课程介绍

市面上不乏python入门教程，但深入讲解python语言和应用的课程甚少，本课程定位深入理解python核心语法，结合实际应用场景，带领初学python或是有其它编程语言背景的程序员能快速掌握这门强大的语言，使其能在开发中发挥强大的生产力。

传智播客c++学院出品

```
http://c.itcast.cn/
```

深入理解python编程

最大的优势在于它的字符串模式匹配能力，其提供一个十分强大的正则表达式匹配引擎。核心实现依赖perl，但语法比perl易懂的多。 高级语言,面向对象,可拓展,可移植,语法清晰,易维护,高效的原型。

我为什么推崇Python?

1. 干某一件事情，C需要100行，JAVA需要50行，Python只需要10行，当你忙于编写代码或是设计框架时，Python程序员已经早早的下班开启了把妹之旅。
2. 面向对象开发，方便团队协作，语言间的万能胶水，当需要高性能的处理时可以自然粘合c/c++模块。
3. 信奉python的哲学

在python解释器中输入，"import this"

```
漂亮比丑陋要好。
直接比含蓄要好。
简单比繁复要好。
繁复比复杂要好。
平铺比嵌套要好。
稀疏比密集要好。
可读性很重要。
特例不能破坏规则。
```

```
尽管实用优于纯正。
错误永远不能安静的通过。
除非明确的让它安静。
拒绝在模糊的地方猜测。
应当有一种，并且最好只有一种，明显的方法去做一件事。
尽管开始时那种方法并不明显，除非你是荷兰人。
现在要比永远不更好。
尽管永远不常常比当前要好。
如果一个实现很难解释，那么它就是一个不好的想法。
如果一个实现容易解释，那么它可能是一个好的想法。
名称空间是一个很伟大的想法，让我们做的更多。
```

python家族:

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

python能干什么？

- 科学计算
- 图形化开发
- 系统脚本
- web服务器
- 网络爬虫
- 服务器集群自动化运维

一. 高效的开发环境与基础

python开发环境

```
Mac/Linux发行版目前默认安装python
ipython
python官方IDE，在python发行版自带
Eclipse+pydev
PyScripter
subline text3
```

Windows下Sublime Text3和python语言环境

sublimetext官方下载地址，请根据自己的操作系统平台选择对应版本

```
http://www.sublimetext.com/3
```

python下载地址

```
https://www.python.org/downloads/
```

Linux下Sublime Text3和python语言环境

Linux系统选择广受大家欢迎的Ubuntu14.04，如果没有此环境可以选择使用传智C++学院配置好的虚拟机镜像

```
下载链接：http://pan.baidu.com/s/1c0yTN4c          密码：b77w
用户名：itcast
密码：itcast
```

sublimetext官方下载地址，请根据自己的操作系统平台选择对应版本

```
http://www.sublimetext.com/3
```

ubuntu14.04默认是安装了python2.7的

Ubuntu 下 ipython

1. 安装

```
sudo apt-get install ipython
```

2. 启动

```
itcast@itcast:~/workspace/chuanzhi/openlessons$ ipython
```

3. 体验

```
print  "hello itcastcpp"          #此为python2的写法
```

4. 退出

```
exit
```

5. 案例

```
import requests
res=requests.get("http://c.itcast.cn")
savefile = open("itcast.html", "w")
savefile.write(res.content)
savefile.close()
```

6. 技巧

%history ：记录敲过的命令，方便从命令转为脚本文件

tab：补齐命令或路径

Sublime使用技巧

1.安装package管理工具

`ctrl+``调出命令输入窗口

```
import urllib.request,os; pf = 'Package Control.sublime-package'; ipp = sublime.installed_packages_path(); url
```

设置vim模式，Sublime Text 内置 Vim 模式支持，你只需到用户设置文件将 "ignored_packages": ["vintage"] 中的 vintage 删除即可。

2.调出installpackage界面

```
ctrl + shift + p
install package
```

3.常用工具包

```
AdvancedNewFile
Djaneiro
Emmet
Git
Side Bar
HTML/CSS/JS Prettify
Python PEP8 Autoformat
SublimeCodeIntel
```

```
ColorPicker
OmniMarkupPreviewer
```

4.常用包使用说明

AdvancedNewFile

可以创建文件，也可以连目录和文件都创建 win+alt+n

Djaneiro

django一些语法快速补齐功能，参考如下

```
https://packagecontrol.io/packages/Djaneiro
```

Emmet

快速缩写html,tab补齐

```
html:5    补齐html
p.foo     补齐class
p#foo     补齐id
>         子元素符号，表示嵌套的元素
+         同级标签符号
^         可以使该符号前的标签提升一行
```

更多参考：http://www.iteye.com/news/27580

Git

集成git

```
ctrl+shift+p
输入git
```

Side Bar

折叠目录树

```
ctrl+k   ctrl+b
```

HTML/CSS/JS Prettify

格式化代码,鼠标右键，从里面选

Python PEP8 Autoformat

格式化python代码

```
ctrl+shift+r
```

SublimeCodeIntel

自动匹配补全代码

```
ctrl+f3    调到变量定义的地方
```

ColorPicker

屏幕拾色器

```
ctrl+shift+c
```

OmniMarkupPreviewer

更多插件，设置OmniMarkupPreviewer的package setting中的default。修改里面的extensions

```
"extensions": ["extra", "codehilite", "toc", "strikeout", "smarty", "subscript", "superscript"]
```

安装语法高亮支持插件

```
sudo pip install pygments
```

ConvertToUTF8

直接在菜单栏中可以转，专为中文设计

Terminal

Sublime版的在当前文件夹内打开

```
ctrl+shift+t
```

SideBarEnhancements

```
右键一下子多处很多选择
```

自带技巧

- 修改同一个变量,光标放在变量后，两次 `ctrl+d`
- 多变量修改，按住ctrl，鼠标点击修改位置
- 查找 `ctrl+f`
- 插入注释 `ctrl+shift+/`
- 注释当前行 `ctrl+/`
- 分屏 Alt+Shift+1（非小键盘）窗口分屏，恢复默认1屏 Alt+Shift+2 左右分屏-2列 Alt+Shift+3 左右分屏-3列 Alt+Shift+4 左右分屏-4列 Alt+Shift+5 等分4屏 Alt+Shift+8 垂直分屏-2屏 Alt+Shift+9 垂直分屏-3屏
- 标签切换 `alt+数字`
- `Ctrl+Shift+P` 打开命令面板
- 关闭当前标签文件 `ctrl+f4`
- f11全屏

脚本一键安装

```
cd ~/home/xwp/.config/sublime-text-3/Packages

echo Install...
echo ================================================

echo === Package Control ===
rm -rf "Package Control"
git clone https://github.com/JustQyx/Sublime-Text-Package-Control.git "Package Control"

echo === Block Cursor Everwhere ===
rm -rf "Block Cursor Everwhere"
git clone https://github.com/ingshtrom/BlockCursorEverywhere.git "Block Cursor Everwhere"
...
```

基础数据类型

- 整型：通常被称为是整型或整数，是正或负整数，不带小数点。
- 长整型：无限大小的整数，整数最后是一个大写(或小写)的L。
- 浮点型：浮点型由整数部分与小数部分组成，浮点型也可以使用科学计数法表示（2.5e2 = 2.5 x 102 = 250）

- 复数：复数的虚部以字母J 或 j结尾 。如：2+3j
- 布尔类型：True , False
- 字符串：单引号，双引号，三个单引号扩起来

获取变量的数据类型 type(var_name)

```
主提示符        >>>     在等待下一条语句
次提示符        ...  在等待当前语句的其他部分
```

变量

变量本质

1. python中的变量不需要先定义，再使用，可以直接使用,还有重新使用用以存储不同类型的值。
2. 变量命名遵循C命名风格。
3. 大小写敏感。
4. 变量引用计数。
5. del语句可以直接释放资源，变量名删除，引用计数减1。
6. 变量内存自动管理回收,垃圾收集。
7. 指定编码在文件开头加入 *# -- coding: UTF-8 --* 或者 #coding=utf-8。

```
>>> a = 12                #无需定义，直接使用，python解释器根据右值决定左侧类型
>>> print a
12
>>> id(a)                 #变量a在内存中的编号
136776784
>>> type(a)               #a的类型为int类型
<class 'int'>
>>> b = 12.34
>>> print b
12.34
>>> id(b)                 #变量b在内存中所占内存编号
3071447616
>>> type(b)
<class 'float'>           #b的类型为float
>>> a = "itcast"          #变量a从新指向一个字符串
>>> print a
itcast
>>> id(a)                 #变量a在内存中的编号为保存“itcast”地方，原来a所指向的内存编号里内容并没有立即释放
3071127936
>>> type(a)               #变量a现在指向一个字符串
<class 'str'>
>>> c = b
>>> print c
12.34
>>> id(c)                 #变量c保存的内存中的编号和b一致
3071447616
>>> type(c)
<class 'float'>

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

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

简单函数

函数定义格式

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

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

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

输入输出函数

输入和raw_input()内置函数

raw_input() 从标准输入获取数据 返回的数据是字符串类型

需要使用int()进行转换

```
输出
print函数
>>> print "%s is %dsthello world "%("tody",1)
tody is 1sthello world
>> 重定向操作符
logfile = open('/tmp/mylog.txt','a')
print >>logfile,'error'
logfile.close()
```

局部变量和全局变量

代码1. 局部变量作用域覆盖全局变量

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

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

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

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

num=10
p_num()
print num
# 结果出错
```

代码3. 函数内部可以直接访问全局变量

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

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

代码4. 函数内修改全局变量,使用global关键字

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

num=10
p_num()
print num
```

特殊变量

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

表达式

算术表达式

```
+a        结果符号不变
-a        对结果符号取负
a +  b    a加b
a -  b    a减b
a ** b    a的b次幂
a *  b    a乘以b
a /  b    a除以b，真正除，浮点数保留小数
a // b    a除以b，向下取整
a %  b    a对b取余数
```

逻辑表达式

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

关系表达式

运算结果为布尔类型

```
==  等于
!=  不等于
<>  不等于(废弃)
>   大于
<   小于
>=  大于等于
<=  小于等于
```

位运算

```
~a      按位取反
```

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

语法格式

缩进表示关系，函数，分支，循环语句后面带':'

分支语句

```
#if-else

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

#if-elif-else

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

循环语句

python里的控制语句和循环语句和C中的非常相似，毕竟python是用C实现的。 注意语句后面的':'不要丢掉，这一点C/C++程序员刚上手python的时候是最容易犯的错误。

```
        while 判断条件：
            执行语句
```

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

    print "Good bye!"
```

Python for循环可以遍历任何序列的项目，如一个列表或者一个字符串。

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

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

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

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

```
    count = 0
    while count < 5:
       print(count, " is  less than 5")
       count = count + 1
```

```
else:
   print(count, " is not less than 5")
```

以上实例输出结果为：

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

break

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

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

break语句用在while和for循环中。

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

continue

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

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

continue语句用在while和for循环中。

list列表

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

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

访问列表中的值

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

更新列表中的值

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

删除列表中的值

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

Python列表脚本操作符

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

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

Python列表截取

Python的列表截取与字符串操作类型，如下所示：

```
L = ['spam', 'Spam', 'SPAM!']
操作：

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

Python列表函数&方法

```
Python包含以下函数:

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

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

内建range函数

```
range(start,end,step)
>>> range(1,5) #代表从1到5(不包含5)
[1, 2, 3, 4]
>>> range(1,5,2) #代表从1到5，间隔2(不包含5)
[1, 3]
>>> range(5) #代表从0到5(不包含5)
[0, 1, 2, 3, 4]
>>>for eachNum in [0,1,2]:
... print eachNum
>>>for eachNum in range(3):
... print eachNum
>>>mystr = 'abc'
>>>for c in mystr:
... print c
range()函数还经常和len()函数一起用于字符串索引
```

元组Tuple

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

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

访问元组

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

修改元组

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

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

删除元素

元组中的元素值是不允许删除的，但我们可以使用del语句来删除整个元组

元组运算符

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

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

元组索引，截取

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

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

无关闭分隔符

任意无符号的对象，以逗号隔开，默认为元组，如下实例：

```
print 'abc', -4.24e93, 18+6.6j, 'xyz';
x, y = 1, 2;
print "Value of x , y : ", x,y;
```

以上实例允许结果：

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

元组内置函数

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

多维元祖访问的示例

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

字典Dictionary

字典是python中的映射数据类型,,由键值(key-value)构成,类似于关联数组或哈希表. key一般以数字和字符串居多,值则可以是任意类型的python对象，如其他容器模型。 字典元素用大括号 {} 包括.比如:

```
>>>dict = {'Alice': '2341', 'Beth': '9102', 'Cecil': '3258'}
>>>aDict = {'host':'noname'}
```

每个键与值用冒号隔开（:），每对用逗号分割，整体放在花括号中（{}）。 键key必须独一无二，但值则不必。值value可以取任何数据类型，但必须是不可变的

访问字典里的值

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

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

修改字典

向字典添加新内容的方法是增加新的键/值对，修改或删除已有键/值对.

```
>>>aDict['port'] = 80    #如果有port key,值将会被更新 否则会被新建一个port key
>>>aDict
    {'host':'noname','port':80}
```

删除字典元素

能删单一的元素也能清空字典，清空只需一项操作.

显示删除一个字典用del命令

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};
del dict['Name']; # 删除键是'Name'的条目
dict.clear();     # 清空词典所有条目
del dict ;        # 删除词典
```

字典键(key)的特性

字典值可以没有限制地取任何python对象，既可以是标准的对象，也可以是用户定义的，但键不行。

两个重要的点需要记住：

1)不允许同一个键出现两次。创建时如果同一个键被赋值两次，后一个值会被记住

```
>>>dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'};
>>>print "dict['Name']: ", dict['Name'];
>>>dict['Name']:  Manni
```

2)键必须不可变，所以可以用数，字符串或元组充当，所以用列表就不行

```
>>>dict = {['Name']: 'Zara', 'Age': 7};
>>>print "dict['Name']: ", dict['Name'];
    raceback (most recent call last):
    File "test.py", line 3, in <module>
>>>dict = {['Name']: 'Zara', 'Age': 7};
    TypeError: list objects are unhashable
```

字典内置函数&方法

Python字典包含了以下内置函数：

```
序号  函数及描述
1   cmp(dict1, dict2)       比较两个字典元素。
2   len(dict)               计算字典元素个数，即键的总数。
3   str(dict)               输出字典可打印的字符串表示。
4   type(variable)          返回输入的变量类型，如果变量是字典就返回字典类型。
Python字典包含了以下内置函数：

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

Python数字类型转换

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

Python数学函数

```
函数      返回值 ( 描述 )
abs(x)      返回数字的绝对值，如abs(-10) 返回 10
ceil(x)     返回数字的上入整数，如math.ceil(4.1) 返回 5
cmp(x, y)   如果 x < y 返回 -1, 如果 x == y 返回 0, 如果 x > y 返回 1
exp(x)      返回e的x次幂(ex),如math.exp(1) 返回2.718281828459045
fabs(x)     返回数字的绝对值，如math.fabs(-10) 返回10.0
floor(x)        返回数字的下舍整数，如math.floor(4.9)返回 4
log(x)      如math.log(math.e)返回1.0,math.log(100,10)返回2.0
log10(x)        返回以10为基数的x的对数，如math.log10(100)返回 2.0
max(x1, x2,...) 返回给定参数的最大值，参数可以为序列。
min(x1, x2,...) 返回给定参数的最小值，参数可以为序列。
modf(x)     返回x的整数部分与小数部分，两部分的数值符号与x相同，整数部分以浮点型表示。
pow(x, y)   x**y 运算后的值。
round(x [,n])   返回浮点数x的四舍五入值，如给出n值，则代表舍入到小数点后的位数。
sqrt(x)     返回数字x的平方根，数字可以为负数，返回类型为实数，如math.sqrt(4)返回 2+0j
```

Python随机数函数

随机数可以用于数学，游戏，安全等领域中，还经常被嵌入到算法中，用以提高算法效率，并提高程序的安全性。

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

二. 字符串处理与特殊函数

用引号括起来的字符集合称之为字符串.引号可以是一对单引号,双引号,三引号(单/双).

备注：python中三引号可以将复杂的字符串进行复制: python三引号允许一个字符串跨多行，字符串中可以包含换行符、制表符以及其他特殊字符。 三引号的语法是一对连续的单引号或者双引号（通常都是成对的用）。

创建字符串很简单，只要为变量分配一个值即可。例如：

```
var1 = 'Hello World!'
var2 = "Python Programming"
var3 = '''hello chuanzhiboke'''
var4 = """hello chuanzhiboke"""
```

但是如果需要输出 Hello "dear"怎么办呢?

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

Python访问字符串中的值

Python不支持单字符类型，单字符也在Python也是作为一个字符串使用。

Python访问子字符串，可以使用方括号来截取字符串——即分片操作

```
var1 = 'Hello World!'
var2 = "Python Programming"
print "var1[0]: ", var1[0]
print "var2[1:5]: ", var2[1:5]
以上实例执行结果：
var1[0]:  H
var2[1:5]:  ytho
```

Python字符串更新

你可以对已存在的字符串进行修改，并赋值给另一个变量.

```
var1 = 'Hello World!'
print "Updated mystr :- ", var1[:6] + 'Python'
Updated mystr :-  Hello Python
```

Python字符串运算符

下表实例变量a值为字符串"Hello"，b变量值为"Python"：

```
操作符 描述           实例
+   字符串连接            a + b 输出结果： HelloPython
*   重复输出字符串      a*2 输出结果：HelloHello
[]  通过索引获取字符         a[1] 输出结果 e
[:]  截取字符串中的一部分      a[1:4] 输出结果 ell
in  成员运算符            如果字符串中包含给定的字符返回 True
not in  成员运算符            如果字符串中不包含给定的字符返回 True
r/R 原始字符串            字符串直接按照字面的意思来使用，没有转义或不能打印的字符
```

原始字符串除在字符串的第一个引号前加上字母"r"（可以大小写）以外，与普通字符串有着几乎完全相同的语法。

```
print r'hello\n'
```

Python字符串格式化

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

字符串各种函数

以下start和end可以缺省参数

```
mystr = 'hello world itcast and itcastcpp'
```

检测 str 是否包含在 mystr中，如果是返回开始的索引值，否则返回-1

```
mystr.find(str, start=0, end=len(mystr))
```

跟find()方法一样，只不过如果str不在 mystr中会报一个异常.

```
mystr.index(str, start=0, end=len(mystr))
```

返回 str在start和end之间 在 mystr里面出现的次数

```
mystr.count(str, start=0, end=len(mystr))
```

以 encoding 指定的编码格式解码 mystr，如果出错默认报一个 ValueError 的 异 常 ， 除非 errors 指定的是 'ignore' 或 者'replace'

```
mystr.decode(encoding='UTF-8', errors='strict')
```

以 encoding 指定的编码格式编码 mystr，如果出错默认报一个ValueError 的异常，除非 errors 指定的是'ignore'或者'replace'

```
mystr.encode(encoding='UTF-8', errors='strict')
```

把 mystr 中的 str1 替换成 str2,如果 count 指定，则替换不超过 count 次.

```
mystr.replace(str1, str2,  mystr.count(str1))
```

以 str 为分隔符切片 mystr，如果 maxsplit有指定值，则仅分隔 maxsplit 个子字符串

```
mystr.split(str=" ", 2)
```

把字符串的第一个字符大写

```
mystr.capitalize()
```

返回一个原字符串居中,并使用空格填充至长度 width 的新字符串

```
mystr.center(width)
```

检查字符串(start,end)是否以obj结束，如果是返回True,否则返回 False.

```
mystr.endswith(obj, 0, len(mystr))
```

检查字符串是否是以 obj 开头, 是则返回 True，否则返回 False

```
mystr.startswith()
```

把字符串 mystr 中的 tab 符号转为空格，默认的空格数 tabsize 是 8.

```
mystr.expandtabs(tabsize=8)
```

如果 mystr 至少有一个数字并且所有字符都是字母或数字则返回 True,否则返回 False

```
mystr.isalnum()
```

如果 mystr 至少有一个字符并且所有字符都是字母(不含数字和空格)则返回 True,否则返回 False

```
mystr.isalpha()
```

如果 mystr 只包含数字则返回 True 否则返回 False.

```
mystr.isdigit()
```

如果 mystr 中只包含空格，则返回 True，否则返回 False.

```
mystr.isspace()
```

如果 mystr 是标题化的(见 title())则返回 True，否则返回 False

```
mystr.istitle()
```

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

```
mystr.isupper()
```

转换 mystr 中所有大写字符为小写.

```
mystr.lower()
```

转换 mystr 中的小写字母为大写

```
mystr.upper()
```

mystr 中每个字符后面插入str,构造出一个新的字符串

```
mystr.join(str)
```

返回一个原字符串左对齐,并使用空格填充至长度 width 的新字符串

```
mystr.ljust(width)
```

返回一个原字符串右对齐,并使用空格填充至长度 width 的新字符串

```
mystr.rjust(width)
```

截掉 mystr 左边的空格

```
mystr.lstrip()
```

删除 mystr 字符串末尾的空格.

```
mystr.rstrip()
```

类似于 find()函数，不过是从右边开始查找.

```
mystr.rfind(str, start=0,end=len(mystr) )
```

类似于 index()，不过是从右边开始.

```
mystr.rindex( str, start=0,end=len(mystr))
```

把mystr以str分割成三部分,str前，str和str后

```
mystr.partition(str)
```

类似于 partition()函数,不过是从右边开始.

```
mystr.rpartition(str)
```

按照行分隔，返回一个包含各行作为元素的列表

```
mystr.splitlines()
```

返回长度为 width 的字符串，原字符串 mystr 右对齐，前面填充0

```
mystr.zfill(width)
```

检查字符串是否只包含十进制字符。这种方法只存在于unicode对象。

```
mystr.isdecimal()
```

时间和日期

Python程序能用很多方式处理日期和时间。转换日期格式是一个常见的例行琐事。

Python有一个time and calendar模组可以帮忙.

什么是Tick？

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

什么是时间元组

```
序号  属性  值
0   tm_year 2008
1   tm_mon  1 到 12
2   tm_mday 1 到 31
3   tm_hour 0 到 23
4   tm_min  0 到 59
5   tm_sec  0 到 61 (60或61 是闰秒)
6   tm_wday 0到6 (0是周一)
7   tm_yday 1 到 366(儒略历)
8   tm_isdst    -1, 0, 1, -1是决定是否为夏令时的旗帜
```

获取当前时间

从返回浮点数的时间辍方式向时间元组转换，只要将浮点数传递给如localtime之类的函数。

```
>>>import time;
>>>localtime = time.localtime(time.time())
>>>print "Local current time :", localtime

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

获取格式化的时间

你可以根据需求选取各种格式，但是最简单的获取可读的时间模式的函数是asctime():

```
localtime = time.asctime( time.localtime(time.time()) )
print "Local current time :", localtime
Local current time : Tue Jan 13 10:17:09 2009
```

获取某月日历

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

Time模块的内置函数

```
1   time.asctime([tupletime])   接受时间元组并返回一个可读的形式为"Tue Dec 11 18:07:14 2008"（2008年12月11日
                                周二  18时07分14秒）的24个字符的字符串。
2   time.clock( )
                                用以浮点数计算的秒数返回当前的CPU时间。用来衡量不同程序的耗时，比time.time()更有用。
3   time.sleep(secs)            推迟调用线程的运行，secs指秒数。
4   time.time( )                返回当前时间的时间戳（1970纪元后经过的浮点秒数）
Time模块包含了以下2个非常重要的属性：
序号  属性及描述
1   time.timezone
    属性time.timezone是当地时区（未启动夏令时）距离格林威治的偏移秒数（>0，美洲;<=0大部分欧洲，亚洲，非洲）。
2   time.tzname
    属性time.tzname包含一对根据情况的不同而不同的字符串，分别是带夏令时的本地时区名称，和不带的。
```

日历（Calendar）模块

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

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

函数高级

定义一个函数

你可以定义一个由自己想要功能的函数，以下是简单的规则：

函数代码块以def关键词开头，后接函数标识符名称和圆括号()。 任何传入参数和自变量必须放在圆括号中间。圆括号之间可以用于定义参数。 函数的第一行语句可以选择性地使用文档字符串—用于存放函数说明。 函数内容以冒号起始，并且缩进。

Return[expression]结束函数，选择性地返回一个值给调用方。不带表达式的return相当于返回 None

语法

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

默认情况下，参数值和参数名称是按函数声明中定义的的顺序匹配起来的。

实例

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

```
def printme( str ):
    "打印传入的字符串到标准显示设备上"
    print str
    return
```

函数调用

定义一个函数只给了函数一个名称，指定了函数里包含的参数，和代码块结构。
这个函数的基本结构完成以后，你可以通过另一个函数调用执行，也可以直接从Python提示符执行。

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

以上实例输出结果：

```
我要调用用户自定义函数!
再次调用同一函数
```

按值传递参数和按引用传递参数

```
1.按值传递,单个变量
2.按引用传递
```

如果你在函数里修改了参数，那么在调用这个函数的函数里，原始的参数也被改变了。例如：

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

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

传入函数的和在末尾添加新内容的对象用的是同一个引用。故输出结果如下：

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

参数

以下是调用函数时可使用的正式参数类型：

必备参数
命名参数
缺省参数
不定长参数

必备参数

必备参数须以正确的顺序传入函数。调用时的数量必须和声明时的一样。
调用printme()函数，你必须传入一个参数，不然会出现语法错误

命名参数

命名参数和函数调用关系紧密，调用方用参数的命名确定传入的参数值。你可以跳过不传的参数或者乱序传参，
因为Python解释器能够用参数名匹配参数值。用命名参数调用printme()函数：

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

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

缺省参数

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

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

#调用printinfo函数
printinfo( age=50, name="miki" );
printinfo( age = 9,name="miki" );
```

以上实例输出结果：

```
Name:  miki
Age  50
Name:  miki
Age  35
```

不定长参数

你可能需要一个函数能处理比当初声明时更多的参数。这些参数叫做不定长参数，声明时不会命名。
基本语法如下：

```
def functionname([formal_args,] *var_args_tuple ):
   "函数_文档字符串"
   function_suite
   return [expression]
```

加了星号（*）的变量名会存放所有未命名的变量参数。选择不多传参数也可。如下实例：

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

# 调用printinfo 函数
printinfo( 10 );
printinfo( 70, 60, 50 );
```

以上实例输出结果：

```
输出:
10
输出:
70
60
50
```

匿名函数

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

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

lambda函数的语法只包含一个语句，如下：

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

如下实例：

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

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

以上实例输出结果：

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

return语句

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

```
def sum( arg1, arg2 ):
# 返回2个参数的和."
total = arg1 + arg2
print "Inside the function : ", total
return total;

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

变量作用域

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

- 全局变量
- 局部变量

闭包

内部函数对外部函数作用域里变量的引用（非全局变量），则称内部函数为闭包。

```
# closure.py

def counter(start=0):
    count=[start]
    def incr():
        count[0] += 1
        return count[0]
    return incr

启动python解释器
>>>import closeure
>>>c1=closeure.counter(5)
>>>print c1()
6
>>>print c1()
7
>>>c2=closeure.counter(100)
>>>print c2()
101
>>>print c2()
```

```
102
```

闭包思考： 1.闭包似有化了变量，原来需要类对象完成的工作，闭包也可以完成 2.由于闭包引用了外部函数的局部变量，则外部函数的局部变量没有及时释放，消耗内存

装饰器函数

装饰器里引入通用功能处理

1.引入日志

2.函数执行时间统计

3.执行函数前预备处理

4.执行函数后清理功能

例1:无参数的函数

```
#decorator.py
from time import ctime, sleep

def timefun(func):
    def wrappedfunc():
        print "%s called at %s"%(func.__name__, ctime())
        return func()
    return wrappedfunc

@timefun
def foo():
    pass

foo()
sleep(2)
foo()
```

例2:被装饰的函数有参数

```
#decorator2.py
from time import ctime, sleep

def timefun(func):
    def wrappedfunc(a, b):
        print "%s called at %s"%(func.__name__, ctime())
        print a, b
        return func(a, b)
    return wrappedfunc

@timefun
def foo(a, b):
    print a+b

foo(3,5)
sleep(2)
foo(2,4)
```

例3:装饰器带参数,在原有装饰器的基础上，设置外部变量

```
#decorator2.py
from time import ctime, sleep

def timefun_arg(pre="hello"):
    def timefun(func):
        def wrappedfunc():
            print "%s called at %s %s"%(func.__name__, ctime(), pre)
            return func()
        return wrappedfunc
    return timefun
```

```
@timefun_arg("itcast")
def foo():
    pass

@timefun_arg("xwp")
def too():
    pass

foo()
sleep(2)
foo()

too()
sleep(2)
too()
```

例4:装饰器和闭包混用(难)

```
#coding=utf-8

from time import time

def logged(when):
    def log(f, *args, **kargs):
        print "fun:%s  args:%r  kargs:%r" %(f, args, kargs)
        #%r字符串的同时，显示原有对象类型

    def pre_logged(f):
        def wrapper(*args, **kargs):
            log(f, *args, **kargs)
            return f(*args, **kargs)
        return wrapper

    def post_logged(f):
        def wrapper(*args, **kargs):
            now=time()
            try:
                return f(*args, **kargs)
            finally:
                log(f, *args, **kargs)
                print "time delta: %s"%(time()-now)
        return wrapper
    try:
        return {"pre":pre_logged, "post":post_logged}[when]
    except KeyError, e:
        raise ValueError(e), 'must be "pre" or "post"'

@logged("post")
def fun(name):
    print "Hello, ", name

fun("itcastcpp!")
```

生成器函数

生成器是这样一个函数，它记住上一次返回时在函数体中的位置。对生成器函数的第二次（或第 n 次）调用跳转至该函数中间，而上次调用的所有局部变量都保持不变。

生成器不仅“记住”了它数据状态；生成器还“记住”了它在流控制构造（在命令式编程中，这种构造不只是数据值）中的位置。 生成器的特点：

```
1.生成器是一个函数，而且函数的参数都会保留。
2.迭代到下一次的调用时，所使用的参数都是第一次所保留下的，即是说，在整个所有函数调用的参数都是第一次所调用时保留的，而不是新创建的
3.节约内存
```

例子：执行到yield时，gen函数作用暂时保存，返回x的值;tmp接收下次c.send("python")，send发送过来的值，c.next()等价c.send(None)

```
#generation.py
def gen():
    for x in xrange(4):
        tmp = yield x
        if tmp == "hello":
            print "world"
        else:
            print "itcastcpp ", str(tmp)

>>>from generation import gen
>>>c=gen()
>>>c.next()
0
>>>c.next()
itcastcpp None
1
>>>c.send("python")
itcastcpp python
2
```

三. 面向对象

面向对象技术简介

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

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

创建类

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

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

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

举例：

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

在Python类中定义的方法通常有三种：实例方法，类方法以及静态方法。

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

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

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

创建实例对象

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

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

访问属性

使用点(.)来访问对象的属性。使用如下类的名称访问类变量:
```
emp1.displayEmployee()
```

你可以添加，删除，修改类的属性，如下所示：
```
emp1.age = 7  # 添加一个 'age' 属性
emp1.age = 8  # 修改 'age' 属性
del emp1.age  # 删除 'age' 属性
```

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

Python内置类属性

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

python对象销毁(垃圾回收)

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

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

类的继承

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

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

在python中继承中的一些特点：

1：在继承中基类的构造（__init__()方法）不会被自动调用，它需要在其派生类的构造中亲自专门调用。
2：在调用基类的方法时，需要加上基类的类名前缀，且需要带上self参数变量。

区别于在类中调用普通函数时并不需要带上self参数

3：Python总是首先查找对应类型的方法，如果它不能在派生类中找到对应的方法，它才开始到基类中逐个查找。（先在本类中查找调用的方法，找不到才去基类中找）。

如果在继承元组中列了一个以上的类，那么它就被称作"多重继承" 。

语法：

派生类的声明，与他们的父类类似，继承的基类列表跟在类名之后，如下所示：

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

实例：

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

class Parent:        # 定义父类
   parentAttr = 100
   def __init__(self):
      print "调用父类构造函数"

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

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

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

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

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

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

以上代码执行结果如下：

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

你可以继承多个类

```
class A:        # 定义类 A
.....

class B:         # 定义类 B
.....

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

你可以使用issubclass()或者isinstance()方法来检测。

issubclass() - 布尔函数判断一个类是另一个类的子类或者子孙类，语法：issubclass(sub,sup)

isinstance(obj, Class) 布尔函数如果obj是Class类的实例对象或者是一个Class子类的实例对象则返回true。

方法重写

如果你的父类方法的功能不能满足你的需求，你可以在子类重写你父类的方法：

实例：

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

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

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

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

执行以上代码输出结果如下：

```
调用子类方法
```

基础重载方法

下表列出了一些通用的功能，你可以在自己的类重写：

```
序号  方法, 描述 & 简单的调用
1   __init__ ( self [,args...] )
```

构造函数

```
简单的调用方法: obj = className(args)
2   __del__( self )
```

析构方法, 删除一个对象

```
简单的调用方法 : dell obj
3   __repr__( self )
```

转化为供解释器读取的形式

```
简单的调用方法 : repr(obj)
4   __str__( self )
```

用于将值转化为适于人阅读的形式

```
简单的调用方法 : str(obj)
5   __cmp__ ( self, x )
```

对象比较

```
简单的调用方法 : cmp(obj, x)
```

运算符重载

Python同样支持运算符重载，实例如下：

```
#!/usr/bin/python

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

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

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

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

以上代码执行结果如下所示:

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

类属性与方法

类的私有属性

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

类的方法

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

类的私有方法

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

实例

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

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

    def count(self):
```

```
        self.__secretCount += 1
        self.publicCount += 1
        print self.__secretCount

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

Python 通过改变名称来包含类名:

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

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

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

执行以上代码，执行结果如下：

```
1
2
2
2
```

课堂例题：

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

class Student(object):
    '''2015 new student'''
    grade=2015
    __school="qinghua"

    def __init__(self, subj, name, age, sex):
        '''this is create fun'''
        self.subj=subj
        self.name=name
        self.age=age
        self.sex=sex
        print "init student"

    def setName(self, newname):
        self.name=newname

    def getName(self):
        return self.name

    def showStudent(self):
        print "subj=", self.subj
        print "name=", self.name
        print "age=", self.age
        print "sex=", self.sex
        print "grade=", Student.grade
        print "school=", Student.__school

    @classmethod
    def updategrade(cls, newgrade):
        cls.grade = newgrade

    @classmethod
    def showClass(cls):
        print "__name__=" ,cls.__name__
        print "__dict__=" ,cls.__dict__
        print "__class__=" ,cls.__class__
```

模块

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

import 语句

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

From…import 语句

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

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

From…import* 语句

把一个模块的所有内容全都导入到当前的命名空间也是可行的，只需使用如下声明：

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

定位模块

当你导入一个模块，Python解析器对模块位置的搜索顺序是：

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

PYTHONPATH变量

作为环境变量，PYTHONPATH由装在一个列表里的许多目录组成。PYTHONPATH的语法和shell变量PATH的一样。
在Windows系统，典型的PYTHONPATH如下：
```
    set PYTHONPATH=c:\python20\lib;
```
在UNIX系统，典型的PYTHONPATH如下：
```
    set PYTHONPATH=/usr/local/lib/python
```

命名空间和作用域

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

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

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

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

```
Money = 2000
def AddMoney():
   # 想改正代码就取消以下注释:
   # global Money
   Money = Money + 1
print Money
AddMoney()
print Money
```

dir()函数

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

```
import math
content = dir(math)
print content;
```

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

globals()和locals()函数

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

Python中的包

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

考虑一个在Phone目录下的pots.py文件。这个文件有如下源代码： #pots.py #coding=utf-8 #!/usr/bin/python

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

同样地，我们有另外两个保存了不同函数的文件：

```
Phone/Isdn.py 含有函数Isdn()
Phone/G3.py 含有函数G3()
```

现在，在Phone目录下创建file **init**.py：

```
Phone/__init__.py
此文件可为空
from Phone.post import post
```

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

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

```
from G3 import G3
```

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

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

Phone.Pots()
Phone.Isdn()
Phone.G3()
以上实例输出结果：

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

如上，为了举例，我们只在每个文件里放置了一个函数，但其实你可以放置许多函数。

你也可以在这些文件里定义Python的类，然后为这些类建一个包。

模块发布

1.mymodule目录结构体如下：

```
.
├── setup.py
├── suba
│   ├── aa.py
│   ├── bb.py
│   └── __init__.py
└── subb
    ├── cc.py
    ├── dd.py
    └── __init__.py
```

2.编写setup.py,py_modules需指明所需包含的py文件

```
from distutils.core import setup

setup(name="xwp", version="1.0", description="xwp's module", author="xingwenpeng", py_modules=['suba.aa', 'sub
```

3.构建模块

```
python setup.py build

构建后目录结构
.
├── build
│   └── lib.linux-i686-2.7
│       ├── suba
│       │   ├── aa.py
│       │   ├── bb.py
│       │   └── __init__.py
│       └── subb
│           ├── cc.py
│           ├── dd.py
│           └── __init__.py
├── setup.py
├── suba
│   ├── aa.py
│   ├── bb.py
│   └── __init__.py
└── subb
    ├── cc.py
    ├── dd.py
    └── __init__.py
```

4.生成发布压缩包

```
python setup.py sdist

打包后,生成最终发布压缩包xwp-1.0.tar.gz , 目录结构
.
├── build
│   └── lib.linux-i686-2.7
│       ├── suba
│       │   ├── aa.py
│       │   ├── bb.py
│       │   └── __init__.py
│       └── subb
│           ├── cc.py
│           ├── dd.py
│           └── __init__.py
├── dist
│   └── xwp-1.0.tar.gz
├── MANIFEST
├── setup.py
├── suba
│   ├── aa.py
│   ├── bb.py
│   └── __init__.py
└── subb
    ├── cc.py
    ├── dd.py
    └── __init__.py
```

四. 文件操作

打印到屏幕

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

读取键盘输入

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

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

```
str = raw_input("Enter your input: ");
print "Received input is : ", str
```
这将提示你输入任意字符串，然后在屏幕上显示相同的字符串。当我输入"Hello Python！"，它的输出如下：

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

input函数

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

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

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

打开和关闭文件

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

File对象的属性

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

close()方法

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

```
fileObject.close();
```

write()方法

Write()方法可将任何字符串写入一个打开的文件。需要重点注意的是，Python字符串可以是二进制数据，而不是仅仅是文字。
Write()方法不在字符串的结尾不添加换行符('\n')：
语法：
fileObject.write(string);

read()方法

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

```
print file.read(10);
```

seek()方法

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

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

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

重命名和删除文件

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

rename()方法：
rename()方法需要两个参数，当前的文件名和新文件名。
语法：

```
    os.rename(current_file_name, new_file_name)
```

remove()方法
你可以用remove()方法删除文件，需要提供要删除的文件名作为参数。
语法：

```
    os.remove(file_name)
```

Python里的目录：

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

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

chdir()方法

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

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

getcwd()方法：

getcwd()方法显示当前的工作目录。
语法：

```
    os.getcwd()
```

rmdir()方法

rmdir()方法删除目录，目录名称以参数传递。

在删除这个目录之前，它的所有内容应该先被清除。

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

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

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

文件、目录相关的方法

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

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

Linux和Windows平台下的差异性

五. 应用案例剖析

经典Python案例，展现python优美神奇的一面

1.图片上加数字

```
#!/usr/bin/env python
#coding: utf-8

myPath = "./"
fontPath = "./"
inputFile = "xwp.jpg"
outputFile = "output.jpg"

import Image, ImageFont, ImageDraw
#打开图片
im = Image.open(myPath + inputFile)
draw = ImageDraw.Draw(im)
#根据图片大小确定字体大小
fontsize = min(im.size)/4
#加文字
font = ImageFont.truetype(fontPath + 'KhmerOS.ttf', fontsize)
draw.text((im.size[0]-fontsize, 0), '5', font = font, fill = (256,0,0))
im.save(myPath + outputFile,"jpeg")
```

2.生成随机数

```
#!/usr/bin/env python
#coding: utf-8
import string, random

#激活码中的字符和数字
field = string.letters + string.digits

#获得四个字母和数字的随机组合
def getRandom():
    return "".join(random.sample(field,4))

#生成的每个激活码中有几组
def concatenate(group):
    return "-".join([getRandom() for i in range(group)])

#生成n组激活码
def generate(n):
    return [concatenate(4) for i in range(n)]

if __name__ == '__main__':
    print generate(200)
```

3.统计单词

```
#!/usr/bin/env python
#coding: utf-8
import re
from collections import Counter
FILESOURCE = './media/abc.txt'

def getMostCommonWord(articlefilesource):
    '''输入一个英文的纯文本文件，统计其中的单词出现的个数'''
    pattern = r'''[A-Za-z]+|\$?\d+%?$'''
    with open(articlefilesource) as f:
        r = re.findall(pattern,f.read())
```

```
        return Counter(r).most_common()

if __name__ == '__main__':
    print getMostCommonWord(FILESOURCE)
```

4.最重要的词

目录下多个txt文件，找出词出现频率最高的

```
#!/usr/bin/env python
#coding: utf-8
import re, os
from collections import Counter

# 目标文件所在目录
FILE_PATH = './media/countword'

def getCounter(articlefilesource):
    '''输入一个英文的纯文本文件，统计其中的单词出现的个数'''
    pattern = r'''[A-Za-z]+|\$?\d+%?$'''
    with open(articlefilesource) as f:
        r = re.findall(pattern, f.read())
        return Counter(r)

#过滤词
stop_word = ['the', 'in', 'of', 'and', 'to', 'has', 'that', 's', 'is', 'are', 'a', 'with', 'as', 'an']

def run(FILE_PATH):
    # 切换到目标文件所在目录
    os.chdir(FILE_PATH)
    # 遍历该目录下的txt文件
    total_counter = Counter()
    for i in os.listdir(os.getcwd()):
        if os.path.splitext(i)[1] == '.txt':
            total_counter += getCounter(i)
    # 排除stopword的影响
    for i in stop_word:
        total_counter[i] = 0
    print total_counter.most_common()[0][0]

if __name__ == '__main__':
    run(FILE_PATH)
```

5.批量图片处理

你有一个目录，装了很多照片，把它们的尺寸变成都不大于 iPhone5 分辨率的大小

思路：遍历给出目录下的图片，把大于iPhone5分辨率的图片都进行缩放。使用Python的PIL库对图片进行处理，IPhone5屏幕分辨率为640 × 1136，将大于该分辨率的图片按照一定比例缩放至适合大小并保存。

```
#!/usr/bin/env python
#coding: utf-8
import Image, os

# 源目录
#myPath = './media/srcimg/'
myPath = '/home/itcast/workspace/chuanzhi/openlessons/demo_python/media/srcimg/'
# 输出目录
#outPath = './media/destimg/'
outPath = '/home/itcast/workspace/chuanzhi/openlessons/demo_python/media/destimg/'

def processImage(filesource, destsource, name, imgtype):
    '''
    filesource是存放待转换图片的目录
    destsource是存放输出转换后图片的目录
    name是文件名
    imgtype是文件类型
    '''
    imgtype = 'jpeg' if imgtype == '.jpg' else 'png'
```

```
    #打开图片
    im = Image.open(filesource + name)
    # 缩放比例
    rate = max(im.size[0]/640.0 if im.size[0] > 640 else 0, im.size[1]/1136.0 if im.size[1] > 1136 else 0)
    if rate:
        im.thumbnail((im.size[0]/rate, im.size[1]/rate))
    im.save(destsource + name, imgtype)

def run():
    # 切换到源目录，遍历源目录下所有图片
    os.chdir(myPath)
    for i in os.listdir(os.getcwd()):
        # 检查后缀
        postfix = os.path.splitext(i)[1]
        if postfix == '.jpg' or postfix == '.png':
            processImage(myPath, outPath, i, postfix)

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

6.统计代码行数（注释，空行，总行数）

思路：获取目录，然后遍历目录下的代码文件，逐个统计每个文件的代码，然后最后汇总输出。

```
#!/usr/bin/env python
#coding: utf-8
import os, re

# 代码所在目录
FILE_PATH = './'

def analyze_code(codefilesource):
    '''
    打开一个py文件，统计其中的代码行数，包括空行和注释
    返回含该文件总行数，注释行数，空行数的列表
    '''
    total_line = 0
    comment_line = 0
    blank_line = 0
    with open(codefilesource) as f:
        lines = f.readlines()
        total_line = len(lines)
        line_index = 0
        # 遍历每一行
        while line_index < total_line:
            line = lines[line_index]
            # 检查是否为注释
            if line.startswith("#"):
                comment_line += 1
            elif re.match("\s*'''", line) is not None:
                comment_line += 1
                while re.match(".*'''$", line) is None:
                    line = lines[line_index]
                    comment_line += 1
                    line_index += 1
            # 检查是否为空行
            elif line == "\n":
                blank_line += 1
            line_index += 1
    print "在%s中：" % codefilesource
    print "代码行数：", total_line
    print "注释行数：", comment_line, "占%0.2f%%" % (comment_line*100.0/total_line)
    print "空行数：  ", blank_line, "占%0.2f%%" % (blank_line*100.0/total_line)
    return [total_line, comment_line, blank_line]


def run(FILE_PATH):
    # 切换到code所在目录
    os.chdir(FILE_PATH)
    # 遍历该目录下的py文件
    total_lines = 0
```

```
    total_comment_lines = 0
    total_blank_lines = 0
    for i in os.listdir(os.getcwd()):
        if os.path.splitext(i)[1] == '.py':
            line = analyze_code(i)
            total_lines, total_comment_lines, total_blank_lines = total_lines + line[0], total_comment_lines +
    print "总代码行数：", total_lines
    print "总注释行数：", total_comment_lines, "占%0.2f%%" % (total_comment_lines*100.0/total_lines)
    print "总空行数：  ", total_blank_lines, "占%0.2f%%" % (total_blank_lines*100.0/total_lines)

if __name__ == '__main__':
    run(FILE_PATH)
```

7.提取HTML正文内容

思路：我把这里的正文理解为网页中我主要内容，那么怎么去抓取这个主要内容呢？我一开始的想法是用beautifulsoup来解析网页，但是又想到如果要抽取正文的话这样做还涉及到比较复杂的算法，而且对于不同的网页来说效果可能做不到很好。后来我发现了Python-goose（Github）这个神器，它是基于NLTK和Beautiful Soup的，分别是文本处理和HTML解析的领导者，目标是给定任意资讯文章或者任意文章类的网页，不仅提取出文章的主体，同时提取出所有元信息以及图片等信息，支持中文网页（用到了结巴分词）。这个正好符合需求，所以直接拿来用了。

安装python goose：

```
git clone https://github.com/grangier/python-goose.git
cd python-goose
pip install -r requirements.txt
python setup.py install
```

```
#!/usr/bin/env python
#coding: utf-8
from goose import Goose
from goose.text import StopWordsChinese
import sys
reload(sys)
sys.setdefaultencoding("utf-8")

# 要分析的网页url
#url = 'http://c.itcast.cn/index.html'
url = 'https://linux.cn/article-6717-1.html'

def extract(url):
    '''
    提取网页正文
    '''
    g = Goose({'stopwords_class': StopWordsChinese})
    article = g.extract(url=url)
    return article.cleaned_text

if __name__ == '__main__':
    print extract(url)
```

8.生成验证码图片

思路：先随机生成验证码，然后用Python的PIL库画出这个激活码的图片，具体点就是创建画布，加验证码的字上去，增加噪点进行干扰，再进行模糊处理，接着保存到名字为验证码的图片中。

```
#!/usr/bin/env python
#coding: utf-8
import Image, ImageDraw, ImageFont, ImageFilter
import string, random

fontPath = "./media/"

# 获得随机四个字母
def getRandomChar():
    return [random.choice(string.letters) for _ in range(4)]
```

```
# 获得颜色
def getRandomColor():
    return (random.randint(30, 100), random.randint(30, 100), random.randint(30, 100))

# 获得验证码图片
def getCodePiture():
    width = 240
    height = 60
    # 创建画布
    image = Image.new('RGB', (width, height), (180,180,180))
    font = ImageFont.truetype(fontPath + 'KhmerOS.ttf', 40)
    draw = ImageDraw.Draw(image)
    # 创建验证码对象
    code = getRandomChar()
    # 把验证码放到画布上
    for t in range(4):
        draw.text((60 * t + 10, 0), code[t], font=font, fill=getRandomColor())
    # 填充噪点
    for _ in range(random.randint(1500,3000)):
        draw.point((random.randint(0,width), random.randint(0,height)), fill=getRandomColor())
    # 模糊处理
    image = image.filter(ImageFilter.BLUR)
    # 保存名字为验证码的图片
    image.save("".join(code) + '.jpg', 'jpeg');

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

六. Linux系统网络编程实战

requests网络库使用

虽然Python的标准库中urllib2模块已经包含了平常我们使用的大多数功能，但是它的API使用起来让人实在感觉不好。它已经不适合现在的时代，不适合现代的互联网了。而Requests的诞生让我们有了更好的选择。

正像它的名称所说的，HTTP for Humans,给人类使用的HTTP库！ 在Python的世界中，一切都应该简单。Requests使用的是urllib3，拥有了它的所有特性，Requests 支持 HTTP 连接保持和连接池， 支持使用 cookie 保持会话，支持文件上传，支持自动确定响应内容的编码， 支持国际化的 URL 和 POST 数据自动编码。现代、国际化、人性化。

1.发送无参数的get请求

```
import requests
r = requests.get('http://httpbin.org/get')
print r.text
```

```
{
  "args": {},
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate, compress",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.2.1 CPython/2.7.6 Linux/3.16.0-30-generic"
  },
  "origin": "61.148.201.2",
  "url": "http://httpbin.org/get"
}
```

2.发送带参数的get请求,将key与value放入一个字典中，通过params参数来传递

```
import requests
myparams={'q':'Linux'}
r=requests.get('https://www.haosou.com/s', params=myparams)
r.url
print r.content
```

3.发送post请求，通过data参数来传递

```
import requests
mydata={'wd':'Linux', 'name':'xwp'}
r = requests.post("http://httpbin.org/post", data=mydata)
print r.text
```

```
{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "name": "xwp",
    "wd": "Linux"
  },
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate, compress",
    "Content-Length": "17",
    "Content-Type": "application/x-www-form-urlencoded",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.2.1 CPython/2.7.6 Linux/3.16.0-30-generic"
  },
  "json": null,
  "origin": "61.148.201.2",
  "url": "http://httpbin.org/post"
}
```

4.发送post请求，通过json参数来传递

```
import json
import requests
mydata={'wd':'Linux', 'name':'xwp'}
r = requests.post('http://httpbin.org/post', data=json.dumps(mydata))
print r.text
```

```
{
  "args": {},
  "data": "{\"wd\": \"Linux\", \"name\": \"xwp\"}",
  "files": {},
  "form": {},
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate, compress",
    "Content-Length": "30",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.2.1 CPython/2.7.6 Linux/3.16.0-30-generic"
  },
  "json": {
    "name": "xwp",
    "wd": "Linux"
  },
  "origin": "61.148.201.2",
  "url": "http://httpbin.org/post"
}
```

5.上传文件

上传一张照片,这时要使用files参数

```
import requests
files = {'file': open('xwp.jpg', 'rb')}
r = requests.post(url, files=files)
print r.text
```

6.实战案例

```
#coding:utf-8
import requests
import re
```

```
url = r'http://www.renren.com/ajaxLogin/login'

user = {'email':'username','password':'passwd'}
s = requests.Session()
r = s.post(url,data = user)

html = r.text
visit = []
first = re.compile(r'</span><span class="time-tip first-tip"><span class="tip-content">(.*?)</span>')
second = re.compile(r'</span><span class="time-tip"><span class="tip-content">(.*?)</span>')
third = re.compile(r'</span><span class="time-tip last-second-tip"><span class="tip-content">(.*?)</span>')
last = re.compile(r'</span><span class="time-tip last-tip"><span class="tip-content">(.*?)</span>')
visit.extend(first.findall(html))
visit.extend(second.findall(html))
visit.extend(third.findall(html))
visit.extend(last.findall(html))
for i in visit:
        print i

print '以下是更多的最近来访'
vm = s.get('http://www.renren.com/myfoot/whoSeenMe')
fm = re.compile(r'"name":"(.*?)"')
visitmore = fm.findall(vm.text)
for i in visitmore:
        print i
```

socket原生网络库使用

server1:原始socket AIP

```
#coding:utf-8

from socket import *

myhost = ''
myport = 8080
sockobj = socket(AF_INET, SOCK_STREAM)
sockobj.bind((myhost, myport))
sockobj.listen(128)
while True:
    connection, address = sockobj.accept()
    print "connect by", address
    while True:
        data = connection.recv(1024)
        if not data:
            break
        connection.send('echo' + data)
    connection.close()
```

server2

```
#coding:utf-8

from SocketServer import TCPServer, BaseRequestHandler
import traceback

class MyBaseRequestHandlerr(BaseRequestHandler):
    """
    #从BaseRequestHandler继承，并重写handle方法
    """

    def handle(self):
        #循环监听（读取）来自客户端的数据
        while True:
            #当客户端主动断开连接时，self.recv(1024)会抛出异常
            try:
                #一次读取1024字节,并去除两端的空白字符(包括空格,TAB,\r,\n)
                data = self.request.recv(1024).strip()

                #self.client_address是客户端的连接(host, port)的元组
```

```
                print "receive from (%r):%r" % (self.client_address, data)

                #转换成大写后写回(发生到)客户端
                self.request.sendall(data.upper())
            except:
                traceback.print_exc()
                break

if __name__ == "__main__":
    #telnet 127.0.0.1 9999
    host = ""       #主机名，可以是ip,像localhost的主机名,或""
    port = 8080     #端口
    addr = (host, port)

    #购置TCPServer对象，
    server = TCPServer(addr, MyBaseRequestHandlerr)

    #启动服务监听
    server.serve_forever()
```

server3:多线程TCPserver

```
#coding:utf-8

from SocketServer import ThreadingTCPServer, StreamRequestHandler
import traceback

class MyStreamRequestHandlerr(StreamRequestHandler):
    def handle(self):
        while True:
            try:
                data = self.rfile.readline().strip()
                print "receive from (%r):%r" % (self.client_address, data)
                self.wfile.write(data.upper())
            except:
                traceback.print_exc()
                break

if __name__ == "__main__":
    host = ""       #主机名，可以是ip,像localhost的主机名,或""
    port = 8080     #端口
    addr = (host, port)

    #ThreadingTCPServer从ThreadingMixIn和TCPServer继承
    #class ThreadingTCPServer(ThreadingMixIn, TCPServer): pass
    server = ThreadingTCPServer(addr, MyStreamRequestHandlerr)
    server.serve_forever()
```

七. 模块化借力C/C++

借力C/C++，提高程序性能，实现代码复用

为什么需要扩展python

1. 性能瓶颈的效率提升
2. 保持源代码的私密性，如加解密算法

创建python扩展流程

1. 创建C/C++功能代码
2. python类型适配，包装C/C++代码
3. 编译与测试

C/C++版本的功能函数

```
#include <stdio.h>
```

```c
#include <stdlib.h>
#include <string.h>

int fac(int n)
{
    if (n < 2)
        return 1;

    return n * fac(n - 1);
}

char *reverse(char *s)
{
    char t, *p = s, *q = (s + (strlen(s)-1));

    while (s && (p < q)) {
        t = *p;
        *p++ = *q;
        *q-- = t;
    }

    return s;
}

//int test(void)
int main(void)
{
    char s[1024];

    printf("4! == %d\n", fac(4));
    printf("8! == %d\n", fac(8));

    strcpy(s, "itcastcpp");
    printf("reversing 'itcastcpp', we get '%s'\n", reverse(s));

    return 0;
}
```

头文件

```c
#ifndef ITCASTCPP_H_
#define ITCASTCPP_H_

int fac(int n);
char *reverse(char *s);
int test(void);

#endif
```

编译测试，保证 c 代码的正确性，避免在python中再去调试 c 模块。

包裹函数

实现包裹，主要分 4 步：

1. 包含Python.h头文件
2. 为每一个函数增加一个PyObject *Module_func()的包裹函数
3. 为模块增加一个PyMethod DefModuleMethods[]的数组
4. 增加模块的初始化函数void initModule()

```
sudo apt-get install python-dev
```

包含Python.h头文件

确保你的系统上安装过python，我的系统是ubuntu14.04，头文件路径为:

```
 /usr/include/python2.7
c中包含此头文件
 #include "Python.h"
```

为每一个函数增加一个PyObject *Module_func()的包裹函数

包裹函数名字:

```
模块名_函数名
```

python调用时:

```
模块名.函数名
```

python到 c ，把python传过来的参数转为 C 的类型:

```
int PyArg_ParseTuple()
```

c到python, 把 C 的数据转为python的一个或一组对象返回

```
PyObject *Py_BuildValue()
```

Python和C/C++之间数据转换：

```
格式代码    python类型    c 类型
s           str         char *
z           str/None    char */NULL
i           int         int
l           long        long
c           str         char
d           float       double
D           complex     Py_Complex*
O           (any)       PyObject *
S           str         PyStringObject
```

为模块增加一个PyMethod DefModuleMethods[]的数组

```
static PyMethodDef ItcastcppMethods[] = {
    {"fac", Itcastcpp_fac, METH_VARARGS},
    {"doppel", Itcastcpp_doppel, METH_VARARGS},
    {"test", Itcastcpp_test, METH_VARARGS},
    {NULL, NULL},
};
```

增加模块的初始化函数void initModule()

模块名和模块所支持的方法

```
void initItcastcpp(void)
{
    Py_InitModule("Itcastcpp", ItcastcppMethods);
}
```

代码实现如下：

```
#include "Python.h"
#include <stdlib.h>
#include <string.h>
#include "Itcastcpp.h"

static PyObject *Itcastcpp_fac(PyObject *self, PyObject *args)
{
    int num;

    if (!PyArg_ParseTuple(args, "i", &num))
        return NULL;
```

```
    return (PyObject *)Py_BuildValue("i", fac(num));
}

static PyObject *Itcastcpp_doppel(PyObject *self, PyObject *args)
{
    char *src;
    char *mstr;
    PyObject *retval;

    if (!PyArg_ParseTuple(args, "s", &src))
        return NULL;

    mstr = malloc(strlen(src) + 1);
    strcpy(mstr, src);
    reverse(mstr);
    retval = (PyObject *)Py_BuildValue("ss", src, mstr);
    free(mstr);

    return retval;
}

static PyObject *Itcastcpp_test(PyObject *self, PyObject *args)
{
    test();

    return (PyObject *)Py_BuildValue("");
}

static PyMethodDef ItcastcppMethods[] = {
    {"fac", Itcastcpp_fac, METH_VARARGS},
    {"doppel", Itcastcpp_doppel, METH_VARARGS},
    {"test", Itcastcpp_test, METH_VARARGS},
    {NULL, NULL},
};

void initItcastcpp(void)
{
    Py_InitModule("Itcastcpp", ItcastcppMethods);
}
```

编译安装到python环境

1. 创建setup.py
2. 运行setup.py编译和链接C的扩展代码
3. 从Python中导入模块
4. 测试

创建setup.py

为了能编译扩展，需要为每一个扩展创建一个Extension实例

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

from distutils.core import setup, Extension

MOD = "Itcastcpp"

setup(name=MOD, ext_modules=[Extension(MOD, sources=['Itcastcpp.c', 'Itcastcppwrapper.c'])])
```

运行setup.py编译和链接C的扩展代码

```
$ python setup.py build

running build
running build_ext
building 'Itcastcpp' extension
creating build
creating build/temp.linux-x86_64-2.7
x86_64-linux-gnu-gcc -pthread -fno-strict-aliasing -DNDEBUG -g -fwrapv -O2 -Wall -Wstrict-prototypes -fPIC -I/
```

```
x86_64-linux-gnu-gcc -pthread -fno-strict-aliasing -DNDEBUG -g -fwrapv -O2 -Wall -Wstrict-prototypes -fPIC -I/
creating build/lib.linux-x86_64-2.7
x86_64-linux-gnu-gcc -pthread -shared -Wl,-O1 -Wl,-Bsymbolic-functions -Wl,-Bsymbolic-functions -Wl,-z,relro -
```

从Python中导入模块和测试

```
$ sudo python setup.py install
如果成功会看到
running install
running build
running build_ext
running install_lib
copying build/lib.linux-x86_64-2.7/Itcastcpp.so -> /usr/local/lib/python2.7/dist-packages
running install_egg_info
Removing /usr/local/lib/python2.7/dist-packages/Itcastcpp-0.0.0.egg-info
Writing /usr/local/lib/python2.7/dist-packages/Itcastcpp-0.0.0.egg-info

$ ipython
import  Itcastcpp
Itcastcpp.fac(5)
120
```

八. web框架Django开发

移动互联网+项目开发

Django开发模型

微信公众号app开发