

PYTHON简明教程

@author 刘振伟

@QQ 570962906

PYTHON创始人: Guido van Rossum,在Python界被誉为仁慈的独裁者。

借鉴了unix shell ,c的特点

1989圣诞期间发布, python已经有20年的历史了, 比java早很多, java第一个版本才是1994年发布的。

2000年10.16 python2.0发布, 此时python才真正成为一个功能完善, 非常好用的语言

2008年12月3日, python3发布, 相比与python2而言, 改变比较大, 不兼容python2.目前是两个版本共存的

python的定位:

- 解释型的通用语言(操作系统内核等没有解释器存在, 是不能在内核或硬件上使用python的)
- 优雅 明确 简单, 在Python中做某件事有且只有一个最优解
- 使用范围: web(国内的豆瓣, 知乎等等), 自动化脚本, 数据分析(spark上原生支持Python)等

python的实现与版本:

- 通常所说的python是由C语开发, 是官方的版本
- jython是由java写的, 运行在jvm上, 可以与现有的JAVA库无缝的兼容
- IronPython 是运行在.net 平台上的, 兼容.net库
- PyPy 是用python写成的python(在科学计算上较广泛使用), 在python中有一个很大的"缺陷", GIL全局库解释器锁, PyPy就是为了解决这个问题而存在的。
- 大版本之间不向前兼容(python3与python2)
- 本次课以python2.7来讲, 这也是目前使用最为广泛的版本

安装PYENV

大多数linux上已经安装了python环境, 但不同的发行版linux安装的python版本会有不同, pyenv管理多版本的python.

我们工作上有时候需要使用到多版本的python,

```
yum -y install gcc gcc-c++ make git patch openssl-devel zlib-devel readline-devel sqlite-devel bzip2-devel
```

安装pyenv:

作者已经给提供了一个安装脚本, 下载下来直接运行即可:

```
https://raw.githubusercontent.com/yyuu/pyenv-installer/master/bin/pyenv-installer
```

配置环境变量:

```
[root@yeahmonitor ~]# cat /etc/profile.d/pyenv.sh
export PATH="$HOME/.pyenv/bin:$PATH"
eval "$(pyenv init -)"
eval "$(pyenv virtualenv-init -)"
[root@yeahmonitor ~]# . /etc/profile
```

使用pyenv 安装python2.7.5

```
[root@yeahmonitor ~]# pyenv install 2.7.5
#rhel7和centos 7中系统自带的python就是2.7.5的版本
[root@yeahmonitor ~]# pyenv install pypy-1.9
#安装pypy-1.9
```

卸载 `pyenv uninstall xxx`

告诉pyenv当前目录使用哪个版本的python

```
[root@yeahmonitor python]# pyenv local 2.7.5
[root@yeahmonitor python]# pyenv rehash#重建环境变量, 告诉pyenv当前使用该版本
[root@yeahmonitor python]# pyenv version
2.7.5 (set by /root/python/.python-version)
#其他目录下还是使用的是系统自带的版本, 这里是7的linux 所以全是2.7.5
```

管理全局的python: `pyenv global 2.7.5`

打开一个python shell:

```
[root@yeahmonitor python]# python
Python 2.7.5 (default, Jun 17 2014, 18:11:42)
[GCC 4.8.2 20140120 (Red Hat 4.8.2-16)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

安装easy_install,

```
wget https://bootstrap.pypa.io/ez_setup.py -O - | python
```

ipython是对python shell的增强(自动补全, 更清晰的查看文档等):

```
[root@yeahmonitor python]# easy_install ipython
```

打开ipython shell:

```
ipython
```

变量命名:

- (下划线或字母) + (任意数目的字母、数字或下划线)
- 变量名必须以下划线或字母开头, 而后面接任意数目的字母、数字或下划线。
- 区分大小写: SPAM和spam不同
- 禁止使用保留字

python基础

最初步骤

- 使用交互式python
- 挑选一个合适的编辑器或者IDE
- 使用源文件
- 可执行的python
- unicode支持

简单的示例:

```
[root@yeahmonitor python]# cat hello.py
print "hello world"
[root@yeahmonitor python]# python hello.py
hello world
```

从第一行往下依次执行代码

示例二: 以可执行文件运行

```
[root@yeahmonitor python]# cat hello.py
#!/usr/bin/env python
print "hello world"
[root@yeahmonitor python]# chmod +x hello.py
[root@yeahmonitor python]# ./hello.py
hello world
```

输出中文:

```
[root@yeahmonitor python]# cat hello.py
#!/usr/bin/env python

print "hello world"
print "你好, 世界!"
[root@yeahmonitor python]# ./hello.py
File "./hello.py", line 4
SyntaxError: Non-ASCII character '\xe4' in file ./hello.py on line 4, but no encoding declared; see http://www.python.org/peps/pep-0263.

#显式的指定编码:

[root@yeahmonitor python]# cat hello.py
#!/usr/bin/env python
# coding=utf-8 #显式指定编码为utf-8
print "hello world"
print "你好, 世界!"
[root@yeahmonitor python]# ./hello.py
hello world
你好, 世界!
```

关于脚本中的第一行内容:

- `#!/usr/bin/python` 这种写法表示直接引用系统的默认的Python版本, 这样的话python程序移植到其他机器上可能运行的时候有问题, 因为别人系统默认的Python版本与你预期的并不一致。
- `#!/usr/bin/env python` 这种写法表示, 引用环境变量里面自定义的Python版本, 具有较强的可移植性, 推荐这种写法。

测试不写第一行内容是否可行。。。在什么情况下正常运行, 在什么情况下会报错, 结合shell来讲。

基本概念

字面常量(看到的是什么样的, 就是什么样),如"hello world"

单独出现的数字, 字符串等
`1, "abc", ['a', 1]`
字面常量是解释器里面的一块内存
单独的字面常量是无意义的, 当一块内存没有变量在引用的时候会被自动释放。

变量

变量是一个指向一段内存的符号
python里所有的变量都是引用,
变量命名规范(由数字, 字母, 下划线组成, 不能以数字开头, 关键字不能作为变量名)

数据类型

python是一种强类型的动态语言, 每一个数据都有一个类型, 不同类型之间的数据不能做运算, 如数字和字符不能相加.所谓的动态语言是指, 数据类型可以在运算时改变。

```
>>> v1 = 1
>>> type(v1)
<type 'int'>
>>> v1 = "string"
>>> type(v1)
<type 'str'>
#在C中是绝对不能这么做的

#不同类型的数据不能做运算
>>> v1 = "string"
>>> type(v1)
<type 'str'>
>>> v2 = 1
>>> v1+v2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

数据类型, 不同的数据类型的二进制是不一样的, 为了能让解释器知道这段内存是什么, 所以要定义数据类型, 也就是规范了数据存放的二进制格式。

在python中分为数字类型和字符串类型, 数字又分为: 整型,长整型和浮点型

```
>>> a = 1.1
>>> type(a)
<type 'float'>
#字符串连接
>>> s1 = 'hello'
>>> s2 = ' world'
>>> s = s1 + s2
>>> s
'hello world'
```

类型转化, `>>> i = 1 >>> float(i)` 1.0 #float要比int精度要高, 从低精度向高精度转换是没有数据损失的, 从高精度转向低精度会有损失。

对象, 在python中万物皆对象, 所有的都是对象

运算符与表达式

算术运算符 `+` `*` `**` `//` `%` 位操作运算符 `<<` `>>` `&` `|` `^` 比较运算符 `<` `<=` `>` `>=` `!=` 逻辑运算符 `and` `or` `not` 赋值运算符 `=` 其他运算符

数学运算符,

```
>>> 10 / 3 #两个整型相除的时候, 等到的还是整型, 会有精度损失
3
#将其中一个转换为float, 会等到一个float型
>>> 10 /float(3)
3.3333333333333335

#幂运算
>>> 2**10
```

```
1024

#除法取整
>>> 10 // 3.0
3.0 #只取整数部分

#取模运
>>> 10 % 3
1
```

位运算符:

```
>>> 2 << 3 #2左移3位, (10 --->10 000)
16
#位运算速度比较快

#按位与 &
>>> 3 & 2
2

#按位或 |
>>> 3 | 2
3

#异或, 两个位置相同取1, 不同则取0
>>> 3 ^ 2
1
```

比较运算符, 与其他语言都相同

逻辑运算符, 与其他语言类似, 只是写法不同

赋值运算符, 与其他语言相同 `a = 1`, 想 `a` 符号指向该数据存在的内存

表达式

由运算符, 连接起来的变量或者常量, 构成表达式, 如 `a = 1` 和 `c = a1 + b2`

优先级

- 单目运算符高于双目运算符, not例外; 单目运算符, 只有一个数的运算符, 如正负号, 按位取反, not, python里没有三目运算符
- 算术运算符高于位运算符:
- 位运算符高于比较运算符
- 比较运算符高于逻辑运算符
- 赋值运算符优先级最低

```
>>> 1 + 2 << 2 #先加再移位
12
>>> 3 << 2
12

#位运算符高于比较运算符

>>> 2 << 2 < 6
False
```

() 可以提升运算符的优先级, 某些情况为了程序的可读性也要有小括号

程序结构

顺序结构

分支结构 (if, else, elif)

if 语句

```
if condition:
    expression
```

示例:

```
[root@yeahmonitor python]# cat if.py
#!/usr/bin/env python

if 3 < 5:
    print "3 less than 5" #语句块里面可以是多个语句
```

```
if 3 > 4:
    print "3 greate than 4"
[root@yeahmonitor python]# ./if.py
3 less than 5
```

else 子句

```
if condition:
    expression
else:
    expression
```

if语句里面还可以嵌套if.python是允许语句的嵌套的。else子句总是if语句的最后一个分支。不能出现在elif子句前面。

elif 字句

```
if condition:
    expression
elif condition:
    expression
#可以有多个elif 子句,但是一旦其中一个分支没执行了,就不会往下再去匹配执行了。
```

python中没有switch语句。只能用elif模拟switch.

循环结构

- while 语句
- for 语句
- break语句
- continue语句
- else子句

while循环

```
while condition:
    expression
```

示例:

```
[root@yeahmonitor python]# cat while.py
#!/usr/bin/env python

a = 3

while a <= 10:
    print a
    a += 1
[root@yeahmonitor python]# chmod +x while.py
[root@yeahmonitor python]# ./while.py
3
4
5
6
7
8
9
10
```

for 语句

```
for item in iterator:
    expression
```

python中for语句后边是要跟一个迭代器(列表,元组等)的。

range可以得到一个列表。range(1,10)

```
[root@yeahmonitor python]# cat for.py
#!/usr/bin/env python

for x in range(10):
    print x
[root@yeahmonitor python]# chmod +x for.py
[root@yeahmonitor python]# ./for.py
0
1
```

```
2
3
4
5
6
7
8
9
```

break语句

- 只能出现在循环块中
- 用于跳出当前循环

示例：

```
[root@yeahmonitor python]# cat for.py
#!/usr/bin/env python

for x in range(10):
    print x
    if x == 5:
        break
[root@yeahmonitor python]# ./for.py
0
1
2
3
4
5
```

continue语句

- 只出现在循环中
- 跳出当前循环循环操作，接着执行下一次循环

Python 内置容器

列表 list

定义和初始化list

```
#定义个空列表
In [1]: li = []

In [2]: type(li)
Out[2]: list

#初始化list,list中的元素没有类型要求，可以是任何类型
In [3]: li = [1,2,'a',['a',4]]

In [4]: li
Out[4]: [1, 2, 'a', ['a', 4]]
```

列表的下标

python中列表的下标是从0开始的。

```
In [4]: li
Out[4]: [1, 2, 'a', ['a', 4]]

In [5]: li[2]
Out[5]: 'a'

In [6]: li[4]
-----
IndexError                                Traceback (most recent call last)
<ipython-input-6-5889debca501> in <module>()
----> 1 li[4]

IndexError: list index out of range

#得到list长度
In [7]: len(li)
Out[7]: 4
```

in关键字

在for循环中使用表示遍历list中的所有元素：

```
In [8]: for x in li:
...:     print x
...:
1
2
a
['a', 4]
```

in也可以作为一个二元操作符使用,查找某个元素是否在list中存在：

```
In [9]: 'a' in li
Out[9]: True
```

del删除list中某个元素：

```
In [10]: li
Out[10]: [1, 2, 'a', ['a', 4]]

In [11]: del li[0]

In [12]: li
Out[12]: [2, 'a', ['a', 4]]
```

在python中一切皆为对象，显然，list也是对象，针对于列表对象来说，常用的操作：

```
In [13]: dir(li)
Out[13]:
['__add__',
 '__class__',
 '...',
 '__str__',
 '__subclasshook__',
 'append',
 'count',
 'extend',
 'index',
 'insert',
 'pop',
 'remove',
 'reverse',
 'sort']
```

查看list对象append方法的帮助信息：

```
In [14]: help(li.append)

In [15]: li
Out[15]: [2, 'a', ['a', 4]]

In [16]: li.append(666)

In [17]: li
Out[17]: [2, 'a', ['a', 4], 666]
```

extend方法，接受一个迭代器，将迭代器的所有元素追加到list中：

```
In [17]: li
Out[17]: [2, 'a', ['a', 4], 666]

In [18]: li.extend(['abc', 'cba'])

In [19]: li
Out[19]: [2, 'a', ['a', 4], 666, 'abc', 'cba']
```

insert方法，li.insert(2,'x'),li.insert(-1,'mm'),在某个索引下标之前插入某个数据。

remove方法，li.remove('a'),从list中移除某个元素。

pop方法，接受一个可选参数index,remove下标所指向的元素，并将元素返回，li.pop(),li.pop(2)：

```
In [19]: li
Out[19]: [2, 'a', ['a', 4], 666, 'abc', 'cba']
```

```
In [20]: li.pop()
Out[20]: 'cba'

In [21]: li
Out[21]: [2, 'a', ['a', 4], 666, 'abc']

In [22]: li.pop(2)
Out[22]: ['a', 4]

In [23]: li
Out[23]: [2, 'a', 666, 'abc']
```

`count`方法, 返回某个值在list出现的次数,`li.count('a')`

`index`方法, 返回第一个匹配value的下标:

```
In [28]: li
Out[28]: [2, 'a', 666]

In [29]: li.index('a')
Out[29]: 1

In [30]: li.index('b')
-----
ValueError                                Traceback (most recent call last)
<ipython-input-30-4639cb7d3bae> in <module>()
----> 1 li.index('b')

ValueError: 'b' is not in list

#还可以指定起始和结束查找的范围参数 li.index('a',2,6)
In [33]: li
Out[33]: [2, 'a', 666, 'a', 'a', 'b']

In [34]: li.index('a',2)
Out[34]: 3
```

`sort`方法, 直接修改list内容, 不返回值.`li.sort()`,还可以接受三个可选参数。`L.sort(cmp=None, key=None, reverse=False)`

`reverse`方法。`li.reverse()`

list的切片

`li[i]`,`li[a:b]`,`li[a:e:b]`--第三个参数为步长,index每次加几, 默认为1

```
In [40]: li
Out[40]: ['b', 'a', 'a', 'a', 666, 2]

In [41]: li[2:4]
Out[41]: ['a', 'a']
####
In [43]: li[:2]
Out[43]: ['b', 'a']
###
In [44]: li[4:]
Out[44]: [666, 2]
##对list做一次深copy
In [45]: li[:]
Out[45]: ['b', 'a', 'a', 'a', 666, 2]
####
In [46]: li[2:4:2]
Out[46]: ['a']

##用切片实现list翻转
In [47]: li
Out[47]: ['b', 'a', 'a', 'a', 666, 2]

In [48]: li[::-1]
Out[48]: [2, 666, 'a', 'a', 'a', 'b']
#得到下标为偶数的list
In [50]: li
Out[50]: ['b', 'a', 'a', 'a', 666, 2]

In [51]: li[::2]
Out[51]: ['b', 'a', 666]
#得到下标为奇数的值
In [52]: li[1::2]
Out[52]: ['a', 'a', 2]
```


列表的切片操作是一个复制操作，并不对原始列表进行修改。

元组tuple

定义和初始化元祖

```
In [1]: t = ()

In [2]: type(t)
Out[2]: tuple

In [3]: t = (1,2,3)

In [4]: t
Out[4]: (1, 2, 3)
```

元祖也支持下标和切片操作。

元祖是不可变对象，不能对元祖的内容做修改

```
In [4]: t
Out[4]: (1, 2, 3)

In [5]: t[1]
Out[5]: 2

In [6]: t[1] = 100
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-6-4f066cd5e53f> in <module>()
----> 1 t[1] = 100

TypeError: 'tuple' object does not support item assignment
```

元祖的不可变是相对的，因为元祖里面的内容可以是各种类型，如，元祖的元素值为列表：

```
In [7]: t1 = ([2,3,4],[19,23,45])

In [8]: t1
Out[8]: ([2, 3, 4], [19, 23, 45])

In [9]: t1[0][2]
Out[9]: 4

In [10]: t1[0][2] = 100 #list的值是可变的，可以被修改的

In [11]: t1
Out[11]: ([2, 3, 100], [19, 23, 45])
```

count,index操作

```
In [12]: t
Out[12]: (1, 2, 3)

In [13]: t.count(2)
Out[13]: 1

In [14]: t.index(3)
Out[14]: 2
```

元组支持切片操作与列表的切片操作一样。

集合 set

定义和初始化集合

列表和元组都是有序的，但是集合是无序的。

```
In [17]: s = {1,2,3}

In [18]: type(s)
Out[18]: set
```

集合中的元素不重复

```
In [19]: s = {1,1,1,12,2,2,3,4}
```

```
In [20]: s
Out[20]: {1, 2, 3, 4, 12}
```

python根据集合中的每个元素的hash值来判断是否重复，所以集合中的每个元素必须是可hash的对象。在python中如果一个对象有一个__hash__的方法，表示该对象可hash。

```
In [21]: 1.__hash__ #整数1的对象有该方法，但是__hash__方法不能直接调用。
1.__hash__
```

hash() 函数直接返回某个对象的hash值。如hash(1)

集合不支持切片操作。

集合的操作：

add操作

```
In [21]: s
Out[21]: {1, 2, 3, 4, 12}

In [22]: s.add(100)

In [23]: s
Out[23]: {1, 2, 3, 4, 12, 100}
```

update操作,迭代器作为参数，将迭代器中的所有元素追加到集合中

```
In [23]: s
Out[23]: {1, 2, 3, 4, 12, 100}

In [24]: s.update([101,102,103])

In [25]: s
Out[25]: {1, 2, 3, 4, 12, 100, 101, 102, 103}
```

remove 删除某个元素，若该元素不存在则报错

discard删除某个元素，若该元素不存在则不做任何操作。

pop()随机删除某个元素，并返回该元素

clear清空集合。s.clear()

集合的运算

difference 两个集合的差集，不修改原来的两个集合

```
In [26]: s1 = {1,2,3,4,5,'a'}

In [27]: s2 = {4,5,7,8,'b','c'}

In [28]: s1.diff
s1.difference          s1.difference_update

In [28]: s1.difference(s2)
Out[28]: {1, 2, 3, 'a'}

#减号可以直接求两个集合的差集
In [29]: s1 - s2
Out[29]: {1, 2, 3, 'a'}
```

difference_update两个集合的差集，但修改原来的集合,不返回值

```
In [30]: s1
Out[30]: {1, 2, 3, 4, 5, 'a'}

In [31]: s2
Out[31]: {4, 5, 7, 8, 'b', 'c'}

In [32]: s1.difference_update(s2)

In [33]: s1
Out[33]: {1, 2, 3, 'a'}

In [34]: s2
Out[34]: {4, 5, 7, 8, 'b', 'c'}
```

`intersection` 两个集合的交集，返回值，不修改原来的集合

`intersection_update` 两个集合的交集，无返回值，修改原来的集合

`&, s1 & s2` 直接求两个集合的交集

`union` 两个集合的并集

`|, s1 | s2` 也是求两个集合的并集

`isdisjoint` 查看两个集合是否有交集，返回bool, `s1.isdisjoint(s2)`

`issubset` 查看是否是子集

列表，元组，集合之间的转换

`list` 函数

```
In [35]: list()
Out[35]: []

In [36]: list('hello')
Out[36]: ['h', 'e', 'l', 'l', 'o']
```

`tuple` 函数

`set` 函数

```
In [37]: t = (1,2,3)

In [38]: type(t)
Out[38]: tuple

In [39]: list(t)
Out[39]: [1, 2, 3]
```

将list转换为set的时候，若列表中存在重复元素，则移除重复的元素。

迭代器iterator

`iter` 函数,构造集合。

`next` 方法依次返回迭代器的值

```
In [42]: li
Out[42]: [1, 2, 3, 4, 5]

In [43]: it = iter(li)

In [44]: it.next()
Out[44]: 1

In [45]: it.next()
Out[45]: 2
```

当所有的元素都返回完的时候，会抛出一个`StopIterator`的异常。

list,tuple,set 都属于集合。

`for` 语句，当将一个列表传递给 `for` 语句的时候，实际上 `for` 语句会将列表转换为迭代器，然后隐示的地执行该迭代器。

字典dict

定义与初始化 In [46]: d = {}

```
In [47]: type(d)
Out[47]: dict
In [48]: d = {'a':1, 'v': 'k'}

In [49]: d
Out[49]: {'a': 1, 'v': 'k'}
```

在字典中key是不允许重复的，所以字典中的key必须是可hash的对象。

字典的操作

`keys` 将字典的所有key作为一个列表返回

```
In [50]: d
Out[50]: {'a': 1, 'v': 'k'}

In [51]: d.keys()
Out[51]: ['a', 'v']
```

`iterkeys` 将字典的所有key作为一个迭代器返回

```
In [52]: d.iterkeys()
Out[52]: <dictionary-keyiterator at 0x12ff7e0>

In [53]: it = d.iterkeys()

In [54]: it.next()
Out[54]: 'a'
```

`values` 返回一个列表，该列表是所有元素的值

```
In [55]: d
Out[55]: {'a': 1, 'v': 'k'}

In [56]: d.values()
Out[56]: [1, 'k']
```

`items` 返回一个列表，列表中的每个元素是一个元组，元组中的两个值分别是key和value

```
In [57]: d
Out[57]: {'a': 1, 'v': 'k'}

In [58]: d.items()
Out[58]: [('a', 1), ('v', 'k')]
```

遍历一个字典：

```
In [59]: for v in d.values():
....:     print v
....:
1
k

In [60]: for k,v in d.items():
....:     print "%s => %s" % (k,v)
....:
a => 1
v => k
```

`get` 按照key取值，若存在则返回，否则返回None

```
In [61]: d.get('a')
Out[61]: 1
```

还可以给`get`传递第二个参数，表示若key不存在，则返回某个值。

```
In [65]: d.get('xx',100)
Out[65]: 100
```

`has_key` 判断某个key是否存在

```
In [63]: d.has_key('a')
Out[63]: True

In [64]: d.has_key('aa')
Out[64]: False
```

给字典增加一个键值对：

```
In [66]: d = {}

In [67]: d['c'] = 1

In [68]: d
Out[68]: {'c': 1}

In [69]: d['xx'] = 100

In [70]: d
```

```
Out[70]: {'c': 1, 'xx': 100}

In [71]: d['xx'] = 200

In [72]: d
Out[72]: {'c': 1, 'xx': 200}
```

update操作

```
In [73]: d
Out[73]: {'c': 1, 'xx': 200}

In [74]: d.update({'a':1,'b':2})

In [75]: d
Out[75]: {'a': 1, 'b': 2, 'c': 1, 'xx': 200}
```

python字典为引用传值，如：

```
In [76]: d
Out[76]: {'a': 1, 'b': 2, 'c': 1, 'xx': 200}

In [77]: d1 = d

In [78]: d1['a'] = 200

In [79]: d1
Out[79]: {'a': 200, 'b': 2, 'c': 1, 'xx': 200}

In [80]: d
Out[80]: {'a': 200, 'b': 2, 'c': 1, 'xx': 200}
```

copy操作：重新复制一个字典

```
In [81]: d
Out[81]: {'a': 200, 'b': 2, 'c': 1, 'xx': 200}

In [82]: d2 = d.copy()

In [83]: d2
Out[83]: {'a': 200, 'b': 2, 'c': 1, 'xx': 200}

In [84]: d2['a'] = 300

In [85]: d2
Out[85]: {'a': 300, 'b': 2, 'c': 1, 'xx': 200}

In [86]: d
Out[86]: {'a': 200, 'b': 2, 'c': 1, 'xx': 200}

In [87]: id(d)
Out[87]: 19662176

In [88]: id(d2)
Out[88]: 19501072

In [89]: id(d1)
Out[89]: 19662176
```

列表解析

[expression for item in iterator] 一个最基本的列表解析,返回一个列表

在expression可以使用item变量

返回一个迭代器

(expression for item in terator)

```
In [90]: li = [1,2,3]

In [91]: l = (x+1 for x in li)

In [92]: l
Out[92]: <generator object <genexpr> at 0x12fa730>

In [93]: l.next()
Out[93]: 2
```

```
In [94]: l.next()
Out[94]: 3
```

迭代器是惰性求值，只有用到了才会计算该值，否则不会。列表是先求出所有值的。所以当数据大的时候迭代器有较好的性能。

带条件的列表解析

[expression for item in iterator if condition]，当满足条件的时候才会append到列表中，并返回。当变为小括号的时候，就会返回一个迭代器。

```
In [95]: li
Out[95]: [1, 2, 3]

In [96]: [x for x in li if x % 2 == 0]
Out[96]: [2]
```

带多个条件 [expression for item in iterator if conditionX if conditionY]

带多个迭代器（笛卡尔积与列表解析）：[expr for x in IterX for y in IterY],类似于两个嵌套的for循环操作

```
In [99]: [(x,y) for x in [1,2,3] for y in [1,3]]
Out[99]: [(1, 1), (1, 3), (2, 1), (2, 3), (3, 1), (3, 3)]
```

列表解析的性能远高于循环语句。

作业：猜数字游戏：

1. 随机产生要猜的数字(1-100)
2. 输入，用于接收用户输入的数字
3. 循环，如果没有猜对则循环接收输入，并打出提示信息
4. 猜到数字或猜测次数达到一定次数后（6次）打印失败并退出

```
[root@localhost ~]# cat g_num.py
#!/usr/bin/env python
#coding=utf-8

import random

secret = random.randint(1,100)
guess,tries = 0,0

print u"你好，请你给出一个1-99的数字，试下你的运气如何？你只有六次机会哟~！！"

while guess != secret and tries < 6:
    print u"请给出你猜的数字试多少："
    guess_str = raw_input()

    g = int(guess_str)

    if g == secret:
        print u"wawawa.....好吧，你猜中了，你说怎么办吧？"
        break
    elif g < secret:
        print str(g),u"Are you a pig?太小了。"
    elif g > secret:
        print str(g),u"Are you a pig?太大了。"

    tries += 1
else:
    print u"都猜了6次了，还没猜到。你能chua!~~~"
    print u"哥来告诉你，这个数字是:",str(secret)
```

这里需要注意的是，假如说，用户输入的是一个字符串怎么办？？？

函数

简单函数的定义

```
def func_name():
    """ comment """
    expression

func_name() #调用函数
```

示例：

```
[root@yeahmonitor ~]# cat func.py
```

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

def func():
    print "Hi,Dog"

func()
```

局部变量和全局变量

- 作用域：变量生效的范围
- 局部变量：在函数内定义的变量，局部变量作用域为函数体
- 全局变量：定义在函数之外，在函数内使用global关键字标记，全局变量的作用域为整个模块
- 全局变量应尽量少用，但是有些配置性的信息可以直接使用全局变量定义较为方便调用
- 变量覆盖

示例，局部变量：

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

arg = 10 #这里是一个全局变量
def func():
    arg = 2 #这里是一个局部变量，与外边的arg是两个不同的变量
    print arg

func()
print arg
```

示例，使用全局变量：

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

arg = 10
def func():
    global arg #这里使用global定义arg是一个全局变量，下边的赋值操作会覆盖函数体外的赋值操作。
    arg = 6
    print arg

func()
print arg
```

在函数体内可以直接调用全局变量：

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

arg = 10
def func():
    print arg

func()
print arg
#在函数体内没有对arg变量有任何声明和赋值的操作，所以这里的arg调用的是全局变量
```

带参数的函数

```
def func_name(args):
    expression

func_name(values)
#参数可以是一个或者多个
```

示例：

```
def funcArg(i):
    print i #i作为参数传递进来，是一个局部变量

funcArg('hello')
```

多个参数用逗号隔开：

```
def funcArg(i,m,k):
    print i,m,k

funcArg('hello',1,2)
```

位置参数和关键字参数

```
def func_name(arg1,arg2):  
    expression
```

通常的调用方式为func_name(v1,v2),但是如果参数非常多,传值的时候一定要记好每个参数的顺序。这种是位置参数。也可以以关键字的行为传递参数func_name(arg1=v1,arg2=v2),顺序就不那么重要了,这就是关键字参数。

```
def funcArg(i,m,k):  
    print i,m,k  
  
funcArg(m = 'hello',k=1,i=2)
```

关键字参数和位置参数也可以混合使用:

```
func_name(v1,arg2=v2)
```

```
def funcArg(i,m,k):  
    print i,m,k  
  
funcArg(m = 'hello',k=1,i=2)  
funcArg(20,k=100,m='hi')#这里的位置参数必须在关键字参数之前,否则会有语法错误。
```

默认参数

对某一个或多个参数指定一个默认值:

```
def func_name(arg1,arg2=v2):  
    expression  
  
def func_name(arg1=v1,arg2=v2):  
    expression
```

示例:

```
def func(i,j,k=1):  
    print i,j,k  
  
func(2,200)  
func(1,2,3)#默认参数有传值,会覆盖掉默认值
```

在函数定义的时候,带默认值的参数必须放在不带默认值参数的后边。

可变参数

在python中也支持可变参数。

参数有两种,一种是位置参数,另一种是关键字参数。那么可变参数同样支持这两种方式。

可变位置参数:

```
def func_name(*args):  
    #函数中的args是一个元组  
    expression  
  
func_name(v1,v2,v3...)
```

示例:

```
def func(*args):  
    print type(args)  
    for x in args:  
        print x,  
    print  
  
func(1,2,2,3,'a','v')  
[root@yeahmonitor ~]# python func.py  
<type 'tuple'>  
1 2 2 3 a v
```

*args这种方式定义的方式,只能以位置传递参数进去,可变位置参数。

可变关键字参数:

```
def func_name(**kwargs):  
    #函数中的kwargs是一个字典  
    expression  
  
func_name(k1=v1,k2=v2,k3=v3...)
```


示例：

```
def func(**kwargs):
    print type(kwargs)
    for k,v in kwargs.items():
        print "%s => %s" % (k,v)

func(i=1,j=2,k='aa')

[root@yeahmonitor ~]# python func.py
<type 'dict'>
i => 1
k => aa
j => 2
```

混合使用：

非可变参数必须要在可变参数之前

```
def func(i,j,*args):
    print i,j
    for x in args:
        print x

func(1,2,'1',"a","abc")
```

可变位置参数要在关键字参数之前

参数解包

参数解包发生在函数调用时。

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

def func(i,j,k):
    print i,j,k

li = [1,2,3]

func(li[0],li[1],li[2]) #传统的调用方法

func(*li) #将列表解包并传递进去,这里是一个解包的过程。
```

* 用于解包序列为位置参数。

** 用于解包字典为关键字参数：

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

def func(i,j,k):
    print i,j,k

li = [1,2,3]

#func(li[0],li[1],li[2])

func(*li)

d = {'i':2,'j':100,'k':300}

func(**d)
```

函数返回值

return关键字

```
def func(i):
    i *= 2
    return i

j = func(2)

print j
```

可以返回任何对象

可以变相返回多值

```
def func(i):
    j = i * 2
    k = i * 9
    return j,k #这里返回两个值，调用的时候需要有两个变量接受，这里其实是将j,k构成一个元组返回。
x,y = func(2)

print x,y
```

默认返回值,当函数中没有return关键字的时候，函数会默认返回None,所以所有的python函数都有返回值。

递归函数

函数体内调用自身的函数

计算一个值的阶乘

```
def fact(n):
    if n <= 1:
        return n
    return fact(n-1) * n
f = fact(10)
print f
```

递归函数需要有合适的退出条件

python中函数递归的最大深度为1000

python中应尽量避免递归,绝大多数递归都是可以转换为迭代的。

求阶乘的函数转换为迭代：

```
def fact2(n):
    ret = 1
    for x in range(n,1,-1):
        ret *= x
    return ret

print fact2(10)
```

python中的迭代要比递归快很多。

以上为函数的基本定义知识点。

在python中，函数是一等对象，可以像值一样的赋值，作为函数的返回值返回。

函数作为参数

```
def func1(arg):
    expression

def func2(func):
    expression

func2(func1)
```

示例：

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

def func(f):
    print "this is func,i will call %s" % f.__name__
    f()

def func2():
    print "this is func2"

func(func2)
```

x.__name__ 获取某个对象的名称，python中函数也是一个对象

高阶函数

像上边那样，以函数对象作为参数的函数叫高阶函数

下边介绍三个内置的高阶函数：

filter:filter(function or none,seq)当函数返回为true的时候，会将序列的当前值加进来。

```
In [1]: li = [1,2,3,4,5,6,7,8,10]
```

```
In [2]: def f(x):
...:     return x % 2 == 0
...:

In [3]: filter(f,li)
Out[3]: [2, 4, 6, 8, 10]
#这类似于列表解析
```

map:对列表中的所有元素执行一个函数

```
In [4]: def f2(x):
...:     return x * 2
...:

In [5]: li
Out[5]: [1, 2, 3, 4, 5, 6, 7, 8, 10]

In [6]: map(f2,li)
Out[6]: [2, 4, 6, 8, 10, 12, 14, 16, 20]
#这个也可以使用列表函数实现
```

reduce:

```
In [1]: li = [1,2,3,4]

In [2]: def sum(x,y):
...:     return x * y
...:

In [3]: reduce(sum,li)
Out[3]: 24
```

函数嵌套定义

def ext(): def internal(): expression

返回函数

```
def external():
    def internal():
        expression
    return internal
```

示例:

```
def ext_func():
    print "this is ext"
    def inn_func():
        print "this is inn"
    return inn_func

f = ext_func() #这里会返回函数inn_func

f()
```

有了这些基础,那么加入我们要计算一个函数的执行时间,但又不想破坏这个函数,不想给这个函数增加其他额外的代码?

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

import time
import datetime

def func(arg):
    time.sleep(arg)

def timeIt(func):
    def warp(arg):
        start = datetime.datetime.now()
        func(arg)
        end = datetime.datetime.now()
        cost = end - start
        print "execute %s spend %s" % (func.__name__,cost.total_seconds())
    return warp

new_func = timeIt(func) #这里会返回一个新的函数

new_func(3) #调用新的函数,并传值进去
```

这样就以一种非侵入式的方式包装的这个函数，并且增加了我们需要的功能。这就是python中的装饰器。类似于java的注解。

装饰器

主要就是解耦合，例如上边的我不需要计算函数的执行时间了，那么我直接调用该函数即可。

在python中对装饰器提供更好的方法。

```
@timeIt
def func(arg):
    time.sleep(arg)

func(arg)
```

示例：

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

import time
import datetime

def timeIt(func):
    def warp(arg):
        start = datetime.datetime.now()
        func(arg)
        end = datetime.datetime.now()
        cost = end - start
        print "execute %s spend %s" % (func.__name__,cost.total_seconds())
    return warp

@timeIt #这里是python提供的一个语法糖
def func(arg):
    time.sleep(arg)

func(3)
```

在这里func函数被装饰器包装，所以func.__name__返回的名字并不是'func'了，而是被装饰器改变为warp。要想保留原来函数的__name__、__doc__等元信息，需要做一下修改：

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

import time
import datetime
import functools

def timeIt(func):
    @functools.wraps(func) #增加这一行,将原函数作为值传递进去，表示将原函数的__name__,__module__,__doc__信息更新到装饰器里
    def warp(arg):
        start = datetime.datetime.now()
        func(arg)
        end = datetime.datetime.now()
        cost = end - start
        print "execute %s spend %s" % (func.__name__,cost.total_seconds())
    return warp

@timeIt
def func(arg):
    time.sleep(arg)

func(3)
print func.__name__
```

lambda匿名函数：

Python虽然不是一种函数式编程语言，但仍然给予了函数式编程很大的重视。接下来就聊一下Python函数式编程的知识，其中本文要说的是匿名函数lambda。

Python使用lambda关键字创造匿名函数。所谓匿名，意即不再使用def语句这样标准的形式定义一个函数。这种语句的目的是由于性能的原因，在调用时绕过函数的栈分配。其语法是：

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

其中，参数是可选的，如果使用参数的话，参数通常也会在表达式之中出现。

lambda语句的使用方法(无参数)：

```
In [1]: def func():
...:     return 1
...:

In [2]: lambda:1 #等价于上边定义的函数,不接受参数,直接返回值
Out[2]: <function __main__.<lambda>>

In [3]: f = lambda:1 #lambda会返回一个函数,将该函数赋值给变量

In [4]: f() #调用函数
Out[4]: 1
```

声明一个带参数的匿名函数:

```
In [5]: f = lambda x,y:x+y #带有两个参数

In [6]: f(10,2)
Out[6]: 12

In [7]: f = lambda x,y=10:x+y #带默认值的参数

In [8]: f(19)
Out[8]: 29

In [9]: f(19,2)
Out[9]: 21

In [10]: f = lambda *x:map(lambda x:x+10,x)#可变位置参数

In [11]: f(1,2,3,4)
Out[11]: [11, 12, 13, 14]

In [12]: f(*[2,3,4,5])
Out[12]: [12, 13, 14, 15]
```

示例:

```
[root@yeahmonitor ~]# cat lam.py
#!/usr/bin/env python

func = {
    "add":lambda x,y:x+y,
    "sub":lambda x,y:x-y,
    "mul":lambda x,y:x*y,
    "div":lambda x,y:x/y
}

print func["add"](1,2)
print func["sub"](3,2)
print func["mul"](4,5)
print func["div"](10,3)
#执行结果
[root@yeahmonitor ~]# python lam.py
3
1
20
3
```

generator 生成器

生成器概念

生成器不会把结果保存到一个系列中,而是保存生成器的状态,在每次进行迭代时计算并返回一个值,直到遇到`StopIteration`异常结束。

生成器语法

生成器表达式:

与列表解析语法相似,只不过把列表解析的`[]`换成`()`

生成器表达式能做的事,列表解析基本都能处理;只不过在需要处理的序列比较大时,列表解析比较费内存。

简单的生成器表达示例:

```
In [1]: gen = (x**2 for x in range(5))

In [2]: gen
Out[2]: <generator object <genexpr> at 0x193e910>
```

```

In [3]: gen.next()
Out[3]: 0

In [4]: gen.next()
Out[4]: 1

In [5]: gen.next()
Out[5]: 4

In [6]: gen.next()
Out[6]: 9

In [7]: gen.next()
Out[7]: 16

In [8]: gen.next()
-----
StopIteration                                Traceback (most recent call last)
<ipython-input-8-b2c61ce5e131> in <module>()
----> 1 gen.next()

StopIteration:

```

生成器表达式，并不能实现较为复杂的功能。所以在工作中一般都是直接使用生成器函数。

生成器函数：

在函数中如果出现了yield关键字，那么该函数就不再是普通函数，而是生成器函数。

生成器函数可以产生一个无限的序列，列表解析是无法这么做的。

yield的作用就是把一个函数变成一个generator,带有yield的函数不再是一个普通函数，Python解释器会将其视为一个generator。

产生无穷奇数的生成器函数：

```

#!/usr/bin/env python

def odd():
    n = 1
    while True:
        yield n
        n += 2

od = odd()

count = 0

for i in od:
    if count >= 5: break
    print i
    count += 1

[root@walter ~]# python gen.py
1
3
5
7
9

```

题外话：生成器是包含有iter()和next()方法的，所以可以直接使用for来迭代.查看帮助信息：

```

#定义一个简单的生成器函数
In [1]: def odd():
...:     yield 1
...:

In [2]: o = odd()
#查看其帮助信息
In [3]: help(o)

odd = class generator(object)
|   Methods defined here:
|   |
|   | __getattribute__(...)
|   |     x.__getattribute__('name') <==> x.name
|   |
|   | __iter__(...)#包含iter方法
|   |     x.__iter__() <==> iter(x)
|   |

```

```

| __repr__(...)
|     x.__repr__() <==> repr(x)
|
| close(...)
|     close() -> raise GeneratorExit inside generator.
|
| next(...)#包含next方法，所以可以直接用在for语句
|     x.next() -> the next value, or raise StopIteration
|
| send(...)#使用该方法传值到生成器中
|     send(arg) -> send 'arg' into generator,
|     return next yielded value or raise StopIteration.
|
| throw(...)#使用该方法抛一个异常进去
|     throw(typ[,val[,tb]]) -> raise exception in generator,
|     return next yielded value or raise StopIteration.

```

由上边帮助信息可以看出，我们可以完全按照迭代器的循环方式遍历一个生成器。

示例说明：

```

#!/usr/bin/env python

def odd():
    n = 1
    while True:
        yield n
        n += 2

od = odd()

count = 0

for i in od:
    if count >= 5: break
    print i
    count += 1

#在for循环执行的时候，每次都会执行odd函数内的代码。
#执行到yield n时，将n的值返回出去，此时函数内的代码不再往下执行，挂起状态，但会保存函内变量的相关信息。
#下次迭代的时，从yield n的下一条语句，n += 2 开始执行，
#这个时候n的值是之前保留的值，函数体内的while循环继续执行，当再次执行到yield n的时候，将n返回出去，此时的n是已经计算过n+=2了
#看起来就好像一个函数在正常执行的过程中被 yield 中断了数次，每次中断都会通过 yield 返回当前的迭代值。

```

yield与return:

在一个生成器中，如果没有return，则默认执行到函数完毕时返回StopIteration；

```

>>> def g1():
...     yield 1
...
>>> g=g1()
>>> next(g)      #第一次调用next(g)时，会在执行完yield语句后挂起，所以此时程序并没有执行结束。
1
>>> next(g)      #程序试图从yield语句的下一条语句开始执行，发现已经到了结尾，所以抛出StopIteration异常。
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>

```

如果遇到return,如果在执行过程中 return，则直接抛出 StopIteration 终止迭代。

```

>>> def g2():
...     yield 'a'
...     return
...     yield 'b'
...
>>> g=g2()
>>> next(g)      #程序停留在执行完yield 'a'语句后的位置。
'a'
>>> next(g)      #程序发现下一条语句是return，所以抛出StopIteration异常，这样yield 'b'语句永远也不会执行。
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration

```

close()

手动关闭生成器函数，后面的调用会直接返回StopIteration异常。

```
>>> def g4():
...     yield 1
...     yield 2
...     yield 3
...
>>> g=g4()
>>> next(g)
1
>>> g.close()
>>> next(g)      #关闭后, yield 2和yield 3语句将不再起作用
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

**** send() ****

生成器函数最大的特点是可以接受外部传入的一个变量，并根据变量内容计算结果后返回。这是生成器函数最难理解的地方，也是最重要的地方，实现后面我会讲到的协程就全靠它了。

```
def gen():
    value=0
    while True:
        receive=yield value
        if receive=='e':
            break
        value = 'got: %s' % receive

g=gen()
print(g.send(None))
print(g.send('aaa'))
print(g.send(3))
print(g.send('e'))

执行结果:
0
got: aaa
got: 3
Traceback (most recent call last):
File "h.py", line 14, in <module>
    print(g.send('e'))
StopIteration
```

执行流程:

1. 通过g.send(None)或者next(g)可以启动生成器函数，并执行到第一个yield语句结束的位置。此时，执行完了yield语句，但是没有给receive赋值。yield value会输出初始值0注意：在启动生成器函数时只能send(None),如果试图输入其它的值都会得到错误提示信息。
2. 通过g.send('aaa')，会传入aaa，并赋值给receive，然后计算出value的值，并回到while头部，执行yield value语句有停止。此时yield value会输出"got: aaa"，然后挂起。
3. 通过g.send(3)，会重复第2步，最后输出结果为"got: 3"
4. 当我们g.send('e')时，程序会执行break然后推出循环，最后整个函数执行完毕，所以会得到StopIteration异常。

总结:

1. 按照鸭子模型理论，生成器就是一种迭代器，可以使用for进行迭代。
2. 第一次执行next(generator)时，会执行完yield语句后程序进行挂起，所有的参数和状态会进行保存。再一次执行next(generator)时，会从挂起的状态开始往后执行。在遇到程序的结尾或者遇到StopIteration时，循环结束。
3. 可以通过generator.send(arg)来传入参数，这是协程模型。
4. next()等价于send(None)

作业待定

IO与文本处理

标准IO设备的操作

print 语句

print 语句可以将内容输出到标准输出上，如print 'hello'；除此之外，print还可以将内容输出到的文件对象里：

```
In [2]: import sys

In [3]: print >> sys.stderr, 'hello' #将内容输出到标准错误中
hello
```

print 函数 包含在__future__ 模块中:


```
In [4]: from __future__ import print_function

In [5]: type(print) #此时的print不再是个语句,而是个函数,这个时候print不能再像上边那么使用了
Out[5]: builtin_function_or_method
```

示例:

```
In [6]: print("hello","world")
hello world

In [7]: print("hello","world",sep="\n")
hello
world
```

`raw_input`函数,是一个内置函数,从标准输入读入内容:

```
In [8]: s = raw_input()
hello

In [9]: s
Out[9]: 'hello'

In [10]: s = raw_input("Plz input >")
Plz input >hello world

In [11]: s
Out[11]: 'hello world'
```

open函数与file对象

`open()`是一个内置函数, `open(name[,mode[,buffering]])`,其实是对file的一个封装,返回一个file对象。下边详细介绍下open函数的modes参数都有哪些:

- `r`以读的方式打开,定位到文件开头,默认的mode
- `r+`以读写的方式打开,定位文件开头,可以写入内容到文件
- `w`以写的方式打开,打开文件的时候会清空文件的内容,并且不能读
- `w+`以读写的方式打开,定位到文件头,并且打开文件的时候也会清空文件的内容
- `a`以写的方式打开,定位到文件的末尾,是一个追加的操作,但并不允许读
- `a+`以读写的方式打开,定位到文件的末尾,追加的方式。
- 在使用以上mode打开文件的时候,如果增加了**b**模式,表示以二进制方式打开

示例:

```
In [12]: f = open('/tmp/test.txt','r')

In [13]: f.read()
Out[13]: 'test\n'

In [14]: f.close()#打开的文件必须要关闭,
#否则随着文件打开的数量,会消耗掉所有的文件描述符

In [15]: f = open('/tmp/test.txt','r+')

In [16]: f.write('123')

In [18]: f.close()
[root@yeahmonitor ~]# cat /tmp/test.txt
123t
```

file对象的操作:

`read()`表示读取文件所有内容到内存里,可以带一个可选参数表示读取多少字节。当打开较大文件的时候慎用,或者是要加个参数表示读取多少字节。`f.read()`或者`f.read(4)`#读取四个字节

`readline()` 读取一行内容

```
In [27]: f = open('/tmp/test.txt','r')

In [28]: f.readline()
Out[28]: 'test\n'

In [29]: f.readline()
Out[29]: 'test\n'

In [30]: f.readline()
Out[30]: 'test\n'

In [31]: f.readline()
```

```
Out[31]: 'testwww\n'
```

`readlines()` 将所有行内容读到一个list里面,会将换行符读取进来

```
In [24]: f = open('/tmp/test.txt','r')

In [25]: f.readlines()
Out[25]: ['test\n', 'test\n', 'test\n', 'testwww\n', 'tadc\n']
```

`write` 写入内容到文件, `f.write('hello')`

`writelines` 将一个序列写入到文件中, 但不会加换行符

```
In [32]: f = open('/tmp/test.txt','w')

In [33]: f.writelines(['11111','hello world','33444'])

In [34]: f.close()
[root@yeahmonitor ~]# cat /tmp/test.txt
11111hello world33444[root@yeahmonitor ~]#
```

`truncate` 清空文件, `f.truncate()`, 但是以a方式打开, 则不会清空, 因为a的方式是定位到文件末尾, 该函数是从文件指针当前的位置开始清空内容的。

`flush` 函数, 将缓冲区的内容写入到硬盘中, `close()` 函数会在关闭文件之前执行此操作。

`seek, seek(offset[, whence])`, `offset` 表示移动多少字节, `whence` 为1的时候表示相对于当前位置移动的; 当2的时候从文件的末尾往后移动, 但不一定所有的平台都支持; 默认为0表示从文件开头往后移动

`tell()` 返回当前文件指针的偏移量:

```
In [35]: f = open('/tmp/test.txt')#默认以r方式打开

In [36]: f.tell()#以r方式打开文件, 定位到文件末尾
Out[36]: 0
In [40]: f.seek(3)

In [41]: f.tell()
Out[41]: 3
In [42]: f.read()
Out[42]: '11hello world33444'
In [43]: f.tell()
Out[43]: 21

In [44]: f.seek(0)#回到文件开头
In [45]: f.tell()
Out[45]: 0

In [46]: f.read()
Out[46]: '11111hello world33444'
```

`close()` 函数, 关闭当前打开的文件

`fileno()` 函数, 返回当前的文件描述符, 一个数字

`isatty()` 函数, 当前打开的文件是否是一个终端设备

`closed` 属性, 当前文件是否关闭, |True,False,f.closed

`file` 对象是一个迭代器:

`next()` 方法, 一行一行的读, 每次读取一行

```
In [48]: f = open('/tmp/test.txt')

In [49]: f.next()
Out[49]: '11111hello world33444'

In [50]: f.next()
-----
StopIteration                                Traceback (most recent call last)
<ipython-input-50-c3e65e5362fb> in <module>()
----> 1 f.next()

StopIteration:
```

with语法

一般情况打开一个文件, 经过操作之后, 都要显式的执行 `xx.close()` 将文件关闭.

with用于需要打开、关闭成对的操作，可以自动关闭打开对象。

```
with expression as obj:#将打开的对象赋值给obj
    expression
#obj的作用域只在with语句中
```

示例：

```
In [55]: with open('/tmp/test.txt') as f:
....:     for line in f:
....:         print(line)
....:
1111hello world3344tadc

tadc
```

with语法是可以自定义的，在面向对象中会更加详细的讲解。

文本处理

字符串处理

分隔和连接

`split`函数，默认以空格分隔，可以传参指定分隔符

```
In [1]: a = "aaa hello world"

In [2]: a.split()
Out[2]: ['aaa', 'hello', 'world']

In [3]: a = "aaa,hello,world"

In [4]: a.split(',')
Out[4]: ['aaa', 'hello', 'world']

In [5]: a.split(',',1)#指定第二个可选参数，指定最大分隔
Out[5]: ['aaa', 'hello,world']
```

`join`函数，字符串的连接,用于连接一个迭代器，返回一个字符串

```
In [6]: it = ["hello","beijing","and xi'an"]

In [7]: ",".join(it) #以逗号连接列表为一个字符串
Out[7]: "hello,beijing,and xi'an"
```

✦也可以连接字符串,由于字符串是不可变的，所以在使用加号连接两个字符串的时候需要创建新的新的内存用来存储数据。

字符串格式化

python中字符串格式化主要有两种方式.

占位符，`%s`字符串，`%d`整数，`%f`浮点数

```
In [8]: "hello %s" % "world"
Out[8]: 'hello world'

In [9]: "hello %s %s" % ("world"," and xi'an")
Out[9]: "hello world  and xi'an"
```

字符串查找

`find`函数，返回子字符串在原字符串中第一次出现的位置,若没有找到则返回-1

```
In [12]: str = "this is a dog"

In [13]: str.find("is")
Out[13]: 2

In [14]: str.find("is",2,4)
Out[14]: 2

In [15]: str.find("is",3,4)#还可以接受两个可选参数，start,end用于标示查找的起始位置和结束位置
Out[15]: -1
```

字符串替换

`replace`函数, 用于替换

```
In [16]: str
Out[16]: 'this is a dog'

In [17]: str.replace("is","are")
Out[17]: 'thare are a dog'
```

`strip`函数, 用来将字符串首尾的空白(空格, 制表符,\r,\n,\t)移除

```
In [18]: s = "hello world "

In [19]: s
Out[19]: 'hello world '

In [20]: s.strip()
Out[20]: 'hello world'

In [21]: s = "hello world\n"

In [22]: s
Out[22]: 'hello world\n'

In [23]: s.strip()
Out[23]: 'hello world'
```

`rstrip`移除右边的空白, `lstrip`移除左边的空白

正则表达式

python中的正则标示是以标准库的形式提供的,在使用之前要先导入 `import re`

正则表达式的基本模式:

- 字面模式: 就是字面长度, 就代表其本身
- 匹配任何字符
- `\w` 匹配一个单词 (字母) `\W`匹配非字母
- `\s` 匹配空白 `\S` 匹配非空白字符
- `\d` 匹配数字
- `^`开头 `$`结尾
- `\` 转义字符

次数的匹配,匹配其前面的字符出现的次数:

```
* 0次或多次
+ 一次或多次
? 零次或一次
{n}出现n次
{m,n}出现m到n次
```

中括号,表示一个范围:

- 中括号用于指向一个字符集合
- 中括号可以使用元字符
- 中括号中的 `.` 表示其字面意思

如 `[a-z]`.

捕获:

- 位置捕获(...)

- 命名捕获(?P...)

```
In [25]: import re

In [26]: s = "this is test for re walter.liu@163.com test"

In [27]: m = re.search(r'[\w.-]+@[ \w.- ]+',s)#r表示字符串中的所有字符都不会被转义

In [28]: m
Out[28]: <_sre.SRE_Match at 0x2156bf8>

In [29]: m.group()
```

```
Out[29]: 'walter.liu@163.com'

In [30]: m = re.search(r'([\w.-]+)@([\w.-]+)',s)#小括号表示位置捕获

In [31]: m.groups()
Out[31]: ('walter.liu',)

In [32]: m = re.search(r'(?P<user>[\w.-]+)@([\w.-]+)',s)#将捕获(小括号中匹配的内容)到的字符赋值给user变量

In [33]: m.groupdict()
Out[33]: {'user': 'walter.liu'}
```

正则表达式中的断言(作为选学):

- 在目标字符串当前匹配位置的前面或后面进行的一次测试,但不占用字符
- 前向断言: (?=...)肯定:(?!...)否定
- 后向断言: (?<=...)肯定:(?<!...)否定

前面出现任何字符,以字符e结尾的字符:

```
In [48]: m = re.findall(r'\w+(?=e)',s)

In [49]: m
Out[49]: ['t', 'r', 'walt', 't']

In [50]: s
Out[50]: 'this is test for re walter.liu@163.com test'
```

前面紧跟@的字符串:

```
In [59]: m = re.findall(r'(?<=@)\w+',s)

In [60]: m
Out[60]: ['163']
In [61]: m = re.search(r'(?<=@)\w+',s)

In [62]: m
Out[62]: <_sre.SRE_Match at 0x235a850>

In [63]: m.group()
Out[63]: '163'
```

下边具体说下re模块的相关方法:

`re.match(p,text)`:p为正则表达式模式, text要查找的字符串, 会返回一个match对象

`re.search(p,text)`:只要在text中匹配到了p就返回, 只返回第一个匹配到的

`re.findall(p,text)`: 将能匹配上的全返回, 会返回一个list

`re.split(p,text)`:按照p匹配, 并且以匹配到的字符为分隔符切割text,返回一个切割后的list

`re.sub(p,s,text)`:替换, 将p匹配到的字符替换为s.

`pattern = re.compile(p)`先编译p模式, 当正则表达式模式比较复杂的时候, 会先编译, 然后再使用`result = patter.match(text)`,这就可以使用编译好的模式去匹配各种字符串了。性能会有所提升。

示例:

```
In [1]: import re

In [2]: text = "c++ python3 python3 perl ruby lua java php"

#match
In [4]: re.match(r'c\++',text)#+要转义
Out[4]: <_sre.SRE_Match at 0x2be66b0>
In [5]: a = re.match(r'c\++',text)#将返回的match对象赋值给一个变量

In [6]: a.group()#使用group方法查看匹配到的内容
Out[6]: 'c++'
#match从头匹配
In [15]: a = re.match(r'java',text)#因为java不在这行字符串的开头, 所以match返回为none.

In [16]: a

#search,不需要从头匹配, 只要能匹配到就返回
In [19]: a = re.search(r'java',text)

In [20]: a
```

```

Out[20]: <_sre.SRE_Match at 0x2466bf8>

In [21]: a.group()
Out[21]: 'java'

#findall 返回所有匹配到的内容, 返回一个列表
In [22]: text
Out[22]: 'c++ python3 python3 perl ruby lua java php'

In [23]: re.findall(r'python',text)
Out[23]: ['python', 'python']

In [24]: li = re.findall(r'python',text)

In [25]: li
Out[25]: ['python', 'python']

#split
In [27]: li = re.split(r' perl',text)

In [28]: li
Out[28]: ['c++ python3 python3', ' ruby lua java php']

#sub
In [29]: re.sub(r'ruby','php5',text)
Out[29]: 'c++ python3 python3 perl php5 lua java php'

```

元字符的使用:

```

In [30]: re.findall(r'p+',text)#匹配一个或多个字符p
Out[30]: ['p', 'p', 'p', 'p', 'p']

In [31]: re.findall(r'p[a-zA-Z]+',text)#p后面跟一个或多个字母
Out[31]: ['python', 'python', 'perl', 'php']

In [32]: re.findall(r'c[a-zA-Z]*',text)*#匹配前面的字符出现0次或多次
Out[32]: ['c']

In [34]: re.findall(r'c[^a-zA-Z]*',text)#^用在括号中意思是取反, 本例中是c后面跟0个或多个非字母的字符
Out[34]: ['c++ ']

In [35]: re.findall(r'c[a-zA-Z]?',text)#?表示前面的字符出现0次或1次
Out[35]: ['c']

In [36]: re.findall(r'[p][a-zA-Z]+',text)
Out[36]: ['python', 'python', 'perl', 'java', 'php']

In [37]: re.findall(r'p[^0-9]+[j][a-zA-Z]+',text)
Out[37]: ['python', 'python', 'perl ruby lua java php']

#^匹配开头
In [38]: re.findall(r'^c..',text)
Out[38]: ['c++']

In [41]: re.findall(r'p\w+',text)#\w还可以匹配到数字
Out[41]: ['python3', 'python3', 'perl', 'php']

In [42]: re.findall(r'p\w\d',text)
Out[42]: ['python3', 'python3']

In [44]: re.findall(r'p\w{5,9}',text)
Out[44]: ['python3', 'python3']

#*?非贪婪模式,+?非贪婪模式
In [52]: re.findall(r'p[^0-9]*',text)#贪婪
Out[52]: ['python', 'python', 'perl ruby lua java php']

In [53]: re.findall(r'p[^0-9]*?',text)#非贪婪
Out[53]: ['p', 'p', 'p', 'p', 'p']

In [54]: re.findall(r'p[^0-9]+?',text)#非贪婪
Out[54]: ['py', 'py', 'pe', 'ph']

In [55]: re.findall(r'p[^0-9]+',text)#贪婪
Out[55]: ['python', 'python', 'perl ruby lua java php']

#()分组, (<name>pattern)命名分组

In [65]: a = re.findall(r'(p[a-zA-Z]+)([0-9])',text)

```

```
In [66]: type(a)
Out[66]: list

In [67]: a[1]
Out[67]: ('python', '3')

In [68]: a[1][0]
Out[68]: 'python'

In [69]: a
Out[69]: [('python', '3'), ('python', '3')]

In [75]: a = re.search(r'(?P<name>p[a-zA-Z]+)(?P<version>[0-9])',text)

In [76]: type(a)
Out[76]: _sre.SRE_Match

In [77]: a.group
a.group      a.groupdict  a.groups

In [77]: a.groupdict()
Out[77]: {'name': 'python', 'version': '3'}

In [78]: a.group()
Out[78]: 'python3'

In [79]: a.group('name')
Out[79]: 'python'

#compile
In [80]: p = re.compile(r'(?P<name>p[a-zA-Z]+)(?P<version>[0-9])')#先编译

In [81]: r = p.search("python3")

In [82]: r.groupdict()
Out[82]: {'name': 'python', 'version': '3'}
```

一篇很全面讲解python正则表达式的文章:<http://www.cnblogs.com/huxi/archive/2010/07/04/1771073.html>

案例分析

有多个日志文件，统计访问前十的IP地址和访问次数。

面向对象基础

在了解面向对象之前，先了解下编程范式：

- 编程范式是一类典型的编程风格，是一种方法学
- 编程范式决定了程序员对程序执行的看法
- OOP中，程序是一系列对象的相互作用
- python支持多种编程范式：面向过程，面向对象，面向切面(装饰器部分)等

OOP思想

- 面向对象的基本哲学：世界由具有各自运动规律和内部状态的对象组成，对象之间的相互作用和通讯构成了世界
- 唯一性，世界上没有两片相同的树叶,同样的没有两个相同的对象
- 分类性，分类是对现实世界的抽象
- 三大特性，继承、多态和封装

再来复习下类型：

- 计算机是用来处理数据的
- 数据是一段内存
- 类型告诉计算机这段内存是如何组织
- 通过对基本类型的组织，可以构造更复杂的类型（如list,set等）

```
In [1]: li = [2,3,4,5]#列表中的每个元素都是整数，整数是一个基本类型，列表是由基本类型组织成的一个复杂的数据类型

In [2]: f = lambda x:sum(x)/len(x)

In [3]: f(li)
Out[3]: 3
```

另一个组织数据的示例：

假如有一个表示门的数据

门有两个属性：门牌号和打开/关闭状态

有两个操作，打开和关闭

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

#定义两个门的数据，用列表表示。第一个元素表示门的编号，
#第二个元素表示门的状态
door1 = [1,'closed']
door2 = [2,'closed']

#定义两个函数，表示可以对门执行的操作：打开、关闭
def openDoor(door):
    door[1] = 'open'

def closeDoor(door):
    door[1] = 'closed'

openDoor(door1)
print door1

closeDoor(door1)
print door1

#以上组织数据的方式有一个弊端，就是假如我们把列表的两个元素换下位置，那么我们对门的操作（打开或关闭）是无法完成的。也就是说作为门这个对象的使用者需要了解
```

用类来组织数据

```
#!/usr/bin/env python
#使用class关键字声明一个门的类
class Door(object):

    def __init__(self,num,status):
        self.num = num
        self.status = status

    def open(self):
        #定义对门的打开操作,作用域在该类里面
        self.status = 'open'

    def close(self):
        #定义对门的关闭操作，作用域在类里面
        self.status = 'closed'

door = Door(1,'closed')#实例化一个门的示例，编号为1的门

door.open()#对编号为1的门执行打开操作
print door.num,door.status#输出门1的相关属性

door.close()#执行关闭操作
print door.num,door.status

print type(door)

#关于类的具体定义下边会详解
```

类与实例

- 类 是一类实例的抽象，抽象的属性和操作，如Door类
- 实例 是类的具体化,door = Door(1,'status')，实例化door就是一个具体的对象了。

定义类

```
class Name:
```

这种方式在python3和python2.3之前比较流行，但是在python2.4之后所有类都要继承自object类。

```
class Name(parents):
```

示例：

```
#!/usr/bin/env python
#object是所有类的父类，定义了一些通用的操作
class Door(object):
    #类里面的操作称之为方法，
    def __init__(self,num,status):
        self.num = num
```



```

        self.status = status

    def open(self):
        self.status = 'open'

    def close(self):
        self.status = 'closed'

door = Door(1,'closed')

door.open()
print door.num,door.status

door.close()
print door.num,door.status

print type(door)

```

定义方法:

```

class Name(parents):
    def method(self,args...):
        #定义一个方法, 第一个参数都是
        #self,表示实例本身.调用方法的时候是不需要传递
        #self这个参数的,解释器会自动的将当前的实例传递
        #给self
        body

```

关于self:

```

door = Door(1,'open')#self代表的就是door这个实例本身

door2 = Door(2,'closed')

```

构造方法:

```

class Name(parents):

    def __init__(self,args...):
        expression

instance = Name(args...)

```

`__init__` 就是python类中的构造方法.实例化类的时候就会调用构造方法 `door = Door(1,'open')` 传递的参数就是构造方法所需要的参数.

self关键字:

self代表实例本身

实例变量

```

class Door(object):

    def __init__(self,num,status):
        self.num = num
        self.status = status

    def open(self):
        self.status = 'open'

    def close(self):
        self.status = 'closed'

```

`self.num` 就是一个实例变量(也可称为属性), 实例变量一般都在构造函数中定义, 但也可以在类的其他方法里面定义(但要尽可能在再构造函数中定义实例变量)。实例变量是依附于某一个具体的实例的。定义语法为 `self.argName`。

实例方法 刚才Door类中定义的方法都称为实例方法(一般简称为方法)。实例方法也是可以动态修改的:

```

#!/usr/bin/env python

class Door(object):

    def __init__(self,num,status):
        self.num = num
        self.status = status

    def open(self):
        self.status = 'open'

```

```

    def close(self):
        self.status = 'closed'

door = Door(1, 'closed')

def test():
    print "test"

door.test = test
#动态的给door实例增加一个test属性，该属性的值为test函数，该属性只对当前实例(door)有效。

door.test()

```

接下来介绍下，类的封装性

私有成员

- 以双下划线开始，不以双下划线结束
- python中其实没有真正的私有成员

```

#!/usr/bin/env python

class Door(object):

    def __init__(self,num,status):
        self.num = num
        self.status = status

    def open(self):
        self.status = 'open'

    def close(self):
        self.status = 'closed'

    def __test(self):#定义了一个私有方法
        print "__test"

door = Door(1, 'closed')
#door.__test()#这一行会报错
door._Door__test()#这一行是可以正常打印的，所以python中没有真正的私有成员，但是在生产上不要使用这种方式调用私有成员。

```

类变量

- 定义在实例方法之外的变量
- 所有实例共享类变量,但某一个实例对类变量的修改不会影响其他实例和类本身
- 类变量可以直接访问

```

#!/usr/bin/env python

class Door(object):
    a = 1#定义一个类变量
    def __init__(self,num,status):
        self.num = num
        self.status = status

    def open(self):
        self.status = 'open'

    def close(self):
        self.status = 'closed'

    def __test(self):
        print "__test"

door1 = Door(1, 'closed')
door2 = Door(2, 'open')

print Door.a#可以直接使用类访问类变量
print door1.a#通过实例访问类变量
print door2.a
#以上三行输出的值是一样的

```

示例二：

```

#!/usr/bin/env python

class Door(object):
    a = 1

```

```

    def __init__(self,num,status):
        self.num = num
        self.status = status

    def open(self):
        self.status = 'open'

    def close(self):
        self.status = 'closed'

    def __test(self):
        print "__test"

door1 = Door(1,'closed')
door2 = Door(2,'open')

print Door.a
print door1.a
door1.a = 100
print door2.a
print Door.a
print door1.a

[root@liuzhenwei ~]# python d2.py
1
1
1
1
100

```

类方法

- 使用@classmethod装饰器装饰的方法
- 第一个参数代表类本身
- 类方法可以直接调用
- 类方法里可以定义类变量

示例：

```

#!/usr/bin/env python

class Door(object):
    a = 1
    def __init__(self,num,status):
        self.num = num
        self.status = status

    def open(self):
        self.status = 'open'

    def close(self):
        self.status = 'closed'

    def __test(self):
        print "__test"
    #定义一个类方法
    @classmethod
    def test(cls):
        cls.b = 200#定义一个类变量
        print "test class func"

Door.test()#直接使用类调用类方法
print Door.b #打印类变量
d1 = Door(1,'open')

d1.test()#通过实例调用类方法

#在类方法里不能调用实例相关的东西，即不能使用self

```

静态方法

- 静态方法以staticmethod装饰器装饰
- 静态方法也可以直接调用
- 静态方法没有限定第一个参数

示例：

```

#!/usr/bin/env python

```

```

class Door(object):
    a = 1
    def __init__(self,num,status):
        self.num = num
        self.status = status

    def open(self):
        self.status = 'open'

    def close(self):
        self.status = 'closed'

    def __test(self):
        print "__test"

    @classmethod
    def test(cls):
        cls.b = 200
        print "test class func"

    #定义一个静态方法
    @staticmethod
    def test2():
        print "test2 static func"

Door.test2()#通过类调用静态方法
d1 = Door(1,'open')

d1.test2()#通过实例调用静态方法

```

属性

- 属性以@property装饰器装饰
- 属性的setter方法
- 属性的适用场合

属性是属于实例的。

示例：

```

#!/usr/bin/env python

class Door(object):
    a = 1
    def __init__(self,num,status):
        self.num = num
        self.status = status

    def open(self):
        self.status = 'open'

    def close(self):
        self.status = 'closed'

    def __test(self):
        print "__test"

    @classmethod
    def test(cls):
        cls.b = 200
        print "test class func"

    @staticmethod
    def test2():
        print "test2 static func"

    #定义一个属性, @property装饰器可以使被装饰
    #的方法称为一个属性, 类似于其他语言的get方法
    @property
    def opened(self):
        return self.status == 'open'

    #当一个方法被@property装饰之后, 自己
    #也就称为另一个装饰器
    @opened.setter
    def opened(self,value):
        #此opened就类似于其他语言的set方法

```

```
#该方法必须与上一个方法同名
    if value:
        self.status = 'open'
    else:
        self.status = 'closed'

d1 = Door(1, 'open')
print d1.opened #实例调用属性

d1.opened = False
#给该属性赋值，相当于调用了该属性的set方法进行赋值操作

print d1.opened
#调用属性，也就是相当于调用了get方法
```

为什么要这么麻烦的使用这个呢？？

下边是一个简单的时间类，默认以字符串的形式返回当前的时间，也可以以字符串的形式设置时间，整体来说是封装了私有属性__time：

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

import datetime

class T(object):

    def __init__(self):
        self.__time = datetime.datetime.now()
    @property
    def time(self):
        return self.__time.strftime('%Y-%m-%d %H:%M:%S')

    @time.setter
    def time(self,value):
        self.__time = datetime.datetime.strptime(value,'%Y-%m-%d %H:%M:%S')

t = T()

print t.time

t.time = '2015-12-20 15:30:00'

print t.time
```

在工作中，有些需求需要对IP地址进行处理，因为在计算机中IP都是整数的形式做运算的，但是展现出来的时候都是字符串形式（点分十进制），便于人类查看。就可以使用上述方法实现对IP操作的封装。

以上就是python中关于类的基本定义与操作。下边再介绍一些类的更高级的操作。

魔术方法

- 魔术方法总是由双下划线包围，如__init__，这些方法都是有特殊用途的
- __new__ , __init__ , __del__
- __cmp__ , __eq__ , __ne__ , __lt__ , __le__ , __gt__ , __ge__
- __add__ , __sub__ , __mul__ , __div__
- __str__ , __repr__ , __unicode__

__new__方法，会在元类里面讲解到，这里暂时不做讨论。

__del__方法，简单理解为是一个析构函数，当一个对象被垃圾回收机制回收的时候，其实就是调用了该方法，可以在该方法中定义一些释放资源的操作，如文件的关闭、数据库连接的关闭等。

__cmp__方法，是一个通用的比较函数。在python3中该方法发生了较大的变化。一般工作中不会使用该方法，而是分别使用__eq__、__ne__等之类的方法。

__eq__方法，定义了一个等于的行为，重载了等于操作符：

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

class T(object):

    def __init__(self,a):
        self.a = a
    def __eq__(self,other):
        return self.a == other.a

t1 = T(100)
```

```
t2 = T(100)

print t1 == t2 #调用了__eq__方法
print t2 == t1
```

`__add__`方法:

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

class A(object):
    def __init__(self,a):
        self.a = a
    def __add__(self,other):
        return A(self.a + other.a)

a1 = A(200)
a2 = A(500)

a3 = a1 + a2

print a3.a #输出700

#其实, str对象也是定义了一个__add__方法, 重载了加号为字符串连接
```

`__str__`方法, 定义了我们使用str方法活直接print对象时候的一个行为:

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

class S(object):

    def __init__(self,a):
        self.a = a

    def __str__(self):
        return "a=%s" % self.a

a = S('hello')

print str(a)#输出 a=hello
```

`__repr__`类似于`__str__`, 不过`__repr__`返回的是机器可读的字符串, `__str__`返回的是人类可读的.

`__unicode__`返回一个unicode的对象u'hello', 类似于`__str__`方法.

`__hash__`方法:

```
In [4]: class A(object):
        def __hash__(self):
            return 123
        ...:

In [5]: a = A()

In [6]: hash(a)
Out[6]: 123

#我们定义类的时候是不需要定义__hash__方法 的, 因为父类object已经定义过了。
```

`__getattr__`方法, 当一个实例有这个属性的时候直接返回, 如果没有会调用该方法。

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

class Test(object):

    def __init__(self):
        self.a = 1
    def foo(self):
        print "call foo"
    def __getattr__(self,name):
        print "get attr %s" % name
        return name

t = Test()

print t.a
t.foo()

print t.bar
```

```
#没这个方法，也没这个属性，所以会调用__getattr__方法
```

`__enter__` 和 `__exit__`，在IO的部分说过with语法，只要一个类实现了这个两个方法就可以使用with 语法：

```
class W(object):

    def __init__(self):
        self.f = open('/tmp/test.txt','w')

    def __enter__(self):
        return self.f

    def __exit__(self,*excinfo):
        #该方法接受一个可变参数，出现异常的时候将异常传
        #值给该参数，在该函数内根据异常可以做相应的处理
        self.f.close()

with W() as f:
    f.write("hello world") #向文件中写一行内容
```

面向对象进阶

继承

在面向对象，继承是一个很重要特性。

子类与父类：子类是对父类的一种扩展，在父类的属性和方法上进行一些扩展

示例：

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

#定义一个带编号和状态的门类
class Door(object):
    def __init__(self,num,status):
        self.num = num
        self.status = status

    def open(self):
        self.status = 'open'

    def close(self):
        self.status = 'closed'

#定义一个可以锁的门的类
class Lockable(object):
    def __init__(self,num,status,locked):
        self.num = num
        self.status = status
        self.locked = locked

    def open(self):
        if not self.locked:
            self.status = 'open'
        else:
            print "the door is locked"

    def close(self):
        self.close = 'closed'
```

上述两个类，其实有很多是相同的东西，只是在Lockable类中新增某些新的门的特性而已。

使用子类继承父类：

```
#继承自父类Door
class Lockable(Door):
    def __init__(self,num,status,locked):
        super(Lockable,self).__init__(num,status)#调用父类的构造函数
        self.locked = locked

    #对open方法进行重载
    def open(self):
        if not self.locked:
            #调用父类的方法
            super(Lockable,self).open()
        else:
            print "the door is locked"
```

重写：

上边的示例中，针对于`open`方法就是进行了重写。子类中的方法与父类的方法名相同，其实构造函数也进行了重写，扩展了一些功能。

上边例子中我只是为了扩展一点内容而已，不是真的需要完全重写父类的方法，所以在子类的同名方法中，我们就需要调用父类的方法：

使用`super(className,self)`来调用父类的同名方法,这样就构造了一个`super`对象，代表的是当前类的父类。

示例：

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

class Door(object):
    def __init__(self,num,status):
        self.num = num
        self.status = status

    def open(self):
        self.status = 'open'

    def close(self):
        self.status = 'closed'

class Lockable(Door):
    def __init__(self,num,status,locked):
        super(Lockable,self).__init__(num,status)
        self.locked = locked

    def open(self):
        if not self.locked:
            super(Lockable,self).open()
        else:
            print "the door is locked"

ld = Lockable(1,'closed',True)

ld.open()
# 输出 the door is locked
```

还可以对子类增加一个`locke`的方法：

```
class Lockable(Door):
    def __init__(self,num,status,locked):
        super(Lockable,self).__init__(num,status)
        self.locked = locked

    def open(self):
        if not self.locked:
            super(Lockable,self).open()
        else:
            print "the door is locked"

    def lock(self):
        self.locked = True
```

多继承

大多数情况只会用到单继承，但是python也是提供多继承的支持的。

多继承语法：

```
class ClassName(p1,p2,...):
```

示例：

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

class A(object):
    #若一个类没有定义构造方法，python解释器默认会生成一个这样的构造方法
    def __init__(self):
        pass
    def ma(self):
        print "A.method.a"

class B(object):
    #该类没那个定义__init__方法，默认会自动生成
    def mb(self):
```



```

        print "B.method.b"

class C(A,B):
    pass

c = C()
#虽然在C类里什么都没做，但它已经从A，B里面继承了两个方法
c.ma()
c.mb()

```

但是多继承中会存在某些问题，加入C的两个父类的某个方法名称是一样的。那么子类会选择第一个父类的方法执行。

```

#!/usr/bin/env python

class A(object):
    def __init__(self):
        pass
    def m(self):
        print "A.method.a"

class B(object):

    def m(self):
        print "B.method.b"

class C(A,B):
    pass

c = C()
c.m()#输出A.method.a

```

目前来看就是子类就是执行的其父类列表里左边的父类的方法。那么看下边的示例：

```

#!/usr/bin/env python

class A(object):
    def __init__(self):
        pass
    def m(self):
        print "A.method.a"

class B(A):

    def m(self):
        print "B.method.b"

class C(A,B):
    pass

c = C()
c.m()
#执行上述代码后会抛出一个错误：
[root@liuzhenwei ~]# python cl.py
Traceback (most recent call last):
  File "cl.py", line 15, in <module>
    class C(A,B):
TypeError: Error when calling the metaclass bases
  Cannot create a consistent method resolution
  order (MRO) for bases A, B

```

MRO:方法确定的一个顺序，在python中存在多继承，当继承列表中有同名方法的时候，python就是根据这个MRO来选择子类该执行哪个方法的。

包管理

如何使用包：

```
import
```

```
from ...import ...
```

```
from ...import ... as ...
```

```
import sys 此时就可以使用sys包了
```

```
from os import path 引用os包下面的path包
```

```
from os import path as p 将引用进来的path包做了个别名为p
```

创建自己的模块：

单文件模块：

- 每个python文件都是一个模块
- 文件名就是模块名

注：包名尽量要全是小写

示例：

```
[root@liuzhenwei t]# vim mod1.py#创建一个空的python文件
#在ipython下直接import即可
In [1]: import mod1

In [2]: help(mod1)
```

目录模块：

- **init.py** 要想使目录称为一个模块，该目录下必须包含这个文件
- 子模块,在这个目录下其他所有文件都称为一个子模块
- **all**属性,在**init.py**中可以没有任何内容，但也可以有**all** 属性

示例：

```
[root@liuzhenwei t]# ls foo/
bar.py  __init__.py

#这个时候就可以使用下属方引入bar子模块
from foo import bar
或
import foo.bar

#__all__属性

[root@liuzhenwei t]# cat foo/__init__.py
__all__ = ['bar',]#保存当前目录下所有子模块的列表

In [1]: from foo import *#会将__all__属性中列表的锁有子模块导入进来

In [2]: bar
Out[2]: <module 'foo.bar' from 'foo/bar.pyc'>
```

如果这个目录下还有一个子目录的话，那么子目录也必须得有**init.py**这个文件。

示例：

```
[root@liuzhenwei t]# ls foo/
bar.py  bar.pyc  __init__.py  __init__.pyc
[root@liuzhenwei t]# cat foo/bar.py
def func():
    print "test mod"

[root@liuzhenwei t]# cat test.py
#!/usr/bin/env python

from foo import bar

bar.func()
```

注：import即执行，即在import某个文件模块的时候，是会执行这个python文件的，如：

```
[root@liuzhenwei t]# cat foo/bar.py
def func():
    print "test mod"

print 11#这有一行输出语句，在执行import导入该模块的时候会被执行
In [1]: from foo import bar
11
```

解决方法：if `__name__ == '__main__':`

```
[root@liuzhenwei t]# cat foo/bar.py
def func():
    print "test mod"
```

```
if __name__ == '__main__':
#在被其他文件作为模块引入的时候，
#该条件判断为false,所以下边的语句被不会执行，
#但是可以单独手动的执行该文件，python bar.py 。
    print 11
```

模块的查找和引用

`sys` 模块下有一个 `path` 属性，保存了Python默认的查找路径，按照顺序依次往后查找所引用的包：

```
>>> import sys
>>> 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/lib2to3', ...]
#列表的第一个值为空，表示当前目录，当使用import导入模块的时候，默认会先查找当前目录下是否有该模块，然后依次查找后边给出的路径。
```

在工作中，有时候也会见到有些同事或者其他朋友写程序的时候，将一个特定的查找路径追加到`sys.path`中，方便程序导入模块。但一般不建议这么做，尽量少修改系统原有的“环境变量”等信息。

简单了解一个模块：

OS 模块

- 为访问操作系统的特定熟悉提供方法
- 提供了对平台模块的封装（对windows,对mac的封装等）

对环境变量的操作：可以修改或者获取环境变量，修改环境变量是持久性修改的。

```
>>> import os
>>> os.environ
{'SSH_ASKPASS': '/usr/libexec/openssh/gnome-ssh-askpass', 'LESSOPEN': '||/usr/bin/lesspipe.sh %s', 'SSH_CLIENT': '10.25.0.1 49909 22', ...}
```

工作目录的相关操作：`os.getcwd()` 获取当前的工作目录

文件系统相关的操作：

`os.access(path,mod)` 判断对一个文件或者目录是否具有指定的权限

`mode`参数的可选值：R_OK, W_OK,和 X_OK

```
>>> os.access('/tmp',os.R_OK)
True
>>> os.access('/tmp',os.X_OK)
True
>>> os.access('/tmp',os.W_OK)
True
```

`os.stat()` 相当于对Linux下stat命令的一个封装

```
>>> os.stat('/tmp')
posix.stat_result(st_mode=17407, st_ino=259075, st_dev=2050L, st_nlink=18, st_uid=0, st_gid=0, st_size=4096, st_atime=1458681523, st_mtime=1458681523, st_ctime=1458681523)
>>> s = os.stat('/tmp')
>>> s.st_atime
1458681523.5486367
>>> s.st_uid
0
>>> s.st_size
4096
```

`os.listdir()` 列车给定目录的内容

```
>>> os.listdir('/tmp')
['orbit-gdm', 'ks-script-V5pzEo.log', 'vmware-walter', 'vmware-config0', 'vgauthsvclog.txt.0', 'pulse-kdbaPr3T3eB', 'pulse-cbhCLNnWv5A']
```

`os.mkdir(path)` 创建目录

`os.makedirs(path)` 创建目录树，相当于 `mkdir -p` 操作

对于OS还有其他好多方法，具体可查看官方文档，接下来说下`os.path`模块。

os.path

`os.path`是`os`的一个子模块,主要是对路径进行解析、创建、测试和其他的一些操作,封装了不同平台的路径操作。

路径解析:

```
>>> from os import path
>>> path.split('/tmp/test/ab')#讲路径切割成了dirname和basename。传入的路径可以不存在
('/tmp/test', 'ab')

>>> path.basename('/tmp/test/ab')
'ab'
#直接返回basename

>>> path.dirname('/tmp/test/ab')
'/tmp/test'
#返回dirname
```

`.splitext(path)`解析扩展名

```
>>> path.splitext('aaa.tar.gz')
('aaa.tar', '.gz')
```

将给定的字符连接成一个路径

```
>>> path.join('a','b','c')
'a/b/c'
```

根据相对路径得到绝对路径:

```
>>> path.abspath('.')
'/root'
>>> path.abspath('../tmp')
'/tmp'
>>> path.abspath('../tmp/test')
'/tmp/test'
```

文件属性相关

`os.path.getatime`,`os.path.getctime`,`os.path.getmtime`,`os.path.getsize`

示例:

```
>>> from os import path
>>> path.getatime('/tmp/test')
1458683334.4796343
```

文件测试

`os.path.isabs`,`os.path.isdir`,`os.path.isfile`,`os.path.islink`等等

示例:

```
>>> from os import path
>>> path.isfile('/tmp/test')
False
>>> path.isdir('/tmp/test')
True
```

判断文件是否存在:

```
>>> from os import path
>>>
>>> path.exists('/tmp/test')
True
>>> path.exists('/tmp/test2')
False
```

sys

接下来再简单了解下`sys`模块,主要提供了系统相关的配置和操作,封装了探测、改变解释器runtime以及资源的交互。

解释器相关信息:

`sys.version`得到解释器的版本信息

`sys.platform`得到当前运行平台,如Linux, Windows等

```
>>> import sys
>>> sys.platform
'linux2'
```

运行时环境:

`sys.argv` 获取传递给脚本的参数, 参数解析类似于bash的方式, 第一个参数代表脚本本身

```
[root@localhost ~]# python arg.py 1 2 3 4 5
['arg.py', '1', '2', '3', '4', '5']
[root@localhost ~]# cat arg.py
#!/usr/bin/env python

import sys

print sys.argv
[root@localhost ~]#
```

`sys.stderr`,`sys.stdin`,`sys.stdout`这些都分别代表一个文件对象

示例:

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

import sys

print >> sys.stderr, "I am error"

#执行脚本, 并重定向标准输出和标准错误
[root@localhost ~]# python err.py
I am error
[root@localhost ~]# python err.py > /tmp/1.log
I am error
[root@localhost ~]# cat /tmp/1.log
[root@localhost ~]# python err.py 2> /tmp/1.log
[root@localhost ~]# cat /tmp/1.log
I am error

#解释清楚, 为什么会出现上面的情形
```

练习

输出给定路径下的所有以.log结尾的文件

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

import os
import sys

def findLogFile(path):
    '''find log file'''
    for filename in os.listdir(path):
        if filename.endswith((".log", ".log.log")):
            yield filename

if __name__ == "__main__":
    for f in findLogFile(sys.argv[1]):
        print f

[root@walter doc]# python dirlog.py /var/log/
wpa_supplicant.log
yum.log
boot.log
```

异常处理

当程序运行过程中出错的时候, 捕捉到该错误, 并执行一些相应的自定义操作。

用法:

```
try:
    expression
except [ex...]:
    expression
#try语句有异常发生后执行except
```

```
try:
    expression
except:
    expression
finally:
    exrepssion
#无论异常是否发生都会执行finally语句块
```

`raise error(message)` 触发异常

`assert condition,message` 当`condition`表达式为`false`的时候触发一个异常, `message`为要打印出的异常信息, 触发异常后并终止程序

简要的异常分类举例:

`a[1]` 如果`a`没有定义, 则触发`NameError`

`a=2;a[1]` 此时会触发一个`TypeError`

`a=[2];a[1]` 触发一个`IndexError`

`a={};a[1]` 触发一个`KeyError`

`raise IndexError` 触发一个异常

`assert False,"error occur"` 条件触发一个异常,并打印异常信息

示例:

```
In [1]: a[1]
-----
NameError                                Traceback (most recent call last)
<ipython-input-1-3ef3908cab7> in <module>()
----> 1 a[1]

NameError: name 'a' is not defined
#由于并没有定义变量a,所以会触发一个NameError的异常
#后边的就是打印出的异常信息
```

使用`raise`手动触发一个`NameError`异常:

```
In [2]: raise NameError("Oops,error")
-----
NameError                                Traceback (most recent call last)
<ipython-input-2-baf5611c9528> in <module>()
----> 1 raise NameError("Oops,error")

NameError: Oops,error
```

在程序中手动产生异常:

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

try:
    a = [0,1]
    print a[1]
    #raise IndexError("error")#如果执行了这一句, 则下一句的assert语句就不会执行
    assert a[1] == 0,"assert error"
except Exception,e:
    print "except:"
    print e
finally:
    print "finally"

#finally是一个可选的语句块, 但是在有些情况下最好要有的,
#比如打开了一个文件, 对文件进行操作, 不管异常是否出现都必须得关闭文件。
```

提高篇

数据结构

前面已经讲过了Python内置的几个数据结构: `list,set,dict,tuple`

下边要介绍一些非内置的数据结构。以标准库的形式提供。

案例1,

需要处理一个日志文件，当前文件的某一行包含你指定的关键字时，你需要返回这一行的前N行内容。

这种情况在监控日志的时候经常有此需求，比如你监控某个服务产生的日志，当某一行出现了"exception"关键字，但仅仅看到exception这一行，对运维来说帮助并不大；这个时候你就需要输出这一行的前面几行的日志内容用来分析。

Python实现了一个高效的deque模块，实现双向队列，可以从两端入队或者出队。

deque是Python的一个标准库，是由C语言开发而成，所以它的效率相当高。

先简单查看下该模块的帮助信息：

```
In [1]: from collections import deque
```

```
In [2]: help(deque)
class deque(__builtin__.object)
| deque([iterable[, maxlen]]) --> deque object
|
| Build an ordered collection with optimized access from its endpoints.
|
| Methods defined here:
```

#由帮助信息，可以简单的看到，该模块的构造函数接收两个可选参数，iterable一个可迭代的元素，若指定这个参数，deque模块会将该元素指定的值添加到队列中；maxlen参数

具体实现代码：

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

import sys
from collections import deque #引入deque队列模块

def search(f,pattern,keepNum):
    ''' if find keywords,return the line and preLines'''
    preLines = deque(maxlen=keepNum)#定义该队列所能保存的最大长度

    for line in f:
        #如果当前行包含指定的关键字，就返回当前行，和队列中保存的数据
        if pattern in line:
            yield line,preLines
            preLines.append(line)#如果当前行没有包含指定的关键字，将当前行保存到队列中

if __name__ == '__main__':

    logFile = sys.argv[1]
    pattern = sys.argv[2]
    keepNum = int(sys.argv[3])

    with open(logFile) as f:
        #search此时是一个生成器函数，直接遍历即可
        for line,preLines in search(f,pattern,keepNum):
            #循环输出队列中保存的内容，也就是所匹配到的行的前N行的内容
            for pline in preLines:
                print pline
            #输出当前所匹配到行的内容
            print line
            print '*****'10

#执行该脚本
[root@walter doc]# python dq.py /var/log/messages Shutdown 5
```

案例2.

有一些task，需要保存到字典中，key为名称，value为内容，但是执行的时候，需要保持存储时的顺序

解决思路：要是使用正常的字典话，将数据保存到字典中，但是读出数据的时候，次序将是无法预测的，及正常的字典返回数据的时候是无序的。

OrderedDict 有序字典,可以按照保存数据的顺序来存储数据。

正常字典无序验证：

```
In [1]: dic = dict()

In [2]: dic['foo'] = 1

In [3]: dic['bar'] = 2

In [4]: dic['fizz'] = 3

In [5]: dic['tom'] = 180
```

```
In [6]: dic['jerry'] = 250

In [7]: dic
Out[7]: {'bar': 2, 'fizz': 3, 'foo': 1, 'jerry': 250, 'tom': 180}#可以看到输出数据的时候并不是保存数据时候的顺序了

In [8]: dic.keys()
Out[8]: ['tom', 'foo', 'bar', 'jerry', 'fizz']#key也是无序保存

In [9]: dic.values()
Out[9]: [180, 1, 2, 250, 3]#值也是无序的
```

OrderedDict有序字典:

```
In [10]: from collections import OrderedDict

In [11]: od = OrderedDict()

In [12]: od['foo'] = 1

In [13]: od['bar'] = 2

In [14]: od['fizz'] = 3

In [15]: od['tom'] = 180

In [16]: od
Out[16]: OrderedDict([('foo', 1), ('bar', 2), ('fizz', 3), ('tom', 180)])#数据的顺序就是当时保存的时候的顺序

In [17]: od.keys()
Out[17]: ['foo', 'bar', 'fizz', 'tom']#key是有序的

In [18]: od.values()
Out[18]: [1, 2, 3, 180]#值也是有序的
```

这个模块一般应用在,某些数据输出的时候,需要与保存的顺序一致。

案例3

统计一篇文章中出现频率最高的前10个单词, (英文文章)

Counter模块, 是字典的一个子类, 用来计数的一个模块。接收一个序列作为初始化参数。它有一个方法为most_common() 出现次数最多的前N项。

```
#!/usr/bin/enc python

import sys
import re
from collections import Counter

with open(sys.argv[1]) as f:
    #注意这里一次读入全部的内容
    worlds = re.findall(r"\w+",f.read().lower())
    print Counter(worlds).most_common(10)

[root@walter doc]# python c.py /var/log/messages
[('walter', 4062), ('mar', 4058), ('27', 3700), ('kernel', 2926), ('17', 1902), ('29', 1877), ('06', 1731), ('48', 1631), ('0', 1488), ('
```

案例4

需要从一个CSV的文件里读入数据, CSV文件的第一行是字段名, 希望读入的数据可以根据字段名来访问。

Python的标准库里面已经提供了一个解决方案, namedtuple 命名元组,而且标准库里也提供了针对csv操作的模块micsv。

```
In [1]: import csv

In [2]: from collections import namedtuple

In [3]: reader = csv.reader(open('./t.csv'))#打开csv文件, 生成一个csv的reader对象, 找个对象就是一个迭代器

In [4]: fieds = reader.next() #先读取迭代器中第一个元素, 也就是csv文件中的第一行内容

In [5]: fieds
Out[5]: ['name', 'gender', 'age', 'salary']#会将每一行的内容按照分隔符", " 拆分成一个list, 即每行内容都会拆分成一个list

In [6]: Contact = namedtuple("Contact", fieds)#生成一个命名元组, 第一个参数"Contact"可以与中等号左边的变量名称不一样

In [7]: Contact
```



```

Out[7]: __main__.Contact

In [8]: type(Contact)#它的类型还是type
Out[8]: type

In [9]: c = Contact(*reader.next())#继续迭代下一行内容。此时读取出来的应该是 csv文件中的第二行内容，是一个list

In [10]: c.name
Out[10]: 'walt0er'

In [11]: c.gender
Out[11]: '1'

In [12]: c.salary
Out[12]: '1000002'

In [13]: type(c) #c的类型为Contact
Out[13]: __main__.Contact

#查看namedtuple帮助，并解释
namedtuple(typename, field_names, verbose=False, rename=False)
#由此可以看出，namedtuple的构造函数，第一个参数为类型名，
#也就是说命名元组其实是创造了一个新的类型出来；
#第二个参数为字段名，相当于这个新造出来的类型的属性名；
#后边两个是可选参数
Returns a new subclass of tuple with named fields.

>>> Point = namedtuple('Point', ['x', 'y'])
>>> Point.__doc__                # docstring for the new class
'Point(x, y)'
>>> p = Point(11, y=22)          # instantiate with positional args or keywords
>>> p[0] + p[1]                  # indexable like a plain tuple
33
>>> x, y = p                     # unpack like a regular tuple
>>> x, y
(11, 22)
>>> p.x + p.y                    # fields also accessible by name
33
>>> d = p._asdict()              # convert to a dictionary
>>> d['x']
11
>>> Point(**d)                  # convert from a dictionary
Point(x=11, y=22)
>>> p._replace(x=100)           # _replace() is like str.replace() but targets named fields
Point(x=100, y=22)

#再看一下创造出来的那个Contact到底是个什么样的类型：
class Contact(__builtin__.tuple)
|   Contact(name, gender, age, salary)
|#由此可见Contact是tuple的一个子类，其构造函数需要四个参数，这四个参数也就是csv文件的第一行内容，生成Contact类型的时候传递进来的。
|   Method resolution order:
|       Contact
|       __builtin__.tuple
|       __builtin__.object
|
|   Methods defined here:

#完整的脚本代码：

#!/usr/bin/env python

import sys
import csv
from collections import namedtuple

contacts = list()#空列表用来保存最后的结果

with open(sys.argv[1], 'rv') as f:
    reader = csv.reader(f)
    Contact = namedtuple("Contact", reader.next())
    for line in reader:
        contacts.append(Contact(*line))
    print contacts

[root@walter doc]# python n.py t.csv
[Contact(name='walt0er', gender='1', age='100', salary='1000002'), Contact(name='wal9ter', gender='1', age='100', salary='1000002'), Cor

```

运维相关

文件目录操作

之前已经介绍过一些Python对文件的操作，这里从运维的角度再了解下。

os模块。

创建目录，os.mkdir

```
In [4]: import os

In [5]: os.mkdir('/tmp/testdir',0644)#0644创建目录的权限

In [6]: ll /tmp/
total 16
-rw-r--r--. 1 root 12 Apr  3 01:29 error.log
-rw-r--r--. 1 root 12 Apr  3 01:29 out.log
-rw-r--r--. 1 root 77 Apr  3 05:40 test2.log
drw-r--r--. 2 root  6 Apr  6 07:03 testdir/
-rw-r--r--. 1 root 66 Apr  3 01:43 test.log
```

递归创建目录。类似mkdir -p操作，os.makedirs

```
In [8]: os.makedirs('/tmp/t1/t2/t3')
```

删除目录，os.rmdir

```
In [11]: os.rmdir('/tmp/testdir/')
```

os模块中暂时没有提供删除非空目录的操作。

修改权限，os.chmod,类似于chmod命令

```
In [13]: os.chmod('/tmp/t1',0777)

In [14]: ll -d /tmp/t1
drwxrwxrwx. 3 root 15 Apr  6 07:06 /tmp/t1/
```

os.chown

```
In [17]: os.chown('/tmp/t1',500,500)#后边两个参数分别为UID和GID

In [18]: ll -d /tmp/t1
drwxrwxrwx. 3 500 15 Apr  6 07:06 /tmp/t1/
```

os.stat与stat命令一样

```
In [19]: os.stat('/tmp/t1')
Out[19]: posix.stat_result(st_mode=16895, st_ino=17822857, st_dev=64768L, st_nlink=3, st_uid=500, st_gid=500, st_size=15, st_atime=14598...
```

其他相关方法os.getlogin(),os.getgid(),os.getuid(),os.getpid()等

shutil模块介绍

该模块简单说是用来复制、归档文件和目录的。

shutil.copyfile(src,dst) 复制一个文件

shutil.copymod(src,dst)复制文件权限.将文件权限同步到其他文件

shutil.copy(src,dst)源必须是文件，但是dst可以是文件或者目录，与cp类似

```
In [24]: shutil.copy('tdir','/tmp/')#tdir为目录

-----
IOError                                Traceback (most recent call last)
<ipython-input-24-bdbceda15912> in <module>()
----> 1 shutil.copy('tdir','/tmp/')

/usr/lib64/python2.7/shutil.pyc in copy(src, dst)
    117     if os.path.isdir(dst):
    118         dst = os.path.join(dst, os.path.basename(src))
--> 119     copyfile(src, dst)
    120     copymode(src, dst)
    121
```

```

/usr/lib64/python2.7/shutil.py in copyfile(src, dst)
    80         raise SpecialFileError("`%s` is a named pipe" % fn)
    81
--> 82     with open(src, 'rb') as fsrc:
    83         with open(dst, 'wb') as fdst:
    84             copyfileobj(fsrc, fdst)

IOError: [Errno 21] Is a directory: 'tdir'

#copy文件
In [29]: shutil.copy('anaconda-ks.cfg','/tmp/')

In [30]: ls /tmp/
anaconda-ks.cfg  error.log  out.log  t1/  test2.log  test.log

```

`shutil.copy2(src,dst)`与`copy`类似,但是源数据也会复制,类似`cp -p`命令,将权限,时间戳等源数据也复制过去。

`shutil.copytree(src,dst)`以`copy2`的方式递归复制一个目录:

```

In [5]: shutil.copytree('/tmp/t1','/tmp/dirt')

In [6]: ls /tmp/dirt/
t2/

```

`shutil.rmtree(path)`递归删除一个目录

```

In [7]: shutil.rmtree('/tmp/dirt')

In [8]: ls /tmp/dirt
ls: cannot access /tmp/dirt: No such file or directory

```

下面简单说一个从2.7版本`shutil`才开始有的新功能:

`shutil.make_archive(base_name, format, root_dir=None, base_dir=None, verbose=0, dry_run=0, owner=None, group=None, logger=None)` 以什么样的格式打包, *basename:打包后的文件名称*, *format:以什么样的格式打包*, *rootdir*压缩的根目录。 *base_dir*开始压缩的目录。 *rootdir*和 *basedir*默认都是当前目录。

```

In [12]: shutil.make_archive('testarch','bztar','/tmp/t1')
Out[12]: '/root/doc/testarch.tar.bz2'

```

查看都支持哪些打包格式:

```

In [13]: shutil.get_archive_formats()
Out[13]:
[('bztar', "bzip2'ed tar-file"),
 ('gztar', "gzip'ed tar-file"),
 ('tar', 'uncompressed tar file'),
 ('zip', 'ZIP file')]

```

使用Python执行Linux命令

`os.system`,并没有返回值,是靠副作用产生输出的

```

In [15]: os.system('ls -l')
total 24
-rw-r--r--. 1 root root 191 Mar 29 07:24 c.py
-rw-r--r--. 1 root root 262 Mar 31 05:57 dirlog.py
-rw-r--r--. 1 root root 527 Mar 27 06:44 dq.py
-rw-r--r--. 1 root root 276 Mar 29 07:55 n.py
-rw-r--r--. 1 root root 459 Mar 29 07:35 t.csv
-rw-r--r--. 1 root root 157 Apr  8 07:00 testarch.tar.bz2
Out[15]: 0#ipython中out表示返回值,可见os.system返回的是执行命令的退出状态码

```

不需要命令返回值的时候,可以使用此函数。

`os.popen`返回一个文件对象

```

In [16]: p = os.popen('ls')

In [17]: p
Out[17]: <open file 'ls', mode 'r' at 0x29fe810>

In [18]: p.read()
Out[18]: 'c.py\n dirlog.py\n dq.py\n n.py\n t.csv\n testarch.tar.bz2\n'

```

`os.popen2`返回两个文件对象,一个是`stdin`,一个是`stdout`

os.popen3返回三个文件对象:stdin,stdout,stderr

```
>>> i,o,e = os.popen3('ls /tmp222')#一个不存在的目录
>>> i.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IOError: File not open for reading
>>> o.read()
''
>>> e.read()
'ls: cannot access /tmp222: No such file or directory\n'
```

os.popen4返回两个文件对象, stdin,stderr和stdout的合并

```
>>> i,o = os.popen4('ls /tmp222')
>>> o.read()
'ls: cannot access /tmp222: No such file or directory\n'
```

就目前来看os.popen4貌似已经满足日常需求了,但是目前的话, popen这组函数已经被标记会废弃的函数了,以后某个版本中可能就不存在这些函数了。

```
In [3]: os.popen3('ls /tmp222')
/usr/bin/ipython:1: DeprecationWarning: os.popen3 is deprecated. Use the subprocess module.#表明已标记为废弃, 推荐使用 subprocess
#!/usr/bin/python
```

在Python中推荐使用subprocess模块来执行命令, 以上的几个函数都可以通过subprocess模块里的某个方法来模拟。

subprocess.Popen(args, stdin=None, stdout=None, stderr=None, shell=False)这里面只列出几个常用的参数:

args 需要执行的命令, 可以是一个序列, 如['ls', '-l'],也可以是一个字符串,如ls -l

stdin,stdout,stderr后边跟一个文件对象, 将相应的输出重定向到指定的文件中。

shell默认为False,当为True的时候表示打开一个shell进程来执行相关的命令。

示例:

```
In [6]: p = subprocess.Popen(['ls', '-l'], stdout=open('/tmp/out', 'w'))#将标准输出保存到文件中

In [7]: p
Out[7]: <subprocess.Popen at 0x1818310>

In [8]: p.returncode

In [9]: p.wait()#执行该函数, 以使命令执行完毕
Out[9]: 0

In [10]: p.returncode#返回命令执行后的退出码
Out[10]: 0

#输出结果
In [11]: cat /tmp/out
total 712
-rw-----. 1 root root    996 Mar 24 06:57 anaconda-ks.cfg
drwxr-xr-x. 2 root root     91 Apr  8 07:00 doc
drwxr-xr-x. 7 root root    60 Apr  3 18:31 python
-rw-r--r--. 1 root root 722768 Mar 24 07:11 setuptools-20.3.1.zip
drwxr-xr-x. 2 root root     6 Apr  6 07:33 tdir
```

但是这种方式在日常工作中显的并不那么优雅, 每次都要再读取文件, 才能获取命令执行的结果。

示例2:

```
In [12]: p = subprocess.Popen(['ls', '-l'], stdout=subprocess.PIPE)#subprocess.PIPE
#默认情况下shell=False, Popen函数会将args参数序列的第一个元素作为可执行程序,
#其他元素作为其参数执行。
In [13]: p
Out[13]: <subprocess.Popen at 0x1818110>

In [14]: p.stdout.read()#读取命令执行结果

Out[14]: 'total 712\n-rw-----. 1 root root    996 Mar 24 06:57 anaconda-ks.cfg\ndrwxr-xr-x. 2 root root     91 Apr  8 07:00 doc\ndrwxr-xr-x. 7 root root    60 Apr  3 18:31 python\n-rw-r--r--. 1 root root 722768 Mar 24 07:11 setuptools-20.3.1.zip\ndrwxr-xr-x. 2 root root     6 Apr  6 07:33 tdir\n'
```

注意, 以上执行的时候, shell参数默认都是False。

示例3:

示例4:

示例5:

总之，当shell=True的时候，会调用系统当前shell来执行命令。如RHEL下的默认shell为bash。一般工作上都将shell设置为True，可以执行Linux下的命令，如当命令中带有管道符"|"的时候。

示例：

执行命令的一个高阶模块：

sh是一个第三方模块，可以将任意Linux命令作为Python的函数来执行，命令参数作为其参数传入：`sh.ls('-l')`，函数返回值是命令的执行结果。`sh.ifconfig()`。

进程管理

psutil是一个第三方模块，需要单独安装：`easy_install psutil`

`psutil.users()` 获取当前系统中登录的用户:

psutil.boot_time() 获取系统启动时间

获取CPU使用百分比:

```
In [6]: psutil.cpu_percent()
```

```
Out[6]: 0.4
```

获取CPU核数:

```
In [7]: psutil.cpu_count()
Out[7]: 1
```

获取CPU使用的百分比: 与top命令获取的是一样的

```
In [10]: psutil.cpu_times_percent()
Out[10]: scputimes(user=0.1, nice=0.0, system=0.2, idle=99.7, iowait=0.0, irq=0.0, softirq=0.0, steal=0.0, guest=0.0, guest_nice=0.0)
```

获取内存信息:

```
In [11]: psutil.virtual_memory()
Out[11]: svmem(total=1025363968, available=816713728, percent=20.3, used=328347648, free=697016320, active=129585152, inactive=81391616,
```

获取某个分区使用情况:

```
In [19]: psutil.disk_usage('/')
Out[19]: sdiskusage(total=13914603520, used=1362849792, free=12551753728, percent=9.8)
```

glances这个监控程序就是基于psutil这个模块开发的。

与进程相关的操作方法:

```
In [20]: p = psutil.process_iter()
#以迭代器的形式返回系统中所有的进程

In [21]: p
Out[21]: <generator object process_iter at 0x2be3fa0>

In [22]: a = p.next()

In [23]: a#返回的第一个进程
Out[23]: <psutil.Process(pid=1, name='systemd') at 46177488>
In [24]: a.
a.as_dict          a.is_running      a.pid
a.children         a.kill           a.ppid
a.cmdline          a.memory_full_info a.resume
a.connections      a.memory_info     a.rlimit
a.cpu_affinity     a.memory_info_ex  a.send_signal
a.cpu_percent      a.memory_maps     a.status
a.cpu_times        a.memory_percent  a.suspend
a.create_time      a.name            a.terminal
a.cwd              a.nice            a.terminate
a.environ          a.num_ctx_switches a.threads
a.exe              a.num_fds         a.uids
a.gids             a.num_threads     a.username
a.io_counters      a.open_files      a.wait
a.ionice           a.parent

#针对某个进程所支持的操作
In [28]: a.exe()#当前进程的执行程序
Out[28]: '/usr/lib/systemd/systemd'
In [30]: a.cpu_times()#当前进程CPU时间的使用
Out[30]: pcputimes(user=0.25, system=1.79, children_user=1.22, children_system=2.3)

In [31]: a.pid#进程ID
Out[31]: 1
```

/proc文件系统

Linux 系统为管理员提供了非常好的方法,使其可以在系统运行时更改内核,而不需要重新引导内核系统,这是通过/proc 虚拟文件系统实现的。/proc 文件虚拟系统是一种内核和内核模块用来向进程(process)发送信息的机制(所以叫做"/proc"),这个伪文件系统允许与内核内部数据结构交互,获取有关进程的有用信息,在运行中(on the fly)改变设置(通过改变内核参数)。与其他文件系统不同,/proc 存在于内存而不是硬盘中。proc

文件系统提供的信息如下:

- 进程信息:系统中的任何一个进程,在proc的子目录中都有一个同名的进程ID,可以找到cmdline、mem、root、stat、statm,以及status。某些信息只有超级用户可见,例如进程根目录。每一个单独含有现有进程信息的进程有一些可用的专门链接,系统中的任何一个进程都有一个单独的自链接指向进程信息,其用处就是从进程中获取命令行信息。
- 系统信息:如果需要了解整个系统信息中也可以从/proc/stat中获得,其中包括CPU占用情况、磁盘空间、内存对换、中断等。
- CPU信息:利用/proc/CPUinfo文件可以获得中央处理器的当前准确信息。
- 负载信息:/proc/loadavg文件包含系统负载信息。
- 系统内存信息:/proc/meminfo文件包含系统内存的详细信息,其中显示物理内存的数量、可用交换空间的数量,以及空闲内存的数量等。

/proc 目录中的主要文件的说明

只简单列出几个经常用到监控中的文件

- /proc/cpuinfo CPU信息
- /proc/loadavg 系统平均负载信息
- /proc/meminfo 内存相关信息，包括物理内存和交换分区
- /proc/swaps 交换分区使用情况

案例1：读取负载信息

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

def load_avg():
    loadavg = {}
    with open('/proc/loadavg') as f:
        avg = f.read().split()#将本行内容以空格分隔为列表
        loadavg['avg_1'] = avg[0]
        loadavg['avg_5'] = avg[1]
        loadavg['avg_15'] = avg[2]
    return loadavg

if __name__ == '__main__':
    for k,v in load_avg().items():
        print k,v
```

案例2:获取内存使用信息

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

from collections import OrderedDict

def meminfo():
    '''get memory info from /proc/meminfo'''
    info = OrderedDict()
    #将读取到的所有内存信息存到一个有序字典里，
    #这样的话，以后可以按照存储的顺序直接读取出来
    with open('/proc/meminfo') as f:
        for line in f:
            fields = line.split(':')
            info[fields[0]] = fields[1].strip()
    return info

if __name__ == '__main__':

    meminfo = meminfo()
    print 'Total memory:{0}'.format(meminfo['MemTotal'])
    print 'Free memory:{0}'.format(meminfo['MemFree'])
```

案例3：监控mariadb进程是否存在

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

def is_mariadb_alive():
    ''' monitor mariadb'''
    ret = os.popen('ps -C mysql -o pid,cmd').readlines()

    return False if len(ret) < 2 else True#python 中实现的类似三元表达式

if __name__ == '__main__':

    print is_mariadb_alive()
    #此处也可以加些功能:
    #如 当返回为False的时候，表示mariadb进程不存在了
    #此时需要启动mariadb: os.system("service apache2 restart")
```

可能会用到的模块argparse,smtplib。

python操作MySQL数据库

安装

```
[root@~]# yum install MySQL-python
```

```
#ipython或者python shell中测试可以正常导入
In [1]: import MySQLdb #不报错,表示安装成功
```

在本地书库创建一个库用于测试:

```
MariaDB [(none)]> create database testpython;
Query OK, 1 row affected (0.00 sec)
```

连接MySQL数据库:

```
In [1]: import MySQLdb

In [2]: conn = MySQLdb.connect(host='localhost',user='root',passwd='123456',db='testpython',port=3306,charset='utf8')
```

- host: MySQL数据库地址
- user:数据库登陆用户名
- passwd:数据库登陆密码
- db:登陆数据库后,需要操作的库名
- port:数据库监听端口,默认为3306
- charset:数据库编码

建立与数据库的连接,其实就是建立了一个MySQLdb.connect()的实例对象conn,这个实例对象常用的操作有:

- commit() 如果数据库表进行了修改,提交保存当前的数据。
- rollback() 如果有限,就取消当前的操作,否则报错
- cursor() 游标指针。

连接成功之后,就要开始操作数据库,MySQLdb用游标cursor的方式操作数据库。

```
#创建一个数据库游标
In [3]: cur = conn.cursor()
```

模块底层其实是调用CAP的,所以,需要先得到当前指向数据库的指针。这也就提醒我们,在操作数据库的时候,指针会移动,如果移动到数据库最后一条了,再查,就查不出什么来了。

下面用cursor()提供的方法来进行操作,方法主要是:

1. 执行命令
2. 接收结果

cursor执行命令的方法:

- execute(query, args):执行单条sql语句。query为sql语句本身, args为参数值的列表。执行后返回值为受影响的行数。
- executemany(query, args):执行单条sql语句,但是重复执行参数列表里的参数,返回值为受影响的行数

首先在数据库创建一个用于测试:

```
MariaDB [testpython]> create table member(id int(2) not null primary key auto_increment,username varchar(40),password varchar(20),email
#建表语句
CREATE TABLE `member` (
  `id` int(2) NOT NULL AUTO_INCREMENT,
  `username` varchar(40) DEFAULT NULL,
  `password` varchar(20) DEFAULT NULL,
  `email` varchar(200) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
```

插入一条数据,返回值为受影响的行数:

```
In [5]: cur.execute("insert into member (username,password,email) values (%s,%s,%s)","python","123456","python@gmail.com")
Out[5]: 1L
```

但是此时并没有完成表真正的提交:

```
MariaDB [testpython]> select * from member;
Empty set (0.00 sec)
```

提交事务:

```
In [7]: conn.commit()
```

数据成功被保存:

```
MariaDB [testpython]> select * from member;
```



```

+---+-----+-----+-----+
| id | username | password | email          |
+---+-----+-----+-----+
| 1  | python   | 123456   | python@gmail.com |
+---+-----+-----+-----+
1 row in set (0.00 sec)

```

插入多条数据:

```

>>> cur.executemany("insert into member (username,password,email) values (%s,%s,%s)",(("google","111222","g@gmail.com"),("facebook","222222","f@gmail.com"),("python","python","python@gmail.com")))
4L
>>> conn.commit()

```

日常工作中, 大多数会先拼接一个SQL语句 然后再执行, 实现插入数据的操作:

```

In [12]: sql = "INSERT INTO member (username,password,email) values ('%s','%s','%s')" % ('walter.liu','123456','zheshiz2@163.com')#注意其
In [13]: cur.execute(sql)
Out[13]: 1L

In [14]: conn.commit()

```

查询数据

查询数据也需要用游标来操作

```

In [4]: cur.execute('select * from member')
Out[4]: 2L #表示有两条数据被检索出来

```

使用游标一下方法, 才能取回检索到的数据:

- fetchall(self):接收全部的返回结果行.
- fetchmany(size=None):接收size条返回结果行. 如果size的值大于返回的结果行的数量,则会返回cursor.arraysize条数据.
- fetchone():返回一条结果行.
- scroll(value, mode='relative'):移动指针到某一行. 如果mode='relative',则表示从当前所在行移动value条. 如果mode='absolute',则表示从结果集的第一行移动value条

实例:

```

In [5]: lines = cur.fetchall()
#取回的数据被放在一个大的元祖里,
#此时游标已经移动到最后一条数据了
#如果再执行一次cur.fetchall()会返回空
In [6]: lines
Out[6]:
((1L, u'python', u'123456', u'python@gmail.com'),
 (2L, u'walter.liu', u'123456', u'zheshiz2@163.com'))

```

遍历取回的数据:

```

In [8]: for line in lines:
...:     print line
...:
(1L, u'python', u'123456', u'python@gmail.com')
(2L, u'walter.liu', u'123456', u'zheshiz2@163.com')

```

更新数据:

```

In [11]: cur.execute('UPDATE member SET username="%s" WHERE id=1' % ('php'))
Out[11]: 1L

In [12]: cur.execute('SELECT * FROM member WHERE id=1')
Out[12]: 1L

In [13]: cur.fetchone()
Out[13]: (1L, u'php', u'123456', u'python@gmail.com')
#此时在python里看到数据已经被更新, 但是在数据库中还没有被真正的更新
#需要执行commit提交事务才行
In [14]: conn.commit()#将数据真正的更新到数据库中

```

关于乱码问题:

可以做如下统一设置:

- Python文件设置编码 utf-8 (文件前面加上 #encoding=utf-8)

- MySQL数据库charset=utf8（数据库的设置方法，可以网上搜索）
- Python连接MySQL是加上参数 charset=utf8（在前面教程中都这么演示了，很重要）
- 设置Python的默认编码为 utf-8 (sys.setdefaultencoding(utf-8),)

Python程序：

```
#encoding=utf-8

import sys
import MySQLdb

reload(sys)
sys.setdefaultencoding('utf-8')

db=MySQLdb.connect(user='root',charset='utf8')
```

MySQL配置：

```
[client] default-character-set = utf8
[mysqld] default-character-set = utf8
```

flask入门

安装flask

```
[root@walter ~]# easy_install flask
#或者 pip install flask
```

定义一个简单的页面：app.py文件

```
__author__ = 'liuzhenwei'

#导入flask模块
from flask import Flask
#实例化一个application对象，将当前的模块名称传递进去
app = Flask(__name__)
#在开发的时候，一般将debug设置为true，这样的话，修改程序文件后，app会自动加载。
app.debug = True

#在flask中使用装饰器的方式，将请求连接映射到相应的函数
@app.route('/')
def hello():
    return 'hello world'
#hello 称之为一个视图函数

if __name__ == '__main__':
    app.run('0.0.0.0')
    #运行上边声明的app,默认值为空表示，监听127.0.0.1的5000端口

#在一个终端运行
[root@walter web]# python app.py
* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
* Restarting with stat
* Debugger is active!
* Debugger pin code: 134-058-099
```

浏览器打开 http://10.25.128.8:5000/。

接下来以一个简单的博客形式，来讲解后续的内容。

博客应有的基本功能：

- 浏览博客
- 发表博客

这里使用MySQL数据库存储博客信息。

库名为：blog

posts表，用来存储博客内容：

```
CREATE TABLE `posts` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `title` varchar(50) DEFAULT NULL,
  `content` text,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

发表一篇博客：

相应的视图函数：

这里需要一个MySQL的连接驱动，这个地方使用前面讲过的 MySQLdb模块。

在app.py文件中，新增一个视图函数

```
@app.route('/add',methods=['GET','POST'])
def add_entry():
    pass
#关于装饰器中的methods参数
#默认不指定的话，表示该连接只接受GET请求，
#这表示该连接接受GET和POST两个请求
#当用户以GET方式请求的时候，页面渲染一个form表单，用于发表博客
#当用户以POST方式请求的时候，表示用户要提交发表的博客内容
```

下面丰富add_entry函数的功能：

完整的代码为：

```
[root@walter web]# cat app.py
__author__ = 'liuzhenwei'

from flask import Flask,request
import MySQLdb

app = Flask(__name__)
app.debug = True
@app.route('/')
def hello():
    return 'hello world'

@app.route('/add',methods=['GET','POST'])
def add_entry():
    #根据request对象的method方法，获取当前请求的方式
    if request.method == 'GET':
        #以DOC的形式返回一个form表单到页面上。
        return '''
        <html>
            <head>
                <title>My Blog</title>
            </head>
            <body>
                <form action='' method='POST'>
                    Title:<input type='text' name='title'>
                    <br />
                    <br />
                    Content:<textarea name='content'></textarea>
                    <br />
                    <br />
                    <input type='submit' value='Submit'>
                </form>
            </body>
        </html>
        '''
    ...

if __name__ == '__main__':
    app.run('0.0.0.0')
```

浏览器：http://10.25.128.8:5000/add

到目前为止，页面也可以正常打开，但是此时我们是将html 代码直接写到python函数内的，这种方式相当的傻了。这接下来将其修改为模板的形式。

在当前目录下创建一个templates的目录。里面放置我们需要的HTML模板文件。

```
[root@walter web]# mkdir templates
[root@walter web]#
[root@walter web]# ll
total 4
-rw-r--r--. 1 root root 662 Apr 14 06:28 app.py
drwxr-xr-x. 2 root root  6 Apr 14 06:32 templates
```

在flask中是约定大于设置的，即flask会在application所在的目录下，找一个叫templates的目录，然后到此目录里找相应的模板文件。不需要在某个配置文件中对其进行设置。

完整的代码修改为：

```
__author__ = 'liuzhenwei'

#render_template用来加载模板文件
from flask import Flask,request,render_template
import MySQLdb

app = Flask(__name__)
app.debug = True
@app.route('/')
def hello():
    return 'hello world'

@app.route('/add',methods=['GET','POST'])
def add_entry():

    if request.method == 'GET':
        return render_template('add_entry.html')
        #render_template方法，接收一个模板文件名为参数
        #表示当前要加载的模板文件

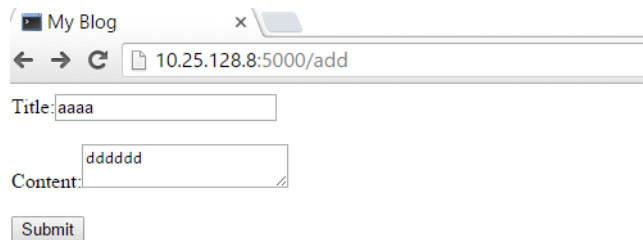
if __name__ == '__main__':
    app.run('0.0.0.0')
```

add_entry添加接收提交内容的功能：

```
@app.route('/add',methods=['GET','POST'])
def add_entry():

    if request.method == 'GET':
        return render_template('add_entry.html')
    if request.method == 'POST':
        title = request.form.get('title')
        content = request.form.get('content')
        return '%s,%s' % (title,content)
```

打开浏览器，提交内容测试，输入内容，提交，页面可以返回提交的内容表示正常。



My Blog

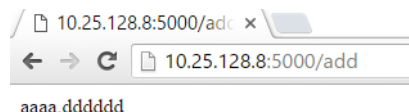
10.25.128.8:5000/add

Title: aaaa

Content: dddddd

Submit

点击提交后：



10.25.128.8:5000/add

aaaa,dddddd

到目前为止，程序可以正常接收我们从页面提交的内容了，接下来需要将内容保存到数据库中：

```
@app.route('/add',methods=['GET','POST'])
def add_entry():

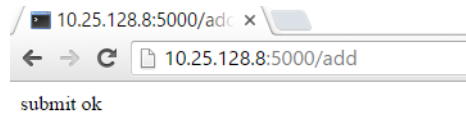
    if request.method == 'GET':
        return render_template('add_entry.html')
    if request.method == 'POST':
        title = request.form.get('title')
        content = request.form.get('content')
        #创建一个数据连接
        conn = MySQLdb.connect(host='localhost',user='root',passwd='',db='blog')
        #创建游标
```

```

cur = conn.cursor()
#拼接SQL语句
sql = "INSERT INTO posts (title,content) VALUES ('%s','%s')" %(title,content)
#执行SQL
cur.execute(sql)
#提交事务
conn.commit()
#关闭游标
cur.close()
#关闭连接
conn.close()
#在视图函数中,必须得return一个值,否则会报错
if r is not None:
    return 'submit ok'
else:
    return 'submit error'

```

提交测试:



查看数据库:

```

MariaDB [blog]> select * from posts;
+----+-----+-----+
| id | title | content |
+----+-----+-----+
| 1  | aaaa  | dddddd  |
+----+-----+-----+
1 row in set (0.00 sec)

```

到目前为止,可以正常发表一篇文章了,接下来需要完成显示文章的列表。新增index视图方法。

```

@app.route('/')
def index():
    conn = MySQLdb.connect(host='localhost',user='root',passwd='',db='blog')
    cur = conn.cursor()
    sql = "SELECT id,title FROM posts ORDER BY id DESC"
    try:
        cur.execute(sql)
        posts = cur.fetchall()
    except Exception,e:
        return e
    finally:
        cur.close()
        conn.close()
    return render_template('index.html',posts=posts)#渲染模板,并将数据传递到模板里

```

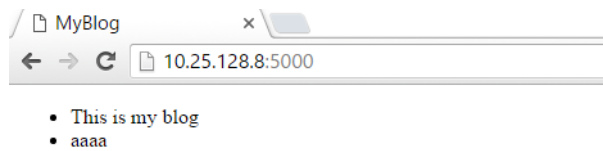
index.html内容

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset='utf-8'>
    <title>MyBlog</title>
  </head>
  <body>
    <ul>
      <!--jinja2模板语法-->
      {% for post in posts %}
      <li>{{ post[1] }}</li>
      {% endfor %}
    </ul>
  </body>
</html>

```

浏览器打开,访问首页:



显示博客详情页面:

```
@app.route('/detail')
def detail():
    #接收从URL以GET方式传递的值,需要用request.args
    p_id = request.args.get('id')
    conn = MySQLdb.connect(host='localhost',user='root',passwd='',db='blog')
    cur = conn.cursor()
    sql = "SELECT id,title,content FROM posts WHERE id=%d" % int(p_id)
    try:
        cur.execute(sql)
        post = cur.fetchone()
    except Exception,e:
        return e
    finally:
        cur.close()
        conn.close()
    return render_template('detail.html',post=post)
```

创建模板文件: detail.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset='utf-8'>
    <title>{{ post[1] }}</title>
  </head>
  <body>
    <h2>{{ post[1] }}</h2>
    <hr>
    <p>
      {{ post[2] }}
    </p>
  </body>
</html>
```

浏览器打开页面:



This is my blog

This is my blogThis is my blogThis is my blogThis is my blogThis is my blogThis is my blogThis is my blogThis is my blog

http://10.25.128.8:5000/detail?id=2, 这种方式是以query string的方式传递的参数, 看起来不是那么好看, 对搜索引擎是不是很友好。我们平时见到的好多连接基本上都是: http://10.25.128.8:5000/detail/2这种方式:

修改视图函数:

```
#<int:p_id>表示一个占位符, int表示此处变量的类型, p_id变量名, 占位符如果不加类型, 默认为str
@app.route('/detail/<int:p_id>')
#直接以参数的形式传递给视图函数
def detail(p_id):
    #p_id = request.args.get('id')#这一行不再需要
    conn = MySQLdb.connect(host='localhost',user='root',passwd='',db='blog')
    cur = conn.cursor()
    sql = "SELECT id,title,content FROM posts WHERE id=%d" % int(p_id)
    try:
        cur.execute(sql)
        post = cur.fetchone()
```

```
except Exception,e:
    return e

finally:
    cur.close()
    conn.close()

return render_template('detail.html',post=post)
```

浏览器打开

http://10.25.128.8:5000/detail/2



This is my blog

This is my blogThis is my blogThis is my blogThis is my blogThis is my blogThis is my blogThis is my blogThis is my blog

修改列表也，加上链接：

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset='utf-8'>
        <title>MyBlog</title>
    </head>
    <body>
        <ul>
            <!--jinja2模板语法-->
            {% for post in posts %}
            <li><a href='{ { url_for('detail',p_id=post[0]) } } '>
            <!--这里的url_for可以根据提供的视图函数，反向解除对应的URL链接。后边直接跟要传递的参数即可，这样的话，无论视图函数里面的
            {{ post[1] }}</a></li>
            {% endfor %}
        </ul>
    </body>
</html>
```

到目前为止，基本的博客功能是实现了，但是在视图函数中，每个函数都有一个数据库连接的操作，过于重复。修改的时候也很麻烦，不符合当代编程的规范。

在flask中有两个较为有用的视图函数：

@app.before_request:在request请求之前执行,在此处添加数据库连接操作

@app.teardown_request：在视图函数执行结束后，执行改段代码，在此处关闭数据库连接

```
@app.before_request
def connect_db():
    app.app_ctx_globals_class.conn = MySQLdb.connect(host='localhost',user='root',passwd='',db='blog')
    app.app_ctx_globals_class.cur = app.app_ctx_globals_class.conn.cursor()

@app.teardown_request
def close_db(*args):
    #该函数需要接受一个参数，为当前刚结束的请求的request
    if hasattr( app.app_ctx_globals_class,'cur'):
        app.app_ctx_globals_class.cur.close()
    if hasattr( app.app_ctx_globals_class,'conn'):
        app.app_ctx_globals_class.conn.close()
```

app.app_ctx_globals_class为flask app中一个全局的对象。添加了上边两个函数，其他视图函数中的数据库连接和数据库关闭操作都可以删除掉了。

到目前为止，所有的SQL语句都是硬编码写的，非常不友好。下边使用flask-SQLAlchemy ORM--对象关系映射

安装：

```
[root@walter web]# easy_install flask-SQLAlchemy
```

在app.py中引入模块：

```
from flask.ext.sqlalchemy import SQLAlchemy
```

使用ORM连接数据库：

```

SQLALCHEMY_DATABASE_URI="mysql+pymysql://root:@localhost:3306/blog"
#格式: mysql+pymysql://USERNAME:PASSWD@HOST:PORT/DBNAME

app.config.from_object(__name__)#从当前模块中导入数据库配置URI

db = SQLAlchemy(app)#实例化一个数据库对象

#定义一个model
class Post(db.Model):
    __tablename__ = 'posts'

    id = db.Column(db.Integer,primary_key=True,autoincrement=True)
    title = db.Column(db.String(45),unique=True,nullable=False)
    content = db.Column(db.Text,nullable=True)

#修改所有的视图函数，主要修改操作数据库的代码

app.route('/')
def index():
    posts = Post.query.all()
    return render_template('index.html',posts=posts)

@app.route('/detail/<int:p_id>')
def detail(p_id):
    post = Post.query.filter_by(id=p_id).first()
    return render_template('detail.html',post=post)

@app.route('/add',methods=['GET','POST'])
def add_entry():

    if request.method == 'GET':
        return render_template('add_entry.html')
    if request.method == 'POST':
        title = request.form.get('title')
        content = request.form.get('content')
        post = Post()
        post.title = title
        post.content = content
        db.session.add(post)
        try:
            db.session.commit()
        except Exception,e:
            db.session.rollback()
        return redirect(url_for('index'))
    #跳转到首页,from flask import redirect,url_for

```

相应的模板函数，也要修改，因为这次传递给模板的变量是一个对象：

```

[root@walter web]# cat templates/index.html
<!DOCTYPE html>
<html>
  <head>
    <meta charset='utf-8'>
    <title>MyBlog</title>
  </head>
  <body>
    <ul>
      <!--jinja2模板语法-->
      {% for post in posts %}
      <li><a href='{{ url_for('detail',p_id=post.id) }}'>{{ post.title }}</a></li>
      {% endfor %}
    </ul>
  </body>
</html>

[root@walter web]# cat templates/detail.html
<!DOCTYPE html>
<html>
  <head>
    <meta charset='utf-8'>
    <title>{{ post.title }}</title>
  </head>
  <body>
    <h2>{{ post.title }}</h2>
    <hr>
    <p>
      {{ post.content }}
    </p>

```



```
</body>
</html>
```

完整的代码：

```
[root@walter web]# cat app.py
__author__ = 'liuzhenwei'

from flask import Flask,request,render_template,redirect,url_for

from flask.ext.sqlalchemy import SQLAlchemy
import sys
app = Flask(__name__)
app.debug = True

SQLALCHEMY_DATABASE_URI="mysql://root:@localhost:3306/blog"

app.config.from_object(__name__)

db = SQLAlchemy(app)

class Post(db.Model):
    __tablename__ = 'posts'

    id = db.Column(db.Integer,primary_key=True,autoincrement=True)
    title = db.Column(db.String(45),unique=True,nullable=False)
    content = db.Column(db.Text,nullable=True)

@app.route('/')
def index():
    posts = Post.query.all()
    return render_template('index.html',posts=posts)

@app.route('/detail/<int:p_id>')
def detail(p_id):
    post = Post.query.filter_by(id=p_id).first()
    return render_template('detail.html',post=post)

@app.route('/add',methods=['GET','POST'])
def add_entry():

    if request.method == 'GET':
        return render_template('add_entry.html')
    if request.method == 'POST':
        title = request.form.get('title')
        content = request.form.get('content')
        post = Post()
        post.title = title
        post.content = content
        db.session.add(post)
        try:
            db.session.commit()
        except Exception,e:
            db.session.rollback()
        return redirect(url_for('index'))
if __name__ == '__main__':
    #第一次运行此程序的时候需要创建表，使用上边的db对象即可，可以根据定义好的每个model类，创建相应的表
    if len(sys.argv)>=2 and sys.argv[1]=='setup':
        db.create_all()
        sys.exit(0)
    #drop所有的表
    if len(sys.argv)>=2 and sys.argv[1]=='drop':
        db.drop_all()
        sys.exit(0)
    app.run('0.0.0.0')
```