这些字符串中有一个是Unicode字符串, 那最终连结的结果也将是一个Unicode字符串。比如 "s1" u"s2" 就会产生 u"s1s2". 这个过程的细节你可以阅读第四章和附录A(the Python library).

如果Python在 -U 命令行参数下运行, 所有的字符都会被解析为Unicode。

方括号[...]定义一个列表, 圆括号(...)定义一个元组, 花括号{...}定义一个字典:

```
a = [ 1, 3.4, 'hello' ]           # A list
b = ( 10, 20, 30 )               # A tuple
c = { 'a': 3, 'b':42 }           # A dictionary
```

1.4. 运算符、分隔符及特殊符号

Python 目前支持以下运算符:

+	-	*	**	//	/	%	<<	>>
&		^						
+=	-=	*=	**=	//=	/=	%=	<<=	>>=
&=	=	^=						
~	<	>	<=	>=	==	!=	<>	

下边的这些可以作为表达式, 列表, 字典, 以及语句不同部分的分隔符号:

()	[]	{	}	,	:	.	`	=
;										

比如, 等号(=), 作为对象名和所分配值之间的分隔符;逗号(,)用来分隔函数参数、列表或tuple中的元素;小数点(.)用在浮点数和扩展切片操作中的省略符(...),

下边这些特殊符号也在语句中使用:

'"#\@

注: Python 2.4 中新增了 @ 符号用作函数修饰符 ---Wei Zhong

字符\$, ?不能在程序语句中出现, 但是可以出现在字符串中。

1.5. 文档字符串

如果一个模块、类、或函数体的第一个语句是未命名一个字符串, 该字符串就自动成为该对象的文档字符串(DocStrings),如下例:

Toggle line numbers

```
1 def fact(n):
2     "This function computes a factorial"
3     if (n <= 1): return 1
4     else: return n*fact(n-1)
```

代码浏览及文档生成工具经常用到文档字符串。通过访问一个对象的__doc__属性, 你可以得到文档字符串:

这个 __doc__ 属性竟然是可写的。--WeiZhong

```
>>> print fact.__doc__
This function computes a factorial
>>>
```

文档字符串的缩进必须与定义中的其他语句一致。此外, 在不同行出现的多个未命名字符串不会自动连结成一个字符串, 即使他们紧挨着。(注意:返回的文档字符串仅仅是第一个字符串, 这与前边讲到的字符串的自动连结有所不同, 注意区别)。

WeiZhong/2006-01-18 (2006-01-18 07:11:12由WeiZhong编辑)

1. 第三章 类型和对象

Python 程序中的一切数据都是对象。对象包括自定义对象及基本的数据类型如数值、字符串、列表、字典等。你能够以类或扩展类型的方式创建自定义对象。本章主要描述 Python对象模型及第四章--运算符与表达式中要用到的一些预备知识。

1.1. 术语

程序中的一切数据都是对象，每个对象都有三个基本属性，即标识(类似人的标识证号)、类型和值。

例如，当你写下 `a = 42` 这行代码，你就创建了一个值为 42 的整数对象。`type()`和`id()`函数用来查看对象的类型标识。`id(a)`可以查看该对象的标识(当前的实现是该对象在内存中的位置)。在这个例子中，`a`就是这个位置的引用。

一个对象的类别决定了可以对该对象进行何种操作(如，这个对象有长度吗?)。当一个特定类型的对象被创建时，这个对象被称为该类型的一个实例(注意：不要将类型的的实例和用户自定义类的实例混淆)。在一个对象被创建之后，它的标识和类型就再不能被改变。某些对象的值是可变的，这些对象就被称为可变对象(`mutable`);另一些对象的值是不可变的，那就被称为不变对象(`immutable`)。某类对象可以包含其它对象的引用，我们称这类对象为容器。

注1:关于类型的不可改变

从python2.2开始，Python开发小组开始有步骤的合并某些类别和类，因此书中的某些结论可能不是百分之百精确和完整。在某些特定条件下，有可能允许改变一个对象的类型。但是，在本手册扩展修订之前，我们就应该一如既往的认为这些经典类型是不可改变的。考虑到兼容性，python2.2和2.3也是这样默认处理的。

注2:不变对象的不可变并不是绝对的，当一个不变容器对象包含一个可变对象的引用时，可变对象的值变化会引起该不变容器对象的值发生变化。这种情况下，我们仍然认为该容器对象为不变对象，因为该容器所包含并不是引用对象的值，而仅仅是该对象的引用，这里的引用可以理解为该对象的内存地址。不管被包含对象的值如何变化，被包含对象的引用确实是始终不变的)。一个

目录

1. 第三章 类型和对象
 1. 术语
 2. 对象的标识与类型
 3. 引用计数与垃圾收集
 4. 引用与副本
 5. 内建类型
 1. None类型
 2. 数值类型
 3. 序列类型
 4. 字符串类型
 5. xrangeType 类型
 6. 缓冲区类型
 7. 映射类型
 8. 可调用类型
 1. 用户定义函数
 2. 用户定义方法
 3. 类和可调用的类实例
 4. 内建函数及内建方法
 9. 模块类型
 10. 类类型
 11. 类实例类型
 12. 文件类型
 13. 内部类型
 1. 代码对象
 2. Frame 对象
 3. traceback 对象
 4. 切片对象
 5. 省略对象
 6. 特殊方法
 1. 对象创建、销毁及表示
 2. 属性访问
 3. 序列和映射的方法
 4. 数学操作
 5. 比较操作
 6. 可调用对象
 7. 性能及内存占用

对象是否可变取决于它的类型，举例来说，数字、字符串、tuple类型是不可变类型，字典与列表是可变类型。

--WeiZhong

除了保存值以外，许多对象还拥有一系列的属性(attribute)。广义的属性是指对象的相关数据或者该对象能够具有的行为（如狗对象拥有颜色体重等相关数据，还拥有叫、吃、跑等行为，这些都是对象的广义的属性），狭义的属性只包含对象的相关数据，对于对象的行为，更常用的叫法是方法(method)。方法是对象可调用的属性，一个对象有多少个方法(method)，它就具有多少种行为。要访问一个对象的属性或者调用一个对象方法，使用点(.)操作符：

```
a = 3 + 4j          # 创建一个复数
r = a.real         # 取得一个复数的实部，访问该对象的一个属性

b = [1, 2, 3]     # 创建一个列表
b.append(7)       # 使用 append 方法为列表加入新的元素
```

1.2. 对象的标识与类型

内建函数id()返回一个对象的标识。该返回值是一个整数，目前的实现该整数通常就是对象在内存中的位置。is 运算符用来比较两个对象的标识。内建函数type()返回一个对象的类型：

Toggle line numbers

```
1 # 比较两个对象
2 def compare(a,b):
3     print 'The identity of a is ', id(a)
4     print 'The identity of b is ', id(b)
5     if a is b:
6         print 'a and b are the same object'
7     if a == b:
8         print 'a and b have the same value'
9     if type(a) is type(b):
10        print 'a and b have the same type'
```

对象的类型也是对象，这个对象具有唯一性。对同一类型的所有实例应用type()函数总是会返回同一个类型对象。因此，类型之间可以使用is运算符来进行比较。标准模块types内包含所有内建类型对象，我们可以通过它来完成类型检查工作：

Toggle line numbers

```
1 import types
2 if type(s) is types.ListType:
3     print 'Is a list'
4 else:
5     print 'Is not a list'
```

若要比两个自定义类实例对象的类型，最好是使用isinstance()函数。函数isinstance(s,C)用于测试s是否是C或C的子类的实例。详细内容请参阅第七章--类和面向对象的编程。

1.3. 引用计数与垃圾收集

一切对象都是引用计数的。当分配一个新的名字给一个对象，或者将其放入到一个容器比如列表、元组、或者字典中，该对象的引用计数就会增加1次。如：

Toggle line numbers

```
1 a = 3.4      # 创建一个对象 '3.4'，引用计数为 1
2 b = a        # 对象 '3.4' 引用计数增加 1，此时对象 '3.4' 的引用计数为 2
3 c = []
4 c.append(b)  # 对象 '3.4' 引用计数增加 1，此时对象 '3.4' 的引用计数为 3
```

例子中创建了一个包含值3.4的一个对象。变量 a 是一个指向该对象的名字。当用 a 来为 b 赋值时，b 成为同一个对象新的名称，此时对象的引用计数就会增1。同样地，当你把 b 放入一个列表中时，对象的引用计数再次增1。在例子中，自始至终只有一个值为 3.4 的整数对象，b 与 c[0] 都仅仅是该对象的引用。

del语句、脱离变量作用域或者变量被重新定义，都会使对象的引用计数减少。

Toggle line numbers

```
1 del a        # 直接删除一个引用，对象 3.4 引用减1
2 b = 7.8      # 某个引用被赋新值，对象 3.4 引用减1
3 c[0]=2.0     # 某个引用被赋新值，对象 3.4 引用减1
```

当一个对象的引用计数减少至零时，它就会在适当时机被垃圾回收车拉走。然而，特定情况(循环引用)会阻止垃圾回收车销毁不再使用的对象，看下面的例子：

Toggle line numbers

```
1 a = { }      # a 的引用为 1
2 b = { }      # b 的引用为 1
3 a['b'] = b   # b 的引用增 1，b的引用为2
4 b['a'] = a   # a 的引用增 1，a的引用为 2
5 del a        # a 的引用减 1，a的引用为 1
6 del b        # b 的引用减 1， b的引用为 1
```

在这个例子中,del语句减少了 a 和 b 的引用计数并删除了用于引用的变量名，可是由于两个对象各包含一个对方对象的引用，虽然最后两个对象都无法通过名字访问了，但引用计数并没有减少到零。因此这个对象不会被销毁，它会一直驻留在内存中，这就造成了内存泄漏。为解决这个问题，Python解释器会定期的运行一个搜索器，若发现一个对象已经无法被访问，不论该对象引用计数是否为 0，都销毁它。这个搜索器的算法可以通过 gc 模块的函数来进行调整和控制。具体内容参阅附录 A: Python 库。

1.4. 引用与副本

当运行语句 a = b 时，就创建了对对象 b 的一个新引用a。对于不可变对象(数字或字符串等)，改变对象的一个引用就会创建一个新对象。

Toggle line numbers

```
1 a=100        #创建一个新对象 100
2 b=a          #对象 100 增加了一个新的引用 b
3 print id(a),id(b) #打印 a 和 b 的标识，你会发现两个标识是相同的
4 b=20         #现在 b 不再是 a 的引用，变成新对象 20 的一个引
```

用了

```
5 print id(a),id(b)           #现在 a 和 b 的标识不再相同
```

对于可变对象(列表或字典等), 改变对象的一个引用就等于改变了该对象所有的引用, 见下例:

Toggle line numbers

```
1 b = [1,2,3,4]
2 a = b                       # a 是 b 的一个引用
3 a[2] = -100                 # 改变 a 中的一个元素
4 print b                     # b的值也随之改变为 '[1, 2, -100, 4]'
```

因为 a 和 b 指向相同的对象, 所以改变了 a 就等于改变了 b。为了避免这种情况, 你应该创建一个可变对象的副本, 然后对该副本进行操作。这样就不会影响到原始对象了。

有两种方法用来创建可变对象的副本: 浅复制(shallown copy)和深复制(deep copy)。浅复制创建一个新对象, 但它包含的子元素仍然是原来对象子元素的引用:

Toggle line numbers

```
1 b = [ 1, 2, [3,4] ]
2 a = b[:]                   # 创建b的一个 浅拷贝 a
3 a.append(100)              # a 对象添加一个新元素
4 print b                    # 打印 b 的值, 得到 '[1,2, [3,4]]', b 没有改变
5 a[0]=-100                  # 改变 a 的一个不可变子对象
6 print b                    # 打印 b 的值, 得到 '[1,2, [3,4]]', b 没有改变
7 a[2][0] = -100            # 改变 a 的一个可变子对象
8 print b                    # 打印 b 得到 '[1,2, [-100,4]]', b 被改变了
```

a 和 b 虽然是彼此独立的对象, 但他们包含的元素却是共享的。这样, 修改 a 中的一个可变元素也会影响 b 中的这个可变元素。

深复制创建一个新对象, 并递归复制所有子对象。python并没有内建的深复制函数, 不过在标准库中提供有一个copy模块, 该模块有一个deepcopy()函数可以漂亮的干这件事:

Toggle line numbers

```
1 import copy
2 b = [1, 2, [3, 4] ]
3 a = copy.deepcopy(b)
```

1.5. 内建类型

Python的解释器内建数个大类, 共二十几种数据类型, 表 3.1列出了全部内建类型。一些类别包含最常见的对象类型, 如数值、序列等, 其它类型则较少使用。后面几节将详细描述这些最常用的类型。

表 3.1 Python内建类型

分类	类型名称	描述
None	NoneType	null 对象

数值	IntType LongType	整数 任意精度
整数		
	FloatType ComplexType	浮点数 复数
序列	StringType UnicodeType	字符串 Unicode
字符串		
	ListType TupleType XRangeType	列表 元组 xrange
()函数返回的对象		
	BufferType	buffer
()函数返回的对象		
映射	DictType	字典
可调类型	BuiltinFunctionType BuiltinMethodType ClassType FunctionType	内建函数 内建方法 类 用户定义
函数		
	InstanceType MethodType	类实例 Bound
class method		
	UnboundMethodType	Unbound
class method		
模块	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 FrameType TracebackType	字节编译码 执行框架 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		

注意: `ClassType`和`InstanceType`在表中之所以出现两次, 是因为在特定环境下类及类实例都能被调用。

1.5.1. None类型

`None`表示空对象。如果一个函数没有显式的返回一个值, `None`就被返回。`None`经常被用做函数中可选参数的默认值。`None`对象没有任何属性。`None`的布尔值为假。

1.5.2. 数值类型

Python拥有四种数值类型: 整型, 长整型, 浮点类型, 以及复数类型。所有数值类型都是不可变类型。

整数类型用来表示从-2147483648 到 2147483647之间的任意整数(在某些电脑系统上这个范围可能会更大, 但绝不会比这个更小)。在系统内部, 一个整数以一个32位或者更多位的二进制补码形式储存。如果某次整数运算的结果超出了这个表示范围, 一般情况下Python会自动将运算结果由整型升级为长整型返回, 不过在有些情况下会引发一个溢出异常, 我们正在努力彻底消灭这个异常(OverflowError)。

长整数可以表示任意范围的整数(只要你的内存足够大就行)。

Python中只有双精度浮点数(64位), 它提供大约17个数字的精确度和-308到308的指数, 这与C中的double类型相同。Python不支持32位单精度的浮点数。如果你的程序很关心精确度和存储空间, 推荐你使用

Numerical Python (<http://numpy.sourceforge.net>)。

复数使用一对浮点数表示, 虚数 z 的实部和虚部分别用 z.real 和 z.imag 访问。

1.5.3. 序列类型

序列是由非负整数索引的对象的有序集合。它包括字符串、Unicode字符串、列表、元组、xrange对象以及缓冲区对象。字符串和缓冲区对象是字符序列, xrange对象是整数的序列, 列表和元组是任意Python对象的序列。字符串、Unicode字符串及元组是不可变序列, 列表是可变序列, 允许插入, 删除, 替换元素等操作。缓冲区对象将在本节后面详细描述。

Table 3.2列出所有序列对象均支持的操作及方法。序列 s 中的元素 i 使用索引运算符 s[i] 来访问, 通过切片运算符 s[i:j] 可以得到一个序列的子序列(这些运算符在第四章有详细介绍)。内建函数 len(s) 可以返回任意序列 s 的长度。你还能使用内建函数 min(s) 和 max(s) 来获得一个序列的最大值和最小值。不过, 这两个函数必须使用在元素可排序的序列中(典型的可排序序列是数值和字符串)。

Table 3.3介绍了可变序列(如列表)支持的其它操作

Table 3.2. 所有序列类型都支持的操作和方法

项目	描述
s [i]	返回序列s的元素i
s [i : j]	返回一个切片
len(s)	序列中元素的个数
min(s)	s 中的最小值
max(s)	s 中的最大值

Table 3.3. 可变序列适用的操作

项目	描述
s [i] = v	给某个元素赋新值
s [i : j] = t	用 序列 t 中的所有元素替换掉 s 序列中的索引从 i 至 j 的元素。
del s [i]	删除序列 s 中索引为 i 的元素。
del s [i : j]	删除序列 s 中的索引从 i 至 j 的元素

除此之外, 列表还支持Table 3.4中的方法。内建函数 list(s) 把可以把任意一个序列对象转换为一个列表。如果 s 本身是一个列表, 这个函数就创建一个 s 的浅拷贝。s.append(x) 方法可以在列表的末尾加入一个元素 x。s.index(x) 方法在列表中查找值 x 第一次出现时的索引, 若没有找到就引发一个ValueError异常。同样地, s.remove(x)方法删除第一次出现的值 x。s.extend(t)方法通过将链表 t 的所有元素添加到 s 的末尾来扩充列表s。s.sort()方法会将列表中的元素进行排序, 该方法接受自定义比较函数, 自定义比较函数必须有两个参数, 若

参数1小于参数2, 则返回-1, 若参数1等于参数2, 返回0, 否则就返回1。

`s.reverse()`方法反转列表中的所有元素。`sort()`和`reverse()`方法都是直接操作列表中元素并返回None。

Table 3.4. 列表的方法

方法	描述
<code>list(s)</code>	把序列s转换为一个列表
<code>s.append(x)</code>	把一个元素添加到列表的结尾, 相当于` s[len(s):] = [x]`
<code>s.extend(t)</code>	将链表 t 的所有元素添加到 s 的末尾来扩充列表 s, 相当于` s[len(s):] = t`
<code>s.count(x)</code>	返回值 x 在列表 s 中出现的次数
<code>s.index(x)</code>	返回列表s中第一个值为 x 的元素的索引值
<code>s.insert(i,x)</code>	在 s[i] 前插入一个元素 x
<code>s.pop([i])</code>	返回 s[i] 的值并将 s[i] 元素从列表中删除。如果 i 被省略, ` s.pop()` 就对最后一个元素进行操作。
<code>s.remove(x)</code>	删除列表中值为 x 的第一个元素
<code>s.reverse()</code>	翻转 s 中的全部元素
<code>s.sort([cmpfunc])</code>	对列表 s 中的元素进行排序, cmpfunc 是一个可选的比较函数

1.5.4. 字符串类型

Python拥有两种字符串类型。标准字符串是单字节字符序列, 允许包含二进制数据和嵌入的null字符。

Unicode 字符串是双字节字符序列, 一个字符使用两个字节来保存, 因此可以有最多65536种不同的unicode字符。尽管最新的Unicode标准支持最多100万个不同的字符, Python现在尚未支持这个最新的标准。

标准字符串和Unicode字符串都支持表 3.5中的方法。虽然这些方法都是用于操作一个字符串实例, 但所有的字符串方法都不会改变原始字符串。它们有的返回一个新得字符串, 如 `s.capitalize()`, `s.center()`, `s.expandtabs()`。有的返回True或者False, 如特征测试方法 `s.isalnum()` 和 `s.isupper()`, 值得一提的是, 这些方法当字符串长度为零时返回False。 `s.find()`、`s.rfind()`、`s.index()`、`s.rindex()` 方法被用来在 s 中寻找一个子串, 如果找到子串, 这些函数都返回s的整数索引值。当找不到子串时, `find()`方法返回-1, 而`index()`方法则引发一个 `ValueError` 异常。有很多数字符串方法接受两个可选的参数:

`start` 和 `end`, 用于指定 s 中开始位置和结束位置的索引。`s.translate()`方法根据一个字典来转换原始字符串, 该函数在附录A中的 `string` 模块中有详细描述。

`s.encode()` 方法用来将字符串转换为指定的字符集, 如'ascii'、'utf-8' 或 'utf-16'等。这个方法主要用于将 Unicode字符串转换为适合输入输出的字符编码, 关于此方法的详细介绍在第九章--输入和输出。要了解更多关于字符串方法的细节请参阅附录A中的 `string` 模块。

Table 3.5. 字符串方法

方法	描述
<code>s.capitalize()</code>	第一个字母变大写
<code>s.count(sub [,start [,end]])</code>	子串sub出现的次数
<code>s.encode([encoding [,errors]])</code>	改变字符串的编码
<code>s.startswith(prefix [,start [,end]])</code>	检查字符串的开头是否为prefix
<code>s.endswith(suffix [,start [,end]])</code>	检查字符串的结尾是否是suffix

<code>s.expandtabs([tabsize])</code>	将制表符转换为一定数量的空格
<code>s.find(sub [,start [,end]])</code>	返回子串 <code>sub</code> 首次出现的位置或者 <code>-1</code>
<code>s.rfind(sub [,start [,end]])</code>	返回子串 <code>sub</code> 末次出现的位置或者 <code>-1</code>
<code>s.index(sub [,start [,end]])</code>	返回子串 <code>sub</code> 首次出现的位置或者引起异常
<code>s.rindex(sub [,start [,end]])</code>	返回子串 <code>sub</code> 末次出现的位置或者引发异常
<code>s.isalnum()</code>	字符是否都为字母或数字
<code>s.isalpha()</code>	字符是否都为字母
<code>s.isdigit()</code>	字符是否都为数字
<code>s.islower()</code>	字符是否都为小写
<code>s.isspace()</code>	字符是否都为空白
<code>s.istitle()</code>	检查字符是否为标题格式(每个单词的第一个字母大写)
<code>s.isupper()</code>	字符是否都为大写
<code>s.join(t)</code>	用 <code>s</code> 连接 <code>t</code> 中的所有字符串
<code>s.center(width)</code>	在长度为 <code>width</code> 范围内将字符串置中
<code>s.ljust(width)</code>	在宽度为 <code>width</code> 内左对齐
<code>s.rjust(width)</code>	在宽度为 <code>width</code> 内右对齐
<code>s.lower()</code>	<code>s</code> 中所有字符小写
<code>s.upper()</code>	<code>s</code> 中所有字符大写
<code>s.replace(old , new [,maxreplace])</code>	将子串 <code>old</code> 替换为 <code>new</code>
<code>s.lstrip()</code>	删去字符串 <code>s</code> 开头的空白
<code>s.rstrip()</code>	删去字符串 <code>s</code> 末尾的空白
<code>s.strip()</code>	删去字符串 <code>s</code> 开头和末尾的空白
<code>s.split([sep [,maxsplit]])</code>	将字符串 <code>s</code> 分割成一个字符串列表, 其中 <code>sep</code> 为分隔符, <code>maxsplit</code> 是最大分割次数
<code>s.splitlines([keepends])</code>	将字符串按行分割为一个字符串列表, 若 <code>keepends</code> 为1, 则保留换行符' <code>\n</code> '
<code>s.swapcase()</code>	串内字符大写变小写, 小写变大写, 没有大小写的不变
<code>s.title()</code>	<code>s</code> 转换为标题格式(每个单词的第一个字母大写)
<code>s.translate(table [,deletechars])</code>	使用字符转换表转换一个字符串

1.5.5. xrangeType 类型

内建函数`range([i,]j[,stride])`建立一个整数列表, 列表内容为`k(i <= k < j)`。第一个参数`i`和第三个参数`stride`是可选的, 默认值分别为0和1。内建函数`xrange([i,]j[,stride])`与`range`有相似之处, 但`xrange`返回的是一个不可改变的`xrangeType`对象。这是一个迭代器, 也就是只有用到那个数时才临时通过计算提供值。当`j`值很大时, `xrange`能更有效地利用内存。`xrangeType`提供一个方法`s.tolist()`, 它可以将自己转换为一个列表对象返回。

1.5.6. 缓冲区类型

缓冲区对象将内存的一个连续区域模拟为一个单字节字符序列。Python没有直接创建缓冲区对象的语句, 你可以使用内建函数`buffer(obj[,offset[,size]])`来

创建此类对象。缓冲区对象与对象 `obj` 共享相同的内存，对于字符串切片操作或者其他字节数据操作来说，这样会有非常高的效率。另外，缓冲区对象还可以用来访问其他Python类型储存的原始数据，比如 `array` 模块中的数组、`Unicode` 字符串等。缓冲器对象是否可变，取决于 `obj` 对象。

1.5.7. 映射类型

映射类型用来表示通过关键字索引的任意对象的集合。和序列不同，映射类型是无序的。映射类型可以使用数字、字符串、或其他不可变对象来索引。映射类型是可变类型。

字典是唯一的内建的映射类型。可以使用任何不可变的对象作为字典的关键字(如字符串、数字、元组等)。列表、字典、及包含可变对象的元组不可以作为关键字。(字典类型需要关键字的值保持不变)

使用索引运算符 `m[k]` (`k` 为关键字) 可以访问映射对象 `m` 中索引为 `k` 的元素。如果映射对象中没有 `k` 这个关键字，则引发 `KeyError` 异常。`len(m)` 函数返回一个映射对象的元素个数。表 3.6 列出了映射对象可用的方法及操作。

Table 3.6. 映射对象的方法和操作

项目	描述
<code>len(m)</code>	返回 <code>m</code> 中的条目个数
<code>m[k]</code>	返回关键字 <code>k</code> 索引的元素
<code>m[k] = x</code>	设置关键字 <code>k</code> 索引的值为 <code>x</code>
<code>del m[k]</code>	删除一个元素
<code>m.clear()</code>	删除所有元素
<code>m.copy()</code>	返回 <code>m</code> 的一个浅拷贝
<code>m.has_key(k)</code>	若 <code>m</code> 中存在 <code>key k</code> 返回 <code>True</code> , 否则返回 <code>False</code>
<code>m.items()</code>	返回包含所有关键字和对应值 (<code>key ,value</code>) 的列表
<code>m.keys()</code>	返回由所有关键字组成的列表
<code>m.update(b)</code>	将字典 <code>b</code> 中的所有对象加入 <code>m</code>
<code>m.values()</code>	返回一个包含 <code>m</code> 中所有对应值的列表
<code>m.get(k[,v])</code>	返回 <code>m[k]</code> , 若 <code>m[k]</code> 不存在时, 返回 <code>v</code>
<code>m.setdefault(k[,v])</code>	返回 <code>m[k]</code> , 若 <code>m[k]</code> 不存在时, 返回 <code>v</code> 并设置 <code>m[k] = v</code>
<code>m.popitem()</code>	从 <code>m</code> 中随机删除一个元素, 并以元组的形式返回其关键字和值

1.5.8. 可调用类型

可调用类型表示所有允许以函数方式调用的对象。它包括用户定义函数、用户定义方法，内建函数、内建方法、`classic` 类及其实例、`new-style` 类及其实例。

1.5.8.1. 用户定义函数

用户定义函数是在 `module` 层使用 `def` 语句或者 `lambda` 操作符创建的可调用对象(在类层次定义的函数有专门的名字叫做方法)。函数是一类对象，用法和其它内建对象相似，允许将函数赋值给变量，也可以把函数放入列表、元组和字典中。看下面的例子:

Toggle line numbers

```
1 def foo(x,y):
2     print '%s + %s is %s' % (str(x), str(y), str(x+y))
```

```

3
4 # 指定为一个新的变量
5 bar = foo
6 bar(3,4)           # 调用上边定义好的foo
7
8 # 放入一个字典中
9 d = { }
10 d['callback'] = foo
11 d['callback'](3,4) # 调用foo

```

用户定义函数 **f** 有如下属性:

属性	描述
<code>f.__module__</code>	函数定义所在的模块名
<code>f.__doc__</code> 或 <code>f.func_doc</code>	文档字符串
<code>f.__name__</code> 或 <code>f.func_name</code>	函数名 (从2.4版开始该属性由只读变为可写)
<code>f.__dict__</code> 或 <code>f.func_dict</code>	支持任意函数属性的函数名字空间
<code>f.func_code</code>	(函数编译后产生的)字节码
<code>f.func_defaults</code>	包含所有默认参数的元组
<code>f.func_globals</code>	函数所在模块的全局名称空间的字典 (只读)
<code>f.func_closure</code>	None or a tuple of cells that contain bindings for the function's free variables. Read-only

用户定义函数对象也支持任意属性(设定值或取出值), 举个例子来说, 它可以用来夹带函数的元数据。用 `(.)`

操作符来存取这类属性。注意目前只有用户定义函数支持任意属性, 内建函数是不支持任意属性这个特性的。(也许将来我们会考虑让内建函数也支持这个特性, 也许...)

用户自定义函数任意属性示例

```

>>> def abc(x,y):
...     print x,y
...
>>> abc.a=100
>>> abc.a
100

```

1.5.8.2. 用户定义方法

用户定义方法是仅作用于对象实例的函数。通常方法在一个类定义中定义, 如 Listing 3.1:

Listing 3.1 定义一个方法

Toggle line numbers

```

1 # 按优先级排序的队列
2 class PriorityQueue:

```

```

3     def __init__(self):
4         self.items = []           # 包含(priority, item)的列表
5     def insert(self,priority,item):
6         for i in range(len(self.items)):
7             if self.items[i][0] > priority:
8                 self.items.insert(i,(priority,item))
9                 break
10        else:
11            self.items.append((priority,item))
12    def remove(self):
13        try:
14            return self.items.pop(0)[1]
15        except IndexError:
16            raise RuntimeError, 'Queue is empty'

```

非绑定方法(unbound method)是类中定义方法的引用，它没有被绑定到具体的类实例。

```
m = PriorityQueue.insert           # m是一个非绑定方法
```

要调用一个非绑定方法，需要将一个类实例做为该方法的第一个参数来调用：

Toggle line numbers

```

1 pq = PriorityQueue()           #pq 是一个类实例
2 m = PriorityQueue.insert       #m 是一个非绑定方法
3 m(pq,5,"Python")              #等于调用 pq.insert(5,"Python")

```

绑定方法(bound method)就是实例方法的别名。

Toggle line numbers

```

1 pq = PriorityQueue()           # 创建 PriorityQueue 实例
2 n = pq.insert                  # n 是一个绑定到 pq 实例的方法

```

绑定方法暗含了实例的引用，所以调用绑定方法时要象下面这样调用：

```
n(5,"Python")                    # 等于调用 pq.insert(5,"Python")
```

绑定和非绑定方法无非是略略封装了一下常规函数，下表列出了方法对象的属性：

属性	描述
m.im_self (见下面小注)	引用类实例对象，如果是非绑定方法，im_self通常为 None
m.im_func	引用类中定义的方法对象
m.im_class	引用定义该方法的类
m.__doc__	等于 m.im_func.__doc__
m.__name__	等于 m.im_func.__name__
m.__module__	等于 m.im_func.__module__

小注： 当一个用户定义方法引用的是一个类方法时，不论是否绑定到类实例，它的 im_self属性都等于其 im_class 属性。 --WeiZhong

注意： 每次访问一个类或类实例的属性时都会有一次从函数对象到方法对象的转换。这个转换要

占用CPU时间。在某些情况(对效率要求比较高的情况下)下, 一个很有效的优化手段就是, 用一个局部变量引用这个经常用到的类属性, 然后调用这个局部变量。还要注意的, 只有类中的用户定义方法才会发生这种转换, 其它可调用对象或不可调用对象不存在这种转换。另外需要注意的一点就是类实例的私有方法不需要这种转换。

1.5.8.3. 类和可调用的类实例

到现在为止, 我们集中讨论了函数和方法。类和类实例也是可调用对象。当一个类被调用时, 就生成该类的一个实例。如果该类定义了一个`__init__()`方法, 则这个方法就用来初始化新建的实例。上边例子中的`PriorityQueue`的创建就演示了这个行为。

如果一个类定义有一个特殊的方法`__call__()`, 那么该类的实例也可以被调用。假设 `x` 是一个可调用的类实例, `x(args)`调用就等同于调用`x.__call__(args)`。

1.5.8.4. 内建函数及内建方法

可调用类型还有内建函数和内建方法。内建函数和内建方法的代码一般位于用C或C++写的扩展模块中。下表列出了内建方法可用的属性:

属性	方法
<code>b.__doc__</code>	文档字符串
<code>b.__name__</code>	函数/方法名
<code>b.__self__</code>	方法所绑定的实例(未绑定时, 返回None)
<code>b.__members__</code>	方法的属性名(返回列表)

对于内建函数比如`len()`, 它的`__self__`是None。这表示这个函数并没有绑定给任何特殊对象。而对于内建函数 `x.append()` 来说(`x` 是一个列表), `__self__`返回 `x`。

1.5.9. 模块类型

模块是容器对象。`import`语句用来将其它模块中包含的对象导入当前模块。举例来说, 语句 `import foo` 中的 `foo` 就是一个模块对象。模块拥有自己的名字空间, 这是通过模块的一个字典属性来实现的。这个名字空间可以通过模块对象的`dict`属性来访问。当一个模块的属性被访问(使用点操作符)时, 比如访问 `m.x`, Python 会自动的去访问 `m.__dict__["x"]`。同样的, 赋值操作 `m[x]=y` 在内部被执行的其实是 `m.__dict__[x]=y`。模块对象拥有以下属性:

属性	描述
<code>m.__dict__</code>	保存模块名字空间的字典
<code>m.__doc__</code>	模块的文档字符串
<code>m.__name__</code>	模块名字
<code>m.__file__</code>	模块的文件名
<code>m.__path__</code>	当一个模块通过一个包被引用时, <code>__path__</code> 是包的名字

注1: 所有内建模块拥有没有`__file__` 属性的特权。

注2: 如果一个模块拥有 `__path__` 属性, `import` 语句就会认为它是一个包(package)。当从一个包中 `import` 一个子模块时, 将使用包的`__path__`属性而不是`sys.path`。

1.5.10. 类 类型

class语句用来创建类，第七章详细介绍了类。和模块类似，类也使用一个字典属性来维护自己的名称空间。访问类的属性时，比如 `c.x` 在执行行将被翻译成

`c.__dict__["x"]`。如果在类的 `__dict__` 里没有找到属性 `x`，那么就会到该类的父类中寻找。如果有多个父类，则搜索按照父类(`base class`)在类定义中顺序从左至右，深度优先。属性赋值如 `c.y = 5`，则总是更新 `c` 的 `__dict__` 属性，而不会更新某个父类的字典。

class对象定义的属性:

属性	描述
<code>c.__dict__</code>	类 <code>c</code> 的名字空间
<code>c.__doc__</code>	类 <code>c</code> 的文档字符串
<code>c.__name__</code>	类 <code>c</code> 的名字
<code>c.__module__</code>	类 <code>c</code> 的定义所在的模块
<code>c.__bases__</code>	类 <code>c</code> 的所有父类 (这是一个元组)

1.5.11. 类实例 类型

调用一个类就会生成该类的一个实例。每个实例也有独立的名字空间(也是字典，注意不要与类的名字空间混淆)。类实例有以下属性:

属性	描述
<code>x.__dict__</code>	实例 <code>x</code> 的名字空间
<code>x.__class__</code>	实例 <code>x</code> 所属的类

访问一个类实例 `x` 的属性时，比如 `x.a`，解释器会先查找 `x.__dict__["a"]`，若没有找到，则接着查询 `x.__class__.__dict__["a"]`，如果还没找到，则按照上面提到的搜索顺序继续查询该类的父类们的名字空间，如果还是没有找到，就要查看该类是否定义了 `__getattr__()` 方法，如果有这个方法就使用这个方法继续查找。如果经过以上种种手段仍然没有找到这个属性，就引发 `AttributeError` 异常。属性赋值如 `x.y = 5`，则总是更新实例 `x` 的 `__dict__` 属性，而不会更新其所属的类或其某个父类的 `__dict__` 字典。

1.5.12. 文件类型

一个文件对象就是一个打开的文件，调用内建函数 `open()` 成功则返回一个文件对象。更多关于文件类型的细节在第九章。

1.5.13. 内部类型

解释器内部使用的一系列对象，它们属于内部类型（用户通常不会遇到它们，不过必要时使用它们会解决一些棘手问题）。内部使用对象包括调试对象 (`traceback objects`)，代码对象 (`code objects`)，frame objects，切片对象 (`slice objects`) 及 省略对象 (`Ellipsis object`)。

1.5.13.1. 代码对象

调用内建函数`compile()`

返回一个代码对象。它表示原始字节编译码或称为字节码。代码对象和函数对象相似，但它不保存被编译代码的上下文信息（被编译代码所在名称空间及参数的默认值等）。代码对象是不可变对象，而函数对象是可变对象。一个代码对象 `c` 拥有如下只读属性：

属性	描述
<code>c.co_argcount</code>	参数的个数(不包括 * 或 ** 参数)
<code>c.co_code</code>	原始字节码字符串
<code>c.co_consts</code>	字节代码用到的常量
<code>c.co_filename</code>	对象 <code>c</code> 所在的文件
<code>c.co_firstlineno</code>	被编译源代码第一行行号
<code>c.co_flags</code>	解释器标志: 1=优化 2=newlocals 4=*arg 8>**arg
<code>c.co_lnotab</code>	源代码行号=>字节码偏移量 这是一个映射字典
<code>c.co_name</code>	该代码对象的名字
<code>c.co_names</code>	字节代码用到的局部变量名 这是一个元组
<code>c.co_nlocals</code>	字节代码用到的局部变量个数
<code>c.co_stacksize</code>	需要的虚拟机堆栈大小(包含内部变量)
<code>c.co_varnames</code>	一个元组, 包括全部的局部变量名和参数名

1.5.13.2. Frame 对象

Frame 对象表示执行 frame。通常在 `traceback` 对象中会遇到这个对象。它拥有以下只读属性：

属性	描述
<code>f.f_back</code>	下一个外部frame对象(对当前frame的调用者来说) 如果已到栈底的话 它的值就是 <code>None</code>
<code>f.f_code</code>	当前frame中正在执行的代码对象
<code>f.f_locals</code>	当前frame可见的局部变量的字典
<code>f.f_globals</code>	当前frame可见的全局变量的字典
<code>f.f_builtins</code>	当前frame可见的内建名字的字典
<code>f.f_restricted</code>	是否在受限模式下运行 0:不受限 1:受限
<code>f.f_lineno</code>	源代码当前行号
<code>f.f_lasti</code>	字节码当前指令索引

下边是frame对象的可写属性(通过调试器或其他工具可以改变下面属性的值)：

<code>f.f_trace</code>	当前frame的跟踪函数(供调试器使用) 或 <code>None</code>
<code>f.f_exc_type</code>	当前frame发生的异常类型 或 <code>None</code>
<code>f.f_exc_value</code>	当前frame发生的异常的值 或 <code>None</code>
<code>f.f_exc_traceback</code>	当前frame发生的 <code>traceback</code> 或 <code>None</code>

1.5.13.3. traceback 对象

`traceback` 对象保存异常的栈追踪信息。只要发生异常就会创建 `traceback` 对象。当一个异常被处理时，可以通过 `sys.exc_info()` 函数输出异常的堆栈追踪信息。

`traceback` 对象 `t` 有以下只读属性：

属性	描述
t.tb_next	栈追踪的下一级（对发生异常的 frame 来说）或 None
t.tb_frame	当前级正在执行的 frame 对象
t.tb_lineno	引发异常的源代码行号
t.tb_lasti	正在执行的指令索引

1.5.13.4. 切片对象

切片对象用于表示在扩展切片语法中的切片。如 `a [i :j :stride]`, `a [i :j , n :m]`, 或者 `a [..., i :j]`。切片对象也可以使用内建函数 `slice([i,] j [,stride])` 创建。切片对象有下列只读属性: 属性 描述 `s.start` 切片的下边界,省略时返回None `s.stop` 切片的上边界,省略时返回None `s.step` 切片的步进值,省略时返回None

1.5.13.5. 省略对象

省略对象用于表示在一个切片中出现了省略(...)。这个类型只有一个对象，通过内建名称 `Ellipsis` 来访问这个对象。它没有任何属性。它的布尔值为 `True`。

1.6. 特殊方法

所有内建的数据类型都拥有一些特殊方法。特殊方法的名字总是由两个下划线(`__`) 开头和结尾。在程序运行时解释器会根据你的代码隐式调用这些方法来完成你想要的功能。例如运行 `z = x + y` 这个代码，解释器内部执行的就是 `z= x.__add__(y)`。 `b=x[k]` 语句解释器就会执行 `b = x.__getitem__(k)`。每个数据类型的行为完全依赖于这些特殊方法的具体实现。

内建类型的特殊方法都是只读的，所以我们无法改变内建类型的行为。虽然如此，我们还是能够使用类定义新的类型，并让它具有象内建类型那样的行为。要做到这一点也不难，只要你能实现本章介绍的这些特殊方法就可以喽！

1.6.1. 对象创建、销毁及表示

表 3.7 中列出的方法用于初始化、销毁及表示对象。 `__init__()` 方法初始化一个对象，它在一个对象创建后立即执行。 `__del__()` 方法在对象即将被销毁时调用，也就是该对象完成它的使命不再需要时调用。需要注意的是语句 `del x` 只是减少对象 `x` 的引用计数，并不调用这个函数。

Table 3.7. 对象创建,删除,表示使用的特殊方法

方法	描述
<code>__init__(self[,args])</code>	初始化self
<code>__del__(self)</code>	删除self
<code>__repr__(self)</code>	创建self的规范字符串表示
<code>__str__(self)</code>	创建self的信息字符串表示
<code>__cmp__(self,other)</code>	比较两个对象,返回负数,零或者正数
<code>__hash__(self)</code>	计算self的32位哈希索引
<code>__nonzero__(self)</code>	真值测试,返回0或者1

`__repr__()` 和 `__str__()` 方法都返回一个字符串来表示 `self` 对象。通常情

况，`__repr__()`方法会返回的这样一个字符串：通过对该字符串取值(`eval`)操作将会重新得到这个对象。如果一个对象拥有`__repr__`方法，当对该对象使用`repr()`函数或后引号(```)操作时，就会调用这个函数做为返回值。例如：

Toggle line numbers

```
1 a = [2,3,4,5]           # 创建一个列表
2 s = repr(a)             # s = '[2, 3, 4, 5]'
3                          # 注：也可以使用 s = `a`
4 b = eval(s)             # 再转换为一个列表
```

如果`repr()`不能返回这样一个字符串，那它应该返回一个格式为`<...message...>`的字符串，例如：

Toggle line numbers

```
1 f = open("foo")
2 a = repr(f)             # a = "<open file 'foo', mode 'r' at
dc030>"
```

当调用`str()`函数或执行`print`语句时，python会自动调用被操作(或打印)对象的`__str__()`方法。与`__repr__()`相比，`__str__()`方法返回的字符串通常更简洁易读，内容一般是该对象的描述性信息。如果一个对象没有被定义该函数，Python就会调用`__repr__()`方法。

`__cmp__(self, other)`

方法用于与另一对象进行比较操作。如果 `self < other`，它返回一个负值；若 `self == other`，返回零；若 `self > other`，返回一个正数。如果一个对象没有定义该函数，对象就改用对象的标识进行比较。另外，一个对象可以给每个相关操作定义两个比较函数(正向反向)，这通常被称为rich comparisons。`__nonzero__()`方法用于对自身对象进行真值测试，应该返回0或1，如果这个方法没有被定义，Python将调用`__len__()`方法来取得该对象的真值。最后`__hash__()`方法计算出一个整数哈希值以便用于字典操作。(内建函数`hash()`也可以用来计算一个对象的哈希值)。相同对象的返回值是相等的。注意，可变对象不能定义这个方法，因为对象的变化会改变其哈希值，这会造成它不能被定位和查询。一个对象在未定义`cmp()`方法的情况下也不能定义`hash()`。

1.6.2. 属性访问

表 3.8列出了读取、写入、或者删除一个对象的属性的方法。

Table 3.8. 访问属性的方法

方法	描述
<code>__getattr__(self, name)</code>	返回属性 <code>self.name</code>
<code>__setattr__(self, name, value)</code>	设置属性 <code>self.name = value</code>
<code>__delattr__(self, name)</code>	删除属性 <code>self.name</code>

例如：

```
a = x.s           # 调用 __getattr__(x, "s")
x.s = b           # 调用 __setattr__(x, "s", b)
del x.s           # 调用 __delattr__(x, "s")
```

对于类实例，`__getattr__()`方法只在类例字典及相关类字典内搜索属性失败时才被调用。这个方法会返回属性值或者在失败时引发`AttributeError`异常。

1.6.3. 序列和映射的方法

表 3.9中介绍了序列和映射对象可以使用的方法。

Table 3.9. 序列和映射的方法

方法	描述
<code>__len__(self)</code>	返回self的长度
<code>len(someObject)</code>	会自动调用 <code>someObject</code> 的 <code>__len__()</code>
<code>__getitem__(self, key)</code>	返回 <code>self[key]</code>
<code>__setitem__(self, key, value)</code>	设置 <code>self[key] = value</code>
<code>__delitem__(self, key)</code>	删除 <code>self[key]</code>
<code>__getslice__(self, i, j)</code>	返回 <code>self[i:j]</code>
<code>__setslice__(self, i, j, s)</code>	设置 <code>self[i:j] = s</code>
<code>__delslice__(self, i, j)</code>	删除 <code>self[i:j]</code>
<code>__contains__(self, obj)</code>	返回 <code>obj</code> 是否在 <code>self</code> 中

例如:

```
a = [1,2,3,4,5,6]
len(a)                # __len__(a)
x = a[2]              # __getitem__(a,2)
a[1] = 7              # __setitem__(a,1,7)
del a[2]              # __delitem__(a,2)
x = a[1:5]            # __getslice__(a,1,5)
a[1:3] = [10,11,12]   # __setslice__(a,1,3,[10,11,12])
del a[1:4]            # __delslice__(a,1,4)
```

内建函数`len(x)`调用对象 `x` 的`__len__()`方法得到一个非负整数。如果一个对象没有定义`__nonzero__()`方法,就由 `__len__()`这个函数来决定其真值。`__getitem__(key)`方法用来访问个体元素。对序列类型, `key`只能是非负整数,对映射类型,关键字可以是任意Python不变对象。`__setitem__()`方法给一个元素设定新值。`__delitem__()`方法和`__delslice__()`方法在使用`del`语句时被自动调用。切片方法用来支持切片操作符 `s[i:j]`。`__getslice__(self,i,j)`方法返回一个`self`类型的切片,索引 `i` 和 `j` 必须是整数,索引的含义由`__getslice__()`方法的具体实现决定。如果省略 `i`, `i`就默认为 `0`, 如果省略 `j`, `j`就默认为 `sys.maxint`。`__setslice__()`方法给为一个切片设定新值。`__delslice__()`删除一个切片中的所有元素。`__contains__()`方法用来实现 `in` 操作符。

除了刚才描述过的方法之外,序列以及映射还实现了一些数学方法,包括`__add__()`, `__radd__()`, `__mul__()`, 和 `__rmul__()`,用于对象连接或复制等操作。下面会对这些方法略作介绍。

Python还支持扩展切片操作,这对于操作多维数据结构(如矩阵和数组)会很方便。你可以这样使用扩展切片:

Toggle line numbers

```
1 a = m[0:100:10]      # 步进切片 (stride=10)
2 b = m[1:10, 3:20]   # 多维切片
3 c = m[0:100:10, 50:75:5] # 多维步进切片
4 m[0:5, 5:10] = n    # 扩展切片分配
5 del m[:10, 15:]    # 扩展切片删除
```

扩展切片的一般格式是`i:j stride, srtdie`是可选的。和普通切片一样,你可以省略每个切片的开始或者结束的值。另外还有一个特殊的对象--省略对象。写做 `(...)`, 用

于扩展切片中表示任何数字:

Toggle line numbers

```
1 a = m[ ..., 10:20]           # 利用省略进行的扩展切片操作
2 m[10:20, ...] = n
```

当进行扩展切片操作时, `__getitem__()`, `__setitem__()`, 和 `__delitem__()`

方法分别用于实现访问、修改、删除操作。然而在扩展切片操作中, 传递给这些方法的参数不是一个整数, 而是一个包含切片对象的元组(有时还会包括一个省略对象)。例如:

```
a = m[0:10, 0:100:5, ...]
```

上面的语句会以下面形式调用 `__getitem__()`:

```
a = __getitem__(m, (slice(0,10,None), slice(0,100,5), Ellipsis))
```

注意: 在Python1.4版开始, 就一直有扩展切片的语法, 却没有任何一个内建类型支持扩展切片操作。

Python 2.3改变了这个现状。从Python 2.3开始, 内建类型终于支持扩展切片操作了, 这要感谢 Michael Hudson。

1.6.4. 数学操作

表3.10列出了与数学运算相关的特殊方法。数学运算从左至右进行, 执行表达式 `x + y` 时, 解释器会试着调用 `x.add(y)`。以 `r` 开头的特殊方法名支持以反转的操作数进行运算。它们在左运算对象未实现相应特殊方法时被调用, 例如 `x + y` 中的 `x` 若未提供 `__add__()` 方法, 解释器就会试着调用函数 `y.__radd__(x)`。

表 3.10. 数学操作的方法

Method	Result
<code>__add__(self ,other)</code>	<code>self + other</code>
<code>__sub__(self ,other)</code>	<code>self - other</code>
<code>__mul__(self ,other)</code>	<code>self * other</code>
<code>__div__(self ,other)</code>	<code>self / other</code>
<code>__mod__(self ,other)</code>	<code>self % other</code>
<code>__divmod__(self ,other)</code>	<code>divmod(self ,other)</code>
<code>__pow__(self ,other [,modulo])</code>	<code>self ** other , pow(self , other , modulo)</code>
<code>__lshift__(self ,other)</code>	<code>self << other</code>
<code>__rshift__(self ,other)</code>	<code>self >> other</code>
<code>__and__(self ,other)</code>	<code>self & other</code>
<code>__or__(self ,other)</code>	<code>self other</code>
<code>__xor__(self ,other)</code>	<code>self ^ other</code>
<code>__radd__(self ,other)</code>	<code>other + self</code>
<code>__rsub__(self ,other)</code>	<code>other - self</code>
<code>__rmul__(self ,other)</code>	<code>other * self</code>
<code>__rdiv__(self ,other)</code>	<code>other / self</code>
<code>__rmod__(self ,other)</code>	<code>other % self</code>

<code>__rdivmod__(self ,other)</code>	<code>divmod(other ,self)</code>
<code>__rpow__(self ,other)</code>	<code>other ** self</code>
<code>__rlshift__(self ,other)</code>	<code>other << self</code>
<code>__rrshift__(self ,other)</code>	<code>other >> self</code>
<code>__rand__(self ,other)</code>	<code>other & self</code>
<code>__ror__(self ,other)</code>	<code>other self</code>
<code>__rxor__(self ,other)</code>	<code>other ^ self</code>
<code>__iadd__(self ,other)</code>	<code>self += other</code>
<code>__isub__(self ,other)</code>	<code>self -= other</code>
<code>__imul__(self ,other)</code>	<code>self *= other</code>
<code>__idiv__(self ,other)</code>	<code>self /= other</code>
<code>__imod__(self ,other)</code>	<code>self %= other</code>
<code>__ipow__(self ,other)</code>	<code>self **= other</code>
<code>__iand__(self ,other)</code>	<code>self &= other</code>
<code>__ior__(self ,other)</code>	<code>self = other</code>
<code>__ixor__(self ,other)</code>	<code>self ^= other</code>
<code>__ilshift__(self ,other)</code>	<code>self <<= other</code>
<code>__irshift__(self ,other)</code>	<code>self >>= other</code>
<code>__neg__(self)</code>	<code>-self</code>
<code>__pos__(self)</code>	<code>+self</code>
<code>__abs__(self)</code>	<code>abs(self)</code>
<code>__invert__(self)</code>	<code>~self</code>
<code>__int__(self)</code>	<code>int(self)</code>
<code>__long__(self)</code>	<code>long(self)</code>
<code>__float__(self)</code>	<code>float(self)</code>
<code>__complex__(self)</code>	<code>complex(self)</code>
<code>__oct__(self)</code>	<code>oct(self)</code>
<code>__hex__(self)</code>	<code>hex(self)</code>
<code>__coerce__(self ,other)</code>	<code>Type coercion</code>

`__iadd__()`, `__isub__()`方法用于实现原地运算(in-place arithmetic), 如 `a+=b` 和 `a-=b`(称为增量赋值)。原地运算与标准运算的差别在于原地运算的实现会尽可能的进行性能优化。举例来说, 如果 `self` 参数是非共享的, 就可以原地修改 `self` 的值而不必为计算结果重新创建一个对象。

`int()`, `long()`, `float()`和`complex()` 返回一个相应类型的新对象, `oct()` 和 `hex()`方法分别返回相应对象的八进制和十六进制的字符串表示。

`x.coerce(self,y)` 用于实现混合模式数学计算。这个方法在需要时对参数 `self` (也就是 `x`) 和 `y` 进行适当的类型转换, 以满足运算的需要。如果转换成功, 它返回一个元组, 其元素为转换后的 `x` 和 `y`, 若无法转换则返回 `None`。在计算`x op y`时(`op`是运算符), 使用以下规则:

1. 若 `x` 有`__coerce__()`方法, 使用`x.__coerce__(y)`返回的值替换 `x` 和 `y`, 若返回`None`, 转到第 3 步。
2. 若 `x` 有`__op __()`方法, 返回 `x.__op __()`。否则恢复 `x` 和 `y` 的原始值, 然后进行下一步。
3. 若 `y` 有`__coerce__()`方法, 使用`y.__coerce__(x)`返回的值替换 `x` 和 `y`。若返回`None`, 则引发异常。
4. 若 `y` 有`__rop __()`方法, 返回`y.__op __()`, 否则引发异常。

虽然字符串定义了一些运算操作, 但 ASCII字符串和Unicode字符串之间的运算并不

使用 `coerce()` 方法。

在内建类型中，解释器只支持少数几种类型进行混合模式运算。常见的有如下几种：

- 如果 `x` 是一个字符串，`x % y` 调用字符串格式化操作，与 `y` 的类型无关
- 如果 `x` 是一个序列，`x + y` 调用序列连结
- 如果 `x` 和 `y` 中一个是序列，另一个是整数，`x * y` 调用序列重复

1.6.5. 比较操作

表 3.11 列出了实现分别各种比较操作 (`<`, `>`, `<=`, `>=`, `==`, `!=`) 的对象特殊方法，这也就是 `rich comparisons`。这个概念在 Python 2.1 中被第一次引入。这些方法都使用两个参数，根据操作数返回适当类型对象 (布尔型, 列表或其他 Python 内建类型)。举例来说，两个矩阵对象可以使用这些方法进行元素智能比较，并返回一个结果矩阵。若比较操作无法进行，则引发异常。

表 3.11. 比较方法

方法	操作
<code>__lt__(self ,other)</code>	<code>self < other</code>
<code>__le__(self ,other)</code>	<code>self <= other</code>
<code>__gt__(self ,other)</code>	<code>self > other</code>
<code>__ge__(self ,other)</code>	<code>self >= other</code>
<code>__eq__(self ,other)</code>	<code>self == other</code>
<code>__ne__(self ,other)</code>	<code>self != other</code>

1.6.6. 可调用对象

最后，一个对象只要提供 `__call__(self[,args])` 特殊方法，就可以象一个函数一样被调用。举例来说，如果对象 `x` 提供这个方法，它就可以这样调用：

`x(arg1 , arg2 , ...)`。解释器内部执行的则是

`x .__call__(self , arg1 , arg2 , ...)`。

1.7. 性能及内存占用

所有的 Python 对象至少拥有一个整型引用记数、一个类型定义描述符及真实数据的表示这三部分。对于在 32 位计算机上运行的 C 语言实现的 Python 2.0，表 3.12 列出了常用内建对象占用内存的估计大小，对于解释器的其它实现或者不同的机器配置，内存占用的准确值可能会有不同。你可能从来不考虑内存占用问题，不过当 Python 在要求高性能及内存紧张的环境下运行时，就必须考虑这个问题。下边这个表可以有效地帮助程序员更好地规划内存的使用：

表 3.12. 内建数据类型使用的内存大小

类型	大小
Integer	12 bytes
Long integer	12 bytes + (nbits/16 + 1)*2 bytes
Floats	16 bytes

Complex	24 bytes
List	16 bytes + 4 bytes(每个元素)
Tuple	16 bytes + 4 bytes(每个条目)
String	20 bytes + 1 byte(每个字符)
Unicode string	24 bytes + 2 bytes(每个字符)
Dictionary	24 bytes + 12*2n bytes, n = log2(nitems)+1
Class instance	16 bytes 加一个字典对象
Xrange object	24 bytes

由于字符串类型太常用了，所以解释器会特别优化它们。可以使用内建函数intern(s)来暂存一个频繁使用的字符串s。这个函数首先在内部哈希表中寻找字符串s的哈希值，如果找到，就创建一个到该字符串的引用，如果找不到，就创建该字符串对象并将其哈希值加入内部哈希表。只要不退出解释器，被暂存的字符串就会一直存在。如果你关心内存占用，那你就一定不要暂存极少使用的字符串。为了使字典查询更有效率，字符串会缓存它们最后的哈希值。

一个字典其实就是一个开放索引的哈希表。The number of entries allocated to a dictionary is equal to twice the smallest power of 2 that's greater than the number of objects stored in the dictionary. When a dictionary expands, its size doubles. On average, about half of the entries allocated to a dictionary are unused.

一个Python程序的执行首先是一系列的函数调用(包括前面讲到的特殊方法)，之后再选择最高效的算法。搞懂Python的对象模型并尽量减少特殊方法的调用次数，可以有效提高你的程序的性能。特别是改良类及模块的名字查询方式效果卓著。看下面的代码：

```
Toggle line numbers
1 import math
2 d = 0.0
3 for i in xrange(1000000):
4     d = d + math.sqrt(i)
```

在上面的代码中，每次循环调用都要进行两次名称查询。第一次在全局名称空间中定位math模块，第二次是搜寻一个名称是sqrt的函数对象。我们将代码改成下面这样：

```
Toggle line numbers
1 from math import sqrt
2 d = 0.0
3 for i in xrange(1000000):
4     d = d + sqrt(i)
```

这个代码每次循环只需要进行一次名称查询。就这样一个简单的调整，在作者的200 MHz PC上运行时,这个简单的变化会使代码运行速度提高一倍多。

注：200 MHz的机器我没有，但在我的2000 MHz机器上效果没有这么明显，不过仍然有10%以上的提高 --Feather
 那时作者用的是 Python 2.0，现在你用的是 2.4。Python一直在不断进步嘛！
 --Weizhong

在Python程序设计中，应该仅在必要时使用临时变量，尽可能的避免非必要的序列或字典查询。下面我们看看 Listing 3.2中的这两个类：

Listing 3.2 计算一个平面多边形的周长

Toggle line numbers

```
1 class Point:
2     def __init__(self,x,y,z):
3         self.x = x
4         self.y = y
5 #低效率的示例
6 class Poly:
7     def __init__(self):
8         self.pts = []
9     def addpoint(self,pt):
10        self.pts.append(pt)
11    def perimeter(self):                #计算周长
12        d = 0.0
13        self.pts.append(self.pts[0])   # 暂时封闭这个多边形
14        for i in xrange(len(self.pts)-1):
15            d2 = (self.pts[i+1].x - self.pts[i].x)**2 +
16            (self.pts[i+1].y - self.pts[i].y)**2
17            d = d + math.sqrt(d2)
18        self.pts.pop()                # 恢复原来的列表
19        return d
```

Poly类中的 `perimeter()` 方法，每次访问 `self.pts[i]`都会产生两次查询--一次查询名字空间字典,另一次查询 `pts` 序列。

下面我们改写一下代码，请看 Listing 3.3:

Listing 3.3 Listing 3.2的改良版本

Toggle line numbers

```
1 class Poly:
2     ...
3     def perimeter(self):
4         d = 0.0
5         pts = self.pts                #提高效率的关键代码
6         pts.append(pts[0])
7         for i in xrange(len(pts)-1):
8             p1 = pts[i+1]
9             p2 = pts[i]
10            d += sqrt((p1.x - p2.x)**2 + (p1.y - p2.y)**2)
11        pts.pop()
12        return d
```

这个代码的关键之处在于用一个局部变量引用了一个类属性，尽管这样的修正对效率提高的并不是很多(15-20%)，了解这些并在你的常规代码中留意这些细节就能够帮助你写出高效的代码。当然，如果对性能要求极高，你也可以 C 语言编写 Python 扩展。

1. 第四章 运算符与表达式

本章的主题是 Python 语言的内建运算符及表达式求值的优先级。

1.1. 数值操作

所有数值类型都提供下列运算:

运算	描述
$x + y$	加
$x - y$	减
$x * y$	乘
x / y	常规除
$x // y$	地板除
$x ** y$	乘方 (x^y)
$x \% y$	取模 ($x \bmod y$)
$-x$	改变操作数的符号位
$+x$	什么也不做
$\sim x$	$\sim x = -(x+1)$

关于常规除 $/$ 与地板除 $//$: 地板除在任何时候都会将小数部分舍为0, 而常规除的行为依操作数的类型而有不同。
常规除 $/$: 整数除以整数时 $/$ 与 $//$ 除完全相同, 商都会被舍去小数部分而返回一个整数。如 $7 / 4$ 的结果是 1 , 而不是 1.75 ;
如果有一个操作数为浮点数, 情形就不同了:
对于 $/$, 会返回一个双精度浮点数
对于 $//$, 也会返回一个双精度浮点数, 只不过小数部分被舍弃

取模操作返回 x/y 的余数, 如 $7 \% 4$ 是 3 。对于浮点数, 取模操作返回的是 $x - \text{int}(x / y) * y$ 。对于复数, 取模操作返回 $x - \text{int}((x / y).real) * y$ 。

下列列出的位运算符只能用于整数或长整数:

操作	描述
$x \ll y$	左移
$x \gg y$	右移
$x \& y$	按位与
$x y$	按位或
$x \wedge y$	按位异或 (exclusive or)
$\sim x$	按位翻转

这些都是相当原始的运算, 操作的是操作数的每一个位。所有的操作数都假定是以二进制补码形式表示。对于长整数, 按位运算符假定符号位可以被无限地向左扩展。

除了这些以外, 下边这些内建函数支持所有的数值类型:

目录

- 1. 第四章 运算符与表达式
 - 1. 数值操作
 - 2. 序列运算
 - 3. 字典的操作
 - 4. 增量赋值语句
 - 5. 属性(.)操作符
 - 6. 类型转换
 - 7. Unicode字符串
 - 8. 布尔表达式
 - 9. 对象的比较与身份
 - 10. 运算优先级

函数	描述
<code>abs(x)</code>	绝对值
<code>divmod(x, y)</code>	返回 $(\text{int}(x / y), x \% y)$
<code>pow(x, y [, modulo])</code>	返回 $(x ** y) \% \text{modulo}$
<code>round(x, [n])</code>	四舍五入, n为小数点位数

`abs()`函数返回一个数的绝对值。`divmod()`函数返回一个包含商和余数的元组。`pow()`函数可以用于代替`**`运算,但它还支持三重取模运算(经常用于密码运算)。`round`函数总是返回一个浮点数。**Python**的四舍五入规则不是银行家四舍五入规则,这一点请大家注意。

下列比较操作有标准的数学解释,返回一个布尔值`True`,或者`False`:

运算符	描述
<code>x < y</code>	小于
<code>x > y</code>	大于
<code>x == y</code>	等于
<code>x != y</code>	不等于(与 <code><></code> 相同)
<code>x >= y</code>	大于等于
<code>x <= y</code>	小于等于

Python的比较运算可以连结在一起,如`w < x < y < z`。这个表达式等价于`w < x and x < y and y < z`。

`x < y > z`这个表达式也是合法的,(注意,这个表达式中`x`和`z`并没有比较操作)。不建议这样的写法,因为这会造成代码的阅读困难。

只可以对复数进行等于(`==`)及不等于(`!=`)比较,任何对复数进行其他比较的操作都会引发`TypeError`异常。

数值操作要求操作数必须是同一类型,若**Python**发现操作数类型不一致,就会自动进行类型的强制转换,转换规则如下:

1. 如果操作数中有一个是复数,另一个也将被转换为复数
2. 如果操作数中有一个是浮点数,另一个将被转换为浮点数
3. 如果操作数中有一个是长整数数,另一个将被转换为长整数数
4. 如果以上都不符合,则这两个数字必然都是整数,不需进行强制转换。

1.2. 序列运算

序列支持以下操作:

操作	描述
<code>s + r</code>	序列连接
<code>s * n, n * s</code>	<code>s</code> 的 <code>n</code> 次拷贝, <code>n</code> 为整数
<code>s % d</code>	字符串格式化(仅字符串)
<code>s[i]</code>	索引
<code>s[i : j]</code>	切片
<code>x in s, x not in s</code>	从属关系
<code>for x in s :</code>	迭代
<code>len(s)</code>	长度
<code>min(s)</code>	最小元素

+ 运算符将两个相同类型的序列连结成一个。s * n 运算符给出一个序列的 n 次浅拷贝。下边的例子可以帮助你理解这点:

Toggle line numbers

```

1 a = [3,4,5]           # 一个列表
2 b = [a]              # 包含a的列表
3 c = 4*b              # b的四次拷贝
4
5 # 修改 a
6 a[0] == -7
7
8 # 打印出 c
9 print c

```

程序将会输出:

```
[[ -7, 4, 5], [-7, 4, 5], [-7, 4, 5], [-7, 4, 5]]
```

这种情况下,列表b中放置了到列表a的引用,当b被重复的时候,仅创建了4个额外的引用。所以,当a被修改的时候,这个变化也影响到所有a的引用。这种情况通常是大多程序员不愿意看到的。你可以通过复制a中的所有元素来解决这种问题。如:

```

a = [3, 4, 5 ]
c = [a[:] for j in range((4))] # [:]代表a的副本而不是到a的引用

```

注:a[:]这种方式也仅仅是创建列表a所有元素的浅拷贝,如果a中有元素为可变元素,仍然可能会有潜在问题。 --Weizhong

标准库中的copy模块也可以用于一个对象的浅复制,另外它还支持深复制。

索引操作符 s[n] 返回序列中的第 n 个对象(s[0]是第一个),如果 n 是负数,在求值之前,就先执行 n+=len(s)。如果尝试读取一个不存在的元素则会引发IndexError异常。

切片操作符s[i:j]返回一个子序列。i 和 j 必须是整数或长整数。如果被省略,那么它们的默认值分别为序列的开始或结束。切片操作同样允许负数索引。你只要记住这个公式: s[n]=s[n-len(s)] (n为正数) 或者 s[n]=s[len(s)+n] (n为负数)就行了。

Toggle line numbers

```

1 s=[1,2,3,4]           # s 上界为 0 下界为 4
2 print s[-100:100]     #返回 [1,2,3,4] -100超出了上界,100超出了下
界: 等价于 s[0:4]
3 print s[-100:-200]    #返回 [] -100,-200均超出了上界,自动取上界: 等
价于s[0:0]
4 print s[100:200]      #返回 [] 100,200均超出了下界,自动取下界值: 等价
于s[4:4]

```

x in s 运算符检验对象 x 是否是 s 的子对象,并返回True或False。not in 运算符刚好与 in 相反。for x in s 操作顺序迭代序列中的全部元素,这将在第五章--控制流中详细介绍。len(s)返回一个序列中的元素个数。min(s)和max(s)返回一个序列的最小值和最大值,这两个函数只有序列中的元素可排序时返回值才有意义。(如果对一个文件对象的列表取最大值或最小值,就毫无意义)

字符串和元组是不可变对象,不能在创建之后对原始对象修改。列表则可以进行以

下操作:

操作	描述
<code>s[i] = x</code>	为 <code>s[i]</code> 重新赋值
<code>s[i:j] = r</code>	将列表片段重新赋值
<code>del s[i]</code>	删除列表中一个元素
<code>del s[i:j]</code>	删除列表中一个片段

`s[i] = x`操作将列表索引为 `i` 的元素重新赋值为对象 `x`，并增加 `x` 的引用记数。如果 `i` 是负数，在求值之前，就先执行 `i += len(s)`，计算结果必须是一个小于 `len(s)` 的非负整数。尝试给一个不存在的索引赋值会引发 `IndexError` 异常。切片分配操作符 `s[i:j] = r` 将列表片段 `s[i:j]` 替换为序列 `r`。如:

Toggle line numbers

```
1 a = [1,2,3,4,5]
2 a[1] = 6          # a = [1,6,3,4,5]
3 a[2:4] = [10,11] # a = [1,6,10,11,5]
4 a[3:4] = [-1,-2,-3] # a = [1,6,10,-1,-2,-3,5]
5 a[2:] = [0]      # a = [1,6,0]
```

`del s[i]`语句从列表 `s` 中删除元素 `i`，并将它的引用记数减1。`del s[i:j]`语句删除一个切片。

序列可以使用 `<`, `>`, `<=`, `>=`, `==` 和 `!=` 来进行比较。当比较两个序列的时候,首先比较序列的第一个元素。如果它们不同,就马上得出结论.如果它们相同,就继续比较第二个元素，直到找到两个不同的元素或者两个序列都没有多余元素为止。字符串通过比较每个字符的内部编码决定大小(如ASCII或Unicode)。

字符串取模运算

`s % d` 返回一个格式化后的字符串。需要一个格式字符串 `s` 作为左操作数，一个独立对象或一个元组或一个映射对象 `d` 作为右操作数。格式字符串 `s` 可以是ASCII字符串，也可以是一个Unicode字符串。这个运算符和 C 语言中的 `sprintf()` 函数类似。格式字符串包含两种对象类型:普通字符(不改变它的值)和转换符(`%` + 转换字符)--在输出结果中，转换符将格式化 `d` 中的相应元素，然后用格式化后的结果填充自身。如果 `s` 内只有一个转换符，则允许一个 `d` 是一个独立的非tuple对象。否则 `d` 就必须是一个tuple或映射对象。如果 `d` 是一个tuple,则转换表示符的个数必须和 `d` 的长度相等; 如果 `d` 是一个映射,每个转换表示符的 `%` 字符之后必须有一个小括号括起来的映射对象中的key值.表Table 4.1详细列出了转换符的使用。

表 4.1. 字符串格式转换

字符	输出格式
<code>d, i</code>	十进制整数或长整数
<code>u</code>	无符号十进制整数或长整数
<code>o</code>	八进制整数或长整数
<code>x</code>	十六进制整数或长整数
<code>X</code>	十六进制整数或长整数(大写字母)
<code>f</code>	浮点数如 <code>[-]m.ddddd</code>
<code>e</code>	浮点数如 <code>[-]m .ddddde ±xx .</code>
<code>E</code>	浮点数如 <code>[-]m .ddddde ±xx .</code>
<code>g, G</code>	指数小于-4或者更高精确度使用 <code>%e</code> 或 <code>%E</code> ; 否则,使用 <code>%f</code>
<code>s</code>	字符串或其他对象,使用 <code>str()</code> 来产生字符串
<code>r</code>	与 <code>repr()</code> 返回的字符串相同
<code>c</code>	单个字符

在 % 和转换字符串之间,允许出现以下修饰符,并且只能按以下顺序:

1. 映射对象的 key,如果被格式化对象是一个映射对象却没有这个成分,会引发KeyError异常.
 2. 下面所列的一个或多个:
 - 左对齐标志
 - +,数值指示必须包含
 - 0,指示一个零填充
 3. 指示最小栏宽的数字.转换值会被打印在指定了最小宽度的栏中并且填充在(或者右边).
 4. 一个小数点用来分割浮点数
 5. A number specifying the maximum number of characters to be printed from a string, the number of digits following the decimal point in a floating-point number, or the minimum number of digits for an integer.
- 另外,形标(*)字符用于在任意宽度的栏中代替数字. If present, the width will be read from the next item in the tuple.下边的代码给出了几个例子:

```
Toggle line numbers
1 a = 42
2 b = 13.142783
3 c = "hello"
4 d = {'x':13, 'y':1.54321, 'z':'world'}
5 e = 5628398123741234L
6
7 print 'a is %d' % a           # "a is 42"
8 print '%10d %f' % (a,b)      # " 42 13.142783"
9 print '%+010d %E' % (a,b)    # "+000000042 1.314278E+01"
10 print '%(x)-10d %(y)0.3g' % d # "13          1.54"
11 print '%0.4s %s' % (c, d['z']) # "hell world"
12 print '%*.*f' % (5,3,b)      # "13.143"
13 print 'e = %d' % e           # "e = 5628398123741234"
```

1.3. 字典的操作

字典用来在名字与对象之间建立映射。对一个字典可进行以下操作:

操作	描述
x = d[k]	通过 key 访问字典元素
d [k] = x	通过 key 对字典元素进行赋值
del d[k]	通过 key 删除某个字典元素
len(d)	字典的元素个数

key 可以是任意不可变对象,如字符串,数字,和元组。另外,字典的关键字也可以是用逗号分隔的多个值。例如:

```
Toggle line numbers
1 d = {}
2 d[1,2,3] = "foo"
3 d[1,0,3] = "bar"
```

在这种情况下,关键字的值其实是一个元组,下面的代码和上面的代码作用是一样的:

```
d[(1,2,3)] = "foo"  
d[(1,0,3)] = "bar"
```

1.4. 增量赋值语句

Python提供以下的增量赋值操作:

操作	等价表达式
<code>x += y</code>	<code>x = x + y</code>
<code>x -= y</code>	<code>x = x - y</code>
<code>x *= y</code>	<code>x = x * y</code>
<code>x /= y</code>	<code>x = x / y</code>
<code>x **= y</code>	<code>x = x ** y</code>
<code>x %= y</code>	<code>x = x % y</code>
<code>x &= y</code>	<code>x = x & y</code>
<code>x = y</code>	<code>x = x y</code>
<code>x ^= y</code>	<code>x = x ^ y</code>
<code>x >>= y</code>	<code>x = x >> y</code>
<code>x <<= y</code>	<code>x = x << y</code>

应用举例:

```
Toggle line numbers  
1 a = 3  
2 a += 1 # a = 4  
3 b = [1,2]  
4 b[1] += 10 # b = [1, 12]  
5 c = "%s %s"  
6 c %= ("Douglas", "Adams") # c = "Douglas Adams"
```

需要指出的是,增量赋值语句并不对对象进行原地修改,因此也不会改变对象的性质。`x += y`语句创建了一个值为`x + y`的新对象,并将这个对象赋给`x`。用户自定义类可通过定义特殊方法重载增量赋值操作符。(参见第三章,类型和对象)

1.5. 属性(.)操作符

点(.)操作符用来访问一个对象的属性,例如:

```
Toggle line numbers  
1 foo.x = 3  
2 print foo.y  
3 a = foo.bar(3,4,5)  
4 del foo.x
```

点操作符并不仅仅可以用于单个表达式,例如`foo.y.a.b`。它还可以用于函数的中间结果,例如`a = foo.bar(3,4,5).spam`。属性可以使用`del`语句来删除,例如`del foo.x`。

1.6. 类型转换

经常对内建类型进行类型转换的需要。下列内建函数提供了显式的类型转换操作:

函数	描述
<code>int(x [,base])</code>	将x转换为一个整数
<code>long(x [,base])</code>	将x转换为一个长整数
<code>float(x)</code>	将x转换到一个浮点数
<code>complex(real [,imag])</code>	创建一个复数
<code>str(x)</code>	将对象 x 转换为字符串
<code>repr(x)</code>	将对象 x 转换为表达式字符串
<code>eval(str)</code>	用来计算在字符串中的有效Python表达式,并返回一个对象
<code>tuple(s)</code>	将序列 s 转换为一个元组
<code>list(s)</code>	将序列 s 转换为一个列表
<code>chr(x)</code>	将一个整数转换为一个字符
<code>unichr(x)</code>	将一个整数转换为Unicode字符
<code>ord(x)</code>	将一个字符转换为它的整数值
<code>hex(x)</code>	将一个整数转换为一个十六进制字符串
<code>oct(x)</code>	将一个整数转换为一个八进制字符串

`repr(x)`函数也可写为 `x`

.注意`str()`函数和`repr()`函数返回的结果经常是不同的. `repr()`函数取得对象的表达式字符串表示,通常可以使用`eval()`函数来重新得到这个对象.而`str()`产生一个对象的简洁格式表示(用于`print`语句). `ord()`函数返回字符在`ascii`或`Unicode`字符编码中的整数顺序值. `chr()`和`unichr()`函数将一个整数分别转换为`ascii`或`Unicode`字符.

将一个字符串转换为数字或其他对象,使用`int()`, `long()`,和 `float()`函数. `eval()`函数也可以将一个包含有效表达式的字符转换为一个对象,例如:

Toggle line numbers

```
1 a = int("34")           # a = 34
2 b = long("0xfe76214", 16) # b = 266822164L (0xfe76214L)
3 b = float("3.1415926")  # b = 3.1415926
4 c = eval("3, 5, 6")     # c = (3,5,6)
```

1.7. Unicode字符串

在同一个程序中使用标准字符串和`Unicode`字符串会有一点点复杂.这是因为对字符串有太多种操作,包括字符串连结,比较,字典关键字查询,以及在函数中用做参数.

内建函数`unicode(s [, encoding [,errors]])`可以把一个标准字符串转换为一个`Unicode`字符串.字符串方法`u.encode([encoding [, errors]])`可以把一个`Unicode`字符串转换为一个标准字符串.这些转换操作需要特殊编码规则来指定16位`Unicode`字符串与标准8位字符来相互映射.编码参数是一个由如下值组成的特定字符串:

值	描述
<code>'ascii'</code>	7-bit ASCII
<code>'latin-1'</code> or <code>'iso-8859-1'</code>	ISO 8859-1 Latin-1
<code>'utf-8'</code>	8-位可变长度编码
<code>'utf-16'</code>	16-位可变长度编码(可能是 <code>little endian</code> 或
<code>big endian</code>)	

'utf-16-le'	UTF-16, little-endian 编码
'utf-16-be'	UTF-16, big-endian 编码
'unicode-escape'	与Unicode文字 u"string" 相同
'raw-unicode-escape'	与原始 Unicode文字 ur"string"相同

默认编码在site模块中设置,可以通过sys.getdefaultencoding()来读取.在多数情况下,默认编码是'ascii',即就是ASCII字符,它的值在区间[0x00,0x7f]内,直接映射到Unicode字符的[U+0000, U+007F].其他关于编码的内容在第九章--输入与输出.

当转换字符串时,如果有一个字符不能被转换,那么就会引起一个UnicodeError异常.比如,如果编码规则为'ascii', Unicode字符U+1F28 就不能被转换,因为它的值太大.同样地,字符串 "\xfc" 也不能被转换到Unicode,因为它也超出了ASCII字符范围. errors参数决定如何处理编码错误.它是一个包含下列值的字符串:

值	描述
'strict'	编码错误时引起一个UnicodeError异常
'ignore'	忽略不可转换字符
'replace'	将不可转换字符用U+FFFD替代(Unicode中的U+FFFD 是标准字符串中的'?')

默认错误处理是 'strict'.

当标准字符串和Unicode字符串在表达式中混用时,标准字符串将被自动转换为Unicode字符串.例如:

```
Toggle line numbers
1 s = "hello"
2 t = u"world"
3 w = s + t           # w = unicode(s) + t
```

当一个字符串方法(第三章中讲到)中使用到Unicode字符串时,结果也将总是Unicode字符串,例如:

```
Toggle line numbers
1 a = "Hello World"
2 b = a.replace("World", u"Bob") # b = u"Hello Bob"
```

此外,即使使用replace()方法进行零替换(替换结果仍是原始字符串)时,结果仍然会是Unicode字符串.

如果一个Unicode字符串使用 % 操作符做格式字符串,所有参数在一开始都将强制转换为Unicode字符串,然后再根据格式规则连结在一起.如果一个Unicode对象被用做 % 操作符的一个参数,整个结果也将是Unicode字符串(Unicode对象被扩充),例如:

```
Toggle line numbers
1 c = "%s %s" % ("Hello", u"World") # c = "Hello " + u"World"
2 d = u"%s %s" % ("Hello", "World") # d = u"Hello " + u"World"
```

当使用Unicode字符串时, str()和repr()函数会自动使用默认编码将Unicode字符串转换为标准字符串.对于一个Unicode字符串u, str(u)相当于u.encode(), repr(u)相当于u"%s" % repr(u.encode('unicode-escape')).

另外,许多库和内建函数只能用于操作标准字符串, Unicode字符串将会被自动使用默认编码转换为标准字符串.如果转换不可执行,会引发UnicodeError异常.

标准字符串和Unicode字符串可以比较.在这种情况下,标准字符串首先会使用默认编码强制转换为Unicode字符串.这个规则在列表和字典元素的比较操作中也同样适用.

例如 'x' in [u'x', u'y', u'z'] 强制将'x'转换为Unicode,并返回True. 对于从属测试, 'W' in u'Hello World' 也同理('W'被转换为Unicode).

当使用 hash() 函数计算哈希值时,标准字符串和Unicode字符串返回同一个值(当Unicode字符串只包含[U+0000, U+007F]中的字符时).这样就可以使标准字符串和Unicode字符串在用做字典关键字时可以互换(但条件还是Unicode字符串只包含[U+0000, U+007F]中的字符),例如:

Toggle line numbers

```
1 a = { }
2 a[u"foo"] = 1234
3 print a["foo"]           # Prints 1234
```

但是,应该注意在默认编码被改变为非ascii或者Unicode字符串包含非ASCII字符时,字典关键字不具有这种可以互换的行为.比如,如果'utf-8'被用做默认编码,字符串比较会返回相等,但哈希值不同:

```
#python
a = u"M\u00fcller"      # Unicode字符串
b = "M\303\274lller"    # utf-8 编码格式的 a
print a == b           # Prints True
print hash(a)==hash(b) # Prints False
```

注意, 上面的例子在python2.4中会引发异常.所以上面这些言论可能已经过时.

--Weizhong

1.8. 布尔表达式

and, or, 和 not 关键字可以组成布尔表达式.这些操作符的特性如下:

操作符	描述
x or y	如果 x 为假,返回 y ; 否则,返回 x
x and y	如果 x 为假,返回 x ; 否则,返回 y
not x	如果 x 为假,返回 True ; 否则,返回 False

当你使用一个表达式用来判断True 或 False时,任何非零的数字或非空列表,元组,字典,都返回True. 零,None,以及空列表,元组,字典返回False. 布尔表达式从左至右进行计算,而且具有短路行为,也就是说只有需要时才会进行右边表达式的计算.例如表达式 a and b 只有当a为True时才计算b.

注: 0 and 10/0 这样的表达式不会引发 除0错误, 因为 10/0 这个表达式被短路了. --Weizhong

1.9. 对象的比较与身份

相等运算符 x==y 检验x和y的值是否相等.在比较列表或元组,只有当所有的元素都相等时这两个对象才是相等的.对于字典,只有当x和y有相同的键和相同的对应值时,才会返回相等.

身份运算符 x is y 和 x is not y 检验两个对象在内存中否指向同一个对象.通常情况下, x==y,但 x is not y.

比较操作也可以在两个不兼容的对象类型之间进行,比如一个文件和一个浮点数,不过返回的结果是任意的,这样的比较没有任何意义.另外,比较两个不兼容的对象有可能会引发异常.

1.10. 运算优先级

Table 4.2列出了Python运算符的运算顺序(优先级).除乘方(**)外的所有运算符都是从左至右进行运算.表中靠前的运算符优先级要比后边的高些,也就是说,靠前的运算符在一个表达式中会先运算.(注:同一栏的运算符,如 $x * y$, x / y , $x \% y$ 有相同的优先级)

Table 4.2. 运算优先级 (由高到低)

运算	名称
<code>(...), [...], {...}</code>	创建元组,列表,字典
<code>`...`</code>	字符串转换
<code>s[i], s[i:j], .attr</code>	索引,切片,属性
<code>f(...)</code>	函数调用s
<code>+x , -x , ~x</code>	一元运算符
<code>x ** y</code>	乘方(从右至左运算)
<code>x * y , x / y , x % y</code>	乘,除,取模
<code>x + y , x - y</code>	加,减
<code>x << y , x >> y</code>	移位
<code>x & y</code>	按位与
<code>x ^ y</code>	按位异或
<code>x y</code>	按位或
<code>x < y , x <= y , x > y , x >= y , x == y , x != y</code>	比较,身份,序列成员检测
<code>x <> y x is y , x is not y x in s , x not in s</code>	
<code>not x</code>	逻辑非
<code>x and y</code>	逻辑与

x or y

逻辑或

lambda args : expr

lambda函数表达式

PythonEssentialRef4 (2006-10-25 06:26:02由WeiZhong编辑)

1. 第五章 控制流

本章描述程序中与控制流有关的语句.主题包括条件语句,循环及异常.

1.1. 条件语句

if,else,以及elif语句用来控制条件代码的执行.条件语句的通常格式如下:

```
if expression:
    statements
elif expression:
    statements
elif expression:
    statements
...
else:
    statements
```

如果不需要判断if条件外的其它情形,条件语句中的else从句和elif从句可以省略. pass语句用于不需要做任何事的特殊情形:

```
if expression:
    pass          # 不做任何事
else:
    statements
```

注: 上面的语句等价于

```
if !expression:
    statements
```

1.2. 循环

可以使用for或while语句实现循环:

```
while expression:
    statements

for i in s:
    statements
```

while语句循环执行块中的语句,直到表达式为假. for语句反复迭代一个序列中的元素,直到迭代完序列中最后一个元素。如果序列中的每个元素都是元素个数统一的元组,

目录

1. 第五章 控制流

1. 条件语句
2. 循环
3. 异常
4. 定义新的异常
5. 断言和__debug__

你可以以下面这样的形式应用for语句:

```
for x,y,z in s:
    statements
```

在这个例子中,s必须为一个元组的序列,每个元组有三个元素.每次循环中,元组的三个元素的值被分别赋值给 x,y,和z.

break语句用于立刻中止循环.下边的例子从用户输入中读入内容,当输入为空时退出循环:

```
while 1:
    cmd = raw_input('Enter command > ')
    if not cmd:
        break          # 无输入,退出循环
    # 运行命令
    ...
```

continue语句用于直接进入下一次循环(忽略当前循环的剩余语句)。这个语句可以使循环跳过不必要的语句.下列这个例子打印出一个序列中的非负元素:

```
Toggle line numbers
1 for a in s:
2     if a < 0:
3         continue      # 忽略负值元素
4     print a
```

break和continue语句只用于语句所在的当前循环,如果你需要退出一个多重循环,你应该使用异常。Python不提供goto语句.

你可以在一个循环结构中使用else语句:

```
Toggle line numbers
1 # while-else
2 while i < 10:
3     do something
4     i = i + 1
5 else:
6     print 'Done'
7
8 # for-else
9 for a in s:
10    if a == 'Foo':
11        break
12 else:
13    print 'Not found!'
```

循环中的else语句只在循环正常完成后运行(for或while循环),或者在循环条件不成立时立即运行(仅while循环),或者迭代序列为空时立即执行(仅for循环).如果循环使用break语句退出的话,else语句将被忽略.

1.3. 异常

异常意味着错误,未经处理的异常会中止程序运行. `raise`语句用来有意引发一个异常.`raise`语句的通常格式是`raise exception [, value]`, `exception`是异常类型, `value`是对于这个异常的特定描述.例如:

```
raise RuntimeError, 'Unrecoverable Error'
```

如果`raise`语句没有使用任何参数,最近一次发生的异常将被再次引发(尽管它只能用于处理一个刚刚发生的异常)

我们可以使用`try`和`except`语句来处理异常:

Toggle line numbers

```
1 try:
2     f = open('foo')
3 except IOError, e:
4     print "Unable to open 'foo': ", e
```

当一个异常发生时,解释器停止运行`try`块中的语句并寻找相应的`except`异常处理从句.若找到,控制就被传递到`except`块的第一条语句.否则,异常就被传递到上一级`try`语句语句块. `try-except`语句可以嵌套.如果异常被传递到整个程序的最顶层依然没有被处理,解释器就会终止程序运行,并显示出错信息.如果需要,不可捕获的错误也可以传递给用户定义函数 `sys.excepthook()` 处理.(参见附录A,The Python Library)

`except`语句中可选的第二个参数代表异常的说明,相当于`raise`语句的第二个可选参数.异常处理语句可以从这个值中得到关于异常原因的有用信息.

多个`except`语句可处理多种不同的异常, 参见下例:

```
try:
    do something
except IOError, e:
    # 处理 I/O error
    ...
except TypeError, e:
    # 处理 Type error
    ...
except NameError, e:
    # 处理 Name error
    ...
```

一个`except`语句也可以处理多个异常:

```
try:
    do something
except (IOError, TypeError, NameError), e:
    # 处理 I/O, Type, 或 Name errors
    ...
```

`pass`语句可以用来忽略异常:

```
try:
    do something
except IOError:
    pass # 不做任何事
```

省略异常名和值就可以捕获所有异常:

```
try:
    do something
except:
    print 'An error occurred'
```

Table 5.1. 内建异常类型

异常	描述
Exception	所有内建异常
SystemExit	由sys.exit()产生
StandardError	除SystemExit外所有内建异常
ArithmeticError	所有运算异常
FloatingPointError	浮点数运算异常
OverflowError	数值溢出
ZeroDivisionError	被零除
AssertionError	assert语句引起的异常
AttributeError	属性名称不可用时引起
EnvironmentError	Python外部错误
IOError	I/O 或与文件有关的错误(输入/输出错误)
OSError	操作系统错误
WindowsError	Windows错误
EOFError	当到达一个文件的末尾时引起
ImportError	import语句失败
KeyboardInterrupt	键盘中断(通常是 Ctrl+C)
LookupError	索引或关键字错误
IndexError	超出序列的范围
KeyError	不存在的字典关键字
MemoryError	内存不足
NameError	寻找局部或全局变量时失败
UnboundLocalError	未绑定变量
RuntimeError	一般运行时错误
NotImplementedError	不可实现的特征
SyntaxError	语法错误
TabError	不一致的制表符使用 (由 -tt 选项产生)
IndentationError	缩进错误
SystemError	解释器致命错误
TypeError	给一个操作传递了一个不适当的类型
ValueError	值错误(不合适或丢失)
UnicodeError	Unicode编码错误

try语句也支持else从句. else从句必须放在最后一个except从句后. 这块代码只在try块中的语句没有引发异常的时候运行.例如:

```
Toggle line numbers
1 try:
2     f = open('foo', 'r')
3 except IOError:
4     print 'Unable to open foo'
5 else:
```

```
6 data = f.read()
7 f.close()
```

`finally`语句定义了要在`try`块中代码的结束操作,例如:

Toggle line numbers

```
1 f = open('foo','r')
2 try:
3     # Do some stuff
4     ...
5 finally:
6     f.close()
7     print "File closed regardless of what happened."
```

`finally`语句并不用于捕获异常.它用来指示无论是否发生异常都要执行的语句块。如果没有引起异常,`finally`块中的语句将在`try`块中语句执行完毕后执行;如果有异常发生,控制将先传递到`finally`块中的第一条语句.在这块语句执行完后,异常被自动再次引发,然后交由异常处理语句处理. `finally`和`except`语句不能在同一个`try`语句中出现.

Table 5.1列出了Python中定义的全部内建异常类型.(关于异常的更多细节,参见附录A)

可以通过异常名称来访问一个异常。例如:

```
try:
    statements
except LookupError:      # 捕获 IndexError 或 KeyError
    statements
或
try:
    statements
except StandardError:    # 捕获任何内建的异常类型
    statements
```

1.4. 定义新的异常

所有的内建异常类型都是使用类来定义的.要定义一个新的异常,就创建一个父类为`exceptions.Exception`的新类:

Toggle line numbers

```
1 import exceptions
2 # Exception class
3 class NetworkError(exceptions.Exception):
4     def __init__(self, args=None):
5         self.args = args
```

`args`应该像上面那样使用.这样就可以使用`raise`语句来引发这个异常,并显示出错误返回信息以及诊断,如:

```
raise NetworkError, "Cannot find host."
```

通过调用`NetworkError("Cannot find host.")`可以创建一个`NetworkError`异常的实例.如:

Toggle line numbers

```
1 a=NetworkError("Cannot find host.")
2 print a                #得到 Cannot find host.
```

如果你使用一个不是 `self.args` 的属性名或你根本没有这个属性, 异常实例就没有这种行为.

当使用 `raise` 语句有意引发一个异常时, `raise` 语句的可选参数将做为该异常的构造函数(`__init__()`方法)参数. 如果异常的构造函数需要一个以上参数, 有两种方法可以用来引发这种异常:

Toggle line numbers

```
1 import exceptions
2 # Exception class
3 class NetworkError(exceptions.Exception):
4     def __init__(self, errno, msg):
5         self.args = (errno, msg)
6         self.errno = errno
7         self.errmsg = msg
8
9 # 方法一
10 raise NetworkError(1, 'Host not found')
11
12 # 方法二
13 raise NetworkError, (1, 'Host not found')
```

基于类的这种异常体制让你能够轻易创建多级异常. 例如, 前边定义的 `NetworkError` 异常可以用做以下异常的基类:

Toggle line numbers

```
1 class HostnameError(NetworkError):
2     pass
3
4 class TimeoutError(NetworkError):
5     pass
6
7 def error3():
8     raise HostnameError
9
10 def error4():
11     raise TimeoutError
12
13 try:
14     error3()
15 except NetworkError:
16     import sys
17     print sys.exc_type    # 打印出异常类型
```

在这个例子中 `except NetworkError` 语句能捕获任何从 `NetworkError` 中继承而来的异常. 通过变量 `sys.exc_type` 可以得到这个特殊异常的名称. `sys.exc_info()` 函数用于返回最近一个异常的信息(不依靠全局变量, 属于安全线程).

1.5. 断言和__debug__

assert语句用来断言某个条件是真的,常用于程序调试. assert语句的一般格式为:

```
assert test [, data]
```

test是一个表达式,它返回True或False. 如果test的值为假, assert语句就引发AssertionError异常,可选的data字符串将被传递给这个异常.例如:

Toggle line numbers

```
1 def write_data(file,data):
2     assert file, "write_data: file is None!"
3     ...
```

实际上assert语句在执行时会被实时翻译为下面的代码:

Toggle line numbers

```
1 if __debug__:
2     if not (test):
3         raise AssertionError, data
```

__debug__是一个内建的只读值,除非解释器运行在最佳化模式(使用 -O 或 -OO 选项), 否则它的值总是 True. 虽然__debug__被设计为供 assert 语句使用,你仍然可以在任何自定义调试代码中使用它.

assert语句不能用于用来确保程序执行正确的场合,因为该语句在最佳化模式下会被忽略掉.尤其不要用assert来检查用户输入. assert语句用于正常情况下应该总是为真的场合;若assert语句引发了异常,那就代表程序中存在bug,是程序员出了问题而不是用户出现了问题.

如果打算将上边的 write_data() 函数交付给最终用户使用, assert语句就应该使用if语句和错误处理语句来重写.

1. 第六章 函数与函数编程

为便于代码维护,绝大多数子程序都被分解并重新组织为函数以使代码模块化。在Python中定义一个函数很简单,Python从其它函数编程语言中借鉴了很多有用的思路用来简化某些特定任务。本章的主题是函数,匿名函数,函数编程特性及eval()与execfile()函数和exec语句.还详细描述了列表内函(list comprehensions),一个强大的列表构建方法。

目录

- 1. 第六章 函数与函数编程
 - 1. 函数
 - 2. 参数传递和返回值
 - 3. 作用域规则
 - 4. 递归
 - 5. apply()函数
 - 6. lambda操作符
 - 7. map(), zip(), reduce(), 和filter()
 - 8. 列表内函
 - 9. eval(), exec, execfile(),和compile()

1.1. 函数

函数使用def语句定义:

Toggle line numbers

```
1 def add(x,y):  
2     return x+y
```

要调用一个函数,只要使用函数名加上小括号括起来的参数表就可以了,例如 a = add(3,4). 参数的顺序和个数必须和函数定义中的相匹配.否则会引发TypeError异常.

定义函数的时可以使用参数默认值, 如:

```
def foo(x,y,z = 42):
```

若函数定义中有存在默认值的参数,这个参数就是可选参数.

默认参数的值在函数定义的时候就被决定,并且不会改变,例如:

Toggle line numbers

```
1 a = 10  
2 def foo(x = a):  
3     print x  
4 a = 5 # Reassign 'a'.  
5 foo() # Prints '10' (默认值没有改变)
```

但是,若使用可变对象作为默认参数值,则会有意料之外的情况发生:

Toggle line numbers

```
1 a = [10]  
2 def foo(x = a):  
3     print x  
4 a.append(20)  
5 foo() # Prints '[10, 20]'
```

如果最后一个参数名前有星号(*),函数就可以接受可变数量的参数,这些不定数量的参数将做为一个元组传递给函数:

Toggle line numbers

```
1 def fprintf(file, fmt, *args):
2     file.write(fmt % args)
3
4 # fprintf.args 被赋值为 (42, "hello world", 3.45)
5 fprintf(out, "%d %s %f", 42, "hello world", 3.45)
```

在这个例子中,所有剩下的参数都被放入一个元组,赋值给args. 使用*args还可以把元组args传递给另一个函数:

Toggle line numbers

```
1 def printf(fmt, *args):
2     # Call another function and pass along args
3     fprintf(sys.stdout, fmt, *args)
```

你也可以明确给每个形参名字绑定一个特定值(这称为关键字参数),然后传递给一个函数,如下:

Toggle line numbers

```
1 def foo(w,x,y,z):
2     print w,x,y,z
3
4 #以关键字参数形式调用函数
5 foo(x=3, y=22, w='hello', z=[1,2])
```

使用这种方式调用函数,参数可以是任意顺序(不必与定义时顺序相同).但是,除非你省略的参数有默认值,否则你必须显式的给函数中所有形参名字指定一个值.如果是省略了某个必须的参数或你提供了一个函数定义中不存在的形参名字,就会引发TypeError异常.

传统的参数与关键字参数可以在同一个函数调用中混合使用,一个前提是必须先给出固定位置的参数,例如:

```
foo('hello', 3, z=[1,2], y=22)
```

如果一个函数定义中的最后一个形参有** (双星号) 前缀,所有正常形参之外的其他的关键字参数都将被放置在一个字典中传递给函数,例如:

Toggle line numbers

```
1 def spam(**parms):
2     print "You supplied the following args:"
3     for k in parms.keys():
4         print "%s = %s" % (k, parms[k])
5 spam(x=3, a="hello", foobar=(2, 3))
```

常规参数, *参数及**参数可以同时使用, 这时**参数必须位于参数表的最后:

Toggle line numbers

```
1 # Accept variable number of positional or keyword arguments
2 def spam(x, *args, **keywords):
3     print x, args, keywords
```

使用**关键字语法也可以把关键字参数传递给另一个函数,如:

Toggle line numbers

```
1 def callfunc(func, *args, **kwargs):
2     print args
3     print kwargs
4     func(*args, **kwargs)
```

从Python 2.1开始,函数和方法可以拥有任意的属性,例如:

Toggle line numbers

```
1 def foo():
2     print "Hello world"
3
4 foo.secure = 1
5 foo.private = 1
```

注意:这仅仅是自定义函数的特权,内建函数或者类的方法是没有这种行为的。

--WeiZhong

函数的属性被储存在一个字典中(函数的 `__dict__` 属性).

某些特定应用程序如语法分析器或网络应用程序需要在一个函数中携带附加信息,函数属性完美的满足了这一需求.在Python2.1之前,只能用文档字符串来储存这些信息(这有很大的局限性,比如只能存储字符串对象,并且有违文档字符串功能的初衷).

1.2. 参数传递和返回值

当调用一个函数时,它的参数是按引用传递给.如果函数的实参一个可变对象(如列表或字典),则函数内对该对象的修改将会影响到函数之外.例如:

Toggle line numbers

```
1 a = [1,2,3,4,5]
2 def foo(x):
3     x[3] = -55    # 修改 x 中的一个元素
4
5 foo(a)          # 传递 a
6 print a        # 显示 [1,2,3,-55,5]
```

`return`语句用于从函数中返回一个对象.如果没有指定返回对象或者`return`语句被省略,则会返回一个`None`对象.如果要返回多个值,可以通过返回一个元组或其它包含对象来完成。

Toggle line numbers

```
1 def factor(a):
2     d = 2
3     while (d <= (a/2)):
4         if ((a/d)*d == a):
5             return ((a/d),d)
6         d = d + 1
7     return (a,1)
```

如果返回值是一个元组,可以通过下面的方式来将返回值一次赋给多个独立变量:

Toggle line numbers

```
1 x,y = factor(1243)    # 返回的值被赋值给 x 和 y.
2 (x,y) = factor(1243) # 同样的效果
```

1.3. 作用域规则

当一个函数开始运行,就会创建一个新的局部名字空间。该名字空间用来存放函数的形参名字及该函数中所使用的全部局部变量名。当解析一个变量名时,解释器首先在这个局部名字空间中搜索.如果没有找到,紧接着搜索全局名字空间.一个函数的全局名字空间就是定义该函数的模块.如果在全局名字空间中还没有找到匹配,解释器接着在内建名字空间中搜索.若仍然找不到这个变量名,则引发NameError异常.

名字空间的一个特性是: 在函数内部即使有一个变量与一个全局变量同名, 也各不相干(因为它们位于不同的名字空间).例如下边的代码:

Toggle line numbers

```
1 a = 42
2 def foo():
3     a = 13
4 foo()
5 print a
```

尽管我们在函数foo中修改了变量a的值,这个例子返回的结果仍然是42.如果一个变量在函数内部被赋值,则它一定是这个函数的局部变量(除非事先使用了global关键字)。在函数foo中的变量a其实是一个全新的值为13的对象,与函数外的a是不同的对象.要在函数内部使用全局变量,你应该在函数内使用global语句.global语句明确的声明一个或多个变量(如果有多个变量,以逗号分隔这些变量)属于全局名字空间.例如:

Toggle line numbers

```
1 a = 42
2 def foo():
3     global a          # 'a' 在全局名字空间
4     a = 13
5 foo()
6 print a
```

所有的Python版本都允许嵌套的函数定义.但在Python 2.1之前的版本,嵌套函数并未提供嵌套作用域.因此在老版本的Python中,嵌套函数的运行结果有可能与你的预期不同.比如下面这个例子,虽然它是合法的,但在Python2.0中,它的执行并不象你想象的那样:

Toggle line numbers

```
1 def bar():
2     x = 10
3     def spam():          # 嵌套函数定义
4         print 'x is ', x # 在bar()的全局名字空间中寻找x
5     while x > 0:
6         spam()           # 若在Python2.0中运行该代码 程序会报错 :
NameError on 'x'
7         x -= 1
```

在Python2.1之前的版本中,当嵌套函数spam()运行时,它的全局名字空间会与bar()相

同,都是函数被定义的模块.所以spam()无法得到它希望得到的bar()名字空间中的变量,这就引发了NameError异常.

从Python 2.1开始支持嵌套作用域(这样,上边的例子就会正常运行):解释器将首先在局部名字空间中搜索变量名,然后一层层向外搜索,最后搜索全局名字空间和内建名字空间。注意嵌套范围在Python 2.1是一个可选的功能,只有当你的程序包含

from __future__ import nested_scopes 时才启用该功能.(具体细节参见在第十章--运行环境).另外,如果你需要考虑和较老Python版本的兼容性,那么就应该避免使用嵌套函数.

```
注: Python 2.4中该功能已经是内建功能,不需要做那个 from __future__
import nested_scopes 操作了 --Weizhong
```

如果一个局部变量在它被赋值之前使用,会引发一个UnboundLocalError异常,例如:

Toggle line numbers

```
1 def foo():
2     print i          # 导致UnboundLocalError exception异常
3     i = 0
```

1.4. 递归

Python对递归函数调用的次数作了限制.函数 sys.getrecursionlimit()返回当前允许的最大递归次数,而函数sys.setrecursionlimit()可以改变该函数的返回值.默认的最大递归次数为1000.当一个函数递归次数超过最大递归次数时,就会引发RuntimeError异常.

1.5. apply() 函数

apply(func [, args [, kwargs]]) 函数用于当函数参数已经存在于一个元组或字典中时间接的调用函数. args是一个包含将要提供给函数的按位置传递的参数的元组. 如果省略了args,任何参数都不会被传递. kwargs是一个包含关键字参数的字典.下面的语句效果是一样的:

Toggle line numbers

```
1 foo(3,"x", name='Dave', id=12345)
2 apply(foo, (3,"x"), { 'name': 'Dave', 'id': 12345 })
```

在Python较老的版本里, apply()是在当参数已经位于元组或字典中时调用函数的唯一机制.不过现在,你还可以使用更直接更简单的方式,如下:

Toggle line numbers

```
1 a = (3,"x")
2 b = { 'name' : 'Dave', 'id': 12345 }
3 foo(*a,**b)      # 与上边的代码相同
```

1.6. lambda操作符

lambda语句用来创建一个匿名函数(没和名字绑定的函数):

lambda args: expression

args是一个用逗号分隔的参数, expression是一个调用这些参数的表达式,例如:

Toggle line numbers

```
1 a = lambda x,y : x+y
2 print a(2,3)           # 打印出 5
```

lambda定义的代码必须是一个合法的表达式.多重语句和其他非表达式语句(如print, for, while等)不能出现在lambda语句中. lambda表达式也遵循和函数一样的作用域规则.

lambda 已经是过时的语句, 即将被废除。 --WeiZhong

1.7. map(), zip(), reduce(), 和filter()

t = map(func, s)函数将序列s中的每个元素传递给func函数做参数, 函数的返回值组成了列表t. 即t[i] = func(s[i]). 需要注意的是, func函数必须有只有一个参数,例如:

Toggle line numbers

```
1 a = [1, 2, 3, 4, 5, 6]
2 def foo(x):
3     return 3*x
4 b = map(foo, a)   # b = [3, 6, 9, 12, 15, 18]
```

上边的例子中的函数也可以用匿名函数来创建:

Toggle line numbers

```
1 b = map(lambda x: 3*x, a)   # b = [3, 6, 9, 12, 15, 18]
```

map()函数也可以用于多个列表,如 t = map(func, s1, s2, ..., sn). 如果是这种形式,t中的每个元素 t[i] = func(s1[i], s2[i], ..., sn[i]). func函数的形参个数必须和列表的个数(n)相同,结果与s1,s2, ... sn中的最长的列表的元素个数相同.在计算过程中,短的列表自动用None扩充为统一长度的列表.

如果函数func为None,则func就被当成是恒等函数处理. 这样函数就返回一个包含元组的列表:

Toggle line numbers

```
1 a = [1,2,3,4]
2 b = [100,101,102,103]
3 c = map(None, a, b)   # c = [(1,100), (2,101), (3,102), (4,103)]
```

上边这个例子也可以用 zip(s1, s2, ..., sn) 函数来完成. zip()用来将几个序列组合成一个包含元组的序列,序列中的每个元素t[i] = (s1[i], s2[i], ..., sn[i]). 与map()不同的是, zip()函数将所有较长的序列序列截的和最短序列一样长:

Toggle line numbers

```
1 d = [1,2,3,4,5,6,7]
2 e = [10,11,12]
3 f = zip(d,e)   # f = [(1,10), (2,11), (3,12)]
```

reduce(func, s)函数从一个序列收集信息,然后只返回一个值(例如求和,最大值,等).它首先以序列的前两个元素调用函数,再将返回值和第三个参数作为参数调用函数,依次执行下去,返回最终的值. func函数有且只有两个参数.例如:

Toggle line numbers

```
1 def sum(x,y):
2     return x+y
3
4 b = reduce(sum, a)    # b = (((1+2)+3)+4) = 10
```

`filter(func,s)`是个序列过滤器，它使用`func()`函数来过滤`s`中的元素。使`func`返回值为`false`的元素被丢弃，其它的存入`filter`函数返回的列表中,例如:

Toggle line numbers

```
1 c = filter(lambda x: x < 4, a)    # c = [1, 2, 3]
```

如果函数`func`为`None`,则`func`就被当成是恒等函数处理。这样,函数就返回序列`s`中值为`True`的元素.

1.8. 列表内涵

列表内涵可以代替许多调用`map()`和`filter()`函数的操作.列表内涵的一般形式是:

```
[表达式 for item1 in 序列1
      for item2 in 序列2
      ...
      for itemN in 序列N
      if 条件表达式]
```

上边的例子等价于:

Toggle line numbers

```
1 s = []
2 for item1 in sequence1:
3     for item2 in sequence2:
4         ...
5         for itemN in sequenceN:
6             if condition: s.append(expression)
```

Listing 6.1 中的例子可以帮助你理解列表内涵

Listing 6.1 列表内涵

Toggle line numbers

```
1 import math
2 a = [-3,5,2,-10,7,8]
3 b = 'abc'
4 c = [2*s for s in a]           # c = [-6,10,4,-20,14,16]
5 d = [s for s in a if s >= 0]  # d = [5,2,7,8]
6 e = [(x,y) for x in a
7       for y in b
8       if x > 0]               # e = [(5,'a'),(5,'b'),(5,'c'),
9                               #      (2,'a'),(2,'b'),(2,'c'),
10                              #      (7,'a'),(7,'b'),(7,'c'),
11                              #      (8,'a'),(8,'b'),(8,'c')]
12 f = [(1,2), (3,4), (5,6)]
13 g = [math.sqrt(x*x+y*y)
14       for x,y in f]           # f = [2.23606, 5.0, 7.81024]
```



```
13 h = reduce(lambda x,y: x+y, # 平方根的和
14             [math.sqrt(x*x+y*y)
15             for x,y in f])
```

提供给列表内涵的序列不必等长,因为系统内部使用嵌套的一系列for循环来迭代每个序列中的每个元素,然后由if从句处理条件表达式,若条件表达式为真,计算表达式的值并放入到列表内涵返回的序列中.if从句是可选的.

当使用列表内涵来构建包含元组的列表时,元组的值必须放在括号里.例如 [(x,y) for x in a for y in b]是一个合法的语句,而 [x,y for x in a for y in b]则不是.

最后,你应该注意在一个列表内涵中定义的变量是与列表内涵本身有同样的作用域,在列表内涵计算完成后会继续存在.例如 [x for x in a] 会覆盖内涵外先前定义的x,最终x的值会是a中的最后一个元素的值.

1.9. eval(), exec, execfile(), 和compile()

eval(str [,globals [,locals]])函数将字符串str当成有效Python表达式来求值,并返回计算结果.

同样地,exec语句将字符串str当成有效Python代码来执行.提供给exec的代码的名称空间和exec语句的名称空间相同.

最后,execfile(filename [,globals [,locals]])函数可以用来执行一个文件,看下面的例子:

```
>>> eval('3+4')
7
>>> exec 'a=100'
>>> a
100
>>> execfile(r'c:\test.py')
hello,world!
>>>
```

默认的,eval(),exec,execfile()所运行的代码都位于当前的名字空间中.eval(),exec,和execfile()函数也可以接受一个或两个可选字典参数作为代码执行的全局名字空间和局部名字空间.例如:

Toggle line numbers

```
1 globals = {'x': 7,
2           'y': 10,
3           'birds': ['Parrot', 'Swallow', 'Albatross']}
4         }
5 locals = { }
6
7 # 将上边的字典作为全局和局部名称空间
8 a = eval("3*x + 4*y", globals, locals)
9 exec "for b in birds: print b" in globals, locals # 注意这里的语法
10 execfile("foo.py", globals, locals)
```

如果你省略了一个或者两个名称空间参数,那么当前的全局和局部名称空间就被使用.如果一个函数体内嵌嵌套函数或lambda匿名函数时,同时又在函数主体中使用exec或

execfile()函数时，由于牵到嵌套作用域，会引发一个SyntaxError异常。（此段原文:If you omit one or both namespaces, the current values of the global and local namespaces are used. Also,due to issues related to nested scopes, the use of exec or execfile() inside a function body may result in a SyntaxError exception if that function also contains nested function definitions or uses the lambda operator.）

在Python2.4中俺未发现可以引起异常 --WeiZhong

注意例子中exec语句的用法和eval(), execfile()是不一样的. exec是一个语句(就象print或while), 而eval()和execfile()则是内建函数.

exec(str) 这种形式也被接受，但是它没有返回值。 --WeiZhong

当一个字符串被exec,eval(),或execfile()执行时,解释器会先将它们编译为字节代码,然后再执行.这个过程比较耗时,所以如果需要对某段代码执行很多次时,最好还是对该代码先进行预编译,这样就不需要每次都编译一遍代码,可以有效提高程序的执行效率。

compile(str, filename, kind)函数将一个字符串编译为字节代码, str是将要被编译的字符串, filename是定义该字符串变量的文件, kind参数指定了代码被编译的类型-- 'single'指单个语句, 'exec'指多个语句, 'eval'指一个表达式. compile()函数返回一个代码对象, 该对象当然也可以被传递给eval()函数和exec语句来执行,例如:

Toggle line numbers

```
1 str = "for i in range(0,10): print i"
2 c = compile(str, '', 'exec')      # 编译为字节代码对象
3 exec c                            # 执行
4
5 str2 = "3*x + 4*y"
6 c2 = compile(str2, '', 'eval')   # 编译为表达式
7 result = eval(c2)                # 执行
```

PythonEssentialRef6 (2006-02-09 09:43:31由WeiZhong编辑)

1. 第七章 类及面向对象编程

类是用来创建数据结构和新类型对象的主要机制.本章的主题就是类,面向对象编程和设计不是本章的重点。本章假定你具有数据结构的背景知识及一定的面向对象的编程经验(其它面向对象的语言,比如java,c++).(参见第三章,类型和对象 了解对象这个术语及其内部实现的附加信息)

Weizhong补充:

这本书出版于2001年,虽然Python有极佳的向下兼容性,但我们应该学习最新的知识。本章很多地方已经明显过时,为了保证大家学到新的知识并维持这本书的完整性,我会在必要的地方说明哪些地方已经过时,哪些地方新增了功能。

Python从2.2起引入了new-style对象模型,以逐步替代已经使用多年的classic对象模型。

由于 classic class 已经行将废止,所以我对本章的例子均作为了适当的修改以支持 new-style对象模型。

参考文档:《Python In a Nutshell》中的一节Python中的新型类及其实例

1.1. class语句

一个类定义了一系列与其实例对象密切关联的属性.典型的属性包括变量(也被称为类变量)和函数(又被称为方法).

class语句用来定义一个类.类的主体中语句在类定义同时执行.(如 Listing 7.1)

Listing 7.1 类

切换行号显示

```
1 class Account(object):
2     "一个简单的类"
3     account_type = "Basic"
4     def __init__(self,name,balance):
5         "初始化一个新 Account 实例"
6         self.name = name
7         self.balance = balance
8     def deposit(self,amt):
9         "存款"
10        self.balance = self.balance + amt
11    def withdraw(self,amt):
12        "取款"
13        self.balance = self.balance - amt
14    def inquiry(self):
15        "返回当前余额"
16        return self.balance
```

1.2. 访问类属性

类对象作为一个名字空间，存放在类定义语句运行时创建的对象.例如,Account里的内容可以这样访问:

```
Account.account_type
Account.__init__
Account.deposit
Account.withdraw
Account.inquiry
```

需要注意的是, class语句并不创建类的实例(例如上边的例子,并没有创建任何帐户). 它用来定义所有实例都应该有的属性.

在类中定义的常规方法的第一个参数总是该类的实例,通常这个参数记为self. 你也可能用其它任何合法的变量名, 不过为了符合惯例, 你最好还是用self. 类中定义的变量,即类变量, 如account_type, 它被所有该类的实例共享. 虽然类定义了一个名字空间,但这个名字空间并不是为类主体中的代码服务的.因此在类中引用一个类的属性必须使用类的全名:

切换行号显示

```
1 class Foo(object):
2     def bar(self):
3         print "bar!"
4     def spam(self):
5         bar(self)      # 错误,引发NameError
6         Foo.bar(self) # 合法的
```

最后, 你不能定义一个不操作实例的方法:

切换行号显示

```
1 class Foo(object):
2     def add(x,y):
3         return x+y
4 a = Foo.add(3,4)      # TypeError. 需要一个类实例作为第一个参数
```

以下为WeiZhong增补部分: **静态方法和类方法(Python2.2以上)**

- 静态方法:

可以直接被类或类实例调用。它没有常规方法那样的特殊行为（绑定、非绑定、默认的第三个参数规则等等）。你完全可以将静态方法当成一个用属性引用方式调用的普通函数。任何时候定义静态方法都不是必须的（静态方法能实现的功能都可以通过定义一个普通函数来实现）。有些程序员认为，当有一堆函数仅仅为某一特定类编写时，将这些函数包装成静态这种方式可以提供使用上的一致性。

根据python2.4最新提供的新语法，你可以用下面的方式创建一个静态方法:

```
class AClass(object):
    @staticmethod      #静态方法修饰符, 表示下面的方法是一个静态方法
    def astatic( ): print 'a static method'
anInstance = AClass( )
AClass.astatic( )      # prints: a static method
anInstance.astatic( ) # prints: a static method
```

注:staticmethod是一个内建函数,用来将一个方法包装成静态方法,在2.4以前版本,只能

用下面这种方式定义一个静态方法(不再推荐使用):

```
class AClass(object):
    def astatic( ): print 'a static method'
    astatic=staticmethod(astatic)
```

这种方法在函数定义本身比较长时经常会忘记后面这一行.

- 类方法

一个类方法就可以通过类或它的实例来调用的方法,不管你是用类来调用这个方法还是类实例调用这个方法,该方法的第一个参数总是定义该方法的对象。记住:方法的第一个参数都是类对象而不是实例对象。按照惯例,类方法的第一个形参被命名为 cls. 任何时候定义类方法都不是必须的(类方法能实现的功能都可以通过定义一个普通函数来实现,只要这个函数接受一个类对象做为参数就可以了)。你可以象下面这样来生成一个类方法:

```
class ABase(object):
    @classmethod          #类方法修饰符
    def aclassmet(cls): print 'a class method for', cls.__name__
class ADeriv(ABase): pass
bInstance = ABase( )
dInstance = ADeriv( )
ABase.aclassmet( )          # prints: a class method for ABase
bInstance.aclassmet( )     # prints: a class method for ABase
ADeriv.aclassmet( )        # prints: a class method for ADeriv
dInstance.aclassmet( )     # prints: a class method for ADeriv
```

注:classmethod是一个内建函数,用来将一个方法封装成类方法,在2.4以前版本,你只能用下面的方式定义一个类方法:

```
class AClass(object):
    def aclassmethod(cls): print 'a class method'
    aclassmethod=classmethod(aclassmethod)
```

并没有人要求必须封装后的方法名字必须与封装前一致,但建议你总是这样做(如果你使用python2.4版本以下时)。这种方法在函数定义本身比较长时经常会忘记后面这一行。

=====
增补部分至此结束

1.3. 类实例

像调用函数一样调用类,可以得到类的实例。生成实例的过程会自动调用类的__init__方法(如果你的类定义了这个方法的话)。

切换行号显示

```
1 # 创建一些帐户
2 a = Account("Guido", 1000.00)      # 调用
Account.__init__(a,"Guido",1000.00)
3 b = Account("Bill", 100000000000L)
```

实例创建之后,就可以使用点(.)操作符来访问它的属性和方法:

切换行号显示

```
1 a.deposit(100.00)      # 调用 Account.deposit(a,100.00)
2 b.withdraw(sys.maxint) # 调用 Account.withdraw(b,sys.maxint)
3 name = a.name          # 得到帐户名称
4 print a.account_type   # 显示帐户类型
```

在系统内部,每个类实例都拥有一个字典(即实例的 `dict` 属性,在第三章中有介绍).这个字典包含每个实例的信息.例如:

```
>>> print a.__dict__
{'balance': 1100.0, 'name': 'Guido'}
>>> print b.__dict__
{'balance': 97852516353L, 'name': 'Bill'}
```

若一个实例的属性被修改,这个字典也随之改变.上例中,属性通过 `Account` 类中定义的方法 `__init()`, `deposit()`, 以及 `withdraw()` 中对 `self` 变量赋值被改变. 不过对于类实例可以随时添加私有属性。

```
a.number = 123456      # 把 'number' 加入到 a.__dict__
```

属性的赋值总是发生在实例字典中,而属性访问则比属性赋值复杂一些。当访问一个属性的时候,解释器首先在实例的字典中搜索,若找不到则去创建这个实例的类的字典中搜索,若还找不到就到类的基类中搜索(在后边'继承'一节中会讲到),如果还找不到最后会尝试调用类的 `__getattr__` 方法来获取属性值(若类中定义了该方法的话).如果这个过程也失败,则引发 `AttributeError` 异常

1.4. 引用记数与实例销毁

所有实例都是引用记数的.若一个实例引用记数变成零,该实例就被销毁.当实例将被销毁前,解释器会搜索该对象的 `__del__` 方法并调用它.但在实际应用中,极少有需要给一个类定义 `__del__` 方法,除非这个对象在销毁前需要执行一些清除操作(如关闭文件,断开网络,或者释放其他系统资源).即使是在这种情况下,依赖 `__del__()` 来执行清除和关闭操作也是危险的,因为不能保证在解释器关闭时会自动调用这个方法.更好的选择是定义一个 `close()` 方法,在需要时显式的调用这个方法来执行这个过程.最后注意一点,如果一个实例拥有 `__del__` 方法,则它永远不会被Python的垃圾收集器回收(这也是不推荐定义 `__del__()` 的理由).关于垃圾回收请参阅附录A中的 `gc` 模块。

有时会使用 `del` 语句来删除对象的引用,如果这导致该对象引用记数变为零,就会自动调用 `__del__()`. `del` 语句并不直接调用 `__del__()`.

1.5. 继承

继承(Inheritance)是创建新类的机制之一,它通过一个已有类进行修改和扩充来生成新类.这个原始类被称为基类(base class)或超类(superclass).新生成的类称为该类的派生类(derived class)或子类(subclass).当通过继承创建一个类时,它会自动'继承'在基类中定义的属性.一个子类也可以重新定义父类中已有的属性或定义新的属性.

Python支持多继承,如果一个类有多个父类,在 `class` 语句中就使用逗号来分隔这个

父类列表。例如:

切换行号显示

```
1 class D(object): pass #D继承自object
2 class B(D): #B是D的子类
3     varB = 42
4     def method1(self):
5         print "Class B : method1"
6 class C(D): #C也是D的子类
7     varC = 37
8     def method1(self):
9         print "Class C : method1"
10    def method2(self):
11        print "Class C : method2"
12 class A(B,C): #A是B和C的子类
13     varA = 3.3
14     def method3(self):
15         print "Class A : method3"
```

当搜索在基类中定义的某个属性时, Python采用深度优先的原则、按照子类定义中的基类顺序进行搜索。****注意****(new-style类已经改变了这种行为)。上边例子中, 如果访问 `A.varB`, 就会按照A-B-D-C-D这个顺序进行搜索, 只要找到就停止搜索.若有多个基类定义同一属性的情况, 则只使用第一个被找到属性值:

切换行号显示

```
1 a = A() # 创建 'A' 的实例
2 a.method3() # 调用 A.method3(a)
3 a.method1() # 调用 B.method1(c)
4 a.varB # 得到 B.varB
```

重要提示: 新旧对象模型的差异:

注意: Python 中现在有两种对象模型均在使用中即classic对象模型和new-style对象模型, 也有两种类: classic class 及 new-style class

在classic对象模型中, 方法和属性按 从左至右 深度优先 的顺序查找(上文中已经提到). 显然, 当多个父类继承自同一个基类时, 这会产生我们不想要的结果.

就上例来说, D是一个new-style类(继承自object), B和C是D的子类, 而A是B和C的子类, 如果按classic对象模型(原文中的提到的对象模型)的属性查找规则是搜索顺序是A-B-D-C-D. 由于Python先查找D后查找C, 即使C对D中的属性进行了重定义, 也只能使用D中定义的版本. 这是classic数据模型的固有问题, 在实际应用中会造成一些麻烦. 为了解决这个及其它一些问题, Python从2.2版本开始引入new-style对象模型.

在new-style对象模型中, 所有内建类型均是object的直接或间接子类. new-style对象模型改变了传统对象模型中的解析顺序, 上面的例子我已经改写为new-style类, 因此, 这个例子实际的搜索顺序是 A-B-C-D.

每个内建类型及new-style类均内建有一个特殊的只读属性 `__mro__`, 这是一个tuple, 它保存着方法解析类型. 只能通过类来引用 `__mro__` (通过实例无法访问).

--WeiZhong Added@20060210

如果一个子类定义了一个和基类具有相同名称的属性, 则子类的实例将使用子类中定义的属性. 如果需要访问原来的属性, 则必须使用全名来限制访问区域:

切换行号显示

```
1 class D(A):
2     def method1(self):
3         print "Class D : method1"
4         A.method1(self)           # 调用基类属性
```

需要注意的一点是子类实例的初始化.当一个子类实例被创建时,基类的 `__init__()` 方法并不会被自动调用.也就是子类必须自力更生来解决实例的初始化.例如:

切换行号显示

```
1 class D(A):
2     def __init__(self, args1):
3         # 初始化基类
4         A.__init__(self)
5         # 初始化自己
6         ...
```

`__del__()` 与 `__init__()` 类似.

1.6. 多态

Python通过上文中提到的属性查询规则来实现多态.当使用`obj.method()`来访问一个方法时,方法的搜索顺序为:实例的 `__dict__` 属性,实例的类定义,基类.第一个被找到的方法被执行.

1.7. 数据隐藏

默认情况下,所有的属性都是'公开'的.这意味着一个类的所有属性均可不受任何限制的访问.这也意味着基类中定义的所有内容都能被子类继承.在面向对象编程实践中,这种行为是我们不希望的.因为它不但暴露了对象的内部实现,而且容易在派生类对象及基类对象之间产生名字空间冲突.

要解决这个问题,只需要在类中将需要隐藏的属性名字以两个下划线开头,例如 `__Foo`.这样系统会自动实时生成一个新的名字 `__Classname__Foo` 并用于内部使用.这样在某种程度上就提供了私有属性(其实这个 `__Classname__Foo` 仍然是不受限制访问的嘿嘿),也解决了名字空间冲突的问题.例如:

切换行号显示

```
1 class A:
2     def __init__(self):
3         self.__X = 3           # self.__A__X
4
5 class B(A):
6     def __init__(self):
7         A.__init__(self)
8         self.__X = 37         # self.__B__X
```

这是一个小技巧,并没有真正阻止访问一个类的*私有*属性.如果已知一个类的名称和它某个私有属性的名称,我们还是可以使用 `__Classname__Foo` 来访问到这个属性.(这不是bug,因为在某些特定的场合这非常有用,比如调试时,所以系统一直保留这个所谓的*问题*)

1.8. 操作符重载

用户自定义对象可以通过在类中实现特殊方法(第三章中已介绍)来重载Python内建操作符.例如 Listing 7.2 中的类,它使用标准的数学运算符实现了复数的运算及类型转换.

Listing 7.2 数学运算及类型转换

切换行号显示

```
1 class Complex(object):
2     def __init__(self,real,imag=0):
3         self.real = float(real)
4         self.imag = float(imag)
5     def __repr__(self):
6         return "Complex(%s,%s)" % (self.real, self.imag)
7     def __str__(self):
8         return "(%g+%gj)" % (self.real, self.imag)
9     # self + other
10    def __add__(self,other):
11        return Complex(self.real + other.real, self.imag +
other.imag)
12    # self - other
13    def __sub__(self,other):
14        return Complex(self.real - other.real, self.imag -
other.imag)
15    # -self
16    def __neg__(self):
17        return Complex(-self.real, -self.imag)
18    # other + self
19    def __radd__(self,other):
20        return Complex.__add__(other,self)
21    # other - self
22    def __rsub__(self,other):
23        return Complex.__sub__(other,self)
24    # 将其他数值类型转换为复数
25    def __coerce__(self,other):
26        if isinstance(other,Complex):
27            return self,other
28        try:    # 检测是否可以被转换为浮点数
29            return self, Complex(float(other))
30        except ValueError:
31            pass
```

在这个例子中,有一些值得研究的地方:

首先__repr__()

用于返回对象的表达式字符串表示,这个返回字符串可以用于再次得到该对象.在本例中,会创建一个类似"Complex(r,i)"的字符串.另外__str__()方法创建一个字符串用于较美观的输出。(通常用于print语句)

然后,要处理复数在运算符左边或右边这两种情况,必须同时提供__op__()和__rop__()方法.

最后,__ceorco__方法用于处理混合类型运算.在本例中,其他的数值类型均被转换为复数,这样才可以继续进行复数的运算.

1.9. 类, 类型, 和成员检测

目前,类型和类是分开的.内建类型,如列表和字典是不能被继承的,类也不能定义一个新类型.事实上,所有的类定义都属于ClassType类型,同样地,类的实例属于InstanceType类型.所以,下面这个表达式对于两个类永远为真(即使这两个实例是由不同的类创建的): `type(a) == type(b)`

```
Python 2.4 已经支持内建类型的继承,类与类型还有差别,但越来越微妙了。
对 new-style 类来说,类的实例并不是 InstanceType 类型。它的类型与类的名字有关。也因此,对new-style类来说,上面的等式只有同一个类的两个不同实例才为真。
--WeiZhong
```

内建函数`isinstance(obj, cname)`用来测试obj对象是否是cname的实例。如果是,函数就返回True.例如:

切换行号显示

```
1 class A(object): pass
2 class B(A): pass
3 class C(object): pass
4
5 a = A()           # 'A'的实例
6 b = B()           # 'B'的实例
7 c = C()           # 'C'的实例
8
9 isinstance(a,A)   # 返回 True
10 isinstance(b,A)  # 返回 True, B 源自 A
11 isinstance(b,C)  # 返回 False, C 与 A 没有派生关系
```

同样地,内建函数`issubclass(A, B)`用来测试类A是否是类B的子类:

```
issubclass(B,A)    # 返回 True
issubclass(C,A)    # 返回 False
issubclass(A,A)    # 永远返回True
```

`isinstance()`函数也可以用于检查任意内建类型:

切换行号显示

```
1 import types
2 isinstance(3, types.IntType)    # 返回 True
3 isinstance(3, types.FloatType) # 返回 False
```

这是一个被推荐的类型检查方法,这样类型和类的差别就可以忽略.

1. 第八章 模块和包

目录

- 1. 第八章 模块和包
 - 1. 模块
 - 2. 模块搜索路径
 - 3. 模块导入和汇编
 - 4. 重新导入模块
 - 5. 包

本章的主题就是模块和包。较大的Python程序基本上都使用模块和包进行组织，Python发行版也包括方方面面许许多多的模块...

1.1. 模块

你可以使用import语句将一个源代码文件作为模块导入。例如: {#!python # file : spam.py a = 37 # 一个变量 def foo: # 一个函数

```
    print "I'm foo"
class bar: # 一个类
    def grok(self):
        print "I'm bar.grok"
```

b = bar() # 创建一个实例 } } } 使用import spam 语句就可以将这个文件作为模块导入。系统在导入模块时，要做以下三件事：

1. 为源代码文件中定义的对象创建一个名字空间，通过这个名字空间可以访问到模块中定义的函数及变量。
2. 在新创建的名字空间里执行源代码文件。
3. 创建一个名为源代码文件的对象，该对象引用模块的名字空间，这样就可以通过这个对象访问模块中的函数及变量，如：

Toggle line numbers

```
1 import spam           # 导入并运行模块 spam
2 print spam.a         # 访问模块 spam 的属性
3 spam.foo()
4 c = spam.bar()
5 ...
```

用逗号分割模块名称就可以同时导入多个模块:

```
import socket, os, regex
```

模块导入时可以使用 as 关键字来改变模块的引用对象名字:

Toggle line numbers

```
1 import os as system
2 import socket as net, thread as threads
3 system.chdir("../")
4 net.gethostname()
```

使用from语句可以将模块中的对象直接导入到当前的名字空间。from语句不创建一个

到模块名字空间的引用对象，而是把被导入模块的一个或多个对象直接放入当前的名字空间：

Toggle line numbers

```
1 from socket import gethostname
2                                     # 将gethostname放如当前名字空间
3 print gethostname()                 # 直接调用
4 socket.gethostname()               # 引发异常NameError: socket
```

from语句支持逗号分割的对象，也可以使用星号(*)代表模块中除下划线开头的对象：

Toggle line numbers

```
1 from socket import gethostname, socket
2 from socket import *               # 载入所有对象到当前名字空间
```

不过，如果一个模块如果定义有列表__all__，则from module import * 语句只能导入__all__列表中存在的对象。

```
# module: foo.py
__all__ = [ 'bar', 'spam' ]      # 定义使用 `*` 可以导入的对象
```

另外, as 也可以和 from 联合使用：

Toggle line numbers

```
1 from socket import gethostname as hostname
2 h = hostname()
```

import 语句可以在程序的任何位置使用，你可以在程序中多次导入同一个模块，但模块中的代码*仅仅*在该模块被首次导入时执行。后面的import语句只是简单的创建一个到模块名字空间的引用而已。sys.modules字典中保存着所有被导入模块的模块名到模块对象的映射。这个字典用来决定是否需要使用import语句来导入一个模块的最新拷贝。

from module import * 语句只能用于一个模块的最顶层。*特别注意*：由于存在作用域冲突，不允许在函数中使用from 语句。

每个模块都拥有 __name__ 属性，它是一个内容为模块名字的字符串。最顶层的模块名称是 __main__。命令行或是交互模式下程序都运行在__main__ 模块内部。利用__name__属性，我们可以让同一个程序在不同的场合（单独执行或被导入）具有不同的行为，象下面这样做：

```
# 检查是单独执行还是被导入
if __name__ == '__main__':
    # Yes
    statements
else:
    # No (可能被作为模块导入)
    statements
```

1.2. 模块搜索路径

导入模块时,解释器会搜索sys.path列表,这个列表中保存着一系列目录。一个典型的

sys.path 列表的值:

```
Linux:
['', '/usr/local/lib/python2.0',
 '/usr/local/lib/python2.0/plat-sunos5',
 '/usr/local/lib/python2.0/lib-tk',
 '/usr/local/lib/python2.0/lib-dynload',
 '/usr/local/lib/python2.0/site-packages']

Windows:
['', 'C:\\WINDOWS\\system32\\python24.zip', 'C:\\Documents and
Settings\\weizhong', 'C:\\Python24\\DLLs', 'C:\\Python24\\lib',
 'C:\\Python24\\lib\\plat-win', 'C:\\Python24\\lib\\lib-tk',
 'C:\\Python24\\Lib\\site-packages\\pythonwin', 'C:\\Python24',
 'C:\\Python24\\lib\\site-packages',
 'C:\\Python24\\lib\\site-packages\\win32',
 'C:\\Python24\\lib\\site-packages\\win32\\lib',
 'C:\\Python24\\lib\\site-packages\\wx-2.6-msw-unicode']
```

空字符串 代表当前目录. 要加入新的搜索路径, 只需要将这个路径加入到这个列表.

1.3. 模块导入和汇编

到现在为止, 本章介绍的模块都是包含Python源代码的文本文件. 不过模块不限于此, 可以被 import 语句导入的模块共有以下四类:

- 使用Python写的程序(.py文件)
- C或C++扩展(已编译为共享库或DLL文件)
- 包(包含多个模块)
- 内建模块(使用C编写并已链接到Python解释器内)

当查询模块 foo 时, 解释器按照 sys.path 列表中目录顺序来查找以下文件(目录也是文件的一种):

1. 定义为一个包的目录 foo
2. foo.so, foomodule.so, foomodule.sl, 或 foomodule.dll (已编译扩展)
3. foo.pyo (只在使用 -O 或 -OO 选项时)
4. foo.pyc
5. foo.py

后面马上介绍包

已编译扩展在附录B:"Extending and Embedding Python."中有详细描述.

对于.py文件, 当一个模块第一次被导入时, 它就被汇编为字节代码, 并将字节码写入一个同名的.pyc文件. 后来的导入操作会直接读取.pyc文件而不是.py文件.(除非.py文件的修改日期更新, 这种情况会重新生成.pyc文件) 在解释器使用 -O 选项时, 扩展名为.pyo的同名文件被使用. pyo文件的内容虽去掉行号, 断言, 及其他调试信息的字节码, 体积更小, 运行速度更快. 如果使用-OO选项代替-O, 则文档字符串也会在创建.pyo文件时也被忽略.

如果在sys.path提供的所有路径均查找失败, 解释器会继续在内建模块中寻找, 如果再次失败, 则引发 ImportError 异常.

.pyc和.pyo文件的汇编, 当且仅当import 语句执行时进行.

当 import 语句搜索文件时, 文件名是大小写敏感的

即使在文件系统大小写不敏感的系统上也是如此(Windows等). 这样, `import foo` 只会导入文件`foo.py`而不会是`FOO.PY`. *注意*:Python的2.1之前的版本的,这个功能在某些平台上会有问题.要写出兼容性好的程序,就避免在模块名中大小混用.

1.4. 重新导入模块

如果更新了一个已经用`import`语句导入的模块, 内建函数`reload()`可以重新导入并运行更新后的模块代码.它需要一个模块对象做为参数.例如:

Toggle line numbers

```
1 import foo
2 ... some code ...
3 reload(foo)           # 重新导入 foo
```

在`reload()`运行之后的针对模块的操作都会使用新导入代码, 不过`reload()`并不会更新使用旧模块创建的对象, 因此有可能出现新旧版本对象共存的情况. *注意* 使用C或C++编译的模块不能通过 `reload()` 函数来重新导入.

记住一个原则, 除非是在调试和开发过程中, 否则不要使用`reload()`函数.

1.5. 包

多个关系密切的模块应该组织成一个包, 以便于维护和使用. 这项技术能有效避免名字空间冲突. 创建一个名字为包名字的文件夹并在该文件夹下创建一个 `__init__.py` 文件就定义了一个包. 你可以根据需要在该文件夹下存放资源文件、已编译扩展及子包. 举例来说, 一个包可能有以下结构:

```
Graphics/
  __init__.py
  Primitive/
    __init__.py
    lines.py
    fill.py
    text.py
    ...
  Graph2d/
    __init__.py
    plot2d.py
    ...
  Graph3d/
    __init__.py
    plot3d.py
    ...
  Formats/
    __init__.py
    gif.py
    png.py
    tiff.py
    jpeg.py
```

`import`语句使用以下几种方式导入包中的模块:

* `import Graphics.Primitive.fill` 导入模块`Graphics.Primitive.fill`,只能以全名访问模块属性,例如 `Graphics.Primitive.fill.floodfill(img,x,y,color)`.

* `from Graphics.Primitive import fill`

导入模块`fill`,只能以 `fill`.属性名 这种方式访问模块属性,例如 `fill.floodfill(img,x,y,color)`.

* `from Graphics.Primitive.fill import floodfill`

导入模块`fill`,并将函数`floodfill`放入当前名称空间,直接访问被导入的属性, 例如 `floodfill(img,x,y,color)`.

无论一个包的哪个部分被导入,在文件`__init__.py`中的代码都会运行.这个文件的内容允许为空,不过通常情况下它用来存放包的初始化代码。导入过程遇到的所有 `__init__.py`

文件都被运行.因此 `import Graphics.Primitive.fill` 语句会顺序运行 `Graphics` 和 `Primitive` 文件夹下的`__init__.py`文件.

下边这个语句具有歧义:

```
from Graphics.Primitive import *
```

这个语句的原意图是想将`Graphics.Primitive`包下的所有模块导入到当前的名称空间.然而,由于不同平台间文件名规则不同(比如大小写敏感问题),`Python`不能正确判定哪些模块要被导入.这个语句只会顺序运行 `Graphics` 和 `Primitive` 文件夹下的 `__init__.py`文件.要解决这个问题,应该在`Primitive`文件夹下面的`__init__.py`中定义一个名字`all`的列表, 例如:

Toggle line numbers

```
1 # Graphics/Primitive/__init__.py
2 __all__ = ["lines","text","fill",...]
```

这样,上边的语句就可以导入列表中所有模块.

下面这个语句只会执行`Graphics`目录下的`__init__.py`文件,而不会导入任何模块:

Toggle line numbers

```
1 import Graphics
2 Graphics.Primitive.fill.floodfill(img,x,y,color) # 失败!
```

不过既然 `import Graphics` 语句会运行 `Graphics` 目录下的 `__init__.py`文件,我们就可以采取下面的手段来解决这个问题:

Toggle line numbers

```
1 # Graphics/__init__.py
2 import Primitive, Graph2d, Graph3d
3
4 # Graphics/Primitive/__init__.py
5 import lines, fill, text, ...
```

这样`import Graphics`语句就可以导入所有的子模块(只能用全名来访问这些模块的属性).

在一个包中,同一目录下的两个模块可以互相引用而不需要提供包的名字.例如 `Graphics.Primitive.fill`模块可以使用`import lines`导入`Graphics.Primitive.lines`. 不过如果两个模块位于同一个包的不同目录,就必须提供包名.例如,如果`Graphics.Graph2d`的 `plot2d`模块需要使用`Graphics.Primitive`下的`lines`模块,就必须使用`from Graphics.Primitive import lines`这样的语句.如果需要,一个模块可以通过 `__name__` 属性得到自己的全名.例如: 下面的代码在仅知道同级子包的名字情况下(不知道它们共同的顶级包名)导入该子包下的一个模块。

Toggle line numbers

```
1 # Graphics/Graph2d/plot2d.py
2
3 # 决定包的名称,以及自身的位置
4 import string
5 base_package = string.join(string.split(__name__, '.')[::-1], '.')
6
7 # 导入 ../Primitive/kill.py 模块
8 exec "from %s.Primitive import kill" % (base_package,)
```

最后,当Python导入一个包时,它定义了一个包含目录列表的特殊变量`__path__`,它用于查找包的模块(`__path__`与`sys.path`变量的作用相似). 可以在`__init__.py`文件中访问`__path__`变量.这个列表的初始值只有一个元素.即包的目录.只要你觉得必要,一个包也可以到其他的目录中去(在`__path__`增加要搜索的目录)搜索模块。(换言之,一个模块可以属于一个包,却不位于这个包所在的目录或子目录下。

PythonEssentialRef8 (2006-02-11 06:34:33由WeiZhong编辑)

1. 第九章 输入输出

本章的主题是Python的输出输出细节：命令行参数、环境变量、文件I/O、Unicode及对象持久化。

1.1. 读取参数及环境变量

当解释器启动时，命令行参数就被放入 `sys.argv` 这个列表中。列表的第一个元素是程序的名字，后面的元素是你提供的命令行参数。下面的程序展示了如何访问命令行参数：

目录

1. 第九章 输入输出
 1. 读取参数及环境变量
 2. 文件
 3. 标准输入,标准输出和标准错误
 4. `print`语句
 5. 对象持久化
 6. Unicode I/O
 1. Unicode 数据编码
 2. Unicode 字符属性

Toggle line numbers

```
1 # printopt.py
2 # 打印出所有命令行参数
3 import sys
4 for i in range(len(sys.argv)):
5     print "sys.argv[%d] = %s" % (i, sys.argv[i])
```

运行该程序，结果如下：

```
% python printopt.py foo bar -p
sys.argv[0] = printopt.py
sys.argv[1] = foo
sys.argv[2] = bar
sys.argv[3] = -p
%
```

通过访问 `os.environ` 字典可以访问环境变量，如下例：

Toggle line numbers

```
1 import os
2 path = os.environ["PATH"]
3 user = os.environ["USER"]
4 editor = os.environ["EDITOR"]
```

要更改环境变量，直接设定 `os.environ` 变量或使用 `os.putenv()` 函数。如下例：

Toggle line numbers

```
1 os.environ["FOO"] = "BAR"
2 os.putenv("FOO", "BAR")
```

1.2. 文件

内建函数 `open(name [,mode])` 打开或创建文件，就象下面这样：

Toggle line numbers

```
1 f = open('foo')           # 以读取模式打开 'foo'
2 f = open('foo','w')       # 以写模式打开 'foo'
```

文件模式 'r' 表示读，'w' 表示写，'a' 表示在文件末尾添加内容。模式字符后面允许跟一个 'b' 表示访问的是二进制数据，比如 'rb' 或 'wb'。对 UNIX(或Linux)这个 'b' 有没有无关紧要，对 Windows 平台则有积极意义。如果你很关心代码的可移植性，那就最好总是加上这个 'b'。另外，还有一种更新模式，你只要在读写模式后增加一个 '+' 就可以使用这种模式，如 'r+' 或 'w+'。当一个文件以更新模式打开，你就可以对这个文件进行读写操作。只要在任何读取操作之前刷新所有的输出缓冲就不会有问题。如果一个文件以 'w+' 模式打开，它的长度就度截为 0。

`open()` 返回一个文件对象，它支持下表中列出的方法

表 9.1. 文件方法

方法	描述
<code>f.read([n])</code>	读取至多 <code>n</code> 字节
<code>f.readline([n])</code>	读取一行中的前 <code>n</code> 字符。如果 <code>n</code> 被省略，就读取整行
<code>f.readlines()</code>	读取所有的行并返回一个包含所有行的列表
<code>f.xreadlines()</code>	返回一个迭代器，每次迭代返回文件的一个新行
<code>f.write(s)</code>	将字符串 <code>s</code> 写入文件
<code>f.writelines(l)</code>	将列表 <code>l</code> 中的所有字符串写入文件
<code>f.close()</code>	结束文件
<code>f.tell()</code>	返回当前的文件指针
<code>f.seek(offset [, where])</code>	定位到一个新的文件位置
<code>f.isatty()</code>	如果 <code>f</code> 是一个交互式终端则返回 1
<code>f.flush()</code>	刷新输出缓冲区
<code>f.truncate([size])</code>	如果文件长于 <code>size</code> 就截短它至 <code>size</code> 大小
<code>f.fileno()</code>	返回一个整型的文件描述符
<code>f.readinto(buffer ,nbytes)</code>	读取 <code>n</code> 字节数据至一个 <code>buffer</code> 对象。

除非给 `read()` 方法一个可选的长度参数，它就会读取整个文件并将文件内容作为一个字符串返回。

`readline()` 返回下一行，包含换行字符。如果在调用 `readline()` 方法时提供一个长度参数 `n`，若 `n` 大于该行长度，则返回前 `n` 个字节。该行剩下的部分并不会被丢弃，在下次读取操作时会被返回。`readlines()` 方法读取所有行，并将这些行作为一个 list 返回。`readline()` 和 `readlines()` 会自动处理换行在不同平台的表示。(众所周知的 '\n','\r','\r\n') `xreadlines()` 返回一个迭代器，允许用迭代的方式得到文件的每一行。下面是一个使用 `xreadlines()` 的例子：

Toggle line numbers

```
1 for line in f.xreadlines():
2     # Do something with line
3     ...
```

`write()` 方法将一个字符串写入文件。`writelines()` 将一个字符串列表中的所有元素顺序写入文件。以上所有操作，字符串中均可包含二进制数据。`seek(offset[,where])` 用来随机存取文件的任一部分。`offset` 是偏移量，`where` 是可选的位置参数(默认值为 0，表示文件开始位置)。

如果 `where` 的值是 1，表示当前位置。如果 `where` 是 2 表示文件结束位置。`fileno()` 返回一个打开文件的整型文件描述编号，有些模块在进行低层次 I/O 操作时会用到。在支持单个文件超过 2GB 容量的机器上，`seek()` 和 `tell()` 使用长整数。不过要允许这个特

性可能需要重新配置并重新编译Python解释器。

文件对象还有下面的数据属性：

属性 描述 `f.closed` 表示文件状态的布尔值: 0 表示文件打开, 1 表示已关闭。 `f.mode` 文件打开模式 `f.name` `open()`函数打开的文件名 否则, 它就是一个表示文件来源的字符串

`f.softspace` 这是一个布尔值 在使用 `print` 语句时表示在打印另一个值之前, 是否要先打印一个空白符。若用类来模仿文件操作则必须提供这样一个可写的属性, 并将其初始化为0。

1.3. 标准输入, 标准输出和标准错误

Python解释器提供三种标准文件对象,标准输入,标准输出,以及标准错误。(即`sys`模块中的`sys.stdin`, `sys.stdout`和 `sys.stderr`对象). `stdin`对象为解释器提供输入字符流。 `stdout`对象接收 `print` 语句产生的输出. `stderr`对象接收出错信息. 通常`stdin`被映射到用户键盘输入,而`stdout`和`stderr`产生屏幕输出.

用上一节介绍的方法就可以实现原始的用户输入/输出.下边的函数从标准输入读取一行文本, 然后返回这行文本:

Toggle line numbers

```
1 def gets():
2     text = ""
3     while 1:
4         c = sys.stdin.read(1)
5         text = text + c
6         if c == '\n': break
7     return text
```

内建函数`raw_input(prompt)`也可以从`stdin`中读取并保存内容:

Toggle line numbers

```
1 s = raw_input("type something : ")
2 print "You typed '%s'" % (s,)
```

最后要说的是, 键盘中断(通常是`Ctrl+C`)会引发`KeyboardInterrupt`异常,该异常可以被异常处理语句捕获并处理。

只要需要, `sys.stdout`、 `sys.stdin`及`sys.stderr`的值均可以使用其它文件对象进行替换。这样 `print` 语句和 `raw_input` 函数都会使用新值。在解释器启动时, `sys.stdout`, `sys.stdin`及`sys.stderr`可以分别使用`sys.stdout`, `sys.stdin`, 和 `sys.stderr`这三个名字来访问。

注意某些场合 `sys.stdout`, `sys.stdin`及`sys.stderr`的默认值会被改变(通常程序运行在一个集成环境时).例如,当在`IDLE`下运行Python代码时, `sys.stdin`会被开发环境提供的一个行为类似文件对象的对象代替.在这样的场合,低层方法如`read()`,`seek()`可能会失效。

《Python In a Nutshell》(2003)

10.7.1 标准输出及标准错误

`sys` 模块有 `stdout` 和 `stderr` 属性, 这是用于输出的两个文件对象。 除非你使用某种 `shell` 重定向, 输出内容将总是发送到执行脚本的终端上。当然现在几乎没有什么真正的终端了: 这个所谓的终端通常是一个支持文本输入输出的窗口 (比方 `windows` 下的一个 控制台 或 `unix` 下 一个 `xterm` 窗口)。

1.4. print语句

`print`语句将一个或多个对象的字符串表示输出到`stdout`对象。`print`可以用逗号分割的一系列对象:

```
print "The values are", x, y, z
```

解释器对每个对象调用`str()`函数来产生最终输出内容,然后再将这些字符串用空格连接起来,并在字符串最后添加一个换行符,最后输出到`stdout`对象.不过当`print`语句的最后有一个逗号时,就会用一个空格代替输出字符串最后的换行。

Toggle line numbers

```
1 print "The values are ", x, y, z, w
2 # 也可以使用两个print语句来打印出相同的字符
3 print "The values are ", x, y, # Omits trailing newline
4 print z, w
```

在第四章--操作符和表达式中介绍过的字符格式运算(`%`)能够实现字符串格式输出:

```
print "The values are %d %7.5f %s" % (x,y,z) # 格式化输出/输入
```

通过对`print`语句添加`>>file`修饰能够将输出内容重定向到`file`文件对象.(`file`是一个可写的文件对象):

Toggle line numbers

```
1 f = open("output","w")
2 print >>f, "hello world"
3 ...
4 f.close()
```

将格式输出与三引号字符串相结合是输出特殊文本的有效方式。假设你需要批量发送一些固定格式的短小信件,包含姓名,项目名,以及一个数字,象下面这样:

```
Dear Mr. Bush,
Please send back my blender or pay me $50.00.

Sincerely yours,

Joe Python User
```

象下面这样做就OK:

Toggle line numbers

```
1 form = ""\
2 Dear %(name)s,
3 Please send back my %(item)s or pay me $%(amount)0.2f.
4
5 Sincerely yours,
6
7 Joe Python User
8 ""
9 print form % { 'name': 'Mr. Bush',
10               'item': 'blender',
```

```
11         'amount': 50.00,  
12     }
```

在输出多行多项目文本时,该方法简单有效,并且条理清晰。

1.5. 对象持久化

将一个对象内容保存到一个文件中,当再次需要该对象时通过读取这个文件重新生成该对象是很用的。你可以写一对函数通过读取和写入特定格式数据实现该功能,不过Python提供的 `Pickle` 和 `shelve` 模块可能是更好的选择。

`Pickle` 模块的 `dump` 方法可以方便的把一个对象保存到一个文件中.例如:

```
import Pickle object = someObject() f = open(filename,'w') Pickle.dump(object, f) # 保存对象
```

之后可以用 `load` 方法重新得到该对象:

```
import Pickle f = open(filename,'r') object = Pickle.load(f) # 恢复对象
```

`shelve`模块与`Pickle`做类似的工作,不过它将对象数据保存在一个字典格式的文本数据库中:

Toggle line numbers

```
1 import shelve  
2 object = someObject()  
3 dbase = shelve.open(filename) # 打开数据库  
4 dbase['key'] = object # 将对象保存在数据库中  
5 ...  
6 object = dbase['key'] # 恢复对象  
7 dbase.close() # 关闭数据库
```

注意:只有支持序列化的对象才可以被保存在文件中。绝大多数Python对象都支持序列化。某些用于特殊目的的对象,例如用来维护系统内部状态的文件等,这样的对象是不能用这种方法来恢复的。关于`Pickle`和`shelve`模块的更多细节,参见附录A.

1.6. Unicode I/O

在系统内部, `Unicode` 字符串被表示为一个16位整数序列, `8-bit` 字符串则是一个字节序列,绝大多数字符串操作被扩展为能够处理更宽范围的字符值。只要 `Unicode` 字符串被转换为字节流,就必然会产生一系列问题(需要解决)。首先,要考虑现有软件的兼容性,对那些仅支持 `ASCII`或其它 `8-bit`的软件来说,将 `Unicode`字符串转化为 `ASCII`字符串是较好的方法。其次, `16-bit` 字符占用两个字节,字节顺序问题虽然比较无聊但必须考虑。对一个`Unicode`字符 `U+HHLL` 来说,小端法编码方案将低位字节放在前面,即 `LL HH`; 大端法编码方案则将高位字节放在前面,即 `HH LL`. 就因为这么点问题,不指定编码方案,你就无法将原始 `Unicode` 数据写入文件。

要解决这些问题,只能根据特定的编码规则将 `Unicode` 字符串进行客观表示。这些规则定义了如何将 `Unicode` 字符表示为字节序列。在第四章,针对 `unicode()`及 `s.encode()` 首先介绍了编码规则。举例来说:


Toggle line numbers

```
1 a = u"M\u00fcller"  
2 b = "Hello World"  
3 c = a.encode('utf-8') # Convert a to a UTF-8 string
```

```
4 d = unicode(b) # Convert b to a Unicode string
```

codecs 模块用类似的技术解决了 Unicode 的输入输出问题。codecs 模块拥有一系列转换函数依据不同的编码方案完成字节数据和 Unicode 字符串的转换。通过调用 codecs.lookup(encoding) 函数来选择一种编码方案。这个函数返回一个包括四个元素的 tuple (enc_func, decode_func, stream_reader, stream_writer)。举例来说:

Toggle line numbers

```
1 import codecs
2 (utf8_encode, utf8_decode, utf8_reader, utf8_writer) = 
3     codecs.lookup('utf-8')
```

enc_func (u [,errors]) 函数接受一个 Unicode 字符串 u，返回值是 tuple(s, len)。其中 s 是转码后的 8-bit 字符串(内容为 u 的一部分或全部), len 是被成功转换的 Unicode 字符数。decode_func(s [,errors]) 函数接受一个 8-bit 字符串，返回值是 tuple(u, len)。其中 u 是一个 Unicode 字符串(内容为 s 的一部分或全部), len 是被成功转换的字符数。errors 决定转化过程中的错误如何处理，它的值可能是 'strict' 或 'ignore' 或 'replace'。若是 'strict' 模式，编码错误将引发 UnicodeError 异常。若是 'ignore' 模式，编码错误将被忽略。若是 'replace' 模式，无法转换的编码将被替换为 '?' 字符 (Unicode 字符 U+FFFD 或 8-bit 字符 '?')。

stream_reader 用来对文件对象进行封装，以支持 Unicode 数据读取。调用 stream_reader (file) 返回封装后的文件对象，它的 read(), readline(), 及 readlines() 方法支持读取 Unicode 字符串数据。stream_writer 用来对文件对象进行封装，以支持将 Unicode 字符串写入文件。调用 stream_writer(file) 返回封装后的文件对象，它的 write() 和 writelines() 方法将 Unicode 字符串按给定的编码转换为字节流写入文件中。

下面的例子演示了如何使用这些方法处理 UTF-8 编码的 Unicode 数据:

Toggle line numbers

```
1 # 输出 Unicode 数据到文件
2 ustr = u'M\u00fcller' # 一个 Unicode 字符串
3
4 outf = utf8_writer(open('foo','w')) # 创建 UTF-8 字节流
5 outf.write(ustr)
6 outf.close()
7
8 # 从一个文件读取 unicode 数据
9 infile = utf8_reader(open('bar'))
10 ustr = infile.read()
11 infile.close()
```

当处理 Unicode 文件时，数据编码通常内嵌在文件本身当中。举例来说，XML 解析器根据文件的前几个字节 '<?xml ...>' 来判断文件编码。如果最初的四个值是 3C 3F 78 6D ('<?xm'), 就认为编码是 UTF-8。如果最初的四个值是 00 3C 00 3F 或 3C 00 3F 00, 就认为编码是 UTF-16 大端表示方案 或 UTF-16 小端表示方案。文档编码可能出现在 MIME 头或者做为其它文档元素的一个属性。举例来说:

```
<?xml ... encoding="ISO-8859-1" .... ?>
```

用类似下面的代码来读取文档的编码:

Toggle line numbers

```

1 f = open("somefile")
2 # Determine encoding
3 ...
4 (encoder,decoder,reader,writer) = codecs.lookup(encoding)
5 f = reader(f) # Wrap file with Unicode reader
6 data = f.read() # Read Unicode data
7 f.close()

```

1.6.1. Unicode 数据编码

表 9.2 列出了codecs模块中目前正在使用的所有编码

表 9.2. codecs 模块中的全部编码器

编码	描述
'ascii'	ASCII 编码
'latin-1', 'iso-8859-1'	Latin-1 或 ISO-8859-1 编码
'utf-8'	8-bit 变长编码
'utf-16'	16-bit 变长编码
'utf-16-le'	UTF-16, 显式小端编码方案
'utf-16-be'	UTF-16, 显式大端编码方案
'unicode-escape'	和 u"string " 格式相同
'raw-unicode-escape'	和 ur"string "格式相同

下面的段落描述了各种编码的细节:

'ascii' 编码:

'ascii' 编码, 字符值的范围被限制在[0,0x7f] 和 [U+0000, U+007F]。超出这个范围的任何字符都是非法的。

'iso-8859-1' 或 'latin-1' 编码:

字符可以是任意的 8-bit 值([0,0xff] 及 [U+0000, U+00FF]). 取值范围 [0,0x7f] 内的字符对应 ASCII 字符集, 取值范围 [0x80,0xff] 内的字符对应 ISO-8859-1 或扩展 ASCII 字符集。超出 [0,0xff] 取值范围的任何字符都会造成错误。

'utf-8' 编码:

UTF-8 是一种变长编码, 它能表示所有的Unicode字符。一个单独的字节用来表示值为 0-127 的 ASCII 字符。所有其它字符均被表示为多字节序列(双字节或3字节)。这些字节的编码见下表

Unicode 字符	Byte 0	Byte 1	Byte 2
U+0000 - U+007F	0nnnnnnn		
U+007F - U+07FF	110nnnnn	10nnnnnn	
U+0800 - U+FFFF	1110nnnn	10nnnnnn	10nnnnnn

对两字节序列, 第一个字节的前三个比特总是 110. 对三字节序列, 第一个字节的前三个比特总是 1110. 多字节序列的所有后来字节的前两个比特都是 10.

UTF-8 格式一个字符最多可以使用六个字节。Python 中, 四字节 UTF-8 序列被称为代理对, 用来对一对 Unicode 字符进行编码。这一对字符的取值都在[U+D800, U+DFFF]范围内并组合成一个 20-bit 的值. 代理对这样编码:四字节序列 111100nn

10nnnnnn 10nnmmmm 10mmmmmm 被编码成这样一对: U+D800 + N, U+DC00 + M, 其中 N 是高10位, M 是低10位。五字节和六字节 UTF-8 序列(开始位分别为 111110 和 1111110) 用来对32比特值的Unicode字符进行编码。Python目前不支持五字节和六字节UTF-8序列。如果数据流中存在这样的数据会引发 UnicodeError 异常。

UTF-8 编码对旧程序支持的相当好. 首先, 标准 ASCII 字符的编码没有发生任何改变。这意味着 UTF-8 编码的 ASCII 字符串与传统的 ASCII 字符串完全相同。其次, UTF-8 编码的多字节序列未内嵌 null 字节。这样现有的基于 C 库的软件和程序所使用的 null-结尾的 8-bit 字符串可以与 UTF-8 字符串相容. 最后, UTF-8 编码 保留了字符串的字典顺序。也就是说如果 a 和 b 是 Unicode 字符串并且 $a < b$, 则当 a 和 b 被转化为UTF-8编码后, $a < b$ 仍然成立。因此, 写给 ASCII 字符串的排序算法及其它与顺序有关的算法也一样可以工作在 UTF-8 编码上。

'utf-16', 'utf-16-be', and 'utf-16-le' 编码:

UTF-16 是一种变长16位编码, 其中 Unicode 被记录为 16-bit 值。如果未指定字节顺序, 则默认为大端法编码方案。另外, 一个特殊的字符 U+FEFF 可以用来显式的标记UTF-16 数据流的字节顺序。大端编码方案, U+FEFF 字符表示 zero-width nonbreaking space, 而 U+FFFE 则是一个非法的 Unicode 字符。因此, 编码器可以使用这个字节顺序 FE FF 或 FF FE 来判断字节顺序。当读取 Unicode 数据时,Python会自动移去这个标志。

'utf-16-be' 编码 显式指定 UTF-16 大端编码(big endian), 'utf-16-le' 显式指定 UTF-16 小端编码(little ending)。

尽管已经有多种 UTF-16 的扩展以支持更多字符, 目前的 Python 并不支持任何这样的扩展。

'unicode-escape' 及 'raw-unicode-escape' 编码:

这些编码方法被用来转换 Unicode 字符串到 Python 使用的 Unicode 字符串及原始 Unicode 字符串。举例来说:

Toggle line numbers

```
1 s = u'\u14a8\u0345\u2a34'  
2 t = s.encode('unicode-escape') #t = '\u14a8\u0345\u2a34'
```

1.6.2. Unicode 字符属性

除了实现输入输出之外, 使用 Unicode 的程序必然会有测试 Unicode 字符属性的需要 (是否大小写、是否数字、是否空白等等)。unicodedata 模块提供了这些 unicode 字符数据库。常规字符属性可以通过 unicodedata.category(c) 函数得到. 例如, unicodedata.category(u"A") 返回 'Lu', 表示这个字符是一个大写字符。更多关于 Unicode 字符数据库及 unicodedata 模块的细节, 请参阅附录A。

1. 第十章 执行环境

本章的主题是Python程序的运行环境，目标是阐述解释器的运行时行为：包括程序启动、站点配置及程序终止。

1.1. 解释器选项及运行环境

解释器有许多选项控制它的运行时行为和运行环境。在UNIX和Windows下,选项以命令行选项的形式传递给解释器:

```
python [option] ... [-c cmd | -m mod | file | -] [arg] ...
```

在Macintosh下,需要使用一个独立程序EditPythonPrefs来修改Python解释器的执行参数。

当前版本共支持以下命令行选项(Python2.4):

选项	描述
-d 或 PYTHONDEBUG=x	生成解析器调试信息
-h	打印帮助信息, 然后退出
-i 或 PYTHONINSPECT=x	程序运行后进入交互模式(程序中的对象会继续存在)
-O 或 PYTHONOPTIMIZE=x	优化生成的字节码
-OO	在-O的基础之上删除文档字符串
-S	阻止包含 site 定义的模块
-t	若发现制表符空格混合缩进给出警告信息
-tt	若发现制表符空格混合缩进引发 TabError 异常
-u 或 PYTHONUNBUFFERED=x	非缓冲二进制标准输出及标准错误
-U	Unicode模式, 所有字符都被转换为Unicode
-v 或 PYTHONVERBOSE=x	冗余模式(跟踪 import 语句).
-V	显示版本信息后退出
-x	忽略程序的第一行
-c cmd	运行字符串 cmd
-W arg	警告控制(arg是行为:message:category:module:lineno)
-E	忽略环境变量(such as PYTHONPATH)
-m mod	mod: 模块名 将模块象脚本一样运行
-Q arg	division options: -Qold (default),
-Qwarn, -Qwarnall, -Qnew	
file	脚本文件
-	从标准输入读取程序代码 (这是默认参数)
arg ...	脚本参数(保存在 sys.argv[1:])

选项 -d 用来调试解释器,普通程序员几乎不会用到这个选项(Python开发团队用的会比较多)

选项 -i 会在程序结束的时候进入交互模式,通常用于调试用户代码。

目录

1. 第十章 执行环境
 1. 解释器选项及运行环境
 2. 交互模式
 3. 运行Python程序
 4. Site配置文件
 5. 启用 Future 特性
 6. 程序终止

选项-O 和 -OO 使用最佳化模式,产生较优化的字节码(在第八章中有介绍).

选项 -S 忽略site 初始化模块(参见后边的 "Site 配置文件" 小节).

选项 -t , -tt,和-v会产生额外的警告和调试信息

选项 -x 会忽略程序的第一行(例如,当第一行是启动Python解释器的脚本时).

选项 -U 强迫解释器将所有字符转化为Unicode.选项 -W 用于过滤警告信息(参阅附录 A).

脚本名字在所有选项后出现.如果未提供文件名,或者提供了一个连字号(-)作为文件名,解释器就从标准输入读入程序.如果标准输入是一个交互的终端,一个信息条和提示符就会出现.否则解释器就打开指定文件并运行它直到文件结束.

选项 -c cmd 用于以命令行选项的形式运行一个小脚本cmd.

在程序名或hyphen()后的命令行参数会被传递给程序中的sys.argv (第九章中的 "读取参数和环境变量" 中介绍)

此外,解释器还读取以下的环境变量:

变量	描述
PYTHONPATH	冒号分隔的模块搜索路径
PYTHONSTARTUP	交互模式启动时自动运行的脚本
PYTHONHOME	Python的安装位置
PYTHONINSPECT	表示 -i 选项
PYTHONUNBUFFERED	表示 -u 选项
PYTHONCASEOK	import 语句 模块名大小写不敏感 (windows)

PYTHONPATH指定一系列模块搜索路径,它将被插入到sys.path列表的前面.

PYTHONSTARTUP指定一个脚本文件,当解释器交互方式启动时该脚本会自动运行.PYTHONHOME变量用于配置Python的安装位置,因为Python自己可以找到它的库以及包的地址,所以这个变量极少用到.如果这个变量包括一个目录,如usr/local,则解释器就试图在这个目录查找所有文件.如果包括两个目录,例如 /usr/local:/usr/local/sparc-solaris-2.6 ,解释器会在第一个目录中查找跨平台的文件,在第二个目录查找依赖平台的文件.若指定目录没有有效的Python安装,则PYTHONHOME环境变量将被忽略.

在Windows下,有些环境变量,比如PYTHONPATH等等,是从注册表的 HKEY_LOCAL_MACHINE/Software/Python 中读入的.在Macintosh下,使用 EditPythonPrefs程序可以改变这些环境变量.

1.2. 交互模式

在命令行下运行Python而不提供执行脚本,就自动进入Python的交互模式. Python会首先显示版本信息,如果PYTHONSTARTUP环境变量设置了有效的脚本, Python就会运行这个脚本,最后显示交互提示符 >>>.这个脚本会被当成用户输入程序的一部分被执行.(也就是说,它并不是以import语句导入的方式被执行的)该脚本的一个典型应用就是读取用户的配置文件(比如.pythonrc).

当接受一个交互输入时,会有两种用户提示. >>>提示符你输入新的语句, ...提示输入你正处于一个缩进块.例如:

```
Python 2.4.2 (#67, Sep 28 2005, 12:41:11) [MSC v.1310 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> for i in range(0,4):
```

```
...     print i
...
0
1
2
3
>>>
```

如果需要, 修改`sys.ps1`和`sys.ps2`这两个变量你就可以自定义这两个提示。

某些系统里, Python可能被编译为使用GNU `readline`库。如果该特性启用的话, 该库就会提供命令历史, 自动完成等其它特性。特殊关键字绑定由`readline`库提供(参见附录A `readline`模块)。

默认情况下, 如果你在交互模式输入一个表达式, 则解释器会调用 `print repr`(你的表达式)来生成输出结果。从Python 2.1开始, 你可以通过设置变量`sys.displayhook`来改变表达式的输出格式。例如:

```
>>> def my_display(x):
...     print "result = %s" % repr(x)
...
>>> sys.displayhook = my_display
>>> 3+4
result = 7
>>>
```

1.3. 运行Python程序

在多数情况下, 你希望程序能自动启动解释器, 而不是每次手动启动解释器。在UNIX下, 你可以在程序的第一行使用魔术字符串告诉系统由哪个解释器来运行这个脚本:

```
Toggle line numbers
1 #!/usr/local/bin/python
2 # Python code from this point on...
3 import string
4 print "Hello world"
5 ...
```

在Windows下, 双击一个`.py`, `.pyw`, `.wpy`, `.pyc` 或者`.pyo`文件都会自动运行解释器。除`.pyw` 后缀文件外(静默运行), 其他的文件都会在一个控制台窗口中运行。如果必须给解释器提供启动选项, Python程序也可以由`.bat`文件启动运行。

在Macintosh下, 点击一个`.py`文件通常会打开创建该脚本的编辑器。不过Macintosh发行版有两个特殊的程序用来创建应用程序。将一个`.py`拖到 `BuildApplet` 程序就会将该脚本转换为一个应用程序(打开该文件即自动调用解释器)。 `BuildApplication` 程序则可以转换一个Python脚本成为一个独立运行的应用程序(不需要Python解释器就可以运行的应用程序)。

1.4. Site配置文件

一个典型的Python安装可能会包括很多第三方模块和包, 要配置这些包, 解释器首先导入`site`模块. `site`模块的任务就是搜索包文件, 必要时向`sys.path`添加搜索目录。另外,

site模块也设置Unicode字符串转换的默认编码。更多细节参阅附录A -- site模块。

1.5. 启用 Future 特性

从Python 2.1开始, 当一个新的语言特性首次出现在发行版中时, 如果该特性与旧版Python不兼容, 则该特性将被默认禁用。要启用这些特性, 使用语句

`from __future__ import *`。举例来说:

```
# Enable nested scopes in Python 2.1
from __future__ import nested_scopes
```

如果使用这个语句, 则该语句必须是模块或程序的第一个语句。此外, `__future__` 模块中存在的特性最终将成为Python语言标准的一部分。到那时, 将不再需要使用`__future__`模块。

1.6. 程序终止

当一个程序正常运行到最后一条语句, 或者出现一个未捕获的`SystemExit`异常(由`sys.exit()`产生), 或者解释器收到一个`SIGTERM`或者`SIGHUP`(在UNIX下)信号时, 程序就会终止。解释器会将所有已知名称空间下的所有对象对象的引用记数清为0(并同时删除所有的名称空间)。当一个对象的引用记数变为零, 就会自动调用它的`__del__()`

来销毁该对象。*注意* 当两个对象存在互相引用时, 在程序结束时, 这两个对象就无法被销毁(这会造成内存泄漏)。尽管Python的垃圾回收机制能在运行时删除这些对象, 在程序结束时该机制不会被自动调用。

由于不能保证对象的`__del__()`方法在程序结束时一定会执行, 一个比较好的办法就是在程序结束时显式的清除某些对象。例如打开的文件以及网络连接等。你可以给一个自定义对象写一个专门的销毁方法(例如`close()`), 也可以写一个终止函数, 并通过 `atexit` 模块将其注册到系统中:

Toggle line numbers

```
1 import atexit
2 connection = open_connection("deaddot.com")
3
4 def cleanup():
5     print "Going away..."
6     close_connection(connection)
7
8 atexit.register(cleanup)
```

也可以以这种方式来调用垃圾回收器:

Toggle line numbers

```
1 import atexit, gc
2 atexit.register(gc.collect)
```

当程序结束时, 有些对象的 `__del__` 方法会访问全局数据或者其他模块中的方法定义。由于这些对象可能已经被删除, `__del__` 方法就有可能引发`NameError`异常。你有可能见到类似下边这样的出错信息:

```
Exception exceptions.NameError: 'c' in <method Bar.__del__ of Bar
instance at c0310>
...
```

如果看到这个信息,说明某个对象的 `__del__` 方法执行失败,这通常意味着有一项重要操作没有完成(例如关闭一个服务器连接)。最好在代码中显式的执行清理操作,而不是依赖解释器来自动做这件事。通过在 `__del__()` 定义时使用默认参数能够避免这个罕见的 `NameError` 异常,例如:

Toggle line numbers

```
1 import foo
2 class Bar:
3     def __del__(self, foo=foo):
4         foo.bar()           # 在模块foo中使用某些东西
```

有时(罕见)必须立刻终止程序,不需要做任何清理操作。这时调用 `os._exit(status)` 即可。这个函数提供一个低层次 `exit()` 系统调用接口,当调用它时,程序会立即停止。

PythonEssentialRef10 (2006-02-12 12:55:49由WeiZhong编辑)