

# Python 新员工教材

楚广明 2012

借用了一些闫小勇\郑纳智同志的文档，向同志们致敬！

## 目录

一、	Python 概述 .....	6
1)	第一个 Python 程序 .....	7
1.	为什么要学习 Python .....	8
2.	代码块与缩进的差异 .....	8
3.	语句结尾的差异 .....	9
4.	注释方法 .....	9
5.	入口方法 .....	9
6.	import 与 using 语句 .....	9
2)	小结 .....	9
二、	基本数据类型 .....	10
1)	第一个程序的解析 .....	10
2)	变量的命名规范 .....	12
3)	空类型 .....	12
4)	布尔类型 .....	12
5)	数值类型 .....	13
6)	字符串类型 .....	13
7)	全局变量 .....	14
三、	基本数据类型-列表(list【】) .....	14
四、	基本数据类型-字典(dict { }) .....	16
五、	基本数据类型-元组(tuple ( )) .....	19
六、	基本数据类型-集合(set) .....	20
七、	运算符、表达式和流程控制 .....	21
1)	算术运算符与算术表达式 .....	21
2)	赋值运算符与赋值表达式 .....	22
3)	关系运算符与关系表达式 .....	22
八、	流程控制 .....	22
1)	条件语句 .....	22
2)	循环 .....	23
九、	异常 .....	24
十、	动态表达式 .....	25
十一、	函数及函数编程 .....	26
1)	函数的定义 .....	26
2)	定义一个交换函数 .....	26
3)	函数的默认参数与返回值 .....	27
4)	返回多个值 .....	27
5)	locals 函数和 globals 函数介绍 .....	27
十二、	类及面向对象 .....	28
1)	类的定义 .....	28
2)	为类添加数据 .....	29
3)	构造函数 .....	29
4)	静态成员与私有成员 .....	30
5)	为类添加方法 .....	30
6)	静态方法 .....	31
7)	单继承 .....	31
十三、	模块与包 .....	33
1)	对于模块的理解 .....	33

2)	模块的显要特性：属性 .....	34
3)	模块创建过程的例子 .....	34
4)	模块的导入 .....	34
5)	模块的属性 .....	35
十四、	字符串与简单的正则表达式 .....	35
1)	解决中文字符的问题 .....	36
2)	字符串的格式化 .....	37
3)	字符串的合并与分割 .....	38
4)	常用字符操作演示 .....	39
十五、	文件 IO .....	42
1)	基本文件功能演示 .....	42
2)	文件的写入 .....	44
3)	文件的删除与复制 .....	44
4)	文件与目录的重命名 .....	45
5)	文件内容的查找和替换 .....	46
6)	文件的比较 .....	47
十六、	Web.Py 框架概述 .....	47
1)	Python 下 web 开发框架的选择 .....	47
1.	Django .....	47
2.	Pylons & TurboGears & repoze.bfg .....	48
3.	Tornado & web.py .....	49
4.	Bottle & Flask .....	50
5.	Quixote .....	50
6.	最后关于框架选择的误区 .....	50
2)	web.py 安装 .....	51
3)	web.py 下常用框架简介 .....	51
4)	最简单的 HelloWorld! .....	51
5)	Web.py 基本使用模板使用 .....	52
6)	进阶模板使用 .....	53
1.	server2.py .....	53
2.	post.html .....	53
7)	URL 控制 .....	54
1.	问题: .....	54
2.	解决: .....	54
3.	路径匹配 .....	54
8)	一个完整的小例子 .....	54
9)	web.seeother 和 web.redirect 转向 .....	55
1.	问题 .....	55
2.	解法 .....	55
3.	区别 .....	55
10)	包含应用 .....	56
1.	实现 .....	56
11)	使用 XML .....	56
1.	问题 .....	56
2.	解法 .....	57
12)	获取 POST 数据 .....	57
1.	login.html 模板 .....	58
2.	主程序 .....	58

13)	获取客户端信息.....	59
1.	问题.....	59
2.	解法.....	59
3.	例子.....	59
4.	'ctx'中的数据成员.....	59
14)	文件上传.....	60
1.	简单的文件上传.....	60
2.	基于新浪云的上传.....	60
十七、	Web.py 使用 Session.....	62
1)	问题.....	62
2)	解法.....	62
3)	在 template 下使用 Session.....	64
1.	问题:.....	64
2.	解决:.....	64
4)	如何操作 Cookies.....	65
1.	问题.....	65
2.	解法.....	65
3.	设置 Cookies.....	65
4.	获得 Cookies.....	66
5)	布局模板.....	67
1.	问题.....	67
2.	方法.....	67
3.	Tip: 在布局文件（layout.html）中定义的页面标题变量.....	68
4.	Tip: 在其他模板中引用 css 文件，如下:.....	68
十八、	Web.py 支持的语法模板.....	69
1)	使用 Templetor 模板语法.....	69
1.	Introduction.....	69
2.	使用模板系统.....	69
3.	表达式用法.....	70
4.	赋值.....	70
5.	过滤.....	71
6.	新起一行用法.....	71
7.	转义 \$.....	72
8.	注释.....	72
9.	控制结构.....	72
10.	使用 def.....	73
11.	使用 var.....	73
12.	内置 和 全局.....	74
2)	使用 Jinja2 模板语法.....	75
1.	上手的一个小例子.....	75
十九、	Web.py 中使用数据库.....	76
1)	使用 web.py 自带的 DB 封装库.....	76
1.	一个简单的数据库访问测试.....	76
2.	一个复杂一点的例子.....	77
3.	更加复杂的例子.....	77
二十、	使用 Sqlalchemy.....	79
1)	简介.....	79
2)	初始化 Sqlite 数据库.....	81

3)	插入数据库 .....	82
4)	在视图中使用 <code>sqlalchemy</code> .....	85
1.	首先创建数据库 <code>models.py</code> .....	85
2.	使用 <code>app.py</code> .....	86
二十一、	<code>Json</code> 的使用 .....	87
二十二、	<code>Urllib</code> 模块 .....	88
1)	<code>urlopen</code> 的基本使用 .....	88
2)	<code>urlretrieve</code> 方法的使用 .....	89
3)	<code>url</code> 编码 .....	89
4)	使用 <code>httplib</code> 抓取 .....	90
二十三、	<code>Urllib2</code> 的使用 .....	92
1)	最简单的爬虫 .....	92
2)	提交表单数据 .....	92
1.	用 <code>GET</code> 方法提交数据 .....	92
二十四、	操作 <code>Memcache</code> .....	93
1)	安装 .....	93
1.	<code>Linux</code> 环境 .....	93
2.	<code>Win</code> 环境 .....	93
3.	<code>linux</code> 启动 <code>memcached</code> .....	93
4.	<code>windows</code> 下启动 .....	94
2)	<code>Python</code> 操作 <code>Memcached</code> .....	94
1.	<code>Python</code> 操作 <code>memcached</code> .....	95
2.	<code>Memcached</code> 常用方法 .....	95
二十五、	<code>SQLite</code> 模块 .....	95
1)	简单的介绍 .....	95
2)	安装与使用 .....	96
3)	一个简单的例子 .....	98
4)	中文处理 .....	100
二十六、	正则表达式 .....	100
1)	简介 .....	100
2)	3 个重要的正则式命令 .....	101
1.	验证字符串 .....	103
2.	验证 <code>url</code> .....	103
3.	验证 <code>Email</code> .....	104
4.	验证值在 0-25 的数字 .....	104
5.	验证格式为 <code>MM/DD/YYYY</code> , <code>YYYY/MM/DD</code> and <code>DD/MM/YYYY</code> 的日期 .....	104
3)	正则表达式 .....	105
1.	你常用的元字符 .....	105
2.	常用的限定符 .....	105
3.	字符串类 .....	106
4.	分枝条件 .....	107
5.	分组 .....	107
6.	反义 .....	108
7.	后向引用 .....	108
8.	贪婪与懒惰 .....	109
4)	在 <code>Python</code> 中使用 .....	109
5)	<code>Re</code> 模块 .....	111
6)	<code>Match</code> 方法 .....	112

# 一、 Python 概述

Python 是一门优雅而健壮的编程语言，它继承了传统编译语言的强大性与通用性，同时也借鉴了简单的脚本和解释语言的易用性。它可以帮你完成工作，而且一段时间之后，你还能看明白自己写的这段代码。你会对自己如此快地学会它和它强大的功能感到十分的惊讶，更不用提你已经完成了工作了，只有你想不到，没有 Python 做不到的。

就算你的项目中有大量的 Python 代码，你也依旧可以有条不紊地通过将其分离为多个文件或模块加以组织管理。而且你可以从一个模块中选取代码，而从另一个模块中读取属性。更棒的是，对于所有模块，Python 的访问语法都是相同的。不管这个模块是 Python 标准库中的还是你一分钟前创建的，哪怕是你用其他语言写的扩展都没有问题！借助这些特点，你会自己根据需要扩展了这门语言，而且你也这么做了。

代码中的瓶颈可能是在性能分析中总排在前面的那些热门或者一些特别强调性能的地方，可以作为 Python 扩展用 C 重写。需要重申的是，这些接口和纯 Python 模块的接口是一模一样的，乃至代码和对象的访问方法也如出一辙的，唯一不同的是，这些代码为性能带来了显著的提升，我们可以利用 Pyrex 这样的工具允许 C 和 Python 混合编程，使编写扩展轻而易举，因为它会把所有的代码都转换成 C 语言代码。

因为 Python 的标准实现是使用 C 语言完成的（也就是 CPython），所以要使用 C 和 C++ 编写 Python 扩展。Python 的 java 实现被称为 Jython，要使用 java 编写其扩展。最后还有 IronPython 这是针对 .net 平台的实现。

在各种不同的系统上你都可以看到 Python 的身影，因为 Python 是用 C 写的，又由于 C 的可移植性，使得 Python 可以运行在任何带有 ANSI C 编译平台上。

## ● 内置的数据类型

Python 提供了一些内置的数据结构，这些数据结构实现了类似 Java 中集合类的功能，Python 的数据结构包括元组、列表、字典等。内置的数据结构简化了程序的设计。元组相当于“只读”的数组，列表可以作为可变长度的数组使用，字典相当于 java 中的 HashTable 类型。

## ● 健壮性

Python 提供了异常处理机制，能捕获程序的异常情况。此外，Python 的堆栈跟踪对象能够指出程序出错的位置和出错的原因。异常机制能够避免不安全退出的情况，同时能帮助程序员调试程序。

## ● 跨平台性

Python 会先编译与平台相关的二进制代码，然后再解释执行，这种方式 and Java 相似。Python 可以运行在 Windows/Linux/MAC/Unix 上

## ● 可扩展性

Python 是采用 C 开发的语言，因此可以使用 C 扩展 Python，可以给 Python 添加新的模块、新的类。

## ● 动态性

Python 与 Javascript、PHP、Perl 等语言类似。Python 不需要声明变量，直接赋予值可创建一个新的变量。

## ● 强类型语言

Python 的变量创建后会对应一种数据类型，Python 会根据赋值表达式的内容决定变量的数据类型。Python 在内部建立了管理这些变量的机制，出现在同一个表达式中的不同类型的变量需要做类型转换。

## ● 应用广泛

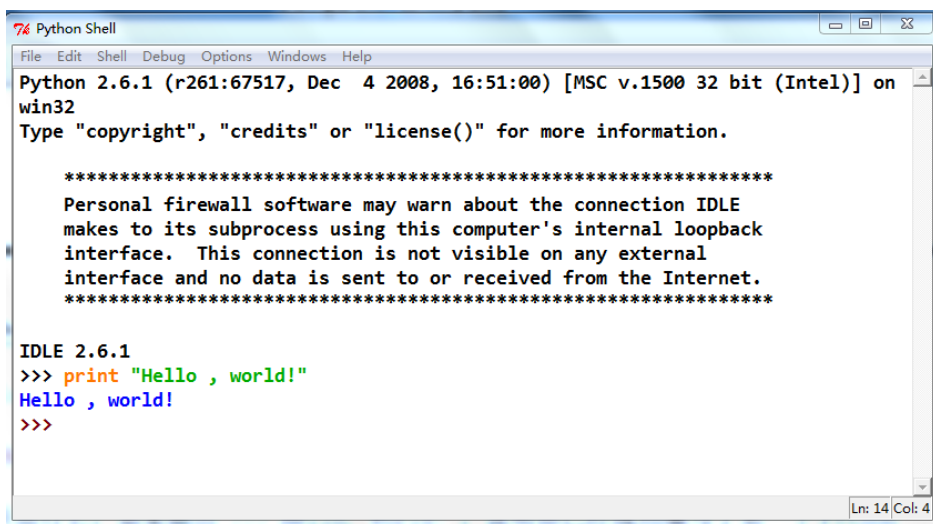
Python 语言应用于数据库、网络、图形图像、数学计算、WEB 开发、操作系统扩展等领域。Python 有许多第三方库的支持。例如，PIL 库用于图像处理、NumPy 库用于数据计算、WxPython 用于 GUI 库的设计、Django 库用于 WEB 应用程序的开发等

## 1) 第一个 Python 程序

在 Python 的官方网站可以下载到 Windows 下的安装包，按照提示一路下去就可以了，记得要将 Python 所在的目录加入到系统 Path 变量中。

Python 的安装包自带了一个简单的集成开发环境 IDIE，你也可以选一个自己喜欢的 IDE，我个人推荐 PythonWin，它的语法提示功能不错，适合初学者使用。

现在你可以打开 IDIE 新建一个 py 为扩展名的 Python 脚本文件，输入以下内容：



```
Python 2.6.1 (r261:67517, Dec 4 2008, 16:51:00) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.

*****
Personal firewall software may warn about the connection IDLE
makes to its subprocess using this computer's internal loopback
interface. This connection is not visible on any external
interface and no data is sent to or received from the Internet.
*****

IDLE 2.6.1
>>> print "Hello , world!"
Hello , world!
>>>
```

保存并运行它，如果输出>>>Hello ,World!，这说明你已经成功编写了第一个 Python 程序，恭喜你！为了比较 Python 与 Java 编码风格等方面的差异，下边给出一个稍微复杂些的“Hello world”程序以及它的 Java 对照版本。

```
1  # -*- coding:utf-8 -*-
2  """
3  我的第一个Python程序
4  """
5  __author__ = 'chuguangming'
6  import sys
7  def Main():
8      sys.stdout.write("Hello world!楚广明\n")
9  if __name__ == "__main__":
10     Main()
```

请注意第一行代码是为了支持中文，没有这一行代码的话，我们在下面如果输出中文就会出错的。

### Java 的实现

```
/**
 * @author ChuguangMing
 *
 */
```

```
public class myfirstjava
{
    public static void main(String[] args)
    {
        System.out.println("这是 Java 的第一个程序");
    }
}
```

### Python 的实现

```
#-*-coding:utf-8-*-
"""
我的第一个应用程序
"""
import sys
def Main():
    sys.stdout.write("Hello World 我的第一个程序\n")
#下面的语句看起来比较奇怪，一会儿我们会解析它
if __name__=="__main__":
    Main()
```

## 1. 为什么要学习 Python

我学习过很多语言，C#、Java、C/C++、PHP，但是效高即高而且简单易懂的也就数 Python 了，可以说它是一种胶水语言，我们可以通过它将许多语言整合在一起使用。

我们来看一段代码：

```
#-*-coding:utf-8-*-
"""
my fist App
"""
import sys
import urllib
def Main():
    htmlresult=urllib.urlopen("http://www.baidu.com").read()
    print htmlresult
#this is test
if __name__=="__main__":
    Main()
```

## 2. 代码块与缩进的差异

**Java 使用 C/C++ 风格的编码形式，除了要求用 {} 组织代码块外，语句间的缩进可以是任意的。**

Python 强制所有程序都有相同的编码风格，它通过缩进来组织代码块，缩进相同的语句被认为是处于同一个代码块中，在 if/else 等语句及函数定义式末尾会有一个冒号，指示代码块的开始。Python 这种强制缩进的做法可以省去 {} 或者 begin/end 等，使程序的结构更为清晰（有的人认为恰好相反），同时也减少了无效的代码行数。

此外需要注意，尽量使用 4 个空格作为 Python 代码的一个缩进单位，最好不要使有 TAB，更不要混用 Tab 和空格，这也算是 Python 的一个非强制性约定吧。



### 3. 语句结尾的差异

Java 用分号结尾，Python 不用任何符号（类似 BASIC）。实际上 Python 也可以使用分号结尾，像这样 `a=1;b=2;c=3;print a,b,c` 不过 Python 中这种风格多用于调试，应为你可以很容易注释掉这一行就删除了所有的调试代码。

另外，当一行很长时，Python 可以用 \ 符号折行显示代码。

### 4. 注释方法

java//用单行注释,用/\*\*/进行多行注释，而 Python 用#符号进行单行注释，用三引号（可单可双）进行多行注释。

java 的条件表达式必须要加括号，而 Python 的条件表达式加不加括号均可。

### 5. 入口方法

java 语言必须要有入口方法 `Main()`，这是程序开始执行的地方。Python 语言中没有入口方法（函数），作为解释型的语言，Python 代码会自动从头执行。

如果你对这点不习惯，可以使用 Python 代码的内置属性 `__name__` 此属性会根据 Python 代码的运行条件变化：当 Python 代码以单个文件运行时，`__name__` 便等于 `__main__`，当你以模块形式导入使用 Python 代码时，`__name__` 属性便是这个模块的名字。

当然，Python 中的 `__name__` 属于并不是为了照顾 C/C++/C# 程序员的编程习惯而准备的，它主要目的是用于模块测试。想像一下在 C# 中编写一个组件或类代码时，一般还得同时编写一个调用程序来测试它。而 Python 中可以把二者合二为一，这就是 `__name__` 属性的真正作用。

### 6. import 与 using 语句

在用 Python 写代码时，我们首先 `import sys`，这是导入了 Python 的 `sys` 模块，然后在代码里我们可以引用 `sys` 模块中的对象 `stdout` 及它的 `write` 方法。在 Python 中这是必须的，否则你无法调用 `sys` 模块中的任何东西。

简单的说，Python 中的 `import` 相当于 java 中的包引用。最后 `import` 可以出现在代码的任何位置，只要在引用它之前出现就可以了，不过为了提高程序可读性，建议还是在所有代码开头书写 `import`

## 2) 小结

- 1、Python 使用强制缩进的编码风格，并以此组织代码块
- 2、Python 语句结尾不用分号
- 3、Python 标明注释用#(单行)或三引号（多行）
- 4、Python 语言没有入口方法（Main），代码会从头到尾顺序执行
- 5、Python 用 `import` 引用所需要的模块

## 二、基本数据类型

### 1) 第一个程序的解析

“一切数据是对象，一切命名是引用”

如果你能理解这句话，说明对 Python 的变量与数据类型已经有了不错的认识，与 Java 不同，Python 在使用变量之前无须定义它的类型，试着运行下面的例子：

```
#-*-coding:utf-8-*-
"""
我的第一个应用程序
"""
import sys
def Main():
    sys.stdout.write("开始程序")
    i=1
    print i
#下面的语句看起来比较奇怪，一会儿我们会解析它
if __name__=="__main__":
    Main()
```

从上边我们可以看到，变量 `i` 在使用前并不需要定义，但是**必须声明以及初始化**该变量。试着运行下面的例子：

```
i=1
print i+j
```

上面的代码会产生一个异常：“NameError:name ‘h’ is not defined”，Python 提示变量 `j` 没有定义。这一点和 BASIC 等弱类型的语言不一样。

另一方面，Python 与 java 有一个很大的差异就是在程序运行过程中，**同一变量名可以（在不同阶段）代表不同类型的数据**，看看下面的例子：

```
#-*-coding:utf-8-*-
"""
我的第一个应用程序
"""
import sys
def Main():
    sys.stdout.write("开始程序\n")
    i=1
    print i,type(i),id(i)
    i=1000000000
    print i,type(i),id(i)
    i=1.1
    print i,type(i),id(i)
#下面的语句看起来比较奇怪，一会儿我们会解析它
if __name__=="__main__":
    Main()
```

变量 **i** 的类型在程序执行过程中分别经历了 **int long float** 的变化,这和静态类型语言如 **C** 语言有很大不同。静态语言只要有一个变量获取了一个数据类型,它就会一直是这个类型,变量名代表的是用来存放数据的内存位置。而 **Python** 中使用的变量名只是各种数据及对象的引用,用 **id()** 获取的才是存放数据的内存位置,我们输入的 **1**、**1000000000**、**1.1** 三个数据均会保存在 **id()** 所指示的这些内存位置中,直到垃圾回收把它们拉走,这是动态语言的典型特征,它确定一个变量的类型是在给它赋值的时候。

另一方面, **Python** 又是强类型的,试着运行下边的例子:

```
#-*-coding:utf-8-*-
"""
我的第一个应用程序
"""
import sys
def Main():
    sys.stdout.write("开始程序\n")
    i=10
    j='虎虎'
    print i+j
#下面的语句看起来比较奇怪，一会儿我们会解析它
if __name__=="__main__":
    Main()
```

这个程序会出错,产生一个异常”**TypeError:unsupported operand type(s) for +:'int' and 'str'”**。一个正确的写法是:

```
#-*-coding:utf-8-*-
"""
我的第一个应用程序
"""
import sys
def Main():
    sys.stdout.write("开始程序\n")
    i=10
    j='虎虎'
    print str(i)+j
#下面的语句看起来比较奇怪，一会儿我们会解析它
if __name__=="__main__":
    Main()
```

所以,我们说 **Python** 即是一种动态类型语言,同时也是一种强类型的语言,这点是和 **java** 不同的地方,对于 **Python** 的这种变量的声明、定义、使用方式。**Java** 程序员可能要花一段时间去适应,不过相信你会很快喜欢上它,因为它让事情变的更加简单（而且不会不安全）。

我们再小试牛刀一下,比如我们想看看 **c** 盘下面有什么数据与目录的话:

```
#-*-coding:utf-8-*-
import sys
import os
def main():
    sys.stdout.write("Hello World 我的第一个程序\n")
    print os.listdir("c:/")
if __name__=="__main__":
    main()
```

## 2) 变量的命名规范

Python 与 java 的变量（函数、类等其他标识符）的命名规范基本一样，同样对大小写敏感。不一样的地方是 Python 中以下划线开始或者结束的标识符通常有特殊意义。例如以一个下划线开始的标识符如 `_foo` 不能用 `from module import *` 语句导入。前名均有两个下划线的标识符，如 `__init__` 被特殊方法保留。前边有两个下划线的标识符，如 `__bar`，被用来实现类私有属性，这个将在“类和面向对象编程”中再说。

最后，Python 的关键字不能作为标识符，不过 Python 的关键字比 java 要少得多，可以 google 一下，这里就列出了。

Python 没有常量，如果你非要定义常量，可以引用 `const` 模块

Python 程序中一切数据都是对象，包括自定义对象及基本数据类型，这一点和 C# 一样，它们都是完全面向对象的语言，所以我想 C# 程序员很容易理解 **Python 的一切数据都是对象** 这个口号。

Python 不区分值类型和引用类型，你可以把所有的类型都理解为引用类型（当然，它们的实现方式是不一样的，这里只是一个类比）。

Python 内建的数据类型有 20 多种，其中有些不常用到的，有些即将合并。本文将主要介绍空类型、布尔类型、整型、浮点型和字符串、元组、列表、集合、字典等 9 种 Python 内置的数据类型。

在这里，我们将前 4 种称为“简单数据类型”，将后 5 种称为“高级数据类型”，实际上 Python 语言本身没有这种叫法，这样分类是我自己设定的，主要是为了和 java 的相关知识对照方便，希望不要误导大家。

## 3) 空类型

空类型 (None) 表示该值是一个空对象，比如没有明确定义返回值的函数就是返回 None。空类型没有任何属性，经常被用做函数中可选参数的默认值。None 的布尔值为假。

Python 的 None 和 java 中的可空类型 `Nullable<T>` 类似，比如 java 可以定义 `Nullable<double> i=null`，与 Python 的空类型类似，但实现原理和用途都不一样。

## 4) 布尔类型

Python 中用 True 和 False 来定义真假，你可以直接用 `a=True` 或 `a=False` 来定义一个布尔型变量，但在 Python2.6 里，True/False 以及 None 却都不是关键字，在 Python3.0 里它们已经是关键字了，这个有点乱，我们可以不用管它，直接使用就 OK 了。

**注意和 java 不同的是，Python 中 True 和 False 的首字母要大写。**

最后一点，在 java 中布尔类型和其他类型之间不存在标准的转换。但是在 Python 中，None、任何数值类型中的 0、空字符串''，空元组 ()，空列表 []，空字典 {} 都被当作 False，其他对象均为 True，这点和 C++ 差不多，要提起注意，请思考一下，下面的 Python 代码会输出什么？

```
#-*-coding:utf-8-*-
"""
我的第一个应用程序
"""
import sys
def Main():
    sys.stdout.write("开始程序\n")
    if 0:
        print 'True'
    else:
        print 'False'
```

```
#下面的语句看起来比较奇怪，一会儿我们会解析它
if __name__=="__main__":
    Main()
```

## 5) 数值类型

Python 拥有四种数值类型：整型、长整型、浮点类型以及复数类型。

整数类型（`int`）用来表示从-2147483648 到 2147483647 之间的任意整数（在某些电脑系统上这个范围可能会更大，但绝不会比这个更小）；长整型（`long`）可以表示任意范围的整数，**实际上我们把 Python 的 `long` 和 `int` 理解为同一种类型就可以了，因为当一个整数超过 `int` 的范围后，Python 会自动将其升级为长整型。**

Python 中只有 64 位双精度浮点数 `float`，与 java 中的 `double` 类型相同（注意在 Python 中浮点数类型名字是 `float` 而不是 `double`），Python 不支持 32 位单精度的浮点数。

除了整数和实数，Python 还提供了一种特殊类型复数（`complex`）。复数使用一对浮点数表示，复数 `z` 的实部和虚部分别用 `z.real` 和 `z.imag` 访问。

在数值运算中，整数与浮点数运算的结果是浮点数，这就是所谓的“提升规则”，也就是“小”类型会被提升为“大”类型参与计算。这一点和 java 是一样的，**提升的顺序依次为：int long float complex**

作为数值类型的最后一个问题，java 程序员需要注意的是，**Python 没有内建的 `decimal` 类型**，但可以导入 `decimal` 模块用来完成与货币相关的计算。

Python 中序列是由非负整数索引的对象的有序集合，它包括字符串、Unicode 字符串，列表、元组、`xrange` 对象以及缓冲区对象。

## 6) 字符串类型

Python 拥有两种字符串类型：标准字符串（`str`）是单字节字符序列，Unicode 字符串（`unicode`）是双字节字符序列。

在 Python 中定义一个标准字符串（`str`）可以使用单引号、双引号、三引号，这使得 Python 输入文本比 java 更方便。比如当 `Str` 的内容中包含双引号时，就可以用单引号定义，反之亦然。当字符中有换行符等特殊字符时，可以直接使用三引号定义。这样就方便了很多不用去记那些乱七八糟的转义字符。当然 Python 也支持转义字符，且含义与 java 基本一样。

下面是一个例子来说明这一点

```
#!/usr/bin/python
#-*-coding:utf-8-*-
"""
我的第一个应用程序
"""
import sys
def Main():
    sys.stdout.write("开始程序\n")
    str1='i am "python"\n'
    str2="i am 'Python' \n"
    str3="""
        i'm "Python",
        <a href="http://www.sina.com.cn"></a>
    """
    #你可以把 html 之类的东西都直接弄进来而不需要处理
    print str1,str2,str3
if __name__=="__main__":
```

```
Main()
```

在 Python 中定义一个 Unicode 字符串，需要在引号前面加上一个字符 `u`，例如

```
#-*-coding:utf-8-*-
import sys
def Main():
    print u"我是派森"
if __name__=="__main__":
    Main()
```

同时注意，当使用 UTF-8 编码时，非 unicode 字符中一个汉字的长度是 3，而使用 gb2312 时是 2，见下边代码：

```
#-*-coding:utf-8-*-
...
test
...
import sys
def Main():
    print u"我是派森"
    unicode =u'我'
    str='我'
    print len(unicode),len(str)
    #输出的 1 3
if __name__=="__main__":
    Main()
```

## 7) 全局变量

不加入 `global` 关键字的话，如果你是在函数内部定义的变量的话，在外部是无法获取的。

```
#coding:utf-8
def testList():
    global list
    list=[]
    list.append("test1")
    list.append("test2")
    print list
testList()
print list
```

## 三、 基本数据类型-列表(list 【】 )

Python 中列表 (list) 类似于 java 中的 ArrayList,用于顺序存储结构。列表用符号 `[]` 表示，中间的元素可以是任何类型（包括列表本身，以实现多维数组），元素之间用逗号分隔。取值或赋值的时候可以像 C 数组一样，按位置索引：

```
#-*-coding:utf-8-*-
import sys
```

```
def Main():
    array=[1,2,3]
    print array[0]
    #输出 1
    array[0]='a'
    print array
    #输出['a',2,3]
    L=[123,'spam',1.23]
    #输出大小
    print len(L)
    print L[0]
    print L[:-1]#不包含最后一个
    print L+[4,5,6]#重新拼接一个新的列表
if __name__=="__main__":
    Main()
```

从上边的代码中你可能发现一个有趣的事情：在 Python 的列表中可以混合使用不同类型的数据，像 ['a',2,3] 这样，不过我不建议你这样做，我觉着没有什么好处。

另外还可以看到，列表是可变的序列，也就是说我们可以在“原地”改变列表上某个位置所存储的对象的值。

Python 中的 list 支持多数的操作，同时 list 也支持“切片”这样的操作。切片指的是抽取序列的一部分，其形式为：list[start:end:step]。其抽取规则是：从 start 开始，每次加上 step，直到 end 为止。默认的 step 为 1；当 start 没有给出时，默认从 list 的第一个元素开始；当 end=-1 时表示 list 的最后一个元素，依次类推。一些简单的例子见下边代码：

```
#-*-coding:utf-8-*-
import sys
def Main():
    test=['never',1,2,'yes',1,'no','maybe']
    print test[0:3]#包括 test[0],不包括 test[3]
    print test[0:6:2]#包括 test[0],不包括 test[6],而且步长为 2
    print test[:-1]#包括开始，不包括最后一个
    print test[-3:]#抽取最后 3 个
if __name__=="__main__":
    Main()
```

字符串、列表、元组都支持切片操作，这个很方便，应该学会熟练使用它。

最后，list 是 Python 中最基础的数据结构，你可以把它当作链表、堆栈或队列来使用，效率还不错。Python 中没有固定长度数组，如果你确实很在意性能，可以改入 array 模块来创建一个 C 风格的数组，它的效率很高，这里就不详细介绍了。

我们还可以对其进行排序与反转

```
#-*-coding:utf-8-*-
import sys
def Main():
    array=[5,2,3,1,8]
    array.sort()
    for s in array:
        print s
    array.reverse()
    for s in array:
        print s
```

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

Python 核心数据类型的一个优秀的特性就是它们支持任意的嵌套。能够以任意的组合对其进行嵌套。这种特性的一个直接应用就是实现矩阵，或者 Python 中的多维数组。一个嵌套列表的组表能够完成这个基本的操作。

```
#-*-coding:utf-8-*-  
import sys  
def Main():  
    M=[[1,2,3],  
        [4,5,6],  
        [7,8,9]]  
    print M[0]  
    print M[1]  
    print M[2]  
if __name__=="__main__":  
    Main()
```

处理序列的操作和列表的方法中，Python 还包括了一个更高级的操作，称作列表解析表达式，从而提供了一种处理像矩阵这样结构的强大工具。列如，假设我们需要从列举的矩阵中提取出第二列。因为矩阵是按照行进行存储的，所以可以通过简单的索引即可获得行，使用列表解析可以同样简单地获得列。

```
#-*-coding:utf-8-*-  
import sys  
def Main():  
    M=[[1,2,3],  
        [4,5,6],  
        [7,8,9]]  
    col2=[row[1] for row in M]  
    print col2  
    col3=[row[1]+1 for row in M]  
    print col3  
    colfilter=[row[1] for row in M if row[1]%2==0]  
    print colfilter  
if __name__=="__main__":  
    Main()
```

## 四、 基本数据类型-字典(dict { })

用过 java 中的字典的人对 Hashtable 应该不会陌生，Python 里的哈希表就是字典(dict)了。与 set 类似，字典是一种无序存储结构，它包括关键字(key)和关键字对应的值(value)。

java 程序员需要了解的就是,在 Python 中 dict 是一种内置的数据类型，定义方式为：

```
dictionary={key:value}#当有多个键值时，用逗号进行分割。
```

字典里的关键字为不可变类型，如字符串、整数、只包含不可变对象的元组，列表等不能作为关键字。字典中一个键只能与一个值关联，对于同一个键，后添加的值会覆盖之前的值。

学过数据结构的人对字典的散列查找效率应该都有认识，所以我建议在可能的情况下尽量多用字典，其它的不多写了。



```
#-*-coding:utf-8-*-
import sys

if __name__=="__main__":
    dict={"a":"apple","b":"banana","g":"grape","o":"orange"}
    print dict
    print dict["a"]
    dict2={1:"apple",2:"banana",3:"grape",4:"orange"}
    print dict2
    print dict2[1]
```

```
#-*-coding:utf-8-*-
import sys

if __name__=="__main__":
    #字典的添加、删除、修改操作
    dict={"a":"apple","b":"banana","g":"grape","o":"orange"}
    dict["w"]="watermelon"
    print dict
    del(dict["a"])
    print dict
    print dict.pop("b")
    #dict.clear()
    #print dict
    #字典的遍历
    for k in dict:
        print "dict[%s]="%k,dict[k]
```

```
#-*-coding:utf-8-*-
import sys

if __name__=="__main__":
    #字典的 keys()与 values()方法
    dict={"a":"apple","b":"banana","g":"grape","o":"orange"}
    #输出 key 的列表
    print dict.keys()
    #输出 values 的列表
    print dict.values()
```

```
#-*-coding:utf-8-*-
import sys
def Main():
    D={'food':'spam','quantity':4,'color':'pink'}
    print D['food']
    D['quantity']+=1
    print D
    #另外一种定义字典的方法
```

```
D={}
D['name']='Bob'
D['job']='dev'
D['age']=40
print D
#使用键值,进行排序
D={'a':1,'b':2,'c':3}
print D
Ks=D.keys()
print Ks
Ks.sort()
print Ks
for key in Ks:
    print key,'=>',D[key]
for key in sorted(D):
    print key,'=>',D[key]
#迭代与优化
squares=[x ** 2 for x in [1,2,3,4,5]]
print squares
#与以下代码是等效的
squares=[]
for x in [1,2,3,4,5]:
    squares.append(x**2)

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

在访问的时候，如果这个键值不存在的话，如果我们没有做任何判断的话，会出现错误，这个时候我们可以用以下代码来进行判断。

```
#!/usr/bin/env python
#-*-coding:utf-8-*-
import sys
def Main():
    D={'food':'spam','quantity':4,'color':'pink'}
    #测试不存在的键值
    if not D.has_key('f'):
        print '不存在这个键值'
    else:
        print D['f']

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

## 五、基本数据类型-元组(tuple ( ))

元组与列表非常相似，它用()而不是[]括起来的序列。元组比列表的速度更快，但元组是一个不可变的序列，也就是与 `str` 一样，无法在原位改变它的值。除此之外，其他属性与列表基本一致。

元组是 Python 中内置的一种数据结构。元组由不同的元素组成，每个元素可以存储不同类型的数据，如字符串、数字甚至元组。**元组是写保护的！**即元组创建后不能再做任何修改操作，元组通常代表一行数据，而元组中的元素代表不同的数据项。

元组的创建

```
#-*-coding:utf-8-*-
import sys
if __name__=="__main__":
    tuple_name=("apple","banana","grape","orange")
    print tuple_name[0]
```

分片输出

```
#-*-coding:utf-8-*-
import sys
if __name__=="__main__":
    tuple_name=("apple","banana","grape","orange")
    print tuple_name[1]
    #牛在支持分片输出
    print tuple_name[-1]
    print tuple_name[-2]
    print tuple_name[1:3]
    #我还可以在元组中包含自己
    print '-----'
    tuple= (('t1','t2'),('t3','t4'))
    print tuple[0][0]
    print tuple[1][0]
```

实现解包的功能

```
#-*-coding:utf-8-*-
import sys
if __name__=="__main__":
    tuple_name=("apple","banana","grape","orange")
    a,b,c,d=tuple_name
    print a,b,c,d
```

元组定义的方法与列表类似，不过在定义只包含一个元素的元组时，注意在后边加一个逗号，请体会以下几句的差异：

```
#-*-coding:utf-8-*-
import sys
def Main():
    test=[0]                #列表可以这样定义
    print type(test)        #输出<type 'list'>
```

```
test=[0,]          #也可以这样定义
print type(test)    #输出<type 'list'>
test=(0,)           #元组可以这样定义
print type(test)    #输出<type 'tuple'>
test=(0)            #但不能这样定义，Python 会认为它是一个括号表达式
print type(test)    #输出<type 'int'>
test=0,             #也可以省略括号，但要注意与 C 的逗号表达式不同
print type(test)    #输出<type 'tuple'>
#还可以简单的交换数据
a=1
b=2
a,b=b,a
print a,b
if __name__=="__main__":
    Main()
```

以上这类语句在 Python 中被广泛应用于变量交换、函数传值等应用，因此 Python 的解析器在不断对其进行优化，现在已经具备了相当高的效率。所以以上代码在 Python2.5 以后的版本中，比 `tmp=a;a=b;b=tmp` 这种常规语句更快。

- Python 是一种动态的强类型语言，在使用变量之前无须定义其类型，但是必须声明和初始化
- “一切命名是引用”，Python 中变量名是对象的引用，同一变量名可以在程序运行的不同阶段代表不同类型的数据
- “一切数据都是对象”，Python 的所有数据类型都是对象，相较 java 具有一致的使用方法
- 把问题想得更简单一点，Python 的数值类型可以说只有两种：整形和浮点
- 多使用 list/tuple/set/dict 这几种很 pythonic 的数据类型，它们分别用 `[]/()/[]/{}`  定义

## 六、基本数据类型-集合(set)

Python 中的 set 和 java 中的集合不是一个概念，这是翻译的问题，Python 中的集合是指无序的、不重复的元素集，类似数学中的集合概念，可对其进行交、并、差、补等逻辑运算。

常见集合的语法为：`s=set(['a','b','c'])`。不过 set 在 Python3.0 中发生了较大的变化，创建一个集合的语法变成了：`S={1,2,3}`，用花括号的方法，与后边要提到的 dict 类似。

如果在 set 中传入重复元素，集合会自动将其合并。这个特性非常有用，比如去除列表里大量的重复的元素，用 set 解决效率很高。示例如下：

```
#-*-coding:utf-8-*-
import sys
def Main():
    a=[133,224,2344,2243,22342,224,133,133,989]
    b=set(a)
    print b
if __name__=="__main__":
    Main()
```

另一个例子，找出两个 list 里面相同的元素(集合求交，其它类推)，代码如下：

```
#-*-coding:utf-8-*-
import sys
def Main():
```

```
a=[133,224,2344,2243,22342,224,133,133,989]
b=set(a)
print b
a=["11","22","33"]
b=["11","33"]
c=set(a)&set(b)
print c
if __name__=="__main__":
    Main()
```

想想你如果自己实现这个算法会怎么写？然后可以找两个大一点的列表，比比和 `set` 实现的效率，你就会有体会了，在以后的程序中多用 `set` 吧。

## 七、运算符、表达式和流程控制

本章介绍 Python 的运算符、表达式、程序流程控制语句以及异常处理语句，在这方面，Python 和 java 是非常类似的，我们仅需要注意它们之间的一些细微差异。另外，在本章我还会简要介绍 Python 语言中的两项有趣功能=列表内涵和动态表达式，虽然它们严格来说属于函数部分的内容，不过我觉得还是放在表达式一章比较合适。

无论使用什么语言，我们编写的大多数代码都包含表达式。一个表达式可以分解为运算符和操作数，运算符的功能是完成某件事，它们由一些数学运算符或者其他特定的关键字表示，运算符需要数据来进行运算，这样的数据被称为操作数。例如，`2+3` 是一个简单的表达式，其中`+`是运算符，`2` 和 `3` 是操作数。

### 1) 算术运算符与算术表达式

算术运算符是程序设计语言最基本的运算符。Python 提供的算术运算符除了`+`、`-`、`*`、`/`、`%`（求余）之外，还提供了两种 java 中没有提供的运算符：求幂(`**`)和取整除(`//`)。下面我们就通过一段代码来解析这两个算术运算符的功能。

```
#-*-coding:utf-8-*-
import sys
def Main():
    x=3.3
    y=2.2
    a=x**y
    print a
    #输出即 3.3 的 2.2 次幂
    b=x//y
    print b
    #输出 1.0 取整除返回商的整数部分
    c=x/y
    print c
    #输出 1.5，注意体会普通除与取整除的区别
if __name__=="__main__":
    Main()
```

## 2) 赋值运算符与赋值表达式

赋值就是给一个变量赋一个新值，除了简单的=赋值之外，Python 和 java 都支持复合赋值，例如 `x+=5`，等价于 `x=x+5`。

Python 不支持 java 中的自增和自减运算符，例如 `X++` 这种语句在 Python 中会被提示语法有错误。java 程序员可能习惯了这种表达式，在 Python 中，请老老实实的写 `X+=1` 就是了。

Python 的逻辑运算符与 java 有较大区别，Python 用关键字 `and` `or` `not` 代替了 java 语言中的逻辑运算符 `&&` `||` `!`，此外 Python 中参与逻辑运算符的操作数不限于布尔类型，任何类型的值都可以参与逻辑运算中去。

用逻辑运算符将操作数或表达式连接起来就是逻辑表达式。与 java 一样，Python 中逻辑表达式是短路执行的，也就是说只有需要时才会进行逻辑表达式右边值的计算，例如表达式 `a and b` 只有当 `a` 为 `true` 时才计算 `b`。思考一下，`if(0 and 10/0)`：这条语句会引发除数为零的异常吗？

此外还要注意：在 Python 中，`and` 和 `or` 所执行的逻辑运算并不返回布尔值，而是返回它们实际进行比较的值之一。下边是一个例子：

```
#-*-coding:utf-8-*-
import sys
def Main():
    print 'a' and 'b'
    print '' and 'b'
if __name__=="__main__":
    Main()
```

## 3) 关系运算符与关系表达式

关系运算符实际上是逻辑运算的一种，关系表达式的返回值总是布尔值。Python 中的比较操作符与 java 完全是一样的，包括 `==` `!=` `>` `<` `>=` `<=` 共 6 种。

除了基本的变量比较外，Python 的关系运算符还包括身份运算符 `is`。在 Python 中，`is` 用来检验两个对象在内存中是否指向同一个对象（还记得一切数据皆对象吗？一切命名皆引用吗）。

三元运算符

三元运算符是 c/c++ 系语言所特有的一类运算符，例如，对表达式 `b?x:y`，先计算条件 `b`，然后进行判断，如果 `b` 的值为 `true`，则计算并返回 `x` 的值，否则计算并返回 `y` 的值。

在 Python 中，提供了专门的逻辑分支表达式来模拟 java 系中的三元运算，我们也可以在一行语句中完成三元运算，例如

```
print '偶数' if x%2==0 else '奇数'
```

# 八、 流程控制

## 1) 条件语句

Python 用 `if`, `elif`, `else` 这三个关键字进行条件判断，与 java 唯一的区别就是用 `elif` 取代 `else if`，少打两个字，其它都一样，此外别忘了在 `if` 等语句后加：号

如果一个流程控制分支南下不做任何事情，记得写一句 `pass` 语句，不然 Python 会报错。例如：

```
if 0:
```

```
pass #这一句语没有什么意义
```

在 Python 中没有 `switch` 语句，你可以使用 `if..elif..else` 语句来完成同样的工作。如果你觉得繁琐，可以试试 `dict` 实现的方式，下边是一个例子，分别对比了两种实现方式。

```
#-*-coding:utf-8-*-
import sys
def Main():
    #使用 if 替代
    x='4'
    print "OK"
    if x=='1':
        print 'one'
    elif x=='2':
        print 'two'
    else:
        print 'nothing!'
    #使用 dict
    numtrans={
        1:'one',
        2:'two',
        3:'three'
    }
    try:
        print numtrans[x]
    except KeyError:
        print 'nothing!'
if __name__=="__main__":
    Main()
```

## 2) 循环

Python 支持两种循环语句 `while` 循环和 `for` 循环，不支持 `java` 中的 `do-while` 循环。在 Python 的 `while` 循环和 `Java` 基本一致，此处我们着重比较两种语言中的 `for` 循环的区别。

说的简单一点，python 中的 `for` 语句相当于 `java` 中的 `foreach` 语句，它用于从集合对象（`list/str/tuple` 等）中遍历数据。例如：

```
for I in [1,2,3,4,5]:
    print i
for I in range(10):
    print i
```

```
#-*-coding:utf-8-*-
import sys
if __name__=="__main__":
    tuple=(("apple","banana"),("grape","orange"))
    for i in range(50,100+1):
        print i
```

```
#-*-coding:utf-8-*-
import sys
if __name__=="__main__":
    for i in range(50,100+1):
        print i
    range(1,5) #代表从 1 到 5(不包含 5)
    range(1,5,2) #代表从 1 到 5，间隔 2(不包含 5)
    range(5) #代表从 0 到 5(不包含 5)
```

```
__author__ = 'TenYear'
#-*-coding:utf-8-*-
"""
my fist App
"""
import sys
import urllib
def Main():
    itemlist=[1,2,3,4,5,4]
    for m in itemlist:
        print m
#this is test
if __name__=="__main__":
    Main()
```

## 九、 异常

Python 和 java 一样支持异常处理，利用 **try/except/finally** 结构，可以很方便的捕获异常，同时可以用 **raise** 语句手动抛出异常（上述四个异常处理的关键字分别对应 Java 中的 **try/catch/finally/throw**）。通过 **except**，您可以将 **try** 标示的语句中出现的错误和异常捕获。

```
#-*-coding:utf-8-*-
import sys
def Main():
    try:
        f=open('firstpython.py')
        s=f.readline()
        print s
    except IOError,(errno,strerror):
        print "I/O error(%s):%s" %(errno,strerror)
    except ValueError:
        print "Could not convert data to an integer"
    except:
        print "Unexpected error:",sys.exc_info()[0]
        raise
    finally:
```



```
f.close()
if __name__=="__main__":
    Main()
```

最后说明一点，python 的 try 也支持 else 语句，如果有一些代码要在 try 没有发生异常的情况下才执行，就可以把它放在 else 中。

## 十、 动态表达式

在 C#语言中，如果需要在文本框中输入 1+2（或更加复杂的数学表达式）后计算它的值，可能会用到表达式解析等。现在我们有 Python，要完成这种任务可以说是非常简单：只要用内置的 eval() 函数，就可以计算并返回任意有效表达式的值。例如：

```
str='1+2'
print eval(str)
```

除了 eval 函数之外，Python 还提供了 exec 语句将字符串 str 当成有效 Python 代码来执行，看下面的例子：

```
exec 'a=100'
print a
```

另外还有 execfile 函数，它用来执行一个外部的 py 文件，上一个例子存为 exec.py 后，运行下边的代码就知道是怎么回事了：

```
execfile(r'c:\exec.py')
```

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

列表内涵是 Python 最强有力的语法之一，常用于从集合对象中有选择地获取并计算元素，虽然多数情况下可以使用 for/if 等语句组合完成同样的任务，但列表内涵书写的代码更加简洁。

```
#-*-coding:utf-8-*-
import sys
def func():
    x=1
    y=2
    m=3
    n=4
    sum=lambda x,y:x+y
    print sum
    sub=lambda m,n:m-n
    print sub
    return sum(x,y)*sub(m,n)
if __name__=="__main__":
    print func()
```

列表内涵的一般形式如下，我们可以把 [] 内的列表内涵写为一行，也可以写为多行。

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

上面的表达式分为三部分，最左边是生成每个元素的表达式，然后是 for 迭代过程，最右边可以设定一个 if 判断作为过滤条件。

列表内涵的一个著名例子是生成九九乘法表：

```
s=[(x,y,x*y) for x in range(1,10) for y in range(1,10) if x>=y]
```

## 十一、 函数及函数编程

在 **java** 中没有独立的函数存在，只有类的（动态或静态）方法这一概念，它指的是类中用于执行计算或其它行为的成员。在 **Python** 中，你可以使用类似 **C#** 的方式定义类的动态或静态成员方法，因为它与 **java** 一样支持完全的面向对象编程。你也可以用过程式编程的方式来编写 **Python** 程序，这时 **Python** 中的函数与类可以没有任何关系，类似 **C** 语言定义和使用函数的方式。此外，**Python** 还支持函数式编程，虽然它对函数编程的支持不如 **LISP** 等语言那样完备，但适合使用还是可以提高我们工作效率的。

### 1) 函数的定义

函数定义是最基本的行为抽象代码，也是软件复用最初级的方式。**Python** 中函数的定义语句由 **def** 关键字、函数名、括号、参数及冒号组成。下面是几个简单的函数定义语句：

```
#-*-coding:utf-8-*-
import sys
def Main():
    def F1():
        print "我是 F1"
    def F2(x,y):
        a=x+y
        return a
    #定义有多个返回值的函数，用逗号分割不同的返回值，
    #返回结果是一个元组
    def F3(x,y):
        a=x/y
        b=x%y
        return a,b
if __name__=="__main__":
    Main()
```

可能你已经注意到了，**Python** 定义函数的时候并没有约束参数的类型，它以最简单的形式支持了泛型编程。你可以输入任意类型的数据作为参数，只要这些类型支持函数内部的操作

### 2) 定义一个交换函数

```
#-*-coding:utf-8-*-
import sys
import random
def compareNum(num1,num2):
    if(num1>num2):
        return 1
    elif(num1==num2):
        return 0
```

```
else:
    return -1
if __name__=="__main__":
    num1=random.randrange(1,9)
    num2=random.randrange(1,9)
    print "num1=",num1
    print "num2=",num2
    print compareNum(num1,num2)
```

### 3) 函数的默认参数与返回值

```
#-*-coding:utf-8-*-
import sys
def arithmetic(x=1,y=1,operator="+"):
    result={
        "+":x+y,
        "-":x-y,
        "*":x*y,
        "/":x/y
    }
    return result.get(operator)
if __name__=="__main__":
    print arithmetic(1,2)
    print arithmetic(1,2,"-")
```

### 4) 返回多个值

Python 支持多个返回值

```
#coding:utf-8
def testList():
    return "aaa","bbb"
a,b=testList()
print a
print b
```

### 5) locals 函数和 globals 函数介绍

Python 有两个内置的函数，`locals()` 和 `globals()`，它们提供了基于字典的访问局部和全局变量的方式。首先，是关于名字空间的一个名词解释。是枯燥，但是很重要，所以要耐心些。Python 使用叫做名字空间的东西来记录变量的轨迹。名字空间只是一个字典，它的键字就是变量名，字典的值就是那些变量的值。实际上，名字空间可以象 Python 的字典一样进行访问，一会我们就会看到。

在一个 Python 程序中的任何一个地方，都存在几个可用的名字空间。每个函数都有着自己的名字空间，叫做局部名字空间，它记录了函数的变量，包括 函数的参数和局部定义的变量。每个模块拥有它自己的名字空间，叫做

全局名字空间，它记录了模块的变量，包括函数、类、其它导入的模块、模块级的变量和常量。还有就是内置名字空间，任何模块均可访问它，它存放着内置的函数和异常。

当一行代码要使用变量 `x` 的值时，Python 会到所有可用的名字空间去查找变量，按照如下顺序：

1. 局部名字空间 - 特指当前函数或类的方法。如果函数定义了一个局部变量 `x`，Python 将使用这个变量，然后停止搜索。
2. 全局名字空间 - 特指当前的模块。如果模块定义了一个名为 `x` 的变量，函数或类，Python 将使用这个变量然后停止搜索。
3. 内置名字空间 - 对每个模块都是全局的。作为最后的尝试，Python 将假设 `x` 是内置函数或变量。

如果 Python 在这些名字空间找不到 `x`，它将放弃查找并引发一个 `NameError` 的异常，同时传递 `There is no variable named 'x'` 这样一条信息，象 Python 中的许多事情一样，名字空间在运行时直接可以访问。特别地，局部名字空间可以通过内置的 `locals` 函数来访问。全局（模块级别）名字空间可以通过 `globals` 函数来访问

```
#coding:utf-8
from __future__ import unicode_literals
from sys import *
def test(arg):
    #函数 test 在它的局部名字空间中有两个变量:arg(它的值被传入函数)
    # 和 z(它在函数内部)
    z=1
    print locals()
#locals 返回一个名字与值的字典，这个字典的键字是字符串形式的变量名
#字典的值是变量的实际值
test(5)
test('楚广明')
print globals()#返回一个全局的字典，包括所有导入的变量
```

## 十二、 类及面向对象

如果你熟悉 java 或者 c#,那么对类和面向对象应该不会陌生。Python 与 c#一样，能够很好的支持面向对象的编程模式。

### 1) 类的定义

与 C#一样，Python 使用 `class` 关键字定义一个类。一个最简单的类定义语句如下：

```
Class A:
    Pass
```

它等价于 C#中的 `class A{}`。当然以上语句没有任何实际意义，它只是告诉我们什么是定义一个类所必需的。即 `class` 关键字，类名和冒号，**pass 关键字只用来占位，相当于用来占位，相当于 C#中花括号的作用。**

类是定义对象格式的模板，而对象则是类的实例，通过类创建对象的过程称为类的实例化。在 C#中，需要使用 `new` 关键字实例化一个类，例如

```
A a=new A();
```

在上条语句，C#完成了两件事，首先声明一个类型为 A 的变量 a，然后用 new 运算符创建一个类型为 A 的对象，并将该对象的引用赋值给变量 a，而在 python 中没有 new 关键字，同时它是一个动态语言，不需要事先指定变量的类型，只需要：

```
a=A()
```

即创建一个类型为 A 的对象，看起来好像是将类当作一个函数调用，返回值是新创建的对象。

## 2) 为类添加数据

通常我们利用类来定义各种新的数据类型，其中即包含数据内容，也包含对数据内容的操作。Python 类的数据添加方法与 C#有一些不同，因为 Python 是一种动态语言，变量在使用之前不需要定义，所以你可以不在类定义中添加成员变量，而是在运行时动态地添加它们，例如：

```
#!/usr/bin/env python
#-*-coding:utf-8-*-
import sys
def Main():
    class A:pass
    a=A()
    a.x=1
    print a.x
if __name__=="__main__":
    Main()
```

## 3) 构造函数

Python 的类提供了类似 C#构造函数的东西：\_\_init\_\_(注意是前后是两个下划线)，类在实例化时会首先调用这个函数，我们可以通过重写\_\_init\_\_函数，完成变量的初始化等工作。与 C#不同的地方是，Python 不支持无参数的初始化函数，你至少需要为初始化函数指定一个参数，即对象实例本身(self)。下面是一段简单的代码。在该代码中，我们重写了函数\_\_init\_\_，定义并初始化一个类的成员变量 X：

```
#!/usr/bin/env python
#-*-coding:utf-8-*-
import sys
def Main():
    class A:
        def __init__(self):
            self.x=1
    a=A()
    print a.x
if __name__=="__main__":
    Main()
```

```
#!/usr/bin/env python
#-*-coding:utf-8-*-
import sys
class Student:
    def __init__(self,name,age):
        self.__name=name
        self.__age=age
    def getName(self):
        format="my name is %s my age is %d"%(self.__name,self.__age)
```

```
print format
def __del__(self):
    print "del"
if __name__=="__main__":
    student=Student("chu",35)
    student.getName()
```

## 4) 静态成员与私有成员

```
#!/usr/bin/python
#-*-coding:utf-8-*-
import sys
class A:
    #定义静态成员变量
    y=2
    def __init__(self):
        #定义公共的成员变量
        self.x=1
        #定义私有的成员变量
        self.__z=1
if __name__=="__main__":
    a=A()
    #打印公共的成员变量
    print a.x
    #打印静态成员变量
    print A.y
    #打印私有成员变量出错
    #print a.__z
```

## 5) 为类添加方法

方法是对类数据内容的操作，在 Python 中，定义类的方法与定义一个普通的函数在语法上基本上相同，c#程序员需要注意的是，在类中定义的常规方法的第一个参数总是该类的实例，即 **self**，同时注意在方法中引用类的另一个方法必须使用类名加方法名的形式，下面是一个定义类的方法的简单的例子。

```
#!/usr/bin/python
#-*-coding:utf-8-*-
import sys
class A:
    def prt(self):
        print "my name is A"
    def reprt(self):
        A.prt(self)
if __name__=="__main__":
    a=A()
    a.prt()
    a.reprt()
```

不能定义一个不操作实例的方法

```
#-*-coding:utf-8-*-
import sys
class Student:
    __name=""
    def __init__(self,name):
        self.__name=name
    def getName(self):
        return self.__name
if __name__=="__main__":
    student=Student("chu")
    print student.getName()
```

## 6) 静态方法

Python 与 C#一样支持静态方法。在 C#中需要使用关键字 **static** 声明一个静态方法，而在 Python 中是通过静态方法修饰符 **@staticmethod** 来实现的，下面是例子：

```
#-*-coding:utf-8-*-
import sys
class A:
    def prt(self):
        print "my name is A"
    def rept(self):
        A.prt(self)
    @staticmethod
    def prt2():
        print "我是静态方法"
if __name__=="__main__":
    a=A()
    a.prt()
    a.reprt()
    A.prt2()
```

如你所见，静态方法可以直接被类调用，它没有常规方法那样的特殊行为（默认第一行参数是 **self** 等），你完全可以将静态方法当成一个用属性引用方式调用的普通函数。

## 7) 单继承

Python 用类名后加扩号的方式实现继承，下面是一个简单的示例：

```
#-*-coding:utf-8-*-
import sys
class A:
    x=1
class B(A):
    y=2
```

```
if __name__=="__main__":  
    print B.x  
    print B.y
```

一个复杂的实例

```
#!/usr/bin/python3  
#-*-coding:utf-8-*-  
import sys  
class SchoolMember:  
    '''Represents any school member.'''  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
        print'(Initialized SchoolMember: %s)'% self.name  
  
    def tell(self):  
        '''Tell my details.'''  
        print'Name:"%s" Age:"%s"'% (self.name, self.age),  
  
class Teacher(SchoolMember):  
    '''Represents a teacher.'''  
    def __init__(self, name, age, salary):  
        SchoolMember.__init__(self, name, age)  
        self.salary = salary  
        print'(Initialized Teacher: %s)'% self.name  
  
    def tell(self):  
        SchoolMember.tell(self)  
        print'Salary: "%d"'% self.salary  
  
class Student(SchoolMember):  
    '''Represents a student.'''  
    def __init__(self, name, age, marks):  
        SchoolMember.__init__(self, name, age)  
        self.marks = marks  
        print'(Initialized Student: %s)'% self.name  
  
    def tell(self):  
        SchoolMember.tell(self)  
        print'Marks: "%d"'% self.marks  
  
if __name__=="__main__":  
    t = Teacher('Mrs. Shrividya',40,30000)  
    s = Student('Swaroop',22,75)
```



## 十三、 模块与包

Python 的脚本都是用扩展名为 `py` 的文本文件保存的，一个脚本可以单独运行，也可以导入另一个脚本中运行。当脚本被导入运行时，我们将其称为模块（`module`）。模块是 Python 组织代码的基本方式。

Python 的程序是由包（`package`）、模块（`module`）和函数组成。包是由一系列模块组成的集合。模块是处理某一类问题的函数和类的集合。

包就是一个完成特定任务的工具箱，Python 提供了许多有用的工具包，如字符串处理、图形用户接口、WEB 接口、图形图像处理等。使用这些工具包，可以提高程序员的开发效率、减少编程的复杂度、达到代码重用的效果。这些自带的工具包和模块安装在 Python 的安装目录下的 `Lib` 子目录中。

例如，`Lib` 目录中的 `xml` 文件夹就是一个包，这个包用于完成 XML 的应用开发。`Xml` 包中有几个子包：`dom`、`sax`、`etree` 和 `parser`。文件 `__init__.py`（注意是两个下滑线）是 `xml` 包的注册文件，如果没有该文件，Python 将不能识别 `xml` 包。在系统字典中定义了 `xml` 包。

包必须至少含有一个 `__init__.py` 文件，该文件的内容可以为空。`__init__.py` 用于表示当前文件夹是一个包。

### 1) 对于模块的理解

用简单的说法来说，每一个以扩展名为 `.py` 结尾的 Python 源代码都是一个模块。其他的文件可以通过导入一个模块读取这个模块的内容。导入从本质上来讲，就是载入另一个文件，并能够读取那个文件的内容。一个模块的内容通过这样的属性能够被外部世界使用。

比如我们举一个简单的例子，首先建立一个 python 文件 `define.py`

```
#-*-coding:utf-8-*-  
myvar="这是一个测试"
```

很简单的代码就二行，我们定义了一个变量 `myvar`。现在我们通过另一个 python 文件来导入它。

```
#-*-coding:utf-8-*-  
import sys  
import define  
def Main():  
    print define.myvar  
    print dir(define)  
    print define.__file__  
if __name__=="__main__":  
    Main()
```

```
import define #这一行语句的意义在于导入 define.py 这个模块
```

```
print define.myvar#这一行的意义在于打印 myvar 这个变量
```

```
print dir(define)#这一行的意义在于输出所有可以使用的变量
```

输出结果，不言而喻。但是在默认情况下，只是在每次会话的第一次运行。在第一次导入之后，其他的导入都不会再工作。

但是如果真的想要 python 在同一次会话中再次运行文件（不停止和重新启动会话），需要调用内置的 `reload` 函数。

## 2) 模块的显要特性：属性

导入和重载提供了一种自然的程序启动的选择，因为导入操作将会在最后一步执行文件。从更宏观的角度来看，模块扮演了一个工具库的角色。我们可以直接使用 `from define import myvar` 这条语句来实现。

```
#-*-coding:utf-8-*-
import sys
from define import myvar
def Main():
    print myvar
if __name__=="__main__":
    Main()
```

## 3) 模块创建过程的例子

模板把一组相关的函数或代码组织到一个文件中。一个文件即是一个模板。模块由代码、函数或类组成。例如，创建一个名为 `myModule.py` 的文件，即定义了一个名为 `myModule` 的模块。在该模块中定义一个函数 `func()` 和一个类 `MyClass`。 `MyClass` 类中定义一个方法 `myFunc()`。

```
#-*-coding:utf-8-*-
import sys
def func():
    print "myModule.func()"
class MyClass:
    def myFunc(self):
        print "myModule.MyClass.myFunc()"
```

然后在 `myModule.py` 所在目录下创建一个名为 `call_myModule.py` 文件，在该文件中调用 `myModule` 模块的函数和类：

```
#-*-coding:utf-8-*-
import sys
import random
import myModule
if __name__=="__main__":
    myModule.func()
    myClass=myModule.MyClass()
    myClass.myFunc()
```

当 `python` 寻入一个模块时，`python` 首先查找当前路径，然后查找 `lib` 目录、`site-packages` 目录 (`python/lib/site-packages`) 和环境变量 `PYTHONPATH` 设置的目录。

## 4) 模块的导入

在使用一个模块中的函数或类之前，首先要导入该模块。

```
import myModule
```

还可以使用 `from ...import..` 语句将模块导入。

```
from module_name import *
from module_name import function_name
```

#### 实现例子

```
#-*-coding:utf-8-*-
import sys
def func():
    print "myModule.func()"
class MyClass:
    def myFunc(self):
        print "myModule.MyClass.myFunc()"
```

```
#-*-coding:utf-8-*-
import sys
import random
from myModule import func
if __name__=="__main__":
    func()
```

## 5) 模块的属性

模块中有许多内置的属性，用于完成特定的任务，如 `__name__` `__doc__`。每个模块都有一个名称。

```
#-*-coding:utf-8-*-
import sys
if __name__=="__main__":
    print __name__
    print __doc__
    print __file__
    print __package__
```

## 十四、 字符串与简单的正则表达式

本章将介绍 Python 中字符串和正则表达式的概念。字符串是程序开发中常用的数据类型，字符串的处理是实际应用中经常面对的问题。Python 提供了功能强大的字符串模块，正则表达式专门用于匹配应用中的数据，能够简化字符串的处理程序，Python 提供了 re 模块用来匹配正则表达式。

如果我们有一个含有 12 个字符的字符串，我们通过内置的 len 函数验证其长度并通过索引操作得到其各个元素。

在 Python 中，索引是按照从最前面的偏移量进行编码的，也就是从 0 开始，第一项索引为 0，第二项索引为 1，依次类推。在 python 中，我们能够反向索引，从最后一个开始。

```
>>> s='chuguangming'
>>> len(s)
12
>>> s[0]
'c'
>>> s[1]
'h'
>>> s[-1]
'g'
>>> s[-2]
'n'
>>>
```

除了简单的从位置进行索引，序列也支持一种所谓分片的操作，这是一种一步就能够提取整个分片的方法。例如：

```
>>> s
'chuguangming'
>>> s[1:3]
'hu'
>>> s[0:3]
'chu'
>>>
```

```
>>> s='chuguangming'
>>> s[1:]
'huguangming'
>>> s[0:3]
'chu'
>>> s[:3]
'chu'
>>> s[:1]
'c'
>>> s[:-1]
'chuguangmin'
>>> s[:]
'chuguangming'
>>>
```

```
>>> s='chu'
>>> s+'guangming'
'chuguangming'
>>> s*3
'chuchuchu'
>>>
```

## 1) 解决中文字符的问题

```
#-*- coding:utf-8 -*-
from __future__ import unicode_literals
print(type("test")) # output: <type 'unicode'>
spam = "测试字符串"
print(spam[1:3]) # output: 试字
```

## 2) 字符串的格式化

C 语言使用函数 `printf()`、`sprintf()` 格式化输出结果，Python 也提供了类似的功能。Python 将若干值值插入到带有%标记的字符串，从而可以动态的输出字符串。字符串的格式化语法如下所示：

```
#-*-coding:utf-8-*-
import sys
if __name__=="__main__":
    #格式化字符串
    str1="version"
    num=1.0
    format="%s"%str1
    print format
    format="%s%d"%(str1,num)
    print format
```

格式	描述
%%	百分号标记
%c	字符及其ASCII码
%s	字符串
%d	有符号整数(十进制)
%u	无符号整数(十进制)
%o	无符号整数(八进制)
%x	无符号整数(十六进制)
%X	无符号整数(十六进制大写字符)
%e	浮点数字(科学计数法)
%E	浮点数字(科学计数法，用E代替e)
%f	浮点数字(用小数点符号)
%g	浮点数字(根据值的大小采用%e或%f)
%G	浮点数字(类似于%g)
%p	指针(用十六进制打印值的内存地址)
%n	存储输出字符的数量放进参数列表的下一个变量中

```
#-*-coding:utf-8-*-
import sys
if __name__=="__main__":
    #带精度的格式化字符串
    print "浮点型数字:%f"%1.25
```

```
print "浮点型数字:%.1f"%1.25
print "浮点型数字:%.2f"%1.254
```

```
#-*-coding:utf-8-*-
import sys
if __name__=="__main__":
    #带精度的格式化字符串
    print "浮点型数字:%f"%1.25
    print "浮点型数字:%.1f"%1.25
    print "浮点型数字:%.2f"%1.254

    #使用字典格式化字符串
    print"%(version)s:%(num).1f"%{"version":"version","num":2}
    #字符串对齐
    word="version3.0"
    print word.center(20)
    print word.center(20,"*")
    #转义字符
    path="hello\tworld\n"
    print path
    print len(path)
    #直接输出转义
    path=r"hello\tworld\n"
    print path
```

### 3) 字符串的合并与分割

与 java 语言一样，Python 使用“+”号连接不同的字符串，Python 会根据加号两侧变量的类型，还决定连接操作或加法运算。

```
#-*-coding:utf-8-*-
import sys
if __name__=="__main__":
    #使用字符串截取子串
    word="world Hello"
    print word[0]
    #string[start:end:step]
    #从 0 开始截取三个
    print word[0:3]
    #从 0 开始
    print word[0::2]
    #使用 split()获取子串
    sentence="Bob Said:1,2,3,4"
    print "使用空格取子串: ",sentence.split()
    print "使用逗号取子串: ",sentence.split(",")
    print "使用两个逗号取子串: ",sentence.split(",",2)
```

```
#-*- coding:utf-8-*-
import sys
def Main():
    print "string test example"
    s='chuguangming'
    #字符串替换
    print s.find('chu')
    print s.replace('chu','***')
    #字符串分割
    line='aaa,BBB,ccc,ddd'
    result= line.split(',')
    for s in result:
        print s
    #大小写转换
    line=line.upper()
    print line
    #判断是不是字符串
    print line.isalnum()
    print line.isalpha()
    #去除空格
    line='aaa,bbb,ccc,ddd\n'
    print line.rstrip()
if __name__=="__main__":
    Main()
```

正则表达式用于搜索、替换、解析字符串。正则表达式的功能强大，使用非常灵活。使用正则表达式需要遵守一定的语法规则，用来编写一些逻辑验证非常方便，例如电子邮件的验证。Python 提供了 **re** 模块实现正则表达式的验证。

```
#-*-coding:utf-8-*-
import sys
import re
if __name__=="__main__":
    #使用^与$的使用方法
    s="HELLO WORLD"
    #匹配 hello 开始的字符串，由于变量 s 中的 HELLO 是大写的所以匹配失败
    print re.findall(r"^hello",s)
    #re.I 表示时忽略大小写
    print re.findall(r"^hello",s,re.I)
    #$的意思是匹配尾部
    print re.findall(r"WORLD$",s)
    #匹配每个英文单词\b 用于分割单词
    print re.findall(r"\b\w+\b",s)
```

## 4) 常用字符操作演示

## 去空格及特殊符号

```
s.strip().lstrip().rstrip(',')
```

## 复制字符串

```
#strcpy(sStr1,sStr2)
sStr1 = 'strcpy'
sStr2 = sStr1
sStr1 = 'strcpy2'
print sStr2
```

## 连接字符串

```
#strcat(sStr1,sStr2)
sStr1 = 'strcat'
sStr2 = 'append'
sStr1 += sStr2
print sStr1
```

## 查找字符

```
#strchr(sStr1,sStr2)
# < 0 为未找到
sStr1 = 'strchr'
sStr2 = 's'
nPos = sStr1.index(sStr2)
print nPos
```

## 比较字符串

```
#strcmp(sStr1,sStr2)
sStr1 = 'strchr'
sStr2 = 'strch'
print cmp(sStr1,sStr2)
```

## 扫描字符串是否包含指定的字符

```
#strspn(sStr1,sStr2)
sStr1 = '12345678'
sStr2 = '456'
#sStr1 and chars both in sStr1 and sStr2
print len(sStr1 and sStr2)
```

## 字符串长度

```
#strlen(sStr1)
sStr1 = 'strlen'
print len(sStr1)
```

## 将字符串中的大小写转换

```
#strlwr(sStr1)
sStr1 = 'JCstrlwr'
sStr1 = sStr1.upper()
#sStr1 = sStr1.lower()
print sStr1
```

## 追加指定长度的字符串

```
#strncat(sStr1,sStr2,n)
sStr1 = '12345'
sStr2 = 'abcdef'
```



```
n = 3
sStr1 += sStr2[0:n]
print sStr1
```

字符串指定长度比较

```
#strncmp(sStr1,sStr2,n)
sStr1 = '12345'
sStr2 = '123bc'
n = 3
print cmp(sStr1[0:n],sStr2[0:n])
```

复制指定长度的字符

```
#strncpy(sStr1,sStr2,n)
sStr1 = ''
sStr2 = '12345'
n = 3
sStr1 = sStr2[0:n]
print sStr1
```

将字符串前 n 个字符替换为指定的字符

```
#strnset(sStr1,ch,n)
sStr1 = '12345'
ch = 'r'
n = 3
sStr1 = n * ch + sStr1[3:]
print sStr1
```

扫描字符串

```
#strpbrk(sStr1,sStr2)
sStr1 = 'cekjgdklab'
sStr2 = 'gka'
nPos = -1
for c in sStr1:
    if c in sStr2:
        nPos = sStr1.index(c)
        break
print nPos
```

翻转字符串

```
#strrev(sStr1)
sStr1 = 'abcdefg'
sStr1 = sStr1[::-1]
print sStr1
```

查找字符串

```
#strstr(sStr1,sStr2)
sStr1 = 'abcdefg'
sStr2 = 'cde'
print sStr1.find(sStr2)
```

分割字符串

```
#strtok(sStr1,sStr2)
```

```
sStr1 = 'ab,cde,fgh,ijk'
sStr2 = ','
sStr1 = sStr1[sStr1.find(sStr2) + 1:]
print sStr1
#或者
s = 'ab,cde,fgh,ijk'
print(s.split(','))
```

连接字符串

```
delimiter = ','
mylist = ['Brazil', 'Russia', 'India', 'China']
print delimiter.join(mylist)
```

PHP 中 addslashes 的实现

```
def addslashes(s):
    d = {'"':'\\"', "'":'\\'', "\\0":'\\\\0', "\\":'\\\\'}
    return ''.join(d.get(c, c) for c in s)

s = "John 'Johny' Doe (a.k.a. \"Super Joe\")\\\\0"
print s
print addslashes(s)
```

只显示字母与数字

```
def OnlyCharNum(s,oth=''):
    s2 = s.lower();
    fomart = 'abcdefghijklmnopqrstuvwxyz0123456789'
    for c in s2:
        if not c in fomart:
            s = s.replace(c,'')
    return s;

print(OnlyStr("a000 aa-b"))
```

## 十五、 文件 IO

### 1) 基本文件功能演示

```
#-*-coding:utf-8-*-
import sys
import re
if __name__=="__main__":
```

```
#创建文件
context=''hello world
hello china ''
f=file('hello.txt','w')
f.write(context)
f.close()
```

```
#-*-coding:utf-8-*-
import sys
import re
if __name__=="__main__":
    #读取文件
    f=open("hello.txt")
    while True:
        line=f.readline()
        if line:
            print line
        else:
            break
    f.close()
```

```
#-*-coding:utf-8-*-
import sys
import re
if __name__=="__main__":
    #使用 readlnes()读取多个文件
    f=file("hello.txt")
    lines=f.readlines()
    print lines
    for line in lines:
        print line
```

```
#-*-coding:utf-8-*-
import sys
import re
if __name__=="__main__":
    #一次性读取
    f=open("hello.txt")
    context=f.read()
    print context
```

```
#-*-coding:utf-8-*-
import sys
import re
if __name__=="__main__":
    f=open("hello.txt")
    context=f.read(5)
```

```
print context #读取前 5 个字节的数据
print f.tell()#显示当前的位置
context=f.read(5)
print context
print f.tell()
f.close()
```

## 2) 文件的写入

```
#-*-coding:utf-8-*-
import sys
import re
if __name__=="__main__":
    #使用 writelines()写文件
    f=file("hello.txt","w+")
    li=["hello chu888\n","hello li\n"]
    f.writelines(li)
    f.close()
```

```
#-*-coding:utf-8-*-
import sys
import re
if __name__=="__main__":
    #使用 writelines()写文件
    f=file("hello.txt","w+")
    li=["hello chu888 楚\n","hello li\n"]
    f.writelines(li)
    f.close()
    #追加文件内容
    f=file("hello.txt","a+")
    new_context="goodbye"
    f.write(new_context)
    f.close()
```

## 3) 文件的删除与复制

```
#-*-coding:utf-8-*-
import sys
import re
import os
if __name__=="__main__":
    file("hello.txt","w")
```

```
if os.path.exists("hello.txt"):
    os.remove("hello.txt")
```

`file` 类没有提供文件拷贝的功能，但是我们可以使用 `read()`、`write()` 方法模拟实现文件的拷贝，但是最好的方法是引入 `shutil` 模块

```
#-*-coding:utf-8-*-
import sys
import re
import os
import shutil
if __name__=="__main__":
    shutil.copyfile("a.txt","hello2.txt")
    shutil.move("a.txt","../")
    shutil.move("hello2.txt","aaa.txt")
```

## 4) 文件与目录的重命名

`os` 模块的函数 `rename()` 可以对文件或目录进行重命名

演示文件重命名的操作。如果当前目录存在名为 `hello.txt` 的文件，则重命名为 `hi.txt`；如果存在 `hi.txt` 的文件则重命名为 `hello.txt`

```
#-*-coding:utf-8-*-
import sys
import os
if __name__=="__main__":
    li=os.listdir(".")#判断当前目录
    print li
    if "hello.txt" in li:
        os.rename("hello.txt","hi.txt")
    elif "hi.txt" in li:
        os.rename("hi.txt","hello.txt")
```

把后缀名为“html”的文件修改为“htm”后缀的文件

```
#-*-coding:utf-8-*-
import sys
import os
if __name__=="__main__":
    files=os.listdir(".")
    for filename in files:
        #查找文件名中.所在的位置并把它给 pos
```

```
pos=filename.find(".")
#print pos
#得到.后面的内容
if filename[pos+1]=="html":
    print filename
    newname=filename[:pos+1]+".htm"
    print newname
    os.rename(filename,newname)
```

```
#-*-coding:utf-8-*-
import sys
import os
if __name__=="__main__":
    files=os.listdir(".")
    for filename in files:
        li=os.path.splitext(filename)
        if li[1]==".html":
            newname=li[0]+".htm"
            os.rename(filename,newname)
```

## 5) 文件内容的查找和替换

从文件中查找字符串 hello，并统计 hello 出现的次数

```
#-*-coding:utf-8-*-
import sys
import os
import re
if __name__=="__main__":
    fl=file("aaa.txt","r")
    count=0
    for s in fl.readlines():
        li=re.findall("hello",s)
        if len(li)>0:
            count=count+li.count("hello")
    print "查找到"+str(count)+"个 hello"
    fl.close()
```

把 hello.txt 中的字符串 hello 全部替换为 hi，并把结果保存在文件 hello2.txt 中

```
#-*-coding:utf-8-*-
import sys
import os
import re
if __name__=="__main__":
    f1=file("aaa.txt","r")
    f2=file("bbb.txt","w")
```

```
for s in f1.readlines():
    f2.write(s.replace("hello","hi"))
f1.close()
f2.close()
```

## 6) 文件的比较

Python 提供了模块 `difflib` 用于实现对序列、文件的比较。如果要比较两个文件，列出两个文件的异同，可以使用 `difflib` 模块的 `SequenceMatcher` 类实现。其中的方法 `get_opcodes()` 可以返回两个序列的比较结果。调用方法 `get_opcodes()` 之前，需要生成 1 个 `SequenceMatcher` 对象。

# 十六、 Web.Py 框架概述

## 1) Python 下 web 开发框架的选择

以下内容为网络摘录，来源：飞龙博客

### 1. Django



Python 框架虽然说是百花齐放，但仍然有那么一家是最大的，它就是 Django。要说 Django 是 Python 框架里最好的，有人同意也有人坚决反对，但说 Django 的文档最完善、市场占有率最高、招聘职位最多估计大家都没什么意见。Django 为人所称道的地方主要有：

- 完美的文档，Django 的成功，我觉得很大一部分原因要归功于 Django 近乎完美的官方文档（包括 Django book）。
- 全套的解决方案，Django 象 Rails 一样，提供全套的解决方案（full-stack framework + batteries included），基本要什么有什么（比如：cache、session、feed、orm、geo、auth），而且全部 Django 自己造，开发网站应手的工具 Django 基本都给你做好了，因此开发效率是不用说的，出了问题也算好找，不在你的代码里就在 Django 的源码里。
- 强大的 URL 路由配置，Django 让你可以设计出非常优雅的 URL，在 Django 里你基本可以跟丑陋的 GET 参数说拜拜。
- 自助管理后台，admin interface 是 Django 里比较吸引眼球的一项 contrib，让你几乎不用写一行代码就拥有一个完整的后台管理界面。

而 Django 的缺点主要源自 Django 坚持自己造所有的轮子，整个系统相对封闭，Django 最为人诟病的地方有：

- 系统紧耦合，如果你觉得 Django 内置的某项功能不是很好，想用喜欢的第三方库来代替是很难的，比如下面将要说的 ORM、Template。要在 Django 里用 SQLAlchemy 或 Mako 几乎是不可能，即使打了一些补丁用上了也会让你觉得非常非常别扭。
- Django 自带的 ORM 远不如 SQLAlchemy 强大，除了在 Django 这一亩三分地，SQLAlchemy 是 Python 世界里事实上的 ORM 标准，其它框架都支持 SQLAlchemy 了，唯独 Django 仍然坚持自己的那一套。Django 的开发人员对 SQLAlchemy 的支持也是有过讨论和尝试的，不过最终还是放弃了，估计是代价太高且跟 Django 其它的模块很难合到一块。
- Template 功能比较弱，不能插入 Python 代码，要写复杂一点的逻辑需要另外用 Python 实现 Tag 或 Filter。关于模板这一点，一直以来争论比较多，最近有两篇关于 Python 模板的比较有意思的文章可供参考：
  1. <http://pydanny.blogspot.com/2010/12/stupid-template-languages.html>（需翻墙）
  2. <http://techspot.zzzeek.org/2010/12/04/in-response-to-stupid-template-languages/>
- URL 配置虽然强大，但全部要手写，这一点跟 Rails 的 Convention over configuration 的理念完全相左，高手和初识 Django 的人配出来的 URL 会有很大差异。
- 让人纠结的 auth 模块，Django 的 auth 跟其它模块结合紧密，功能也挺强的，就是做的有点过了，用户的数据库 schema 都给你定好了，这样问题就来了，比如很多网站要求 email 地址唯一，可 schema 里这个字段的值不是唯一的，纠结是必须的了。
- Python 文件做配置文件，而不是更常见的 ini、xml 或 yaml 等形式。这本身不是什么问题，可是因为理论上来说 settings 的值是能够动态的改变的（虽然大家不会这么干），但这不是最佳实践的体现。

总的来说，Django 大包大揽，用它来快速开发一些 Web 运用是很不错的。如果你顺着 Django 的设计哲学来，你会觉得 Django 很好用，越用越爽；相反，你如果不能融入或接受 Django 的设计哲学，你用 Django 一定会很痛苦，趁早放弃的好。所以说在有些人眼里 Django 无异于仙丹，但对有一些人来说它又是毒药且剧毒。

Django 案例有 [disqus.com](http://disqus.com)、[bitbucket.org](http://bitbucket.org)、[海报网](http://海报网)等。

## 2. [Pylons](#) & [TurboGears](#) & [repoze.bfg](#)



除了 Django 另一个大头就是 Pylons 了，因为 TurboGears2.x 是基于 Pylons 来做的，而 repoze.bfg 也已经并入 Pylons project 里这个大的项目里，后面不再单独讨论 TurboGears 和 repoze.bfg 了。

Pylons 和 Django 的设计理念完全不同，Pylons 本身只有两千行左右的 Python 代码，不过它还附带有一些几乎就是 Pylons 御用的第三方模块。Pylons 只提供一个架子和可选方案，你可以根据自己的喜好自由的选择 Template、ORM、form、auth 等组件，系统高度可定制。我们常说 Python 是一个胶水语言(glue language)，那么我们完全可以说 Pylons 就是一个用胶水语言设计的胶水框架：)

选择 Pylons 多是选择了它的自由，选择了自由的同时也预示着你选择了噩梦：

- 学习噩梦，Pylons 依赖于许多第三方库，它们并不是 Pylons 造，你学 Pylons 的同时还得学这些库怎么使用，关键有些时候你都不知道你要学什么。Pylons 的学习曲线相对比 Django 要高的多，而之前 Pylons 的官方文档



也一直是人批评的对象，好在后来出了 [The Definitive Guide to Pylons](#) 这本书，这一局面有所改观。因为这个原因，Pylons 一度被誉为只适合高手使用的 Python 框架。

- 调试噩梦，因为牵涉到的模块多，一旦有错误发生就比较难定位问题处在哪儿。可能是你写的程序的错、也可能是 Pylons 出错了、再或是 SQLAlchemy 出错了、搞不好是 formencode 有 bug，反正很凌乱了。这个只有用的很熟了才能解决这个问题。
- 升级噩梦，安装 Pylons 大大小小共要安装近 20 个 Python 模块，各有各的版本号，要升级 Pylons 的版本，哪个模块出了不兼容的问题都有可能，升级基本上很难很难。至今 reddit 的 Pylons 还停留在古董的 0.9.6 上，SQLAlchemy 也还是 0.5.3 的版本，应该跟这条有关系。所以大家玩 Pylons 一定要结合 virtualenv 来玩，给自己留条后路，不然会死得很惨。

Pylons 和 repoze.bfg 的融合可能会催生下一个能挑战 Django 地位的框架。

Pylons 的案例有 [reddit.com](#)、[dropbox.com](#)、[quora.com](#) 等。

### 3. Tornado & web.py



Tornado 即是一个 web server（对此本文不作详述），同时又是一个类 web.py 的 micro-framework，作为框架 Tornado 的思想主要来源于 web.py，大家在 web.py 的网站首页也可以看到 Tornado 的大佬 [Bret Taylor](#) 的这么一段话（他这里说的 FriendFeed 用的框架跟 Tornado 可以看作是一个东西）：

“[web.py inspired the] web framework we use at FriendFeed [and] the webapp framework that ships with App Engine...”

因为这层关系，后面不再单独讨论 Tornado。

web.py 的设计理念力求精简（Keep it simple and powerful），总共就没多少行代码，也不像 Pylons 那样依赖大量的第三方模块，而是只提供一个框架所必须的一些东西，如：URL 路由、Template、数据库访问，其它的就交给用户自己去做好了。

一个框架精简的好处在于你可以聚焦在业务逻辑上，而不用太多的去关心框架本身或受框架的干扰，同时缺点也很明显，许多事情你得自己操刀上。

我个人比较偏好这种精简的框架，因为你很容易通过阅读源码弄明白整个框架的工作机制，如果框架那一块不是很合意的话，我完全可以 Monkey patch 一下按自己的要求来。

早期的 reddit 是用 web.py 写的，Tornado 的案例有 [friendfeed.com](#)、[bit.ly](#)、[quora.com](#) 和我的开源站点 [poweredsites.org](#) 等。

## 4. Bottle & Flask



Bottle 和 Flask 作为新生代 Python 框架的代表，挺有意思的是都采用了 **decorator** 的方式配置 URL 路由，如：

```
from bottle import route, run

@route('/:name')
def index(name='World'):
    return '<b>Hello %s!</b>' % name

run(host='localhost', port=8080)
```

Bottle、Flask 跟 web.py 一样，都非常精简，Bottle 甚至所有的代码都在那一个两千来行的.py 文件里。另外 Flask 和 Pylons 一样，可以跟 Jinja2、SQLAlchemy 之类结合的很好。

不过目前不管是 Bottle 还是 Flask 成功案例都还很少。

## 5. Quixote

之所以要特别说一下 Quixote，是因为国内的最大的用 Python 开发的网站“[豆瓣网](#)”是用 Quixote 开发的。我只简单翻了一下源代码，没有做过研究，不发表评论，有经验的来补充下。我只是在想，如果豆瓣网交到现在来开发，应该会有更多的选择。

## 6. 最后关于框架选择的误区

在框架的选择问题上，许多人很容易就陷入了下面两个误区中而不自知：

1. 哪个框架最好 — 世上没有最好的框架，只有最适合你自己、最适合你的团队的框架。编程语言选择也是一个道理，你的团队 Python 最熟就用 Python 好了，如果最熟悉的是 Ruby 那就用 Ruby 好了，编程语言、框架都只是工具，能多、快、好、省的干完活就是好东西，管 TMD 是日本鬼子还是美帝造呢！
2. 过分关注性能 — 其实大部分人是没必要太关心框架的性能的，因为你开发的网站根本就是个小站，能上 1 万的 IP 的网站已经不多了，上 10 万的更是很少很少。在没有一定的访问量前谈性能其实是没有多大意义的，因为你的 CPU 和内存一直就闲着呢。而且语言和框架一般也不会是性能瓶颈，性能问题最常出现在数据库访问和文件读写上。PHP 的 Zend Framework 是出了名的慢，但是 Zend Framework 一样有大站，如：digg.com；常被人说有性能问题的 Ruby 和 Rails，不是照样可以开发出 twitter 吗？再者现在的硬件、带宽成本其实是很低的，特别有了云计算平台后，人力成本才是最贵的，没有上万的 IP 根本就不用太在意性能问题，流量上去了花点钱买点服务器空间好了，简单快速的解决性能问题。

注：前面有网友质疑我“Quora 是用 Pylons 开发的”这样的说法不客观，特说明一下，这里所说的某个网站 A 是用 B 开发的，只是指 A 主要或部分是由 B 开发的，大家就不要再纠结 A 还用 C 了。

## 2) web.py 安装

首先下载 Web.Py 的安装包：

```
http://webpy.org/static/web.py-0.37.tar.gz
```

或者从 GITHUB 下载最新的安装包也可以：

```
https://github.com/webpy/webpy/tarball/master
```

使用命令行安装

```
python setup.py install
```

如果是 Linux 下面的话，需要使用 Sudo

```
sudo python setup.py install
```

## 3) web.py 下常用框架简介

- jinja2, python 中一个很流行的模板框架，用了它，写模板就是如此的享受。
- sqlalchemy, python 最强大的 orm，没有之一，掌握了它，数据库？so easy!
- formalchemy, 配套 sqlalchemy 的表单框架，可以根据 sqlalchemy 中数据表的定义生成 html 表单
- kendo ui, 这是一个 jquery 框架，提供了很多插件，里面最让我欣赏的，就是 mvvm 的开发模式和 datasource 的 ajax，让本来已经很少的 jquery 代码变得更少了。

## 4) 最简单的 HelloWorld!

```
# -*- coding:utf-8 -*-
import web

#定义 url, 将地址映射到对应的类
urls = (
    "/", "index",
)

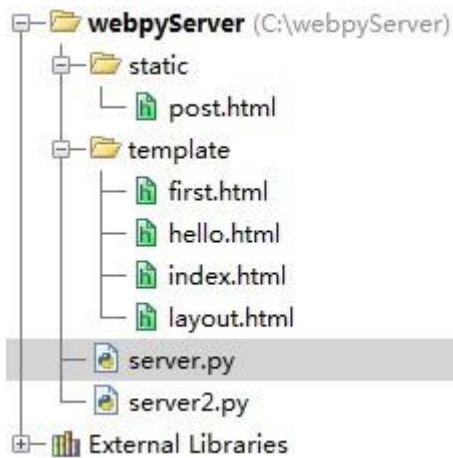
app = web.application(urls, globals())

#定义 index 类
class index:
    #get 请求
    def GET(self):
        return "Hello World"

if __name__ == "__main__":
    app.run()
```

## 5) Web.py 基本使用模板使用

项目结构



server.py

```
#!/usr/bin/env python
#-*-coding:utf-8-*-
import web
urls = (
    '/', 'index'
)
render = web.template.render('template/')
class index:
    def GET(self):
        #return render.hello('楚广明')
        return render.first('world','name')
if __name__ == "__main__":
    app = web.application(urls, globals())
    app.run()
```

first.html

```
$def with (name1,name2)
<html><head>
    <title>my si

    </title>
</head><body>
Hello $name1 , hello $name2!
<table>
    $for c in ["a", "b", "c", "d"]:
        <tr class="$loop.parity">
            <td>$loop.index</td>
            <td>$c</td>
        </tr>
    </table>
</body></html>
```

## 6) 进阶模板使用

### 1. server2.py

```
#-*-coding:utf-8-*-
import web
from web import form as form
urls = (
    '/', 'index',
    '/add','add'
)
render = web.template.render('template/')
class index:
    def GET(self):
        return render.post()
class add:
    def POST(self):
        print web.input()['title1']
        print web.data()
        raise web.seeother('/')

if __name__ == "__main__":
    app = web.application(urls, globals())
    app.run()
```

### 2. post.html

```
<html><head>
    <title>post test

    </title>
</head><body>
<form method="post" action="/add">
    <input type="text" name="title" />
    <input type="text" name="title1" />
    <input type="text" name="title2" />
    <input type="text" name="title3" />
    <input type="submit" value="Add" />
</form>

</body></html>
```

## 7) URL 控制

### 1. 问题：

如何为整个网站设计一个 URL 控制方案 / 调度模式

### 2. 解决：

web.py 的 URL 控制模式是简单的、强大的、灵活的。在每个应用的最顶部，你通常会看到整个 URL 调度模式被定义在元组中：

```
urls = (
    "/tasks/?", "signin",
    "/tasks/list", "listing",
    "/tasks/post", "post",
    "/tasks/chgpass", "chgpass",
    "/tasks/act", "actions",
    "/tasks/logout", "logout",
    "/tasks/signup", "signup"
)
```

这些元组的格式是：URL 路径，处理类 这组定义有多少可以定义多少。如果你并不知道 URL 路径和处理类之间的关系。

### 3. 路径匹配

你可以利用强大的正则表达式去设计更灵活的 URL 路径。比如 `/(test1|test2)` 可以捕捉 `/test1` 或 `/test2`。要理解这里的关键，匹配是依据 URL 路径的。比如下面的 URL：

```
http://localhost/myapp/greetings/hello?name=Joe
```

你可以捕捉 URL 的参数，然后用在处理类中：

```
/users/list/(.+), "list_users"
```

在 `list/` 后面的这块会被捕捉，然后作为参数被用在 GET 或 POST：

```
class list_users:
    def GET(self, name):
        return "Listing info about user: {0}".format(name)
```

## 8) 一个完整的小例子

```
#-*-coding:utf-8-*-
import web

urls = (
    "/", "hello",
    "/chu888/?", "chu888",
```

```
        "/parmtest/(.+)", "listmy"
    )
app = web.application(urls, globals())

class hello:
    def GET(self):
        return 'Hello, world!'
class chu888:
    def GET(self):
        return "你访问的是 chu888"
class listmy:
    def GET(self, name):
        return "Listing info about user: {0}".format(name)
if __name__ == "__main__":
    app.run()
```

## 9) web.seeother 和 web.redirect 转向

### 1. 问题

在处理完用户输入后（比方说处理完一个表单），如何跳转到其他页面？

### 2. 解法

```
class SomePage:
    def POST(self):
        # Do some application logic here, and then:
        raise web.seeother('/someotherpage')
```

POST 方法接收到一个 post 并完成处理之后，它将给浏览器发送一个 303 消息和新网址。接下来，浏览器就会对这个新网址发出 GET 请求，从而完成跳转。

注意：web.seeother 和 web.redirect 不支持 0.3 以下版本。

### 3. 区别

用 web.redirect 方法似乎也能做同样的事情，但通常来说，这并不友好。因为 web.redirect 发送的是 301 消息——这是永久重定向。因为大多数 Web 浏览器会缓存新的重定向，所以当我们再次执行该操作时，会自动直接访问重定向的新网址。很多时候，这不是我们所想要的结果。所以在提交表单时，尽量使用 seeother。但是在下面要提到的这种场合，用 redirect 却是最恰当的：我们已经更改了网站的网址结构，但仍想让用户书签/收藏夹中的旧网址不失效。

（注：要了解 seeother 和 redirect 的区别，最好是看一下 http 协议中不同消息码的含义。）

## 10) 包含应用

### 1. 实现

在 `blog.py` 中：

```
import web
urls = (
    "", "reblog",
    "/(.*)", "blog"
)

class reblog:
    def GET(self): raise web.seeother('/')

class blog:
    def GET(self, path):
        return "blog " + path

app_blog = web.application(urls, locals())
```

当前的主应用 `code.py`：

```
import web
import blog
urls = (
    "/blog", blog.app_blog,
    "/(.*)", "index"
)

class index:
    def GET(self, path):
        return "hello " + path

app = web.application(urls, locals())

if __name__ == "__main__":
    app.run()
```

## 11) 使用 XML

### 1. 问题

如何在 `web.py` 中提供 XML 访问？

如果需要为第三方应用收发数据，那么提供 `xml` 访问是很有必要的。



## 2. 解法

根据要访问的 xml 文件(如 response.html)创建一个 XML 模板。如果 XML 中有变量，就使用相应的模板标签进行替换。下面是一个例子：

```
$def with (code)  
<?xml version="1.0"?>  
<RequestNotification-Response>  
<Status>$code</Status>  
</RequestNotification-Response>
```

为了提供这个 XML，需要创建一个单独的 web.py 程序(如 response.py)，它要包含下面的代码。注意：要用 "web.header('Content-Type', 'text/xml')" 来告知客户端——正在发送的是一个 XML 文件。

```
import web  
  
render = web.template.render('template/', cache=False)  
  
urls = (  
    '/(.*)', 'index'  
)  
  
app = web.application(urls, globals())  
  
class index:  
    def GET(self, code):  
        web.header('Content-Type', 'text/xml')  
        return render.index(code)  
  
web.webapi.internalerror = web.debugerror  
if __name__ == '__main__': app.run()
```

## 12) 获取 POST 数据

```
class RequestHandler():  
    def POST():  
        data = web.data() # 通过这个方法可以取到数据
```

## 1. login.html 模板

```
<form id="form1" name="form1" method="post" action="/post/">
  <h1>Sign-up form</h1>

  <p>This is the basic look of my form without table</p>
  <label>Name
    <span class="small">Add your name</span>
  </label>
  <input type="text" name="UserName" id="UserName"/>

  <label>Email
    <span class="small">Add a valid address</span>
  </label>
  <input type="text" name="Email" id="Email"/>

  <label>Password
    <span class="small">Min. size 6 chars</span>
  </label>
  <input type="text" name="PassWord" id="PassWord"/>
  <button type="submit">Sign-up</button>
  <div class="spacer"></div>
```

## 2. 主程序

```
#-*-coding:utf-8-*-
import web

urls = (
    "/", "index",
    "/login/", "login",
    "/post/", "post"
)
render = web.template.render('template/')
app = web.application(urls, globals())

class index:
    def GET(self):
        return "Hello web.py!"

class login:
    def GET(self):
        return render.login()

class post:
    def POST(self):
        return web.input()['UserName']

if __name__ == "__main__":
    app.run()
```

## 13) 获取客户端信息

### 1. 问题

如何在代码中得到客户端信息？比如：来源页面(referring page)或是客户端浏览器类型

### 2. 解法

使用 `web.ctx` 即可。首先讲一点架构的东西：`web.ctx` 基于 `threadeddict` 类，又被叫做 `ThreadDict`。这个类创建了一个类似字典(dictionary-like)的对象，对象中的值都是与线程 `id` 相对应的。这样做很妙，因为很多用户同时访问系统时，这个字典对象能做到仅为某一特定的 HTTP 请求提供数据(因为没有数据共享，所以对象是线程安全的)

`web.ctx` 保存每个 HTTP 请求的特定信息，比如客户端环境变量。假设，我们想知道正在访问某页面的用户是从哪个网页跳转而来的：

### 3. 例子

```
class example:
    def GET(self):
        referer = web.ctx.env.get('HTTP_REFERER', 'http://google.com')
        raise web.seeother(referer)
```

上述代码用 `web.ctx.env` 获取 `HTTP_REFERER` 的值。如果 `HTTP_REFERER` 不存在，就会将 `google.com` 做为默认值。接下来，用户就会被重定向回到之前的来源页面。

`web.ctx` 另一个特性，是它可以被 `loadhook` 赋值。例如：当一个请求被处理时，会话(`Session`)就会被设置并保存在 `web.ctx` 中。由于 `web.ctx` 是线程安全的，所以我们可以象使用普通的 `python` 对象一样，来操作会话(`Session`)。

### 4. 'ctx'中的数据成员

#### Request

- `environ` 又被写做 `env` – 包含标准 `WSGI` 环境变量的字典。
- `home` – 应用的 `http` 根路径(译注：可以理解为应用的起始网址，协议+站点域名+应用所在路径)例：  
`http://example.org/admin`
- `homedomain` – 应用所在站点(可以理解为协议+域名) `http://example.org`
- `homepath` – 当前应用所在的路径，例如： `/admin`
- `host` – 主机名(域名)+用户请求的端口(如果没有的话，就是默认的 80 端口)，例如： `example.org,example.org:8080`
- `ip` – 用户的 IP 地址，例如： `xxx.xxx.xxx.xxx`
- `method` – 所用的 `HTTP` 方法，例如： `GET`
- `path` – 用户请求路径，它是基于当前应用的相对路径。在子应用中，匹配外部应用的那部分网址将被去掉。例如：主应用在 `code.py` 中，而子应用在 `admin.py` 中。在 `code.py` 中，我们将 `/admin` 关联到 `admin.app`。在 `admin.py` 中，将 `/stories` 关联到 `stories` 类。在 `stories` 中，`web.ctx.path` 就是 `/stories`，而非 `/admin/stories`。形如： `/articles/845`
- `protocol` – 所用协议，例如： `https`

- `query` – 跟在'?' 字符后面的查询字符串。如果不存在查询参数，它就是一个空字符串。例如：`?fourlegs=good&twolegs=bad`
- `fullpath` 可以视为 `path` + `query` – 包含查询参数的请求路径，但不包括'homepath'。例如：`/articles/845?fourlegs=good&twolegs=bad`

## Response

- `status` – HTTP 状态码（默认是'200 OK'）401 Unauthorized 未经授权
- `headers` – 包含 HTTP 头信息(headers)的二元组列表。
- `output` – 包含响应实体的字符串。

## 14) 文件上传

### 1. 简单的文件上传

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

class mytest:
    def GET(self):
        return "<b><h1>我扩展测试一下</h1><b>"

class Upload:
    def GET(self):
        return """<html><head></head><body>
<form method="POST" enctype="multipart/form-data" action="">
<input type="file" name="myfile" />
<br/>
<input type="submit" />
</form>
</body></html>"""

    def POST(self):
        x = web.input(myfile={})
        web.debug(x['myfile'].filename) # 这里是文件名
        web.debug(x['myfile'].value) # 这里是文件内容
        web.debug(x['myfile'].file.read()) # 或者使用一个文件对象
        print x['myfile'].filename
        print x['myfile'].value
        print x['myfile'].file.read()
        raise web.seeother('/todo/upload')
```

### 2. 基于新浪云的上传

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

```
# coding: utf-8
import web
import uuid
import random
import time
import sae.storage
from config import settings
from datetime import datetime

render = settings.render

class SaveUpload:
    def GET(self):
        # 初始化一个 Storage 客户端。
        s = sae.storage.Client()
        imagelist=s.list('pythondb')
        return render.SaveUpload(imagelist)

    # def GET(self):
    #     web.header("Content-Type","text/html; charset=utf-8")
    #     return """<html><head><title>文件上传并且保存 demo</title></head><body>
    #         <form method="POST" enctype="multipart/form-data" action="">
    #         <input type="file" name="myfile" />
    #         <br/>
    #         <input type="submit" />
    #         </form>
    #         </body></html>"""

    def POST(self):
        x = web.input(myfile={})
        if 'myfile' in x:
            #读取所有文件内容
            imagevalue=x.myfile.file.read()
            # 初始化一个 Storage 客户端。
            s = sae.storage.Client()

            # 设置 object 的属性
            ob = sae.storage.Object(imagevalue)
            #保存文件
            saveuploadfile= str(time.time()) + str(random.random())+x.myfile.filename
            s.put('pythondb',saveuploadfile, ob)

        raise web.seeother('/todo/saveupload')
```

```
$def with(imagelist)
<!DOCTYPE html>
<html>
```

```
<head>
  <title></title>
  <style type="text/css">
    .imagediv
    {
      width: 200px;
      height: 200px;
      border: solid 1px;
    }
  </style>
</head>
<body>
<form method="POST" enctype="multipart/form-data" action="">
  <input type="file" name="myfile" />
  <br/>
  <input type="submit" />
</form>
$for i in imagelist:
  <div ></div>
</body>
</html>
```

## 十七、 Web.py 使用 Session

### 1) 问题

如何在 web.py 中使用 session

### 2) 解法

注意!!!: session 并不能在调试模式(Debug mode)下正常工作,这是因为 session 与调试模式下的重调用相冲突(有点类似 firefox 下著名的 Firebug 插件,使用 Firebug 插件分析网页时,会在火狐浏览器之外单独对该网页发起请求,所以相当于同时访问该网页两次),下一节中我们会给出在调试模式下使用 session 的解决办法。web.session 模块提供 session 支持。下面是一个简单的例子——统计有多少人正在使用 session(session 计数器):

```
import web
web.config.debug = False
urls = (
    "/count", "count",
    "/reset", "reset"
)
app = web.application(urls, locals())
```

```
session = web.session.Session(app, web.session.DiskStore('sessions'),
initializer={'count': 0})

class count:
    def GET(self):
        session.count += 1
        return str(session.count)

class reset:
    def GET(self):
        session.kill()
        return ""

if __name__ == "__main__":
    app.run()
```

`web.py` 在处理请求之前，就加载 `session` 对象及其数据；在请求处理完之后，会检查 `session` 数据是否被改动。如果被改动，就交由 `session` 对象保存。

上例中的 `initializer` 参数决定了 `session` 初始化的值，它是个可选参数。如果用数据库代替磁盘文件来存储 `session` 信息，只要用 `DBStore` 代替 `DiskStore` 即可。使用 `DBStore` 需要建立一个表，结构如下：

```
create table sessions (
    session_id char(128) UNIQUE NOT NULL,
    atime timestamp NOT NULL default current_timestamp,
    data text
);
```

`DBStore` 被创建要传入两个参数：`db` 对象和 `session` 的表名。

```
db = web.database(dbn='postgres', db='mydatabase', user='myname', pw='')
store = web.session.DBStore(db, 'sessions')
session = web.session.Session(app, store, initializer={'count': 0})
```

`'web.config'` 中的 `sessions_parameters` 保存着 `session` 的相关设置，`sessions_parameters` 本身是一个字典，可以对其修改。默认设置如下：

```
web.config.session_parameters['cookie_name'] = 'webpy_session_id'
web.config.session_parameters['cookie_domain'] = None
web.config.session_parameters['timeout'] = 86400, #24 * 60 * 60, # 24 hours in seconds
web.config.session_parameters['ignore_expiry'] = True
web.config.session_parameters['ignore_change_ip'] = True
web.config.session_parameters['secret_key'] = 'fLjUfxqXtfNoIldA0A0J'
web.config.session_parameters['expired_message'] = 'Session expired'
```

- `cookie_name` - 保存 `session id` 的 `Cookie` 的名称
- `cookie_domain` - 保存 `session id` 的 `Cookie` 的 `domain` 信息
- `timeout` - `session` 的有效时间，以秒为单位
- `ignore_expiry` - 如果为 `True`，`session` 就永不过期
- `ignore_change_ip` - 如果为 `true`，就表明只有在访问该 `session` 的 `IP` 与创建该 `session` 的 `IP` 完全一致时，`session` 才被允许访问。
- `secret_key` - 密码种子，为 `session` 加密提供一个字符串种子
- `expired_message` - `session` 过期时显示的提示信息。

## 3) 在 template 下使用 Session

### 1. 问题:

我想在模板中使用 `session`（比如：读取并显示 `session.username`）

### 2. 解决:

在应用程序中的代码:

```
render = web.template.render('templates', globals={'context': session})
```

在模板中的代码:

```
<span>You are logged in as <b>${context.username}</b></span>
```

你可以真正的使用任何符合语法的 `python` 变量名, 比如上面用的 `context`。我更喜欢在应用中直接使用 '`session`'。

`server.py`

```
#-*-coding:utf-8-*-
import web
web.config.debug = False
urls = (
    "/count", "count",
    "/reset", "reset",
    "/", "index"
)
app = web.application(urls, locals())

db = web.database(dbn='mysql', db='todo', user='root', pw='chu123')
store = web.session.DBStore(db, 'sessions')
session = web.session.Session(app, store, initializer={'count': 0})
render = web.template.render('template', globals={'context': session})

class index:
    def GET(self):
        return render.index()
class count:
    def GET(self):
        session.count += 1
        return str(session.count)

class reset:
    def GET(self):
        session.kill()
        return ""

if __name__ == "__main__":
    app.run()
```



index.html

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
<h1>我是楚广明</h1>
<h1><span>You are logged in as <b>$context.count</b></span></h1>
</body>
</html>
```

## 4) 如何操作 Cookies

### 1. 问题

如何设置和获取用户的 Cookie?

### 2. 解法

对 web.py 而言，设置/获取 Cookie 非常方便。

### 3. 设置 Cookies

#### 概述

```
setcookie(name, value, expires="", domain=None, secure=False):
```

- name (string) - Cookie 的名称，由浏览器保存并发送至服务器。
- value (string) - Cookie 的值，与 Cookie 的名称相对应。
- expires (int) - Cookie 的过期时间，这是个可选参数，它决定 cookie 有效时间是多久。以秒为单位。它必须是一个整数，而绝不能是字符串。
- domain (string) - Cookie 的有效域—在该域内 cookie 才是有效的。一般情况下，要在某站点内可用，该参数值该写做站点的域（比如.webpy.org），而不是站主的主机名（比如 wiki.webpy.org）
- secure (bool) - 如果为 True，要求该 Cookie 只能通过 HTTPS 传输。.

#### 示例

用 web.setcookie() 设置 cookie, 如下:

```
class CookieSet:
    def GET(self):
        i = web.input(age='25')
        web.setcookie('age', i.age, 3600)
        return "Age set in your cookie"
```

用 GET 方式调用上面的类将设置一个名为 age, 默认值是 25 的 cookie(实际上，默认值 25 是在 web.input

中赋予 **i.age** 的，从而间接赋予 **cookie**，而不是在 **setcookie** 函式中直接赋予 **cookie** 的)。这个 **cookie** 将在一小时后(即 3600 秒)过期。

`web.setcookie()` 的第三个参数—**"expires"**是一个可选参数，它用来设定 **cookie** 过期的时间。如果是负数，**cookie** 将立刻过期。如果是正数，就表示 **cookie** 的有效时间是多久，以秒为单位。如果该参数为空，**cookie** 就永不过期。

## 4. 获得 Cookies

### 概述

获取 **Cookie** 的值有很多方法，它们的区别就在于找不到 **cookie** 时如何处理。

方法 1（如果找不到 **cookie**，就返回 **None**）：

```
web.cookies().get(cookieName)

#cookieName is the name of the cookie submitted by the browser
```

方法 2（如果找不到 **cookie**，就抛出 **AttributeError** 异常）：

```
foo = web.cookies()

foo.cookieName
```

方法 3（如果找不到 **cookie**，可以设置默认值来避免抛出异常）：

```
foo = web.cookies(cookieName=defaultValue)

foo.cookieName # return the value (which could be default)

#cookieName is the name of the cookie submitted by the browser
```

示例：

用 `web.cookies()` 访问 **cookie**。如果已经用 `web.setcookie()` 设置了 **Cookie**，就可以象下面这样获得 **Cookie**：

```
class CookieGet:

    def GET(self):

        c = web.cookies(age="25")

        return "Your age is: " + c.age
```

这个例子为 **cookie** 设置了默认值。这么做的原因是在访问时，若 **cookie** 不存在，`web.cookies()` 就会抛出异常，如果事先设置了默认值就不会出现这种情况。

如果要确认 **cookie** 值是否存在，可以这样做：

```
class CookieGet:

    def GET(self):

        try:

            return "Your age is: " + web.cookies().age

        except:

            # Do whatever handling you need to, etc. here.

            return "Cookie does not exist."
```

或

```
class CookieGet:

    def GET(self):

        age=web.cookies().get(age)

        if age:

            return "Your age is: %s" % age

        else:

            return "Cookie does not exist."
```

## 5) 布局模板

### 1. 问题

如何让站点每个页面共享一个整站范围的模板？（在某些框架中，称为模板继承，比如 **ASP.NET** 中的母版页）

### 2. 方法

我们可以用 **base** 属性来实现：

```
render = web.template.render('templates/', base='layout')
```

现在如果你调用 `render.foo()` 方法，将会加载 `templates/foo.html` 模板，并且它将会被 `templates/layout.html` 模板包裹。

"**layout.html**" 是一个简单模板格式文件，它包含了一个模板变量，如下：

```
$def with (content)
<html>
<head>
    <title>Foo</title>
</head>
<body>
    $:content
</body>
</html>
```

在某些情况，如果不想使用基本模板，只需要创建一个没有 **base** 属性的 **reander** 对象，如下：

```
render_plain = web.template.render('templates/')
```

程序

```
#!/usr/bin/env python
#-*-coding:utf-8-*-
import web
web.config.debug = True
urls = (
    "/", "index"
)
app = web.application(urls, locals())
render = web.template.render('template/', base='layout')

class index:
    def GET(self):
        return render.index('楚广明 6666')

if __name__ == "__main__":
    app.run()
```

layout.html

```
$def with (content)
<html>
<head>
    <title>Foo</title>
```

```
</head>
<body>
$:content
</body>
</html>
```

index.html

```
$def with (name)
<h1>我是$name</h1>
```

### 3. Tip: 在布局文件 ( layout.html ) 中定义的页面标题变量

templates/index.html

```
$var title: This is title.

<h3>Hello, world</h3>
```

templates/layout.html

```
$def with (content)
<html>
<head>
    <title>${content.title}</title>
</head>
<body>
$:content
</body>
</html>
```

### 4. Tip: 在其他模板中引用 css 文件，如下：

templates/login.html

```
$var cssfiles: static/login.css static/login2.css

hello, world.
```

templates/layout.html

```
$def with (content)
<html>
<head>
    <title>${content.title}</title>

    $if content.cssfiles:
        $for f in content.cssfiles.split():
            <link rel="stylesheet" href="$f" type="text/css" media="screen" charset="utf-8"/>

</head>
<body>
```

```
$:content
</body>
</html>
```

输入的 HTML 代码如下：

```
<link rel="stylesheet" href="static/login.css" type="text/css" media="screen" charset="utf-8"/>
<link rel="stylesheet" href="static/login2.css" type="text/css" media="screen"
charset="utf-8"/>
```

## 十八、 Web.py 支持的语法模板

### 1) 使用 Templetor 模板语法

#### 1. Introduction

web.py 的模板语言叫做 Templetor，它能负责将 python 的强大功能传递给模板系统。在模板中没有重新设计语法，它是类 python 的。如果你会 python，你可以顺手拈来。

这是一个模板示例：

```
$def with (name)
Hello $name!
```

第一行表示模板定义了一个变量 name。第二行中的 \$name 将会用 name 的值来替换。

#### 2. 使用模板系统

通用渲染模板的方法：

```
render = web.template.render('templates')
return render.hello('world')
```

render 方法从模板根目录查找模板文件，render.hello(..) 表示渲染 hello.html 模板。实际上，系统会在根目录去查找叫 hello 的所有文件，直到找到匹配的。（事实上他只支持 .html 和 .xml 两种）

完整程序：

```
#-*-coding:utf-8-*-
import web
web.config.debug = True
urls = (
    "/", "index"
)
app = web.application(urls, locals())
render = web.template.render('template', globals)

class index:
    def GET(self):
        return render.index('楚广明 6666')
```

```
if __name__ == "__main__":  
    app.run()
```

模板：

```
$def with (name)  
<!DOCTYPE html>  
<html>  
<head>  
    <title></title>  
</head>  
<body>  
<h1>我是$name</h1>  
</body>  
</html>
```

除了上面的使用方式，你也可以直接用文件的方式来处理模板 `frender`：

```
hello = web.template.frender('templates/hello.html')  
render hello('world')
```

直接使用字符串方式：

```
template = "$def with (name)\nHello $name"  
hello = web.template.Template(template)  
return hello('world')
```

### 3. 表达式用法

特殊字符 `$` 被用于特殊的 `python` 表达式。表达式能够被用于一些确定的组合当中 `()` 和 `{}`：

```
Look, a $string.  
Hark, an ${arbitrary + expression}.  
Gawk, a $dictionary[key].function('argument').  
Cool, a $(limit)ing.
```

### 4. 赋值

有时你可能需要定义一个新变量或给一些变量重新赋值，如下：

```
$ bug = get_bug(id)  
<h1>$bug.title</h1>  
<div>  
    $bug.description  
</div>
```

**注意** `$`在赋值变量名称之前要有一个空格，这有区别于常规的赋值用法。

**完整程序：**

```
#-*-coding:utf-8-*-  
import web  
web.config.debug = True
```

```
urls = (
    "/", "index"
)
app = web.application(urls, locals())
render = web.template.render('template', globals)

class index:
    def GET(self):
        return render.index('楚广明 6666')

if __name__ == "__main__":
    app.run()
```

```
$def with (name)
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
<h1>我是$name</h1>
$def get_bug(id='1'):
    Hello $name!
$ bug = get_bug(id)
<h1>$bug</h1>
</body>
</html>
```

## 5. 过滤

模板默认会使用 `web.websafe` 过滤 `html` 内容(`encoding` 处理)。

```
>>> render.hello("1 < 2")
"Hello 1 &lt; 2"
```

不需要过滤可以在 `$` 之后 使用 `:`。示例：该 `Html` 内容不会被转义

```
$:form.render()
```

## 6. 新起一行用法

在行末添加 `\` 代表显示层该内容不会被真实处理成一行。

```
If you put a backslash \
at the end of a line \
(like these) \
then there will be no newline.
```

## 7. 转义 \$

使用 `$$` 可以在输出的时候显示字符 `$`。

```
Can you lend me $$50?
```

## 8. 注释

`$#` 是注释指示符。任何以 `$#` 开始的某行内容都被当做注释。

```
$# this is a comment  
Hello $name.title()! $# display the name in title case
```

## 9. 控制结构

模板系统支持 `for`, `while`, `if`, `elif` 和 `else`。像 `python` 一样，这里是需要缩进的。

```
$for i in range(10):  
    I like $i  
  
$for i in range(10): I like $i  
$ a= [1,2,3,4,5]  
$while a:  
    hello $a.pop()  
$ times=100  
$ max=200  
$if times > max:  
    Stop! In the name of love.  
$else:  
    Keep on, you can do it.
```

`for` 循环内的成员变量只在循环内发生可用：

```
loop.index: the iteration of the loop (1-indexed)  
loop.index0: the iteration of the loop (0-indexed)  
loop.first: True if first iteration  
loop.last: True if last iteration  
loop.odd: True if an odd iteration  
loop.even: True if an even iteration  
loop.parity: "odd" or "even" depending on which is true  
loop.parent: the loop above this in nested loops
```

有时候，他们使用起来很方便：

```
<table>  
$for c in ["a", "b", "c", "d"]:  
    <tr class="$loop.parity">  
        <td>$loop.index</td>  
        <td>$c</td>  
    </tr>  
</table>
```



## 10. 使用 `def`

可以使用 `$def` 定义一个新的模板函数，支持使用参数。

```
$def say_hello(name='world'):  
    Hello $name!  
  
$say_hello('web.py')  
$say_hello()
```

其他示例：

```
$def tr(values):  
    <tr>  
    $for v in values:  
        <td>$v</td>  
    </tr>  
  
$def table(rows):  
    <table>  
    $for row in rows:  
        $:row  
    </table>  
  
$ data = [['a', 'b', 'c'], [1, 2, 3], [2, 4, 6], [3, 6, 9] ]  
$:table([tr(d) for d in data])
```

代码

可以在 `code` 块书写任何 `python` 代码：

```
$code:  
    x = "you can write any python code here"  
    y = x.title()  
    z = len(x + y)  
  
    def limit(s, width=10):  
        """limits a string to the given width"""  
        if len(s) >= width:  
            return s[:width] + "..."  
        else:  
            return s  
  
And we are back to template.  
The variables defined in the code block can be used here.  
For example, $limit(x)
```

## 11. 使用 `var`

`var` 块可以用来定义模板结果的额外属性：

```
$def with (title, body)
```

```
$var title: $title
$var content_type: text/html

<div id="body">
$body
</div>
```

以上模板内容的输出结果如下：

```
>>> out = render.page('hello', 'hello world')
>>> out.title
u'hello'
>>> out.content_type
u'text/html'
>>> str(out)
'\n\n<div>\nhello world\n</div>\n'
```

## 12. 内置 和 全局

像 python 的任何函数一样，模板系统同样可以使用内置以及局部参数。很多内置的公共方法像 `range`, `min`, `max` 等，以及布尔值 `True` 和 `False`，在模板中都是可用的。部分内置和全局对象也可以使用在模板中。全局对象可以使用参数方式传给模板，使用 `web.template.render`：

```
import web
import markdown

globals = {'markdown': markdown.markdown}
render = web.template.render('templates', globals=globals)
```

内置方法是否可以在模板中也是可以控制的：

```
# 禁用所有内置方法

render = web.template.render('templates', builtins={})
```

这个比较难理解，我可以给出一个例子：

模板很简单 `index.html`

```
$:title
```

`code.py`

```
#-*- coding:utf-8 -*-
import web
import sys
#如果页面有修改可以即时体现
reload(sys)
sys.setdefaultencoding('utf-8')
#定义 url，将地址映射到对应的类
urls = (
    "/", "index",
)
app = web.application(urls, globals())

def render(params={}, partial=False):
```

```
global_vars = dict(params.items())

if partial:
    #print "true"
    return web.template.render('templates/', globals=global_vars)
else:
    #print "false"
    return web.template.render('templates/', base='layout', globals=global_vars)
#定义 index 类
class index:
    #get 请求
    def GET(self):
        return render({'title': '这是一个测试'}, True).index()

if __name__ == "__main__":
    app.run()
```

## 2) 使用 Jinja2 模板语法

### 1. 上手的一个小例子

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-
import web
# 引入 web.py 对调用 jinja2 的模块
from web.contrib.template import render_jinja

urls = (
    "/", "index",
)
app = web.application(urls, globals())

render = render_jinja(
    'templates', # 模板存放的目录名称
    encoding='utf-8', # 模板使用的编码
)

class index:

    def GET(self):
        return render.index(word="Hello World") # 使用模板目录下的 index.html 模板

if __name__ == "__main__":
```

```
app.run()
```

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h1>{{ word }}</h1>
</body>
</html>
```

## 十九、 Web.py 中使用数据库

### 1) 使用 web.py 自带的 DB 封装库

首先需要大家用 `easy_install` 安装 `mysql` 的驱动，也可以自行去下载软件。有一一定要注意就是 64 位的驱动与 32 位的是不兼容的，这一点要注意了。

`web.py` 是一个非常精巧的 `web` 框架，不过其自带的 `db` 模块也是非常精简而高效。

#### 1. 一个简单的数据库访问测试

```
#-*-coding:utf-8-*-
import sys
import web
#定义数据库对象 db
db = web.database(dbn='mysql', db='todolist', user='root', pw='chu123')
#定义要操作的数据库名称
tb = 'todo'
def get_by_id(id):
    s = db.select(tb, where='id=$id', vars=locals())
    if not s:
        return False
    return s[0]
if __name__=="__main__":
    resultValue=get_by_id(4)
    print resultValue
```

如果语句里面使用了\$符号，就要添加 `vars=locals()` 这一句，不然可能会报错。里面的 `$id` 是传递过来的 `id` 变量。

## 2. 一个复杂一点的例子

```
#!/usr/bin/env python
#-*-coding:utf-8-*-
import sys
import web
import time
#定义数据库对象 db
db = web.database(dbn='mysql', db='todolist', user='root', pw='chu123')
#定义要操作的数据库名称
tb = 'todo'
def getTableInfo(id=1):
    result = db.query('select * from todo where id>$id', vars=locals())
    for s in result:
        print s.id, s.title
if __name__=="__main__":
    #简单查询
    result = db.query('select * from todo where id>$id', vars={'id':1})
    for s in result:
        print s.id, s.title
    #插入,第一参数是表名 todo
    db.insert('todo',
title='Michael',finished=0,post_date=time.strftime('%Y-%m-%d',time.localtime(time.time(
))))
    getTableInfo(5)
    #更新
    db.update('todo', where='id=$id', vars={'id':5}, title='楚广明')
    getTableInfo(5)
    #删除
    db.delete('todo', where='id=$id', vars={'id':6})
    getTableInfo(5)
```

## 3. 更加复杂的例子

```
#!/usr/bin/env python
#-*-coding:utf-8-*-
import sys
import web
import time
#定义数据库对象 db
db = web.database(dbn='mysql', db='todolist', user='root', pw='chu123')
#定义要操作的数据库名称
tb = 'todo'
if __name__=="__main__":
    # 查询表
    entries = db.select('todo')
```

```
# where 条件
myvar = dict(title="楚广明")
results = db.select('todo', myvar, where="title = $title")
for s in results:
    print s.id
results = db.select('todo', where="id>100")
for s in results:
    print s.id
# 查询具体列
results = db.select('todo', what="id,title")
# order by
results = db.select('todo', order="post_date DESC")
# group
results = db.select('todo', group="title")
# limit
results = db.select('todo', limit=10)
#update
db.update('todo', where="id = 10", title = "foo")
#delete
db.delete('todo', where="id=10")
#复杂
# count
results = db.query("SELECT COUNT(*) AS total_users FROM todo")
print results[0].total_users

# join
#results = db.query("SELECT * FROM entries JOIN users WHERE entries.author_id =
users.id")

# 防止 SQL 注入可以这么干
#results = db.query("SELECT * FROM users WHERE id=$id", vars={'id':10})

'''多数据库操作
db1 = web.database(dbn='mysql', db='dbname1', user='foo')
db2 = web.database(dbn='mysql', db='dbname2', user='foo')

print db1.select('foo', where='id=1')
print db2.select('bar', where='id=5')
'''

'''事务
t = db.transaction()
try:
    db.insert('person', name='foo')
    db.insert('person', name='bar')
except:
    t.rollback()
    raise
```

```
else:
    t.commit()

# Python 2.5+ 可以用 with
from __future__ import with_statement
with db.transaction():
    db.insert('person', name='foo')
    db.insert('person', name='bar')
...
```

## 二十、使用 SQLAlchemy

### 1) 简介

web 开发中一般的主要版块就是视图、模板、数据库操作，前面已经讲了最基础的视图跟模板，这一章主要是介绍下数据库操作，如第一章前言所说，我们的数据库操作是使用 **sqlalchemy** 来实现的。

对象关系映射器（Object Relational Mappers, ORM）在过去数年吸引了不少人的目光。主要原因是 ORM 经常会在 Web 应用程序框架中被提起，因为它是快速开发（Rapid Development）栈中的关键组件。**Django** 和 **Ruby on Rails** 等 Web 框架采用了设计一个独立栈的方法，将自主开发的 ORM 紧密集成到该框架中。而其他框架，如 **Pylons**、**Turbogears** 和 **Grok**，则采用更加基于组件的架构结合可交换的第三方组件。两种方法都有各自的优势：紧密集成允许非常连贯的体验（如果问题映射到框架），而基于组件的架构则允许最大的设计灵活性。但是，本文的主题并不是 Web 框架；而是 **SQLAlchemy**。

**SQLAlchemy** 在构建在 **WSGI** 规范上的下一代 **Python Web** 框架中得到了广泛应用，它是由 **Mike Bayer** 和他的核心开发人员团队开发的一个单独的项目。使用 ORM 等独立 **SQLAlchemy** 的一个优势就是它允许开发人员首先考虑数据模型，并能决定稍后可视化数据的方式（采用命令行工具、Web 框架还是 **GUI** 框架）。这与先决定使用 Web 框架或 **GUI** 框架，然后再决定如何在框架允许的范围内使用数据模型的开发方法极为不同。

不同于很多 web 开发，python 开发 web 的数据库很多是通过建立数据表类，绑定数据库连接，然后自动创建数据表，这种设计很大程度上降低了在开发阶段修改数据库结构的麻烦，而且由于表是对应一个个类创建而成，可以很清晰的在任何时候了解到该 app 的数据库是如何设计的，这对维护的意义也是非常大的。

#### 什么是 WSGI？

**WSGI** 是下一代 **Python Web** 框架、应用程序和服务端应该遵循的规范。**WSGI** 中一个有趣的方面是创建 **Python** 中间件，并在使用 **Python** 或任何语言创建的 Web 应用程序中使用。请参阅 参考资料，获取关于 **WSGI** 和 **WSGI** 社区（**Pytypefitters**）的大量链接。

**SQLAlchemy** 的一个目标是提供能兼容众多数据库（如 **SQLite**、**MySQL**、**Postgres**、**Oracle**、**MS-SQL**、**SQLServer** 和 **Firebird**）的企业级持久性模型。**SQLAlchemy** 正处于积极开发阶段，当前最新的 API 将围绕版本 0.5 设计。请参阅参考资料部分，获取官方 API 文档、教程和 **SQLAlchemy** 书籍的链接。

**SQLAlchemy** 取得成功的一个证明就是围绕它已建立了丰富的社区。针对 **SQLAlchemy** 的扩展和插件包括：**declarative**、**Migrate**、**Elixir**、**SQLSoup**、**django-sqlalchemy**、**DBSprockets**、**FormAlchemy** 和 **z3c.sqlalchemy**。在本文中，我们将学习一篇关于新 0.5 API 的教程，探究一些第三方库，以及如何在 **Pylons** 中使用它们。

#### 谁是 Mike Bayer？

**Michael Bayer** 是居住在纽约的一名软件承包商，他拥有十余年处理各类关系数据库的经验。他曾使用 **C**、**Java™** 和 **Perl** 编写了许多自主研发的数据库抽象层，并在 **Major League Baseball** 与大量多服务器 **Oracle** 系统打了多年交道，借助这些经验，他成功编写了“终极工具包” **SQLAlchemy**，用于生成 **SQL** 和处理数据库。

其目标是贡献一个世界级、独树一帜的面向 Python 的工具包，以帮助 Python 成为一个广泛普及的编程平台。

#### 例子一

```
# -*- coding:utf-8 -*-
from sqlalchemy import *
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import scoped_session, sessionmaker

db_config = {
    'host': 'localhost',
    'user': 'root',
    'passwd': 'sdss',
    'db': 'test',
    'charset': 'utf8'
}

engine = create_engine('mysql://%s:%s@%s/%s?charset=%s'%(db_config['user'],
                                                         db_config['passwd'],
                                                         db_config['host'],
                                                         db_config['db'],
                                                         db_config['charset'])), echo=True)

metadata = MetaData()
users_table = Table('users222', metadata,
    Column('id', Integer, primary_key=True),
    Column('name', String(50)),
    Column('fullname', String(50)),
    Column('password', String(100))
)
metadata.create_all(engine)
```

#### 例子二

```
# -*- coding:utf-8 -*-
from sqlalchemy import *
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import scoped_session, sessionmaker

#定义数据库的账号、端口、密码、数据库名，使用的连接模块，这里用的是 mysqldb
engine = create_engine(
    'mysql+mysqldb://root:dddd123@localhost:3306/todo?charset=utf8',
    echo=True#是否输出数据库操作过程，很方便调试
)

Base = declarative_base()

class User(Base):
    __tablename__ = "user2222"
    id = Column(Integer, primary_key=True)
    name = Column(String(20), unique=True)
    email = Column(String(32), unique=True)
    password = Column(String(32))
```



```
superuser = Column(Boolean, default=False)

metadata = Base.metadata

if __name__ == "__main__":
    metadata.create_all(engine)#运行 python models.py 就会自动创建定义的所有表
```

以上就是 User 表的定义，可以在 User 类里非常清晰的看出这个数据表是如何设计的。接下来先在 mysql 创建一个叫"tech"的数据库，接着只要在命令行运行 `python models.py` 就可以看到数据库中出现了 User 表了，用数据库操作工具看看是否与定义的内容一样吧：)

## 2) 初始化 Sqlite 数据库

```
from sqlalchemy import create_engine
from sqlalchemy import Column, Integer, String
from sqlalchemy.ext.declarative import declarative_base

engine = create_engine('sqlite:///mydatabase.db', echo=True)
Base = declarative_base()

class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    name = Column(String)
    fullname = Column(String)
    password = Column(String)

    def __init__(self, name, fullname, password):
        self.name = name
        self.fullname = fullname
        self.password = password

    def __repr__(self):
        return "<User('%s','%s', '%s')>" % (self.name, self.fullname, self.password)

users_table = User.__table__
metadata = Base.metadata
if __name__ == "__main__":
    metadata.create_all(engine)
```

### 3) 插入数据库

```
# -*- coding:utf-8 -*-
from sqlalchemy import *
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import scoped_session, sessionmaker

# 定义数据库的账号、端口、密码、数据库名，使用的连接模块，这里用的是 mysqldb

db_config = {
    'host': 'localhost',
    'user': 'root',
    'passwd': 'sdss',
    'db': 'todo',
    'charset': 'utf8'
}

engine = create_engine('mysql://%s:%s@%s/%s?charset=%s'%(db_config['user'],
                                                         db_config['passwd'],
                                                         db_config['host'],
                                                         db_config['db'],
                                                         db_config['charset']),
                      echo=True)

# 定义一个函数，用来获取 sqlalchemy 的 session
def bindSQL():
    return scoped_session(sessionmaker(bind=engine))

Base = declarative_base()

class User(Base):
    __tablename__ = "user22"
    id = Column(Integer, primary_key=True)
    name = Column(String(20), unique=True)
    email = Column(String(32), unique=True)
    password = Column(String(32))
    superuser = Column(Boolean, default=False)
    def __init__(self, name='', email='', password='', superuser=True):
        self.name = name
        self.email = email
        self.password = password
        self.superuser = superuser

metadata = Base.metadata
```

```
if __name__ == "__main__":
    metadata.create_all(engine) # 运行 python models.py 就会自动创建定义的所有表
    my_vp = User('楚广明', 'ss3林@sss.com', '22月333', False)
    session = bindSQL()
    session.add(my_vp)
    session.commit()
```

除了将 `user` 表创建出来，我们并没有进行其他的操作，而往往我们的 `web` 开发，总需要在网站中设计一个超级管理员，这个超级管理员不是注册出来的，而是在数据库创建初期就有的，那么，我们可以在 `models.py` 中创建 `user` 表，并同时向 `user` 表插入超级管理员的信息，而在插入超级管理员或者以后的修改密码、新增用户的时候，我们的密码都是需要加密的，那么这时就可以添加一个事件触发，当插入设置 `password` 字段时，自动加密，好，需求有了，接下来就是行动了，我们把原来的 `models.py` 修改成以下代码：

```
# -*- coding:utf-8 -*-
from sqlalchemy import *
from sqlalchemy import event
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import scoped_session, sessionmaker
import hashlib

# 这里定义一个 password 加密混淆
password_prefix = "Ad%cvcsadefr^!deaf"

# 定义数据库的账号、端口、密码、数据库名，使用的连接模块，
# 这里用的是 mysqldb
engine = create_engine(
    'mysql+mysqldb://root:dddd123@localhost:3306/todo?charset=utf8',
    echo=True # 是否输出数据库操作过程，很方便调试
)

# 定义一个函数，用来获取 sqlalchemy 的 session

def bindSQL():
    return scoped_session(sessionmaker(bind=engine))

Base = declarative_base()
# 定义数据表使用 InnoDB
Base.__table_args__ = {'mysql_engine': 'InnoDB'}

class User(Base):
    __tablename__ = "user5555"
    id = Column(Integer, primary_key=True)
    name = Column(String(20), unique=True)
    email = Column(String(32), unique=True)
```

```
password = Column(String(32))
superuser = Column(Boolean, default=False)

metadata = Base.metadata

# 定义一个回调函数用于响应触发事件

def setPassword(target, value, oldvalue, initiator):
    # 如果新设置的值与原有的值相等，那么说明用户并没有修改密码，返回原先的值
    if value == oldvalue:
        return oldvalue
    # 如果新值与旧值不同，说明密码发生改变，进行加密，加密方法可以根据自己需求改变
    return hashlib.md5("%s%s" % (password_prefix, value)).hexdigest()
# 设置事件监听，event.listen(表单或表单字段，触发事件，回调函数，是否改变插入值)
event.listen(User.password, "set", setPassword, retval=True)

# 为了避免重复插入数据，定义一个 get_or_create 函数
def get_or_create(session, model, **kwargs):
    if "defaults" in kwargs:
        defaults = kwargs["defaults"]
        del kwargs["defaults"]
    else:
        defaults = {}

    instance = session.query(model).filter_by(**kwargs).first()
    if instance:
        return instance, False
    else:
        kwargs.update(defaults)
        instance = model(**kwargs)
        session.add(instance)
        session.flush()
        session.refresh(instance)
        return instance, True

# 定义初始化函数

def initModel():
    metadata.create_all(engine) # 创建数据库
    db = bindSQL() # 获取 sqlalchemy 的 session
    # 创建超级管理员，这里为了避免多次运行 initModel
    # 而发生重复插入的情况，使用了 get_or_create 方法
    obj, created = get_or_create(
        db,
        User,
        name="administrator",
```

```
defaults={
    "email": "332535694@qq.com",
    "password": "administrator",
    "superuser": True
}

)
db.commit() # 记得 commit 喔，不然数据最后还是没插入
db.remove()

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

## 4) 在视图中使用 sqlalchemy

### 1. 首先创建数据库 models.py

```
from sqlalchemy import create_engine
from sqlalchemy import Column, Integer, String
from sqlalchemy.ext.declarative import declarative_base

engine = create_engine('sqlite:///mydatabase.db', echo=True)
Base = declarative_base()

class User(Base):
    __tablename__ = 'users'

    id = Column(Integer, primary_key=True)
    name = Column(String)
    fullname = Column(String)
    password = Column(String)

    def __init__(self, name, fullname, password):
        self.name = name
        self.fullname = fullname
        self.password = password

    def __repr__(self):
        return "<User('%s', '%s', '%s')>" % (self.name, self.fullname, self.password)

users_table = User.__table__
metadata = Base.metadata
if __name__ == "__main__":
    metadata.create_all(engine)
```

## 2. 使用 app.py

```
import string
import random
import web

from sqlalchemy.orm import scoped_session, sessionmaker
from models import *

urls = (
    "/", "add",
    "/view", "view"
)

def load_sqla(handler):
    web.ctx.orm = scoped_session(sessionmaker(bind=engine))
    try:
        return handler()
    except web.HTTPError:
        web.ctx.orm.commit()
        raise
    except:
        web.ctx.orm.rollback()
        raise
    finally:
        web.ctx.orm.commit()

app = web.application(urls, globals())
app.add_processor(load_sqla)

class add:
    def GET(self):
        web.header('Content-type', 'text/html')
        fname = "".join(random.choice(string.letters) for i in range(4))
        lname = "".join(random.choice(string.letters) for i in range(7))
        u = User(name=fname, fullname=fname + ' ' + lname, password=542)
        web.ctx.orm.add(u)
        return "added:" + web.websafe(str(u)) \
            + "<br/>" \
            + '<a href="/view">view all</a>'

class view:
```

```
def GET(self):
    web.header('Content-type', 'text/plain')
    return "\n".join(map(str, web.ctx.orm.query(User).all()))

if __name__ == "__main__":
    app.run()
```

## 二十一、Json 的使用

首先 Python 内置的 json 包是支持 json 的解析的，我们平时如果不涉及复杂的业务，这个内置的包功能完全够用。先上一个例子

```
#!/usr/bin/python
#coding:utf-8
import json
#Function:Analyze json script
#Json is a script can descript data structure as xml,
#for detail, please refer to "http://json.org/json-zh.html".

#Note:
#1.Also, if you write json script from python,
#you should use dump instead of load. please refer to "help(json)".

#json file:
#The file content of temp.json is:
#{
# "name":"00_sample_case1",
# "description":"an example."
#}
#f = file("temp.json");
#s = json.load(f)
#print s
#f.close

#json string:
s = json.loads('{"name":"test", "type":{"name":"seq", "parameter":["1", "2"]}}')
print s
print s.keys()
print s["name"]
print s["type"]["name"]
print s["type"]["parameter"][1]
```

## 二十二、Urllib 模块

### 1) urlopen 的基本使用

urllib 模块提供接口用来打开 URL, 让我们可以访问 **www** 和 **ftp** 上的数据并且可以像访问本地文件一样操作他们。

```
urllib.urlopen(url[, data[, proxies]]) :
```

url: 表示远程数据的路径

data: 以 post 方式提交到 url 的数据

proxies: 用于设置代理

urllib.urlopen(url[, data[, proxies]]) 这个方法打开一个 URL 并返回远程 url 的类文件对象，我们可以像操作本地文件一样操作 url 类的文件对象获取数据，其中参数 url 是远程数据的地址（URL），data 是以 post 方式提交到服务器的数据，proxies 是设置代理端口的；

urlopen 返回的类文件对象有以下方法：

- 1 read(), readline(), readlines() 这些方法跟操作文件一样；
- 2 fileno() 以整数返回文件描述符；
- 3 close() 关闭链接；
- 4 getcode() 返回 HTTP 响应码，例如：成功会返回 200，未找到文件的返回 404；
- 5 geturl() 用于返回请求的 URL；

```
#coding:utf-8
import urllib
import sys
google=urllib.urlopen("http://www.baidu.com")
#获取服务器的表头信息
print "this is a header:\n%s"%google.info()
print "-----"
#返回整数状态码
print "this is a status:\n%s"%google.getcode()
print "-----"
#返回 url
print "this is a url:\n%s"%google.geturl()
print "-----"
#返回文件描述信息
print google.read().decode('utf-8')
#print google.read()
#获取系统默认编码
#print sys.getdefaultencoding()
```



## 2) urllibretrieve 方法的使用

```
urllib.urlretrieve(url[, filename[, reporthook[, data]]]):  
filename 指定保存到本地的路径（若未指定该，urllib 生成一个临时文件保存数据）  
reporthook 回调函数，当连接上服务器、以及相应的数据块传输完毕的时候会触发该回调
```

data 指 post 到服务器的数据

urllib.urlretrieve(url[, filename[, reporthook[, data]]) 此函数将远程的数据下载到本地，其中参数 url 是 URL 字符串，filename 指定数据保存到本地的路径；reporthook(block, size, total) 是一个回调函数当链接上服务器，以及相应的数据传送完毕时会触发该函数（即每下载一块(block)就调用一次回调函数），我们可以用该函数来显示当前数据下载的进度，也可以用来限速，data 是指 post 到服务器的数据；该方法返回一个包含两个元素的元组(filename, headers)，filename 表示保存到本地的路径，header 表示服务器的响应头。

```
from __future__ import unicode_literals  
import urllib  
import re  
def callback_f(downloaded_size, block_size, remote_total_size):  
    per = 100.0 * downloaded_size * block_size / remote_total_size  
    if per > 100:  
        per = 100  
    print "%.2f%%" % per  
def getHtml(url):  
    page=urllib.urlopen(url)  
    html=page.read()  
    page.close()  
    return html  
def getImg(html):  
    imgre=re.compile(r'<img\s.*?\s?src\s*=\s*["\']?([^"\']+).*?>', re.I)  
    imglist=imgre.findall(html)  
    i=0  
    for imgurl in imglist:  
        if not imgurl.find("http://"):  
            urllib.urlretrieve(imgurl, "img/%d.jpg"%(i), callback_f)  
            i+=1  
        print imgurl  
ht=getHtml(r"http://www.cnbeta.com/articles/234303.htm")  
getImg(ht)
```

## 3) url 编码

- 1 urllib.quote(str[, safe]) 对字符串进行编码，而参数 safe 指定不需要编码的字符；
- 2 urllib.unquote(str) 对字符串进行解码；
- 3 urllib.quote\_plus(str[, safe]) 和 quote 类似，只是把参数 str 中的空格转化成“+” 而 quote 是把

空格转化成“%20”

4 urllib.unquote\_plus(str) 对字符串进行解码；

5 urllib.urlencode(query[,dosep]) 参数 query 可以是两个元素的元组也可以是一个字典,将其转成 url；

6 urllib.url2pathname(path) 将 url 路径转化成本地路径；

7 urllib.pathname2url 将本地路径转化成 url 路径；

```
#coding:utf-8
import urllib
date="whoiam=i am student+i like python"
#对字符串编码
date1=urllib.quote(date)
print date1
#对字符串解码
print urllib.unquote(date1)
print urllib.unquote_plus(date1)
#将参数字典转成 url
date5=urllib.urlencode({"name":"python","love":"python"})
print date5
```

## 4) 使用 httplib 抓取

```
httplib.HTTPConnection ( host [ , port [ ,strict [ , timeout ] ] ] )
```

host 表示服务器主机

port 为端口号，默认值为 80

strict 的 默认值为 false，表示在无法解析服务器返回的状态行时(status line)（比较典型的状态行如：HTTP/1.0 200 OK），是否抛 BadStatusLine 异常

可选参数 timeout 表示超时时间。

HTTPConnection 提供的方法：

```
- HTTPConnection.request ( method , url [ ,body [ , headers ] ] )
```

调用 request 方法会向服务器发送一次请求

method 表示请求的方法，常用有方法有 get 和 post ；

url 表示请求的资源的 url ；

body 表示提交到服务器的数据，必须是字符串（如果 method 是” post”，则可以把 body 理解为 html 表单中的数据）；

headers 表示请求的 http 头。

```
- HTTPConnection.getresponse ()
```

获取 Http 响应。返回的对象是 HTTPResponse 的实例，关于 HTTPResponse 在下面会讲解。

```
- HTTPConnection.connect ()
```

连接到 Http 服务器。

```
- HTTPConnection.close ()
```

关闭与服务器的连接。

- `HTTPConnection.set_debuglevel ( level )`

设置高度的级别。参数 `level` 的默认值为 `0`，表示不输出任何调试信息。

### `httplib.HTTPResponse`

-`HTTPResponse` 表示服务器对客户端请求的响应。往往通过调用 `HTTPConnection.getresponse()` 来创建，它有如下方法和属性：

-`HTTPResponse.read([amt])`

获取响应的消息体。如果请求的是一个普通的网页，那么该方法返回的是页面的 `html`。可选参数 `amt` 表示从响应流中读取指定字节的数据。

-`HTTPResponse.getheader(name[, default])`

获取响应头。`Name` 表示头域(`header field`)名，可选参数 `default` 在头域名不存在的情况下作为默认值返回。

-`HTTPResponse.getheaders()`

以列表的形式返回所有的头信息。

-`HTTPResponse.msg`

获取所有的响应头信息。

-`HTTPResponse.version`

获取服务器所使用的 `http` 协议版本。`11` 表示 `http/1.1`；`10` 表示 `http/1.0`。

-`HTTPResponse.status`

获取响应的状态码。如：`200` 表示请求成功。

-`HTTPResponse.reason`

返回服务器处理请求的结果说明。一般为“OK”

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

```
# -*- coding:utf-8 -*-
```

```
def use_httplib():
```

```
    import httplib
```

```
    conn = httplib.HTTPConnection("www.baidu.com")
```

```
    i_headers = {"User-Agent":
```

```
                "Mozilla/5.0 (Windows; U; Windows NT 5.1; zh-CN; rv:1.9.1) Gecko/20090624
```

```
Firefox/3.5",
```

```
                "Accept": "text/plain"}
```

```
    conn.request("GET", "/", headers=i_headers)
```

```
    r1 = conn.getresponse()
```

```
    print "version:", r1.version
```

```
    print "-----"
```

```
    print "reason:", r1.reason
```

```
print "-----"
print "status:", r1.status
print "-----"
print "msg:", r1.msg
print "-----"
print "headers:", r1.getheaders()
print "-----"
data = r1.read()
print data.decode("utf-8")
print "-----"
print "data length:%s"%len(data)
conn.close()

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

## 二十三、Urllib2 的使用

### 1) 最简单的爬虫

网络爬虫是一个自动提取网页的程序，它为搜索引擎从万维网上下载网页，是搜索引擎的重要组成。python 的 urllib\urllib2 等模块很容易实现这一功能，下面的例子实现的是对 baidu 首页的下载。具体代码如下：

```
import urllib2
page=urllib2.urlopen("http://www.baidu.com")
print page.read().decode('utf-8')
```

### 2) 提交表单数据

#### 1. 用 GET 方法提交数据

提交表单的 GET 方法是把表单数据编码至 URL。在给出请示的页面后，加上问号，接着是表单的元素。

如在百度中搜索“马伊琍”得到 url 为：

```
http://www.baidu.com/s?wd=%E9%A9%AC%E4%BC%8A%E7%90%8D&pn=100&rn=20&ie=utf-8&usm=4&rsv\_page=1
```

其中？后面为表单元素。wd=%E9%A9%AC%E4%BC%8A%E7%90%8D 表示搜索的词是“马伊琍”，pn 表示从第 100 条信息所在页开始显示（感觉是这样，我试了几次，当写 100 时，从其所在页显示，但如果写 10，就是从第 1 页显示），rn=20 表示每页显示 20 条，ie=utf-8 表示编码格式，usm=4 没明白是什么意思，换了 1、2、3 试了下，没发现什么变化，rsv\_page=1 表示第几页。如果要下载以上页面比较简单的方法是直接用上面的网址进行提取。如代码：

```
#coding:utf-8
import urllib2
```

```
keyword=urllib2.quote('马伊琍')
page=urllib2.urlopen("http://www.baidu.com/s?wd="+keyword+"&pn=100&rn=20&ie=utf-8&usm=4&rsv_page=1")
print page.read()
```

## 二十四、 操作 Memcache

简单介绍: **memcached** 很强大, 它可以支持分布式的共享内存缓存, 大型站点都用它。对小站点来说, 有足够内存的话, 使用它也可以得到超赞的效果。

### 1) 安装

#### 1. Linux 环境

##### 安装包

对于大多数 Linux 发行版本来说, 可以使用官方推荐的方法:

```
Debian/Ubuntu
apt-get install memcached
```

```
Redhat/Fedora/CentOS
yum install memcached
```

#### 2. Win 环境

##### 1. 下载 memcache 的 windows 稳定版

下载地址 <http://jehiah.cz/projects/memcached-win32/files/memcached-1.2.1-win32.zip>

解压放某个盘下面, 比如在 c:\memcached

#### 3. linux 启动 memcached

启动参数说明:

```
-d 选项是启动一个守护进程

-m 是分配给 Memcache 使用的内存数量, 单位是 MB, 默认 64MB

-M return error on memory exhausted (rather than removing items)
```

```
-u 是运行 Memcache 的用户，如果当前为 root 的话，需要使用此参数指定用户

-l 是监听的服务器 IP 地址，默认为所有网卡

-p 是设置 Memcache 的 TCP 监听的端口，最好是 1024 以上的端口

-c 选项是最大运行的并发连接数，默认是 1024

-P 是设置保存 Memcache 的 pid 文件

-f chunk size growth factor (default: 1.25)

-I Override the size of each slab page. Adjusts max item size(1.4.2 版本新增)
```

```
/usr/local/memcached/bin/memcached -d -m 100 -c 1000 -u root -p 11211
```

可以启动多个守护进程，但是端口不能重复。设置开机启动的话可以将上行命令增加到/etc/rc.d/rc.local 文件中。

## 4. windows 下启动

在终端（也即 cmd 命令界面）cd 到解压目录（这里是 c:\memcached），运行 memcached.exe -d install 安装服务

运行 memcached.exe -d start，memcached 会使用默认的端口(11211)来启动，你可以在任务管理器中看到 memcached.exe

```
-p 监听的端口
-l 连接的 IP 地址，默认是本机
-d start 启动 memcached 服务
-d restart 重起 memcached 服务
-d stop|shutdown 关闭正在运行的 memcached 服务
-d install 安装 memcached 服务
-d uninstall 卸载 memcached 服务
-u 以的身份运行（仅在以 root 运行的时候有效）
-m 最大内存使用，单位 MB。默认 64MB
-M 内存耗尽时返回错误，而不是删除项
-c 最大同时连接数，默认是 1024
-f 块大小增长因子，默认是 1.25
-n 最小分配空间，key+value+flags 默认是 48
-h 显示帮助
```

## 2) Python 操作 Memcached

memcached API 地址 <http://code.google.com/p/memcached/wiki/Clients>

网上流传说 Python-API 中效率最高的是 python-libmemcached，这里居然看到了 hongqn（豆瓣首席架构师，后来也得到证实 python-libmemcached 是豆瓣贡献），看来豆瓣的阳光真的是撒满了 Python 的各个角落。

另外还有 python-memcached（100%纯 Python），python-memcache（据说有内存泄漏问题？），cmemcache（代码多年未更新？）。

不过由于目前需要中效率并没有太高要求，于是选择了使用最多的 `python-memcached`：

安装 `python-memcached`

```
easy_install python-memcached
```

## 1. Python 操作 memcached

```
import memcache
mc = memcache.Client(['127.0.0.1:11211'], debug=True)
mc.set('name', 'luo', 60)
print mc.get('name')
```

## 2. Memcached 常用方法

`memcache` 其实是一个 `map` 结构，最常用的几个函数：

保存数据

```
set(key,value,timeout) 把 key 映射到 value, timeout 指的是什么时候这个映射失效
add(key,value,timeout) 仅当存储空间中不存在键相同的数据时才保存
replace(key,value,timeout) 仅当存储空间中存在键相同的数据时才保存
```

获取数据

```
get(key) 返回 key 所指向的 value
get_multi(key1,key2,key3,key4) 可以非同步地同时取得多个键值，比循环调用 get 快数十倍
```

删除数据

```
delete(key, timeout) 删除键为 key 的数据，timeout 为时间值，禁止在 timeout 时间内名为 key 的键保存新数据（set 函数无效）。
```

# 二十五、 SQLite 模块

## 1) 简单的介绍

`SQLite` 数据库是一款非常小巧的嵌入式开源数据库软件，也就是说没有独立的维护进程，所有的维护都来自于程序本身。它是遵守 `ACID` 的关联式数据库管理系统，它的设计目标是嵌入式的，而且目前已经在很多嵌入式产品中使用了它，它占用资源非常的低，在嵌入式设备中，可能只需要几百 K 的内存就够了。它能够支持 `Windows/Linux/Unix` 等等主流的操作系统，同时能够跟很多程序语言相结合，比如 `Tcl`、`C#`、`PHP`、`Java` 等，还有 `ODBC` 接口，同样比起 `Mysql`、`PostgreSQL` 这两款开源世界著名的数据库管理系统来讲，它的处理速度比他们都快。`SQLite` 第一个 `Alpha` 版本诞生于 2000 年 5 月。至今已经有 10 个年头，`SQLite` 也迎来了一个版本 `SQLite 3` 已经发布。

## 2) 安装与使用

### 1. 导入 Python SQLITE 数据库模块

Python2.5 之后，内置了 SQLite3，成为了内置模块，这给我们省了安装的功夫，只需导入即可~

```
import sqlite3
```

### 2. 创建/打开数据库

在调用 `connect` 函数的时候，指定库名称，如果指定的数据库存在就直接打开这个数据库，如果不存在就新建一个再打开。

```
cx = sqlite3.connect("E:/test.db")
```

也可以创建数据库在内存中。

```
con = sqlite3.connect(":memory:")
```

### 3. 数据库连接对象

打开数据库时返回的对象 `cx` 就是一个数据库连接对象，它可以有以下操作：

- `commit()`--事务提交
- `rollback()`--事务回滚
- `close()`--关闭一个数据库连接
- `cursor()`--创建一个游标

关于 `commit()`，如果 `isolation_level` 隔离级别默认，那么每次对数据库的操作，都需要使用该命令，你也可以设置 `isolation_level=None`，这样就变为自动提交模式。

### 4. 使用游标查询数据库

我们需要使用游标对象 SQL 语句查询数据库，获得查询对象。 通过以下方法来定义一个游标。

```
cu=cx.cursor()
```

- 游标对象有以下的操作：
- `execute()`--执行 sql 语句
- `executemany`--执行多条 sql 语句
- `close()`--关闭游标
- `fetchone()`--从结果中取一条记录，并将游标指向下一条记录
- `fetchmany()`--从结果中取多条记录
- `fetchall()`--从结果中取出所有记录
- `scroll()`--游标滚动

#### 1. 建表

```
cu.execute("create table catalog (id integer primary key,pid integer,name varchar(10) UNIQUE,nickname text NULL)")
```

上面语句创建了一个叫 `catalog` 的表，它有一个主键 `id`，一个 `pid`，和一个 `name`，`name` 是不可以重复的，以及一个 `nickname` 默认为 `NULL`。

#### 2. 插入数据

请注意避免以下写法：

```
# Never do this -- insecure 会导致注入攻击
pid=200
c.execute("... where pid = '%s'" % pid)
```

正确的做法如下，如果 `t` 只是单个数值，也要采用 `t=(n,)` 的形式，因为元组是不可变的。

```
for t in[(0,10,'abc','Yu'),(1,20,'cba','Xu')]:
```



```
cx.execute("insert into catalog values (?, ?, ?, ?)", t)
```

简单的插入两行数据,不过需要提醒的是,只有提交了之后,才能生效.我们使用数据库连接对象 **cx** 来进行提交 **commit** 和回滚 **rollback** 操作.

```
cx.commit()
```

### 3. 查询

```
cu.execute("select * from catalog")
```

要提取查询到的数据,使用游标的 **fetch** 函数,如:

```
In [10]: cu.fetchall()
Out[10]: [(0, 10, u'abc', u'Yu'), (1, 20, u'cba', u'Xu')]
```

如果我们使用 **cu.fetchone()**,则首先返回列表中的第一项,再次使用,则返回第二项,依次下去.

### 4. 修改

```
In [12]: cu.execute("update catalog set name='Boy' where id = 0")
In [13]: cx.commit()
```

注意,修改数据以后提交

### 5. 删除

```
cu.execute("delete from catalog where id = 1")
cx.commit()
```

### 6. 使用中文

请先确定你的 IDE 或者系统默认编码是 **utf-8**,并且在中文前加上 **u**

```
x=u'鱼'
cu.execute("update catalog set name=? where id = 0",x)
cu.execute("select * from catalog")
cu.fetchall()
[(0, 10, u'\u9c7c', u'Yu'), (1, 20, u'cba', u'Xu')]
```

如果要显示出中文字体,那需要依次打印出每个字符串

```
In [26]: for item in cu.fetchall():
.....:     for element in item:
.....:         print element,
.....:     print
.....:
0 10 鱼 Yu
1 20 cba Xu
```

游标提供了一种对从表中检索出的数据进行操作的灵活手段,就本质而言,游标实际上是一种能从包括多条数据记录的结果集中每次提取一条记录的机制。游标总是与一条 **SQL** 选择语句相关联。因为游标由结果集（可以是零条、一条或由相关的选择语句检索出的多条记录）和结果集中指向特定记录的游标位置组成。当决定对结果集进行处理时,必须声明一个指向该结果集的游标。如果曾经用 **C** 语言写过对文件进行处理的程序,那么游标就像您打开文件所得到的文件句柄一样,只要文件打开成功,该文件句柄就可代表该文件。对于游标而言,其道理是相同的。可见游标能够实现按与传统程序读取平面文件类似的方式处理来自基础表的结果集,从而把表中数据以平面文件的形式呈现给程序。

我们知道关系数据库管理系统实质是面向集合的,在 **Sqlite** 中并没有一种描述表中单一记录的表达形式,除非使用 **where** 子句来限制只有一条记录被选中。因此我们必须借助于游标来进行面向单条记录的数据处理。由此可见,游标允许应用程序对查询语句 **select** 返回的行结果集中每一行进行相同或不同的操作,而不是一次对整个结果集进行同一种操作;它还提供对基于游标位置而对表中数据进行删除或更新的能力;正是游标把作为面向集合的数据库管理系统和面向行的程序设计两者联系起来,使两个数据处理方式能够进行沟通。

### 3) 一个简单的例子

```
#coding:utf-8
import sqlite3
import sys

def sqlite_basic():
    # Connect to db
    conn = sqlite3.connect('test1.db')
    # create cursor
    c = conn.cursor()
    # Create table
    c.execute('''
        create table if not exists stocks
        (date text, trans text, symbol text,
        qty real, price real)
        ''')

    # Insert a row of data
    c.execute('''
        insert into stocks
        values ('2006-01-05', 'BUY', '楚', 100, 35.14)
        ''')

    # query the table
    rows = c.execute("select * from stocks")
    # print the table
    for row in rows:
        print(row)
    # delete the row
    c.execute("delete from stocks where symbol=='REHT'")
    # Save (commit) the changes
    conn.commit()
    # Close the connection
    conn.close()

def sqlite_adv():
    conn = sqlite3.connect('test2.db')
    c = conn.cursor()
    c.execute('''
        create table if not exists employee
        (id text, name text, age inteage)
        ''')

    # insert many rows
    for t in [('1', 'itech', 10),
              ('2', 'jason', 10),
```

```
        ('3', 'jack', 30),
]:
    c.execute('insert into employee values (?, ?, ?)', t)
# create index
create_index = 'CREATE INDEX IF NOT EXISTS idx_id ON employee (id);'
c.execute(create_index)
# more secure
t = ('jason',)
c.execute('select * from employee where name=?', t)
# fetch query result
for row in c.fetchall():
    print(row)
conn.commit()
conn.close()

def sqlite_adv2():
    reload(sys)
    sys.setdefaultencoding('utf-8')
    # memory db
    con = sqlite3.connect(":memory:")
    cur = con.cursor()
    # execute sql
    cur.executescript('''
create table book(
    title,
    author,
    published
);
insert into book(title, author, published)
values (
    'AAA book',
    'Douglas Adams', "kkk");
''')
    rows = cur.execute("select * from book")
    for row in rows:
        print("title:" + row[0])
        print("author:" + row[1])
        print("published:" + str(row[2]))

def sqlite_adv3():
    import datetime

    # Converting SQLite values to custom Python types
    # Default adapters and converters for datetime and timestamp
    con = sqlite3.connect(":memory:", detect_types=sqlite3.PARSE_DECLTYPES |
sqlite3.PARSE_COLNAMES)
```

```
cur = con.cursor()
cur.execute("create table test(d date, ts timestamp)")

today = datetime.date.today()
now = datetime.datetime.now()

cur.execute("insert into test(d, ts) values (?, ?)", (today, now))
cur.execute("select d, ts from test")
row = cur.fetchone()
print today, "=>", row[0], type(row[0])
print now, "=>", row[1], type(row[1])

cur.execute('select current_date as "d [date]", current_timestamp as "ts [timestamp]"
from test')
row = cur.fetchone()
print "current_date", row[0], type(row[0])
print "current_timestamp", row[1], type(row[1])

sqlite_basic()
```

## 4) 中文处理

# 二十六、 正则表达式

## 1) 简介

编写验证规则最流行和最简单的方法就是正则表达式了，但唯一的一个问题是正则表达式的语法太隐晦了，让人蛋疼无比。很多开发者为了在项目中应用复杂的验证，经常要使用一些小抄来记住正则式的复杂语法和各种常用命令。



也许你是初学者，那以防万一，我先来讲讲什么是正则表达式吧：

正则表达式可以帮助我们更好的描述复杂的文本格式。一旦你描述清楚了这些格式，那你就可以利用它们对文本数据进行检索、替换、提取和修改操作。

下面有一个正则表达式的简单例子。第一步先要引入有关正则式的包：

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

第二步就是用指定的正则式构建一个正则表达式对象，下面的正则式是用来搜索长度为 10 的 a-z 的英文字母：

```
# -*- coding: utf-8 -*-  
import re  
regex = re.compile('[a-z]{10}')
```

最后，根据正则式在指定数据中检索匹配项，如果匹配 `isMatch` 方法就会返回 `true`。

```
# -*- coding: utf-8 -*-  
import re  
regex = re.compile('[a-z]{10}')  
m1 = re.match(regex, '1234567890')  
m2 = re.match(regex, 'abcdefghig')  
# m = regex.search("test1234.567").group()  
if m1 is not None:  
    print 'True'  
if m2 is not None:  
    print 'True'
```

## 2) 3 个重要的正则式命令

记住正则语法最好的办法就是记住这三样东西：**Bracket**（括号），**caret**（插入符号）和 **Dollars**（美元符号）。

B for Brackets

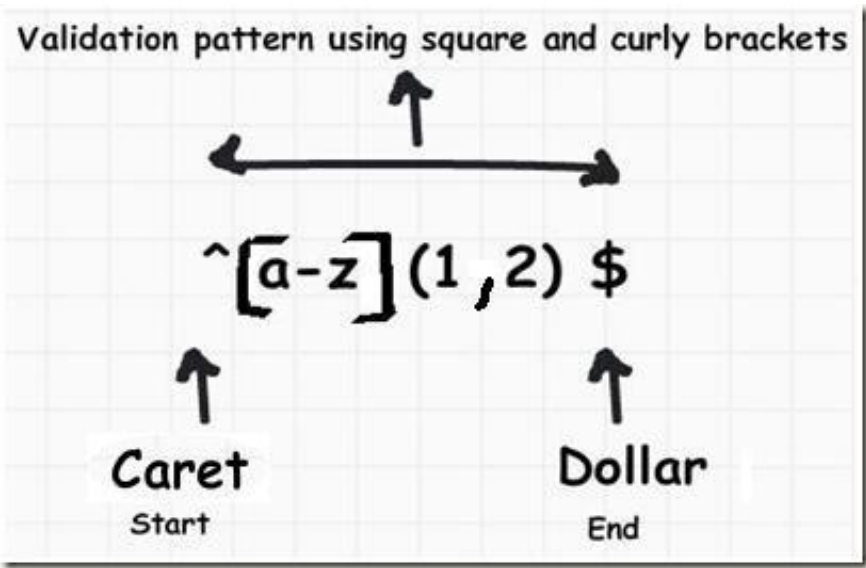
C for Caret

D for Dollar



B（括号）	在正则表达式中有 3 种类型的括号 方括号 "[" 和花括号 "{" 。 方括号 "[" 内是需要匹配的字符，花括号 "{" 内是指定匹配字符的数量。 圆括号 "(" 则是用来分组的。
C（插入符号）	插入符号 "^" 表示正则式的开始。
D（美元符号）	美元符号 "\$" 表示正则式的结束。

现在你知道上面的 3 个语法后，你就可以写世界上任何一条验证规则了。比如下面的例子就很好的说明了上面 3 条正则语法是如何协调运作的。



（注：上图有个错误，"()"应为"{}"）

- 上面的这条正则式只能匹配 a-z 的英文字母，同样是在中括号中标明匹配范围。
- 花括号中则是标明匹配字符串的最小长度和最大长度。

- 最后为了让表达式更规则，分别在开头和结尾加上了插入符号"^"和美元符号"\$"。

## 1. 验证字符串

好了，现在我们就用上面的 3 条语法来实现一些正则表达式的验证规则吧。

检查用户是否输入了 shivkoirala?

```
shivkoirala
```

让我们开始第一个验证，输入的字符在 a-g 之间？

```
[a-g]
```

输入的字符在 a-g 之间并且长度为 3？

```
[a-g]{3}
```

输入的字符在 a-g 之间并且最大长度为 3 最小长度为 1？

```
[a-g]{1,3}
```

我如何在匹配像 91230456, 01237648 那样的固定 8 位数？

```
^[0-9]{8}$
```

如何验证最小长度为 3 最大长度为 7 的数字，如：123, 1274667, 87654？

```
^[0-9]{3,7}$
```

如何验证像 LJI10201020 那样的发票编号，前 3 个是字母剩余为 8 位长度的数字？前三个是字母：

```
^[a-z]{3}
```

后面是 8 位长度的数字：

```
[0-9]{8}
```

所以整个表达式为：

```
^[a-z]{3}[0-9]{8}$
```

验证像 INV19020303 或 inv82083003 那样的前 3 位是不区分大小写的英文字母，剩余 8 位是数字在前面的表达式中只能匹配前 3 个是小写英文字母的发票编号，如果我们输入大写字母那就不能匹配了。所以为了确保前 3 个字母是不区分大小写的，我们就要用表达式^[a-zA-Z]{3}。

完整的正则式如下：

```
^[a-zA-Z]{3}[0-9]{8}$
```

## 2. 验证 url

我们可以验证简单的网址 URL 格式吗？

第一步：检查是否存在 www：

```
^www.
```

第二步：域名必须是长度在 1-15 的英文字母：

```
.[a-z]{1,15}
```

第三步：以 .com 或者 .org 结束：

```
.(com|org)$
```

完整的表达式如下：

```
^www.[a-z]{1,15}.(com|org)$
```

3. 验证 Email

让我们在来看看 BCD（其实也就是上面说的 3 条基本语法）如何验证 email 格式

第一步：email 开始是长度在 1-10 的英文字母，最后跟一个"@":

```
^[a-zA-Z0-9]{1,10}@
```

第二步：@后面是长度在 1-10 的英文字母，后面跟一个".":

```
[a-zA-Z]{1,10}.
```

第三步：最后以.com 或.org 结束:

```
.(com|org)$
```

最后完整的表达式如下:

```
^[a-zA-Z0-9]{1,10}@([a-zA-Z]{1,10}).(com|org)$
```

4. 验证值在 0-25 的数字

```
^((([0-9])|([0-1][0-9])|([0-2][0-5])))$
```

5. 验证格式为 MM/DD/YYYY, YYYY/MM/DD and DD/MM/YYYY 的日期

步骤	正则式	描述说明
先来检查 DD. 首先 DD 的长度为 1-29 ( 2 月份) , 1-30 (月小) , 1-31 (月大) . 所以 DD 就是 1-9 或 01-09	[1-9] 0[1-9]	允许用户输入 1-9 或者 01-09.
再为 DD 添加匹配 10-19	[1-9] 1[0-9]	允许用户输入 01-19.
再为 DD 添加匹配 20-29	[1-9] 1[0-9] 2[0-9]	允许用户输入 01-29.
i 再为 DD 添加匹配 30-31	[1-9] 1[0-9] 2[0-9] 3[0-1]	最后用户可以输入 01-31.
再来匹配日期期间的分隔符"/", "-"	[/ . -]	允许用户输入日期分隔符.
MM 也是类似的操作	[1-9] 0[1-9] 1[0-2]	让用户输入月份值 01-12.
最后就是 YY 的操作	1[9][0-9][0-9] 2[0][0-9][0-9]	允许用户输入年份 1900-2099.

最后 DD/MM/YYYY 格式的日期的正则表达式为:

```
^([1-9]|0[1-9]|1[0-9]|2[0-9]|3[0-1])[- / .]([1-9]|0[1-9]|1[0-2])[- / .](1[9][0-9][0-9]|2[0][0-9][0-9])$
```



MM/DD/YYYY 格式日期：

```
^([1-9]|0[1-9]|1[0-2])[- /.]([1-9]|0[1-9]|1[0-9]|2[0-9]|3[0-1])[- /.]([1-9]|0[0-9]|1[0-9]|2[0-9]|3[0-9])$
```

YYYY/MM/DD 格式日期：

```
^(1[9]|0[0-9]|2[0-9]|3[0-9])[- /.]([1-9]|0[1-9]|1[0-2])[- /.]([1-9]|0[1-9]|1[0-9]|2[0-9]|3[0-1])$
```

### 3) 正则表达式

#### 1. 你常用的元字符

代码	说明
.	匹配除换行符以外的任意字符
\w	匹配字母或数字或下划线或汉字
\s	匹配任意的空白符
\d	匹配数字
\b	匹配单词的开始或结束
^	匹配字符串的开始
\$	匹配字符串的结束

比如一个网站如果你填写的 QQ 号必须为 5 位到 12 位数字时，可以使用：

```
^d{5,12}$
```

#### 2. 常用的限定符

代码	说明
*	重复零次或更多次
+	重复一次或更多次
?	重复零次或一次
{n}	重复 n 次
{n,}	重复 n 次或更多次
{n,m}	重复 n 到 m 次

下面是一些使用重复的例子：

**Windows\d+** 匹配 **Windows** 后面跟 1 个或更多数字

**^w+** 匹配一行的第一个单词(或整个字符串的第一个单词，具体匹配哪个意思得看选项设置)

**^**: 匹配字符串的开始

**\$**: 匹配字符串的结尾

**\b**: 匹配一个单词的边界

**\d**: 匹配任意数字

**\D**: 匹配任意非数字字符

**x?**: 匹配一个可选的 x 字符（换句话说，它匹配 1 次或者 0 次 x 字符）

`x*`: 匹配 0 次或者多次 `x` 字符  
`x+`: 匹配 1 次或者多次 `x` 字符  
`x{n,m}`: 匹配 `x` 字符，至少 `n` 次，至多 `m` 次  
`{a|b|c}`: 要么匹配 `a`，要么匹配 `b`，要么匹配 `c`  
`(x)`: 一般情况下表示一个记忆组(`remembered group`)  
你可以利用 `re.search` 函数返回对象的 `groups()` 函数获取它的值

### 3. 字符串类

要想查找数字，字母或数字，空白是很简单的，因为已经有了对应这些字符集合的元字符，但是如果你想匹配没有预定义元字符的字符集合(比如元音字母 `a,e,i,o,u`)，应该怎么办？

很简单，你只需要在方括号里列出它们就行了，像 `[aeiou]` 就匹配任何一个英文元音字母，`[.?!]` 匹配标点符号(或?或!)。

我们也可以轻松地指定一个字符范围，像 `[0-9]` 代表的含意与 `\d` 就是完全一致的：一位数字；同理 `[a-z0-9A-Z_]` 也完全等同于 `\w`（如果只考虑英文的话）。

如果我们要匹配一个电话区号：像 `(010)88886666`，或 `022-22334455`，或 `02912345678` 等  
第一步：前面的区号可能有括号也可能没有，区号与电话号码之间可能有-号也可能没有

首先是一个转义字符 `\(`，它能出现 0 次或 1 次 `?`

然后是一个 `0`，后面跟着 2 个数字 `\d{2}`

然后是 `)` 或-或空格中的一个，它出现 1 次或不出现 `?`，最后是 8 个数字 `\d{8}`

```
\(?:\d{2}[- ]?)\d{8}
```

第二步：匹配后面的电话

```
\(?:\d{2}[- ]?)\d{8}
```

“(”和“)”也是元字符，后面的分组节里会提到，所以在这里需要使用转义。

这个表达式可以匹配几种格式的电话号码，像 `(010)88886666`，或 `022-22334455`，或 `02912345678` 等。我们对它进行一些分析吧：首先是一个转义字符 `\(`，它能出现 0 次或 1 次(`?`)，然后是一个 `0`，后面跟着 2 个数字 `\d{2}`，然后是 `)` 或-或空格中的一个，它出现 1 次或不出现(`?`)，最后是 8 个数字(`\d{8}`)。

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

regex = re.compile('\(?:\d{2}[- ]?)\d{8}')
m1 = re.search(regex, '(010)88886666')
m2 = re.search(regex, '022-22334455')
m3 = re.search(regex, '02912345678')
m4 = re.search(regex, '(010)88886666 022-22334455 02912345678')
if m4 is not None:
    print m4.group()
m5 = re.findall(regex, '(010)88886666 022-22334455 02912345678')
print m5
```

## 4. 分枝条件

不幸的是，刚才那个表达式也能匹配 `010)12345678` 或 `(022-87654321` 这样的“不正确”的格式。

要解决这个问题，我们需要用到分枝条件。正则表达式里的分枝条件指的是有几种规则，如果满足其中任何一种规则都应该当成匹配，具体方法是用 `|` 把不同的规则分隔开。听不明白？没关系，看例子：

`0\d{2}-\d{8}|0\d{3}-\d{7}` 这个表达式能匹配两种以连字号分隔的电话号码：一种是三位区号，8 位本地号如 `010-12345678`，一种是 4 位区号，7 位本地号 `0376-2233445`

`\(0\d{2}\)[- ]?\d{8}|0\d{2}[- ]?\d{8}` 这个表达式匹配 3 位区号的电话号码，其中区号可以用小括号括起来，也可以不用，区号与本地号间可以用连字号或空格间隔，也可以没有间隔。你可以试试用分枝条件把这个表达式扩展成也支持 4 位区号的。

`\d{5}-\d{4}|\d{5}` 这个表达式用于匹配美国的邮政编码。美国邮编的规则是 5 位数字，或者用连字号间隔的 9 位数字。之所以要给出这个例子是因为它能说明一个问题：使用分枝条件时，要注意各个条件的顺序。如果你把它改成 `\d{5}|\d{5}-\d{4}` 的话，那么就只会匹配 5 位的邮编（以及 9 位邮编的前 5 位）。原因是匹配分枝条件时，将会从左到右地测试每个条件，如果满足了某个分枝的话，就不会去再管其它的条件了。

## 5. 分组

我们已经提到了怎么重复单个字符（直接在字符后面加上限定符就行了）；但如果想要重复多个字符又该怎么办？你可以用小括号来指定子表达式（也叫做分组），然后你就可以指定这个子表达式的重复次数了，你也可以对子表达式进行其它一些操作（后面会有介绍）。

`(\d{1,3}\.){3}\d{1,3}` 是一个简单的 IP 地址匹配表达式。要理解这个表达式，请按下列

顺序分析它：

`\d{1,3}` 匹配 1 到 3 位的数字

`(\d{1,3}\.){3}` 匹配三位数字加上一个英文句号（这个整体也就是这个分组）重复 3 次

最后再加上一个一到三位的数字 `(\d{1,3})`

IP 地址中每个数字都不能大于 255

不幸的是，它也将匹配 `256.300.888.999` 这种不可能存在的 IP 地址。如果能使用算术比较的话，或许能简单地解决这个问题，但是正则表达式中并不提供关于数学的任何功能，所以只能使用冗长的分组，选择，字符类来描述一个正确的 IP 地址：

`((2[0-4]\d|25[0-5]|[01]?\d\d?)\.){3}(2[0-4]\d|25[0-5]|[01]?\d\d?)`

理解这个表达式的关键是理解 `2[0-4]\d|25[0-5]|([01])?\d\d?`，这里我就不细说了，你自己应该能分析得出来它的意义。

## 6. 反义

有时需要查找不属于某个能简单定义的字符类的字符。比如想查找除了数字以外，其它任意字符都行的情况，这时需要用到反义：

表 3.常用的反义代码	
代码/语法	说明
<code>\W</code>	匹配任意不是字母，数字，下划线，汉字的字符
<code>\S</code>	匹配任意不是空白符的字符
<code>\D</code>	匹配任意非数字的字符
<code>\B</code>	匹配不是单词开头或结束的位置
<code>[^x]</code>	匹配除了 <code>x</code> 以外的任意字符
<code>[^aeiou]</code>	匹配除了 <code>aeiou</code> 这几个字母以外的任意字符

例子：

`\S+` 匹配不包含空白符的字符串  
`<a[^>]+>` 匹配用尖括号括起来的以 `a` 开头的字符串

## 7. 后向引用

使用小括号指定一个子表达式后，匹配这个子表达式的文本（也就是此分组捕获的内容）可以在表达式或其它程序中作进一步的处理。默认情况下，每个分组会自动拥有一个组号，规则是：从左向右，以分组的左括号标志，第一个出现的分组的组号为 1，第二个为 2，以此类推。

- 分组 0 对应整个正则表达式
- 实际上组号分配过程是要从左向右扫描两遍的：第一遍只给未命名组分配，第二遍只给命名组分配——因此所有命名组的组号都大于未命名的组号
- 你可以使用 `(?:exp)` 这样的语法来剥夺一个分组对组号分配的参与权。

后向引用用于重复搜索前面某个分组匹配的文本。例如，`\1` 代表分组 1 匹配的文本。难以理解？请看示例：  
`\b(w+)\b\s+\1\b` 可以用来匹配重复的单词，像 `go go`，或者 `kitty kitty`。这个表达式首先是一个单词，也就是单词开始处和结束处之间的多于一个的字母或数字 `(\b(w+)\b)`，这个单词会被捕获到编号为 1 的分组中，然后是 1 个或几个空白符 `(\s+)`，最后是分组 1 中捕获的内容（也就是前面匹配的那个单词）`(\1)`。

你也可以自己指定子表达式的组名。要指定一个子表达式的组名，请使用这样的语法：`(?<Word>w+)`（或者把尖括号换成单引号也行：`(?'Word'w+)`），这样就把 `w+` 的组名指定为 `Word` 了。要反向引用这个分组捕获的内容，你可以使用 `\k<Word>`，所以上一个例子也可以写成这样：`\b(?<Word>w+)\b\s+\k<Word>\b`。  
 使用小括号的时候，还有很多特定用途的语法。下面列出了最常用的一些：

表 4.常用分组语法		
分类	代码/语法	说明
捕获	<code>(exp)</code>	匹配 <code>exp</code> ，并捕获文本到自动命名的组里
	<code>(?&lt;name&gt;exp)</code>	匹配 <code>exp</code> ，并捕获文本到名称为 <code>name</code> 的组里，也可以写成 <code>(?'name'exp)</code>
	<code>(?:exp)</code>	匹配 <code>exp</code> ，不捕获匹配的文本，也不给此分组分配组号
零宽断言	<code>(?=exp)</code>	匹配 <code>exp</code> 前面的位置

	(?<=exp)	匹配 exp 后面的位置
	(?!exp)	匹配后面跟的不是 exp 的位置
	(?<!exp)	匹配前面不是 exp 的位置
注释	(?#comment)	这种类型的分组不对正则表达式的处理产生任何影响，用于提供注释让人阅读

我们已经讨论了前两种语法。第三个(? :exp)不会改变正则表达式的处理方式，只是这样的组匹配的内容不会像前两种那样被捕获到某个组里面，也不会拥有组号。“我为什么会想要这样做？”——好问题，你觉得为什么呢？

## 8. 贪婪与懒惰

当正则表达式中包含能接受重复的限定符时，通常的行为是（在使整个表达式能得到匹配的前提下）匹配尽可能多的字符。以这个表达式为例：`a.*b`，它将会匹配最长的以 `a` 开始，以 `b` 结束的字符串。如果用它来搜索 `aabab` 的话，它会匹配整个字符串 `aabab`。这被称为**贪婪**匹配。

有时，我们更需要**懒惰**匹配，也就是匹配尽可能少的字符。前面给出的限定符都可以被转化为懒惰匹配模式，只要它后面加上一个问号`?`。这样`.*?`就意味着匹配任意数量的重复，但是在能使整个匹配成功的前提下使用最少的重复。现在看看懒惰版的例子吧：

`a.*?b` 匹配最短的，以 `a` 开始，以 `b` 结束的字符串。如果把它应用于 `aabab` 的话，它会匹配 `aab`（第一到第三个字符）和 `ab`（第四到第五个字符）。

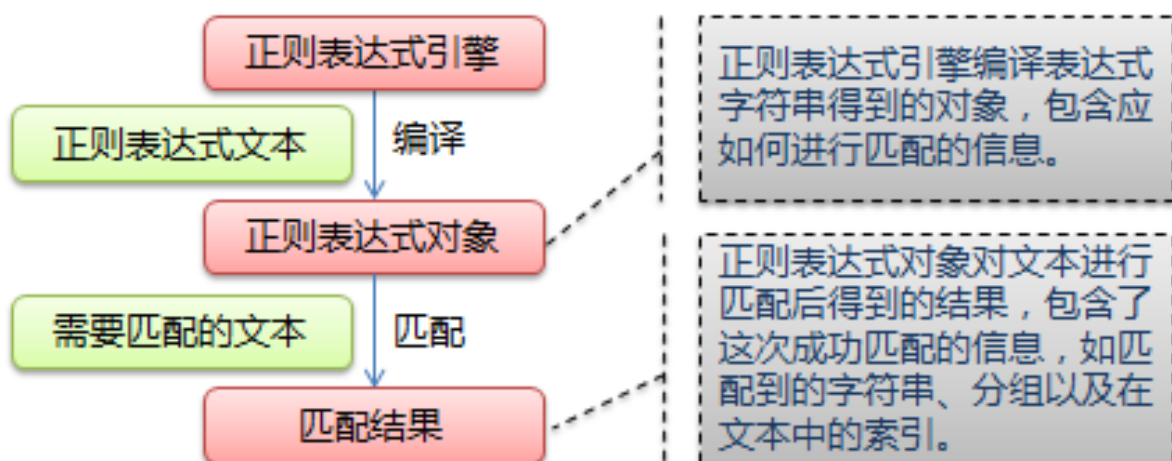
为什么第一个匹配是 `aab`（第一到第三个字符）而不是 `ab`（第二到第三个字符）？简单地说，因为正则表达式有另一条规则，比懒惰 / 贪婪规则的优先级更高：最先开始的匹配拥有最高的优先权——**The match that begins earliest wins.**

表 5. 懒惰限定符	
代码 / 语法	说明
<code>*?</code>	重复任意次，但尽可能少重复
<code>+?</code>	重复 1 次或更多次，但尽可能少重复
<code>??</code>	重复 0 次或 1 次，但尽可能少重复
<code>{n,m}?</code>	重复 n 到 m 次，但尽可能少重复
<code>{n,}??</code>	重复 n 次以上，但尽可能少重复

## 4) 在 Python 中使用

正则表达式并不是 Python 的一部分。正则表达式是用于处理字符串的强大工具，拥有自己独特的语法以及一个独立的处理引擎，效率上可能不如 `str` 自带的方法，但功能十分强大。得益于这一点，在提供了正则表达式的语言里，正则表达式的语法都是一样的，区别只在于不同的编程语言实现支持的语法数量不同；但不用担心，不被支持的语法通常是不常用的部分。如果已经在其他语言里使用过正则表达式，只需要简单看一看就可以上手了。

下图展示了使用正则表达式进行匹配的流程：



正则表达式的大致匹配过程是：依次拿出表达式和文本中的字符比较，如果每一个字符都能匹配，则匹配成功；一旦有匹配不成功的字符则匹配失败。如果表达式中有量词或边界，这个过程会稍微有一些不同，但也是很好理解的，看下图中的示例以及自己多使用几次就能明白。

下图列出了 Python 支持的正则表达式元字符和语法：

语法	说明	表达式实例	完整匹配的字符串
字符			
一般字符	匹配自身	abc	abc
.	匹配任意除换行符"\n"外的字符。 在DOTALL模式中也能匹配换行符。	a.c	abc
\	转义字符，使后一个字符改变原来的意思。 如果字符串中有字符*需要匹配，可以使用\*或者字符集[*]。	a\.c a\\c	a.c a\c
[...]	字符集（字符类）。对应的位置可以是字符集中任意字符。 字符集中的字符可以逐个列出，也可以给出范围，如[abc]或[a-c]。第一个字符如果是^则表示取反，如[^abc]表示不是abc的其他字符。 所有的特殊字符在字符集中都失去其原有的特殊含义。在字符集中如果要使用]、-或^，可以在前面加上反斜杠，或把]、-放在第一个字符，把^放在非第一个字符。	a[bcd]e	abe ace ade
预定义字符集（可以写在字符集[...]中）			
\d	数字：[0-9]	a\dc	a1c
\D	非数字：[^\d]	a\Dc	abc
\s	空白字符：[<空格>\t\r\n\f\v]	a\s c	a c
\S	非空白字符：[^\s]	a\S c	abc
\w	单词字符：[A-Za-z0-9_]	a\wc	abc
\W	非单词字符：[^\w]	a\Wc	a c
数量词（用在字符或(...)之后）			
*	匹配前一个字符0或无限次。	abc*	ab abccc
+	匹配前一个字符1次或无限次。	abc+	abc abccc
?	匹配前一个字符0次或1次。	abc?	ab abc
{m}	匹配前一个字符m次。	ab{2}c	abbc
{m,n}	匹配前一个字符m至n次。 m和n可以省略：若省略m，则匹配0至n次；若省略n，则匹配m至无限次。	ab{1,2}c	abc abbc
*? +? ?? {m,n}?	使 * + ? {m,n}变成非贪婪模式。	示例将在下文中介绍。	



边界匹配（不消耗待匹配字符串中的字符）			
<code>^</code>	匹配字符串开头。 在多行模式中匹配每一行的开头。	<code>^abc</code>	abc
<code>\$</code>	匹配字符串末尾。 在多行模式中匹配每一行的末尾。	<code>abc\$</code>	abc
<code>\A</code>	仅匹配字符串开头。	<code>\Aabc</code>	abc
<code>\Z</code>	仅匹配字符串末尾。	<code>abc\Z</code>	abc
<code>\b</code>	匹配\w和\W之间。	<code>a\b!bc</code>	a!bc
<code>\B</code>	<code>[^\b]</code>	<code>a\Bbc</code>	abc
逻辑、分组			
<code> </code>	代表左右表达式任意匹配一个。 它总是先尝试匹配左边的表达式，一旦成功匹配则跳过匹配右边的表达式。 如果 没有被包括在()中，则它的范围是整个正则表达式。	<code>abc def</code>	abc def
<code>(...)</code>	被括起来的表达式将作为分组，从表达式左边开始每遇到一个分组的左括号'('，编号+1。 另外，分组表达式作为一个整体，可以后接数量词。表达式中的 仅在该组中有效。	<code>(abc){2}</code> <code>a(123 456)c</code>	abccabc a456c
<code>(?P&lt;name&gt;...)</code>	分组，除了原有的编号外再指定一个额外的别名。	<code>(?P&lt;id&gt;abc){2}</code>	abccabc
<code>\&lt;number&gt;</code>	引用编号为<number>的分组匹配到的字符串。	<code>(\d)abc\1</code>	1abc1 5abc5
<code>(?P=name)</code>	引用别名为<name>的分组匹配到的字符串。	<code>(?P&lt;id&gt;\d)abc(?P=id)</code>	1abc1 5abc5
特殊构造（不作为分组）			
<code>(?:...)</code>	(...)的不分组版本，用于使用' '或后接数量词。	<code>(?:abc){2}</code>	abccabc
<code>(?iLmsux)</code>	iLmsux的每个字符代表一个匹配模式，只能用在正则表达式的开头，可选多个。匹配模式将在下文中介绍。	<code>(?i)abc</code>	AbC
<code>(?#...)</code>	#后的内容将作为注释被忽略。	<code>abc(?#comment)123</code>	abc123
<code>(?=...)</code>	之后的字符串内容需要匹配表达式才能成功匹配。 不消耗字符串内容。	<code>a(=?\d)</code>	后面是数字的a
<code>(?!...)</code>	之后的字符串内容需要不匹配表达式才能成功匹配。 不消耗字符串内容。	<code>a(?!\d)</code>	后面不是数字的a
<code>(?&lt;=...)</code>	之前的字符串内容需要匹配表达式才能成功匹配。 不消耗字符串内容。	<code>(?&lt;=\d)a</code>	前面是数字的a
<code>(?&lt;!=...)</code>	之前的字符串内容需要不匹配表达式才能成功匹配。 不消耗字符串内容。	<code>(?&lt;!\d)a</code>	前面不是数字的a
<code>(?(id/name)yes-pattern no-pattern)</code>	如果编号为id/别名为name的组匹配到字符，则需要匹配yes-pattern，否则需要匹配no-pattern。  no-pattern可以省略。	<code>(\d)abc(?:\d abc)</code>	1abc2 abccabc

<http://www.cnblogs.com/huxi>

## 5) Re 模块

Python 通过 `re` 模块提供对正则表达式的支持。使用 `re` 的一般步骤是先将正则表达式的字符串形式编译为 `Pattern` 实例，然后使用 `Pattern` 实例处理文本并获得匹配结果（一个 `Match` 实例），最后使用 `Match` 实例获得信息，进行其他的操作。

```
# encoding: UTF-8
import re

# 将正则表达式编译成 Pattern 对象
pattern = re.compile(r'hello')

# 使用 Pattern 匹配文本，获得匹配结果，无法匹配时将返回 None
match = pattern.match('hello world!')

if match:
    # 使用 Match 获得分组信息
    print match.group()

### 输出 ###
# hello
```

### re.compile(strPattern[, flag]):

这个方法是 **Pattern** 类的工厂方法，用于将字符串形式的正则表达式编译为 **Pattern** 对象。第二个参数 **flag** 是匹配模式，取值可以使用按位或运算符'|'表示同时生效，比如 **re.I** | **re.M**。另外，你也可以在 **regex** 字符串中指定模式，比如 **re.compile('pattern', re.I | re.M)**与 **re.compile('(?im)pattern')**是等价的。可选值有：

- **re.I**(**re.IGNORECASE**): 忽略大小写（括号内是完整写法，下同）
- **M**(**MULTILINE**): 多行模式，改变'^'和'\$'的行为（参见上图）
- **S**(**DOTALL**): 点任意匹配模式，改变'.'的行为
- **L**(**LOCALE**): 使预定字符类 **\w** **\W** **\b** **\B** **\s** **\S** 取决于当前区域设定
- **U**(**UNICODE**): 使预定字符类 **\w** **\W** **\b** **\B** **\s** **\S** **\d** **\D** 取决于 **unicode** 定义的字符属性
- **X**(**VERBOSE**): 详细模式。这个模式下正则表达式可以是多行，忽略空白字符，并可以加入注释。以下两个正则表达式是等价的：

```
# encoding: UTF-8
import re

a = re.compile(r"""\d + # the integral part
               \.    # the decimal point
               \d *  # some fractional digits""", re.X)
b = re.compile(r"\d+\.\d*")
```

**re** 提供了众多模块方法用于完成正则表达式的功能。这些方法可以使用 **Pattern** 实例的相应方法替代，唯一的好处是少写一行 **re.compile()**代码，但同时也无法复用编译后的 **Pattern** 对象。这些方法将在 **Pattern** 类的实例方法部分一起介绍。如上面这个例子可以简写为：

```
m = re.match(r'hello', 'hello world!')
print m.group()
```

**re** 模块还提供了一个方法 **escape(string)**，用于将 **string** 中的正则表达式元字符如 **\***/**+**/**?**等之前加上转义符再返回，在需要大量匹配元字符时有那么一点用。

## 6) Match 方法



**Match** 对象是一次匹配的结果，包含了很多关于此次匹配的信息，可以使用 **Match** 提供的可读属性或方法来获取这些信息。

属性：

- **string**: 匹配时使用的文本。
- **re**: 匹配时使用的 **Pattern** 对象。
- **pos**: 文本中正则表达式开始搜索的索引。值与 **Pattern.match()**和 **Pattern.seach()**方法的同名参数相同。
- **endpos**: 文本中正则表达式结束搜索的索引。值与 **Pattern.match()**和 **Pattern.seach()**方法的同名参数相同。
- **lastindex**: 最后一个被捕获的分组在文本中的索引。如果没有被捕获的分组，将为 **None**。
- **lastgroup**: 最后一个被捕获的分组的别名。如果这个分组没有别名或者没有被捕获的分组，将为 **None**。

方法：

- **group([group1, ...])**:  
获得一个或多个分组截获的字符串；指定多个参数时将以元组形式返回。**group1** 可以使用编号也可以使用别名；编号 **0** 代表整个匹配的子串；不填写参数时，返回 **group(0)**；没有截获字符串的组返回 **None**；截获了多次的组返回最后一次截获的子串。
- **groups([default])**:  
以元组形式返回全部分组截获的字符串。相当于调用 **group(1,2,...last)**。**default** 表示没有截获字符串的组以这个值替代，默认为 **None**。
- **groupdict([default])**:  
返回以有别名的组的别名为键、以该组截获的子串为值的字典，没有别名的组不包含在内。**default** 含义同上。
- **start([group])**:  
返回指定的组截获的子串在 **string** 中的起始索引（子串第一个字符的索引）。**group** 默认值为 **0**。
- **end([group])**:  
返回指定的组截获的子串在 **string** 中的结束索引（子串最后一个字符的索引+1）。**group** 默认值为 **0**。
- **span([group])**:  
返回(**start(group)**, **end(group)**)。
- **expand(template)**:  
将匹配到的分组代入 **template** 中然后返回。**template** 中可以使用 **\id** 或 **\g<id>**、**\g<name>** 引用分组，但不能使用编号 **0**。**\id** 与 **\g<id>** 是等价的；但 **\10** 将被认为是第 **10** 个分组，如果你想表达 **\1** 之后是字符 **'0'**，只能使用 **\g<1>0**。

```
# encoding: UTF-8
import re
m = re.match(r'(\w+) (\w+)(?P<sign>.*)', 'hello world! sign')

print "m.string:", m.string
print "m.re:", m.re
print "m.pos:", m.pos
print "m.endpos:", m.endpos
print "m.lastindex:", m.lastindex
print "m.lastgroup:", m.lastgroup

print "m.group(1,2):", m.group(1, 2)
print "m.groups():", m.groups()
print "m.groupdict():", m.groupdict()
print "m.start(2):", m.start(2)
print "m.end(2):", m.end(2)
print "m.span(2):", m.span(2)
print r"m.expand(r'\2 \1\3'):", m.expand(r'\2 \1\3')
```

```
#### output ####
# m.string: hello world!
# m.re: <_sre.SRE_Pattern object at 0x016E1A38>
# m.pos: 0
# m.endpos: 12
# m.lastindex: 3
# m.lastgroup: sign
# m.group(1,2): ('hello', 'world')
# m.groups(): ('hello', 'world', '!')
# m.groupdict(): {'sign': '!'}
# m.start(2): 6
# m.end(2): 11
# m.span(2): (6, 11)
# m.expand(r'\2 \1\3'): world hello!
```

## 7) Pattern 方法

Pattern 对象是一个编译好的正则表达式，通过 Pattern 提供的一系列方法可以对文本进行匹配查找。

Pattern 不能直接实例化，必须使用 `re.compile()` 进行构造。

Pattern 提供了几个可读属性用于获取表达式的相关信息：

- `pattern`: 编译时用的表达式字符串。
- `flags`: 编译时用的匹配模式。数字形式。
- `groups`: 表达式中分组的数量。
- `groupindex`: 以表达式中有别名的组的别名为键、以该组对应的编号为值的字典，没有别名的组不包含在内。

```
# encoding: UTF-8
import re
p = re.compile(r'(\w+) (\w+)(?P<sign>.*)', re.DOTALL)

print "p.pattern:", p.pattern
print "p.flags:", p.flags
print "p.groups:", p.groups
print "p.groupindex:", p.groupindex

#### output ####
# p.pattern: (\w+) (\w+)(?P<sign>.*)
# p.flags: 16
# p.groups: 3
# p.groupindex: {'sign': 3}
```

实例方法[ | re 模块方法]:

- **`match(string[, pos[, endpos]])` | `re.match(pattern, string[, flags])`:**

这个方法将从 `string` 的 `pos` 下标处起尝试匹配 `pattern`；如果 `pattern` 结束时仍可匹配，则返回一个 `Match` 对象；如果匹配过程中 `pattern` 无法匹配，或者匹配未结束就已到达 `endpos`，则返回 `None`。

`pos` 和 `endpos` 的默认值分别为 0 和 `len(string)`；`re.match()` 无法指定这两个参数，参数 `flags` 用于编译 `pattern` 时指定匹配模式。注意：这个方法并不是完全匹配。当 `pattern` 结束时若 `string` 还有剩余字符，仍然视为成功。想要完全匹配，可以在表达式末尾加上边界匹配符 `'$'`。

- **`search(string[, pos[, endpos]])` | `re.search(pattern, string[, flags])`:**

这个方法用于查找字符串中可以匹配成功的子串。从 `string` 的 `pos` 下标处起尝试匹配 `pattern`，如果 `pattern` 结束时仍可匹配，则返回一个 `Match` 对象；若无法匹配，则将 `pos` 加 1 后重新尝试匹配；直到 `pos=endpos` 时仍无法匹配则返回 `None`。

`pos` 和 `endpos` 的默认值分别为 0 和 `len(string)`；`re.search()` 无法指定这两个参数，参数 `flags` 用于编译 `pattern` 时指定匹配模式。

```
# encoding: UTF-8
import re

# 将正则表达式编译成 Pattern 对象
pattern = re.compile(r'world')

# 使用 search() 查找匹配的子串，不存在能匹配的子串时将返回 None
# 这个例子中使用 match() 无法成功匹配
match = pattern.search('hello world!')

if match:
    # 使用 Match 获得分组信息
    print match.group()

### 输出 ###
# world
```

- **`split(string[, maxsplit])` | `re.split(pattern, string[, maxsplit])`:**

按照能够匹配的子串将 `string` 分割后返回列表。`maxsplit` 用于指定最大分割次数，不指定将全部分割。

```
import re

p = re.compile(r'\d+')
print p.split('one1two2three3four4')

### output ###
# ['one', 'two', 'three', 'four', '']
```

- **`findall(string[, pos[, endpos]])` | `re.findall(pattern, string[, flags])`:**

搜索 `string`，以列表形式返回全部能匹配的子串。

```
# encoding: UTF-8
import re

p = re.compile(r'\d+')
print p.findall('one1two2three3four4')

### output ###
# ['1', '2', '3', '4']
```

## **finditer(string[, pos[, endpos]]) | re.finditer(pattern, string[, flags]):**

搜索 string，返回一个顺序访问每一个匹配结果（Match 对象）的迭代器。

```
# encoding: UTF-8
import re

p = re.compile(r'\d+')
for m in p.finditer('one1two2three3four4'):
    print m.group(),

### output ###
# 1 2 3 4
```

## **sub(repl, string[, count]) | re.sub(pattern, repl, string[, count]):**

使用 repl 替换 string 中每一个匹配的子串后返回替换后的字符串。

当 repl 是一个字符串时，可以使用\id 或\g<id>、\g<name>引用分组，但不能使用编号 0。

当 repl 是一个方法时，这个方法应当只接受一个参数（Match 对象），并返回一个字符串用于替换（返回的字符串中不能再引用分组）。

count 用于指定最多替换次数，不指定时全部替换。

```
# encoding: UTF-8
import re

p = re.compile(r'(\w+) (\w+)')
s = 'i say, hello world!'

print p.sub(r'\2 \1', s)

def func(m):
    return m.group(1).title() + ' ' + m.group(2).title()

print p.sub(func, s)

### output ###
# say i, world hello!
# I Say, Hello World!
```

## **subn(repl, string[, count]) | re.subn(pattern, repl, string[, count]):**

返回 (sub(repl, string[, count]), 替换次数)。

```
# encoding: UTF-8
import re

p = re.compile(r'(\w+) (\w+)')
s = 'i say, hello world!'

print p.subn(r'\2 \1', s)

def func(m):
    return m.group(1).title() + ' ' + m.group(2).title()
```

```
print p.subn(func, s)
```

```
### output ###
```

```
# ('say i, world hello!', 2)
```

```
# ('I Say, Hello World!', 2)
```

```
# -*- coding: utf-8 -*-
```

```
import re
```

```
#1. 匹配多个字符串
```

```
bt = 'bat|bet|bit'
```

```
m = re.search(bt, 'He bit bet')
```

```
if m is not None:
```

```
    print m.group()
```

```
m = re.findall(bt, 'He bit bet')
```

```
print m
```

```
#2. 匹配单个字符(.)
```

```
patt314 = '3.14'
```

```
pi_patt = '3\\.14'
```

```
m = re.match(pi_patt, '3.14')
```

```
if m is not None: print m.group()
```

```
m = re.match(patt314, '3014')
```

```
if m is not None: print m.group()
```

```
#3. 创建字符集合([])
```

```
m = re.match('[cr][23][dp][o2]', 'c3po')
```

```
if m is not None: print m.group()
```

```
#4. 重复、特殊字符和子组
```

```
patt = '\\w+@(?\\w+\\.)?\\w+\\.com'
```

```
print re.match(patt, 'nobody@XXX.com').group()
```

```
print re.match(patt, 'nobody@www.xxx.com').group()
```

```
patt = '\\w+@(?\\w+\\.)*\\w+\\.com'
```

```
print re.match(patt, 'nobody@www.xxx.yyy.com').group()
```

```
m = re.match('(\\w{3})-(\\d{3})', 'abc-123')
```

```
print m.group()      #所有匹配的部分,和 m.group(0)相同
```

```
print m.group(1)     #匹配的子组 1
```

```
print m.group(2)     #匹配的子组 2
```

```
print m.groups()     #所有匹配子组 ('abc', '123')
```

```
m = re.match('(a(b))', 'ab') #两个子组
```

```
print m.group(),m.group(1),m.group(2),m.groups()
```

```
#5. 从字符串的开头或结尾匹配及在单词边界上的匹配
```

```
re.search('^The', 'The end.') #匹配
```

```
re.search('^The', 'end.The')  #不匹配
```

```
re.search(r'\bthe','bit thedog')    #匹配,单词左边界
#r'\bthe'中的 r 表示原始字符串(raw strings),即\b 会被理解成一
#个正则表达式的一个特殊符号,而不是退格键
re.search(r'\bthe','bit the dog')    #匹配
re.search(r'\bthe','bitthe dog')     #不匹配
re.search(r'\Bthe','bitthe dog')     #匹配,单词右边界
re.search(r'\Bthe','bit the dog')    #不匹配

#6. 用 findall()找到每个出现的匹配部分
#结果: ['a','a','a']
print re.findall('c(.)r', 'carry the barcardi to the car')
#结果: ['car', 'car', 'car']
print re.findall('c.r', 'carry the barcardi to the car')

#7. 用 sub()[和 subn()]进行搜索和替换
print re.sub('X','Mr. Smith', 'attn: X\n\nDear X,\n')
#结果: ('XbcdXf', 2)
print re.subn('[ae]', 'X', 'abcdef')

#8. 用 split()分割(分割模式)
# re 模块和正则表达式的方法 split()与字符串的 split()方法相似,
#前者根据正则表达式模式分割
# 字符串,后者根据固定的字符串分割.
#结果: ['123', '', '456', '789']
print "123\n\n456\n789".split('\n')
#结果: ['123', '456', '789']
print re.split('\n+', "123\n\n456\n789")

#9. 贪心匹配
# 正则表达式本身默认是贪心匹配的。即,如果正则表达式模式中
#使用到通配字,那它在按照从左到右
# 的顺序求值时,会尽量抓取满足模式的最长字符串.
testdata = 'gov::123143294538-6-8'
patt = '.*+(\d+-\d+-\d+)'
print re.match(patt, testdata).group(1) #结果: 8-6-8
#结果: 123143294538-6-8
print re.search('\d+-\d+-\d+', testdata).group()
# 解决贪心匹配的一个办法是用"非贪婪"操作符"?",
#这个操作符可以用在*,+,或?的后面.它的作用是要求
# 正则表达式引擎匹配的字符越少越好.
#结果: 123143294538-6-8
print re.match('.+?(\d+-\d+-\d+)', testdata).group(1)
```