python 基础笔记_疯道人 

一、概括

1.python 起源

贵铎.范.罗萨姆(Guid van Rossum)于 1989 年底创始了 python,1991 年初 python 发布了第一个公开发行版,为了更好的完成荷兰 CWI(国家数学和计算机科学研究院)的第一个研究项目而创建.

2.python 的特点

1)高级:有高级的数据结构,缩短开发时间与代码量.
2)面向对象:为数据和逻辑相分离的结构化和过程化编程添加了新的活力.
3)可升级:提供了基本的开发模块,可以在它上面开发软件,实现代码重用.
4)可扩展:通过将其分离为多个文件或模块加以组织管理.
5)可移植性:python 是用 c 写的,又由于 c 的可移植性,使得 python 可以运行在任何带有 ANSIC 编辑器的平台上.
6)易学:python 关键字少,结构简单,语法清晰.
7)易读:没有其他语言通常用来访问变量,定义代码块和进行模式匹配的命令式符号.
8)内存管理器:内存管理是由 python 解释器负责的.

3.python 语法结构

python 代码块通过缩进对齐表达的逻辑而不是使用大括号,缩进表达一个语句属于哪个代码块,缩进风格:4 个空格.

4.缩进及代码组

缩进相同的一组语句构成一个代码块,称之代码组.
首行以关键字开始,以冒号':'结束,该行之后的一行或多行代码构成代码组.
如果代码组只有一行,可以将其直接写在冒号后面,但是这样的写法可读性差,不推荐.

5.注释

和大部分脚本及 unix-shell 语言一样,python 也使用'#'符号表示注释.从#开始,直到一行结束的内容都是注释.
良好的注释习惯可以方便其他人了解程序功能,方便自己在日后读懂代码.

6.文档字符串

可以当做一种特殊的注释,在模块,类或者函数的起始添加一个字符串,起到在线文档的功能.
简单的说明可以使用单引号或双引号,较长的文字说明可以使用三引号.

7.环境准备

下载地址:http://www.python.org/
weindows 版本使用 python.msi
类 unix 系统默认已经安装,或使用源码包
[root@localhost ~]# ./configure
[root@localhost ~]# make && make install

二、基础入门

1.python 运行方法

1)通过交互解释器

```
[root@localhost ~]# python
Python 2.7.5 (default, Feb 11 2014, 07:46:25)
[GCC 4.8.2 20140120 (Red Hat 4.8.2-13)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

2)以文件的方式运行

```
[root@localhost ~]# vim test.py
#!/usr/bin/env python
...
[root@localhost ~]# python test.py
```

3)给文件加权限用 shell 方式运行

```
[root@localhost ~]# chmod +x test.py
[root@localhost ~]# ./test.py
```

2.导入 tab 模块

在交互解释器中,为了实现按<tab>健提示,自动补全功能,可以导入 tab 模块

```
[root@localhost ~]# cp tab.py /usr/lib64/python2.6/site-packages/
[root@localhost ~]# python
Python 2.6.6 (r266:84292, Nov 22 2013, 12:16:22)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import tab
```

3.tab 模块脚本内容

```
#!/usr/bin/env python
import sys
import readline
import rlcompleter
import atexit
import os
readline.parse_and_bind('tab: complete')
histfile = os.path.join(os.environ['HOME'], '.pythonhistory')
try:
    readline.read_history_file(histfile)
except IOError:
    pass
atexit.register(readline.write_history_file,histfile)
del os, histfile, readline, rlcompleter
```

4.vim 编辑器使用

1)在 vim 中实现自动补全字符串功能功能:ctrl+n 或 ctrl+p

2)实现自动缩进,以及防止将 8 个空格转换成制表符

```
[root@localhost ~]# vim ~/.vimrc
```

set ai #启用自动缩进
set expandtab #禁止将 8 个空格转换成制表符

5.编写 python 程序模块示例

```
[root@localhost ~]# vim star.py
#!/usr/bin/env python
'''star module

just a sample module.
only include one function.'''

def pstar():
    'do not accept args.Used to print 50 stars'
    print '*' * 50
```

6.导入模块,查看在线文档

```
[root@localhost ~]# python
Python 2.6.6 (r266:84292, Nov 22 2013, 12:16:22)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import star
>>> help(star)
Help on module star:
NAME
    star - star module
FILE
    /root/star.py
DESCRIPTION
    just a sample module.
    only include one function.
FUNCTIONS
    pstar()
        do not accept args.Used to print 50 stars
(END)
```

7.程序输出

print 将数据输出到屏幕

```
>>> print 'Hello world!'
Hello world!
>>> print 'Hello','world' #逗号拼接字符串时产生一个空格
Hello world
>>> print 'Hello'+'world' #加号拼接字符串时不产生空白或制表符
Helloworld
```

8 程序输入

使用 raw_input()函数读取用户输入数据

```
>>> user = raw_input('Enter your name:')
```

```
Enter your name:bob
>>> print 'Your name is:',user
Your name is: bob
>>> i = raw_input('Input a number:') #raw_input()接受的数据默认为字符串
Input a number:10
>>> i+1 #字符串不能与数字进行运算
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects #不能连接”str”和”int”对象
```

9.程序输入输出脚本

```
#!/usr/bin/env python
username = raw_input('username: ')
print 'Welcome',username
```

10.同行多个语句

分号;允许你将多个语句写在同一行上,但是有些语句不能再这行开始一个新的代码块,因为可读性变差,所以不推荐使用.

```
>>> a = 3;print a
3
>>> x = 3;def foo():print x
  File "<stdin>", line 1
    x = 3;def foo():print x
                  ^
SyntaxError: invalid syntax #无效的语法
>>> def foo():x = 2;print x
...
>>> print foo()
2
None
```

11.注释(#)及续行(\)

首要说明的是:尽管 python 是可读性最好的语言之一,这并不意味着程序员在代码中可以不写注释.和很多 unix 脚本类似,python 注释语句从#字符开始,注释可以在一行的任何地方开始,解释器会忽略掉该行#之后的所有内容.一行过长的语句可以使用反斜杠\分解成几行.

*列表也有续行的效果:

```
#!/usr/bin/env python
my_list = [
      1,{"name":'bob'},
      2,{'name':'alice'}
]
print my_list
print my_list[1]
[root@localhost mypy]# python ttt.py
[1, {'name': 'bob'}, 2, {'name': 'alice'}]
{'name': 'bob'}
```

12.基本风格

编写程序时,应该建立一种统一且容易阅读的结构,并将它应用到每一个文件中去.

```
#!/usr/bin/env python    #起始行,定义解释器
"this is a test module"    #模块文档字符串
import sys #导入模块
import os
debug = True #全局变量声明
class FooClass(object): #类定义
        'Foo class'
        pass
def test(): #函数定义
        'test function'
        foo = FooClass()
if __name__ == '__main__': #程序主体
        test()
```

13.主程序中的测试代码

优秀的程序员和软件工程师,总是会为其应用程序提供一组测试代码或者简单教程.测试代码仅当该文件被直接执行时运行,即被其他模块导入时不执行.利用__name__变量这个有利条件,将测试代码放在一个函数中,如果该模块是被当成脚本运行,就调用这个函数.

14.客户端 vnc 使用

```
[root@localhost Desktop]# yum -y install tigervnc
[root@localhost Desktop]# vncviewer &
```

三、运算符

1.算术运算符

1)单目运算符:

正号 +

负号 -

2)双目运算符:

+ 加

- 减

* 乘

/ 除

% 取余

** 幂运算

从 python2.2 起,还增加了一种新的整除运算符//,即地板除

// 地板除(对结果进行四舍五入)

```
>>> 3/2
1
>>> print 3.0/2
1.5
>>> 3//2
1
>>> print 3.0//2
```

1.0
>>> print round(3.0/2)
2.0

* round(flt, ndig=0) 接受一个浮点数 flt 并对其四舍五入，保存 ndig 位小数。若不提供 ndig 参数，则默认小数点后 0 位。
>>> round(3.1415926,2)
3.14
>>> round(3.1415926)#默认为 0
3
>>> round(3.1415926,-2)
0.0
>>> round(3.1415926,-1)
0.0
>>> round(314.15926,-1)
310.0

2.标准类型运算符
在做数值运算时,必须努力保证操作数是合适的类型.相同类型数值运算,得到结果也是该类型的数值.不同类型数值运算,需要(显示或隐式的)做数值类型的转换.
>>> print 5/3
1
>>> print 5.0/3
1.66666666667
>>> print 5. / 3
1.66666666667
>>> print 5. / float(3)
1.66666666667
>>> 10 + 999999999999999999999999
1000000000000000000000000009L
>>> int(10L)
10
>>> long(10)
10L
>>> float(10)
10.0

3.比较运算符
< 小于
<= 小于等于
> 大于
>= 大于等于
== 等于
!= 不等于
<> 不等于(将被淘汰)

4.布尔值

True False

5.布尔逻辑运算符

not:非 and:与 or:或

not 运算符拥有最高优先级,只比所有比较运算符低一级,and 和 or 运算符比相应的在低一级.

not expr:expr 的逻辑非(否)

expr1 and expr2:expr1 和 expr2 的逻辑与

expr1 or expr2:expr1 和 erpr2 的逻辑或

6.位运算符

位运算符只适用于整数

1)~num:单目运算,对数的每一位取反

2)num1<<num2:num1 左移 num2 位

```
>>> 5 << 2
20
>>> bin(5) #转为 2 进制左移两位右面补零
'0b101'
>>> 0b10100
20
```

3)num1>>num2:num1 右移 num2 位

```
>>> 5 >> 2
1
>>> bin(5) #转为 2 进制,右移两位,去除后两位
'0b101'
>>> 0b1
1
```

4)num1&num2:num1 与 num2 按位与

5)num1^num2:num1 异或 num2

6)num1|num2:num1 与 num2 按位或

四、数字

1.数字简介

创建数值对象并赋值,数字提供了标量贮存和直接访问,创建数值对象和给变量赋值一样简单

```
>>> anlnt = 1
>>> anlnt
1
>>> aLong = -9999999999999999999999
>>> aLong
-9999999999999999999999L
>>> aFloat = 3.1415926923840129132
>>> aFloat
3.141592692384013
>>> aComplex = 1.23+2.56j
>>> aComplex
```

(1.23+2.5600000000000001j)

2.基本数字类型

int:有符号整数
log:长整数
bool:布尔值(True:1,False:0)
float:浮点数
complex:复数

3.更新数字对象

数字是不可更改类型,也就说变更数字的值会生成新的对象.在 python 中,变量更像一个指针指向装变量值的盒子.对不可改变类型来说,你无法更改盒子的内容,但你可以将指针指向一个新盒子.

```
>>> i = 3
>>> id(i)
29340424
>>> i = 4
>>> id(i)
29340400
```

4.删除数字对象

按照 python 的法则,无法真正删除一个数值对象,仅仅是不在使用它而已.删除一个数值对象的引用,使用 del 语句.删除对象的引用之后,就不能再使用这个引用(变量名),除非给他赋一个新值.绝大多数情况下并不需要明确 del 一个对象.

```
>>> a = 1
>>> a
1
>>> del a
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined #名字' a '没有定义
```

4.布尔型

该类型的取值只有两个,即 True 和 False,True 的值为 1,False 的值为 0.
在判断语句中,空列表,空元组,空字符串,空字典,数值为 0 的数字以及 None 皆为 False,其他皆为 True.

```
>>> True+2
3
>>> if 'hello':print 'yes'
...
yes
>>> if not []:print 'yes'
...
yes
```

5.标准整数类型

标准整数类型是最通用的数字类型,python 标准整数类型等价于 C 的(有符号)长整型.
python 默认以十进制数显示

数字以 0 开头表示 8 进制数
数字以 0x 或 0X 开头表示 16 进制数
数字以 0b 或 0B 开头表示 2 进制数

```
>>> 11 #十进制
11
>>> 011 #八进制
9
>>> 0x11 #十六进制
17
>>> 0b11 #二进制
3
```

6.查看文件权限及 py 修改权限

```
[root@localhost ~]# ll
rwx rw- r-- root root star.py
r->4
w->2
x->1
[root@localhost ~]# python
>>> import os
>>> os.chomd('star.py',0755) #用八进制数修改权限
```

7.长整型

C 或其他编译型语言的长整数类型的取值范围是 32 位或 64 位.python 的长整数类型能表达的数值仅仅与机器支持的(虚拟)内存大小有关.在一个整数值后面加个 L(大小写随意),表示这个整数是长整数.这个整数可以是十进制,八进制,或十六进制.

8.内建函数

1)标准类型函数

cmp(num1,num2)

num1 大于 num2 结果为正值
num1 小于 num2 结果为负值
num1 等于 num2 结果为 0

```
>>> cmp(10,2)
1
>>> cmp(10,20)
-1
>>> cmp(10,10)
0
>>> cmp('x','abc')
1
```

str(num):将 num 转换成字符串表示格式

type(obj):判断 obj 类型

2)数字类型函数:将其他数值类型转换为相应的数值类型

int():转为整数

long():转为长整数

float():转为浮点数
complex():转为复数
abs() 返回给定参数的绝对值

```
>>> abs(10)
10
>>> abs(-10)
10
```

divmod():内建函数把除法和取余运算结合起来,返回一个包含商和余数的元组

```
>>> 10/3
3
>>> 10%3
1
>>> divmod(10,3)
(3, 1)
>>> x,y = divmod(10,3)
>>> x
3
>>> y
1
```

pow():进行指数运算

```
>>> pow(2,3)
8
>>> 2 ** 3
8
```

round():用于对浮点数进行四舍五入运算

```
>>> print 5.0 / 3
1.66666666667
>>> print round(5.0 / 3)
2.0
>>> print round(5.0 / 3 , 2) #逗号后边表示保留的小数位
1.67
```

3)仅用于整数的函数
hex():转换为字符串形式的 16 进制数

```
>>> 0x11
17
>>> hex(17)
'0x11'
```

oct():转化为字符串形式的 8 进制数

```
>>> oct(10)
'012'
```

bin():转化为字符串形式的 2 进制数

```
>>> bin(10)
'0b1010'
```

ord():接受一个字符,返回其对应的 ASCII 值

```
>>> ord('a')
97
```

chr():接受一个单字节 ASCII 码整数值,返回一个字符串

```
>>> chr(65)
'A'
```

9.有关金融的精确计算需要使用 decimal 模块

```
>>> a = 0.1
>>> a + a + a - 0.3
5.551115123125783e-17
>>> import decimal
>>> a = decimal.Decimal('0.1')
>>> a + a + a - decimal.Decimal('0.3')
Decimal('0.0')
```

10.operator 提供了一系列的函数操作

```
>>> import operator
>>> operator.add(3,10) #加法运算
13
>>> operator.sub(10,5) #加法运算
5
>>> operator.mul(20,4) #乘法运算
80
>>> operator.div(20,4) #除法运算
5
```

五、变量

1.变量定义

变量名称约定:第一个字符只能是大小写字母或下划线,后续字符智能功能是大小写字母或数字或下划线,区分大小写,python 是动态类型语言,即不需要预先声明变量的类型.

2.标识符

python 标识符字符串规则和其他大部分用 C 编写的高级语言相似

首字符:字母,下划线

其他字符:字母,下划线,数字

区分大小写

3.变量赋值

1)标准赋值

变量的类型和值在赋值的那一刻别初始化,变量赋值通过等号来执行.python 语言中,等号'='是主要的赋值运算符.赋值并不是直接将一个赋值给一个变量.在 python 中,对象是通过引用传递的,在赋值时,不管这个对象是新创建的,还是一个已经存在的,都是将该对象的引用(并不是值)赋值给变量.

```
>>> counter = 0
```

```
>>> name = 'bob'
```
2)链式多重赋值
```
>>> x=y=1
>>> x
1
>>> y
1
```
3)增量赋值:从 python2.0 开始,等号可以和一个算术运算符组合在一起,将计算结果重新赋值给左边的变量,这被称为增量赋值
```
>>> x = 1
>>> x = x + 1
>>> x
2
>>> x += 1
>>> x
3
>>> x++
  File "<stdin>", line 1
    x++
      ^
SyntaxError: invalid syntax #无效的语法
>>> ++x  写前面时只是表示正负号
1
>>> --x
1
>>> -x
-1
```
4)多元赋值:另一种将多个变量同时赋值的方法称为多元赋值,采用这种方式赋值时,等号两边的对象都是元组.
```
>>> x,y,z = 1,2,'a string'
>>> print 'x=%d,y=%d' % (x,y)
x=1,y=2
>>> x,y = y,x
>>> print 'x=%d,y=%d' % (x,y)
x=2,y=1
```

4.关键字

和其他的高级语言一样,python 也拥有一些被称作关键字的保留字符,任何语言的关键字应该保持相对的稳定,但是因为 python 是一门不对成长和进化的语言,其关键字偶尔会更新.关键字列表和 iskeyword()函数都放入了 keyword 模块以便查阅.
```
>>> import keyword
>>> keyword.iskeyword('if')
True
>>> keyword.kwlist
['and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'exec', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'not', 'or', 'pass', 'print', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

5.内建函数

除了关键字之外,python 还有可以在任何一级代码使用的‘内建’的名字集合,这些名字可以由解释器设置或使用,虽然 built-in 不是关键字,但是应该把他当做‘系统保留字’,保留的常量如:True,False,None.

```
>>> len
<built-in function len>
>>> len('adfjie')
6
>>> len = 10
>>> len('adfjie')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'int' object is not callable
>>> del len
>>> len
<built-in function len>
>>> len('adfjie')
6
```

6.99 乘法表例子

1)标准版

```
#!/usr/bin/env python
for i in range(1,10):
    for j in range(1,i + 1):
        print "%sX%s=%s" % (j,i,i*j),
    print
```

2)升级版乘法表可以给定数字

```
#!/usr/bin/env python
def mtable(num = 9):
    for i in range(1,num+1):
        for j in range(1,i + 1):
            print "%sX%s=%s" % (j,i,i*j),
        print
if __name__ == "__main__":
    mtable()
```

```
>>> import mtable2
>>> mtable2.mtable(5)
1X1=1
1X2=2 2X2=4
1X3=3 2X3=6 3X3=9
1X4=4 2X4=8 3X4=12 4X4=16
1X5=5 2X5=10 3X5=15 4X5=20 5X5=25
```

7.创建文件脚本

1)初始版

```
#!/usr/bin/env python
import os
content = []
```

```
while True:
    fname = raw_input('filename:')
    if not os.path.exists(fname):
        break
    print '%s alread exists.Try again.' % fname

while True:
    data = raw_input('enter line(. to quit)>')
    if data == '.':
        break
    content.append(data + '\n')
fobj = open(fname,'w')
fobj.writelines(content)
fobj.close()
```

2)用格式化字符串的方法加入换行符

```
#!/usr/bin/env python
import os
content = []
while True:
    fname = raw_input('filename:')
    if not os.path.exists(fname):
        break
    print '%s alread exists.Try again.' % fname

while True:
    data = raw_input('enter line(. to quit)>')
    if data == '.':
        break
    content.append(data)
fobj = open(fname,'w')
fobj.writelines(['%s\n' % x for x in content])
fobj.close()
```

8.列表解析方式给每个选项加上相同的字符

```
>>> content = ['hello world!','greet!']
>>> ['%s\n' % x for x in content]
['hello world!\n', 'greet!\n']
```

六、字符串

1.定义字符串

python 中字符串被定义为引号之间的字符集合,python 支持使用成对的单引号或双引号,无论单引号还是双引号,表示的意义相同.

python 还支持三引号(三个连续的单引号或者双引号),可以用来包含特殊字符.

python 不区分字符和字符串.

2.字符串切片(切片操作符:[],[:],[::])

使用索引运算符[]和切片运算符[:]可得到子字符串.第一个字符串的索引是 0,最后一个字符的索引是-1.字符串包含切片中的起始下标,但不包括结束下标.

```
>>> pyStr = 'python'
>>> pyStr[:]
'python'
>>> pyStr[0]
'p'
>>> pyStr[1]
'y'
>>> pyStr[-1]
'n'
>>> pyStr[-2]
'o'
>>> pyStr[2:4]
'th'
>>> pyStr[2:]
'thon'
>>> pyStr[:4]
'pyth'
>>> pyStr[::2] #步长为 2,隔一个字符取一个,取奇数
'pto'
>>> pyStr[1::2] #取偶数
'yhn'
>>> pyStr[::-1] #将字符串反转取值
'nohtyp'
>>> pyStr[::-2] #将字符串反转取值,取奇数值
'nhy'
>>> astr = "[1,2,3]"
>>> astr[0]
'['
>>> astr[1]
'1'
>>> astr[2]
','
```

3.字符串连接操作

1)使用+号可以将多个字符串拼接在一起,使用*号可以将字符串重复多次.

```
>>> pyStr = 'python'
>>> isCool = 'is cool'
>>> print pyStr +' '+ isCool
python is cool
>>> pyStr * 2
'pythonpython'
```

2)比较操作符:字符串大小按 ASCII 码值大小进行比较

```
>>> 'a' > 'b'
```

```
False
>>> ord('a')
97
>>> ord('b')
98
```

3)成员关系操作符:in,not in

```
>>> py_str= 'Python'
>>> 't' in py_str
True
>>> 'th' in py_str
True
>>> 'to' in py_str
False
```

4.格式化操作符

字符串可以使用格式化符号类表示特定含义.

%c 转换成字符

```
>>> '%c' % 97
'a'
>>> '%c' % 66
'B'
```

%s 优先使用 str()函数进行字符串转换

```
>>> '%s' % 100
'100'
>>> '%s %s' % ([12,34],30)
'[12, 34] 30'
```

%d/%i 转换成有符号十进制数

```
>>> '%d' % 100
'100'
```

%o 转换成无符号八进制数

```
>>> '%o' % 10
'12'
>>> '%#o' % 10
'012'
>>> '%#x' % 10 #十六进制
'0xa'
```

%e/%E 转换成科学计数法

```
>>> '%e' % 1000000000
'1.000000e+09'
```

%f/%F 转换成浮点数

```
>>> '%f' % (5. /3)
'1.666667'
>>> '%4.2f' % (5. / 3)
'1.67'
```

5.格式化操作符辅助指令

```
>>> print 'His %s is %d' % ('age',23)
His age is 23
```

1)*:定义宽度或者小数点精度

```
>>> print 'Your name is %*s' % (5,'bob')
Your name is      bob
>>> print 'Your name is %*s' % (10,'bob')
Your name is                bob
```

2)-:左对齐

```
>>> print '%8s:%8s' % ('name','age')
       name:         age
>>> print '%-8s:%8s' % ('name','age')
name       :         age
>>> '%10s%5s' % ('name','age')
'           name  age'
>>> '%-10s%-5s' % ('name','age')
'name           age   '
>>> '%-*s%-*s' % (8,'name',4,'age')
'name       age '
```

3)+:在正数前面显示加好

```
>>> '%+d' % 10
'+10'
>>> '%+d' % -10
'-10'
```

4)<sp>:在正数前面显示空格

5)#:在八进制数前面显示零 0,在十六进制前面显示‘0x’或者‘0X’

```
>>> print 'int %d is hex %#x' % (10,10)
int 10 is hex 0xa
```

6)0:显示的数字前面填充 0 而不是默认的空格

```
>>> '%5s' % 10
'    10'
>>> '%05d' % 10
'00010'
```

6.字符串模板

string 模块提供了一个 Template 对象,利用该对象可以实现字符串模板的功能

```
>>> import string
>>> origTxt = 'Hi ${name},I will see you ${day}'
>>> t = string.Template(origTxt)
>>> t.substitute(name='bob',day='tomorrow')
'Hi bob,I will see you tomorrow'
>>> t.substitute(name='tom',day='the day after tomorrow')
'Hi tom,I will see you the day after tomorrow'
```

7.原始字符串操作符:r

原始字符串操作符是为了对付那些在字符串中出现的特殊字符,在原始字符串里,所有的字符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.

```
>>> winPath = 'c:\windows\temp'
>>> winPath
'c:\\windows\temp'
>>> print winPath
c:\windows          emp
>>> newPath = r'c:\windows\temp'
>>> newPath
'c:\\windows\\temp'
>>> print newPath
c:\windows\temp
```

8.内建函数

string.capitalize():把字符串的第一个字母大写.

```
>>> 'hello world!'.capitalize()
'Hello world!'
```

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.

```
>>> print '|'+'hello world!'.center(20)+'|'
|       hello world!       |
```

string.count(str,beg=0,end=len(string)):返回 str 在 string 里面出现的次数,如果 beg 或者 end 指定则返回指定范围内 str 出现的次数

```
案例空缺
```

string.endswith(obj,beg=0,end=len(string)):检查字符串实否以 obj 结束,如果 beg 或 end 指定则检查指定的范围内是否以 obj 结束,如果是返回 True,否则返回 False

```
>>> 'hello world!'.endswith('d!')
True
>>> 'hello world!'.endswith('hello')
False
```

string.islower():如果 string 中包含至少一个区分大小写字符,并且所有这些字符都是小写,则返回 True,否则返回 False

```
>>> 'hello world!'.islower() #检测字符串是否由小写字母组成
True
>>> 'hello world!'.isupper() #检测字符串中所有的字母是否都为大写
False
>>> '1423'.isdigit() #检测字符串是否只由数字组成
True
>>> '1423k'.isdigit()
False
```

string.strip():删除 string 字符串两段的空白

```
>>> '\t hello world\n'.strip()
'hello world'
>>> '\t hello world\n'.lstrip() #删除左边空白
'hello world\n'
>>> '\t hello world\n'.rstrip() #删除右边空白
'\t hello world'
```

string.upper():转换 string 中的小写字母为大写

```
>>> 'hello'.upper()
'HELLO'
>>> 'HELLO'.lower() #转大写为小写
'hello'
```

string.split(str='',num=string.count(str)):以 str 为分隔符切片 string,如果 num 有指定值,则仅分割 num 个自字符串

```
>>> 'mytest.tar.gz'.split('.')
['mytest', 'tar', 'gz']
>>> testStr = '''hello everyone.
... welcome to python.
... it is a great language.'''
>>> testStr.splitlines() #按行分隔
['hello everyone.', 'welcome to python.', 'it is a great language.']
```

9.内建函数综合练习

```
>>> hi = 'hello world'
>>> hi.capitalize() #字符串开头转为大写
'Hello world'
>>> hi.center(30) #共占 30 个字符,以空白填充
'           hello world           '
>>> hi.center(30,'+') #共占 30 个字符,以加号填充
'+++++++++hello world++++++++++'
>>> hi.rjust(30) #共占 30 个字符,开始以空白填充
'                    hello world'
>>> hi.ljust(30) #共占 30 个字符,结尾以空白填充
'hello world                    '
>>> hi.count('o') #字符 o 重复出现几次
2
>>> hi.count('l')
3
>>> hi.count('ll')
1
>>> hi.count('ll',5) #从第五个下标开始搜索
0
>>> hi.endswith('d') #是否以 d 结尾
True
>>> hi.endswith('ld')
True
>>> hi.startswith('he') #是否以 he 开头
True
>>> hi.startswith('ha')
False
>>> hi.islower() #字符串全是小写
True
>>> 'abc123'.islower()
True
>>> 'Abc123'.islower()
```

```
False
>>> hi.isupper() #字符串全是大写
False
>>> hi.upper() #全部转为大写
'HELLO WORLD'
>>> hi
'hello world'
>>> 'HELLO'.lower() #全部转为小写
'hello'
>>> astr = '    hello world! \n'
>>> astr.strip() #去除两段空白
'hello world!'
>>> astr.lstrip() #去除左端空白
'hello world! \n'
>>> astr.rstrip() #去除右端空白
'    hello world!'
>>> astr.rstrip(' \n!') #结尾是空格\n!都去掉
'    hello world'
>>> hi.split() #默认以空格分隔
['hello', 'world']
>>> 'home.tar.gz'.split('.') #以.分隔
['home', 'tar', 'gz']
>>> hi.replace('o','a') #将 o 替换成 a，默认全替换
'hella warld'
>>> hi.replace('l','a')
'heaao worad'
>>> hi.replace('l','a',1) #第三个是替换的次数
'healo world'
>>> hi.replace('l','a',2)
'heaao world'
```

10.检查标识符是否可用脚本

1)标准版

```
#!/usr/bin/env python
import string
first_chs = string.letters + '_'
all_chs = first_chs + string.digits
my_id = raw_input('Id tu check: ')
if len(my_id) < 2:
        print 'You must input id longer than 2.'
else:
        if my_id[0] not in first_chs:
                print '1st char invalid.'
        else:
                for ch in my_id[1:]:
                        if ch not in all_chs:
```

```
                print 'other char invalid.'
                break
        else:
            print '%s is valid' % my_id
```

2)检查标识符是否可用脚本,升级版用函数方式

```
#!/usr/bin/env python
import string
first_chs = string.letters + '_'
all_chs = first_chs + string.digits
def check_id(my_id):
    if len(my_id) < 2:
        print 'You must input id longer than 2.'
    else:
        if my_id[0] not in first_chs:
            print '1st char invalid.'
        else:
            for i in range(len(my_id[1:])):
                ind = i + 1
                if my_id[ind] not in all_chs:
                    print 'char in position %s inval.' % ind
                    break
            else:
                print '%s is valid' % my_id
if __name__ == "__main__":
    check_id('a')
    check_id('1aaa')
    check_id('ass@3d')
    check_id('abcd')
```

11.格式化输出的脚本

```
#!/usr/bin/env python
width = 48
content = []
while True:
    line = raw_input('enter line(.to quit)>')
    if line == '.':
        break
    content.append(line)
print '+%s+' % ('*' * width)
for line in content:
    sp_wid,extra = divmod((width - len(line)) , 2)
    print '+%s%s%s+' % (' ' * sp_wid,line,' ' * (sp_wid + extra))
print '+%s+' % ('*' * width)
[root@localhost ~]# python qwer.py
enter line(.to quit)>qwer
enter line(.to quit)>qeace
```

```
enter line(.to quit)>.
+**********************************************+
+                                qwer                                +
+                               qeace                                +
+**********************************************+
```

12.center 函数可以简化脚本

```
>>> 'hello'.center(20)
'            hello               '
```

13.创建用户,并发送邮件脚本

```
#!/usr/bin/env python
import os
import sys
import randpass2
import string

content = """username: ${user}
password: ${password}"""
t = string.Template(content)

def adduser(username,pwd):
        os.system('useradd %s' % username)
        os.system('echo %s | passwd --stdin %s' % (pwd,username))
        os.system('echo -e "%s" | mail -s "user info" root' % (t.substitute(user=username,password=pwd)))

if __name__ == '__main__':
        if len(sys.argv) != 2:
                print 'Usage: %s username' % sys.argv[0]
        else:
                adduser(sys.argv[1],randpass2.gen_pass())
```

14.strip 替代函数,消除空白脚本

```
#!/usr/bin/env python

white_spaces = ' \n\t\f\v\r'
def rm_lsp(astr):
        if astr == '':
                return astr

        for ind in range(len(astr)):
                if astr[ind] not in white_spaces:
                        break
        else:
                return ''
        return astr[ind:]
```

```
if __name__ == '__main__':
    str1 = '    \thello    '
    str2 = ''
    str3 = '          '
    print '|%s|' % rm_lsp(str1)
    print '|%s|' % rm_lsp(str2)
    print '|%s|' % rm_lsp(str3)
```

七、列表

1.定义列表

可以将列表当成普通的‘数组’,它能保存任意数量任意类型的 python 对象,像字符串一样,列表也支持下标和切片操作,列表中的项目可以改变.

```
>>> aList = [1,'tom',2,'alice']
>>> aList[1]
'tom'
>>> aList[1][1]
'o'
>>> aList[1] = 'bob'
>>> aList
[1, 'bob', 2, 'alice']
>>> aList[2:]
[2, 'alice']
```

2.创建及访问列表

列表是有序,可变的数据类型,列表中可包含不同类型的对象,列表可有由[]或工厂函数创建,支持下标及切片操作.

```
>>> aList = [1,2,'hello']
>>> bList = list('new')
>>> print aList
[1, 2, 'hello']
>>> print bList
['n', 'e', 'w']
```

3.更新列表

1)使用 in 或 not in 判断成员关系,

```
>>> aList = [1,'tom',2,'alice']
>>> 'tom' in aList
True
>>> 'alice' not in aList
False
```

2)通过下标只能更新值,不能使用下标添加新值,可以使用 append 方法添加新值.

```
>>> aList = ['hello','world']
>>> aList[0] = 'hi'
>>> aList
['hi', 'world']
>>> aList[2] = 'new'
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range #索引错误
>>> aList.append('new')
>>> aList
['hi', 'world', 'new']
>>> aList.append('old')
>>> aList
['hi', 'world', 'new', 'old']
>>> aList[1:3] = ['tom','lili']
>>> aList
['hi', 'tom', 'lili', 'old']
>>> aList[2:2] = [100,'abc'] #在下标 2 的位置添加新内容
>>> aList
['hi', 'tom', 100, 'abc', 'lili', 'old']
```

4.删除列表

可以使用 del 删除列表项或整个列表,删除列表项可以使用 pop()及 remove()方法.

```
>>> aList = ['hello','world','new','list','name']
>>> aList.pop() #默认弹出最后一项,并返回该值 pop()可指定下标
'name'
>>> aList
['hello', 'world', 'new', 'list']
>>> aList.pop(2)
'new'
>>> del aList[2]
>>> aList
['hello', 'world']
>>> aList.remove('world')
>>> aList
['hello']
>>> del aList
>>> aList
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'aList' is not defined #名字'aList'没有定义
```

5.列表操作进阶

由于列表也是序列类型,所以+,*,in,not in 都适用于列表,但是需要注意参与运算的对象属于同一类型

```
>>> ['hello','world'] * 2
['hello', 'world', 'hello', 'world']
>>> ['hello','world'] + 'new'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate list (not "str") to list #只能连接列表(不是”str”)
>>> ['hello','world'] + ['new']
['hello', 'world', 'new']
```

6.作用于列表的函数

与字符串类似,列表也支持如下函数

len() 长度

max() 最大值

min() 最小值

sorted() 排序

enumerate() 对应下标和值

sum() 求和

zip() 使每项变为列表元祖

```
>>> aList = [23,345,656]
>>> zip(aList)
[(23,), (345,), (656,)]
```

7.列表内建函数

list.append(obj):向列表中添加一个对象 obj

list.count(obj):返回一个对象 obj 在列表中出现的次数

list.extend(seq):把序列 seq 的内容添加到列表中

```
>>> aList = ['tom',26,'lili',20]
>>> aList.extend('new')
>>> aList
['tom', 26, 'lili', 20, 'n', 'e', 'w']
>>> aList.extend(('new','world'))#追加单个词需是元组,可以写成 aList.extendf(('ae',))
>>> aList
['tom', 26, 'lili', 20, 'n', 'e', 'w', 'new', 'world']
```

list.index(obj):返回 obj 对象的下标

```
>>> aList.index(26)
1
```

list.insert(index,obj):在索引量为 index 的位置插入对象 obj

list.reverse():原地翻转列表

list.sort():排序

```
>>> import random 打乱顺序
>>> random.shuffle(aList)
>>> aList
['ae', 'world', 'tom', 26, 'w', 'n', 'e', 'lili', 'new', 20]
```

8.用列表构建栈结构

```
#!/usr/bin/env python

stack = []

def pushit():
    item = raw_input('Item to push: ')
    stack.append(item)

def popit():
```

```
    if stack:
        print "popped item: ",stack.pop()
    else:
        print "Empty stack."

def viewit():
    print stack

CMDs = {'0':pushit,'1':popit,'2':viewit}

def show_menu():
    prompt = """(0)push tack
(1)pop stack
(2)view stack
(3)quit
Please input your choice(0/1/2/3): """
    while True:
        choice = raw_input(prompt).strip()[0]
        if choice not in '0123':
            print "Invalid choice.Try again."
            continue
        if choice == '3':
            break
        CMDs[choice]()
if __name__ == '__main__':
    show_menu()
```

9.列表版随机密码

```
#!/usr/bin/env python
import string
import random

all_chs = string.letters + string.digits

def gen_pass(num = 8):
    pwd = []
    for i in range(num):
        pwd.append(random.choice(all_chs))
    return ''.join(pwd)
if __name__ == '__main__':
    print gen_pass()

>>> aList = list('hello')
>>> aList
['h', 'e', 'l', 'l', 'o']
>>> '-'.join(aList)
```

```
'h-e-l-l-o'
>>> '.'.join(aList)
'h.e.l.l.o'
>>> ''.join(aList)
'hello'
>>> '.'.join([])
''
```

10.去除尾部空白脚本

```
#!/usr/bin/env python

white_spaces = ' \n\t\v\f\r'
def rm_rsp(astr):
    if astr == '':
        return astr
    alist = list(astr)
    while alist:
        if alist[-1] in white_spaces:
            alist.pop()
        else:
            break
    return ''.join(alist)
if __name__ == '__main__':
    str1 = '      hello        '
    print '|%s|' % rm_rsp(str1)
```

八、序列

1.序列类型操作符

seq[ind]:获得下标为 ind 的元素
seq[ind1:ind2]:获得下标从 ind1 到 ind2 间的元素集合
seq * expr:序列重复 expr 次
seq1 + seq2:连接序列 seq1 和 seq2
obj in seq:判断 obj 元素是否包含在 seq 中
obj not in seq:判断 obj 元素是否不包含在 seq 中

2.内建函数

list(iter) 把可迭代对象转换为列表

```
>>> list('hello')
['h', 'e', 'l', 'l', 'o']
>>> list(('hello','world'))
['hello', 'world']
>>> list(('hello'))
['h', 'e', 'l', 'l', 'o']
```

str(obj) 把 obj 对象转换为字符串

```
>>> str(['hello','world'])
"['hello', 'world']"
>>> str(10)
'10'
```

tuple(iter) 把一个可迭代对象转换成一个元组对象

```
>>> tuple('hello')
('h', 'e', 'l', 'l', 'o')
len(seq):返回 seq 长度
>>> len('abcd')
4
>>> len(['hello','world'])
2
>>> len(('hello','world'))
2
```

max(iter,key=None):返回 iter 中的最大值

```
>>> max('abcd')
'd'
>>> max(1,34,54,234)
234
>>> max((1,34,54,234))
234
>>> min('asdf') #最小值
'a'
```

enumerate:接受一个可迭代对象作为参数,返回一个 enumerate 对象(返回下标和元素)

```
>>> aList = ['hello','world']
>>> for i,j in enumerate(aList):
...         print 'index %d:%s' % (i,j)
...
index 0:hello
index 1:world
>>> enumerate('abc')
<enumerate object at 0x2599af0>
>>> for i,j in enumerate('abc'):
...         print "#%s:%s" % (i,j)
...
#0:a
#1:b
#2:c
```

reversed(seq):接受一个序列作为参数,返回一个以逆序访问的迭代器

```
>>> aList = [32,43,323,55]
>>> reversed(aList)
<listreverseiterator object at 0x7fdd973de310>
>>> for item in reversed(aList):
...         print item
...
55
```

```
323
43
32
>>> a = [23,43,134,22,2]
>>> reversed(a)
<listreverseiterator object at 0x2356f90>
>>> for i in reversed(a):
...         print i,
...
2 22 134 43 23
```

sorted(iter):接受一个可迭代对象作为参数,返回一个有序的列表

```
>>> aList = [32,43,323,55]
>>> sorted(aList) #排序从小到大
[32, 43, 55, 323]
>>> a = [23,43,134,22,2]
>>> a
[23, 43, 134, 22, 2]
>>> sorted(a)
[2, 22, 23, 43, 134]
```

九、元祖

1.元组的定义及操作

可以认为元组是‘静态’的列表,元组一旦定义,不能改变.

```
>>> aTuple = (1,'tom',2,'alice')
>>> aTuple[1]
'tom'
>>> 'tom' in aTuple
True
>>> aTuple[0] = 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment #tuple 对象不支持项目分配
```

2.创建元组

通过()或工厂函数 tuple()创建元组,元组是有序的不可变类型,与列表类似,作用于列表的操作,绝大多数也可以作用于元组.

```
>>> aTuple = ('one','two','three')
>>> bTuple = tuple(['hello','world'])
>>> print aTuple
('one', 'two', 'three')
>>> print bTuple
('hello', 'world')
>>> atuple = tuple('hello')
>>> atuple
('h', 'e', 'l', 'l', 'o')
>>> ''.join(atuple) #join 用于拼接
```

'hello'

3.元组操作符

由于元组也是序列类型,所以可以作用在序列上的操作都可以作用于元组,通过 in,not in 判断成员关系.

```
>>> bTuple
('hello', 'world')
>>> 'hello' in bTuple
True
```

4.元组特性

单元素元组:如果一个元组中只有一个元素,那么创建该元组的时候,需要加上一个逗号.

```
>>> cTuple = ('hello')
>>> print cTuple
hello
>>> type(cTuple)
<type 'str'>
>>> dTuple = ('hello',)
>>> print dTuple
('hello',)
>>> type(dTuple)
<type 'tuple'>
```

5.更新元组

虽然元组本事是不可变的,但是因为它同时属于容器类型,也就意味着元组的某一个元素是可变的容器类型,那么这个元素中的项目仍然可变

```
>>> aTuple = ('bob',['bob@tarena.com','beijing'])
>>> aTuple[0] = 'tom'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment #元组对象不支持
>>> aTuple[1][1] = 'shanghai'
>>> print aTuple
('bob', ['bob@tarena.com', 'shanghai'])
>>> aTuple.count('bob') #统计出现的次数
1
>>> aTuple.index('bob') #返回下标
0
```

十、字典

1.字典的定义及操作

1)字典是有键-值(key-value)对构成的映射数据类型,通过键取值,不支持下标操作.

```
>>> userDict = {'name':'bob','age':23}
>>> userDict
{'age': 23, 'name': 'bob'}
>>> userDict['name']
'bob'
```

```
>>> userDict[name]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'name' is not defined
>>> userDict['gender'] = 'male'
>>> userDict
{'gender': 'male', 'age': 23, 'name': 'bob'}
>>> 'name' in userDict
True
>>> userDict[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 0
>>> userDict[5]= 'fea'
>>> userDict
{'age': 23, 'name': 'bob', 5: 'fea'}
>>> userDict[5]
'fea'
```

2)通过{}操作符创建字典,通过 dict()工厂方法创建字典,通过 fromkeys()创建具有相同值的默认字典.

```
>>> dict()
{}
>>> aDict = {'name':'bob','age':23}
>>> aDict
{'age': 23, 'name': 'bob'}
>>> bDict = dict((['name','bob'],['age',23]))
>>> bDict
{'age': 23, 'name': 'bob'}
>>> cDict = {}.fromkeys(('bob','alice'),23)
>>> cDict
{'bob': 23, 'alice': 23}
>>> print '%(alice)s' % bdict
23
```

2.访问字典

字典是映射类型,意味着它没有下标,访问字典中的值需要使用相应的键

```
>>> aDict = {'name':'bob','age':23}
>>> for eachKey in aDict:
...         print 'key=%s,value=%s' % (eachKey,aDict[eachKey])
...
key=age,value=23
key=name,value=bob
>>> print '%(name)s' % aDict
bob
```

3 更新字典

通过键更新字典,如果字典中有该键,则更新相关值.如果字典中没有该键,则向字典中添加新值.

```
>>> print aDict
{'age': 23, 'name': 'bob'}
>>> aDict['age'] = 22
>>> print aDict
{'age': 22, 'name': 'bob'}
>>> aDict['email'] = 'bob@tarena.com.cn'
>>> print aDict
{'age': 22, 'name': 'bob', 'email': 'bob@tarena.com.cn'}
```

4.删除字典

通过 del 可以删除字典中的元素或整个字典,使用内部方法 clear()可以清空字典.使用 pop()方法可以‘弹出’字典中的元素.

```
>>> print aDict
{'age': 22, 'name': 'bob', 'email': 'bob@tarena.com.cn'}
>>> del aDict['email']
>>> print aDict
{'age': 22, 'name': 'bob'}
>>> aDict.pop('age')
22
>>> print aDict
{'name': 'bob'}
>>> aDict.clear()
>>> print aDict
{}
```

5.字典操作符

使用字典键查找操作符[],查找键所对应的值,使用 in 和 not in 判断键是否存在于字典中.

```
>>> aDict = {'age':23,'name':'bob'}
>>> print aDict['name']
bob
>>> 'bob' in aDict
False
>>> 'name' in aDict
True
```

6.字典相关函数

len():返回字典中元素的数目

```
>>> print aDict
{'age': 23, 'name': 'bob'}
>>> print len(aDict)
2
```

hash():本身不是为字典设计的,但是可以判断某个对象是否可以作为字典的键

```
>>> hash(3)
3
>>> hash('abc')
1453079729188098211
```

```
>>> hash((1,2))
3713081631934410656
>>> hash([1,2])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

7.字典内建方法

dict.copy():返回字典(浅复制)的一个副本,只复制值,不复制 id

```
>>> print aDict
{'age': 23, 'name': 'bob'}
>>> bDict=aDict.copy()
>>> bDict
{'age': 23, 'name': 'bob'}
>>> bDict['name']='alice'
>>> print aDict
{'age': 23, 'name': 'bob'}
>>> print bDict
{'age': 23, 'name': 'alice'}
```

dict.get(key,default=None):对字典 dict 中的键 key,返回它对应的值 value,如果字典中不存在此键,则返回 default 的值

```
>>> print aDict
{'age': 23, 'name': 'bob'}
>>> aDict.get('name','not found')
'bob'
>>> aDict.get('email','not found')
'not found'
```

dict.setdefault(key,default=None):如果字典中不存在 key 键,由 dict[key]=default 为它赋值

```
>>> print aDict
{'age': 23, 'name': 'bob'}
>>> aDict.setdefault('age',20)
23
>>> print aDict
{'age': 23, 'name': 'bob'}
>>> aDict.setdefault('phone',123456789)
123456789
>>> print aDict
{'phone': 123456789, 'age': 23, 'name': 'bob'}
```

dict.items():返回一个包含字典中(键,值)对元组列表

```
>>> bdict
{'bob':10,'alice':18,'tom':18}
>>> bdict.items()
[('bob', 10), ('alice', 18), ('tom', 18)]
```

dict.keys():返回一个包含字典中键的列表

dict.values():返回一个包含字典中所有值的列表

dict.update(dict2):将字典 dict2 的键-值对添加到字典 dict

8.用户登录信息系统脚本

```
#!/usr/bin/env python
import getpass

user_dict = {}
def new_user():
    username = raw_input('username: ')
    if username not in user_dict:
        password = raw_input('password: ')
        user_dict[username] =  password

def old_user():
    username = raw_input('username: ')
    password = getpass.getpass('password: ')
    if user_dict.get(username) == password:
        print 'Login seccessful!'
    else:
        print 'Login incorrect'

CMDs = {'0':new_user,'1':old_user}
def show_menu():
    prompt = """(0)register user
(1)user login
(2)quit
Please input your choice(0/1/2): """
    while True:
        choice = raw_input(prompt).strip()[0]
        if choice not in '012':
            print 'INvalid choice.Try again.'
            continue
        if choice == '2':
            break
        CMDs[choice]()
if __name__ == '__main__':
    show_menu()
```

9.ip 地址和 10 进制数的转换脚本

```
#!/usr/bin/env python

def ip2int(ipaddr):
    ip_list = ipaddr.split('.')
    num = 0
    for i in range(4):
        num += int(ip_list[i]) * 256 ** (3 - i)
    return num
```

```
def int2ip(num):
    ip_list = []
    for i in range(4):
        num,extra = divmod(num,256)
        ip_list.insert(0,str(extra))
    return '.'.join(ip_list)

if __name__ == "__main__":
    print ip2int('192.168.1.10')
    print int2ip(3232235786)
```

十一、类型集合

1.数据类型比较

1)按存储模型分类

标量类型:数值,字符串

容器类型:列表,元组,字典

2)按更新模型分类

可变类型:列表,字典

不可变类型:数字,字符串,元组

3)按访问模型分类

直接访问:数字

顺序访问:字符串,列表,元组

映射访问:字典

2.python 对象特性

所有的 python 对象都拥有三个特性:

1)身份 id():每一个对象都有一个唯一的身份标识自己,任何对象的身份可以使用内建函数 id()来得到.

2)类型 type():决定了该对象可以保存什么类型的值,可以进行什么样的操作,以及遵循什么样的规则,用内建函数 type()查看对象的类型.

3)值:对象表示的数据项.

3.python 对象属性

某些 python 对象有属性,值或相关联的可执行代码

用点.标记法来访问属性

最常用的属性是函数和方法,不过有一些 python 类型也有数据属性

含有数据属性的对象包括(但不限于):类,类实例,模块,复数和文件

4.python 对象类型

1)数字:

整型

```
>>> a = 3
>>> a
3
>>> type(a)
<type 'int'>
```

长整型

```
>>> a = 3L
>>> a
3L
>>> type(a)
<type 'long'>
```

布尔型

```
>>> type(True)
<type 'bool'>
>>> type(False)
<type 'bool'>
>>> True == 1
>>> False == 0
```

,浮点型
复数型
2)字符串
3)列表
4)元组
5)字典

5.其他内建类型

在做 python 开发时,还可能用到的数据类型有:
1)标准类型

```
>>> type(type)
<type 'type'>
```

2)Null 对象(None)

```
>>> def foo(): return
...
>>> a = foo()
>>> print a
None
>>> if None: print 'hello'
...
>>> a == None
True
>>> a is None
True
>>> a is not None
False
```

3)文件
4)集合/固定集合
5)函数/方法
6)模块
7)类
8)内部类型
以下类型通常用不到,只是为了知识的完整性,简要的列出:
代码,帧,跟踪记录,切片,省略,xrange

6.标准类型运算符

1)对象值的比较

expr1<expr2 expr1 小于 expr2

expr1>expr2 expr1 大于 expr2

expr1<=expr2 expr1 小于等于 expr2

expr1>=expr2 expr1 大于等于 expr2

expr1==expr2 expr1 等于 expr2

expr1!=expr2 expr1 不等于 expr2

expr1<>expr2 expr1 不等于 expr2(不推荐)

2)对象身份比较

对象可以通过引用被赋值到另一个变量,因为每个变量都指向同一个(共享的)数据对象,只要人好一个引用发生改变,该对象的其他引用也会随着改变.

```
>>> aList = ['hello','bob']
>>> bList = aList
>>> id(aList)
140582071303632
>>> id(bList)
140582071303632
>>> aList is bList
True
>>> bList[1] = 'tom'
>>> print aList
['hello', 'tom']
```

7.集合类型

数学上,把 set 称作由不同的元素组成的集合,集合(set)的成员通常被称作集合元素.

集合对象是一组无序排列的可哈希的值,集合有两种类型:可变集合 set,不可变集合 frozenset.

```
>>> s1 = set('hello')
>>> s2 = frozenset('hello')
>>> s1
set(['h', 'e', 'l', 'o'])
>>> s2
frozenset(['h', 'e', 'l', 'o'])
```

8 集合类型操作符

集合支持 in 和 not in 操作符检查成员,能够通过 len()检查集合大小,能够使用 for 迭代结合成员,不能取切片,没有键.

```
>>> len(s2)
4
>>> for ch in s1:
...         print ch
...
h
e
l
o
```

|:联合,取并集

```
>>> s1 = set('abc')
>>> s2 = set('cde')
>>> s1|s2
set(['a', 'c', 'b', 'e', 'd'])
```

&:交集

```
>>> s1&s2
set(['c'])
```

-:差补

```
>>> s1-s2
set(['a', 'b'])
```

9.集合内建方法

set.add():添加成员

```
>>> print s1
set(['a', 'c', 'b'])
>>> s1.add('new')
>>> print s1
set(['a', 'new', 'c', 'b'])
```

set.update():批量添加成员

```
>>> s1.update('new')
>>> print s1
set(['a', 'c', 'b', 'e', 'n', 'w', 'new'])
```

set.remove():移除成员

```
>>> s1.remove('n')
>>> print s1
set(['a', 'c', 'b', 'e', 'w', 'new'])
```

s.issubsset(t):如果 s 是 t 的子集,则返回 True,否则返回 False

s.issuperset(t):如果 t 是 s 的超集,则返回 Ture,否则返回 False

s.union(t):返回一个新集合,该集合是 s 和 t 的并集

s.intersection(t):返回一个新集合,该集合是 s 和 t 的交集

s.difference(t):返回一个新集合,该集合是 s 的成员,但不是 t 的成员

10.集合练习

```
>>> s1 = set('hellllo')
>>> s1
set(['h', 'e', 'l', 'o'])
>>> s2 = frozenset('hello')
>>> s2
frozenset(['h', 'e', 'l', 'o'])
>>> s3 = set(['hello','world','line'])
>>> s3
set(['world', 'line', 'hello'])
>>> 'h' in s1
True
>>> len(s1)
```

```
4
>>> for ch in s1:print ch,
...
h e l o
>>> s1 = set('abc')
>>> s1
set(['a', 'c', 'b'])
>>> s2 = set('cde')
>>> s2
set(['c', 'e', 'd'])
>>> s1 & s2
set(['c'])
>>> s1 | s2
set(['a', 'c', 'b', 'e', 'd'])
>>> s1 - s2
set(['a', 'b'])
>>> s1.add('new')
>>> s1
set(['a', 'new', 'c', 'b'])
>>> s1.update(['hello','world'])
>>> s1
set(['a', 'c', 'b', 'world', 'new', 'hello'])
>>> s1.update('new')
>>> s1
set(['a', 'c', 'b', 'e', 'n', 'w', 'world', 'new', 'hello'])
>>> s1.remove('n')
>>> s1
set(['a', 'c', 'b', 'e', 'w', 'world', 'new', 'hello'])
>>> s1 = set('abcde')
>>> s1
set(['a', 'c', 'b', 'e', 'd'])
>>> s2 = set('abc')
>>> s2
set(['a', 'c', 'b'])
>>> s2.issubset(s1)
True
>>> s1.issuperset(s2)
True
>>> s1.union(s2)
set(['a', 'c', 'b', 'e', 'd'])
>>> s1.intersection(s2)
set(['a', 'c', 'b'])
>>> s1.difference(s2)
set(['e', 'd'])
```

十二、if 语句

1.标准 if 条件语句的语法

```
if expression:
        if_suite
else:
        else_suite
```

如果表达式的值非 0 或者为布尔值 True,则代码组 if_suite 被执行,否则就去执行 else_suite.
代码组是一个 python 术语,它由一条或多条语句组成,表示一个子代码块.
如果代码块仅包含一行代码,也可以和前面的写在一行,但可读性差,不建议使用.

```
if expression: expr_true_suite
```

2.if 语句示例解析

1)只要表达式数字为非零值即为 True.

```
>>> if 10:
...         print 'yes'
...
yes
```

2)空字符串,空列表,空元组,空字典的值均为 False

```
>>> if '':
...         print 'yes'
... else:
...         print 'no'
...
no
```

3.扩展 if 条件语句的语法:elif 语句

1)检查多个表达式是否为真,并在为真时执行特定代码块中的代码.没有 case 或 switch 语句,可以使用其他数据结构模拟.

```
if expression1:
        expr1_true_suite
elif expression2:
        expr2_true_suite
elif expressionN:
        exprN_ture_suite
else:
        none_of_the_above_suite
```

2)只有满足相关条件,相应的子语句才会执行,对于多个分支,只有一个满足条件的分支被执行.

```
>>> if x > 0:
...         print 'Positive'
... elif x < 0:
...         print 'Negative'
... else:
...         print 'Zero'
```

4.多重条件

单个 if 语句可以通过使用布尔操作符 and,or 和 not 实现多重判断条件或是否定判断条件.为了实现更好的可读性,可

以使用()分组,但 python 并不强制这么做.

```
if not warn and(system_load >= 10):
    print 'WARNING:losing resources'
    warn += 1
```

5.条件表达式

python 在很长一段时间里没有条件表达式(C？X:Y)或称三元运算符,因为范.罗萨姆一直拒绝加入这样的功能.
从 python2.5 集成的语法确定为:X if C else Y

```
>>> x,y = 3,4
>>> smaller = x if x<y else y
>>> print smaller
3
```

6.模拟 case 语句

```
#!/usr/bin/env python
actions = ["create","delete","modify"]

op = raw_input("operation(create/delete/modify): ")

if op in actions:
    print "%s user"    % op
else:
    print "input error"
```

7.判断合法用户的脚本

```
#!/usr/bin/env python
#import getpass
username = raw_input('username: ')
password = raw_ipput('password: ')
#passwd = getpass.getpass('password: ')  密文显示
if username == "bob" and password == "123":
    print "Login successful!"
else:
    print "Login incorrect."
```

8.编写判断成绩的程序

```
#!/usr/bin/env python
#coding:utf8
score = int(raw_input("Input score: "))
if score >= 90:
    print "优秀"
elif score >= 80:
    print "良好"
elif score >= 70:
    print "好"
elif score >= 60:
    print "及格"
```

```
else:
    print "你得努力了"
```

9.石头,剪刀,布游戏脚本

```
#!/usr/bin/env python
#coding:utf8
import random
all_choice = ['石头','剪刀','布']
win_list = [['石头','剪刀'],['剪刀','布'],['布','石头']]
prompt = """(0)石头
(1)剪刀
(2)布
请选择(0/1/2):"""
ind = int(raw_input(prompt))
player = all_choice[ind]
computer = random.choice(all_choice)

print "You choice: %s computer choice: %s" % (player,computer)
if player == computer:
    print "\033[32;43;1m 平局\033[0m"
elif [player,computer] in win_list:
    print "\033[31;43;1mYou Win!!\033[0m"
else:
    print "\033[31;43;1mYou Lose!!\033[0m"
```

十三、while 循环

1.while 循环语法结构

当需要语句不断的重复执行时,可以使用 while 循环(通常在当循环次数不确定时使用 while 循环)

语法示例：

```
while expression:
    while_suite
```

语句 while_suit 会被连续不断的循环执行,直到表达式的值变成 0 或 False

示例:求 1 加到 100 的和的脚本

```
#!/usr/bin/env pythn
sum100 = 0
counter = 1

while counter <= 100:
    sum100 += counter
    counter += 1

print 'result is %d' % sum100
```

2.break 语句

break 语句可以结束当前循环然后跳转到下条语句,写程序的时候,应尽量避免重复的代码,在这种情况下可以使用 while-break 结构.

示例:直到输入的 tom,否则一直执行循环的脚本

```
name = raw_input('username:')
while name != 'tom':
    name =raw_input('username:')
#可以替换为
while True:
    name = raw_input('username:')
    if name == 'tom':
        break
```

3.continue 语句

当遇到 continue 语句时,程序会终止当前循环,并忽略剩余的语句,然后回到循环的顶端,如果仍然满足循环条件,循环体内语句继续执行,否则退出循环.

示例:100 内的奇数或偶数之和

```
#!/usr/bin/env python
sum100 = 0
counter = 0
while counter < 100:
    counter += 1
    if counter % 2:          偶数之和,等效于 if counter % 2 == 1:
  #if not counter % 2:    奇数之和,等效于 if counter % 2 == 0:
        continue
    sum100 += counter
print 'result is %d' % sum100
```

4.else 语句

python 中的 while 语句也支持 else 子句,else 子句只在循环完成后执行,break 会跳过 else 块.

```
#!/usr/bin/env python
sum10 = 0
i = 1
while i <= 10:
    sum10 += i
    i += 1
else:
    print sum10
```

十四、for 循环

1.for 循环语法结构

python 中的 for 接受可迭代对象(如序列或迭代器)作为其参数,每次迭代其中一个元素,与 while 循环一样,支持 break,continue,else 语句.一般情况下,循环次数未知采用 while 循环,循环次数已知,采用 for 循环.

语法示例:

```
for iter_var in iterable:
    suit_to_repeat
```

示例:求 1 加到 100 的和的脚本

```
#!/usr/bin/env python
sum100 = 0
```

```
for i in range(1,101):
    sum100 += i
print sum100
```

2.range 函数

for 循环常与 range 函数一起使用,range 函数提供循环条件,range 函数完整的语法为:range(start,end,step=1)

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(1,5)
[1, 2, 3, 4]
>>> range(1,5,2) #步长为 2
[1, 3]
>>> range(10,0,-1) #倒序排列
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

3.xrange 函数

xrange()类似 range(),不过当你有一个很大的范围列表时,xrange()可能更为合适,因为它不会在内存里创建列表的完整拷贝,它只被用在 for 循环中,在 for 循环外使用它没有意义,它的性能远高于 range(),因为它不生成整个列表.

```
>>> xrange(3)
xrange(3)
>>> for x in xrange(3):
...     print x
...
0
1
2
```

4.else 语句

与其他语言不同,python 的 while 和 for 语句还支持 else 语句,只有循环体正常结束时,else 的代码块才会执行,如果循环体内的代码被中断,那么 else 代码块也被中断,不会执行.

```
while expression:
    suite_to_repeat
else:
    else_clause
```

5.break 语句

break 语句可以结束当前循环,然后跳转到循环以下语句,可以用在 while 和 for 循环中,一般通过 if 语句检查,当某个外部条件被触发立即从循环中退出.

```
>>> while True:
...     username = raw_input('username:')
...     if username == 'bob':
...         break
...
username:bob
```

6.continue 语句

当遇到 continue 语句时,程序会终止当前循环,并忽略剩余的语句,然后回到循环的顶端.

在开始下一次循环前,如果是条件循环,将验证条件表达式,如果是迭代循环,将验证是否还有元素可以迭代.
只有验证成功后,才开始下一次迭代或循环.

```
>>> for i in range(5):
...         if i % 2:
...                 continue
...         print i
...
0
2
4
```

7.pass 语句

python 没有使用传统的大括号来标记代码块,有时有些地方再语法上要求要有代码,python 提供了 pass 语句,它不做任何事情

```
>>> def foo():
...
  File "<stdin>", line 2

    ^
IndentationError: expected an indented block #缩进的错误:预期的一个缩进块
>>> def foo():
...         pass
...
```

8.列表解析

语法:[expr for iter_var in iterable]

它是一个非常有用,简单,而且灵活的工具,可以用来动态的创建列表.这个语句的核心是 for 循环,他迭代 iterable 对象的所有条目.expr 应用于序列的每个成员,最后的结果值是该表达式产生的列表.

```
>>> [30 for i in range(3)]
[30, 30, 30]
>>> [i for i in range(3)]
[0, 1, 2]
>>> [i ** 2 for i in range(1,6)] #1 到 5 的平方
[1, 4, 9, 16, 25]
>>> [i ** 2 for i in range(1,11) if i % 2] #1 到 10 的奇数的平方
[1, 9, 25, 49, 81]
```

9.迭代器和 iter()函数

为序列对象提供了一个类序列的接口,从根本上说,迭代器就是有一个 next()方法的对象.

python 的迭代无缝的支持序列对象,而且它还允许程序员迭代非序列类型,不能向后移动或回到开始,也不能复制一个迭代器.

对一个对象调用 iter()就可以得到它的迭代器,当迭代器条目全部取出后,会引发一个 StopIteration 异常,但并不表示错误发生,只是告诉外部调用者迭代完成.

```
>>> i = iter('py')
>>> i.next()
'p'
>>> i.next()
```

```
'y'
>>> i.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

10.生成器表达式

生成器表达式是列表解析的一个扩展,它们的基本语法基本相同,不真正创建列表,而是返回一个生成器,生成器在每次计算出一个条目后,把这个条目‘产生’(yield)出来,使用‘延迟计算’,所以它在使用内存上更有效.

语法:(expr for iter_var in iterable if cond_expr)

```
>>> [i ** 2 for i in range(1,6) if i % 2]
[1, 9, 25]
>>> (i ** 2 for i in range(1,6) if i % 2)
<generator object <genexpr> at 0x24a40f0>
>>> a = (i ** 2 for i in range(1,6) if i % 2)
>>> a.next()
1
>>> a.next()
9
>>> a.next()
25
>>> a.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

11.迭代机制练习

```
#!/usr/bin/env python

astr = 'hello'
alist = ['hello','world']
atuple = ('new','python')
adict = {'name':'bob','age':20}
aiter = iter(['abc','xyz'])
fname = '/etc/hosts'
aset = set('abcd')

for ch in astr: print ch
for item in alist: print item
for item in atuple: print item
for key in adict: print '%s:%s' % (key,adict[key])
for i in aiter: print i
for i in aset: print i
fobj = open(fname)
for line in fobj: print line,
fobj.close()
```

12.斐波那契数列

```
#!/usr/bin/env python
fibs = [0,1]
for i in range(8):
    fibs.append(fibs[-1]+fibs[-2])
print fibs
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

#!/usr/bin/env python
a,b = 0,1
for i in range(10):
    print a,
    a,b = b,a+b
```

十五、文件对象

1.文件的打开方法

open 及 file 内建函数

作为打开文件之门的‘钥匙’,内建函数 open()以及 file()提供了初始化输入/输出(I/O)操作的通用接口.成功打开文件的.时候回返回一个文件对象,否则引发一个错误.open()方法和 file()方法可以完全相互替换.

基本语法:file_object = open(file_name,access_mode='r',buffering=-1) ,file_name 如果是变量不需加引号,否则要.

```
>>> f = open('passwd')
>>> f.close()
>>> f = file('passwd')
>>> f.close()
```

2.文件对象访问模式

r 以读方式打开(文件不存在则报错)
w 以写方式打开(文件存在则清空,不存在则创建)
a 以追加模式打开(必要时创建新文件)
r+ 以读写模式打开(参见 r)
w+ 以读写模式打开(参见 w)
a+ 以读写模式打开(参见 a)

3.文件输入

1)read 方法:read()方法用来直接读取字节到字符串中,最多读取给定数目个字节.如果没有给定 size 参数(默认值为-1)或者 size 值为负,文件将被读取直至末尾.

```
>>> data = fobj.read()
>>> print data
```

2)readline 方法:读取打开文件的一行(读取下个结束符之前的所有字节),然后整行,包括行结束符,作为字符串返回,它也有一个可选的 size 参数,默认为-1,代表读至行结束符,如果提供了该参数,那么在超过 size 个字节后会返回不完整的行.

```
>>> data = fobj.readline()
>>> print data
```

3)readlines 方法:readlines()方法读取所有(剩余的)行然后把它们作为一个字符串列表返回.

```
>>> data = fobj.readlines()
```

```
>>> print data
```

4.文件输入综合练习

```
>>> f = open('/etc/passwd') #打开文件
>>> f.read(50) #读入 50 个字节
'root:x:0:0:root:/root:/bin/bash\nbin:x:1:1:bin:/bin'
>>> f.readline() #读入一行（不包括前 50 个字节）
':/sbin/nologin\n'
>>> f.readline() #读入一行
'daemon:x:2:2:daemon:/sbin:/sbin/nologin\n'
>>> f.readline(2)  读入一行的两个字节
'ad'
>>> f.readline(3)
'ot:'
>>> f.readline(4)
'x:0:'
>>> f.readlines() #读入剩余的所有行,并作为列表
剩余文件的全部行
>>> f.close() #关闭文件
```

5.文件输出

1)write 方法:write()内建方法功能与 read()和 readline()相反,它把含有文本数据或二进制数据块的字符串写入到文件中去,写入文件时,不会自动添加行结束标志,需要程序员手工输入.

```
flush() #立即同步数据到文件中
语法示例:file_name.write('object\n')
>>> f = open('hello','w') #打开一个文件无则创建,有则清空
>>> f.write('hello world!')
>>> f.flush() #刷新,将缓存数据写入硬盘
>>> f.write('\n') #写入换行
>>> f.flush()
>>> f.write('2nd line.\n')
>>> f.close() #关闭时将缓存数据写入硬盘,每 4096 字节将自动写入硬盘,一页等于 4096 字节(k)
```

2)writelines 方法:和 readlines()一样,writelines()方法是针对列表的操作,它接受一个字符串列表作为参数,将它们写入文件,行结束符并不会自动加入,所有如果需要的话,必须在调用 writelines()前给每行结尾加上行结束符.

```
语法示例:bobj.writelines(['Hello World!\n','python programing\n'])
>>> f = open('hello','w')
>>> f.writelines(['hello world!\n','2nd line.\n'])
>>> f.close()
```

6.文件迭代

如果需要逐行处理文件,可以结合 for 循环迭代文件,迭代文件的方法与处理其他序列类型的数据类似.

```
>>> fobj = open('star.py')
>>> for eachLine in fobj:
...         print eachLine,
...
#!/usr/bin/env python
'''star module
```

```
just a sample module.
only include one function.'''
def pstar():
    'do not accept args.Used to print 50 stars'
    print '*' * 50
```

7.迭代练习

```
>>> f = open('/etc/hosts')
>>> for line in f:
...     print line,     print 默认会打印空行,','可以抑制空行的产生
...
127.0.0.1   localhost localhost.localdomain localhost4 localhost4.localdomain4
::1         localhost localhost.localdomain localhost6 localhost6.localdomain6
```

8.with 子句

with 语句是用来简化代码的,在将打开文件的操作放在 with 语句中,代码块结束后,文件将自动关闭.

```
>>> with open('ip.py') as f:
...     data = f.readlines()
...
>>> f.closed
True

>>> with open('/etc/hosts') as f:
...     for line in f:
...         print line,
...
127.0.0.1   localhost localhost.localdomain localhost4 localhost4.localdomain4
::1         localhost localhost.localdomain localhost6 localhost6.localdomain6
```

9.文件内移动

1)seek(offset[,whence]):移动文件指针到不同的位置;offset 是相对于某个位置的偏移量;whence 的值,0 表示文件开头,1 表示当前位置,2 表示文件结尾.

2)tell():返回当前文件指针的位置(字节)

10.标准文件应用

标准文件:程序一执行,就可以访问三个标准文件.

标准输入:一般是键盘,使用 sys.stdin

标准输出:一般是显示器缓冲输出,使用 sys.stdout

标准错误:一般是显示器的非缓冲输出,使用 sys.stderr

```
>>> import sys
>>> a = sys.stdin.readline()
hello
>>> a
'hello\n'
>>> sys.stdout.write('hello world') #默认没有换行符
hello world>>>
>>> sys.stdout.write('hello world\n')
```

```
hello world
>>> sys.stderr.write('hello world\n')
hello world
```

11.文件相关模块

os 模块:对文件系统的访问大多通过 python 的 os 模块实现,该模块是 python 访问操作系统功能的主要接口.

```
>>> import os
```

symlink():创建符号链接

```
>>> os.symlink('u2d.py','unix2dos') #1,原文件  2,目标文件
```

listdir():列出指定目录的文件

```
>>> os.listdir('/root') #需要给定指定的目录,显示所有目录和文件,不递归
['.bash_logout', '.bash_profile', '.bashrc',.............]
>>> os.curdir #当前目录
'.'
>>> os.listdir(os.curdir)
.....
>>> os.pardir #上级目录
'..'
>>> os.listdir(os.pardir)
.....
```

getcwd():返回当前工作目录

```
>>> os.getcwd()
'/root/PycharmProjects/mypy'
```

mkdir():创建目录

```
>>> os.mkdir('/home/demos')
```

chmod():改变权限模式

```
>>> os.chmod('bar.py',0755)
```

getatime():返回最近访问时间

```
[root@localhost mypy]# stat bar.py #统计文件
  文件:"bar.py"
  大小:32              块:8          IO 块:4096   普通文件
设备:fd00h/64768d    Inode:140611872   硬链接:1
权限:(0755/-rwxr-xr-x)  Uid:(    0/    root)   Gid:(    0/    root)
环境:unconfined_u:object_r:admin_home_t:s0
最近访问:2016-03-03 16:18:41.136690598 +0800
最近更改:2016-02-24 16:33:49.970735897 +0800
最近改动:2016-03-03 17:07:42.172288956 +0800
创建时间:-
>>> os.path.getatime('bar.py')
1456993121.1366906   #1970.01.01.0 点到现在的秒数
>>> import time
>>> time.ctime(1456993121.1366906) #转换为格式
'Thu Mar  3 16:18:41 2016'
>>> time.ctime() #当前系统时间
'Thu Mar  3 17:13:39 2016'
>>> time.ctime(0)
```

```
'Thu Jan    1 08:00:00 1970' #八点是因为时区问题
```

12.文件相关模块练习

```
>>> os.chdir('/home') #切换目录
>>> os.getcwd()
'/home'
>> os.mknod('/home/abc') #创建文件
>>> os.walk('.') #walk 用于遍历目录
<generator object walk at 0x103d500>
>>> list(os.walk('/home/ace')) #(路径,目录,文件)
[('/home/ace', ['xyz'], ['eeea', 'eebe']), ('/home/ace/xyz', [], ['aer'])]
[root@localhost mypy]# tree /home/ace #tree 命令用于查看目录结构
/home/ace
|-- eebe
|-- eeea
`-- xyz
    `-- aer
1 directory, 3 files

>>> os.path.basename('/home/ace/xa')
'xa'
>>> os.path.dirname('/home/ace/xa')
'/home/ace'
>>> os.path.split('/home/ace/xa')
('/home/ace', 'xa')
>>> os.path.join('/home/ace/xa')
'/home/ace/xa'
>>> os.path.isfile('/etc') #判断是否为文件
False
>>> os.path.isfile('/etcae')
False
>>> os.path.isfile('/etc/hosts')
True
>>> os.path.isdir('/etc') #判断是否为目录
True
>>> os.path.islink('/etc') #判断是否为链接文件
False
>>> os.path.islink('/etc/grub2.cfg')
True
>>> os.path.ismount('/boot') #是否为挂载点
True
```

13.文件操作模块练习

```
>>> import shutil #高层次的文件操作模块
>>> shutil.copy('/bin/ls','/home') #将 ls 文件拷贝到 home 目录下(包含权限)
>>> shutil.copy('/bin/ls','/home/list') #将 ls 文件内容拷贝到 home 目录下并改名 list(无权限)
```

```
>>> shutil.move('/root/bin','/home') #剪切
>>> shutil.copytree('/root/Public','/home') #拷贝目录
>>> shutil.rmtree('/home/bin') #删除目录
```

14.cPickle 模块

python 提供一个标准的模块,称为 pickle,使用它可以在一个文件中储存任何 python 对象,之后又可以把它完整无缺的取出来.还有另一模块称为 cPickle,它的功能和 pickle 模块完全相同,只不过它使用 C 语言编写的,因此要快得多.分别调用 dump()和 load()可以存储,写入.

应用示例 1:

```
>>> import cPickle as p
>>> shoplistfile = 'shoplist.data'
>>> shoplist = ['apple','mango','carrot']
>>> f = file(shoplistfile,'w')
>>> p.dump(shoplist,f)
>>> f.close()
>>>
>>> f = file(shoplistfile)
>>> storedlist = p.load(f)
>>> print storedlist
['apple', 'mango', 'carrot']
```

应用示例 2:

```
>>> import pickle
>>> import cPickle
>>> pickle.__file__
'/usr/lib64/python2.7/pickle.pyc'
>>> cPickle.__file__
'/usr/lib64/python2.7/lib-dynload/cPickle.so'
>>> import StringIO
>>> import cStringIO
```

应用示例 3:

```
>>> import cPickle as p
>>> shoplist = ['apple','egg','banana']
>>> f = open('m.data','w')
>>> p.dump(shoplist,f)
>>> f = open('m.data')
>>> newlist = p.load(f)
>>> newlist
['apple', 'egg', 'banana']
>>> type(newlist)
<type 'list'>
>>> newlist[0]
'apple'
```

15.模拟 cp 操作

```
#!/usr/bin/env python
s_fname = '/bin/ls'
```

```
d_fname = '/root/list'
s_fobj = open(s_fname)
d_fobj = open(d_fname,'w')
buf_size = 4096
while True:
    data = s_fobj.read(buf_size)
    if not data: #if data == '':
        break
    d_fobj.write(data)
s_fobj.close()
d_fobj.close()
```

16 三局两胜的石头剪刀布

```
#!/usr/bin/env python
#coding:utf8
import random
all_choice = ['石头','剪刀','布']
win_list = [['石头','剪刀'],['剪刀','布'],['布','石头']]
prompt = """(0)石头
(1)剪刀
(2)布
请选择(0/1/2):"""
cwin = 0
pwin = 0

while cwin < 2 and pwin < 2:
    ind = int(raw_input(prompt))
    player = all_choice[ind]
    computer = random.choice(all_choice)

    print "You choice: %s computer choice: %s" % (player,computer)
    if player == computer:
        print "\033[32;43;1m 平局\033[0m"
    elif [player,computer] in win_list:
        pwin += 1
        print "\033[31;43;1mYou Win!!\033[0m"
    else:
        cwin += 1
        print "\033[31;43;1mYou Lose!!\033[0m"

if pwin == 2:
    print "Finall,You WIN!!"
else:
    print "Finall,You Lose!!"
```

17.去除文件中的空行和#开头行

1)if,and 的版本

```
#!/usr/bin/env python
import sys

def keep_lines(f):
    for line in f:
        if line.strip() and (not line.startswith('#')):
            print line,

if __name__ == '__main__':
    fname = sys.argv[1]
    fobj = open(fname)
    keep_lines(fobj)
    fobj.close()
```

2)if,or 版本

```
#!/usr/bin/env python
import sys

def keep_lines(f):
    for line in f:
        if (not line.strip()) or line.startswith('#'):
            continue
        print line,

if __name__ == '__main__':
    fname = sys.argv[1]
    fobj = open(fname)
    keep_lines(fobj)
    fobj.close()
```

18.原位输出数字脚本

```
#!/usr/bin/env python

import time
import sys

for i in range(1,11):
    sys.stdout.write('\r%s' % i)
    sys.stdout.flush() #如果注释掉此行,不会时时输出,会过 10 秒后,直接输出数字 10
    time.sleep(1)
```

19.a 在#上循环跑脚本

```
#!/usr/bin/env python
import sys
import time
```

```
counter = 0
sys.stdout.write('#' * 20)
sys.stdout.flush()
while True:
    sys.stdout.write('\r%sa' % ('#' * counter))
    counter += 1
    sys.stdout.flush()
    if counter == 20:
        counter = 0
        sys.stdout.write('\r%s' % ('#' * 20))
        sys.stdout.flush()
    time.sleep(0.3)
```

20.linux 文本转 win 文本脚本

```
#!/usr/bin/env python

import sys

def unix2dos(fname):
    fobj = open(fname)
    newf = open(fname + '.txt','w')
    for line in fobj:
        newf.write(line.strip('\r\n') + '\r\n')
    fobj.close()
    newf.close()
if __name__ == '__main__':
    if len(sys.argv) != 2:
        print 'Usage: %s filename' % sys.argv[0]
    else:
        unix2dos(sys.argv[1])
```

21.linux 文本转 win 文本(unix2dos,dos2unix 命令)

```
[root@localhost ~]# yum -y install unix2dos
[root@localhost ~]# unix2
unix2dos    unix2mac
[root@localhost ~]# unix2dos    keep_lines.py #linux 转 win
unix2dos: converting file keep_lines.py to DOS format ...
[root@localhost ~]# dos2unix keep_lines.py    #win 转 linux
dos2unix: converting file keep_lines.py to Unix format ...
```

22.补充知识,tr 大小写转换命令

```
[root@localhost mypy]# tr 'a-z' 'A-Z'  小写转大写
qwe  输入的,回车
QWE  输出的
QWE  输入的,回车
QWE  输出的
[root@localhost mypy]# tr 'a-z' 'A-Z' < case.py #将文件中的内容小写转大写,输出到屏幕
```

[root@localhost mypy]# tr 'a-z' 'A-Z' < case.py > aaa #将文件中的内容小写转大写,输出到 aaa 中

十六、函数(def)

1.函数基本概念

函数是对程序逻辑进行结构化或过程化的一种编程方法,将整块代码巧妙的隔离成已于管理的小块,把重复的代码放到函数中而不是进行大量的拷贝,这样既能节省空间,也有助于保持一致性,通常函数都是用于实现某一种功能.

2.创建函数

函数使用 def 语句创建,语法如下:

```
def function_name(arguments):
    'function_documentation_string' #文档字符串
    function_body_suite
```

标题行由 def 关键字,函数的名字,以及参数的集合(如果有的话)组成.

def 子语句的剩余部分包括了一个虽然可选但是强烈推荐的文档字符,和必须的函数体.

3.前向引用

函数不允许在函数未声明之前对其进行引用或者调用

```
>>> def foo():
...         print 'in foo'
...         bar()
...
>>> foo() #报错,因为 bar 没有定义
in foo
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in foo
NameError: global name 'bar' is not defined
-------------------------------------------------
>>> def foo():
...         print 'in foo'
...         bar()
...
>>> def bar():
...         print 'in bar'
...
>>> bar()
in bar
>>> foo() #正常执行,虽然 bar 的定义在 foo 定义后面
in foo
in bar
```

4.函数属性

函数属性是 python 另外一个使用了句点属性标识符并拥有名字空间的领域

```
>>> def foo():
...         'foo()-- orioerly created doc sting'
...     print 'in foo'
```

```
...
>>> foo.__doc__
'foo()-- orioerly created doc sting'
>>> foo.version = 1.0
>>> foo.version
1.0
```

5.内部函数

在函数体内创建另外一个函数是完全合法的,这种函数叫做内部函数/内嵌函数

```
>>> def foo():
...         def bar():
...                 print 'bar() is called'
...         print 'foo() is called'
...         bar()
...
>>> foo()
foo() is called
bar() is called
>>> bar()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'bar' is not defined
```

6.调用函数

同大多数语言一样,python 用一对圆括号调用函数,如果没有加圆括号,只是对函数的引用.任何输入的参数都必须放置在括号中.

应用示例 1:

```
>>> def foo():
...         print 'Hello world!'
...
>>> foo()
Hello world!
>>> foo
<function foo at 0x7fb1092d2938>
```

应用示例 2:

```
>>> def pstar():
...         print "*" * 20
...
>>> pstar()
********************
>>> pstar
<function pstar at 0x2454c08>
>>> a = pstar #引用
>>> a
<function pstar at 0x2454c08>
>>> a()
```

```
>>> b = pstar() #调用并执行
**********
>>> print b
None
```

7.函数的返回值

多数情况下,函数并不直接输出数据,而是向调用者返回值,函数的返回值使用 return 关键字,没有 return 的话,函数默认返回 None.

```
>>> def foo():
...         res = 3+4
...
>>> i = foo()
>>> print i
None
```

8.函数参数

形式参数:函数定义时,紧跟在函数名后(圆括号内)的参数被称为形式参数,简称形参,由于它不是实际存在变量,所以又称为虚拟变量.

实际参数:在主调函数中调用一个函数时,函数名后面括弧中的参数(可以是一个表达式)称为‘实际参数’,简称为实参.

应用示例 1:

```
>>> def pstar(num):
...         print "*" * num
...
>>> pstar(10)
**********
>>> pstar(20)
********************
>>> pstar()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: pstar() takes exactly 1 argument (0 given)
```

应用示例 2:

```
>>> def pstar(num = 20): #默认参数
...         print "*" * num
...
>>> pstar()
********************
>>> pstar(10)
**********
```

9.传递参数

调用函数时,实际的个数需要与形参个数一致,实参将依次传递给形参.

```
>>> def foo(x,y):
...         print 'x = %d,y = %d' % (x,y)
...
>>> foo()
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
TypeError: foo() takes exactly 2 arguments (0 given)
>>> foo(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() takes exactly 2 arguments (1 given)
>>> foo(3,4)
x = 3,y = 4
```

10.位置参数

与 shell 脚本类似,程序名以及参数都以位置参数的方式传递给 python 程序,使用 sys 模块的 argv 列表接收.

```
[root@localhost test]# vim args.py
#!/usr/bin/env python
import sys
print sys.argv
[root@localhost test]# chmod +x args.py
[root@localhost test]# ./args.py hello world
['./args.py', 'hello', 'world']      #0 为程序名
```

11.默认参数

默认参数就是声明了默认值的参数,因为给参数赋予了默认值,所以在函数调用时,不向该参数传入值也是可以的.

```
>>> def pstar(num = 30):
...         print '*' * num
...
>>> pstar()
******************************
>>> pstar(50)
**************************************************
```

12.关键字参数

关键字参数的概念仅仅针对函数的调用,这种理念是让调用者通过函数调用中的参数名字来区分参数,这样规范允许参数缺失或者不按顺序.

```
>>> def getInfo(name,age):
...         print "%s's age is %s" % (name,age)
...
>>> getInfo(23,'bob')
23's age is bob
>>> getInfo(age = 23,name = 'bob')
bob's age is 23
>>> getInfo(age=20,'bob')
  File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
>>> getInfo(20,name='bob')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: getInfo() got multiple values for keyword argument 'name'
>>> getInfo('bob',age=20)
```

bob's age is 20

13.参数组

python 允许程序员执行一个没有显式定义参数的函数,相应的方法是通过一个把元组(非关键参数)或字典(关键字参数)作为参数传递给函数.

语法示例:func(*tuple_grp_nonkw_args,**dict_grp_kw_args)

```
#!/usr/bin/env python
def func1(*args):
        print args
def func2(**kw_args):
        print kw_args
def func3(name,*args,**kw_args):
        print name
        print args
        print kw_args
func1()
func1(10)
func1(10,20)
func1(10,'abc',123)
func2()
func2(name='bob')
func2(name='tom',age=20)
func3('bob',23,30,qq='11111',em='sss@aa.com')
[root@localhost mypy]# python func1.py
()
(10,)
(10, 20)
(10, 'abc', 123)
{}
{'name': 'bob'}
{'age': 20, 'name': 'tom'}
bob
(23, 30)
{'qq': '11111', 'em': 'sss@aa.com'}
```

*operator 函数操作模块(用于解开组)

```
>>> import operator
>>> nums = [10,20]
>>> operator.add(nums) #operator.add(x, y)等于 x+y
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: op_add expected 2 arguments, got 1
>>> operator.add(*nums) #将组解开
30
```

14.匿名函数:lambda

python 允许用 lambda 关键字创造匿名函数,匿名是因为不需要以标准的 def 方式来声明.一个完整的 lambda ‘语句’

代表了一个表达式,这个表达式的定义体必须和声明放在同一行.
语法示例:lambda[arg1[,arg2,...argN]]:expression

```
>>> a = lambda x,y:x+y
>>> print a(3,4)
7
#!/usr/bin/env python
def myadd(x,y):
    return x+y
a = lambda x,y:x+y
print myadd(3,4)
print a(5,6)
```

15.filter()函数

filter(func,seq):调用一个布尔函数 func 来迭代遍历每个序列中的元素,返回一个使 func 返回值为 tru 的元素的序列,如果布尔函数比较简单,直接使用 lambda 匿名函数显得非常方便了.

```
>>> data = filter(lambda x:x % 2,[num for num in range(10)]) #过滤出 10 以内的奇数
>>> print data
[1, 3, 5, 7, 9]
```

16map()函数

map(func,seq1[,seq2...]):将函数 func 作用于给定序列的每个元素,并用一个列表来提供返回值

```
>>> data = map(lambda x:x+2,[num for num in range(5)])
>>> print data
[2, 3, 4, 5, 6]
```

17.reduce()函数

reduce(func,seq[,init]):将二元函数作用于 seq 序列的元素,每次携带一对(先前的结果以及下一个序列元素),连续的将现有的结果和下一个给值作用在获得的随后的结果上,最后减少序列为一个单一的返回值

```
>>> data = reduce(lambda x,y:x+y,range(1,6)) #将 1 至 5 累加
>>> print data
15
```

18.综合练习

```
#!/usr/bin/env python
def odd(num):
    return num % 2
nums = [i for i in range(1,11)]
print filter(odd,nums)
print filter(lambda i:i%2,nums)
print map(lambda i:i*2+1,nums)
print reduce(lambda i,j:i+j,nums)
ot@localhost mypy]# python func10.py
[1, 3, 5, 7, 9]
[1, 3, 5, 7, 9]
[3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
55
```

19.变量作用域

1)全局变量

标识符的作用域是定义为其声明在程序里的可应用范围,也就是变量的可见性,在一个模块中最高级别的变量有全局作用域,

全局变量的一个特征是除非被删除掉,否则它们的存活到脚本运行结束,且对于所有的函数,他们的值都是可以被访问的.

```
#!/usr/bin/env python
x = 10
def foo():
        print x
foo()
[root@localhost mypy]# python func20.py
10
```

2)局部变量

局部变量只是暂时的存在,仅仅只依赖于定义他们的函数现阶段是否处于活动,当一个函数调用出现时,期局部变量就进入声明它们的作用域.在那一刻,一个新的局部变量名为那个对象创建了.一旦函数完成,框架被释放,变量将会离开作用域.如果局部与全局有相同名称的变量,那么函数运行时,局部变量的名称将会把全局变量名称遮住.

应用示例 1:

```
>>> x = 4
>>> def foo():
...         x = 10
...         print 'in foo,x =',x
...
>>> foo()
in foo,x = 10
>>> print 'in main,x = ',x
in main,x =    4
```

应用示例 2:

```
#!/usr/bin/env python
x = 10
def foo():
        y = 20
        print x
        print y
foo()
print y
[root@localhost mypy]# python func20.py
10
20
Traceback (most recent call last):
   File "func20.py", line 8, in <module>
        print y
NameError: name 'y' is not defined
```

应用示例 3:

```
#!/usr/bin/env python
x = 10
```

```
def foo():
    x = 20
    print x
foo()
print x
[root@localhost mypy]# python func20.py
20
10
```

20.global 语句

因为全局变量的名字能被局部变量给遮盖掉,所以为了明确的引用一个已命名的全局变量,必须使用 global 语句

应用示例 1:

```
>>> x = 4
>>> def foo():
...         global x
...         x = 10
...         print 'in foo,x = ',x
...
>>> foo()
in foo,x =    10
>>> print 'in main,x = ',x
in main,x =    10
```

应用示例 2:

```
#!/usr/bin/env python
x = 10
def foo():
    global x
    x = 20
    print x
foo()
print x
[root@localhost mypy]# python func20.py
20
20
```

21.名字空间

任何时候,总有一个到三个活动的作用域(内建,全局和局部),标识符的搜索顺序依次是局部,全局和内建.提到名字空间,可以想象是否有这个标识符,提到变量作用域,可以想象是否可以‘看见’这个标识符.

```
#!/usr/bin/env python
print len
print len('abc')

len = 10
print len

def aaa():
```

```
    len = 'hello'
    print len
aaa()
[root@localhost mypy]# python getuser.py
<built-in function len>
3
10
hello
```

22.偏函数

偏函数的概念是将函数式编程的概念和默认参数以及可变参数结合在一起,一个带有多个参数的函数,如果其中某些参数基本上固定的,那么久可以通过偏函数为这些参数赋默认值.

```
>>> from operator import add
>>> from functools import partial
>>> add10 = partial(add,10)
>>> print add10(25)
35
```

应用示例：

```
#!/usr/bin/env python
import operator
import functools
print operator.add(10,3)
print operator.add(10,6)
print operator.add(10,30)
add10 = functools.partial(operator.add,10)
print add10(3)
print add10(6)
print add10(30)
[root@localhost mypy]# python func30.py
13
16
40
13
16
40
```

23.偏函数应用

```
[root@localhost mypy]# cat u2d.py
#!/usr/bin/env python
import sys
def unix2dos(line_sep,fname):
    fobj = open(fname)
    newf = open(fname + '.txt','w')
    for line in fobj:
        newf.write(line.strip('\r\n') + line_sep)
    fobj.close()
```

```
        newf.close()
if __name__ == '__main__':
        try:
                unix2dos('\r\n',sys.argv[1])
        except IndexError:
                print 'usage: %s filename' % sys.argv[0]
[root@localhost mypy]# cat d2u.py
#!/usr/bin/env python
import u2d
import functools
import sys
dos2unix = functools.partial(u2d.unix2dos,'\n')
try:
        dos2unix(sys.argv[1])
except IndexError:
        print 'Usage: %s filename' % sys.argv[0]
```

24.递归函数

如果函数包含了对其自身的调用,该函数就是递归的,在操作系统中,查看某一目录内所有文件,修改权限等都是递归的应用.

```
>>> def func(num):
...         if num == 1:
...                 return 1
...         else:
...                 return num * func(num - 1)
...
>>> print func(5)
120
>>> print func(10)
3628800
```

25.内部函数

闭包:闭包将内部函数自己的代码和作用域以及外部函数的作用结合起来.闭包的词法变量不属于全局名字空间域或者局部的而属于其他的名字空间,带着‘流浪’的作用域.闭包对于安装计算,隐藏状态,以及在函数对象和作用域中随意的切换时很有用的.闭包也是函数,但是他们能携带一些额外的作用域.

26.闭包实例

在电子商务平台下,每件商品都有一个计数器说明该商品售出数量,这个计数器可以通过闭包实现

```
>>> def counter(start_at = 0):
...         count = [start_at]
...         def incr():
...                 count[0] += 1
...                 return count[0]
...         return incr
...
>>> a = counter()
>>> print a()
```

```
1
>>> b = counter(10)
>>> print b()
11
>>> print a()
2
```
应用示例:
```
#!/usr/bin/env python
def counter(start_at = 0):
    count = [start_at]
    def incr():
        count[0] += 1
        return count[0]
    return incr
if __name__ == '__main__':
    a = counter()
    print a();print a();print a()
    b = counter(10)
    print b();print b();print b()
    print a()
    print b()
[root@localhost mypy]# python counter.py
1
2
3
11
12
13
4
14
```

27.装饰器

装饰器是在函数调用之上的修饰,这些修饰仅是当声明一个函数或者方法的时候,才会应用的额外调用,使用装饰器的情形有:引入日志,增加计时逻辑来检测性能,给函数加入事务的能力.

28.输出程序运行的起止时间

1)标准应用
```
#!/usr/bin/env python
import time
def func():
    alist = []
    for i in range(10):
        alist.append(i)
        time.sleep(0.5)
    return alist
if __name__ == '__main__':
```

```
    print '%s start at: %s' % (func.__name__,time.ctime())
    print func()
    print '%s done at: %s' % (func.__name__,time.ctime())
```

2)标准应用升级

```
#!/usr/bin/env python
import time
def timeit(f):
    print '%s start at: %s' % (f.__name__,time.ctime())
    result = f()
    print '%s done at: %s' % (f.__name__,time.ctime())
    return result
def func():
    alist = []
    for i in range(10):
        alist.append(i)
        time.sleep(0.5)
    return alist
if __name__ == '__main__':
    print timeit(func)
```

3)装饰器应用

```
#!/usr/bin/env python
import time
def deco(f):
    def timeit(*args):
        print '%s start at: %s' % (f.__name__,time.ctime())
        result = f(*args)
        print '%s done at: %s' % (f.__name__,time.ctime())
        return result
    return timeit
@deco
def func(num):
    alist = []
    for i in range(num):
        alist.append(i)
        time.sleep(0.5)
    return alist
if __name__ == '__main__':
    print func(4)
```

29.生成器

从语法上将,生成器是一个带 yield 语句的函数,一个函数或者子程序只返回一次,但一个生成器能暂停执行并返回一个中间的结果.

yield 语句返回一个值给调用者并暂停执行,当生成器的 next()方法被调用的时候,它会准确的从离开地方继续.

与迭代器相似,生成器以另外的方式来运作,当到达一个真正的返回或者函数结束没有更多的值返回,StopIteration 异常就会被抛出.

应用示例 1:

```
>>> def simpGen():
...         yield '1'
...         yield '2 -> punch'
...
>>> mygen = simpGen()
>>> mygen.next()
'1'
>>> mygen.next()
'2 -> punch'
>>> mygen.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

应用示例 2:

```
>>> [i ** 2 for i in range(10) if i % 2]
[1, 9, 25, 49, 81]
>>> (i ** 2 for i in range(10) if i % 2)
<generator object <genexpr> at 0x7f8e16ccbdc0>
>>> a = (i ** 2 for i in range(10) if i % 2)
>>> a.next()
1
>>> a.next()
9
>>> for i in a:
...         print i,
...
25 49 81
>>> a.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

应用示例 3:

```
#!/usr/bin/env python
def sim_gen():
        yield 'hello'
        yield 'world'
        yield 10
a = sim_gen()
print a.next();print a.next();print a.next();print a.next()
b = sim_gen()
for i in b:
        print i,
[root@localhost mypy]# python simgen.py
hello
world
10
```

```
Traceback (most recent call last):
  File "simgen.py", line 7, in <module>
    print a.next();print a.next();print a.next();print a.next()
StopIteration
```

应用示例 4:

```
#!/usr/bin/env python
def blocks(fobj):
    block = []
    count = 0
    while True:
        line = fobj.readline()
        if not line:
            break
        block.append(fobj.readline())
        count += 1
        if count == 10:
            yield block
            count = 0
            block = []
    yield block
if __name__ == '__main__':
    with open('/etc/passwd') as f:
        lines = blocks(f)
        for b in lines:
            print b
```

30.生成器特性

用户可以通过 send()将值回送给生成器,还可以在生成器中抛出异常,以及要求生成器退出

```
>>> def counter(num):
...     for i in range(num):
...         val = yield i
...
>>> c = counter(10)
>>> c.next()
0
>>> c.close()
>>> c.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

31.函数版斐波那契数列

```
#!/usr/bin/env python
def fibs(num):
    fib = [0,1]
    for i in range(num - 2):
```

```
            fib.append(fib[-1] + fib[-2])
        return fib
print fibs(10)
print fibs(20)
[root@localhost test]# python file.py
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181]
```

32.位置参数练习,定义个数的斐波那契数列

```
#!/usr/bin/env python
import sys
def fibs(num):
        fib = [0,1]
        for i in range(num - 2):
                fib.append(fib[-1] + fib[-2])
        return fib
num = int(sys.argv[1])
print fibs(num)
[root@localhost mypy]# python fei.py 5
[0, 1, 1, 2, 3]
```

33.数学游戏脚本

```
#!/usr/bin/env python

import random
import operator

ops = {'+':operator.add,'-':operator.sub}

def probe():
        nums = [random.randint(1,50) for i in range(2)]
        nums.sort(reverse=True)
        op = random.choice('+-')
        anwser = ops[op](*nums)
        prompt = '%s %s %s = ' % (nums[0],op,nums[1])
        tries = 0

        while tries < 3:
                result = int(raw_input(prompt))
                if anwser == result:
                        print 'Very good!'
                        break
                else:
                        print 'Wrong anwser.'
                        tries += 1
        else:
```

```
            print '%s%s' % (prompt,anwser)

def main():
        while True:
                probe()
                try:
                        goon = raw_input('Next one(y/n)?').strip()[0]
                except IndexError:
                        continue
                except (KeyboardInterrupt,EOFError):
                        goone = 'n'
                if goon in 'nN':
                        break

if __name__ == '__main__':
        main()
```

34.random 随机选取模块

```
>>> import random
>>> nums = [random.randint(1,50) for i in range(2)]
>>> nums
[36, 7]
>>> nums.sort() #排序

>>> nums
[7, 36]
>>> nums.reverse() #反向
排序
>>> nums
[36, 7]
>>> nums.sort(reverse=True)
>>> nums
[36, 7]
```

35.简单 GUI 类的例子

```
[root@localhost Desktop]# cd ~/software/tkinter/
[root@localhost tkinter]# yum -y localinstall * rpm
#!/usr/bin/env python
import Tkinter
from functools import partial
root = Tkinter.Tk()
MyButton = partial(Tkinter.Button,root,fg='white',bg='blue')
b1 = MyButton(text='Button 1')
b2 = MyButton(text='Button 2')
qb = MyButton(text='QUIT',command=root.quit)
b1.pack()
```

```
b2.pack()
qb.pack()
root.mainloop()
```

36.简单 GUI 类的例子（增强版）

```
#!/usr/bin/env python
import Tkinter
from functools import partial
def say_hi(hi):
    def greet():
        print 'hello %s' % hi
    return greet
root = Tkinter.Tk()
MyButton = partial(Tkinter.Button,root,fg='white',bg='blue')
b1 = MyButton(text="Button 1",command=say_hi('world'))
b2 = MyButton(text="Button 2",command=say_hi('tom'))
qb = MyButton(text="quit",bg='red',command=root.quit)
b1.pack()
b2.pack()
qb.pack()
root.mainloop()
```

37.time 模块的扩展

```
>>> import time
>>> import datetime
>>> time.ctime()
'Tue Mar    8 18:00:25 2016'
>>> time.strftime('%Y%m%d')
'20160308'
>>> time.strftime('%Y-%m-%d')
'2016-03-08'
```

38.递归目录脚本

```
>>> import os
>>> os.listdir('/')
['boot', 'dev', 'home', 'proc', 'run', 'sys', 'etc', 'root', 'tmp', 'var', 'usr', 'bin', 'sbin', 'lib', 'lib64', 'media', 'mnt', 'opt', 'srv', 'isos']
>>> os.path.isdir('home')
False
>>> os.path.join('/','home')
'/home'
```

```
#!/usr/bin/env python
import sys
import os
def lsdir(folder):
    content = os.listdir(folder)
    print '%s:\n%s' % (folder,content)
```

```
        print content
        for item in content:
                full_path = os.path.join(folder,item)
                if os.path.isdir(full_path):
                        lsdir(full_path)
if __name__ == '__main__':
        try:
                dir_name = sys.argv[1]
        except IndexError:
                print 'Usage: %s directory' % sys.argv[0]
        else:
                if os.path.isdir(dir_name):
                        lsdir(dir_name)
                else:
                        print 'No such directory'
```

39.递归目录脚本升级直接使用模块

```
>>> import os
>>> os.walk('/home/demo')
<generator object walk at 0x7f266ba09dc0>
>>> list(os.walk('/home/demo'))
[('/home/demo', ['123', '111.txt'], ['passwd']), ('/home/demo/123', [], []), ('/home/demo/111.txt', [], [])]

#!/usr/bin/env python
import os
import sys
folder = sys.argv[1]
for path,dirname,filename in os.walk(folder):
        print '%s:\n%s' % (path,dirname+filename)
```

40.理财程序脚本

```
[root@localhost mypy]# cat initm.py
#!/usr/bin/env python
import cPickle as p
with open('wallet.data','w') as f:
        p.dump(10000,f)
with open('record.txt','w') as f:
        pass

[root@localhost Desktop]# cat finance.py
#!/usr/bin/env python

import time
import cPickle as p

def spend_money(wallet, record):
```

```
    amount = int(raw_input("amount: "))
    comment = raw_input("comment: ")
    when = time.strftime("%Y%m%d")

    with open(wallet) as f:
        balance = p.load(f) - amount

    with open(wallet, 'w') as f:
        p.dump(balance, f)

    with open(record, 'a') as f:
        f.write(
            "%-12s%-8s%-8s%-10s%-30s\n" % (when, amount, 'N/A', balance, comment)
        )

def save_money(wallet, record):
    amount = int(raw_input("amount: "))
    comment = raw_input("comment: ")
    when = time.strftime("%Y%m%d")

    with open(wallet) as f:
        balance = p.load(f) + amount

    with open(wallet, 'w') as f:
        p.dump(balance, f)

    with open(record, 'a') as f:
        f.write(
            "%-12s%-8s%-8s%-10s%-30s\n" % (when, 'N/A', amount, balance, comment)
        )

def query_money(wallet, record):
    with open(wallet) as f:
        print "latest balance: %s" % p.load(f)

    print "%-12s%-8s%-8s%-10s%-30s" % \
          ('time', 'spend', 'save', 'balance', 'comment')

    with open(record) as f:
        for line in f:
            print line,

def show_menu():
    CMDs = {'0': spend_money, '1': save_money, '2': query_money}
    prompt = """(0) spend money
(1) save money
```

```
(2) query
(3) quit
Please input your choice(0123): """
    while True:
        choice = raw_input(prompt).strip()[0]
        if choice not in '0123':
            print "Invalid choice. Try again."
            continue

        if choice == '3':
            break

        CMDs[choice]('/root/PycharmProjects/finance/wallet.data',
                     '/root/PycharmProjects/finance/record.txt')

if __name__ == "__main__":
    show_menu()
```

41.理财改进版本

```
[root@localhost Desktop]# cat finance.py
#!/usr/bin/env python

import time
import cPickle as p

def spend_money(wallet, record, amount, comment):
    when = time.strftime("%Y%m%d")

    with open(wallet) as f:
        balance = p.load(f) - amount

    with open(wallet, 'w') as f:
        p.dump(balance, f)

    with open(record, 'a') as f:
        f.write(
            "%-12s%-8s%-8s%-10s%-30s\n" % (when, amount, 'N/A', balance, comment)
        )

def save_money(wallet, record, amount, comment):
    when = time.strftime("%Y%m%d")

    with open(wallet) as f:
        balance = p.load(f) + amount

    with open(wallet, 'w') as f:
```

```
            p.dump(balance, f)

        with open(record, 'a') as f:
            f.write(
                "%-12s%-8s%-8s%-10s%-30s\n" % (when, 'N/A', amount, balance, comment)
            )

def query_money(wallet, record):
        with open(wallet) as f:
            print "latest balance: %s" % p.load(f)

        print "%-12s%-8s%-8s%-10s%-30s" % \
              ('time', 'spend', 'save', 'balance', 'comment')

        with open(record) as f:
            for line in f:
                print line,

def show_menu():
        CMDs = {'0': spend_money, '1': save_money, '2': query_money}
        prompt = """(0) spend money
(1) save money
(2) query
(3) quit
Please input your choice(0123): """
        wallet = '/root/PycharmProjects/finance/wallet.data'
        record = '/root/PycharmProjects/finance/record.txt'
        args = (wallet, record)
        while True:
            choice = raw_input(prompt).strip()[0]
            if choice not in '0123':
                print "Invalid choice. Try again."
                continue

            if choice == '3':
                break

            if choice in '01':
                amount = int(raw_input("amount: "))
                comment = raw_input("comment: ")
                args = (wallet, record, amount, comment)

            apply(CMDs[choice], args)

if __name__ == "__main__":
        show_menu()
```

十七、模块

1.模块的基本概念

模块是从逻辑上组织 python 代码的形式,当代码量变得相当大时候,最好把代码分成一些有组织的代码段,前提是保证它们的彼此交互.这些代码片段相互间有一定的联系,可能是一个包含数据成员和方法的类,也可能是一组相关但彼此独立的操作函数.这些代码段是共享的,所以 python 允许‘调入’一各模块,允许使用其他模块的属性来利用之前的工作成果,实现代码重用.

2.创建模块

模块物理层面上组织模块的方法是文件,一个文件被看做是一个独立模块,一个模块也可以被看做是一个文件,每一个以.py 作为结尾的 python 文件都是一个模块.模块名称切记不要与系统中已存在的模块重名,模块文件名字去掉后面的扩展名(.py)即为模块名.

3.使用模块(import)

使用 import 导入模块,模块被导入后,程序会自动生成 pyc 的字节码文件以提高性能.可以在一行导入多个模块,但是可读性会下降.模块属性通过‘模块名.属性’的方法调用,如果仅需模块中某些属性,也可以单独导入.导入模块时,可以为模块取别名.

```
>>> import sys
>>> import cPickle as p
>>> import os,string
>>> string.digits
'0123456789'
>>> from random import randint
>>> randint(1,10)
6
```

可以使用错误异常的方式导入,如果 try 下语句报错,则导入 except 语句.

```
>>> try:
...         import cPickle as p
... except ImportError:
...         import pickle as p
...
```

4.名称空间

名称空间就是一个从名称到对象的关系映射集合,给定一个模块名之后,只可能有一个模块被导入到 python 解释器中,所以在不同模块间不会出现名称交叉现象.每个模块都定义了它自己的唯一的名称空间.

```
[root@localhost mypy]# cat foo.py
hi = 'hello'
[root@localhost mypy]# cat bar.py
hi = 'greet'
[root@localhost mypy]# python
>>> import foo
>>> import bar
>>> foo.hi #调用 foo 模块中的额 hi 变量
'hello'
>>> bar.hi #调用 bar 模块中的 hi 变量
'greet'
```

5.模块导入(import)加载(load)

一个模块只被加载(load)一次,无论它被导入(import)多少次,只加载一次可以阻止多重导入时代码被多次执行.

如果两个文件相互导入,放了无限的相互加载.模块加载时,顶层代码会自动执行,所以只将函数放入模块的顶层是良好的编程习惯.

```
[root@localhost ~]# cat foo.py
hi = 'hello'
print hi
[root@localhost ~]# python
>>> import foo
hello #第一次导入,执行 print 语句
>>> import foo #再次导入,print 语句不在执行
>>>
```

6.搜索路径

模块的导入需要一个叫做‘路径搜索’的过程,python 在文件系统‘预定义区域’中查找要调用的模块,搜索路径在 sys.path 中定义.

```
>>> import sys
>>> print sys.path
['', '/usr/lib64/python26.zip', '/usr/lib64/python2.6', '/usr/lib64/python2.6/plat-linux2', '/usr/lib64/python2.6/lib-tk', '/usr/lib64/python2.6/lib-old', '/usr/lib64/python2.6/lib-dynload', '/usr/lib64/python2.6/site-packages', '/usr/lib64/python2.6/site-packages/gst-0.10', '/usr/lib64/python2.6/site-packages/gtk-2.0', '/usr/lib64/python2.6/site-packages/webkit-1.0', '/usr/lib/python2.6/site-packages']
```

7.从 zip 文件中导入

在 2.3 版中,python 加入了从 zip 归档文件导入模块的功能,如果搜索路径中存在一个包含 python 模块(.py,.pyc,或.pyo 文件)的.zip 文件,导入时会把 zip 文件当做目录处理.

```
[root@localhost ~]# cat foo.py
hi = 'hello'
print hi
[root@localhost ~]# zip pymodule.zip foo.py
  adding: foo.py (stored 0%)
[root@localhost ~]# python
>>> import sys #导入 sys 膜,在搜索路径中加入相应的 zip 文件
>>> sys.path.append('/root/pymodule.zip')
>>> import foo #导入 pymodule.zip 压缩文件中的 foo 模块
hello
>>>
```

8.包,目录结构

包是一个有层次的目录文件结构,为平坦的名称空间加入有层次的组织结构,允许程序员吧有联系的模块组合在一起.

包目录下必须有一个__init__.py 文件.

包示例:

```
phone/
        __init__.py
        common_util.py
        voicedata/
                __init__.py #可以为空文件
                post.py
```

```
[root@localhost mypy]# tree aaa/
aaa/
|-- foo.py
`-- __init__.py
0 directories, 2 files
[root@localhost mypy]# cat aaa/__init__.py
def ppp():
        print '*' * 20
[root@localhost mypy]# cat aaa/foo.py
hi = 'hello'
[root@localhost mypy]# python
>>> import aaa
>>> aaa.ppp()
********************
>>> import aaa.foo
>>> aaa.foo.hi
'hello'
```

9.绝对导入

包的使用越来越广泛,很多情况下导入子包会导致和真正的标准库模块发生冲突.因此,所有的导入现在都被认为是绝对的,也就是说这些名字必须通过 python 路径(sys.path 或 PYTHONPATH)来访问.

10.相对导入

绝对导入的特性使得程序员失去了 import 的自由,为此出现的相对导入.因为 impot 语句总是绝对导入的,所以相对导入值应用于 from-import 语句.

```
[root@localhost ~]# ls -R phone/
phone/:
common_util.py    __init__.py    voicedata
phone/voicedata:
__init__.py    post.py
[root@localhost ~]# cat phone/voicedata/post.py
from .. import common_util
```

11.内置模块

hashlib 模块:hashlib 用来替换 md5 和 sha 模块,并使他们的 API 一致,专门提供 hash 算法.包括 md5,sha1,sha224,sha256,sha384,sha512,使用非常简单,方便.

应用示例 1:

```
>>> import hashlib
>>> m = hashlib.md5()
>>> m.update('hello world!')
>>> m.hexdigest()
```

'fc3ff98e8c6a0d3087d515c0473f8677'
应用示例 1:

```
>>> import hashlib
>>> m = hashlib.md5()
>>> with open('/etc/passwd') as f:
...         data = f.read()
...
>>> m = hashlib.md5(data)
>>> m
<md5 HASH object @ 0x17b9d00>
>>> m.hexdigest()
'f4a6515cb73bd44fc2418cd3b4d2ade7'
```

12.hashlib 练习

```
#!/usr/bin/env python
import hashlib
m = hashlib.md5()
with open('/etc/passwd') as f:
        while True:
                data = f.read(100)
                if not data:
                        break
                m.update(data)
print m.hexdigest()
```

13.tarfile 模块

tarfile 模块允许创建,访问 tar 文件,同时支持 gzip,bzip2 格式.

```
[root@localhost ~]# ls /home/demo/
install.log    mima
[root@localhost ~]# python
>>> import tarfile
>>> tar = tarfile.open('/home/demo.tar.gz','w:gz')  创建占位空文件
>>> tar.add('demo')  追加文件,绝对路径也会被打包进去
>>> tar.close()
```

14.tarfile 练习

```
#!/usr/bin/env python
import tarfile
import os
os.chdir('/etc')
tar = tarfile.open('/home/demo.tar.gz','w:gz')
tar.add('/home')
tar.add('passwd')
tar.close()
```

15.模块导入的特性

模块具有一个__name__特殊属性,当模块文件直接执行时,__name__的值为'__main__',当模块被另一个文件导入

时,__name__的值就是该模块的名字.
语句示例:if __name__ == "__main__":

```
[root@localhost test]# vim foo.py
#!/usr/bin/env python
print __name__
[root@localhost test]# chmod +x foo.py
[root@localhost test]# ./foo.py
__main__
[root@localhost test]# python
>>> import foo
foo
```

string.__file__模块,用于查找模块的完整路径

```
>>> import string
>>> string.__file__ #查找模块的完整路径
'/usr/lib64/python2.7/string.pyc'
```

16.生成随机密码的脚本

```
#!/usr/bin/env python
import string
import random
all_chs = string.letters + string.digits
pwd = ''
for i in range(8):
    pwd += random.choice(all_chs)
print pwd
```

17.生成随机密码的脚本(升级版)

```
#!/usr/bin/env python
import string
import random
all_chs = string.letters + string.digits
def gen_pass(num=8):
    pwd = ''
    for i in range(num):
        pwd += random.choice(all_chs)
    return pwd
if __name__ == "__main__":
    print gen_pass()
    print gen_pass(6)
[root@localhost mypy]# python randpass2.py
2BNpjN9F
EDORGz
```

导入随机密码模块

```
[root@localhost mypy]# python
>>> import randpass2
>>> randpass2.__name__
```

```
'randpass2'
>>> randpass2.gen_pass()
'SRVjTrnB'
>>> randpass2.gen_pass(5)
'yduMy'
>>> randpass2.all_chs
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'
```

18.备份程序脚本

```
>>> import time
>>> time.strftime('%a')
'Fri'
>>> time.strftime('%A')
'Friday'

>>> import os
>>> os.path.split('/home/demo/abc')
('/home/demo', 'abc')
>>> os.path.split('/home/demo/abc/aaa')
('/home/demo/abc', 'aaa')
>>> os.path.split('aaa')
('', 'aaa')

>>> os.walk('/home/demo')
<generator object walk at 0x17ce1e0>
>>> list(os.walk('/home/demo'))
[('/home/demo', ['123', '111.txt'], ['passwd']), ('/home/demo/123', [], []), ('/home/demo/111.txt', [], [])]
>>> for path,dir_name,file_name in os.walk('/home/demo'):
...         for each_file in file_name:
...                 print os.path.join(path,each_file)
...
/home/demo/passwd

>>> adict = {'name':'bob','age':20}
>>> adict.get('name')
'bob'
>>> adict.get('namee')
>>> print adict.get('namee')
None

>>> full_path = '/home/demo/abc/aaa/sss.txt'
>>> full_path.split()
['/home/demo/abc/aaa/sss.txt']
>>> full_path.split('/home')
['', '/demo/abc/aaa/sss.txt']
>>> full_path.split('/home')[1].lstrip('/')
```

'demo/abc/aaa/sss.txt'

```
#!/usr/bin/env python

import time
import tarfile
import hashlib
import os
import cPickle as p

def md5check(fname):
    m = hashlib.md5()
    with open(fname) as f:
        while True:
            data = f.read(4096)
            if not data:
                break
            m.update(data)
    return m.hexdigest()

def full_backup(src_dir,dst_dir,md5_file ):
    md5_dict = {}
    base_dir,backup_dir = os.path.split(src_dir)
    back_name = '%s_full_%s.tar.gz' % (backup_dir,time.strftime('%Y%m%d'))
    full_path = os.path.join(dst_dir,back_name)

    os.chdir(base_dir)
    tar = tarfile.open(full_path,'w:gz')
    tar.add(backup_dir)
    tar.close()

    for path,dir_names,file_names in os.walk(src_dir):
        for each_file in file_names:
            abs_path = os.path.join(path,each_file)
            md5_dict[abs_path] = md5check(abs_path)

    with open(md5_file,'w') as f:
        p.dump(md5_dict,f)

def incr_backup(src_dir,dst_dir,md5_file):
    cur_md5 = {}
    base_dir,backup_dir = os.path.split(src_dir)
    back_name = '%s_incr_%s.tar.gz' % (backup_dir,time.strftime('%Y%m%d'))
    full_path = os.path.join(dst_dir,back_name)

    for path,dir_names,file_names in os.walk(src_dir):
```

```
            for each_file in file_names:
                abs_path = os.path.join(path,each_file)
                cur_md5[abs_path] = md5check(abs_path)

        with open(md5_file) as f:
            old_md5 = p.load(f)
        with open(md5_file,'w') as f:
            p.dump(cur_md5,f)

        os.chdir(base_dir)
        tar = tarfile.open(full_path,'w:gz')
        for key in cur_md5:
            #if key not in old_md5 or cur_md5[key] != old_md5[key]:
            if old_md5.get(key) != cur_md5[key]:
                tar.add(key.split(base_dir)[1].lstrip('/'))
        tar.close()

if __name__ == '__main__':
        src_dir = '/home/demo'
        dst_dir = '/root/backup'
        md5_file = '/root/backup/md5.data'
        if time.strftime('%a') == 'Mon':
            full_backup(src_dir,dst_dir,md5_file)
        else:
            incr_backup(src_dir,dst_dir,md5_file)
```

十八、错误和异常

1.基本概念

1)错误

从软件方面来说,错误是语法或逻辑上的,语法错误指示软件的结构上有错误,导致不能被解释器解释或编译器无法编译,这些错误必须在程序执行前纠正,逻辑错误可能是由于不完整或是不合法的输入所致,还可能是逻辑无法生成,计算,或是输出结果需要的过程无法执行.

2)异常

当 python 检测到一个错误时,解释器就会指出当前流已经无法继续执行下去,这时候就会出现异常,异常是因为程序出现了错误而在正常控制流以外采取的行为,这个行为又分为两个阶段,首先是引起异常发生的错误.然后是检测(和采取可能的措施)阶段.

2.异常

python 中的异常:当程序运行时,因为遇到未解的错误而导致终止运行,便会出现 traceback 消息,打印异常.

NameError 未声明/初始化对象

IndexError 序列中没有此索引

SyntaxError 语法错误

KeyboardInterrupt 用户中断执行

EOFError 没有内建输入,到达 EOF 标记

IOError 输入/输出操作失败

3.检测和处理异常

try-except 语句,定义了进行异常监控的一段代码,并且提供了处理异常的机制.

语法示例:

```
try:
    try_suite #监控这里的异常
except Exception[,reason]:
    except_suite #异常处理代码
```

应用:打开不存在的文件.

```
>>> try:
...     f = open('foo.txt')
... except IOError:
...     print 'No such file'
...
No such file
```

示例,使用错误和异常的位置

原文件

```
#!/usr/bin/env python
import time
print time.ctime()
for i in xrange(10000000):
    a = 10 + 20
print time.ctime()
```

使用位置

```
#!/usr/bin/env python
import time
import sys
for i in range(1,11):
    sys.stdout.write('\r%s' % i)
    sys.stdout.flush()
    try:
        time.sleep(1)
    except KeyboardInterrupt:
        pass
```

4.带有多个 expect 的 try 语句

可以把多个 except 语句连接在一起,处理一个 try 块中可能发生的多个异常.

语法示例:

```
>>> try:
...     data = int(raw_input('input a number:'))
... except KeyboardInterrupt:
...     print 'user cancelled'
... except ValueError:
...     print 'you must input a number!'
...
input a number:hello
you must input a number!
```

应用示例

```
#!/usr/bin/env python
try:
    num = int(raw_input('Number: '))
except ValueError,e:
    print "Error",e
except (KeyboardInterrupt,EOFError):
    print 'User cancelled'
```

5.捕获所有异常:BaseException

如果出现的异常没有出现在指定要捕捉的异常列表中,程序仍然会中断,可以使用.在异常继承的树结构中,BaseException 是在最顶层的,所以使用它可以捕获任意类型的异常.

```
>>> try:
...     data = int(raw_input('input a number:'))
... except BaseException:
...     print '\nsome error'
...
input a number:     [ctrl+c]
some error
```

6.异常参数

异常也可以有参数,异常引发后会被传递给异常处理器.当异常被引发后参数是作为附加帮助信息传递给异常处理器的.

```
>>> print 10/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>> try:
...     data = 10/0
... except ZeroDivisionError,e: #并不是只能是 e,可以是任何变量
...     print 'Error:',e
...
Error: integer division or modulo by zero
```

7.else 子句

在 try 范围中没有异常被检查到时,执行 else 子句.在 else 范围中的任何代码运行前,try 范围中的所有代码必须完全成功.

```
>>> try:
...     res = 10 / int(raw_input('input a number:'))
... except BaseException,e:
...     print 'Error:',e
... else:
...     print res
...
input a number:5
2
```

写成脚本的样子

```
#!/usr/bin/env python

try:
        num = 100 / int(raw_input('number: '))
except (ValueError,ZeroDivisionError),e:
        print 'Error:',e
except (KeyboardInterrupt,EOFError):
        print 'exit'
else:
        print num
```

8.finally 子句

finally 字句是无论异常是否发生,是否捕捉都会执行的一段代码.如果打开文件后,因为发生异常导致文件没有关闭,可能会发生数据损坏,使用 finally 可以保证文件总说正常的关闭.

```
>>> try:
...         try:
...                 ccfile = open('carddata.txt','r')
...                 txns = ccfile.readlines()
...         except IOError:
...                 log.write('no txns this month\n')
... finally:
...         if ccfile:
...                 ccfile.close()
```

脚本示例:

```
#!/usr/bin/env python

try:
        num = 100 / int(raw_input('number: '))
except (ValueError,ZeroDivisionError),e:
        print 'Error:',e
except (KeyboardInterrupt,EOFError):
        print 'exit'
else:
        print num
finally:
        print 'done'
```

9.with 子句

with 语句是用来简化代码的,在将打开文件的操作放在 with 语句中,代码块结束后,文件将自动关闭

```
>>> with open('ip.py') as f:
...         data = f.readlines()
...
>>> f.closed #检测文件是否关闭.
True
```

语法示例:

```
>>> with open('/etc/hosts') as f:
...         for line in f:
...                 print line,
...
127.0.0.1       localhost localhost.localdomain localhost4 localhost4.localdomain4
::1             localhost localhost.localdomain localhost6 localhost6.localdomain6
```

10.触发异常:raise

要想引发异常,最简单的形式就是输入关键字 raise,后面跟要引发的异常的名称,执行 raise 语句时,python 会创建指定的异常类的一个对象.raise 语句还可指定异常对象进行初始的参数.

```
>>> number = 3
>>> try:
...         if number < 10:
...                 raise ValueError,'Number is too smalle.'
... except ValueError,e:
...         print 'Error:',e
...
Error: Number is too smalle.
```

脚本示例:

```
#!/usr/bin/env python

for i in range(100):
        if i > 10:
                raise ValueError,'Number too big'
        print i,
```

11.断言:assert

断言是一句必须等价于布尔值为真的判定,此外,发生异常也就意味着表达式为假.

```
>>> number = 3
>>> try:
...         assert number > 10,'number is too small'
... except AssertionError,e:
...         print 'Error:',e
...
Error: number is too small
```

```
>>> assert 3 > 0,'asefdfe'
>>> assert 3 < 0,'asefdfe'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError: asefdfe
```

12.简化除法判断的脚本

```
#!/usr/bin/env python

try:
```

```
    num = 100 / int(raw_input('number: '))
except (ValueError,ZeroDivisionError),e:
    print 'Error:',e
except (KeyboardInterrupt,EOFError):
    print 'exit'
```

13.错误异常版,将 linux 文件转为 win 文件

```
#!/usr/bin/env python

import sys

def unix2dos(fname):
    fobj = open(fname)
    newf = open(fname + '.txt','w')
    for line in fobj:
        newf.write(line.strip('\r\n') + '\r\n')
    fobj.close()
    newf.close()
if __name__ == '__main__':
    try:
        unix2dos(sys.argv[1])
    except IndexError:
        print 'usage: %s filename' % sys.argv[0]
```

十九、面向对象(类)

1.oop 介绍

编程的发展已经从简单控制流中按步的指令序列进入到更有组织的方式中,结构化的或过程性编程可以让我们把程序组织成逻辑块,以便重复或重用.面向对象编程增强了结构化编程,实现了数据与动作的融合,数据层和逻辑层由一个可用以创建这些对象的简单抽象层来描述.

2.常用术语

1)抽象/实现:抽象指对现实世界问题和实体的本质表现,行为和特征建模,建立一个相关的子集,可以用于描绘程序结构,从而实现这种模型.

2)封装/接口:封装描述了对数据/信息进行隐藏的观念,它对数据属性提供接口和访问函数,客户端根本就不用需要知道在封装后,数据属性是如何组织的,这就需要在设计时,对数据提供相应的接口.

3)合成:合成扩充了对类的描述,使得多个不同的类和成为一个大的类,进来解决现实问题.

4)派生/继承:派生描述了子类的创建,新类保留已存类类型中所有需要的数据和行为,但允许修改或者其他的自定义操作,都不会修改原类的定义.

5)多态:指出了对象如何通过他们共同的属性和动作来操作及访问,而不需要考虑他们具体的类.

6)泛化/特化:泛化表示所有子类与其父类及祖先类有一样的特点,特化描述所有子类的自定义,也就是什么属性让它与其祖先类不同.

7)自省/反射:自省表示给予程序员某种能力来进行像‘手工类型检查’的工作,它也被称为反射,这个性质展示了某对象是如何在运行期取得自身信息的

3.创建类

类是一种数据结构,我们可以用它来定义对象,对象把数据值和行为特性融合在一起,python 使用 class 关键字来创建

类.通常类名的第一个字母大写.
语法示例:

```
class ClassName(bases):
    'class documentation string' #‘类文档字符串’
    class_suite #类体
```

语法应用:

```
>>> class MyClass(object):
...     def ppp(self):
...         print '*' * 20
...
>>> c = MyClass()
>>> c.ppp()
********************
```

4.类的数据属性

数据属性仅仅是所定义的类的变量.这种属性已是静态变量,或者是静态数据,它们表示这些数据是它们所属的类对象绑定的,不依赖于任何类实例.静态成员通常仅用来跟踪与类相关的值.

```
>>> class C(object):
...     foo = 100
...
>>> print C.foo
100
>>> C.foo += 1
>>> print C.foo
101
```

5.特殊的类属性

C.__name__ 类 C 的名字(字符串)
C.__doc__ 类 C 的文档字符串
C.__bases__ 类 C 的所有父类构成的元组
C.__dict__ 类 C 的属性
C.__module__ 类 C 定义所在的模块
C.__class__ 实例 C 对应的类

6.实例

如果说类是一种数据结构定义类型,那么实例则声明了一个这种类型的变量.类被实例化得到实例,该实例的类型就是这个被实例化的类.创建实例与调用函数类似,调用了一个类就创建了它的一个实例.

```
>>> class C(object):
...     foo = 100
...
>>> c = C()
>>> print c
<__main__.C object at 0x7f5f8e7dc2d0>
```

7.实例属性

实例仅拥有数据属性,数据属性只是与某个类的实例相关联的数据值,并且可通过句点属性标识法类访问.设置实例的属性可以在实例创建后任意时间进行,也可以在能访问实例的代码中进行.

```
>>> class C(object):
...         pass
...
>>> c = C()
>>> c.hi = 'hello'
>>> print c.hi
hello
```

8.特殊的实例属性

I.__class__ 实例化 I 的类

I.__dict__ I 的属性

```
>>> class C(object):
...         pass
...
>>> c = C()
>>> c.hi = 'hello'
>>> c.__dict__
{'hi': 'hello'}
>>> print c.__class__
<class '__main__.C'>
```

9.类与实例属性对比

类属性仅是类相关的数据值,类属性和实例无关.静态成员(类的数据属性)不会因为实例而改变他们的值,除非实例中显式改变它.类和实例都是名字空间,各不相同.

```
>>> class C(object):
...         version = 1.0
...
>>> c = C()
>>> c.version += 0.1
>>> print 'in C version=%s,in c version=%s' % (C.version,c.version)
in C version=1.0,in c version=1.1
>>> c.version += 0.1
>>> print 'in C version=%s,in c version=%s' % (C.version,c.version)
in C version=1.0,in c version=1.2
```

10.组合及派生

__init__方法:__init__()是实例创建后第一个被调用的方法,设置实例的属性可以在实例创建后任意时间进行,但是通常情况下优先在__init__方法中实现.

语法示例:

```
[root@localhost ~]# vim AddrBook.py
#!/usr/bin/env python
class AddrBook(object):
        def __init__(self,nm,ph):
                self.name = nm
                self.phone = ph
>>> import AddrBook
>>> bob = AddrBook.AddrBook('bob green','12344556678')
```

```
>>> bob.name
'bob green'
>>> bob.phone
'12344556678'
```
语法应用 1:
```
#!/usr/bin/env python
class AddrBook(object):
        def __init__(self,nm,ph):
                self.name = nm
                self.phone = ph
tom = AddrBook('Tom Smith','123456789')
jerry = AddrBook('Jerry Green','12335346679')
print tom.phone
print jerry.phone
[root@localhost mypy]# python addrbook.py
123456789
12335346679
```
语法应用 2:
```
#!/usr/bin/env python
class AddrBook(object):
        version = 1.0
        def __init__(self,nm,ph):
                self.name = nm
                self.phone = ph
tom = AddrBook('Tom Smith','123456789')
jerry = AddrBook('Jerry Green','12335346679')
tom.version += 2
print tom.version
print jerry.version
print AddrBook.version
[root@localhost mypy]# python addrbook.py
3.0
1.0
1.0
```

11.绑定方法

方法仅仅是类内部定义的函数,方法只有在其所属的类拥有实例时,才能被调用.任何一个方法定义中的第一个参数都是变量 self,它表示调用此方法的实例对象.

```
[root@localhost ~]# vim AddrBook.py
#!/usr/bin/env python
class AddrBook(object):
        def getPhone(self):
                return self.phone
>>> import AddrBook
>>> AddrBook.AddrBook.getPhone()
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
TypeError: unbound method getPhone() must be called with AddrBook instance as first argument (got nothing instead) #未绑定方法 getPhone()必须以 AddrBook 实例作为第一个参数来调用(没有)
```

应用实例 1:

```
#!/usr/bin/env python
class AddrBook(object):
    version = 1.0
    def __init__(self,nm,ph):
        self.name = nm
        self.phone = ph
    def get_name(self):
        return self.name
    def get_phone(self):
        return self.phone
tom = AddrBook('Tom Smith','123456789')
jerry = AddrBook('Jerry Green','12335346679')
print tom.get_name()
print jerry.get_phone()
[root@localhost mypy]# python addrbook.py
Tom Smith
12335346679
```

应用实例 2:

```
#!/usr/bin/env python
class AddrBook(object):
    version = 1.0
    def __init__(self,nm,ph):
        self.name = nm
        self.phone = ph
    def get_name(self):
        return self.name
    def get_phone(self):
        return self.phone
    def update_ph(self,newph):
        self.phone = newph
        print 'phone number updated.'
tom = AddrBook('Tom Smith','123456789')
jerry = AddrBook('Jerry Green','12335346679')
print jerry.get_phone()
jerry.update_ph('1909430495')
print jerry.get_phone()
print AddrBook.get_name(tom)
[root@localhost mypy]# python addrbook.py
12335346679
phone number updated.
1909430495
Tom Smith
```

12.非绑定方法

调用非绑定方法并不经常用到,需要调用一个没有实例的类中的方法的一个主要场景是你在派生一个子类,而且要覆盖父类的方法.

语法示例:

```
[root@localhost ~]# vim AddrBook.py
#!/usr/bin/env python
class AddrBook(object):
        def __init__(self,nm,ph):
                self.name = nm
                self.phone = ph
        def getPhone(self):
                return self.phone
class EmplAddrBook(AddrBook):
        def __init__(self,nm,ph,id,em):
                AddrBook.__init__(self,nm,ph)
                self.emid = id
                self.email = em
```

应用实例:

```
#!/usr/bin/env python
class AddrBook(object):
        def __init__(self,nm,ph):
                self.name = nm
                self.phone = ph
        def get_name(self):
                return self.name
        def get_phone(self):
                return self.phone
        def update_ph(self,newph):
                self.phone = newph
                print 'phone number updated.'
class EmplAddrBook(AddrBook):
        def __init__(self,nm,ph,eid,em):
                AddrBook.__init__(self,nm,ph)
                self.eid = eid
                self.email = em
        def get_email(self):
                return self.email
        def update_email(self,newem):
                self.email = newem
                print 'email updated'
tom = EmplAddrBook('Tom Smit','11122334353','1001','taom@tedu.cn')
print tom.get_name()
print tom.get_email()
[root@localhost mypy]# python addrbook.py
Tom Smit
```

taom@tedu.cn

13.组合

类被定义后,目标就是要把它当成一个模块来使用,并把这些对象嵌入到你的代码中去.组合就是让不同的类混合并加入到其它类中来增加功能和代码重用行.可以在一个大点的类中创建其类的实例,实现一些其它属性和方法来增强对原来的类对象.

应用实例:

```
#!/usr/bin/env python
class Info(object):
    def __init__(self,ph,em,qq):
        self.phone = ph
        self.email = em
        self.qq = qq
    def get_phone(self):
        return self.phone
    def update_phone(self):
        self.phone = newph
class AddrBook(object):
    def __init__(self,nm,ph,em,qq):
        self.name = nm
        self.info = Info(ph,em,qq)
tom = AddrBook('tom Smith','134235345','tom@tedu.cn','2345465')
print tom.info.get_phone()
[root@localhost mypy]# python addrbook2.py
134235345
```

14.实现组合

创建符合对象,应用组合可以实现附加的功能.例如,通过组合实现上述地址簿功能的增强.

```
[root@localhost ~]# vim Info.py
#!/usr/bin/env python
class Info(object):
    def __init__(self,ph,em,qq):
        self.phone = ph
        self.email = em
        self.qq = qq
    def updatePhone(self,newph):
        self.phone = newph
class Contact(object):
    def __init__(self,name,ph,em,qq):
        self.name = name
        self.info = Info(ph,em,qq)
```

15.子类和派生

子类:当类之间有着显著不同,并且较小的类是较大的类所需的组件是组合表现的很好,但当设计‘相同的类但有一些不同的功能’时,派生就是一个更加合理的选择了.oop 的更强大方面之是能使用一个已经定义好的类,扩展它或者对其进行修改,而不会影响系统中使用现存类的其他代码片段.ood(面向对象设计)允许类特征在子孙类或子类中进行继承.创建子类只需要在圆括号中写明从哪个父类继承即可.

语法示例:

```
>>> class A(object):
...     def printStar(self):
...         print '*' * 20
...
>>> class B(A):
...     pass
...
```

16.继承

继承描述了基类的属性如何‘遗传’给派生类.子类可以继承它的基类的任何属性,不管是数据属性还是方法.

语法示例:

```
>>> class A(object):
...     def printStar(self):
...         print '*' * 20
...
>>> class B(A):
...     pass #类 B 并没有定义任何方法
...
>>> b = B() #b 是 B 的实例
>>> b.printStar() #B 继承了类 A 的方法,实例 b 也就具备了该功能
**********
```

17.通过继承覆盖方法

如果子类中有和父类同名的方法,父类方法将被覆盖.如果需要访问父类的方法,则要调用一个未绑定的父类方法,明确给出子类的实例.

语法示例:

```
>>> class P(object):
...     def foo(self):
...         print 'in P-foo'
...
>>> class C(P):
...     def foo(self):
...         print 'in C-foo'
...
>>> c = C()
>>> c.foo()
in C-foo
>>> P.foo(c)
in P-foo
```

18.多重继承

python 允许多重继承,即一个类可以是多个父类的子类,子类可以拥有所有父类的属性

语法示例:

```
>>> class A(object):
...     def foo(self):
...         print 'foo method'
```

```
...
>>> class B(object):
...         def bar(self):
...                 print 'bar method'
...
>>> class C(A,B):
...         pass
...
>>> c = C()
>>> c.foo()
foo method
>>> c.bar()
bar method
```

19.类和实例的内建函数

issubcalass() 判断一个类是另一个类的子类或子孙类
isinstance() 在判定一个对象是否是另一个给定类的实例
hasattr() 判断一各对象是否有一个特定的属性
getattr() 获得一个对象的属性值
setattr() 设置一个对象的属性
delattr() 删除一个对象的属性

20.私有化

python 为类元素(属性和方法)的私有性提供了初步的形式,由双下线开始的属性在运行时被‘混淆’,所以直接访问时不允许的.
语法示例:

```
>>> class C(object):
...         def __init__(self,nm):
...                 self.__name = nm
...         def getName(self):
...                 return self.__name
...
>>> c = C('bob')
>>> c.getName()
'bob'
>>> c.__name #私有化的数据不能再外部直接使用
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'C' object has no attribute '__name'
```

应用实例 1:

```
>>> class P(object):
...         def __init__(self,nm):
...                 self.name = nm
...
>>> p = P('tom')
>>> p.name
```

```
'tom'
>>> class M(object):
...         def __init__(self,nm):
...                 self.__name = nm
...
>>> m = M('jerry')
>>> m.__name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'M' object has no attribute '__name'
>>> m._M__name
'jerry'
>>> m._M__name = 'alice'
>>> m._M__name
'alice'
```

应用实例 2:

```
>>> class Book(object):
...         def __init__(self,title,author):
...                 self.title = title
...                 self.author = author
...
>>> python = Book('core python','Green')
>>> python
<__main__.Book object at 0x120a590>
>>> print python
<__main__.Book object at 0x120a590>
>>> class Book(object):
...         def __init__(self,title,author):
...                 self.title = title
...                 self.author = author
...         def __str__(self):
...                 return self.title
...
>>> python = Book('core python','Green')
>>> print python
core python
```

应用实例 3:

```
>>> class MyClass(object):
...         def say_hi(self):
...                 print 'hello world'
...         def __call__(self):
...                 self.say_hi()
...
>>> a = MyClass()
>>> a()
hello world
```

应用实例 4:

```
>>> class MyCount(object):
...         def __init__(self,num):
...                 self.num = num
...         def __add__(self,other):
...                 return self.num + other
...
>>> mc = MyCount(10)
>>> print mc + 3
13
>>> print 3 + mc
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'MyCount'
>>> class MyCount(object):
...         def __init__(self,num):
...                 self.num = num
...         def __add__(self,other):
...                 return self.num + other
...         def __radd__(self,other):
...                 return self.num + other
...
>>> mc = MyCount(10)
>>> print 3 + mc
13
```

21.编写酒店类脚本

```
#!/usr/bin/env python
class Hotel(object):
        def __init__(self,rm = 180,cf = 1.0,br = 15):
                self.room = rm
                self.cutoff = cf
                self.breakfast = br
        def calc_all(self,days = 1):
                return (self.room * self.cutoff + self.breakfast) * days
std_room = Hotel()
print std_room.calc_all()
print std_room.calc_all(3)
big_bed = Hotel(rm = 230,cf = 0.9)
print big_bed.calc_all()
print big_bed.calc_all(2)
[root@localhost mypy]# python hotel.py
195.0
585.0
222.0
444.0
```

二十、正则表达式(re)

1.match 函数

尝试用正则表达式模式从字符串的开头匹配,如果匹配成功,则返回一个匹配对象,否则返回 None.

```
>>> import re #导入 re(正则)模块
>>> m = re.match('foo','food')
>>> print m
<_sre.SRE_Match object at 0x7f188719e4a8> #成功匹配
>>> m = re.match('foo','seafood') #match 只能匹配字符串的开头
>>> print m
None #未能匹配
```

2.search 函数

在字符串中查找正则表达式模式的第一次出现,如果成功匹配,则返回一个匹配对象,否则返回 None.

```
>>> import re
>>> m = re.search('foo','food')
>>> print m
<_sre.SRE_Match object at 0x7f188719e4a8>
>>> m = re.search('foo','seafood') #可以匹配在字符串中间的模式
>>> print m
<_sre.SRE_Match object at 0x7f188719e920>
>>> m = re.search('foo','seafoo')
>>> print m
<_sre.SRE_Match object at 0x7f188719e4a8>
>>> m = re.search('foo','seafo')
>>> print m
None
```

3.group 方法

使用 match 或 search 匹配成功后,返回的匹配对象可以通过 group 方法获得匹配内容

```
>>> import re
>>> m = re.match('foo','food')
>>> print m.group()
foo
>>> m = re.search('foo','seafood')
>>> print m.group()
foo
```

4.findall 函数

在字符串中查找正则表达式模式的所有(非重复)出现,返回一个匹配对象的列表

```
>>> import re
>>> m = re.search('foo','seafood is food')
>>> print m.group() #serach 只匹配模式的第一次出现
foo
>>> m = re.findall('foo','seafood is food') #可以匹配全部匹配的出现
>>> print m
```

['foo', 'foo']

5.finditer 函数

和 findall()函数有相同的功能,但返回的不是列表而是迭代器,对于每个匹配,该迭代器返回一个匹配对象

```
>>> import re
>>> m = re.finditer('foo','serfood is food')
>>> for item in m:
...         print item.group()
...
foo
foo
```

6.compile 函数

对正则表达式模式进行编译,返回一个正则表达式对象.不是必须要用这种方式,但是在大量匹配的情况下,可以提升效率.

```
>>> import re
>>> patt = re.compile('foo')
>>> m = patt.match('food')
>>> print m.group()
foo
```

7.split 方法

根据正则表达式的分隔符把字符分割为一个列表,并返回成功匹配的列表.字符串也有类似的方法,但是正则表达式更加灵活.

```
>>> import re
>>> mylist = re.split('\.|-','hello-world.data') #使用.和-作为字符串的分隔符
>>> print mylist
['hello', 'world', 'data']
```

8.sub 方法

把字符串中所有匹配正则表达式的地方替换成新的字符串

```
>>> import re
>>> m = re.sub('X','Mr.Smith','attn:X\nDear X')
>>> print m
attn:Mr.Smith
Dear Mr.Smith
```

9.在 vi 中使用正则,给 mac 地址加冒号

```
[root@localhost mypy]# cat ip-M.data
192.168.1.1         000C29AB3456
192.168.1.2         32843ABD2AB3
:%s/\(..\)\(..\)\(..\)\(..\)\(..\)\(..\)$/\1:\2:\3:\4:\5:\6/
[root@localhost mypy]# cat ip-M.data
192.168.1.1         00:0C:29:AB:34:56
192.168.1.2         32:84:3A:BD:2A:B3
```

10.正则表达式(匹配)

1)匹配单个字符

. 匹配任意字符(换行符除外)

[x-y] 匹配字符串组里的任意字符,如果要匹配-,就不能放中间,要放开头或结尾

[xyz] 匹配指定的单个字符

[^x-y] 匹配不在字符串组里的任意字符,如果要匹配^,就不能放在开头

\d 匹配任意数字,与[0-9]同义

\D 匹配任意非数字

\w 匹配任意数字字母字符,与[0-9a-zA-Z_]同义

\W 匹配任意非数字字母字符

\s 匹配空白字符,与[\r\v\f\t\n]同义

\S 匹配非空白字符

2)匹配一组字符

literal 匹配字符串的值

re1|re2 匹配正则表达式 re1 或 re2,|有时需要转义\|

* 匹配前面出现的正则表达式零次或多次(匹配*前面的)

+ 匹配前面出现的正则表达式一次或多次(至少一次)

? 匹配前面出现的正则表达式零次或一次

{M,N} 匹配前面出现的正则表达式至少 M 次最多 N 次

{M} 匹配前面出现的正则表达式出现 M 次

{,N} 匹配前面出现的正则表达式最多出现 N 次

{M,} 匹配前面出现的正则表达式最少出现 M 次

3)其他元字符

^ 匹配字符串的开始

$ 匹配字符串的结尾

\b 匹配单词的边界

() 对正则表达式分组

\nn 匹配以保存的子组

```
(),\nn 语法示例
>>> import re
>>> m = re.search('(((f)o)o)d','seafood is food')
>>> m.group()
'food'
>>> m.group(1)
'foo'
>>> m.group(2)
'fo'
>>> m.group(3)
'f'
>>> m = re.search('(fo)(od)','seafood is food')
>>> m.group()
'food'
>>> m.group(1)
'fo'
>>> m.group(2)
'od'
```

11.贪婪匹配

*,+和？都是贪婪匹配操作符,在其后加上？可以取消其贪婪匹配的行为.正则表达式匹配对象通过 groups 函数获取子组.

应用实例 1:

```
>>> import re
>>> data = 'My phone number is:15088998899'
>>> m = re.search('.+(\d+)',data)
>>> print m.groups()
('9',)
>>> m = re.search('.+?(\d+)',data)
>>> m.groups()
('15088998899',)
```

应用实例 2:

```
>>> import re
>>> data = 'My phone number is : 15099923432'
>>> m = re.search('.+(\d+)',data)
>>> m.group()
'My phone number is : 15099923432'
>>> m.group(1)
'2'
>>> m = re.search('.+?(\d+)',data)
>>> m.group()
'My phone number is : 15099923432'
>>> m.group(1)
'15099923432'
```

12.分析 apache 访问日志脚本

1)无排序,初始版本

```
#!/usr/bin/env python
import re
def count_patt(fname,patt):
    patt_dict = {}
    with open(fname) as f:
        for line in f:
            m = re.search(patt,line)
            if m:
                key = m.group()
                patt_dict[key] = patt_dict.get(key,0) + 1
    return patt_dict
if __name__ == '__main__':
    filename = '/var/log/httpd/access_log'
    ip_patt = '^\d+\.\d+\.\d+\.\d+'
    br_patt = 'MSIE|Firefox'
    ip_count = count_patt(filename,ip_patt)
    br_count = count_patt(filename,br_patt)
```

```
    print ip_count
    print br_count
```

2)有排序,改进版

items()可以将字典转换为列表元祖,dict（）可以将列表元祖转换为字典

```
>>> adict = {'name':'bob','age':'30'}
>>> adict.items()
[('age', '30'), ('name', 'bob')]
>>> dict([('age', '30'), ('name', 'bob')])
{'age': '30', 'name': 'bob'}
```

```
#!/usr/bin/env python
import re
def count_patt(fname,patt):
    patt_dict = {}
    with open(fname) as f:
        for line in f:
            m = re.search(patt,line)
            if m:
                key = m.group()
                patt_dict[key] = patt_dict.get(key,0) + 1
    return patt_dict
def sort(adict):
    alist = []
    items = adict.items()
    for i in range(len(items)):
        greater = items[0]
        for remain in items[1:]:
            greater = greater if greater[1] >= remain[1] else remain
        alist.append(greater)
        items.remove(greater)
    return alist
if __name__ == '__main__':
    filename = '/var/log/httpd/access_log'
    ip_patt = '^\d+\.\d+\.\d+\.\d+'
    br_patt = 'MSIE|Firefox'
    ip_count = count_patt(filename,ip_patt)
    br_count = count_patt(filename,br_patt)
    sort_ip = sort(ip_count)
    print ip_count
    print sort_ip
    print br_count
```

3)用模块排序版

collections 模块使用

```
>>> import collections
```

```
>>> c = collections.Counter()
>>> c
Counter()
>>> c.update('hello')
>>> c
Counter({'l': 2, 'h': 1, 'e': 1, 'o': 1})
>>> c.update(['hello'])
>>> c
Counter({'l': 2, 'h': 1, 'e': 1, 'o': 1, 'hello': 1})
>>> c.update(['hello'])
>>> c.update(['hello'])
>>> c.update(['hello'])
>>> c
Counter({'hello': 4, 'l': 2, 'h': 1, 'e': 1, 'o': 1})
```

```
#!/usr/bin/env python
import re
import collections
def count_patt(fname,patt):
    patt_dict = collections.Counter()
    with open(fname) as f:
        for line in f:
            m = re.search(patt,line)
            if m:
                key = m.group()
                patt_dict.update([key])
    return patt_dict
if __name__ == '__main__':
    filename = '/var/log/httpd/access_log'
    ip_patt = '^\d+\.\d+\.\d+\.\d+'
    br_patt = 'MSIE|Firefox'
    ip_count = count_patt(filename,ip_patt)
    br_count = count_patt(filename,br_patt)
    print ip_count
    print ip_count.most_common(3)
```

4)类方法版本

```
#!/usr/bin/env python
import re
import collections
class CountWeb(object):
    def __init__(self,patt):
        self.patt = patt
    def count_patt(self,fname):
        patt_dict = collections.Counter()
        with open(fname) as f:
            for line in f:
```

```
                m = re.search(self.patt,line)
                if m:
                    patt_dict.update([m.group()])
        return patt_dict
if __name__ == '__main__':
    filename = '/var/log/httpd/access_log'
    ip_patt = '^(\d+\.){3}\d+'
    c = CountWeb(ip_patt)
    print c.count_patt(filename)
```

二十一、网络编程

1.socket 模块:C/S 架构

服务器是一个软件或硬件,用于提供客户需要的"服务".硬件上,客户端常见的就是平时所使用的 PC 机,服务器常见的有联想,DELL 等厂商生产的各种系列的服务器.软件上,服务器提供的服务主要是程序的运行,数据的发送与接收,合并,升级或其它的程序或数据的操作.

2.套接字

套接字是一种具有‘通讯端点’概念的计算机网络数据结构.套接字起源于 20 世纪 70 年代加利福尼亚大学伯克利分校版本的 linux.一种套接字是 Unix 套接字,其"家族名"为 AF_UNIX.另一种套接字基于网络的,"家族名"为 AF_INET.如果把套接字比作电话的接口,那么主机与端口就像区号与电话号码的一对组合.

4.面向连接与无连接

无论你使用的是哪一种地址家族,套接字的类型只有两种,一种是面向连接的套接字,另一种的无连接的套接字.面向连接的主要协议就是传输控制协议 TCP,套接字类型为 SOCK_STREAM.无连接的主要协议用户数据报协议 UDP,套接字类型为 SOCK_DGRAM.python 中使用 socket 模块的 socket 函数实现套接字的创建.

4.socket 函数与方法

创建 TCP 服务器,创建 TCP 服务器的主要步骤如下:

1,创建服务器套接字:s = socket.socket()

2,绑定地址到套接字:s.bind()

3,启动监听:s.listen()

4,接受客户连接:s.accept()

5,与客户端通信:recv()/send()

6,关闭套接字:s.close()

应用示例:

```
#!/usr/bin/env python
import socket
host = ''
port = 12345
addr = (host,port)
s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
s.bind(addr)
s.listen(5)
cli_sock,cli_addr = s.accept()
print 'Client from: ', cli_addr
```

```
data = cli_sock.recv(2048)
print data
cli_sock.send('Have fum!\n')
cli_sock.close()
s.close()
[root@localhost Desktop]# netstat -tlnp | grep :12345
tcp        0      0 0.0.0.0:12345               0.0.0.0:*                   LISTEN      28412/python
[root@localhost Desktop]# telnet 192.168.0.123 12345
Trying 192.168.0.123...
Connected to 192.168.0.123.
Escape character is '^]'.
hello
Have fum!
Connection closed by foreign host.
```

5.创建 TCP 时间戳服务器

应用示例:

```
#!/usr/bin/env python
import socket
import time
host = ''
port = 12345
addr = (host,port)
s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
s.bind(addr)
s.listen(5)
while True:
    cli_sock,cli_addr = s.accept()
    print 'Client from: ', cli_addr
    while True:
        data = cli_sock.recv(2048)
        if not data.strip():
            break
        cli_sock.send('[%s] %s' % (time.ctime(),data))
    cli_sock.close()
s.close()
```

6.创建 TCP 客户端,创建 TCP 客户端的步骤主要如下:

1,创建客户端套接字:cs = socket.socket()

2,尝试连接服务器:cs.connect()

3,与服务通信:cs.send()/cs.recv()

4,关闭客户端套接字:cs.close()

7.创建 TCP 时间戳客户端

```
#!/usr/bin/env python
import socket
```

```
host = '192.168.0.123'
port = 12345
addr = (host,port)
c = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
c.connect(addr)
while True:
    try:
        data = raw_input('> ')
    except KeyboardInterrupt:
        print
        break
    if not data.strip():
        break
    c.send(data)
    print c.recv(2048)
c.close()
```

8.创建 UDP 服务器,创建 UDP 服务器的主要步骤如下:

1,创建服务器套接字:s = socket.socket()
2,绑定服务器套接字:s.bind()
3,接收,发送数据:s.recvfrom()/ss.sendto()
4,关闭套接字:s.close()

9.创建 UDP 时间戳服务器

```
#!/usr/bin/env python
import socket
import time
host = ''
port = 12345
addr = (host,port)
s = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
s.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
s.bind(addr)
while True:
    data,cli_addr = s.recvfrom(2048)
    print 'recv from:', cli_addr
    s.sendto('[%s] %s' % (time.ctime(),data),cli_addr)
s.close()
```

10.创建 UDP 客户端,创建 UDP 客户端的步骤主要如下:

1,创建客户套接字:cs = socket.socket()
2,与服务器通信:cs.sendto()/cs.recvfrom()
3,关闭客户端套接字:cs.close()

11.创建 UDP 时间戳客户端

```
#!/usr/bin/env python
import socket
```

```
host = '192.168.0.123'
port = 12345
addr = (host,port)
c = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
while True:
    data = raw_input('> ')
    if not data.strip():
        print
        break
    c.sendto(data,addr)
    print c.recvfrom(2048)[0]
c.close()
```

12.xinetd 服务器

xinetd 可以统一管理很多服务进程,它能够:绑定,侦听,和接受对自服务器每个端口的请求.有客户端访问时,调用相应的服务器程序响应.节约了系统内存资源.同时响应多个客户端的连接请求.windows 系统没有该功能.多数 unix 系统使用的是 inetd 实现相同功能.

安装 xinetd:

[root@localhost mypy]# yum -y install xinetd

[root@localhost mypy]# cat /etc/xinetd.d/time-stream

配置文件解析:

flags 如果只指定 NAMEINARGS,那么它就使参数和 intetd 一样地传递

type 如果服务不在/etc/services 中,则使用 UNLISTED,否则可以忽略这一行

port 如果 type=UNILISTED,则在这里指定端口号

socket_type 如果是 TCP,使用 stream,如果是 UDP,使用 dgram

protocol 指定是 TCP,还是 UDP

wait TCP 设置为 no,对于 UDP,如果服务器连接远程主机并为不同客户端建立新的进程,则为 no;如果 UDP 在它的端口上处理所有的信息包,直到它被终止,则为 yes

user 指定程序的运行身份

server 服务程序的完整路径

server_args 参数,为了和 inetd 兼容,flags 设置为 NAMEINARGS,则参数使用服务器名

13.编写 xinetd 程序

使用标准输入输出,当使用 xinetd 的时候,通过两个方法传递 socket:如文件描述符 0 和 1.它们和文件描述符一样,分别代表标准输入和标准输出.因为标准输出 sys.stdout 默认是被缓冲的,所以为了实时性,需要使用到 sys.stdout.flush()函数.通过这种方法,服务器端程序既可以当成网络服务器程序使用,也可以像普通的脚本程序一样执行.

测试文件:

```
#!/usr/bin/env python
import sys
print 'Welcome.'
print 'Please enter a string:'
sys.stdout.flush()
line = sys.stdin.readline().strip()
print 'You enterd %d characters' % len(line)
```

配置文件

[root@localhost mypy]# vim /etc/xinetd.d/pyserver

```
service pyserver
{
        flags           =       NAMEINARGS
        type            =        UNLISTED
        port            =        21567
        socket_type     =       stream
        protocol        =       tcp
        wait            =        no
        user            =        root
        server          =        /root/PycharmProjects/mypy/example.py
        server_args     =       /root/PycharmProjects/mypy/example.py
}
[root@localhost mypy]# systemctl start xinetd
[root@localhost mypy]# systemctl enable xinetd
[root@localhost mypy]# netstat -tlnp | grep :21567
tcp6       0      0 :::21567                :::*                    LISTEN      1628/xinetd
[root@localhost mypy]# vim /root/PycharmProjects/mypy/example.py
#!/usr/bin/env python
import sys
print 'Welcome!'
print 'Please imput someting.'
sys.stdout.flush()
line = sys.stdin.readline()
print 'You entered %s chars.' % len(line)
[root@localhost mypy]# chmod +x example.py
[root@localhost mypy]# systemctl restart xinetd
[root@localhost mypy]# netstat -tlnp | grep :21567
tcp6       0      0 :::21567                :::*                    LISTEN      32121/xinetd
[root@localhost mypy]# ifconfig eno16777736 192.168.0.123
[root@localhost mypy]# telnet 192.168.0.123 21567
Trying 192.168.0.123...
Connected to 192.168.0.123.
Escape character is '^]'.
Welcome!
Please imput someting.
hello
You entered 7 chars.
Connection closed by foreign host.
```

14.使用标准输入输出搭建 TCP 服务器

```
[root@localhost mypy]# cat example.py
#!/usr/bin/env python
import sys
import time
while True:
    data = sys.stdin.readline()
```

```
    if not data.strip():
        break
    sys.stdout.write('[%s] %s' % (time.ctime(),data))
    sys.stdout.flush()
[root@localhost mypy]# chmod +x example.py
[root@localhost mypy]# telnet 192.168.0.123 21567
```

客户端

```
#!/usr/bin/env python
import socket
host = '192.168.0.123'
port = 21567
addr = (host,port)
c = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
c.connect(addr)
while True:
    try:
        data = raw_input('> ')
    except KeyboardInterrupt:
        print
        break
    if not data.strip():
        break
    c.send(data + '\n')
    print c.recv(2048)
c.close()
[root@localhost mypy]# python tcpcli.py
```

15.使用 socket 对象

通过调用 socket.fromfd()可以建立 socket 对象.建立 socket 对象需要使用 xinetd 传递给程序的文件描述符.fromfd()函数需要文件数量和一些标准的参数,这些参数与前一章内容相同.文件描述符可以通过 filen()函数得到.

```
#!/usr/bin/env python
import sys
import time
import socket
s = socket.fromfd(sys.stdin.fileno(),socket.AF_INET,socket.SOCK_STREAM)
s.sendall('Welcome!\n')
s.sendall('you are connected from %s.\n' % str(s.getpeername()))
s.sendall('The local time is:%s.\n' % time.ctime())
```

标准版,一次的

```
[root@localhost mypy]# cat example.py
#!/usr/bin/env python
import sys
import socket
import time
s = socket.fromfd(sys.stdin.fileno(),socket.AF_INET,socket.SOCK_STREAM)
```

```
s.sendall('Welcome!\n')
s.sendall('You are connected from %s.\n' % str(s.getpeername()))
s.sendall('Now it is\n: %s' % time.ctime())
[root@localhost mypy]# chmod +x example.py
[root@localhost mypy]# telnet 192.168.0.123 21567
Trying 192.168.0.123...
Connected to 192.168.0.123.
Escape character is '^]'.
Welcome!
You are connected from ('192.168.0.123', 41774).
Now it is
: Mon Mar 21 16:10:21 2016Connection closed by foreign host.

[root@localhost mypy]# vim /var/log/messages  查看日志

循环的
#!/usr/bin/env python
import time
import sys
import socket
s = socket.fromfd(sys.stdin.fileno(),socket.AF_INET,socket.SOCK_STREAM)
while True:
    data = s.recv(20148)
    if not data.strip():
        break
    s.send('[%s] %s' % (time.ctime(),data))
```

16.forking

forking 工作原理:fork(分岔)在 linux 系统中使用非常广泛.当某一命令执行时,父进程(当前进程)fork 出一个子进程.父进程将自身资源拷贝一份,命令在子进程中运行时,就具有和父进程完全一样的运行环境.

17.进程的生命周期

父进程 fork 出子进程并挂起,子进程运行完毕后,释放大部分资源并通知父进程,这个时候,子进程被称作僵尸进程.父进程获知子进程结束,子进程所有资源释放.

```
--->parent process-----------wait()------------------>parent process--->
        |                                                                |
    fork()                                                              |
        |                                                                |
    child process---exec()--->child process---exit()--->zombie process
```

18.僵尸进程

僵尸进程没有任何可执行代码,也不能被调度.如果系统中存在过多的僵尸进程,将因为没有可用的进程号而导致系统不能产生新的进程.对于系统管理员来说,可以试图杀死父进程或重启系统来消除僵尸进程.

19.forking 编程,forking 编程基本思路:

需要使用 os 模块.os.fork()函数实现 forking 功能.python 中,绝大多数的函数只返回一次,os.forking 将返回两次.对 fork()的调用,针对父进程返回子进程的 PID;对于子进程,返回 PID0.因为所有的父子进程拥有相同的资源,所以在编写进程

时要避免资源冲突.
语法示例:

```
pid = os.fork() #实现 forking
if pid: #在父进程中关闭子进程连接
    close_child_conn #接着处理其他的连接请求
    handle_more_conn
else: #子进程关闭父进程连接,响应当前的用户连接
    close_parent_conn
    process_this_conn
```

应用示例 1:

```
#!/usr/bin/env python
import os
print 'stat prog.'
os.fork()
print 'hello'
[root@localhost mypy]# python sim_fork.py
stat prog.
hello
hello
```

应用示例 2:

```
#!/usr/bin/env python
import os
print 'stat prog.'
pid = os.fork()
if pid:
    print 'hello from parent'
else:
    print 'hello from child'
print 'hello from both'
[root@localhost mypy]# python sim_fork.py
stat prog.
hello from parent
hello from both
hello from child
hello from both
```

20.forking 基础应用

模拟僵尸进程

```
#!/usr/bin/env python
import os
import time
pid = os.fork()
if pid:
    print 'parentstart at:', time.ctime()
    time.sleep(30)
    print 'parent done at:',time.ctime()
```

```
else:
    print 'child start at:',time.ctime()
    time.sleep(10)
    print 'child done at:',time.ctime()
```
[root@localhost mypy]# watch -n 1 ps a 每 1 秒执行一次 ps a 命令

21.使用轮询解决 zombie 问题

父进程通过 os.wait()来得到子进程是否终止的信息.在子进程和父进程调用 wait()之间的这段时间,子进程被称为 zombie(僵尸)进程.如果子进程还没有终止,父进程先退出了,那么子进程会持续工作,系统自动将子进程的父进程设置为 init 进程,init 将来负责清理僵尸进程.

python 可以使用 waitpid()来处理子进程:

waitpid()接受两个参数,第一个参数设置为-1,表示与 wait()函数相同,第二参数如果设置为 0 表示挂起父进程,直到子进程退出,设置为 1 表示不挂起父进程

waitpid()的返回值:如果子进程尚未结束则返回 0,否则返回子进程的 PID

示例应用 1:

```
#!/usr/bin/env python
import os
import time
def reap():
    result = os.waitpid(-1,os.WNOHANG) #WNOHANG 即值为 1
    print 'Reaped child process %d' % result[0]
pid = os.fork()
if pid:
    print 'In parent.sleeping 15s...'
    time.sleep(15)
    reap()
    time.sleep(5)
    print 'parent done'
else:
    print 'In child.Sleeping 5s...'
    time.sleep(5)
    print 'Child terminating.'
```

示例应用 2:

```
#!/usr/bin/env python
import os
import time
def reap():
    result = os.waitpid(-1,os.WNOHANG)
    print 'Reapled child process:',result
pid = os.fork()
if pid:
    print 'parentstart at:', time.ctime()
    time.sleep(20)
    reap()
    time.sleep(10)
    print 'parent done at:',time.ctime()
```

```
else:
    print 'child start at:',time.ctime()
    time.sleep(10)
    print 'child done at:',time.ctime()
```

示例应用 3:

```
#!/usr/bin/env python
import os
import time
def reap():
    result = os.waitpid(-1,0)
    print 'Reapled child process:',result
pid = os.fork()
if pid:
    print 'parentstart at:', time.ctime()
    reap()
    time.sleep(10)
    print 'parent done at:',time.ctime()
else:
    print 'child start at:',time.ctime()
    time.sleep(10)
    print 'child done at:',time.ctime()
```

22.forking 服务器

在网络服务器中,forking 被广泛使用.如果服务器需要同时响应多个客户端,那么 forking 是解决问题的最常用的方法之一.父进程负责接收客户端的连接请求.子进程负责处理客户端的请求.

23.利用 forking 创建 TCP 时间戳服务器

```
#!/usr/bin/env python
import socket
import time
import os
import sys
host = ''
port = 12345
addr = (host,port)
s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
s.bind(addr)
s.listen(1)
while True:
    cli_sock,cli_addr = s.accept()
    print 'Got connection from:',cli_addr
    pid = os.fork()
    if pid:
        cli_sock.close()
        continue
```

```
        else:
                s.close()
                while True:
                        data = cli_sock.recv(2048)
                        if not data.strip():
                                break
                        cli_sock.send('[%s] %s' % (time.ctime(),data))
                sys.exit()
s.close()
[root@localhost mypy]# watch -n 1 ps a
[root@localhost mypy]# python sim_fork.py
[root@localhost mypy]# telnet 192.168.0.123 12345
[root@localhost mypy]# telnet 127.0.0.1 12345
```

24.改进版,收割僵尸进程

```
#!/usr/bin/env python
import socket
import time
import os
import sys
def reap():
        while True:
                result = os.waitpid(-1,os.WNOHANG)
                if not result[0]:
                        break
                print 'Reaped child process:',result[0]
host = ''
port = 12345
addr = (host,port)
s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
s.bind(addr)
s.listen(1)
while True:
        try:
                reap()
        except OSError:
                pass
        cli_sock,cli_addr = s.accept()
        print 'Got connection from:',cli_addr
        pid = os.fork()
        if pid:
                cli_sock.close()
                continue
        else:
                s.close()
```

```
            while True:
                data = cli_sock.recv(2048)
                if not data.strip():
                    break
                cli_sock.send('[%s] %s' % (time.ctime(),data))
            sys.exit()
s.close()
```

25.shell 检测某个网段存活的主机 ip

```
[root@localhost mypy]# for i in {1..254}
> do
> echo -e "172.40.3.$i\tserver$i.tedu.cn" >> /etc/hosts
> done
```

26.探测网络里主机存活脚本

shell 方法

```
[root@localhost mypy]# cat ping.sh
#!/bin/bash
for ip in 172.40.5.{1..254}
do
    ping -c1 $ip &> /dev/null && echo "$ip:up" || echo "$ip:down"
done
```

27.python 类 shell 方法

```
#!/usr/bin/env python
import os
for i in range(1,255):
    ip = '172.40.5.' + str(i)
    result = os.system('ping -c1 %s &> /dev/null' % ip)
    if result == 0:
        print '%s:up' % ip
    else:
        print '%s:down' % ip
```

28.python 的 fork 方法

```
#!/usr/bin/env python
import os
for i in range(1,255):
    ip = '172.40.5.' + str(i)
    pid = os.fork()
    if pid:
        continue
    else:
        result = os.system('ping -c1 %s &> /dev/null' % ip)
        if result == 0:
            print '%s:up' % ip
        else:
```

```
            print '%s:down' % ip
        break
```

29.多线程

多线程工作原理(多线程的动机):

在多线程(MT)编程出现之前,电脑程序的运行有一个执行序列组成,执行序列按顺序在主机的中央处理器(CPU)中运行.无论是任务本身要求顺序执行还是整个程序是由多个子任务组成,程序都是按这种方式执行的.即使子任务相互独立,互相无关(即,一个子任务的结果不影响其它子任务的结果)时也是这样.如果并行运行这些相互独立的子任务可以大幅度地提升整个任务的效率.

30.多线程任务的工作特点

它们本质上就是异步,需要有多个并发事务.各个事务的运行顺序可以是不确定的,随机的,不可预测的.这样的编程任务可以被分成多个执行流,每个流都有一个要完成的目标.根据应用的不同,这些子任务可能都要计算出一个中间结果,用于合并得到最后的结果.

31.什么是进程

计算机程序只不过是磁盘中可以执行的,二进制(或其他类型)的数据.进程(有时候被称为重量级进程)是程序的一次执行.每个进程都有自己的地址空间,内存以及其它记录其运行轨迹的辅助数据.操作系统管理在其上运行的所有进程,并为这些进程公平的分配时间.

32.什么是线程

线程(有时被称为轻量级进程)跟进程有些相似,不同的是,所有的线程运行在同一进程中,共享相同的运行环境.线程有开始,顺序执行和结束三部分.进程运行可能被抢占(中断),或暂时的被挂起(也叫睡眠),让其它的线程运行,这叫做让步.一个进程中的各个线程之间共享一片数据空间,所以线程之间可以比进程之间更方便的共享数据以及相互通讯.线程一般都是并发执行的,正是由于这种并行和数据共享的机制使得多个任务的合作变为可能.需要注意的是,在单个 cpu 的系统中,真正的并发是不可能的,每个线程会被安排成每次只运行一小会,然后就把 cpu 让出来,让其他的线程去运行.

33.多线程编程

多线程相关模块

thread 和 threading 模块允许程序员创建和管理线程

thread 模块提供了基本的线程和锁的支持,而 threading 提供了更高级别,功能更强的线程管理功能

推荐使用更高级别的 threadingmok

只建议那些有经验的专家在想访问线程的底层结构的时候,才使用 thread 模块

http://pypi.python.org #py 官方收录的模块

34.传递函数给 Thread 类

多线程编程有多种方法,传递函数给 threading 模块的 Thread 类是介绍的第一种方法.Thread 对象使用 start()方法喀什线程的执行,使用 join()方法挂起程序,直到线程结束.

案例跳过,不使用下面这个

```
#!/usr/bin/env python
import threading
import time
nums = [4,2]
def loop(nloop,nsec): #定义函数,打印运行的起止时间
    print 'start loop %d,at %s' % (nloop,time.ctime())
    time.sleep(nsec)
```

```
    print 'loop %d done at %s' % (nloop,time.ctime())
def main():
    print 'starting at:%s' % time.ctime()
    threads = []
    for i in range(2): #创建两个线程,放入列表
        t = threading.Thread(target = loop,args = (0,nums[i]))
        threads.append(t)
    for i in range(2):
        threads[i].start() #同时运行两个线程
    for i in range(2):
        threads[i].join() #主程序挂起,直到所有线程结束
    print 'all Done at %s' % time.ctime()
if __name__ == '__main__':
    main()
```

35.多线程的 ping 程序

```
#!/usr/bin/env python
import threading
import os
def ping(ip):
    result = os.system('ping -c1 %s &> /dev/null' % ip)
    if result:
        print '%s:down' % ip
    else:
        print '%s:up' % ip
if __name__ == '__main__':
    for i in range(1,255):
        ip = '172.40.5.' + str(i)
        t = threading.Thread(target=ping,args=(ip,))
        t.start()
```

36.传递可调用类给 Thread 类

传递可调用类给 Thread 类是介绍的第二种方法.相对于一个或几个函数来说,由于类对象里可以使用类的强大的功能,可以保存更多的信息,这种方法更为灵活.

应用示例:

```
#!/usr/bin/env python
import threading
import time
nums = [4,2]
class ThreadFunc(object): #定义可调用的类
    def __init__(self,func,args,name=''):
        self.name = name
        self.func = func
        self.args = args
    def __call__(self):
        apply(self.func,self.args)
```

```
def loop(nloop,nsec): #定义函数,打印运行的起止时间
    print 'start loop %d,at %s' % (nloop,time.ctime())
    time.sleep(nsec)
    print 'loop %d done at %ss' % (nloop,time.ctime())
def main():
    print 'starting at:%s' % time.ctime()
    threads = []
    for i in range(2):
        t = threading.Thread(target = ThreadFunc(loop,(i,nums[i]),loop.__name__))
        threads.append(t) #创建两个线程,放入列表
    for i in range(2):
        threads[i].start()
    for i in range(2):
        threads[i].join()
    print 'all Done at %s' % time.ctime()
if __name__ == '__main__':
    main()
```

37.多线程的 ping 程序

```
#!/usr/bin/env python
import threading
import os
class MtPing(object):
    def __init__(self,func,args):
        self.func = func
        self.args = args
    def __call__(self):
        apply(self.func,self.args)
def ping(ip):
    result = os.system('ping -c1 %s &> /dev/null' % ip)
    if result:
        print '%s:down' % ip
    else:
        print '%s:up' % ip
if __name__ == '__main__':
    for i in range(1,255):
        ip = '172.40.5.' + str(i)
        t = threading.Thread(target=MtPing(ping,(ip,)))
        t.start()
```

38.含有线程的服务器

多数的线程服务器有同样的结构.主线程是负责侦听请求的线程.主线程收到一个请求的时候,新的工作线程会被建立起来,处理客户端请求.客户端断开时,工作线程将终止.线程划分为用户线程和后台(daemon)进程,setDaemon 将线程设置为后台进程.

```
#!/usr/bin/env python
import time
```

```
import threading
import socket
def handle_child(cli_sock):
    while True:
        data = cli_sock.recv(2048)
        if not data.strip():
            break
        cli_sock.send('[%s] %s' % (time.ctime(),data))
    cli_sock.close() #没这句,回车无法退出
if __name__ == '__main__':
    host = ''
    port = 12345
    addr = (host,port)
    s = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
    s.bind(addr)
    s.listen(1)
    while True:
        cli_sock,cli_addr = s.accept()
        print 'Got connection from:',cli_addr
        t = threading.Thread(target=handle_child,args=(cli_sock,))
        t.setDaemon(1)
        t.start()
```