# Linux 系统编程-python核心编程

传智播客-邢文鹏[1]

2015-06-06

[1]http://www.itcast.cn

# 前言

## 学习目标

成为全栈式系统程序员

## 学习态度

* 谦虚
* 严谨
* 勤思
* 善问

## 学习方法

只听不练肯定学不会Linux，每个知识点都需要去动手实践才能快速掌握python编程

# 目录

# 第 1 章

# python初体验

## 1.1 what

## 1.2 why

最大的优势在于它的字符串模式匹配能力，其提供一个十分强大的正则表达式匹配引擎。核心实现依赖perl，但语法比perl易懂的多。

## 1.3 how

> 高级语言,面向对象,可拓展,可移植,语法清晰,易维护,高效的原型

## 1.4 python家族

C语言实现，CPython，扩展可用C/C++
Java实现，Jython，扩展可用Java
.Net实现，IronPython，扩展可用C#

### 1.4.1 python开发环境

Mac/Linux发行版目前默认安装python
ipython
python官方IDE，在python发行版自带
Eclipse+pydev
PyScripter

## 1.5 python能干什么？

### 1.5.1 图形化开发

### 1.5.2 系统脚本

### 1.5.3 web服务器

### 1.5.4 网络爬虫

### 1.5.5 服务器集群自动化运维

# 第 2 章

# 核心语法

## 2.1 变量和函数

### 2.1.1 变量本质

1.python中的变量不需要先定义，再使用，可以直接使用,还有重新使用用以存储不同类型的值。

2.变量命名遵循C命名风格。

3.大小写敏感。

4.变量引用计数。

5.del语句可以直接释放资源，变量名删除，引用计数减1。

6.变量内存自动管理回收,垃圾收集。

7.指定编码在文件开头加入 # -- coding: UTF-8 -- 或者 #coding=utf-8。

```
xingwenpeng@ubuntu:~$ python3
Python 3.4.3 (default, Apr  3 2015, 13:00:43)
[GCC 4.6.3] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> a = 12               #无需定义，直接使用，python解释器根据右值决定左侧类型
>>> print(a)
12
>>> id(a)                #变量a在内存中的编号
136776784
>>> type(a)              #a的类型为int类型
<class 'int'>
>>> b = 12.34
>>> print(b)
12.34
>>> id(b)                #变量b在内存中所占内存编号
3071447616
>>> type(b)
<class 'float'>          #b的类型为float
>>> a = "itcast"         #变量a从新指向一个字符串
>>> print(a)
itcast
>>> id(a)                #变量a在内存中的编号为保存“itcast”地方，原来a所指向的内存编号里内容并没有立即释放
3071127936
```

```
>>> type(a)              #变量a现在指向一个字符串
<class 'str'>
>>> c = b
>>> print(c)
12.34
>>> id(c)                #变量c保存的内存中的编号和b一致
3071447616
>>> type(c)
<class 'float'>

>>> b = 12               #解释器在内存中发现有保存12的这个单元，于是变量b指向了此单元，减少了存储空间
的反复申请与释放
>>> id(b)
136776784
>>> type(b)
<class 'int'>
>>> print(b)
12

>>> print(a)
itcast
>>> del(a)
>>> print(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

### 2.1.2　基本数据类型

```
获取变量的数据类型  type(var_name)

有符号整数(长整型)，浮点数，布尔值，字符串,复数
主提示符                >>>  在等待下一条语句
次提示符                ...          在等待当前语句的其他部分
```

### 2.1.3　输入/输出函数

```
输入和raw_input()内置函数
raw_input() 从标准输入获取数据  返回的数据是字符串类型
        需要使用int()进行转换
输出
print函数
        >>> print "%s is %dsthello world "%("tody",1)
        tody is 1sthello world
>> 重定向操作符
        logfile = open('/tmp/mylog.txt','a')
        print >>logfile,'error'
        logfile.close()
```

### 2.1.4 简单函数

```
函数定义格式

def add(x, y):
    z = x + y
    return z

res = add(3, 5)
print(res)
8

1.def定义函数的关键字
2.x和y为形参，不需要类型修饰
3.函数定义行需跟':'
4.函数体整体缩进
5.函数可以拥有返回值，若无返回值，返回None，相当于C中的NULL
```

### 2.1.5 局部变量和全局变量

代码1. 局部变量作用域覆盖全局变量

```
def p_num():
    num=5
    print(num)


num=10
p_num()
print(num)
#结果:  5 10
```

代码2. 函数内有局部变量定义，解释器不使用全局变量，局部变量的定义晚于被引用，报错

```
def p_num():
    print(num)
    num=5
    print(num)


num=10
p_num()
print(num)
# 结果出错
```

代码3. 函数内部可以直接访问全局变量

```
def p_num():
    print(num)


num=10
p_num()
print(num)
# 结果: 10  10
```

代码4. 函数内修改全局变量,使用global关键字

```
def p_num():
global num
print(num)
num = 20
print(num)


num=10
p_num()
print(num)
```

### 2.1.6 特殊变量

```
_xxx    from module import *无法导入
__xxx__   系统定义的变量
__xxx   类的本地变量

变量名没有类型，但对象有
```

## 2.2 表达式

```
由操作符符和操作数以求得有意义的值的组合.
```

### 2.2.1 算术表达式

常见的算术表达式由加减乘除，取模取余，取负以及幂次方(**)等运算符组成,支持int类型和float类型。

```
+a      结果符号不变
-a      对结果符号取负
```

```
a +  b   a加b
a -  b   a减b
a ** b   a的b次幂
a *  b   a乘以b
a /  b   a除以b，真正除，浮点数保留小数
a // b   a除以b，向下取整
a %  b   a对b取余数
```

### 2.2.2 逻辑表达式

```
not a        a的逻辑非           bool
a and b      a和b逻辑与          bool
a or b       a和b逻辑或          bool
a is b       a和b是同一个对象    bool
a is not b   a和b不是同一个对象  bool
```

### 2.2.3 关系表达式

```
==        等于 - 比较对象是否相等        (a == b) 返回 False。
!=        不等于 - 比较两个对象是否不相等        (a != b) 返回 true.
<>        不等于 - 比较两个对象是否不相等        (a <> b) 返回 true。这个运算符类似 != 。
>        大于 - 返回x是否大于y        (a > b) 返回 False。
<        小于 - 返回x是否小于y。
>=        大于等于 - 返回x是否大于等于y。        (a >= b) 返回 False。
<=        小于等于 - 返回x是否小于等于y。        (a <= b) 返回 true。
```

### 2.2.4 位运算

```
~a     按位取反
a << n     a左移n位
a >> n     a右移n位
a &  b     a和b按位与
a |  b     a和b按位或
a ^  b     a和b按位异或
```

### 2.2.5 Python成员运算符

```
除了以上的一些运算符之外，Python还支持成员运算符，测试实例中包含了一系列的成员，包括字符串，列表或元组。
运算符        描述        实例
in                x 在 y序列中 ，如果x在y序列中返回True。
not in        x 不在 y序列中 ，如果x不在y序列中返回True。
```

## 2.3 循环控制语句

python里的控制语句和循环语句和C中的非常相似，毕竟python是用C实现的。

注意语句后面的’:’不要丢掉，这一点C/C++程序员刚上手python的时候是最容易犯的错误。

### 2.3.1 if-else

```
if a > b:
    print("aaa")
else:
    print("bbb")
```

### 2.3.2 if-elif-else

```
if a > b:
    print("a>b")
elif a == b:
    print("a==b")
else:
    print("a<b")
```

### 2.3.3 while

```
while 判断条件：
    执行语句
```

例如：

```
var = 1
while var == 1 :  # 该条件永远为true，循环将无限执行下去
   num = raw_input("Enter a number  :")
   print("You entered: ", num)

print("Good bye!")
```

### 2.3.4 for

Python for循环可以遍历任何序列的项目，如一个列表或者一个字符串。

```
for iterating_var in sequence:
    执行语句
```

例如：

```
for letter in 'Python':     # First Example
   print('Current Letter :', letter)

fruits = ['banana', 'apple',  'mango']
for fruit in fruits:        # Second Example
   print('Current fruit :', fruit)
```

在 python 中，for … else 表示这样的意思，for 中的语句和普通的没有区别，else 中的语句会在循环正常执行完（即 for 不是通过 break 跳出而中断的）的情况下执行，while … else 也是一样。

```
count = 0
while count < 5:
   print(count, " is  less than 5")
   count = count + 1
else:
   print(count, " is not less than 5")
```

以上实例输出结果为：

```
0 is less than 5
1 is less than 5
2 is less than 5
3 is less than 5
4 is less than 5
5 is not less than 5
```

### 2.3.5 break

Python break语句，就像在C语言中，打破了最小封闭for或while循环。

break语句用来终止循环语句，即循环条件没有False条件或者序列还没被完全递归完，也会停止执行循环语句。

break语句用在while和for循环中。

如果您使用嵌套循环，break语句将停止执行最深层的循环，并开始执行下一行代码。

### 2.3.6 continue

Python continue 语句跳出本次循环，而break跳出整个循环。

continue 语句用来告诉Python跳过当前循环的剩余语句，然后继续进行下一轮循环。

continue语句用在while和for循环中。 ## 内置range()函数的用法 range(start,end,step) >>> range(1,5) #代表从1到5(不包含5) [1, 2, 3, 4] >>> range(1,5,2) #代表从1到5，间隔2(不包含5) [1, 3] >>> range(5) #代表从0到5(不包含5) [0, 1, 2, 3, 4] >>>for eachNum in [0,1,2]: … print eachNum >>>for eachNum in range(3): … print eachNum

>>>mystr = ‘abc’ >>>for c in mystr: … print c range()函数还经常和len()函数一起用于字符串索引 ## 数字 Python 数字数据类型用于存储数值。 数据类型是不允许改变的,这就意味着如果改变数字数据类型得值，将重新分配内存空间. 以下实例在变量赋值时数字对象将被创建： ### 数字变量的创建和销毁 var1 = 1 var2 = 10 您也可以使用del语句删除一些数字对象引用。

```
del语句的语法是：
del var1[,var2[,var3[....,varN]]]]

在del之后就不能再引用变量
Python 支持四种不同的数值类型：
```

## 2.3.7 数字的数据类型

```
整型(Int) - 通常被称为是整型或整数，是正或负整数，不带小数点。
长整型(long integers)
             - 无限大小的整数，整数最后是一个大写(或小写)的L。
浮点型(floating point real values)
           - 浮点型由整数部分与小数部分组成，浮点型也可以使用科学计数法表示（2.5e2 = 2.5 x 102 = 250）
复数( (complex numbers))
                 - 复数的虚部以字母J 或 j结尾 。如：2+3i
```

## 2.3.8 Python数字类型转换

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

## 2.3.9 Python数学函数

```
函数                        返回值 ( 描述 )
abs(x)                                  返回数字的绝对值，如abs(-10) 返回 10
ceil(x)                                  返回数字的上入整数，如math.ceil(4.1) 返回 5
```

```
cmp(x, y)                                   如果 x < y 返回 -1, 如果 x == y 返回 0, 如果 x > y 返回 1
exp(x)                                      返回e的x次幂(ex),如math.exp(1) 返回2.718281828459045
fabs(x)                                     返回数字的绝对值，如math.fabs(-10) 返回10.0
floor(x)                                    返回数字的下舍整数，如math.floor(4.9)返回 4
log(x)                                      如math.log(math.e)返回1.0,math.log(100,10)返回2.0
log10(x)                                    返回以10为基数的x的对数，如math.log10(100)返回 2.0
max(x1, x2,...)                             返回给定参数的最大值，参数可以为序列。
min(x1, x2,...)                             返回给定参数的最小值，参数可以为序列。
modf(x)                                     返回x的整数部分与小数部分，两部分的数值符号与x相同，整
数部分以浮点型表示。
pow(x, y)                                   x**y 运算后的值。
round(x [,n])                               返回浮点数x的四舍五入值，如给出n值，则代表舍入到小数点后的位数。
sqrt(x)                                     返回数字x的平方根，数字可以为负数，返回类型为实数，如
math.sqrt(4)返回 2+0j
```

## 2.3.10 Python随机数函数

随机数可以用于数学，游戏，安全等领域中，还经常被嵌入到算法中，用以提高算法效率，并提高程序的安全性。
Python包含以下常用随机数函数：

```
函数        描述
choice(seq)         从序列的元素中随机挑选一个元素，比如random.choice(range(10))，从0到9中随机挑选一
个整数。
randrange ([start,] stop [,step])       从指定范围内，按指定基数递增的集合中获取一个随机数，基数缺
省值为1
random()                                随机生成下一个实数，它在[0,1)范围内。
seed([x])                               改变随机数生成器的种子seed。如果你不了解其原理，你不必特别去设定
seed，Python会帮你选择seed。
shuffle(lst)                            将序列的所有元素随机排序
uniform(x, y)                           随机生成下一个实数，它在[x,y]范围内。
```

## 2.3.11 Python三角函数

Python包括以下三角函数：

```
函数        描述
acos(x)                                     返回x的反余弦弧度值。
asin(x)                                     返回x的反正弦弧度值。
atan(x)                                     返回x的反正切弧度值。
atan2(y, x)                                 返回给定的 X 及 Y 坐标值的反正切值。
cos(x)                                      返回x的弧度的余弦值。
hypot(x, y)                                 返回欧几里德范数 sqrt(x*x + y*y)。
sin(x)                                      返回的x弧度的正弦值。
tan(x)                                      返回x弧度的正切值。
degrees(x)                                  将弧度转换为角度,如degrees(math.pi/2) ,  返回90.0
radians(x)                                  将角度转换为弧度
```

## 2.4 字符串

用引号括起来的字符集合称之为字符串.引号可以是一对单引号,双引号,三引号(单双).
字符串是 Python 中最常用的数据类型。我们可以使用引号来创建字符串。

创建字符串很简单，只要为变量分配一个值即可。例如：

```
var1 = 'Hello World!'
var2 = "Python Programming"
```

但是如果需要输出 Hello "dear"怎么办呢?

```
        print "hello \"dear\""        #利用\的转义意义
```

还可以用三引号 print '''***''' 或者 """*** """

```
        print '''hello "dear"'''
```

### 2.4.1 Python三引号（triple quotes）

python中三引号可以将复杂的字符串进行复制:

python三引号允许一个字符串跨多行，字符串中可以包含换行符、制表符以及其他特殊字符。

三引号的语法是一对连续的单引号或者双引号（通常都是成对的用）。

```
 >>> hi = '''hi
there'''
>>> hi   # repr()
'hi\nthere'
>>> print hi  # str()
hi
there
```

总结: repr()表达式 和str()意义相近 但是结果不一样

### 2.4.2 Python访问字符串中的值

Python不支持单字符类型，单字符也在Python也是作为一个字符串使用。
Python访问子字符串，可以使用方括号来截取字符串----即前面提到的分片操作

```
var1 = 'Hello World!'
var2 = "Python Programming"
print "var1[0]: ", var1[0]
print "var2[1:5]: ", var2[1:5]
```

以上实例执行结果：

```
var1[0]:  H
var2[1:5]:  ytho
```

### 2.4.3 Python字符串更新

```
你可以对已存在的字符串进行修改，并赋值给另一个变量.
var1 = 'Hello World!'
print "Updated String :- ", var1[:6] + 'Python'
Updated String :-  Hello Python
```

### 2.4.4 Python字符串运算符

```
下表实例变量a值为字符串"Hello"，b变量值为"Python"：
操作符        描述         实例
        +         字符串连接         a + b 输出结果： HelloPython
        *         重复输出字符串         a*2 输出结果：HelloHello
[]                通过索引获取字符串中字符         a[1] 输出结果 e
[ : ]         截取字符串中的一部分         a[1:4] 输出结果 ell
in                成员运算符 - 如果字符串中包含给定的字符返回 True         H in a 输出结果 1
not in        成员运算符 - 如果字符串中不包含给定的字符返回 True         M not in a 输出结果 1
r/R                原始字符串 - 原始字符串：所有的字符串都是直接按照字面的意思来使用，没有转义特殊或
不能打印的字符。
               原始字符串除在字符串的第一个引号前加上字母"r"（可以大小写）以外，与普通字符串有着几乎
完全相同的语法。
               print r'\n' prints \n 和 print R'\n' prints \n
```

### 2.4.5 Python字符串格式化

```
Python 支持格式化字符串的输出 。
在 Python 中，字符串格式化使用与 C 中 sprintf 函数一样的语法。
print "My name is %s and weight is %d kg!" % ('Zara', 21)
```

### 2.4.6 python的字符串内建函数

```
常用函数:
string.find(str, beg=0, end=len(string))检测 str 是否包含在 string 中，如果是返回开始的索引值，否则
返回-1
示例代码:

string.index(str, beg=0, end=len(string)) 跟find()方法一样，只不过如果str不在 string中会报一个异常.

string.count(str, beg=0, end=len(string))返回 str在beg和end之间 在 string 里面出现的次数

string.decode(encoding='UTF-8', errors='strict') 以 encoding 指定的编码格式解码 string,
如果出错默认报一个 ValueError 的 异 常 ， 除 非 errors 指 定 的 是 'ignore' 或 者'replace'

string.encode(encoding='UTF-8', errors='strict')         以 encoding 指定的编码格式编码 string,
```

```
如果出错默认报一个ValueError 的异常，除非 errors 指定的是'ignore'或者'replace'

string.replace(str1, str2,  num=string.count(str1))
把 string 中的 str1 替换成 str2,如果 num 指定，则替换不超过 num 次.

string.split(str="", num=string.count(str))        以 str 为分隔符切片 string,
                                如果 num有指定值，则仅分隔 num 个子字符串

其他的函数:
string.capitalize()                把字符串的第一个字符大写

string.center(width)    返回一个原字符串居中,并使用空格填充至长度 width 的新字符串

string.endswith(obj, beg=0, end=len(string))检查字符串(beg,end)是否以obj结束，如果是返回True,否则返回 False.

string.expandtabs(tabsize=8)       把字符串 string 中的 tab 符号转为空格，默认的空格数 tabsize 是 8.

string.isalnum()  如果 string 至少有一个字符并且所有字符都是字母或数字则返回 True,否则返回 False

string.isalpha()  如果 string 至少有一个字符并且所有字符都是字母则返回 True,否则返回 False

string.isdecimal()如果 string 只包含十进制数字则返回 True 否则返回 False.

string.isdigit()  如果 string 只包含数字则返回 True 否则返回 False.

string.islower()  如果 string 中包含至少一个区分大小写的字符,
                                并且所有这些(区分大小写的)字符都是小写，则返回 True，否则返回 False
string.isnumeric()        如果 string 中只包含数字字符，则返回 True，否则返回 False

string.isspace()    如果 string 中只包含空格，则返回 True，否则返回 False.

string.istitle()           如果 string 是标题化的(见 title())则返回 True，否则返回 False

string.isupper()    如果 string 中包含至少一个区分大小写的字符并且所有这些(区分大小写的)字符都是大写,
                                        则返回 True，否则返回 False

string.join(seq)         Merges (concatenates)以 string 作为分隔符,
                                        将 seq 中所有的元素(的字符串表示)合并为一个新的字符串

string.ljust(width) 返回一个原字符串左对齐,并使用空格填充至长度 width 的新字符串

string.lower()               转换 string 中所有大写字符为小写.

string.lstrip()               截掉 string 左边的空格

string.rfind(str, beg=0,end=len(string) )        类似于 find()函数，不过是从右边开始查找.

string.rindex( str, beg=0,end=len(string))        类似于 index()，不过是从右边开始.

string.rjust(width)        返回一个原字符串右对齐,并使用空格填充至长度 width 的新字符串

string.rpartition(str)        类似于 partition()函数,不过是从右边开始查找.

string.rstrip()        删除 string 字符串末尾的空格.
```

```
string.splitlines(num=string.count('\n'))  按照行分隔，返回一个包含各行作为元素的列表，
                                如果 num 指定则仅切片 num 个行.

string.startswith(obj, beg=0,end=len(string))        检查字符串是否是以 obj 开头，
                               是则返回 True，否则返回 False。如果beg 和 end 指定值，则在指定范围内检
查.

string.title()                          返回"标题化"的 string,就是说所有单词都是以大写开始，其余字母均
为小写(见 istitle())

string.translate(str, del="")  根据 str 给出的表(包含 256 个字符)转换 string 的字符，
                                                  要过滤掉的字符放到 del 参数中
string.upper()                          转换 string 中的小写字母为大写

string.zfill(width)                返回长度为 width 的字符串，原字符串 string 右对齐，前面填充0

string.isdecimal()                  isdecimal()方法检查字符串是否只包含十进制字符。这种方法只存在于
unicode对象。
```

## 2.5 列表List

```
序列都可以进行的操作包括索引，切片，加，乘，检查成员。
序列中的每个元素都分配一个数字 - 它的位置，或索引，第一个索引是0，第二个索引是1，依此类推。

列表和元组二者均能保存任意类型的python对象,索引访问元素从开始
列表元素用[]包括,元素个数,值都可以改变
元组元素用()包括  通过切片 [] [:] 得到子集,此操作同于字符串相关操作
        切片使用的基本样式[下限:上限:步长]
```

### 2.5.1 访问列表中的值

```
        >>>aList = [1,2,3,4]
        >>>aList
        [1,2,3,4]
        >>>aList[0]
        1
        >>>aList[2:]
        [3,4]
        >>>aList[:3]
        [1,2,3]
```

### 2.5.2 更新列表中的值

```
        >>>aList[1] = 5
```

```
>>>aList
[1,5,3,4]
```

### 2.5.3 删除列表中的值

```
>>> del aList[1]
>>>aList
[1,3,4]
```

### 2.5.4 Python列表脚本操作符

列表对 + 和 * 的操作符与字符串相似。+ 号用于组合列表，* 号用于重复列表。

| Python表达式 | 结果 | 描述 |
| --- | --- | --- |
| len([1, 2, 3]) | 3 | 长度 |
| [1, 2, 3] + [4, 5, 6] | [1, 2, 3, 4, 5, 6] | 组合 |
| ['Hi!'] * 4 | ['Hi!', 'Hi!', 'Hi!', 'Hi!'] | 重复 |
| 3 in [1, 2, 3] | True | 元素是否存在于列表中 |
| for x in [1, 2, 3]: print x, | 1 2 3 | 迭代 |

### 2.5.5 Python列表截取

Python的列表截取与字符串操作类型，如下所示：

```
L = ['spam', 'Spam', 'SPAM!']
```

操作：

| Python 表达式 | 结果 | 描述 |
| --- | --- | --- |
| L[2] | 'SPAM!' | 读取列表中第三个元素 |
| L[-2] | 'Spam' | 读取列表中倒数第二个元素 |
| L[1:] | ['Spam', 'SPAM!'] | 从第二个元素开始截取列表 |

### 2.5.6 Python列表函数&方法

Python包含以下函数：

| 序号 | 函数 | |
| --- | --- | --- |
| 1 | cmp(list1, list2) | 比较两个列表的元素 |
| 2 | len(list) | 列表元素个数 |
| 3 | max(list) | 返回列表元素最大值 |
| 4 | min(list) | 返回列表元素最小值 |
| 5 | list(seq) | 将元组转换为列表 |

Python包含以下方法:

| 序号 | 方法 | |
|---|---|---|
| 1 | list.append(obj) | 在列表末尾添加新的对象 |
| 2 | list.count(obj) | 统计某个元素在列表中出现的次数 |
| 3 | list.extend(seq) | 在列表末尾一次性追加另一个序列中的多个值（用新列表扩展原来的列表） |
| 4 | list.index(obj) | 从列表中找出某个值第一个匹配项的索引位置 |
| 5 | list.insert(index, obj) | 将对象插入列表 |
| 6 | list.pop(obj=list[-1]) | 移除列表中的一个元素（默认最后一个元素），并且返回该元素的值 |
| 7 | list.remove(obj) | 移除列表中某个值的第一个匹配项 |
| 8 | list.reverse() | 反向列表中元素 |
| 9 | list.sort([func]) | 对原列表进行排序 |

## 2.6 元组Tuple

Python的元组与列表类似，不同之处在于元组的元素不能修改。也可进行分片 和 连接操作.
元组使用小括号，列表使用方括号。

```
>>>aTuple = ('et',77,99.9)
>>>aTuple
('et',77,99.9)
```

### 2.6.1 访问元组

```
>>>aTuple[2]
99
```

### 2.6.2 修改元组

```
>>>aTuple[1] = 5  #真的不能修改呀
报错啦
>>>tup2 = (1, 2, 3, 4, 5, 6, 7 )
>>>print "tup2[1:5]: ", tup2[1:5]
>>>tup2[1:5]:  (2, 3, 4, 5)

>>>tup3 = tup2 + aTuple;
>>>print tup3
(1, 2, 3, 4, 5, 6, 7,'et',77,99.9)
```

### 2.6.3 删除元素

元组中的元素值是不允许删除的，但我们可以使用del语句来删除整个元组

## 2.6.4 元组运算符

与字符串一样，元组之间可以使用 + 号和 * 号进行运算。这就意味着他们可以组合和复制，运算后会生成一个新的元组。

```
Python 表达式        结果        描述
len((1, 2, 3))        3        计算元素个数
(1, 2, 3) + (4, 5, 6)        (1, 2, 3, 4, 5, 6)        连接
['Hi!'] * 4        ['Hi!', 'Hi!', 'Hi!', 'Hi!']        复制
3 in (1, 2, 3)        True        元素是否存在
for x in (1, 2, 3): print x,        1 2 3        迭代
```

## 2.6.5 元组索引，截取

因为元组也是一个序列，所以我们可以访问元组中的指定位置的元素，也可以截取索引中的一段元素.

```
L = ('spam', 'Spam', 'SPAM!')
Python 表达式        结果        描述
L[2]        'SPAM!'        读取第三个元素
L[-2]        'Spam'        反向读取；读取倒数第二个元素
L[1:]        ('Spam', 'SPAM!')        截取元素
```

## 2.6.6 无关闭分隔符

任意无符号的对象，以逗号隔开，默认为元组，如下实例：

```
print 'abc', -4.24e93, 18+6.6j, 'xyz';
x, y = 1, 2;
print "Value of x , y : ", x,y;
以上实例允许结果：

abc -4.24e+93 (18+6.6j) xyz
Value of x , y : 1 2
```

## 2.6.7 元组内置函数

```
Python元组包含了以下内置函数
        序号        方法及描述
        1        cmp(tuple1, tuple2)        比较两个元组元素。
        2        len(tuple)        计算元组元素个数。
        3        max(tuple)        返回元组中元素最大值。
        4        min(tuple)        返回元组中元素最小值。
        5        tuple(seq)        将列表转换为元组。
```

### 2.6.8 多维元祖访问的示例

```
>>> tuple1 = [(2,3),(4,5)]
>>> tuple1[0]
(2, 3)
>>> tuple1[0][0]
2
>>> tuple1[0][2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
>>> tuple1[0][1]
3
>>> tuple1[2][2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> tuple2 = tuple1+[(3)]
>>> tuple2
[(2, 3), (4, 5), 3]
>>> tuple2[2]
3
>>> tuple2[2][0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not subscriptable
```

## 2.7 字典Dictionary

```
字典是python中的映射数据类型,,由键值(key-value)构成,类似于关联数组或哈希表.
key一般以数字和字符串居多,值则可以是任意类型的python对象，如其他容器模型。
字典元素用大括号        {} 包括.比如:
>>>dict = {'Alice': '2341', 'Beth': '9102', 'Cecil': '3258'}
>>>aDict = {'host':'noname'}
每个键与值用冒号隔开（:），每对用逗号分割，整体放在花括号中（{}）。
键key必须独一无二，但值则不必。值value可以取任何数据类型，但必须是不可变的
```

### 2.7.1 访问字典里的值

```
>>>aDict['host']
'noname'
>>> for key in aDict
... print key,aDict[key]
...
host noname
port 80
```

```
>>>aDict.keys()
['host','port']
>>>aDict.values()
```

### 2.7.2 修改字典

```
向字典添加新内容的方法是增加新的键/值对，修改或删除已有键/值对.
>>>aDict['port'] = 80    #如果有port key,值将会被更新 否则会被新建一个port key
>>>aDict
        {'host':'noname','port':80}
```

### 2.7.3 删除字典元素

```
能删单一的元素也能清空字典，清空只需一项操作.
显示删除一个字典用del命令
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};
del dict['Name']; # 删除键是'Name'的条目
dict.clear();     # 清空词典所有条目
del dict ;        # 删除词典
```

### 2.7.4 字典键(key)的特性

```
字典值可以没有限制地取任何python对象，既可以是标准的对象，也可以是用户定义的，但键不行。
两个重要的点需要记住：
1)        不允许同一个键出现两次。创建时如果同一个键被赋值两次，后一个值会被记住
>>>dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'};
>>>print "dict['Name']: ", dict['Name'];
>>>dict['Name']:  Manni

2)        键必须不可变，所以可以用数，字符串或元组充当，所以用列表就不行
>>>dict = {['Name']: 'Zara', 'Age': 7};
```

>>>print “dict[‘Name’]:”, dict[‘Name’]; raceback (most recent call last): File “test.py”, line 3, in >>>dict = {[‘Name’]: ‘Zara’, ‘Age’: 7}; TypeError: list objects are unhashable ### 字典内置函数&方法

```
Python字典包含了以下内置函数：

序号        函数及描述
1        cmp(dict1, dict2)                比较两个字典元素。
2        len(dict)                        计算字典元素个数，即键的总数。
3        str(dict)                        输出字典可打印的字符串表示。
```

```
4        type(variable)                          返回输入的变量类型，如果变量是字典就返回字典类型。
Python字典包含了以下内置函数：

序号         函数及描述
1        radiansdict.clear()                删除字典内所有元素
2        radiansdict.copy()                返回一个字典的浅复制
3        radiansdict.fromkeys()        创建一个新字典，以序列seq中元素做字典的键，val为字典所有键对应
的初始值
4        radiansdict.get(key, default=None)              返回指定键的值，如果值不在字典中返回default
值
5        radiansdict.has_key(key)        如果键在字典dict里返回true，否则返回false
6        radiansdict.items()                        以列表返回可遍历的(键，值）元组数组
7        radiansdict.keys()                        以列表返回一个字典所有的键
8        radiansdict.setdefault(key, default=None)
                                                    和get()类似，但如果键不已经存在于字典中，将会添
加键并将值设为default
9        radiansdict.update(dict2)        把字典dict2的键/值对更新到dict里
10        radiansdict.values()                以列表返回字典中的所有值
print dict['one'] # 输出键为'one' 的值
print dict[2] # 输出键为 2 的值
print tinydict # 输出完整的字典
print tinydict.keys() # 输出所有键
print tinydict.values() # 输出所有值
```

## 2.8 Python 日期和时间

```
Python程序能用很多方式处理日期和时间。转换日期格式是一个常见的例行琐事。
Python有一个time and calendar模组可以帮忙.
```

### 2.8.1 什么是Tick？

```
时间间隔是以秒为单位的浮点小数。
每个时间戳都以自从1970年1月1日午夜（历元）经过了多长时间来表示。
Python附带的受欢迎的time模块下有很多函数可以转换常见日期格式。
如函数time.time()用ticks计时单位返回从12:00am, January 1, 1970(epoch) 开始的记录的当前操作系统时间.
>>>ticks = time.time()
print "Number of ticks since 12:00am, January 1, 1970:", ticks
Number of ticks since 12:00am, January 1, 1970: 7186862.73399
```

### 2.8.2 什么是时间元组

```
序号         属性         值
0        tm_year         2008
1        tm_mon        1 到 12
```

```
2        tm_mday        1 到 31
3        tm_hour        0 到 23
4        tm_min        0 到 59
5        tm_sec        0 到 61 (60或61 是闰秒)
6        tm_wday        0到6 (0是周一)
7        tm_yday        1 到 366(儒略历)
8        tm_isdst        -1, 0, 1, -1是决定是否为夏令时的旗帜
```

### 2.8.3 获取当前时间

```
从返回浮点数的时间辍方式向时间元组转换，只要将浮点数传递给如localtime之类的函数。
>>>import time;
>>>localtime = time.localtime(time.time())
>>>print "Local current time :", localtime

Local current time : time.struct_time(tm_year=2013, tm_mon=7,
tm_mday=17, tm_hour=21, tm_min=26, tm_sec=3, tm_wday=2, tm_yday=198, tm_isdst=0)
```

### 2.8.4 获取格式化的时间

```
你可以根据需求选取各种格式，但是最简单的获取可读的时间模式的函数是asctime():
localtime = time.asctime( time.localtime(time.time()) )
print "Local current time :", localtime
Local current time : Tue Jan 13 10:17:09 2009
```

### 2.8.5 获取某月日历

```
>>>cal = calendar.month(2008, 1)
>>>print "Here is the calendar:"
>>>print cal;
Here is the calendar:
    January 2008
Mo Tu We Th Fr Sa Su
    1  2  3  4  5  6
 7  8  9 10 11 12 13
14 15 16 17 18 19 20
21 22 23 24 25 26 27
28 29 30 31
```

### 2.8.6 Time模块的内置函数

```
1                time.asctime([tupletime])                接受时间元组并返回一个可读的形式
为"Tue Dec 11 18:07:14 2008" (2008年12月11日
```

周二 18时07分14秒）的24个字符的字符串。
2 time.clock( )
用以浮点数计算的秒数返回当前的CPU时间。用来衡量不同程序的耗时，比time.time()更有用。
3 time.sleep(secs) 推迟调用线程的运行，secs指秒数。
4 time.time( ) 返回当前时间的时间戳（1970纪元后经过的浮点秒数）
Time模块包含了以下2个非常重要的属性：
序号 属性及描述
1 time.timezone
属性time.timezone是当地时区（未启动夏令时）距离格林威治的偏移秒数（>0，美洲;<=0大部分欧洲，亚洲，非洲）。
2 time.tzname
属性time.tzname包含一对根据情况的不同而不同的字符串，分别是带夏令时的本地时区名称，和不带的。

## 2.8.7 日历（Calendar）模块

此模块的函数都是日历相关的，例如打印某月的字符月历。
星期一是默认的每周第一天，星期天是默认的最后一天。
更改设置需调用calendar.setfirstweekday()函数。模块包含了以下内置函数：

序号 函数及描述
1 calendar.calendar(year,w=2,l=1,c=6) 返回一个多行字符串格式的year年年历，3个月一行，间隔距离为c。
每日宽度间隔为w字符。每行长度为21* W+18+2* C。l是每星期行数。
2 calendar.firstweekday( ) 返回当前每周起始日期的设置。默认情况下，首次载入caendar模块时返回0，即星期一。
3 calendar.isleap(year) 是闰年返回True，否则为false。
4 calendar.leapdays(y1,y2) 返回在Y1，Y2两年之间的闰年总数。
5 calendar.month(year,month,w=2,l=1) 返回一个多行字符串格式的year年month月日历，两行标题，一周一行。
每日宽度间隔为w字符。每行的长度为7* w+6。l是每星期的行数。
6 calendar.monthcalendar(year,month) 返回一个整数的单层嵌套列表。每个子列表装载代表一个星期的整数。
Year年month月外的日期都设为0;范围内的日子都由该月第几日表示，从1开始。
7 calendar.monthrange(year,month) 返回两个整数。
第一个是该月的星期几的日期码，第二个是该月的日期码。日从0（星期一）到6（星期日）;月从1到12。
8 calendar.prcal(year,w=2,l=1,c=6) 相当于 print calendar.calendar(year,w,l,c).
9 calendar.prmonth(year,month,w=2,l=1) 相当于 print calendar.calendar（year, w, l, c）。
10 calendar.setfirstweekday(weekday) 设置每周的起始日期码。0（星期一）到6（星期日）。
11 calendar.timegm(tupletime)
和time.gmtime相反：接受一个时间元组形式，返回该时刻的时间辍（1970纪元后经过的浮点秒数）。
12 calendar.weekday(year,month,day) 返回给定日期的日期码。0（星期一）到6（星期日）。
月份为 1（一月） 到 12（12月）。

## 2.9 函数高级

### 2.9.1 定义一个函数

你可以定义一个由自己想要功能的函数，以下是简单的规则：

函数代码块以def关键词开头，后接函数标识符名称和圆括号()。
任何传入参数和自变量必须放在圆括号中间。圆括号之间可以用于定义参数。
函数的第一行语句可以选择性地使用文档字符串-用于存放函数说明。
函数内容以冒号起始，并且缩进。
Return[expression]结束函数，选择性地返回一个值给调用方。不带表达式的return相当于返回 None

语法

```
def functionname( parameters ):
   "函数_文档字符串"
   function_suite
   return [expression]
```

默认情况下，参数值和参数名称是按函数声明中定义的的顺序匹配起来的。

实例

以下为一个简单的Python函数，它将一个字符串作为传入参数，再打印到标准显示设备上。

```
def printme( str ):
```

“打印传入的字符串到标准显示设备上” print str return ### 函数调用 定义一个函数只给了函数一个名称，指定了函数里包含的参数，和代码块结构。 这个函数的基本结构完成以后，你可以通过另一个函数调用执行，也可以直接从Python提示符执行。

```
#coding=utf-8
# Function definition is here
def printme( str ):
   "打印任何传入的字符串"
   print str;
   return;
# Now you can call printme function
printme("我要调用用户自定义函数!");
printme("再次调用同一函数");
```

以上实例输出结果：

```
我要调用用户自定义函数!
再次调用同一函数
```

### 2.9.2 按值传递参数和按引用传递参数

所有参数（自变量）在Python里都是按引用传递。
如果你在函数里修改了参数，那么在调用这个函数的函数里，原始的参数也被改变了。例如：

```
# 可写函数说明
def changeme( mylist ):
   "修改传入的列表"
   mylist.append([1,2,3,4]);
   print "函数内取值: ", mylist
   return

# 调用changeme函数
mylist = [10,20,30];
changeme( mylist );
print "函数外取值: ", mylist
```

传入函数的和在末尾添加新内容的对象用的是同一个引用。故输出结果如下：

```
函数内取值:  [10, 20, 30, [1, 2, 3, 4]]
函数外取值:  [10, 20, 30, [1, 2, 3, 4]]
```

### 2.9.3 参数

以下是调用函数时可使用的正式参数类型：

必备参数
命名参数
缺省参数
不定长参数

必备参数

　　必备参数须以正确的顺序传入函数。调用时的数量必须和声明时的一样。
　　调用printme()函数，你必须传入一个参数，不然会出现语法错误
命名参数
　　命名参数和函数调用关系紧密，调用方用参数的命名确定传入的参数值。你可以跳过不传的参数或者乱序传参，
　　因为Python解释器能够用参数名匹配参数值。用命名参数调用printme()函数：

```
　　def printme( str,name ):
　　　　"打印任何传入的字符串"
　　　　print str;
　　　　print name;
　　　　return;

　　#调用printme函数
　　printme( name = "test",str = "My string");
```

缺省参数

　　调用函数时，缺省参数的值如果没有传入，则被认为是默认值。下例会打印默认的age，如果age没有被传入：

```
　　#可写函数说明
　　def printinfo( name, age = 35 ):
　　   "打印任何传入的字符串"
　　   print "Name: ", name;
　　   print "Age ", age;
　　   return;
```

```
        #调用printinfo函数
        printinfo( age=50, name="miki" );
        printinfo( age = 9,name="miki" );
        以上实例输出结果：

        Name:  miki
        Age  50
        Name:  miki
        Age  35
不定长参数
        你可能需要一个函数能处理比当初声明时更多的参数。这些参数叫做不定长参数，声明时不会命名。
        基本语法如下：
        def functionname([formal_args,] *var_args_tuple ):
           "函数_文档字符串"
           function_suite
           return [expression]
        加了星号（*）的变量名会存放所有未命名的变量参数。选择不多传参数也可。如下实例：

        # 可写函数说明
        def printinfo( arg1, *vartuple ):
           "打印任何传入的参数"
           print "输出: "
           print arg1
           for var in vartuple:
              print var
           return;

        # 调用printinfo 函数
        printinfo( 10 );
        printinfo( 70, 60, 50 );
        以上实例输出结果：

        输出:
        10
        输出:
        70
        60
        50
```

### 2.9.4 匿名函数

用lambda关键词能创建小型匿名函数。这种函数得名于省略了用def声明函数的标准步骤。

Lambda函数能接收任何数量的参数但只能返回一个表达式的值，同时只能不能包含命令或多个表达式。
匿名函数不能直接调用print，因为lambda需要一个表达式。
lambda函数拥有自己的名字空间，且不能访问自有参数列表之外或全局名字空间里的参数。
虽然lambda函数看起来只能写一行，却不等同于C或C++的内联函数，后者的目的是调用小函数时不占用栈内存从而增加运行效率。
语法

lambda函数的语法只包含一个语句，如下：

```
lambda [arg1 [,arg2,.....argn]]:expression
```

如下实例：

```
#可写函数说明
sum = lambda arg1, arg2: arg1 + arg2;

#调用sum函数
print "Value of total : ", sum( 10, 20 )
print "Value of total : ", sum( 20, 20 )
```

以上实例输出结果：

```
Value of total :  30
Value of total :  40
```

## 2.9.5 return语句

return语句[表达式]退出函数，选择性地向调用方返回一个表达式。不带参数值的return语句返回None

```
def sum( arg1, arg2 ):
```

# 返回2个参数的和.” total = arg1 + arg2 print “Inside the function :”, total return total;

```
# 调用sum函数
total = sum( 10, 20 );
print "Outside the function : ", total
```

## 2.9.6 变量作用域

一个程序的所有的变量并不是在哪个位置都可以访问的。访问权限决定于这个变量是在哪里赋值的。
变量的作用域决定了在哪一部分程序你可以访问哪个特定的变量名称。两种最基本的变量作用域如下：

全局变量
局部变量

## 2.9.7 变量和局部变量

定义在函数内部的变量拥有一个局部作用域，定义在函数外的拥有全局作用域。
局部变量只能在其被声明的函数内部访问，而全局变量可以在整个程序范围内访问。
调用函数时，所有在函数内声明的变量名称都将被加入到作用域中。如下实例：

```
total = 0; # This is global variable
# 可写函数说明
def sum( arg1, arg2 ):
   #返回2个参数的和."
   total = arg1 + arg2; # total在这里是局部变量.
```

```
   print "Inside the function local total : ", total
   return total;

#调用sum函数
sum( 10, 20 );
print "Outside the function global total : ", total
以上实例输出结果：

Inside the function local total :  30
Outside the function global total :  0
```

## 2.10 模块

```
模块让你能够有逻辑地组织你的Python代码段。
把相关的代码分配到一个 模块里能让你的代码更好用，更易懂。
模块也是Python对象，具有随机的名字属性用来绑定或引用。
简单地说，模块就是一个保存了Python代码的文件。模块能定义函数，类和变量。模块里也能包含可执行的代码。
举例:
        一个叫做aname的模块里的Python代码一般都能在一个叫aname.py的文件中找到
```

### 2.10.1 import 语句

```
想使用Python源文件，只需在另一个源文件里执行import语句.可以自动防止重复import
形如:
        import module1,mudule2...
当解释器遇到import语句，如果模块在当前的搜索路径就会被导入。
# 导入模块
import support
# 现在可以调用模块里包含的函数了
support.print_func("Zara")
```

### 2.10.2 From…import 语句

```
Python的from语句让你从模块中导入一个指定的部分到当前命名空间中。语法如下：
from modname import name1[, name2[, ... nameN]]
例如，要导入模块fib的fibonacci函数，使用如下语句：

from fib import fibonacci
这个声明不会把整个fib模块导入到当前的命名空间中，它只会将fib里的fibonacci单个引入到执行这个声明的模
块的全局符号表
```

### 2.10.3 From…import* 语句

把一个模块的所有内容全都导入到当前的命名空间也是可行的，只需使用如下声明：

```
from modname import *
```

这提供了一个简单的方法来导入一个模块中的所有项目。然而这种声明不该被过多地使用。

### 2.10.4 定位模块

当你导入一个模块，Python解析器对模块位置的搜索顺序是：

当前目录
如果不在当前目录，Python则搜索在shell变量PYTHONPATH下的每个目录。
如果都找不到，Python会察看默认路径。UNIX下，默认路径一般为/usr/local/lib/python/
模块搜索路径存储在system模块的sys.path变量中。变量里包含当前目录，PYTHONPATH和由安装过程决定的默认目录。

### 2.10.5 PYTHONPATH变量

作为环境变量，PYTHONPATH由装在一个列表里的许多目录组成。PYTHONPATH的语法和shell变量PATH的一样。
在Windows系统，典型的PYTHONPATH如下：

```
        set PYTHONPATH=c:\python20\lib;
```

在UNIX系统，典型的PYTHONPATH如下：

```
        set PYTHONPATH=/usr/local/lib/python
```

### 2.10.6 命名空间和作用域

变量是拥有匹配对象的名字（标识符）。
命名空间是一个包含了变量名称们（键）和它们各自相应的对象们（值）的字典。
一个Python表达式可以访问局部命名空间和全局命名空间里的变量。同名隐藏的原则同C/C++

每个函数都有自己的命名空间。类的方法的作用域规则和通常函数的一样。
默认任何在函数内赋值的变量都是局部的。
因此，如果要给全局变量在一个函数里赋值，必须使用global语句。

global VarName的表达式会告诉Python， VarName是一个全局变量，这样Python就不会在局部命名空间里寻找这个变量了。

例如，我们在全局命名空间里定义一个变量money。我们再在函数内给变量money赋值，然后Python会假定money是一个局部变量。
然而，我们并没有在访问前声明一个局部变量money，结果就是会出现一个UnboundLocalError的错误。
取消global语句的注释就能解决这个问题。

```
Money = 2000
```

```
def AddMoney():
   # 想改正代码就取消以下注释:
   # global Money
   Money = Money + 1
print Money
AddMoney()
print Money
```

### 2.10.7 dir()函数

```
dir()函数一个排好序的字符串列表，内容是一个模块里定义过的名字。
返回的列表容纳了在一个模块里定义的所有模块，变量和函数。
import math
content = dir(math)
print content;

['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan',
'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp',
'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',
'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh']
```

### 2.10.8 globals()和locals()函数

```
根据调用地方的不同，globals()和locals()函数可被用来返回全局和局部命名空间里的名字。
如果在函数内部调用locals()，返回的是所有能在该函数里访问的命名。
如果在函数内部调用globals()，返回的是所有在该函数里能访问的全局名字。
两个函数的返回类型都是字典。所以名字们能用keys()函数摘取。
```

### 2.10.9 Python中的包

```
包是一个分层次的文件目录结构，它定义了一个由模块及子包，和子包下的子包等组成的Python的应用环境。

考虑一个在Phone目录下的pots.py文件。这个文件有如下源代码：

#coding=utf-8
#!/usr/bin/python

def Pots():
   print "I'm Pots Phone"

同样地，我们有另外两个保存了不同函数的文件：

Phone/Isdn.py 含有函数Isdn()
Phone/G3.py 含有函数G3()
现在，在Phone目录下创建file __init__.py：
```

Phone/__init__.py
当你导入Phone时，为了能够使用所有函数，你需要在__init__.py里使用显式的导入语句，如下：

```
from Pots import Pots
from Isdn import Isdn
from G3 import G3
```

当你把这些代码添加到__init__.py之后，导入Phone包的时候这些类就全都是可用的了。

```
# Now import your Phone Package.
import Phone

Phone.Pots()
Phone.Isdn()
Phone.G3()
```

以上实例输出结果：

```
I'm Pots Phone
I'm 3G Phone
I'm ISDN Phone
```

如上，为了举例，我们只在每个文件里放置了一个函数，但其实你可以放置许多函数。
你也可以在这些文件里定义Python的类，然后为这些类建一个包。

## 2.11 文件操作

### 2.11.1 打印到屏幕

最简单的输出方法是用print语句，你可以给它传递零个或多个用逗号隔开的表达式。或者使用占位符

### 2.11.2 读取键盘输入

Python提供了两个内置函数从标准输入读入一行文本，默认的标准输入是键盘。如下：
raw_input() input()函数

raw_input函数
raw_input([prompt]) 函数从标准输入读取一个行，并返回一个字符串（去掉结尾的换行符）：

```
str = raw_input("Enter your input: ");
print "Received input is : ", str
```

这将提示你输入任意字符串，然后在屏幕上显示相同的字符串。当我输入"Hello Python！"，它的输出如下：

```
Enter your input: Hello Python
Received input is :  Hello Python
```

input函数

input([prompt]) 函数和raw_input([prompt]) 函数基本可以互换，
但是input会假设你的输入是一个有效的Python表达式，并返回运算结果。

```
str = input("Enter your input: ");
print "Received input is : ", str
这会产生如下的对应着输入的结果：

Enter your input: [x*5 for x in range(2,10,2)]
Recieved input is :  [10, 20, 30, 40]
```

### 2.11.3 打开和关闭文件

```
你可以用file对象做大部分的文件操作。
open函数
        你必须先用Python内置的open()函数打开一个文件，创建一个file对象，相关的辅助方法才可以调用它进行读写。
file object = open(file_name [, access_mode][, buffering])
各个参数的细节如下：
file_name：file_name变量是一个包含了你要访问的文件名称的字符串值。
access_mode：access_mode决定了打开文件的模式：只读，写入，追加等。同C
buffering:   缓冲区的大小.(为0，就不会有寄存;为1，访问文件时会寄存行;大于1，寄存区的缓冲大小。
                        如果取负值，寄存区的缓冲大小则为系统默认
```

### 2.11.4 File对象的属性

```
一个文件被打开后，你有一个file对象，你可以得到有关该文件的各种信息。
以下是和file对象相关的所有属性的列表：
属性        描述
file.closed               返回true如果文件已被关闭，否则返回false。
file.mode                 返回被打开文件的访问模式。
file.name                 返回文件的名称。
file.softspace        如果用print输出后，必须跟一个空格符，则返回false。否则返回true。
```

### 2.11.5 Close()方法

```
File对象的close（）方法刷新缓冲区里任何还没写入的信息，并关闭该文件，这之后便不能再进行写入。
当一个文件对象的引用被重新指定给另一个文件时，Python会关闭之前的文件。用close（）方法关闭文件是一个很好的习惯。

fileObject.close();
```

### 2.11.6 Write()方法

```
Write()方法可将任何字符串写入一个打开的文件。需要重点注意的是，Python字符串可以是二进制数据，而不是仅仅是文字。
```

```
Write()方法不在字符串的结尾不添加换行符('\n')：
语法：
fileObject.write(string);
```

## 2.11.7 read()方法

```
read（）方法从一个打开的文件中读取一个字符串。需要重点注意的是，Python字符串可以是二进制数据，而不是仅仅是文字。
语法：
fileObject.read([count]);
//返回的为读取数据的引用

print file.read(10);

文件位置：
Tell()方法告诉你文件内的当前位置；换句话说，下一次的读写会发生在文件开头这么多字节之后：
seek（offset [,from]）方法改变当前文件的位置。Offset变量表示要移动的字节数。
From变量指定开始移动字节的参考位置。如果from被设为0，这意味着将文件的开头作为移动字节的参考位置。
如果设为1，则使用当前的位置作为参考位置。如果它被设为2，那么该文件的末尾将作为参考位置。

# 查找当前位置
position = fo.tell();
print "Current file position : ", position

# 把指针再次重新定位到文件开头
position = fo.seek(0, 0);
str = fo.read(10);
print "Again read String is : ", str
# 关闭打开的文件
fo.close()
```

## 2.11.8 重命名和删除文件

```
Python的os模块提供了帮你执行文件处理操作的方法，比如重命名和删除文件。
要使用这个模块，你必须先导入它，然后可以调用相关的各种功能。

rename()方法：
rename()方法需要两个参数，当前的文件名和新文件名。
语法：
        os.rename(current_file_name, new_file_name)
remove()方法
你可以用remove()方法删除文件，需要提供要删除的文件名作为参数。
语法：
        os.remove(file_name)
```

### 2.11.9 Python里的目录：

所有文件都包含在各个不同的目录下，不过Python也能轻松处理。os模块有许多方法能帮你创建，删除和更改目录。
mkdir()方法
可以使用os模块的mkdir()方法在当前目录下创建新的目录们。你需要提供一个包含了要创建的目录名称的参数。
语法：

```
os.mkdir("newdir")
```

### 2.11.10 chdir()方法

可以用chdir()方法来改变当前的目录。chdir()方法需要的一个参数是你想设成当前目录的目录名称。
语法：

```
os.chdir("newdir")
```

### 2.11.11 getcwd()方法：

getcwd()方法显示当前的工作目录。
语法：

```
os.getcwd()
```

### 2.11.12 rmdir()方法

rmdir()方法删除目录，目录名称以参数传递。

在删除这个目录之前，它的所有内容应该先被清除。

语法：

```
os.rmdir('dirname')
```

例子：
以下是删除" /tmp/test"目录的例子。目录的完全合规的名称必须被给出，否则会在当前目录下搜索该目录。

```
#coding=utf-8
#!/usr/bin/python
import os

# 删除”/tmp/test”目录
os.rmdir( "/tmp/test"  )
```

### 2.11.13 文件、目录相关的方法

三个重要的方法来源能对Windows和Unix操作系统上的文件及目录进行一个广泛且实用的处理及操控，如下：

File 对象方法：file对象提供了操作文件的一系列方法。
OS 对象方法：提供了处理文件及目录的一系列方法。

## 2.12 类与对象

### 2.12.1 面向对象技术简介

类(Class)：用来描述具有相同的属性和方法的对象的集合。它定义了该集合中每个对象所共有的属性和方法。对象是类的实例。
对象：通过类定义的数据结构实例。对象包括两个数据成员（类变量和实例变量）和方法。
实例化：创建一个类的实例，类的具体对象。

方法：类中定义的函数。
数据成员：类变量或者实例变量用于处理类及其实例对象的相关的数据。
方法重载：如果从父类继承的方法不能满足子类的需求，可以对其进行改写，这个过程叫方法的覆盖（override），重载。
实例变量：定义在方法中的变量，只作用于当前实例的类。
类变量：类变量在整个实例化的对象中是公用的。类变量定义在类中且在函数体之外。类变量通常不作为实例变量使用。
继承：　　即一个派生类（derived class）继承基类（base class）的字段和方法。
　　继承也允许把一个派生类的对象作为一个基类对象对待。
例如，有这样一个设计：一个Dog类型的对象派生自Animal类，这是模拟"是一个（is-a）"关系（例图，Dog是一个Animal）

### 2.12.2 创建类

使用class语句来创建一个新类，class之后为类的名称并以冒号结尾，如下实例:
类的属性包括成员变量和方法，其中方法的定义和普通函数的定义非常类似，但方法必须以self作为第一个参数。
可以直接在类外通过对象名访问，如果想定义成私有的，则需在前面加2个下划线 ' __'
构造方法__init__()方法是一种特殊的方法，被称为类的构造函数或初始化方法，当创建了这个类的实例时就会调用该方法.
构造方法支持重载，如果用户自己没有重新定义构造方法，系统就自动执行默认的构造方法。

析构方法__del__(self)在释放对象时调用，支持重载，可以在里面进行一些释放资源的操作，不需要显示调用。

```
class ClassName:
   '类的帮助信息'   #类文档字符串
        类变量               #类体 class_suite 由类成员，方法，数据属性组成
        def __init__(self,paramers):
        def 函数(self,...)
        ......
```

举例：

```
>>>class Employee:
        classSpec="itis a test class"
        def __init__(self, name, salary):
          self.name = name
          self.salary = salary
          Employee.empCount += 1
        def        hello(self,name):
        print"name= "+name
```

在Python类中定义的方法通常有三种：实例方法，类方法以及静态方法。
这三者之间的区别是实例方法一般都以self作为第一个参数，必须和具体的对象实例进行绑定才能访问，
而类方法以cls作为第一个参数，cls表示类本身，定义时使用@classmethod,那么通过cls引用的必定是类对象的属性和方法；

而静态方法不需要默认的任何参数,跟一般的普通函数类似.定义的时候使用@staticmethod
静态方法中不需要额外定义参数，因此在静态方法中引用类属性的话，必须通过类对象来引用。

而实例方法的第一个参数是实例对象self，那么通过self引用的可能是类属性、也有可能是实例属性（这个需要具体分析）
，不过在存在相同名称的类属性和实例属性的情况下，实例属性优先级更高。

### 2.12.3　创建实例对象

要创建一个类的实例，你可以使用类的名称，并通过__init__方法接受参数。

```
"创建 Employee 类的第一个对象"
emp1 = Employee("Zara", 2000)
"创建 Employee 类的第二个对象"
emp2 = Employee("Manni", 5000)
```

### 2.12.4　访问属性

使用点(.)来访问对象的属性。使用如下类的名称访问类变量:

```
emp1.displayEmployee()
```

你可以添加，删除，修改类的属性，如下所示：

```
emp1.age = 7  # 添加一个 'age' 属性
emp1.age = 8  # 修改 'age' 属性
del emp1.age  # 删除 'age' 属性
```

getattr(obj, name[, default]) : 访问对象的属性。
hasattr(obj,name) : 检查是否存在一个属性。
setattr(obj,name,value) : 设置一个属性。如果属性不存在，会创建一个新属性。
delattr(obj, name) : 删除属性。

### 2.12.5 Python内置类属性

```
__dict__ : 类的属性（包含一个字典，由类的数据属性组成）
__doc__ :类的文档字符串
__name__: 类名
__module__: 类定义所在的模块（类的全名是'__main__.className',
                        如果类位于一个导入模块mymod中，那么className.__module__ 等于 mymod）
__bases__ : 类的所有父类构成元素（包含了以个由所有父类组成的元组）
```

### 2.12.6 python对象销毁(垃圾回收)

在Python内部记录着所有使用中的对象各有多少引用。
一个内部跟踪变量，称为一个引用计数器。

当对象被创建时， 就创建了一个引用计数， 当这个对象不再需要时， 也就是说， 这个对象的引用计数变为 0 时，
它被垃圾回收。但是回收不是"立即"的， 由解释器在适当的时机，将垃圾对象占用的内存空间回收。

### 2.12.7 类的继承

面向对象的编程带来的主要好处之一是代码的重用，实现这种重用的方法之一是通过继承机制。
继承完全可以理解成类之间的类型和子类型关系。

继承语法 ：
class 派生类名（基类名）：//... 基类名写作括号里，基本类是在类定义的时候，在元组之中指明的。

在python中继承中的一些特点：

1：在继承中基类的构造（__init__()方法）不会被自动调用，它需要在其派生类的构造中亲自专门调用。
2：在调用基类的方法时，需要加上基类的类名前缀，且需要带上self参数变量。
            区别于在类中调用普通函数时并不需要带上self参数
3：Python总是首先查找对应类型的方法，如果它不能在派生类中找到对应的方法，它才开始到基类中逐个查找。
        （先在本类中查找调用的方法，找不到才去基类中找）。
如果在继承元组中列了一个以上的类，那么它就被称作"多重继承" 。

语法：

派生类的声明，与他们的父类类似，继承的基类列表跟在类名之后，如下所示：

```
class SubClassName (ParentClass1[, ParentClass2, ...]):
   'Optional class documentation string'
   class_suite
```

实例：

```
#coding=utf-8
#!/usr/bin/python

class Parent:        # 定义父类
```

```
   parentAttr = 100
   def __init__(self):
      print "调用父类构造函数"

   def parentMethod(self):
      print '调用父类方法'

   def setAttr(self, attr):
      Parent.parentAttr = attr

   def getAttr(self):
      print "父类属性 :", Parent.parentAttr

class Child(Parent): # 定义子类
   def __init__(self):
      print "调用子类构造方法"

   def childMethod(self):
      print '调用子类方法 child method'

c = Child()          # 实例化子类
c.childMethod()      # 调用子类的方法
c.parentMethod()     # 调用父类方法
c.setAttr(200)       # 再次调用父类的方法
c.getAttr()          # 再次调用父类的方法
```

以上代码执行结果如下：

```
调用子类构造方法
调用子类方法 child method
调用父类方法
父类属性 : 200
```

你可以继承多个类

```
class A:        # 定义类 A
.....

class B:         # 定义类 B
.....

class C(A, B):   # 继承类 A 和 B
.....
```

你可以使用issubclass()或者isinstance()方法来检测。

issubclass() - 布尔函数判断一个类是另一个类的子类或者子孙类，语法：issubclass(sub,sup)
isinstance(obj, Class) 布尔函数如果obj是Class类的实例对象或者是一个Class子类的实例对象则返回true。
方法重写
如果你的父类方法的功能不能满足你的需求，你可以在子类重写你父类的方法：

实例：

```
#coding=utf-8
#!/usr/bin/python

class Parent:        # 定义父类
   def myMethod(self):
      print '调用父类方法'
```

```
class Child(Parent): # 定义子类
   def myMethod(self):
      print '调用子类方法'

c = Child()          # 子类实例
c.myMethod()         # 子类调用重写方法
```

执行以上代码输出结果如下：

```
调用子类方法
基础重载方法
```

下表列出了一些通用的功能，你可以在自己的类重写：

```
序号        方法，描述 & 简单的调用
1        __init__ ( self [,args...] )
构造函数
简单的调用方法: obj = className(args)
2        __del__( self )
析构方法，删除一个对象
简单的调用方法 : dell obj
3        __repr__( self )
转化为供解释器读取的形式
简单的调用方法 : repr(obj)
4        __str__( self )
用于将值转化为适于人阅读的形式
简单的调用方法 : str(obj)
5        __cmp__ ( self, x )
对象比较
简单的调用方法 : cmp(obj, x)
```

运算符重载

Python同样支持运算符重载，实例如下：

```
#!/usr/bin/python

class Vector:
   def __init__(self, a, b):
      self.a = a
      self.b = b

   def __str__(self):
      return 'Vector (%d, %d)' % (self.a, self.b)

   def __add__(self,other):
      return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)
v2 = Vector(5,-2)
print v1 + v2
```

以上代码执行结果如下所示:

```
Vector(7,8)
```

类属性与方法

类的私有属性

__private_attrs：两个下划线开头，声明该属性为私有，不能在类地外部被使用或直接访问。在类内部的方法中使用时 self.__private_attrs。

类的方法

在类地内部，使用def关键字可以为类定义一个方法，与一般函数定义不同，类方法必须包含参数self,且为第一个参数

类的私有方法

__private_method：两个下划线开头，声明该方法为私有方法，不能在类地外部调用。在类的内部调用 slef.__private_methods

实例

```
#coding=utf-8
#!/usr/bin/python

class JustCounter:
        __secretCount = 0  # 私有变量
        publicCount = 0    # 公开变量

        def count(self):
                self.__secretCount += 1
                self.publicCount += 1
                print self.__secretCount

counter = JustCounter()
counter.count()
counter.count()
print counter.publicCount
print counter.__secretCount  # 报错，实例不能访问私有变量
```

Python 通过改变名称来包含类名:

```
1
2
2
Traceback (most recent call last):
  File "test.py", line 17, in <module>
    print counter.__secretCount  # 报错，实例不能访问私有变量
AttributeError: JustCounter instance has no attribute '__secretCount'
```

Python不允许实例化的类访问私有数据，但你可以使用 object._className__attrName 访问属性，将如下代码替换以上代码的最后一行代码：

```
.......................
print counter._JustCounter__secretCount
```

执行以上代码，执行结果如下：

```
1
2
2
2
```

# 2.13 Python正则表达式

正则表达式是一个特殊的字符序列，它能帮助你方便的检查一个字符串是否与某种模式匹配。
Python 自1.5版本起增加了re 模块，它提供 Perl 风格的正则表达式模式。

re 模块使 Python 语言拥有全部的正则表达式功能。
re 模块也提供了与这些方法功能完全一致的函数，这些函数使用一个模式字符串做为它们的第一个参数。

compile 函数根据一个模式字符串和可选的标志参数生成一个正则表达式对象。
该对象拥有一系列方法用于正则表达式匹配和替换。

re.match函数
re.match 尝试从字符串的开始匹配一个模式。
函数语法： re.match(pattern, string, flags=0)
函数参数说明：

参数 描述
pattern 匹配的正则表达式
string 要匹配的字符串。
flags 标志位，用于控制正则表达式的匹配方式，如：是否区分大小写，多行匹配等等。

匹配成功re.match方法返回一个匹配的对象，否则返回None。

我们可以使用group(num) 或 groups() 匹配对象函数来获取匹配表达式。

匹配对象方法 描述
group(num=0)
匹配的整个表达式的字符串，group() 可以一次输入多个组号，在这种情况下它将返回一个包含那些组所对应值的元组。
groups()
返回一个包含所有小组字符串的元组，从 1 到 所含的小组号。
实例：

```
#!/usr/bin/python
import re

line = "Cats are smarter than dogs"

matchObj = re.match( r'(.*) are (.*?) .*', line, re.M|re.I)

if matchObj:
   print "matchObj.group() : ", matchObj.group()
   print "matchObj.group(1) : ", matchObj.group(1)
   print "matchObj.group(2) : ", matchObj.group(2)
else:
   print "No match!!"
```

以上实例执行结果如下：

```
matchObj.group() :  Cats are smarter than dogs
matchObj.group(1) :  Cats
matchObj.group(2) :  smarter
```

re.search方法
re.match 尝试从字符串的开始匹配一个模式。

函数语法：

```
re.search(pattern, string, flags=0)
```

函数参数说明：

| 参数 | 描述 |
| --- | --- |
| pattern | 匹配的正则表达式 |
| string | 要匹配的字符串。 |
| flags | 标志位，用于控制正则表达式的匹配方式，如：是否区分大小写，多行匹配等等。 |

匹配成功re.search方法方法返回一个匹配的对象，否则返回None。

我们可以使用group(num) 或 groups() 匹配对象函数来获取匹配表达式。

| 匹配对象方法 | 描述 |
| --- | --- |
| group(num=0) | 匹配的整个表达式的字符串，group() 可以一次输入多个组号，在这种情况下它将返回一个包含那些组所对应值的元组。 |
| groups() | 返回一个包含所有小组字符串的元组，从 1 到 所含的小组号。 |

实例：

```
#!/usr/bin/python
import re

line = "Cats are smarter than dogs";

matchObj = re.match( r'(.*) are (.*?) .*', line, re.M|re.I)

if matchObj:
   print "matchObj.group() : ", matchObj.group()
   print "matchObj.group(1) : ", matchObj.group(1)
   print "matchObj.group(2) : ", matchObj.group(2)
else:
   print "No match!!" 以上实例执行结果如下：
matchObj.group() :  Cats are smarter than dogs
matchObj.group(1) :  Cats
matchObj.group(2) :  smarter
```

re.match与re.search的区别

re.match只匹配字符串的开始，如果字符串开始不符合正则表达式，则匹配失败，函数返回None；而re.search匹配整个字符串，直到找到一个匹配。

实例：

```
#!/usr/bin/python
import re

line = "Cats are smarter than dogs";

matchObj = re.match( r'dogs', line, re.M|re.I)
if matchObj:
   print "match --> matchObj.group() : ", matchObj.group()
else:
   print "No match!!"

matchObj = re.search( r'dogs', line, re.M|re.I)
if matchObj:
   print "search --> matchObj.group() : ", matchObj.group()
else:
```

```
   print "No match!!" 以上实例运行结果如下：
No match!!
search --> matchObj.group() :  dogs
```

检索和替换

Python 的re模块提供了re.sub用于替换字符串中的匹配项。

语法：

```
re.sub(pattern, repl, string, max=0)
```

返回的字符串是在字符串中用 RE 最左边不重复的匹配来替换。如果模式没有发现，字符将被没有改变地返回。

可选参数 count 是模式匹配后替换的最大次数；count 必须是非负整数。缺省值是 0 表示替换所有的匹配。

实例：

```
#!/usr/bin/python
import re

phone = "2004-959-559 # This is Phone Number"

# Delete Python-style comments
num = re.sub(r'#.*$', "", phone)
print "Phone Num : ", num

# Remove anything other than digits
num = re.sub(r'\D', "", phone)
print "Phone Num : ", num 以上实例执行结果如下：
Phone Num :  2004-959-559
Phone Num :  2004959559
```

正则表达式修饰符 - 可选标志

正则表达式可以包含一些可选标志修饰符来控制匹配的模式。修饰符被指定为一个可选的标志。多个标志可以通过按位 OR(|) 它们来指定。如 re.I | re.M 被设置成 I 和 M 标志：

| 修饰符 | 描述 |
| --- | --- |
| re.I | 使匹配对大小写不敏感 |
| re.L | 做本地化识别（locale-aware）匹配 |
| re.M | 多行匹配，影响 ^ 和 $ |
| re.S | 使 . 匹配包括换行在内的所有字符 |
| re.U | 根据Unicode字符集解析字符。这个标志影响 \w, \W, \b, \B. |
| re.X | 该标志通过给予你更灵活的格式以便你将正则表达式写得更易于理解。 |

正则表达式模式

模式字符串使用特殊的语法来表示一个正则表达式：

字母和数字表示他们自身。一个正则表达式模式中的字母和数字匹配同样的字符串。

多数字母和数字前加一个反斜杠时会拥有不同的含义。

标点符号只有被转义时才匹配自身，否则它们表示特殊的含义。

反斜杠本身需要使用反斜杠转义。

由于正则表达式通常都包含反斜杠，所以你最好使用原始字符串来表示它们。模式元素(如 r'/t'，等价于'//t')匹配相应的特殊字符。

下表列出了正则表达式模式语法中的特殊元素。如果你使用模式的同时提供了可选的标志参数，某些模式元素的含义会改变。

```
模式        描述
^        匹配字符串的开头
$        匹配字符串的末尾。
.        匹配任意字符，除了换行符，当re.DOTALL标记被指定时，则可以匹配包括换行符的任意字符。
[...]        用来表示一组字符,单独列出：[amk] 匹配 'a', 'm'或'k'
[^...]        不在[]中的字符：[^abc] 匹配除了a,b,c之外的字符。
re*        匹配0个或多个的表达式。
re+        匹配1个或多个的表达式。
re?        匹配0个或1个由前面的正则表达式定义的片段，贪婪方式
re{ n}
re{ n,}        精确匹配n个前面表达式。
re{ n, m}        匹配 n 到 m 次由前面的正则表达式定义的片段，贪婪方式
a| b        匹配a或b
(re)        G匹配括号内的表达式，也表示一个组
(?imx)        正则表达式包含三种可选标志：i, m, 或 x 。只影响括号中的区域。
(?-imx)        正则表达式关闭 i, m, 或 x 可选标志。只影响括号中的区域。
(?: re)        类似 (...), 但是不表示一个组
(?imx: re)        在括号中使用i, m, 或 x 可选标志
(?-imx: re)        在括号中不使用i, m, 或 x 可选标志
(?#...)        注释.
(?= re)        前向肯定界定符。如果所含正则表达式，以 ... 表示，在当前位置成功匹配时成功，否则失败。
但一旦所含表达式已经尝试，匹配引擎根本没有提高；模式的剩余部分还要尝试界定符的右边。
(?! re)        前向否定界定符。与肯定界定符相反；当所含表达式不能在字符串当前位置匹配时成功
(?> re)        匹配的独立模式，省去回溯。
\w        匹配字母数字
\W        匹配非字母数字
\s        匹配任意空白字符，等价于 [\t\n\r\f].
\S        匹配任意非空字符
\d        匹配任意数字，等价于 [0-9].
\D        匹配任意非数字
\A        匹配字符串开始
\Z        匹配字符串结束，如果是存在换行，只匹配到换行前的结束字符串。c
\z        匹配字符串结束
\G        匹配最后匹配完成的位置。
\b        匹配一个单词边界，也就是指单词和空格间的位置。例如，  'er\b' 可以匹配"never" 中的 'er'，但
不能匹配 "verb" 中的 'er'。
\B        匹配非单词边界。'er\B' 能匹配 "verb" 中的 'er'，但不能匹配 "never" 中的 'er'。
\n, \t, 等.        匹配一个换行符。匹配一个制表符。等
\1...\9        比赛第n个分组的子表达式。
\10        匹配第n个分组的子表达式，如果它经匹配。否则指的是八进制字符码的表达式。
正则表达式实例
字符匹配

实例        描述
python        匹配 "python".
字符类

实例        描述
[Pp]ython        匹配 "Python" 或 "python"
rub[ye]        匹配 "ruby" 或 "rube"
[aeiou]        匹配中括号内的任意一个字母
[0-9]        匹配任何数字。类似于 [0123456789]
[a-z]        匹配任何小写字母
[A-Z]        匹配任何大写字母
[a-zA-Z0-9]        匹配任何字母及数字
```

```
[^aeiou]        除了aeiou字母以外的所有字符
[^0-9]        匹配除了数字外的字符
特殊字符类

实例         描述
.         匹配除 "\n" 之外的任何单个字符。要匹配包括 '\n' 在内的任何字符，请使用象 '[.\n]' 的模式。
\d         匹配一个数字字符。等价于 [0-9]。
\D         匹配一个非数字字符。等价于 [^0-9]。
\s         匹配任何空白字符，包括空格、制表符、换页符等等。等价于 [ \f\n\r\t\v]。
\S         匹配任何非空白字符。等价于 [^ \f\n\r\t\v]。
\w         匹配包括下划线的任何单词字符。等价于'[A-Za-z0-9_]'。
\W         匹配任何非单词字符。等价于 '[^A-Za-z0-9_]'。
```

## 2.14 python操作MySQL数据库编程

```
python 标准数据库接口为 Python DB-API，Python DB-API为开发人员提供了数据库应用编程接口
不同的数据库你需要下载不同的DB API模块，例如你需要访问Mysql数据库，你需要下载MySQL数据库模块。
DB-API 是一个规范. 它定义了一系列必须的对象和数据库存取方式，以便为各种各样的底层数据库系统和
多种多样的数据库接口程序提供一致的访问接口 。
Python的DB-API，为大多数的数据库实现了接口，使用它连接各数据库后，就可以用相同的方式操作各数据库。
Python DB-API使用流程：
        引入 API 模块。
        获取与数据库的连接。
        执行SQL语句和存储过程。
        关闭数据库连接。
```

### 2.14.1 什么是MySQLdb?

```
MySQLdb 是用于Python链接Mysql数据库的接口，它实现了 Python 数据库 API 规范 V2.0，基于 MySQL C API 上
建立的。
```

### 2.14.2 如何安装MySQLdb?

```
为了用DB-API编写MySQL脚本，必须确保已经安装了MySQL。复制以下代码，并执行：

$python
>>>import MySQLdb
如果执行后的输出结果如下所示，意味着你没有安装 MySQLdb 模块：

Traceback (most recent call last):
  File "test.py", line 3, in <module>
    import MySQLdb
ImportError: No module named MySQLdb
安装MySQLdb，请访问 http://sourceforge.net/projects/mysql-python ,
(Linux平台可以访问：        https://pypi.python.org/pypi/MySQL-python)
```

从这里可选择适合您的平台的安装包，分为预编译的二进制文件和源代码安装包。

如果您选择二进制文件发行版本的话，安装过程基本安装提示即可完成。
如果从源代码进行安装的话，则需要切换到MySQLdb发行版本的顶级目录，并键入下列命令：

```
$ gunzip MySQL-python-1.2.2.tar.gz
$ tar -xvf MySQL-python-1.2.2.tar
$ cd MySQL-python-1.2.2
$ python setup.py build
$ python setup.py install
```

注意：请确保您有root权限来安装上述模块。

## 2.14.3 数据库连接

连接数据库前，请先确认以下事项：

您已经创建了数据库 TESTDB.
在TESTDB数据库中您已经创建了表 EMPLOYEE
EMPLOYEE表字段为 FIRST_NAME, LAST_NAME, AGE, SEX 和 INCOME。
连接数据库TESTDB使用的用户名为 "testuser" ，密码为 "test123",你可以可以自己设定或者直接使用root用户名及其密码，Mysql数据库用户授权请使用Grant命令。
在你的机子上已经安装了 Python MySQLdb 模块。
如果您对sql语句不熟悉，可以访问我们的 SQL基础教程
实例：

以下实例链接Mysql的TESTDB数据库：

```
# encoding: utf-8
#!/usr/bin/python

import MySQLdb

# 打开数据库连接
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# 使用execute方法执行SQL语句
cursor.execute("SELECT VERSION()")

# 使用 fetchone() 方法获取一条数据库。
data = cursor.fetchone()

print "Database version : %s " % data

# 关闭数据库连接
db.close() 执行以上脚本输出结果如下：
Database version : 5.0.45
创建数据库表
如果数据库连接存在我们可以使用execute()方法来为数据库创建表，如下所示创建表EMPLOYEE：

# encoding: utf-8
#!/usr/bin/python
```

```
import MySQLdb

# 打开数据库连接
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# 如果数据表已经存在使用 execute() 方法删除表。
cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")

# 创建数据表SQL语句
sql = """CREATE TABLE EMPLOYEE (
         FIRST_NAME  CHAR(20) NOT NULL,
         LAST_NAME  CHAR(20),
         AGE INT,
         SEX CHAR(1),
         INCOME FLOAT )"""

cursor.execute(sql)

# 关闭数据库连接
db.close()
```

### 2.14.4 数据库插入操作

以下实例使用执行 SQL INSERT 语句向表 EMPLOYEE 插入记录：

```
# encoding: utf-8
#!/usr/bin/python

import MySQLdb

# 打开数据库连接
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# SQL 插入语句
sql = """INSERT INTO EMPLOYEE(FIRST_NAME,
         LAST_NAME, AGE, SEX, INCOME)
         VALUES ('Mac', 'Mohan', 20, 'M', 2000)"""
try:
   # 执行sql语句
   cursor.execute(sql)
   # 提交到数据库执行
   db.commit()
except:
   # Rollback in case there is any error
   db.rollback()
```

```
# 关闭数据库连接
db.close()
```

以上例子也可以写成如下形式：

```
# encoding: utf-8
#!/usr/bin/python

import MySQLdb

# 打开数据库连接
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# SQL 插入语句
sql = "INSERT INTO EMPLOYEE(FIRST_NAME, \
       LAST_NAME, AGE, SEX, INCOME) \
       VALUES ('%s', '%s', '%d', '%c', '%d' )" % \
       ('Mac', 'Mohan', 20, 'M', 2000)
try:
   # 执行sql语句
   cursor.execute(sql)
   # 提交到数据库执行
   db.commit()
except:
   # 发生错误时回滚
   db.rollback()

# 关闭数据库连接
db.close()
```

实例：

以下代码使用变量向SQL语句中传递参数:

```
.................................
user_id = "test123"
password = "password"

con.execute('insert into Login values("%s", "%s")' % \
            (user_id, password))
.................................
```

### 2.14.5 数据库查询操作

Python查询Mysql使用 fetchone() 方法获取单条数据, 使用fetchall() 方法获取多条数据。

fetchone(): 该方法获取下一个查询结果集。结果集是一个对象
fetchall():接收全部的返回结果行.
rowcount: 这是一个只读属性，并返回执行execute()方法后影响的行数。
实例：

查询EMPLOYEE表中salary（工资）字段大于1000的所有数据：

```
# encoding: utf-8
#!/usr/bin/python

import MySQLdb

# 打开数据库连接
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# SQL 查询语句
sql = "SELECT * FROM EMPLOYEE \
       WHERE INCOME > '%d'" % (1000)
try:
   # 执行SQL语句
   cursor.execute(sql)
   # 获取所有记录列表
   results = cursor.fetchall()
   for row in results:
      fname = row[0]
      lname = row[1]
      age = row[2]
      sex = row[3]
      income = row[4]
      # 打印结果
      print "fname=%s,lname=%s,age=%d,sex=%s,income=%d" % \
             (fname, lname, age, sex, income )
except:
   print "Error: unable to fecth data"

# 关闭数据库连接
db.close()
```

以上脚本执行结果如下：

```
fname=Mac, lname=Mohan, age=20, sex=M, income=2000
```

数据库更新操作

更新操作用于更新数据表的的数据，以下实例将 TESTDB表中的 SEX 字段全部修改为 'M'，AGE 字段递增1：

```
# encoding: utf-8
#!/usr/bin/python

import MySQLdb

# 打开数据库连接
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# SQL 更新语句
sql = "UPDATE EMPLOYEE SET AGE = AGE + 1
                          WHERE SEX = '%c'" % ('M')
try:
```

```
        # 执行SQL语句
        cursor.execute(sql)
        # 提交到数据库执行
        db.commit()
    except:
        # 发生错误时回滚
        db.rollback()

    # 关闭数据库连接
    db.close()
```

### 2.14.6 执行事务

事务机制可以确保数据一致性。

事务应该具有4个属性：原子性、一致性、隔离性、持久性。这四个属性通常称为ACID特性。

原子性（atomicity）。一个事务是一个不可分割的工作单位，事务中包括的诸操作要么都做，要么都不做。

一致性（consistency）。事务必须是使数据库从一个一致性状态变到另一个一致性状态。一致性与原子性是密切相关的。

隔离性（isolation）。一个事务的执行不能被其他事务干扰。即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。

持久性（durability）。持续性也称永久性（permanence），指一个事务一旦提交，它对数据库中数据的改变就应该是永久性的。接下来的其他操作或故障不应该对其有任何影响。

Python DB API 2.0 的事务提供了两个方法 commit 或 rollback。

实例：

```
# SQL删除记录语句
sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (20)
try:
   # 执行SQL语句
   cursor.execute(sql)
   # 向数据库提交
   db.commit()
except:
   # 发生错误时回滚
   db.rollback()
```

对于支持事务的数据库， 在Python数据库编程中，当游标建立之时，就自动开始了一个隐形的数据库事务。

commit()方法游标的所有更新操作，rollback（）方法回滚当前游标的所有操作。每一个方法都开始了一个新的事务。

## 2.15 python操作MongoDB数据库编程

# 2.16 Python使用SMTP发送邮件

SMTP（Simple Mail Transfer Protocol）即简单邮件传输协议,它是一组用于由源地址到目的地址传送邮件的规则,
由它来控制信件的中转方式。
python的smtplib提供了一种很方便的途径发送电子邮件。它对smtp协议进行了简单的封装

## 2.16.1 Python创建 SMTP 对象

语法如下：
import smtplib
smtpObj = smtplib.SMTP( [host [, port [, local_hostname]]] )
参数说明：
host: SMTP 服务器主机。 你可以指定主机的ip地址或者域名如:w3cschool.cc，这个是可选参数。
port: 如果你提供了 host 参数，你需要指定 SMTP 服务使用的端口号，一般情况下SMTP端口号为25。
local_hostname: 如果SMTP在你的本机上，你只需要指定服务器地址为 localhost 即可。

## 2.16.2 Python SMTP对象使用sendmail方法发送邮件

语法如下：
SMTP.sendmail(from_addr, to_addrs, msg[, mail_options, rcpt_options]
参数说明：
from_addr: 邮件发送者地址。
to_addrs: 字符串列表，邮件发送地址。
msg: 发送消息
这里要注意一下第三个参数，msg是字符串，表示邮件。我们知道邮件一般由标题，发信人，收件人，邮件内容，附件等构成,
发送邮件的时候，要注意msg的格式。这个格式就是smtp协议中定义的格式
实例

以下是一个使用Python发送邮件简单的实例：

```
#!/usr/bin/python

import smtplib

sender = 'from@fromdomain.com'
receivers = ['to@todomain.com']

message = """From: From Person <from@fromdomain.com>
To: To Person <to@todomain.com>
Subject: SMTP e-mail test

This is a test e-mail message.
"""

try:
```

```
   smtpObj = smtplib.SMTP('localhost')
   smtpObj.sendmail(sender, receivers, message)
   print "Successfully sent email"
except SMTPException:
   print "Error: unable to send email"
```

使用Python发送HTML格式的邮件

Python发送HTML格式的邮件与发送纯文本消息的邮件不同之处就是将MIMEText中_subtype设置为html。具体代码如下：

```
import smtplib
from email.mime.text import MIMEText
mailto_list=["YYY@YYY.com"]
mail_host="smtp.XXX.com"  #设置服务器
mail_user="XXX"    #用户名
mail_pass="XXXX"   #口令
mail_postfix="XXX.com"  #发件箱的后缀

def send_mail(to_list,sub,content):  #to_list：收件人；sub：主题；content：邮件内容
    me="hello"+"<"+mail_user+"@"+mail_postfix+">"   #这里的hello可以任意设置，收到信后，将按照设置显示
    msg = MIMEText(content,_subtype='html',_charset='gb2312')    #创建一个实例，这里设置为html格式邮件
    msg['Subject'] = sub    #设置主题
    msg['From'] = me
    msg['To'] = ";".join(to_list)
    try:
        s = smtplib.SMTP()
        s.connect(mail_host)  #连接smtp服务器
        s.login(mail_user,mail_pass)  #登陆服务器
        s.sendmail(me, to_list, msg.as_string())  #发送邮件
        s.close()
        return True
    except Exception, e:
        print str(e)
        return False
if __name__ == '__main__':
    if send_mail(mailto_list,"hello","<a href='http://www.cnblogs.com/xiaowuyi'>小五义</a>"):
        print "发送成功"
    else:
        print "发送失败"
```

或者你也可以在消息体中指定Content-type为text/html,如下实例:

```
#!/usr/bin/python

import smtplib

message = """From: From Person <from@fromdomain.com>
To: To Person <to@todomain.com>
MIME-Version: 1.0
Content-type: text/html
Subject: SMTP HTML e-mail test

This is an e-mail message to be sent in HTML format

<b>This is HTML message.</b>
<h1>This is headline.</h1>
```

```
"""

try:
   smtpObj = smtplib.SMTP('localhost')
   smtpObj.sendmail(sender, receivers, message)
   print "Successfully sent email"
except SMTPException:
   print "Error: unable to send email"
```

### 2.16.3 Python发送带附件的邮件

发送带附件的邮件，首先要创建MIMEMultipart()实例，然后构造附件，如果有多个附件，可依次构造，最后利用smtplib.smtp发送。

```
        from email.mime.text import MIMEText
        from email.mime.multipart import MIMEMultipart
        import smtplib

        #创建一个带附件的实例
        msg = MIMEMultipart()

        #构造附件1
        att1 = MIMEText(open('d:\\123.rar', 'rb').read(), 'base64', 'gb2312')
        att1["Content-Type"] = 'application/octet-stream'
         att1["Content-Disposition"] = 'attachment; filename="123.doc"'#这里的filename可以任意写，写什么名字，邮件中显示什么名字
        msg.attach(att1)

        #构造附件2
        att2 = MIMEText(open('d:\\123.txt', 'rb').read(), 'base64', 'gb2312')
        att2["Content-Type"] = 'application/octet-stream'
        att2["Content-Disposition"] = 'attachment; filename="123.txt"'
        msg.attach(att2)

        #加邮件头
        msg['to'] = 'YYY@YYY.com'
        msg['from'] = 'XXX@XXX.com'
        msg['subject'] = 'hello world'
        #发送邮件
        try:
            server = smtplib.SMTP()
            server.connect('smtp.XXX.com')
            server.login('XXX','XXXXX')#XXX为用户名，XXXXX为密码
            server.sendmail(msg['from'], msg['to'],msg.as_string())
            server.quit()
            print '发送成功'
        except Exception, e:
            print str(e)
        以下实例指定了Content-type header 为 multipart/mixed，并发送/tmp/test.txt 文本文件：

        #!/usr/bin/python

        import smtplib
```

```
import base64

filename = "/tmp/test.txt"

# 读取文件内容并使用 base64 编码
fo = open(filename, "rb")
filecontent = fo.read()
encodedcontent = base64.b64encode(filecontent)  # base64

sender = 'webmaster@tutorialpoint.com'
reciever = 'amrood.admin@gmail.com'

marker = "AUNIQUEMARKER"

body ="""
This is a test email to send an attachement.
"""
# 定义头部信息
part1 = """From: From Person <me@fromdomain.net>
To: To Person <amrood.admin@gmail.com>
Subject: Sending Attachement
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary=%s
--%s
""" % (marker, marker)

# 定义消息动作
part2 = """Content-Type: text/plain
Content-Transfer-Encoding:8bit

%s
--%s
""" % (body,marker)

# 定义附近部分
part3 = """Content-Type: multipart/mixed; name=\"%s\"
Content-Transfer-Encoding:base64
Content-Disposition: attachment; filename=%s

%s
--%s--
""" %(filename, filename, encodedcontent, marker)
message = part1 + part2 + part3

try:
   smtpObj = smtplib.SMTP('localhost')
   smtpObj.sendmail(sender, reciever, message)
   print "Successfully sent email"
except Exception:
   print "Error: unable to send email"
```

## 2.17 Python 多线程编程

## 2.17.1 多线程运行有如下优点：

使用线程可以把占据长时间的程序中的任务放到后台去处理。
用户界面可以更加吸引人，这样比如用户点击了一个按钮去触发某些事件的处理，可以弹出一个进度条来显示处理的进度
程序的运行速度可能加快在一些等待的任务实现上如用户输入、文件读写和网络收发数据等，线程就比较有用了。
在这种情况下我们可以释放一些珍贵的资源如内存占用等等。
每个线程都有他自己的一组CPU寄存器，称为线程的上下文，该上下文反映了线程上次运行该线程的CPU寄存器的状态。

指令指针和堆栈指针寄存器是线程上下文中两个最重要的寄存器，
线程总是在进程得到上下文中运行的，这些地址都用于标志拥有线程的进程地址空间中的内存。

线程可以被抢占（中断）。
在其他线程正在运行时，线程可以暂时搁置（也称为睡眠） -- 这就是线程的退让。

## 2.17.2 线程的创建

Python中使用线程有两种方式：函数或者用类来包装线程对象。
函数式：调用thread模块中的start_new_thread()函数来产生新线程。语法如下:

```
thread.start_new_thread ( function, args[, kwargs] )
```

参数说明:

function - 线程函数。
args - 传递给线程函数的参数,他必须是个tuple类型。
kwargs - 可选参数。

```
# 为线程定义一个函数
def print_time( threadName, delay):
   count = 0
   while count < 5:
      time.sleep(delay)
      count += 1
      print "%s: %s" % ( threadName, time.ctime(time.time()) )

# 创建两个线程
try:
   thread.start_new_thread( print_time, ("Thread-1", 2, ) )
   thread.start_new_thread( print_time, ("Thread-2", 4, ) )
except:
   print "Error: unable to start thread"

while 1:
   pass
```

### 2.17.3 线程模块

Python通过两个标准库thread和threading提供对线程的支持。thread提供了低级别的、原始的线程以及一个简单的锁。

thread 模块提供的其他方法：

threading.currentThread(): 返回当前的线程变量。
threading.enumerate(): 返回一个包含正在运行的线程的list。正在运行指线程启动后、结束前，不包括启动前和终止后的线程。
threading.activeCount(): 返回正在运行的线程数量，与len(threading.enumerate())有相同的结果。
除了使用方法外，线程模块同样提供了Thread类来处理线程，Thread类提供了以下方法:

run(): 用以表示线程活动的方法。
start():启动线程活动。
join([time]): 等待至线程中止。这阻塞调用线程直至线程的join() 方法被调用中止-正常退出或者抛出未处理的异常-或者是可选的超时发生。
isAlive(): 返回线程是否活动的。
getName(): 返回线程名。
setName(): 设置线程名。

### 2.17.4 使用Threading模块创建线程

使用Threading模块创建线程，直接从threading.Thread继承，然后重写__init__方法和run方法：

```
#coding=utf-8
#!/usr/bin/python

import threading
import time

exitFlag = 0

class myThread (threading.Thread):   #继承父类threading.Thread
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):                   #把要执行的代码写到run函数里面 线程在创建后会直接运行run函数
        print "Starting " + self.name
        print_time(self.name, self.counter, 5)
        print "Exiting " + self.name

def print_time(threadName, delay, counter):
    while counter:
        if exitFlag:
            thread.exit()
        time.sleep(delay)
        print "%s: %s" % (threadName, time.ctime(time.time()))
```

```
                counter -= 1

        # 创建新线程
        thread1 = myThread(1, "Thread-1", 1)
        thread2 = myThread(2, "Thread-2", 2)

        # 开启线程
        thread1.start()
        thread2.start()

        print "Exiting Main Thread"
```

### 2.17.5 线程同步

如果多个线程共同对某个数据修改，则可能出现不可预料的结果，为了保证数据的正确性，需要对多个线程进行同步。
使用Thread对象的Lock和Rlock可以实现简单的线程同步，这两个对象都有acquire方法和release方法，
对于那些需要每次只允许一个线程操作的数据，可以将其操作放到acquire和release方法之间。
如下：
多线程的优势在于可以同时运行多个任务（至少感觉起来是这样）。但是当线程需要共享数据时，可能存在数据不同步的问题。
考虑这样一种情况：一个列表里所有元素都是0，线程"set"从后向前把所有元素改成1，而线程"print"负责从前往后读取列表
并打印。
那么，可能线程"set"开始改的时候，线程"print"便来打印列表了，输出就成了一半0一半1，这就是数据的不同步。
为了避免这种情况，引入了锁的概念。
锁有两种状态——锁定和未锁定。每当一个线程比如"set"要访问共享数据时，必须先获得锁定；
如果已经有别的线程比如"print"获得锁定了，那么就让线程"set"暂停，也就是同步阻塞；等到线程"print"访问完毕，
释放锁以后，再让线程"set"继续。
经过这样的处理，打印列表时要么全部输出0，要么全部输出1，不会再出现一半0一半1的尴尬场面。

实例：

```
#coding=utf-8
#!/usr/bin/python

import threading
import time

class myThread (threading.Thread):
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):
        print "Starting " + self.name
       # 获得锁，成功获得锁定后返回True
       # 可选的timeout参数不填时将一直阻塞直到获得锁定
       # 否则超时后将返回False
        threadLock.acquire()
```

```
        print_time(self.name, self.counter, 3)
        # 释放锁
        threadLock.release()

def print_time(threadName, delay, counter):
    while counter:
        time.sleep(delay)
        print "%s: %s" % (threadName, time.ctime(time.time()))
        counter -= 1

threadLock = threading.Lock()
threads = []

# 创建新线程
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# 开启新线程
thread1.start()
thread2.start()

# 添加线程到线程列表
threads.append(thread1)
threads.append(thread2)

# 等待所有线程完成
for t in threads:
    t.join()
print "Exiting Main Thread"
```

### 2.17.6 线程优先级队列（ Queue ）

Python的Queue模块中提供了同步的、线程安全的队列类，包括FIFO（先入先出)队列Queue，LIFO（后入先出）队列LifoQueue，和优先级队列PriorityQueue。这些队列都实现了锁原语，能够在多线程中直接使用。可以使用队列来实现线程间的同步。

Queue模块中的常用方法:

```
        Queue.qsize() 返回队列的大小
        Queue.empty() 如果队列为空，返回True,反之False
        Queue.full() 如果队列满了，返回True,反之False
        Queue.full 与 maxsize 大小对应
        Queue.get([block[, timeout]])获取队列，timeout等待时间
        Queue.get_nowait() 相当Queue.get(False)
        Queue.put(item) 写入队列，timeout等待时间
        Queue.put_nowait(item) 相当Queue.put(item, False)
        Queue.task_done() 在完成一项工作之后，Queue.task_done()函数向任务已经完成的队列发送一个信号
        Queue.join() 实际上意味着等到队列为空，再执行别的操作
```

示例代码:

```
        #coding=utf-8
        #!/usr/bin/python
```

```
import Queue
import threading
import time

exitFlag = 0

class myThread (threading.Thread):
    def __init__(self, threadID, name, q):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.q = q
    def run(self):
        print "Starting " + self.name
        process_data(self.name, self.q)
        print "Exiting " + self.name

def process_data(threadName, q):
    while not exitFlag:
        queueLock.acquire()
        if not workQueue.empty():
            data = q.get()
            queueLock.release()
            print "%s processing %s" % (threadName, data)
        else:
            queueLock.release()
        time.sleep(1)

threadList = ["Thread-1", "Thread-2", "Thread-3"]
nameList = ["One", "Two", "Three", "Four", "Five"]
queueLock = threading.Lock()
workQueue = Queue.Queue(10)
threads = []
threadID = 1

# 创建新线程
for tName in threadList:
    thread = myThread(threadID, tName, workQueue)
    thread.start()
    threads.append(thread)
    threadID += 1

# 填充队列
queueLock.acquire()
for word in nameList:
    workQueue.put(word)
queueLock.release()

# 等待队列清空
while not workQueue.empty():
    pass

# 通知线程是时候退出
exitFlag = 1

# 等待所有线程完成
```

```
for t in threads:
    t.join()
print "Exiting Main Thread"
```

## 2.18 Python XML解析

## 2.19 Python GUI编程

## 2.20 Python JSON

### 2.20.1 环境配置

在使用 Python 编码或解码 JSON 数据前，我们需要先安装 JSON 模块。本教程我们会下载 Demjson 并安装：

```
http://deron.meranda.us/python/demjson/
$tar xvfz demjson-1.6.tar.gz
$cd demjson-1.6
$python setup.py install
```

### 2.20.2 JSON 函数

```
函数        描述
encode        将 Python 对象编码成 JSON 字符串
decode        将已编码的 JSON 字符串解码为 Python 对象
encode
Python encode() 函数用于将 Python 对象编码成 JSON 字符串。

语法                demjson.encode(self, obj, nest_level=0)
实例

以下实例将数组编码为 JSON 格式数据：

#!/usr/bin/python
import demjson

data = [ { 'a' : 1, 'b' : 2, 'c' : 3, 'd' : 4, 'e' : 5 } ]

json = demjson.encode(data)
print json
以上代码执行结果为：

[{"a":1,"b":2,"c":3,"d":4,"e":5}]
decode
Python 可以使用 demjson.decode() 函数解码 JSON 数据。该函数返回 Python 字段的数据类型。

语法                demjson.decode(self, txt)
实例
```

以下实例展示了Python 如何解码 JSON 对象：

```
#!/usr/bin/python
import demjson

json = '{"a":1,"b":2,"c":3,"d":4,"e":5}';

text = demjson.decode(json)
print  text
```

以上代码执行结果为：

```
{u'a': 1, u'c': 3, u'b': 2, u'e': 5, u'd': 4}
```

### 2.20.3 常用内建函数

```
help     帮助
dir      查看内置类型和函数
int      转换为整形
str      转换为字符串
len      返回元素个数
open     open(name, mode)
range(start,stop,step)   生成列表，返回指定范围元素
raw_input 接受键盘输入，返回字符串
type     返回变量类型
```

# 第 3 章

# 系统编程

# 第 4 章

# 网络服务

# 第 5 章

# GUI图形化

# 第 6 章

# 系统运维

# 第 7 章

# 项目实战

## 7.1 网络爬虫

# 第 8 章

# 速查手册

## 8.1 关键字

```
|_. ID |_.表名 |
| 1 | 项目1 |
| 2 | 项目2 |
```

## 8.2 http协议速查

## 8.3 html速查