

Python 基础教程



Python 是一种解释型、面向对象、动态数据类型的高级程序设计语言。

Python 由 Guido van Rossum 于 1989 年底发明，第一个公开发行人版发行于 1991 年。

像 Perl 语言一样，Python 源代码同样遵循 GPL (GNU General Public License) 协议。

[现在开始学习 Python !](#)

[Python IDE \(集成开发环境\) 介绍](#)

[Python 3.X 版本的教程](#)

[Python 在线工具](#)

谁适合阅读本教程？

本教程适合想从零开始学习 Python 编程语言的开发人员。当然本教程也会对一些模块进行深入，让你更好的了解 Python 的应用。

学习本教程前你需要了解

在继续本教程之前，你应该了解一些基本的计算机编程术语。如果你学习过 PHP ， ASP 等编程语言，将有助于你更快的了解 Python 编程。

执行 Python 程序

对于大多数程序语言，第一个入门编程代码便是“Hello World ”，以下代码为使用 Python 输出“Hello World ”：

```
#!/usr/bin/python
print "Hello, World!";
```

python 3.0版本已经把 print作为一个内置函数，正确输出“Hello World ”代码如下：

```
#!/usr/bin/python

print("Hello, World!");
```

Python 简介

Python 是一个高层次的结合了解释性、编译性、互动性和面向对象的脚本语言。

Python 的设计具有很强的可读性，相比其他语言经常使用英文关键字，其他语言的一些标点符号，它具有比其他语言更有特色语法结构。

- Python **是一种解释型语言：** 这意味着开发过程中没有了编译这个环节。类似于 PHP 和 Perl 语言。
- Python **是交互式语言：** 这意味着，您可以在一个 Python 提示符，直接互动执行写你的程序。
- Python **是面向对象语言：** 这意味着 Python 支持面向对象的风格或代码封装在对象的编程技术。

- Python **是初学者的语言**：Python 对初级程序员而言，是一种伟大的语言，它支持广泛的应用程序开发，从简单的文字处理到 WWW 浏览器再到游戏。

Python 发展历史

Python 是由 Guido van Rossum 在八十年代末和九十年代初，在荷兰国家数学和计算机科学研究所设计出来的。

Python 本身也是由诸多其他语言发展而来的,这包括 ABC 、Modula-3、C、C++、Algol-68、SmallTalk、Unix shell和其他的脚本语言等等。

像 Perl语言一样，Python 源代码同样遵循 GPL(GNU General Public License)协议。

现在 Python 是由一个核心开发团队在维护,,Guido van Rossum 仍然占据着至关重要的作用，指导其进展。

Python 特点



- 1.**易于学习**：Python 有相对较少的关键字，结构简单，和一个明确定义的语法，学习起来更加简单。
- 2.**易于阅读**：Python 代码定义的更清晰。
- 3.**易于维护**：Python 的成功在于它的源代码是相当容易维护的。
- 4.**一个广泛的标准库**：Python 的最大的优势之一是丰富的库，跨平台的，在 UNIX，Windows 和 Macintosh 兼容很好。
- 5.**互动模式**：互动模式的支持，您可以从终端输入并获得结果的语言，互动的测试和调试代码片断。
- 6.**便携式**：Python 可以运行在多种硬件平台和所有平台上都具有相同的接口。
- 7.**可扩展**：可以添加低层次的模块到 Python 解释器。这些模块使程序员可以添加或定制自己的工具，更有效。
- 8.**数据库**：Python 提供所有主要的商业数据库的接口。
- 9.**GUI 编程**：Python 支持 GUI 可以创建和移植到许多系统调用。
- 10.**可扩展性**：相比 shell脚本，Python 提供了一个更好的结构，且支持大型程序。

Python 环境搭建

本章节我们将向大家介绍如何在本地搭建 Python 开发环境。

Python 可应用于多平台包括 Linux 和 Mac OS X 。

你可以通过终端窗口输入 “python” 命令来查看本地是否已经安装 Python 以及 Python 的安装版本。

- Unix (Solaris, Linux, FreeBSD, AIX, HP/UX, SunOS, IRIX, 等等。)
- Win 9x/NT/2000
- Macintosh (Intel, PPC, 68K)
- OS/2
- DOS (多个 DOS 版本)
- PalmOS
- Nokia 移动手机
- Windows CE
- Acorn/RISC OS
- BeOS
- Amiga
- VMS/OpenVMS
- QNX
- VxWorks
- Psion
- Python 同样可以移植到 Java 和 .NET 虚拟机上。

Python 下载

Python 最新源码，二进制文档，新闻资讯等可以在 Python 的官网查看到：

Python 官网：<http://www.python.org/>

你可以在一下链接中下载 Python 的文档，你可以下载 HTML 、PDF 和 PostScript 等格式的文档。

Python 文档下载地址：www.python.org/doc/

Python 安装

Python 已经被移植在许多平台上（经过改动使它能够工作在不同平台上）。

您需要下载适用于您使用平台的二进制代码，然后安装 Python。

如果您平台的二进制代码是不可用的，你需要使用 C 编译器手动编译源代码。

编译的源代码，功能上有更多的选择性，为 python 安装提供了更多的灵活性。

以下为不同平台上安装 Python 的方法：

Unix & Linux **平台安装** Python：

以下为在 Unix & Linux 平台上安装 Python 的简单步骤：

- 打开 WEB 浏览器访问 <http://www.python.org/download/>
- 选择使用于 Unix/Linux的源码压缩包。
- 下载及解压压缩包。
- 如果你需要自定义一些选项修改 Modules/Setup
- **执行** ./configur脚本
- make
- make install

执行以上操作后，Python 会安装在 /usr/local/bin 目录中，Python 库安装在 /usr/local/lib/pythonXXX 为你使用的 Python 的版本号。

Window **平台安装** Python：

以下为在 Window 平台上安装 Python 的简单步骤：

- 打开 WEB 浏览器访问 <http://www.python.org/download/>
- 在下载列表中选择 Window 平台安装包，包格式为：python-XYZ.msi 文件，XYZ 为你要安装版本号。
- 要使用安装程序 python-XYZ.msi，Windows 系统必须支持 Microsoft Installer 搭配使用。只要保存安装文件到本地计算机，然后运行它，看看你的机器支持 MSI。Windows XP 和更高版本已经有 MSI，很多老机器也可以安装 MSI。
- 下载后，双击下载包，进入 Python 安装向导，安装非常简单，你只需要使用默认的设置一直点击“下一步”直到安装完成即可。

MAC 平台安装 Python:

最近的 Macs 系统都自带 Python 环境，但是自带的 Python 版本为旧版本，你可以通过链接 <http://www.python.org/download/mac/> 查看 MAC 上 Python 的新版功能介绍。

MAC 上完整的 Python 安装教程你可以查看: <http://www.cwi.nl/~jack/macpython.html>

环境变量配置

程序和可执行文件可以在许多目录，而这些路径很可能不在操作系统提供可执行文件的搜索路径中。

path (路径) 存储在环境变量中，这是由操作系统维护的一个命名的字符串。这些变量包含可用的命令行解释器和其他程序的信息。

Unix 或 Windows 中路径变量为 PATH (UNIX 区分大小写，Windows 不区分大小写)。

在 Mac OS 中，安装程序过程中改变了 python 的安装路径。如果你需要在其他目录引用 Python，你必须在 path 中添加 Python 目录。

在 Unix/Linux 设置环境变量

- 在 csh shell: 输入

```
setenv PATH "$PATH:/usr/local/bin/python"
```

，按下“Enter”。

- 在 bash shell (Linux) 输入

```
export PATH="$PATH:/usr/local/bin/python"
```

，按下“Enter”。

- 在 sh 或者 ksh shell: 输入

```
PATH="$PATH:/usr/local/bin/python"
```

，按下“Enter”。

注意：`/usr/local/bin/python`是Python 的安装目录。

在 Windows 设置环境变量

在环境变量中添加 Python 目录：

在命令提示框中 (cmd) : 输入

```
path %path%;C:\Python
```

，按下“Enter”。

注意：`C:\Python` 是 Python 的安装目录。

Python 环境变量

下面几个重要的环境变量，它应用于 Python：

变量名	描述
PYTHONPATH	PYTHONPATH 是 Python 搜索路径，默认我们 import 的模块都会从 PYTHONPATH 里面寻找。
PYTHONSTARTUP	Python 启动后，先寻找 PYTHONSTARTUP 环境变量，然后执行此文件中变量指定的执行代码。
PYTHONCASEOK	加入 PYTHONCASEOK 的环境变量，就会使 python 导入模块的时候不区分大小写。
PYTHONHOME	另一种模块搜索路径。它通常内嵌于的 PYTHONSTARTUP 或 PYTHONPATH 目录中，使得两个模块库更容易切换。

运行 Python

有三种方式可以运行 Python：

1、交互式解释器：

你可以通过命令行窗口进入 python 并开在交互式解释器中开始编写 Python 代码。

你可以在 Unix，DOS 或任何其他提供了命令行或者 shell的系统进行 python 编码工作。

```
$python # Unix/Linux

或者

python% # Unix/Linux

或者

C:>python # Windows/DOS
```

以下为 Python 命令行参数：

选项	描述
-d	在解析时显示调试信息
-O	生成优化代码（.pyo文件）
-S	启动时不引入查找 Python 路径的位置
-v	输出 Python 版本号
-X	从 1.6 版本之后基于内建的异常（仅仅用于字符串）已过时。
-c cmd	执行 Python 脚本，并将运行结果作为 cmd 字符串。
file	在给定的 python 文件执行 python 脚本。

2、命令行脚本

在你的应用程序中通过引入解释器可以在命令行中执行 Python 脚本，如下所示：

```
$python script.py # Unix/Linux

或者

python% script.py # Unix/Linux
```

或者

```
C:>python script.py # Windows/DOS
```

注意：在执行脚本时，请检查脚本是否有可执行权限。

3、集成开发环境（IDE：Integrated Development Environment ）

您可以使用图形用户界面（GUI）环境来编写及运行 Python 代码。以下推荐各个平台上使用的 IDE：

- Unix: IDLE 是 UNIX 上最早的 Python IDE 。
- Windows: PythonWin 是一个 Python 集成开发环境,在许多方面都比 IDE 优秀
- Macintosh: Python 的 Mac 可以使用 IDLE IDE，你可以在网站上下载对应 MAC 的 IDLE 。

继续下一章之前，请确保您的环境已搭建成功。如果你不能够建立正确的环境，那么你就可以从您的系统管理员的帮助。

在以后的章节中给出的例子已在 Centos（Linux）下 Python2.4.3 版本测试通过。

Python 中文编码

前面章节中我们已经学会了如何用 Python 输出 "Hello, World!" 英文没有问题，但是如果你输出中文字符 "你好，世界" 就有可能会碰到中文编码问题。

Python 文件中如果未指定编码，在执行过程会出现报错：

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

```
print "你好，世界";
```

以上程序执行输出结果为：

```
File "test.py", line 2
```

```
SyntaxError: Non-ASCII character '\xe4' in file test.py on line 2, but  
no encoding declared; see http://www.python.org/peps/pep-0263.html for  
details
```

以上出错信息显示了我们为指定编码，解决方法为只要在文件开头加入 `# -*- coding: UTF-8 -*-` 或者 `#coding=utf-8` 就行了。

```
#coding=utf-8

#!/usr/bin/python

print "你好，世界";
```

输出结果为：

你好，世界

所以如果大家再学习过程中，代码中包含中文，就需要在头部指定编码。

Python 基础语法

Python 语言与 Perl, C 和 Java 等语言有许多相似之处。但是，也存在一些差异。

在本章中我们将来学习 Python 的基础语法，让你快速学会 Python 编程。

第一个 Python 程序

交互式编程

交互式编程不需要创建脚本文件，是通过 Python 解释器的交互模式进来编写代码。

linux上你只需要在命令行中输入 Python 命令即可启动交互式编程,提示窗口如下：

```
$ python

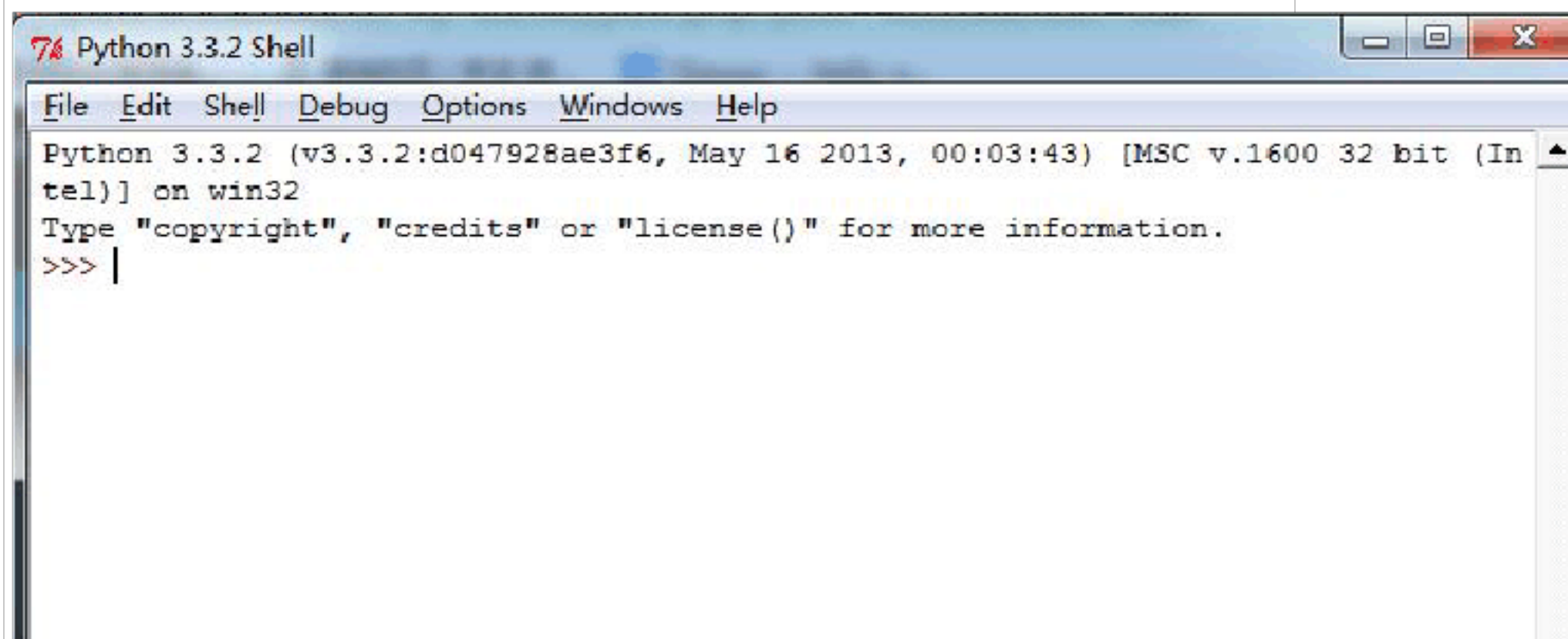
Python 2.4.3 (#1, Nov 11 2010, 13:34:43)

[GCC 4.1.2 20080704 (Red Hat 4.1.2-48)] on linux2
```

```
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>>
```

Window 上在安装 Python 时已经已经安装了默认的交互式编程客户端，提示窗口如下：



在 python 提示符中输入以下文本信息，然后按 Enter 键查看运行效果：

```
>>> print "Hello, Python!";
```

在 Python 2.4.3 版本中,以上事例输出结果如下：

```
Hello, Python!
```

如果您运行的是新版本的 Python，那么你就需要在 print 语句中使用括号如：

```
>>> print ("Hello, Python!");
```

脚本式编程

通过脚本参数调用解释器开始执行脚本，直到脚本执行完毕。当脚本执行完成后，解释器不再有效。

让我们写一个简单的 Python 脚本程序。所有 Python 文件将以 .py 为扩展名。将以下的源代码拷贝至 test.py 文件中。

```
print "Hello, Python!";
```

这里，假设你已经设置了 Python 解释器 PATH 变量。使用以下命令运行程序：

```
$ python test.py
```

输出结果：

```
Hello, Python!
```

让我们尝试另一种方式来执行 Python 脚本。修改 test.py 文件，如下所示：

```
#!/usr/bin/python

print "Hello, Python!";
```

这里，假定您的 Python 解释器在 /usr/bin 目录中，使用以下命令执行脚本：

```
$ chmod +x test.py      #脚本文件添加可执行权限
```

```
$ ./test.py
```

输出结果：

```
Hello, Python!
```

Python 标识符

在 python 里，标识符有字母、数字、下划线组成。

在 python 中，所有标识符可以包括英文、数字以及下划线（_），但不能以数字开头。

python 中的标识符是区分大小写的。

以下划线开头的标识符是有特殊意义的。以单下划线开头(`_foo`)的代表不能直接访问的类属性，需通过类提供的接口进行访问，不能用“`from xxx import 甬`”导入；

以双下划线开头的(`__foo`)代表类的私有成员；以双下划线开头和结尾的(`__foo__`)代表 python 里特殊方法专用的标识，如 `__init__()` 代表类的构造函数。

Python 保留字符

下面的列表显示了在 Python 中的保留字。这些保留字不能用作常数或变数，或任何其他标识符名称。

and	exec	not
assert	finally	or
break	for	pass
class	from	print
continue	global	raise
def	if	return
del	import	try
elif	in	while
else	is	with
except	lambda	yield

所有 Python 的关键字只包含小写字母。

行和缩进

学习 Python 与其他语言最大的区别就是，Python 的代码块不使用大括号（`{}`）来控制类，函数以及其他逻辑判断。python 最具特色的就是用缩进来写模块。

缩进的空白数量是可变的，但是所有代码块语句必须包含相同的缩进空白数量，这个必须严格执行。如下所示：

```
if True:

    print "True"

else:

    print "False"
```

以下代码将会执行错误：

```
if True:

    print "Answer"

    print "True"

else:

    print "Answer"

    print "False"
```

因此，在 Python 的代码块中必须使用相同数目的行首缩进空格数。

以下实例包含了相同数目的行首缩进代码语句块的例子：

```
#!/usr/bin/python

import sys

try:

    # open file stream

    file = open(file_name, "w")
```

```
except IOError:

    print "There was an error writing to", file_name

    sys.exit()

print "Enter '", file_finish,

print "' When finished"

while file_text != file_finish:

    file_text = raw_input("Enter text: ")

    if file_text == file_finish:

        # close the file

        file.close

        break

    file.write(file_text)

    file.write("\n")

file.close()

file_name = raw_input("Enter filename: ")

if len(file_name) == 0:

    print "Next time please enter something"

    sys.exit()

try:

    file = open(file_name, "r")

except IOError:

    print "There was an error reading file"

    sys.exit()

file_text = file.read()
```

```
file.close()

print file_text
```

多行语句

Python 语句中一般以换行作为为语句的结束符。

但是我们可以使用斜杠（ \ ）将一行的语句分为多行显示，如下所示：

```
total = item_one + \
        item_two + \
        item_three
```

语句中包含[]， 或（ ） 括号就不需要使用多行连接符。如下实例：

```
days = ['Monday', 'Tuesday', 'Wednesday',
        'Thursday', 'Friday']
```

Python 引号

Python 接收单引号（' ），双引号（" ），三引号（''' """）表示字符串，引号的开始与结束必须为相同的类型。

其中三引号可以由多行组成，编写多行文本的快捷语法，常用于文档字符串，在文件的特定地点，被当做注释。

```
word = 'word'

sentence = "This is a sentence."
```

```
paragraph = """This is a paragraph. It is  
  
made up of multiple lines and sentences."""
```

Python 注释

python 中单行注释采用 # 开头。

python 没有块注释，所以现在推荐的多行注释也是采用的 # 比如：

```
#!/usr/bin/python  
  
# First comment  
  
print "Hello, Python!"; # second comment
```

输出结果：

```
Hello, Python!
```

注释可以在语句或表达式行末：

```
name = "Madisetti" # This is again comment
```

多条评论：

```
# This is a comment.  
  
# This is a comment, too.  
  
# This is a comment, too.  
  
# I said that already.
```

Python 空行

函数之间或类的方法之间用空行分隔，表示一段新的代码的开始。类和函数入口之间也用一行空行分隔，以突出函数入口的开始。

空行与代码缩进不同，空行并不是 Python 语法的一部分。书写时不插入空行，Python 解释器运行也不会出错。但是空行的作用在于分隔两段不同功能或含义的代码，便于日后代码的维护或重构。

记住：空行也是程序代码的一部分。

等待用户输入

下面的程序在按回车键后就会等待用户输入：

```
#!/usr/bin/python

raw_input("\n\nPress the enter key to exit.")
```

以上代码中，`"\n\n"`在结果输出前会输出两个新的空行。一旦用户按下键时，程序将退出。

同一行显示多条语句

Python 可以在同一行中使用多条语句，语句之间使用分号 (;) 分割，以下是一个简单的实例：

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```

多个语句构成代码组

缩进相同的一组语句构成一个代码块，我们称之代码组。

像 `if` `while`、`def` 和 `class` 这样的复合语句，首行以关键字开始，以冒号 (:) 结束，该行之后的一行或多行代码构成代码组。

我们将首行及后面的代码组称为一个子句 (clause)。

如下实例：

```
if expression :  
  
    suite  
  
elif expression :  
  
    suite  
  
else :  
  
    suite
```

命令行参数

很多程序可以执行一些操作来查看一些基本信，Python 可以使用-h 参数查看各参数帮助信息：

```
$ python -h  
  
usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...  
  
Options and arguments (and corresponding environment variables):  
  
-c cmd : program passed in as string (terminates option list)  
  
-d      : debug output from parser (also PYTHONDEBUG=x)  
  
-E      : ignore environment variables (such as PYTHONPATH)  
  
-h      : print this help message and exit  
  
[ etc. ]
```

Python 变量类型

变量存储在内存中的值。这就意味着在创建变量时会在内存中开辟一个空间。

基于变量的数据类型，解释器会分配指定内存，并决定什么数据可以被存储在内存中。

因此，变量可以指定不同的数据类型，这些变量可以存储整数，小数或字符。

变量赋值

Python 中的变量不需要声明，变量的赋值操作既是变量声明和定义的过程。

每个变量在内存中创建，都包括变量的标识，名称和数据这些信息。

每个变量在使用前都必须赋值，变量赋值以后该变量才会被创建。

等号(=)用来给变量赋值。

等号(=)运算符左边是一个变量名,等号(=)运算符右边是存储在变量中的值。例如：

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

counter = 100 # 赋值整型变量
miles = 1000.0 # 浮点型
name = "John" # 字符串

print counter
print miles
print name
```

以上实例中，100，1000.0 和"John"分别赋值给 counter，miles，name 变量。

执行以上程序会输出如下结果：

```
100
1000.0
John
```

多个变量赋值

Python 允许你同时为多个变量赋值。例如：

```
a = b = c = 1
```

以上实例，创建一个整型对象，值为 1，三个变量被分配到相同的内存空间上。

您也可以为多个对象指定多个变量。例如：

```
a, b, c = 1, 2, "john"
```

以上实例，两个整型对象 1 和 2 的分配给变量 a 和 b，字符串对象"john"分配给变量 c。

标准数据类型

在内存中存储的数据可以有多种类型。

例如，person.s 年龄作为一个数值存储和他或她的地址是字母数字字符存储。

Python 有一些标准类型用于定义操作上，他们和为他们每个人的存储方法可能。

Python 有五个标准的数据类型：

- Numbers （数字）
- String(字符串)
- List(列表)
- Tuple（元组）
- Dictionary(字典)

Python 数字

数字数据类型用于存储数值。

他们是不可改变的数据类型，这意味着改变数字数据类型会分配一个新的对象。

当你指定一个值时，Number 对象就会被创建：

```
var1 = 1
var2 = 10
```

您也可以使用 del 语句删除一些对象引用。

del 语句的语法是：

```
del var1[,var2[,var3[...[,varN]]]]
```

您可以通过使用 del 语句删除单个或多个对象。例如：

```
del var
del var_a, var_b
```

Python 支持四种不同的数值类型：

- int(有符号整型)
- long (长整型[也可以代表八进制和十六进制])
- float(浮点型)
- complex (复数)

实例

一些数值类型的实例：

int	long	float	complex
10	51924361L	0.0	3.14j
100	-0x19323L	15.20	45. j
-786	0122L	-21.9	9.322e-36j
080	0xDEFABCECBDAECBFBAE1	32.3+e18	.876j
-0490	535633629843L	-90.	-.6545+0J
-0x260	-052318172735L	-32.54e100	3e+26J
0x69	-4721885298529L	70.2-E12	4.53e-7j

- 长整型也可以使用小写“l”，但是还是建议您使用大写“L”，避免与数字“1”混淆。Python 使用“L”来显示长整型。

- Python 还支持复数，复数由实数部分和虚数部分构成，可以用 `a + b` 或者 `complex(a, b)` 表示， 复数的实部 `a` 和虚部 `b` 都是浮点型

Python 字符串

字符串或串 (String)是由数字、字母、下划线组成的一串字符。

一般记为：

```
s="a1a2...an" (n>=0)
```

它是编程语言中表示文本的数据类型。

python 的字串列表有 2 种取值顺序：

- 从左到右索引默认 0 开始的，最大范围是字符串长度少 1
- 从右到左索引默认-1 开始的，最大范围是字符串开头

如果你的实要取得一段子串的话，可以用到变量[头下标:尾下标]，就可以截取相应的字符串，其中下标是从 0 开始算起，可以是正数或负数，下标可以为空表示取到头或尾。

比如：

```
s = 'ilovepython'
```

`s[1:5]`的结果是 love。

当使用以冒号分隔的字符串，python 返回一个新的对象，结果包含了以这对偏移标识的连续的内容，左边的开始是包含了下边界。

上面的结果包含了 `s[1]`的值 l，而取到的最大范围不包括上边界，就是 `s[5]`的值 p。

加号 (+) 是字符串连接运算符，星号 (*) 是重复操作。如下实例：

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

```
str = 'Hello World!'
```

```
print str # 输出完整字符串
```

```
print str[0] # 输出字符串中的第一个字符
```

```
print str[2:5] # 输出字符串中第三个至第五个之间的字符串
```

```
print str[2:] # 输出从第三个字符开始的字符串
print str * 2 # 输出字符串两次
print str + "TEST" # 输出连接的字符串
```

以上实例输出结果：

```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

Python 列表

List(列表) 是 Python 中使用最频繁的数据类型。

列表可以完成大多数集合类的数据结构实现。它支持字符，数字，字符串甚至可以包含列表（所谓嵌套）。

列表用[] 标识。是 python 最通用的复合数据类型。看这段代码就明白。

列表中的值得分割也可以用到变量[头下标:尾下标]，就可以截取相应的列表，从左到右索引默认0 开始的，从右到左索引默认-1 开始，下标可以为空表示取到头或尾。

加号(+) 是列表连接运算符，星号(*) 是重复操作。如下实例：

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

list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']

print list # 输出完整列表
print list[0] # 输出列表的第一个元素
print list[1:3] # 输出第二个至第三个的元素
print list[2:] # 输出从第三个开始至列表末尾的所有元素
print tinylist * 2 # 输出列表两次
print list + tinylist # 打印组合的列表
```

以上实例输出结果：

```
['abcd', 786, 2.23, 'john', 70.2]
abcd
[786, 2.23]
[2.23, 'john', 70.2]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.2, 123, 'john']
```

Python 元组

元组是另一个数据类型，类似于 List(列表)。

元组用"()"标识。内部元素用逗号隔开。但是元素不能二次赋值，相当于只读列表。

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

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
tinytuple = (123, 'john')

print tuple # 输出完整元组
print tuple[0] # 输出元组的第一个元素
print tuple[1:3] # 输出第二个至第三个的元素
print tuple[2:] # 输出从第三个开始至列表末尾的所有元素
print tinytuple * 2 # 输出元组两次
print tuple + tinytuple # 打印组合的元组
```

以上实例输出结果：

```
('abcd', 786, 2.23, 'john', 70.2)
abcd
(786, 2.23)
(2.23, 'john', 70.2)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.2, 123, 'john')
```

以下是元组无效的，因为元组是不允许更新的。而列表是允许更新的：

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

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
```

```
tuple[2] = 1000 # 元组中是非法应用
list[2] = 1000 # 列表中是合法应用
```

Python 元字典

字典(dictionary)是除列表以外 python 之中最灵活的内置数据结构类型。列表是有序的对象结合，字典是无序的对象集合。

两者之间的区别在于：字典当中的元素是通过键来存取的，而不是通过偏移存取。

字典用“{”标识。字典由索引(key)和它对应的值 value 组成。

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

dict = {}
dict['one'] = "This is one"
dict[2] = "This is two"

tinydict = {'name': 'john', 'code':6734, 'dept': 'sales'}


print dict['one'] # 输出键为'one' 的值
print dict[2] # 输出键为 2 的值
print tinydict # 输出完整的字典
print tinydict.keys() # 输出所有键
print tinydict.values() # 输出所有值
```

输出结果为：

```
This is one This is two {'dept': 'sales', 'code': 6734, 'name':
'john'} ['dept', 'code', 'name'] ['sales', 6734, 'john']
```

Python 数据类型转换

有时候，我们需要对数据内置的类型进行转换，数据类型的转换，你只需要将数据类型作为函数名即可。

以下几个内置的函数可以执行数据类型之间的转换。这些函数返回一个新的对象，表示转换的值。

函数	描述
<code>int(x [,base])</code>	将 x 转换为一个整数
<code>long(x [,base])</code>	将 x 转换为一个长整数
<code>float(x)</code>	将 x 转换到一个浮点数
<code>complex(real [,imag])</code>	创建一个复数
<code>str(x)</code>	将对象 x 转换为字符串
<code>repr(x)</code>	将对象 x 转换为表达式字符串
<code>eval(str)</code>	用来计算在字符串中的有效 Python 表达式,并返回一个对象
<code>tuple(s)</code>	将序列 s 转换为一个元组
<code>list(s)</code>	将序列 s 转换为一个列表
<code>set(s)</code>	转换为可变集合
<code>dict(d)</code>	创建一个字典。d 必须是一个序列 (key,value)元组。
<code>frozenset(s)</code>	转换为不可变集合
<code>chr(x)</code>	将一个整数转换为一个字符
<code>unichr(x)</code>	将一个整数转换为 Unicode 字符
<code>ord(x)</code>	将一个字符转换为它的整数值
<code>hex(x)</code>	将一个整数转换为一个十六进制字符串
<code>oct(x)</code>	将一个整数转换为一个八进制字符串

什么是运算符？

本章节主要说明 Python 的运算符。举个简单的例子 `4 +5 = 9` 例子中，4 和 5 被称为操作数，“+”号为运算符。

Python 语言支持以下类型的运算符：

- 算术运算符
- 比较（关系）运算符
- 赋值运算符
- 逻辑运算符

- [位运算符](#)
- [成员运算符](#)
- [身份运算符](#)
- [运算符优先级](#)

接下来让我们一个个来学习 Python 的运算符。

Python 算术运算符

以下假设变量 a 为 10，变量 b 为 20：

运算符	描述	实例
+	加 - 两个对象相加	a + b 输出结果 30
-	减 - 得到负数或是一个数减去另一个数	a - b输出结果 -10
*	乘 - 两个数相乘或是返回一个被重复若干次的字符串	a * b输出结果 200
/	除 - x除以 y	b / a输出结果 2
%	取模 - 返回除法的余数	b % a 输出结果 0
**	幂 - 返回 x 的 y 次幂	a**b 为 10 的 20 次方， 输出结果 100000000000000000000
//	取整除 - 返回商的整数部分	9//2 输出结果 4 ， 9.0//2.0输出结果 4.0

以下实例演示了 Python 所有算术运算符的操作：

```
#!/usr/bin/python

a = 21

b = 10

c = 0

c = a + b
```

```
print "Line 1 - Value of c is ", c
```

```
c = a - b
```

```
print "Line 2 - Value of c is ", c
```

```
c = a * b
```

```
print "Line 3 - Value of c is ", c
```

```
c = a / b
```

```
print "Line 4 - Value of c is ", c
```

```
c = a % b
```

```
print "Line 5 - Value of c is ", c
```

```
a = 2
```

```
b = 3
```

```
c = a**b
```

```
print "Line 6 - Value of c is ", c
```

```
a = 10
```

```
b = 5
```

```
c = a//b
```

```
print "Line 7 - Value of c is ", c
```

以上实例输出结果：

```
Line 1 - Value of c is 31

Line 2 - Value of c is 11

Line 3 - Value of c is 210

Line 4 - Value of c is 2

Line 5 - Value of c is 1

Line 6 - Value of c is 8

Line 7 - Value of c is 2
```

Python 比较运算符

以下假设变量 a 为 10，变量 b 为 20：

运算符	描述	实例
==	等于 - 比较对象是否相等	(a == b) 返回 False。
!=	不等于 - 比较两个对象是否不相等	(a != b) 返回 true。
<>	不等于 - 比较两个对象是否不相等	(a <> b) 返回 true 这个运算符类似 != 。
>	大于 - 返回 x 是否大于 y	(a > b) 返回 False。
<	小于 - 返回 x 是否小于 y。所有比较运算符返回 1 表示真，返回 0 表示假。 这分别与特殊的变量 True 和 False 等价。注意，这些变量名的大写。	(a < b) 返回 true
>=	大于等于 - 返回 x 是否大于等于 y。	(a >= b) 返回 False。
<=	小于等于 - 返回 x 是否小于等于 y。	(a <= b) 返回 true

以下实例演示了 Python 所有比较运算符的操作：

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

```
a = 21
```

```
b = 10
```

```
c = 0
```

```
if ( a == b ):
```

```
    print "Line 1 - a is equal to b"
```

```
else:
```

```
    print "Line 1 - a is not equal to b"
```

```
if ( a != b ):
```

```
    print "Line 2 - a is not equal to b"
```

```
else:
```

```
    print "Line 2 - a is equal to b"
```

```
if ( a <> b ):
```

```
    print "Line 3 - a is not equal to b"
```

```
else:
```

```
    print "Line 3 - a is equal to b"
```

```
if ( a < b ):
```

```
    print "Line 4 - a is less than b"
```

```
else:
```

```
    print "Line 4 - a is not less than b"
```

```
if ( a > b ):

    print "Line 5 - a is greater than b"

else:

    print "Line 5 - a is not greater than b"


a = 5;

b = 20;

if ( a <= b ):

    print "Line 6 - a is either less than or equal to  b"

else:

    print "Line 6 - a is neither less than nor equal to  b"


if ( b >= a ):

    print "Line 7 - b is either greater than  or equal to b"

else:

    print "Line 7 - b is neither greater than  nor equal to b"
```

以上实例输出结果:

```
Line 1 - a is not equal to b

Line 2 - a is not equal to b

Line 3 - a is not equal to b

Line 4 - a is not less than b

Line 5 - a is greater than b
```

Line 6 - a is either less than or equal to b

Line 7 - b is either greater than or equal to b

Python 赋值运算符

以下假设变量 a 为 10，变量 b 为 20：

运算符	描述	实例
=	简单的赋值运算符	c = a + b将 a + b 的运算结果赋值为 c
+=	加法赋值运算符	c += a 等效于 c = c + a
-=	减法赋值运算符	c -= a等效于 c = c - a
*=	乘法赋值运算符	c *= a等效于 c = c * a
/=	除法赋值运算符	c /= a等效于 c = c / a
%=	取模赋值运算符	c %= a 等效于 c = c % a
**=	幂赋值运算符	c **= a等效于 c = c ** a
//=	取整除赋值运算符	c //= a等效于 c = c // a

以下实例演示了 Python 所有赋值运算符的操作：

```
#!/usr/bin/python

a = 21

b = 10

c = 0

c = a + b

print "Line 1 - Value of c is ", c
```

```
c += a

print "Line 2 - Value of c is ", c


c *= a

print "Line 3 - Value of c is ", c


c /= a

print "Line 4 - Value of c is ", c


c = 2

c %= a

print "Line 5 - Value of c is ", c


c **= a

print "Line 6 - Value of c is ", c


c //= a

print "Line 7 - Value of c is ", c
```

以上实例输出结果：

```
Line 1 - Value of c is 31

Line 2 - Value of c is 52

Line 3 - Value of c is 1092
```

```
Line 4 - Value of c is 52

Line 5 - Value of c is 2

Line 6 - Value of c is 2097152

Line 7 - Value of c is 99864
```

Python 位运算符

按位运算符是把数字看作二进制来进行计算的。Python 中的按位运算法则如下：

运算符	描述	实例
&	按位与运算符	(a & b)输出结果 12 ，二进制解释： 0000 1100
	按位或运算符	(a b)输出结果 61 ，二进制解释： 0011 1101
^	按位异或运算符	(a ^ b)输出结果 49 ，二进制解释： 0011 0001
~	按位取反运算符	(~a)输出结果 -61 ，二进制解释： 1100 0011 ，在一个有符号二进制数的补码形式。
<<	左移动运算符	a << 2 输出结果 240 ，二进制解释： 1111 0000
>>	右移动运算符	a >> 2 输出结果 15 ，二进制解释： 0000 1111

以下实例演示了 Python 所有位运算符的操作：

```
#!/usr/bin/python

a = 60          # 60 = 0011 1100

b = 13          # 13 = 0000 1101

c = 0

c = a & b;      # 12 = 0000 1100
```

```
print "Line 1 - Value of c is ", c

c = a | b;          # 61 = 0011 1101

print "Line 2 - Value of c is ", c

c = a ^ b;          # 49 = 0011 0001

print "Line 3 - Value of c is ", c

c = ~a;             # -61 = 1100 0011

print "Line 4 - Value of c is ", c

c = a << 2;          # 240 = 1111 0000

print "Line 5 - Value of c is ", c

c = a >> 2;          # 15 = 0000 1111

print "Line 6 - Value of c is ", c
```

以上实例输出结果:

```
Line 1 - Value of c is 12

Line 2 - Value of c is 61

Line 3 - Value of c is 49

Line 4 - Value of c is -61

Line 5 - Value of c is 240

Line 6 - Value of c is 15
```


Python 逻辑运算符

Python 语言支持逻辑运算符，以下假设变量 a 为 10，变量 b 为 20：

运算符	描述	实例
and	布尔“与” -如果 x 为 False，x and y返回 False，否则它返回 y 的计算值。	(a and b)返回 true
or	布尔“或” -如果 x 是 True，它返回 True，否则它返回 y 的计算值。	(a or b)返回 true
not	布尔“非” -如果 x 为 True，返回 False。如果 x 为 False，它返回 True。	not(a and b)返回 false

以下实例演示了 Python 所有逻辑运算符的操作：

```
#!/usr/bin/python

a = 10

b = 20

c = 0

if ( a and b ):

    print "Line 1 - a and b are true"

else:

    print "Line 1 - Either a is not true or b is not true"

if ( a or b ):

    print "Line 2 - Either a is true or b is true or both are true"

else:

    print "Line 2 - Neither a is true nor b is true"
```

```
a = 0

if ( a and b ):

    print "Line 3 - a and b are true"

else:

    print "Line 3 - Either a is not true or b is not true"


if ( a or b ):

    print "Line 4 - Either a is true or b is true or both are true"

else:

    print "Line 4 - Neither a is true nor b is true"


if not( a and b ):

    print "Line 5 - Either a is not true or b is not true or both are not true"

else:

    print "Line 5 - a and b are true"
```

以上实例输出结果：

```
Line 1 - a and b are true

Line 2 - Either a is true or b is true or both are true

Line 3 - Either a is not true or b is not true

Line 4 - Either a is true or b is true or both are true
```

Line 5 - Either a is not true or b is not true or both are not true

Python 成员运算符

除了以上的一些运算符之外，Python 还支持成员运算符，测试实例中包含了一系列的成员，包括字符串，列表或元组。

运算符	描述	实例
in	如果在指定的序列中找到值返回 True，否则返回 False。	x 在 y 序列中 ，如果 x 在 y 序列中返回 True。
not in	如果在指定的序列中没有找到值返回 True，否则返回 False。	x 不在 y 序列中 ，如果 x 不在 y 序列中返回 True。

以下实例演示了 Python 所有成员运算符的操作：

```
#!/usr/bin/python

a = 10

b = 20

list = [1, 2, 3, 4, 5 ];

if ( a in list ):

    print "Line 1 - a is available in the given list"

else:

    print "Line 1 - a is not available in the given list"


if ( b not in list ):

    print "Line 2 - b is not available in the given list"
```

```
else:

    print "Line 2 - b is available in the given list"

a = 2

if ( a in list ):

    print "Line 3 - a is available in the given list"

else:

    print "Line 3 - a is not available in the given list"
```

以上实例输出结果：

```
Line 1 - a is not available in the given list

Line 2 - b is not available in the given list

Line 3 - a is available in the given list
```

Python 身份运算符

身份运算符用于比较两个对象的存储单元

运算符	描述	实例
is	is是判断两个标识符是不是引用自一个对象	x is y,如果 id(x)等于 id(y) is返回结果 1
is not	is not是判断两个标识符是不是引用自不同对象	x is not y,如果 id(x)不等于 id(y)is not返回结果 1

以下实例演示了 Python 所有身份运算符的操作：

```
#!/usr/bin/python

a = 20
```

```
b = 20

if ( a is b ):

    print "Line 1 - a and b have same identity"

else:

    print "Line 1 - a and b do not have same identity"


if ( id(a) == id(b) ):

    print "Line 2 - a and b have same identity"

else:

    print "Line 2 - a and b do not have same identity"


b = 30

if ( a is b ):

    print "Line 3 - a and b have same identity"

else:

    print "Line 3 - a and b do not have same identity"


if ( a is not b ):

    print "Line 4 - a and b do not have same identity"

else:

    print "Line 4 - a and b have same identity"
```

以上实例输出结果：

```
Line 1 - a and b have same identity

Line 2 - a and b have same identity

Line 3 - a and b do not have same identity

Line 4 - a and b do not have same identity
```

Python 运算符优先级

以下表格列出了从最高到最低优先级的所有运算符：

运算符	描述
**	指数（最高优先级）
~ + -	按位翻转，一元加号和减号（最后两个的方法名为 +@ 和 -@）
* / % //	乘，除，取模和取整除
+ -	加法减法
>> <<	右移，左移运算符
&	位 'AND'
^	位运算符
<= < > >=	比较运算符
<> == !=	等于运算符
= %= /= //= -= += *= **=	赋值运算符
is is not	身份运算符
in not in	成员运算符
not or and	逻辑运算符

以下实例演示了 Python 所有运算符优先级的操作：

```
#!/usr/bin/python

a = 20

b = 10

c = 15

d = 5

e = 0


e = (a + b) * c / d      #( 30 * 15 ) / 5

print "Value of (a + b) * c / d is ", e


e = ((a + b) * c) / d    # (30 * 15 ) / 5

print "Value of ((a + b) * c) / d is ", e


e = (a + b) * (c / d);    # (30) * (15/5)

print "Value of (a + b) * (c / d) is ", e


e = a + (b * c) / d;      # 20 + (150/5)

print "Value of a + (b * c) / d is ", e
```

以上实例输出结果：

```
Value of (a + b) * c / d is 90

Value of ((a + b) * c) / d is 90

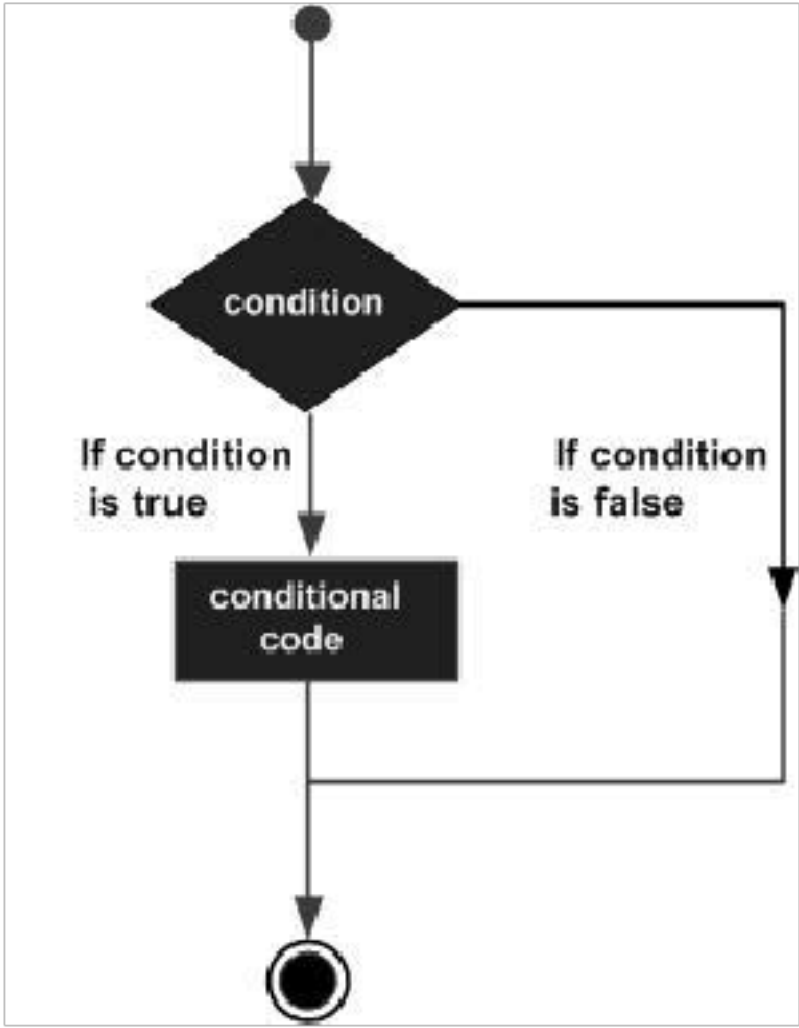
Value of (a + b) * (c / d) is 90
```

Value of $a + (b * c) / d$ is 50

Python 条件语句

Python 条件语句是通过一条或多条语句的执行结果（True 或者 False）来决定执行的代码块。

可以通过下图来简单了解条件语句的执行过程：



Python 程序语言指定任何非 0 和非空（null）值为 true，0 或者 null 为 false

Python 编程中 if 语句用于控制程序的执行，基本形式为：

```
if 判断条件:

    执行语句.....

else:

    执行语句.....
```

其中“判断条件”成立时（非零），则执行后面的语句，而执行内容可以多行，以缩进来区分表示同一范围。

else 为可选语句，当需要在条件不成立时执行内容则可以执行相关语句，具体例子如下：


```
# coding=utf8

# 例 1: if 基本用法

flag = False

name = 'luren'

if name == 'python':           判断变量否为'python'

    flag = True                条件成立时设置标志为真

    print 'welcome boss'      并输出欢迎信息

else:

    print name                  条件不成立时输出变量名称
```

输出结果为:

```
>>> luren                      # 输出结果
```

if语句的判断条件可以用>（大于）、<（小于）、==（等于）、>=（大于等于）、<=（小于等于）来表示其关系。

当判断条件为多个值是，可以使用以下形式：

```
if 判断条件 1:

    执行语句 1.....

elif 判断条件 2:

    执行语句 2.....

elif 判断条件 3:

    执行语句 3.....

else:
```

执行语句 4.....

实例如下：

```
# coding=utf8

# 例 2: elif 用法

num = 5

if num == 3:           判断 num 的值

    print 'boss'

elif num == 2:

    print 'user'

elif num == 1:

    print 'worker'

elif num < 0:          值小于零时输出

    print 'error'

else:

    print 'roadman'    条件均不成立时输出
```

输出结果为：

```
>>> roadman           # 输出结果
```

由于 python 并不支持 switch 语句，所以多个条件判断，只能用 elif来实现，如果判断需要多个条件需同时判断时，可以使用 or（或），表示两个条件有一个成立时判断条件成功；使用 and（与）时，表示只有两个条件同时成立的情况下，判断条件才成功。

```
# coding=utf8

# 例 3: if 语句多个条件
```

```
num = 9

if num >= 0 and num <= 10:    #判断值是否在 0~10 之间

    print 'hello'

>>> hello                    # 输出结果
```

```
num = 10

if num < 0 or num > 10:    #判断值是否在小于 0 或大于 10

    print 'hello'

else:

    print 'undefine'

>>> undefine                # 输出结果
```

```
num = 8

# 判断值是否在 0~5 或者 10~15 之间

if (num >= 0 and num <= 5) or (num >= 10 and num <= 15):

    print 'hello'

else:

    print 'undefine'

>>> undefine                # 输出结果
```

当 if 有多个条件时可使用括号来区分判断的先后顺序，括号中的判断优先执行，此外 and 和 or 的优先级低于 >（大于）、<（小于）等判断符号，即大于和小于在没有括号的情况下会比与或要优先判断。

简单的语句组

你也可以在同一行的位置上使用 `i` 条件判断语句，如下实例：

```
#!/usr/bin/python

var = 100

if ( var == 100 ) : print "Value of expression is 100"

print "Good bye!"
```

以上代码执行输出结果如下：

```
Value of expression is 100

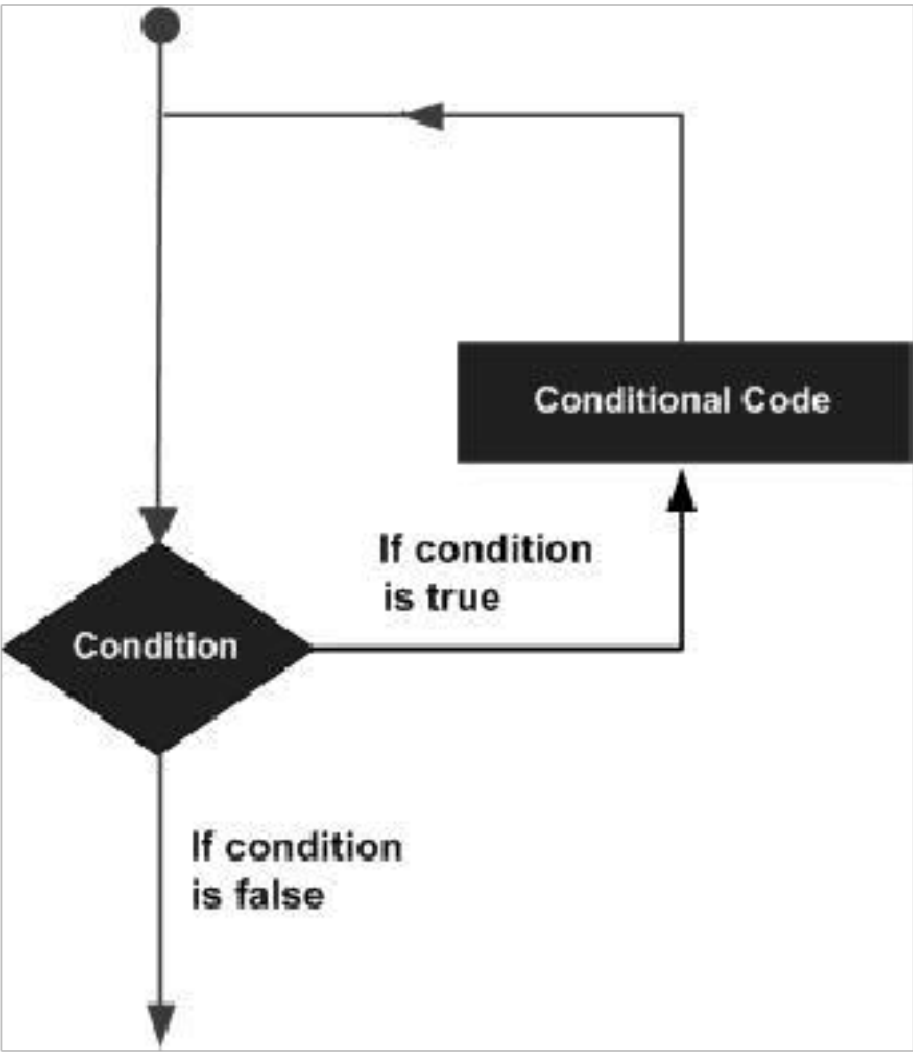
Good bye!
```

Python 循环语句

本章节将向大家介绍 Python 的循环语句，程序在一般情况下是按顺序执行的。

编程语言提供了各种控制结构，允许更复杂的执行路径。

循环语句允许我们执行一个语句或语句组多次，下面是在大多数编程语言中的循环语句的一般形式：



Python 提供了 for循环和 while循环（在 Python 中没有 do..while循环）：

循环类型	描述
<u>while 循环</u>	在给定的判断条件为 true 时执行循环体，否则退出循环体。
<u>for 循环</u>	重复执行语句
<u>嵌套循环</u>	你可以在 while循环体中嵌套 for循环

循环控制语句

循环控制语句可以更改语句执行的顺序。Python 支持以下循环控制语句：

控制语句	描述
<u>break 语句</u>	在语句块执行过程中终止循环，并且跳出整个循环
<u>continue 语句</u>	在语句块执行过程中终止当前循环，跳出该次循环，执行下一次循环。
<u>pass 语句</u>	pass 是空语句，是为了保持程序结构的完整性。

Python While 循环语句

Python 编程中 while 语句用于循环执行程序，即在某条件下，循环执行某段程序，以处理需要重复处理的相同任务。其基本形式为：

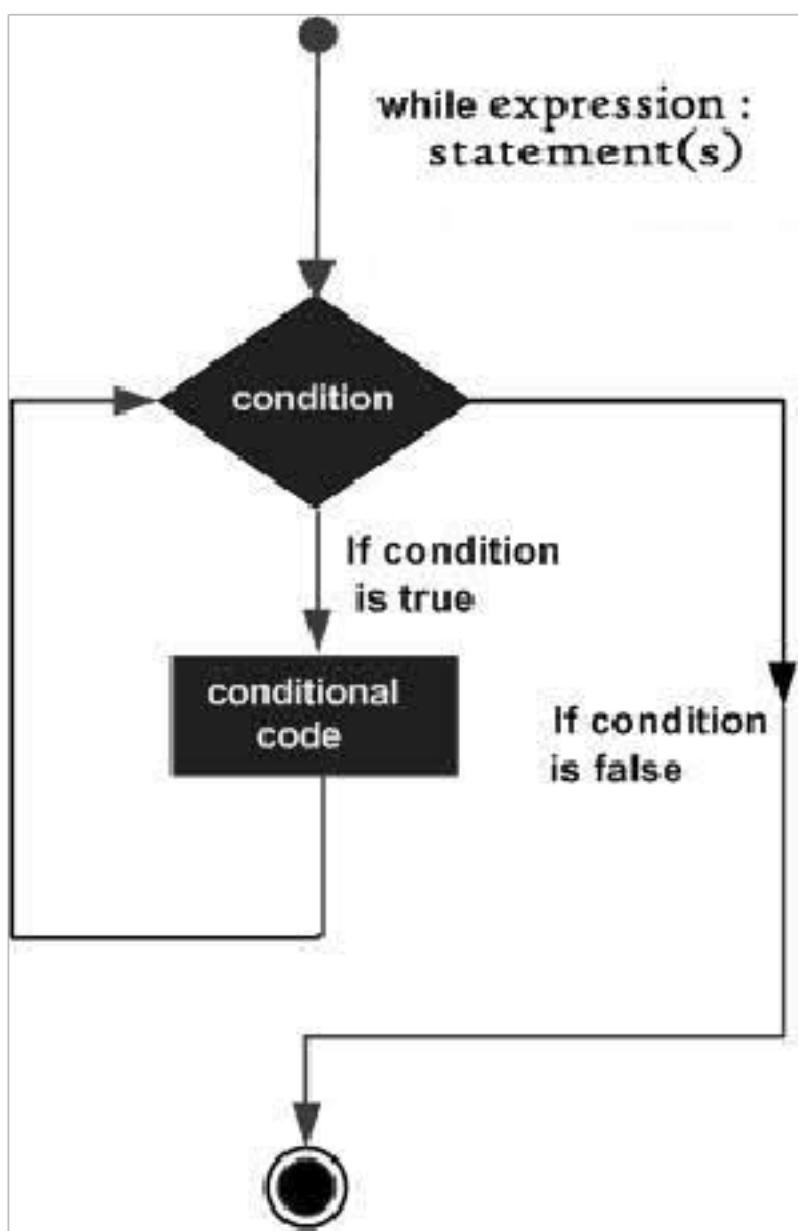
```
while 判断条件:

    执行语句.....
```

执行语句可以是单个语句或语句块。判断条件可以是任何表达式，任何非零、或非空（null）的值均为 true。

当判断条件假 false 时，循环结束。

执行流程图如下：



实例：

```
#!/usr/bin/python

count = 0

while (count < 9):
```

```
print 'The count is:', count

count = count + 1

print "Good bye!"
```

以上代码执行输出结果：

```
The count is: 0

The count is: 1

The count is: 2

The count is: 3

The count is: 4

The count is: 5

The count is: 6

The count is: 7

The count is: 8

Good bye!
```

while 语句时还有另外两个重要的命令 `continue`, `break` 来跳过循环, `continue` 用于跳过该次循环, `break` 则是用于退出循环, 此外“判断条件”还可以是个常值, 表示循环必定成立, 具体用法如下:

```
# continue 和 break 用法

i = 1

while i < 10:

    i += 1
```

```
    if i%2 > 0:        非双数时跳过输出

        continue

    print i            输出双数 2、4、6、8、10

i = 1

while 1:              循环条件为 1 必定成立

    print i            输出 1~10

    i += 1

    if i > 10:         当 i 大于 10 时跳出循环

        break
```

无限循环

如果条件判断语句永远为 `true` 循环将会无限的执行下去，如下实例：

```
#coding=utf-8

#!/usr/bin/python

var = 1

while var == 1 :    #该条件永远为 true，循环将无限执行下去

    num = raw_input("Enter a number  :")

    print "You entered: ", num
```

```
print "Good bye!"
```

以上实例输出结果：

```
Enter a number :20

You entered:  20

Enter a number :29

You entered:  29

Enter a number :3

You entered:  3

Enter a number between :Traceback (most recent call last):

  File "test.py", line 5, in <module>

    num = raw_input("Enter a number :")

KeyboardInterrupt
```

注意： 以上的无限循环你可以使用 CTRL+C 来中断循环。

循环使用 else 语句

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

```
#!/usr/bin/python

count = 0

while count < 5:
```

```
    print count, " is less than 5"

    count = count + 1

else:

    print count, " is not less than 5"
```

以上实例输出结果为：

```
0 is less than 5

1 is less than 5

2 is less than 5

3 is less than 5

4 is less than 5

5 is not less than 5
```

简单语句组

类似 `if` 语句的语法，如果你的 `while` 循环体中只有一条语句，你可以将该语句与 `while` 写在同一行中，如下所示：

```
#!/usr/bin/python

flag = 1

while (flag): print 'Given flag is really true!'

print "Good bye!"
```

注意： 以上的无限循环你可以使用 CTRL+C 来中断循环。

Python for 循环语句

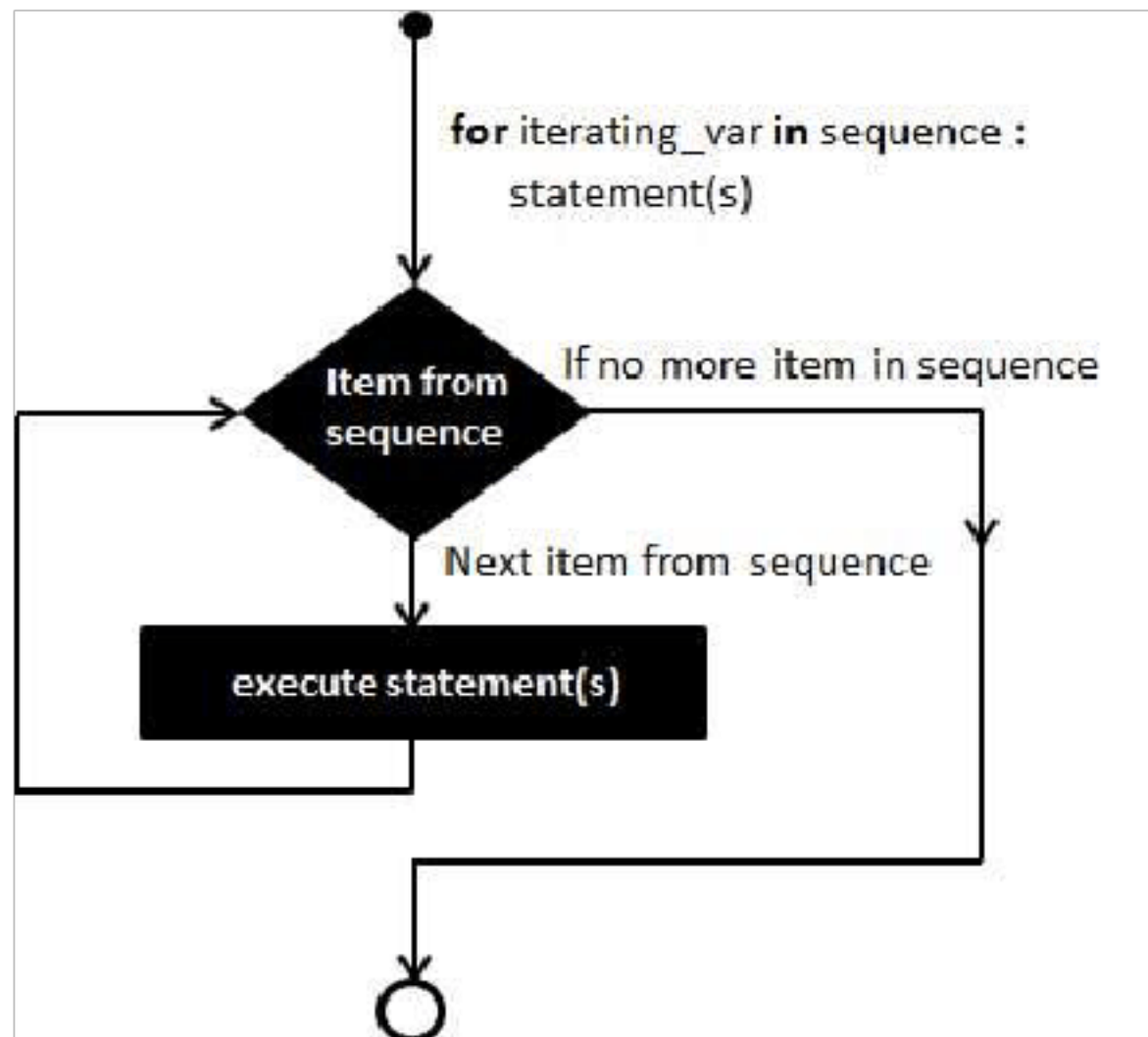
Python for循环可以遍历任何序列的项目，如一个列表或者一个字符串。

语法：

for循环的语法格式如下：

```
for iterating_var in sequence:
    statements(s)
```

流程图：



实例：

```
#!/usr/bin/python

for letter in 'Python':    # First Example

    print 'Current Letter :', letter
```

```
fruits = ['banana', 'apple', 'mango']

for fruit in fruits:      # Second Example

    print 'Current fruit :', fruit


print "Good bye!"
```

以上实例输出结果：

```
Current Letter : P

Current Letter : y

Current Letter : t

Current Letter : h

Current Letter : o

Current Letter : n

Current fruit : banana

Current fruit : apple

Current fruit : mango

Good bye!
```

通过序列索引迭代

另外一种执行循环的遍历方式是通过索引，如下实例：

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

```
fruits = ['banana', 'apple', 'mango']

for index in range(len(fruits)):

    print 'Current fruit :', fruits[index]


print "Good bye!"
```

以上实例输出结果：

```
Current fruit : banana

Current fruit : apple

Current fruit : mango

Good bye!
```

以上实例我们使用了内置函数 `len()` 和 `range()` 函数 `len()` 返回列表的长度，即元素的个数。
`range` 返回一个序列的数。

循环使用 `else` 语句

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

如下实例：

```
#!/usr/bin/python

for num in range(10,20):    #to iterate between 10 to 20

    for i in range(2,num):  #to iterate on the factors of the number

        if num%i == 0:      #to determine the first factor
```

```
j=num/i          #to calculate the second factor

print '%d equals %d * %d' % (num,i,j)

break #to move to the next number, the #first FOR

else:            # else part of the loop

    print num, 'is a prime number'
```

以上实例输出结果:

```
10 equals 2 * 5

11 is a prime number

12 equals 2 * 6

13 is a prime number

14 equals 2 * 7

15 equals 3 * 5

16 equals 2 * 8

17 is a prime number

18 equals 2 * 9

19 is a prime number
```

Python 循环嵌套

Python 语言允许在一个循环体里面嵌入另一个循环。

Python for 循环嵌套语法:

```
for iterating_var in sequence:

    for iterating_var in sequence:

        statements(s)
```

```
statements(s)
```

Python while 循环嵌套语法：

```
while expression:

    while expression:

        statement(s)

    statement(s)
```

你可以在循环体内嵌入其他的循环体，如在 while 循环中可以嵌入 for 循环，反之，你可以在 for 循环中嵌入 while 循环。

实例：

以下实例使用了嵌套循环输出 2~100 之间的素数：

```
#coding=utf-8

#!/usr/bin/python

i = 2

while(i < 100):

    j = 2

    while(j <= (i/j)):

        if not(i%j): break

        j = j + 1

    if (j > i/j) : print i, "是素数"

    i = i + 1

print "Good bye!"
```

以上实例输出结果：

2 是素数

3 是素数

5 是素数

7 是素数

11 是素数

13 是素数

17 是素数

19 是素数

23 是素数

29 是素数

31 是素数

37 是素数

41 是素数

43 是素数

47 是素数

53 是素数

59 是素数

61 是素数

67 是素数

71 是素数

73 是素数

79 是素数


```
83 是素数

89 是素数

97 是素数

Good bye!
```

Python break 语句

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

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

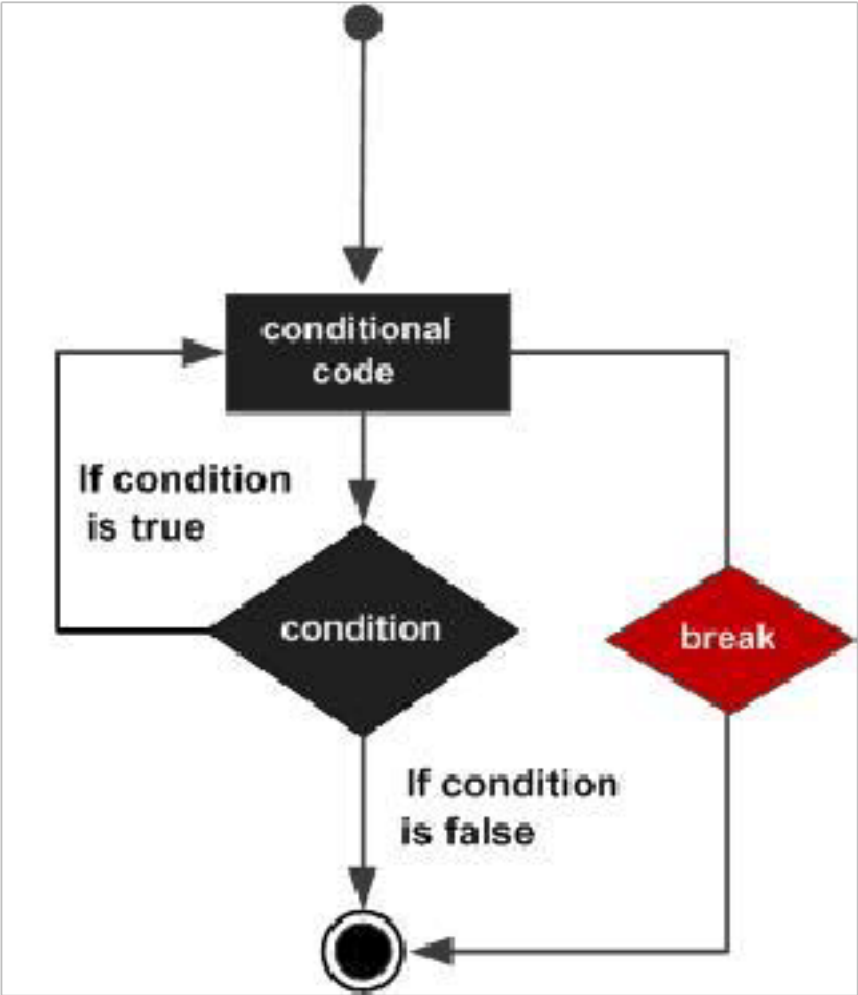
break 语句用在 while 和 for 循环中。

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

Python 语言 break 语句语法：

```
break
```

流程图：



实例：

```
#!/usr/bin/python

for letter in 'Python':    # First Example

    if letter == 'h':

        break

    print 'Current Letter :', letter


var = 10                    # Second Example

while var > 0:

    print 'Current variable value :', var

    var = var -1

    if var == 5:

        break


print "Good bye!"
```

以上实例执行结果：

```
Current Letter : P

Current Letter : y

Current Letter : t

Current variable value : 10

Current variable value : 9

Current variable value : 8

Current variable value : 7
```

```
Current variable value : 6

Good bye!
```

Python continue 语句

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

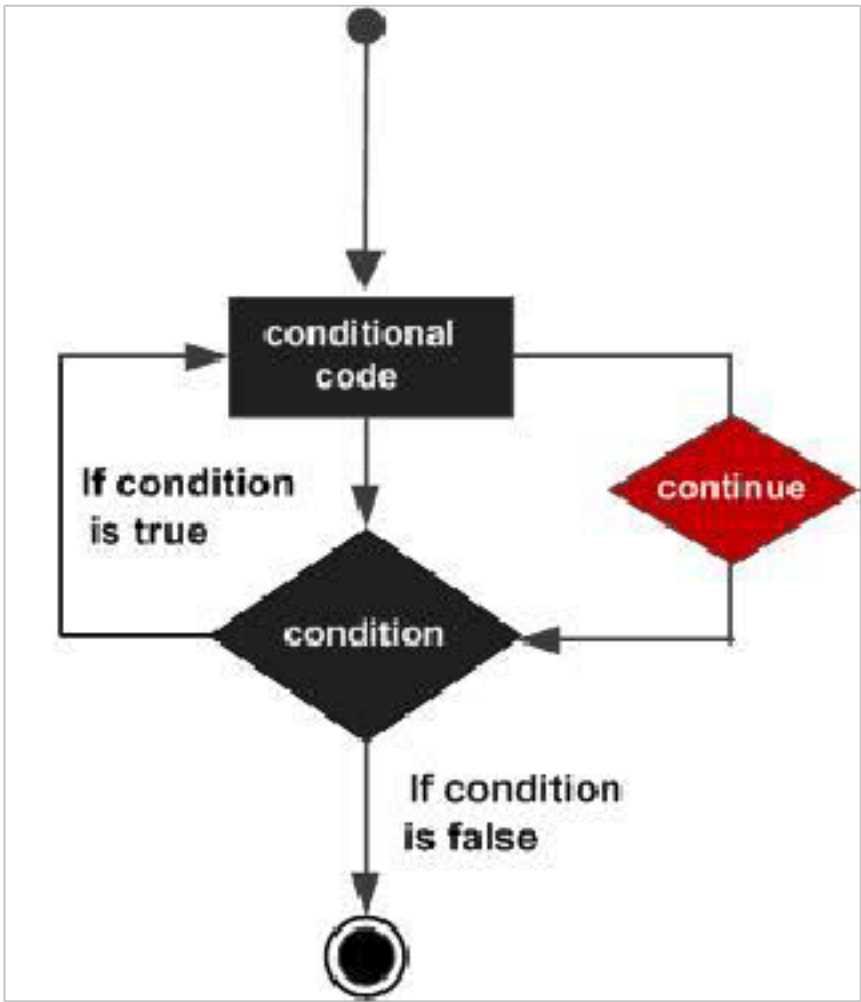
continue 语句用来告诉 Python 跳过当前循环的剩余语句，然后继续进行下一轮循环。

continue 语句用在 while 和 for 循环中。

Python 语言 continue 语句语法格式如下：

```
continue
```

流程图：



实例：

```
#!/usr/bin/python

for letter in 'Python':    # First Example
```

```
    if letter == 'h':

        continue

    print 'Current Letter :', letter


var = 10                                # Second Example

while var > 0:

    var = var -1

    if var == 5:

        continue

    print 'Current variable value :', var

print "Good bye!"
```

以上实例执行结果：

```
Current Letter : P

Current Letter : y

Current Letter : t

Current Letter : o

Current Letter : n

Current variable value : 9

Current variable value : 8

Current variable value : 7

Current variable value : 6

Current variable value : 4

Current variable value : 3
```

```
Current variable value : 2
```

```
Current variable value : 1
```

```
Current variable value : 0
```

```
Good bye!
```

Python `pass` 语句

Python `pass` 是空语句，是为了保持程序结构的完整性。

Python 语言 `pass` 语句语法格式如下：

```
pass
```

实例：

```
#!/usr/bin/python

for letter in 'Python':

    if letter == 'h':

        pass

    print 'This is pass block'

    print 'Current Letter :', letter

print "Good bye!"
```

以上实例执行结果：

```
Current Letter : P
```

```
Current Letter : y
```

```
Current Letter : t
```

```
This is pass block
```

```
Current Letter : h
```

```
Current Letter : o
```

```
Current Letter : n
```

```
Good bye!
```

Python 数字

Python 数字数据类型用于存储数值。

数据类型是不允许改变的,这就意味着如果改变数字数据类型得值，将重新分配内存空间。

以下实例在变量赋值时数字对象将被创建：

```
var1 = 1
```

```
var2 = 10
```

您也可以使用 del 语句删除一些数字对象引用。

del 语句的语法是：

```
del var1[,var2[,var3[....,varN]]]
```

您可以通过使用 del 语句删除单个或多个对象，例如：

```
del var
```

```
del var_a, var_b
```

Python 支持四种不同的数值类型：

- **整型**(Int)-通常被称为是整型或整数，是正或负整数，不带小数点。
- **长整型**(long integers)-无限大小的整数，整数最后是一个大写或小写的 L。

- **浮点型**(floating point real values)浮点型由整数部分与小数部分组成，浮点型也可以使用科学计数法表示 (2.5e2 = 2.5 x 10²= 250)
- **复数**(complex numbers) – 复数的虚部以字母 j 或 J 结尾 。如： 2+3j

int	long	float	complex
10	51924361L	0.0	3.141592653589793j
100	-0x19323L	15.20	45.2389j
-786	0122L	-21.9	9.32156j
080	0xDEFABCECBDAECBFBAE1	32.3+e18	.876j
-0490	535633629843L	-90.	-.6267j
-0x260	-052318172735L	-32.54e100	3e+100j
0x69	-4721885298529L	70.2-E12	4.54j

- 长整型也可以使用小写“l”，但是还是建议您使用大写“L”，避免与数字“1”混淆。Python 使用“L”来显示长整型。
- Python 还支持复数，复数由实数部分和虚数部分构成，可以用 a + bj 或者 complex(a,b) 表示， 复数的实部 a 和虚部 b 都是浮点型

Python 数字类型转换

<code>int(x [,base])</code>	将 x 转换为一个整数
<code>long(x [,base])</code>	将 x 转换为一个长整数
<code>float(x)</code>	将 x 转换到一个浮点数
<code>complex(real [,imag])</code>	创建一个复数
<code>str(x)</code>	将对象 x 转换为字符串
<code>repr(x)</code>	将对象 x 转换为表达式字符串
<code>eval(str)</code>	用来计算在字符串中的有效 Python 表达式,并返回一个对象
<code>tuple(s)</code>	将序列 s 转换为一个元组

<code>list(s)</code>	将序列 <code>s</code> 转换为一个列表
<code>chr(x)</code>	将一个整数转换为一个字符
<code>unichr(x)</code>	将一个整数转换为 Unicode 字符
<code>ord(x)</code>	将一个字符转换为它的整数值
<code>hex(x)</code>	将一个整数转换为一个十六进制字符串
<code>oct(x)</code>	将一个整数转换为一个八进制字符串

Python 数学函数

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

Python 随机数函数

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

Python 包含以下常用随机数函数：

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

Python 三角函数

Python 包括以下三角函数：

函数	描述
<code>acos(x)</code>	返回 x 的反余弦弧度值。
<code>asin(x)</code>	返回 x 的正弦弧度值。
<code>atan(x)</code>	返回 x 的反正切弧度值。
<code>atan2(y, x)</code>	返回给定的 X 及 Y 坐标值的反正切值。
<code>cos(x)</code>	返回 x 的弧度的余弦值。
<code>hypot(x, y)</code>	返回欧几里德范数 $\sqrt{x*x + y*y}$
<code>sin(x)</code>	返回的 x 弧度的正弦值。

<code>tan(x)</code>	返回 x 弧度的正切值。
<code>degrees(x)</code>	将弧度转换为角度,如 <code>degrees(math.pi/2)</code> ， 返回 90.0
<code>radians(x)</code>	将角度转换为弧度

Python 数学常量

常量	描述
pi	数学常量 pi（圆周率，一般以 π 来表示）
e	数学常量 e，e 即自然常数（自然常数）。

Python 字符串

字符串是 Python 中最常用的数据类型。我们可以使用引号来创建字符串。

创建字符串很简单，只要为变量分配一个值即可。例如：

```
var1 = 'Hello World!'

var2 = "Python Programming"
```

Python 访问字符串中的值

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

Python 访问子字符串，可以使用方括号来截取字符串，如下实例：

```
#!/usr/bin/python

var1 = 'Hello World!'

var2 = "Python Programming"
```

```
print "var1[0]: ", var1[0]

print "var2[1:5]: ", var2[1:5]
```

以上实例执行结果:

```
var1[0]:  H

var2[1:5]:  ytho
```

Python 字符串更新

你可以对已存在的字符串进行修改，并赋值给另一个变量，如下实例：

```
#!/usr/bin/python

var1 = 'Hello World!'

print "Updated String :- ", var1[:6] + 'Python'
```

以上实例执行结果

```
Updated String :-  Hello Python
```

Python 转义字符

需要在字符中使用特殊字符时，python 用反斜杠(\)转义字符。如下表：

转义字符	描述
\(在行尾时)	续行符
\\	反斜杠符号

转义字符	描述
\'	单引号
\"	双引号
\a	响铃
\b	退格 (Backspace)
\e	转义
\000	空
\n	换行
\v	纵向制表符
\t	横向制表符
\r	回车
\f	换页
\oyy	八进制数，yy 代表的字符，例如：\o12 代表换行
\xyy	十六进制数，yy 代表的字符，例如：\x0a 代表换行
\other	其它的字符以普通格式输出

Python 字符串运算符

下表实例变量 a 值为字符串"Hello"， b 变量值为"Python"：

操作符	描述	实例
+	字符串连接	a + Hel
*	重复输出字符串	a*2 Hel
[]	通过索引获取字符串中字符	a[1]
[:]	截取字符串中的一部分	a[1]

in	成员运算符 – 如果字符串中包含给定的字符返回 True	H i
not in	成员运算符 – 如果字符串中不包含给定的字符返回 True	M n
r/R	原始字符串 – 原始字符串：所有的字符串都是直接按照字面的意思来使用，没有转义特殊或不能打印的字符。 原始字符串除在字符串的第一个引号前加上字母”r”(可以大小写) 以外，与普通字符串有着几乎完全相同的语法。	pri 和
%	格式字符串	请

Python 字符串格式化

Python 支持格式化字符串的输出 。尽管这样可能会用到非常复杂的表达式，但最基本的用法是将一个值插入到一个有字符串格式符 `%s` 的字符串中。

在 Python 中，字符串格式化使用与 C 中 `sprintf`函数一样的语法。

如下实例：

```
#!/usr/bin/python

print "My name is %s and weight is %d kg!" % ('Zara', 21)
```

以上实例输出结果：

```
My name is Zara and weight is 21 kg!
```

python 字符串格式化符号：

<tbody</tbody

符 号	描述
%c	格式化字符及其 ASCII 码
%s	格式化字符串
%d	格式化整数
%u	格式化无符号整型

%o	格式化无符号八进制数
%x	格式化无符号十六进制数
%X	格式化无符号十六进制数（大写）
%f	格式化浮点数字，可指定小数点后的精度
%e	用科学计数法格式化浮点数
%E	作用同%e，用科学计数法格式化浮点数
%g	%f 和%e 的简写
%G	%f 和 %E 的简写
%p	用十六进制数格式化变量的地址

格式化操作符辅助指令：

符号	功能
*	定义宽度或者小数点精度
-	用做左对齐
+	在正数前面显示加号(+)
<sp>	在正数前面显示空格
#	在八进制数前面显示零('0')，在十六进制前面显示'0x'或者'0X'，取决于用的是'x'还是'X'）
0	显示的数字前面填充'0'而不是默认的空格
%	'%%' 输出一个单一的' %'
(var)	映射变量(字典参数)
m. n.	m 是显示的最小总宽度,n 是小数点后的位数(如果可用的话)

Python 三引号 (triple quotes)

python 中三引号可以将复杂的字符串进行复制：

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

三引号的语法是一对连续的单引号或者双引号（通常都是成对的用）。

```
>>> hi = '''hi
there'''

>>> hi    # repr()

'hi\nthere'

>>> print hi    # str()

hi

there
```

三引号让程序员从引号和特殊字符串的泥潭里面解脱出来，自始至终保持一小块字符串的格式是所谓的 WYSIWYG （所见即所得）格式的。

一个典型的用例是，当你需要一块 HTML 或者 SQL 时，这时用字符串组合，特殊字符串转义将会非常的繁琐。

```
errHTML = '''

<HTML><HEAD><TITLE>

Friends CGI Demo</TITLE></HEAD>

<BODY><H3>ERROR</H3>

<B>%s</B><P>

<FORM><INPUT TYPE=button VALUE=Back

ONCLICK="window.history.back()"></FORM>

</BODY></HTML>

'''

cursor.execute('''

CREATE TABLE users (

login VARCHAR(8),
```

```
uid INTEGER,

prid INTEGER)

'''
```

Unicode 字符串

Python 中定义一个 Unicode 字符串和定义一个普通字符串一样简单：

```
>>> u'Hello World !'

u'Hello World !'
```

引号前小写的“u”表示这里创建的是一个 Unicode 字符串。如果你想加入一个特殊字符，可以使用 Python 的 Unicode-Escape 编码。如下例所示：

```
>>> u'Hello\u0020World !'

u'Hello World !'
```

被替换的 \u0020 标识表示在给定位置插入编码值为 0x0020 的 Unicode 字符（空格符）。

python 的字符串内建函数

字符串方法是从 python1.6 到 2.0 慢慢加进来的——它们也被加到了 Jython 中。

这些方法实现了 string 模块的大部分方法，如下表所示列出了目前字符串内建支持的方法，所有的方法都包含了对 Unicode 的支持，有一些甚至是专门用于 Unicode 的。

方法	描述
<code>string.capitalize()</code>	把字符串的第一个字符大写
<code>string.center(width)</code>	返回一个原字符串居中,并使用空格填充至长度 width 的新字符串
<code>string.count(str, beg=0, end=len(string))</code>	返回 str 在 string 里面出现的次数，如果 beg 或者 end 指定则返回的次数

<code>string.decode(encoding='UTF-8', errors='strict')</code>	以 encoding 指定的编码格式解码 string 如果出错默认报一个 Value 除 非 errors 指 定 的 是 'ignore'或 者'replace'
<code>string.encode(encoding='UTF-8', errors='strict')</code>	以 encoding 指定的编码格式编码 string 如果出错默认报一个 Value errors 指定的是'ignore或者'replace'
<code>string.endswith(obj, beg=0, end=len(string))</code>	检查字符串是否以 obj 结束, 如果 beg 或者 end 指定则检查指定的 束, 如果是, 返回 True, 否则返回 False.
<code>string.expandtabs(tabsize=8)</code>	把字符串 string中的 tab 符号转为空格, 默认的空格数 tabsize 是
<code>string.find(str, beg=0, end=len(string))</code>	检测 str是否包含在 string中, 如果 beg 和 end 指定范围, 则检查 内, 如果是返回开始的索引值, 否则返回-1
<code>string.index(str, beg=0, end=len(string))</code>	跟 find方法一样, 只不过如果 str不在 string中会报一个异常.
<code>string.isalnum()</code>	如果 string至少有一个字符并且所有字符都是字母或数字则返 回 True, 否则返回 False
<code>string.isalpha()</code>	如果 string至少有一个字符并且所有字符都是字母则返回 True, 否则返回 False
<code>string.isdecimal()</code>	如果 string只包含十进制数字则返回 True 否则返回 False.
<code>string.isdigit()</code>	如果 string只包含数字则返回 True 否则返回 False.
<code>string.islower()</code>	如果 string中包含至少一个区分大小写的字符, 并且所有这些(区分大 写, 则返回 True, 否则返回 False
<code>string.isnumeric()</code>	如果 string中只包含数字字符, 则返回 True, 否则返回 False
<code>string.isspace()</code>	如果 string中只包含空格, 则返回 True, 否则返回 False.
<code>string.istitle()</code>	如果 string是标题化的(见 title则返回 True, 否则返回 False
<code>string.isupper()</code>	如果 string中包含至少一个区分大小写的字符, 并且所有这些(区分大 写, 则返回 True, 否则返回 False
<code>string.join(seq)</code>	Merges (concatenates)以 string作为分隔符, 将 seq 中所有的元素(一个新的字符串
<code>string.ljust(width)</code>	返回一个原字符串左对齐, 并使用空格填充至长度 width 的新字符串
<code>string.lower()</code>	转换 string中所有大写字符为小写.

<code>string.lstrip()</code>	截掉 string左边的空格
<code>string.maketrans(intab, outtab]</code>	maketrans() 方法用于创建字符映射的转换表，对于接受两个参数的最一个参数是字符串，表示需要转换的字符，第二个参数也是字符串表示
<code>max(str)</code>	返回字符串 str中最大的字母。
<code>min(str)</code>	返回字符串 str中最小的字母。
<code>string.partition(str)</code>	有点像 find(和 split的结合体,从 str出现的第一个位置起,把字 符个 3 元 素 的 元 组 (string_pre_str, str, string_post如果),stringstring_pre_str == string.
<code>string.replace(str1, str2, num=string.count(str1))</code>	把 string中的 str1 替换成 str2如果 num 指定，则替换不超过 num
<code>string.rfind(str, beg=0, end=len(string)</code>	类似于 find函数，不过是从右边开始查找.
<code>string.rindex(str, beg=0, end=len(string)</code>	类似于 index(), 不过是从右边开始.
<code>string.rjust(width)</code>	返回一个原字符串右对齐,并使用空格填充至长度 width 的新字符串
<code>string.rpartition(str)</code>	类似于 partition函数,不过是从右边开始查找.
<code>string.rstrip()</code>	删除 string字符串末尾的空格.
<code>string.split(str="", num=string.count(str))</code>	以 str为分隔符切片 string 如果 num 有指定值，则仅分隔 num 个
<code>string.splitlines(num=string.count('\n'))</code>	按照行分隔，返回一个包含各行作为元素的列表，如果 num 指定则仅
<code>string.startswith(obj, beg=0, end=len(string))</code>	检查字符串是否是以 obj 开头，是则返回 True，否则返回 False。如值，则在指定范围内检查.
<code>string.strip([obj])</code>	在 string上执行 lstrip和 rstrip()
<code>string.swapcase()</code>	翻转 string中的大小写
<code>string.title()</code>	返回"标题化"的 string就是说所有单词都是以大写开始，其余字母均为
<code>string.translate(str, del="")</code>	根据 str给出的表(包含 256 个字符)转换 string的字符， 要过滤掉的字符放到 del 参数中
<code>string.upper()</code>	转换 string中的小写字母为大写
<code>string.zfill(width)</code>	返回长度为 width 的字符串，原字符串 string右对齐，前面填充 0

`string.isdecimal()`

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

Python 列表(Lists)

序列是 Python 中最基本的数据结构。序列中的每个元素都分配一个数字 - 它的位置，或索引，第一个索引是 0，第二个索引是 1，依此类推。

Python 有 6 个序列的内置类型，但最常见的是列表和元组。

序列都可以进行的操作包括索引，切片，加，乘，检查成员。

此外，Python 已经内置确定序列的长度以及确定最大和最小的元素的方法。

列表是最常用的 Python 数据类型，它可以作为一个方括号内的逗号分隔值出现。

列表的数据项不需要具有相同的类型

创建一个列表，只要把逗号分隔的不同的数据项使用方括号括起来即可。如下所示：

```
list1 = ['physics', 'chemistry', 1997, 2000];
```

```
list2 = [1, 2, 3, 4, 5 ];
```

```
list3 = ["a", "b", "c", "d"];
```

与字符串的索引一样，列表索引从 0 开始。列表可以进行截取、组合等。

访问列表中的值

使用下标索引来访问列表中的值，同样你也可以使用方括号的形式截取字符，如下所示：

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

```
list1 = ['physics', 'chemistry', 1997, 2000];
```

```
list2 = [1, 2, 3, 4, 5, 6, 7 ];
```

```
print "list1[0]: ", list1[0]

print "list2[1:5]: ", list2[1:5]
```

以上实例输出结果：

```
list1[0]:  physics

list2[1:5]:  [2, 3, 4, 5]
```

更新列表

你可以对列表的数据项进行修改或更新，你也可以使用 `append()` 方法来添加列表项，如下所示：

```
#!/usr/bin/python

list = ['physics', 'chemistry', 1997, 2000];

print "Value available at index 2 : "

print list[2];

list[2] = 2001;

print "New value available at index 2 : "

print list[2];
```

注意：我们会在接下来的章节讨论 `append()` 方法的使用

以上实例输出结果：

```
Value available at index 2 :

1997

New value available at index 2 :
```

删除列表元素

可以使用 `del` 语句来删除列表的元素，如下实例：

```
#!/usr/bin/python

list1 = ['physics', 'chemistry', 1997, 2000];

print list1;

del list1[2];

print "After deleting value at index 2 : "

print list1;
```

以上实例输出结果：

```
['physics', 'chemistry', 1997, 2000]

After deleting value at index 2 :

['physics', 'chemistry', 2000]
```

注意：我们会在接下来的章节讨论 `remove()` 方法的使用

Python 列表脚本操作符

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

如下所示：

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

Python 列表截取

Python 的列表截取与字符串操作类型，如下所示：

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

操作：

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

Python 列表函数& 方法

Python 包含以下函数：

序号	函数
1	<u>cmp(list1, list2)</u> 比较两个列表的元素
2	<u>len(list)</u> 列表元素个数
3	<u>max(list)</u>

	返回列表元素最大值
4	<code>min(list)</code> 返回列表元素最小值
5	<code>list(seq)</code> 将元组转换为列表

Python 包含以下方法：

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

Python 元组

Python 的元组与列表类似，不同之处在于元组的元素不能修改。

元组使用小括号，列表使用方括号。

元组创建很简单，只需要在括号中添加元素，并使用逗号隔开即可。

如下实例：

```
tup1 = ('physics', 'chemistry', 1997, 2000);

tup2 = (1, 2, 3, 4, 5 );

tup3 = "a", "b", "c", "d";
```

创建空元组

```
tup1 = ();
```

元组中只包含一个元素时，需要在元素后面添加逗号

```
tup1 = (50,);
```

元组与字符串类似，下标索引从 0 开始，可以进行截取，组合等。

访问元组

元组可以使用下标索引来访问元组中的值，如下实例：

```
#!/usr/bin/python

tup1 = ('physics', 'chemistry', 1997, 2000);

tup2 = (1, 2, 3, 4, 5, 6, 7 );

print "tup1[0]: ", tup1[0]

print "tup2[1:5]: ", tup2[1:5]
```

以上实例输出结果：


```
tup1[0]:  physics
```

```
tup2[1:5]:  (2, 3, 4, 5)
```

修改元组

元组中的元素值是不允许修改的，但我们可以对元组进行连接组合，如下实例：

```
#coding=utf-8

#!/usr/bin/python


tup1 = (12, 34.56);

tup2 = ('abc', 'xyz');


# 以下修改元组元素操作是非法的。

# tup1[0] = 100;


# 创建一个新的元组

tup3 = tup1 + tup2;

print tup3;
```

以上实例输出结果：

```
(12, 34.56, 'abc', 'xyz')
```

删除元组

元组中的元素值是不允许删除的，但我们可以使用 del 语句来删除整个元组，如下实例：

```
#!/usr/bin/python

tup = ('physics', 'chemistry', 1997, 2000);

print tup;

del tup;

print "After deleting tup : "

print tup;
```

以上实例元组被删除后，输出变量会有异常信息，输出如下所示：

```
('physics', 'chemistry', 1997, 2000)

After deleting tup :

Traceback (most recent call last):

  File "test.py", line 9, in <module>

    print tup;

NameError: name 'tup' is not defined
```

元组运算符

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

Python 表达式	结果	描述
len((1, 2, 3))	3	计算元素个数

(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	连接
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	复制
3 in (1, 2, 3)	True	元素是否存在
for x in (1, 2, 3): print x,	1 2 3	迭代

元组索引，截取

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

元组：

```
L = ('spam', 'Spam', 'SPAM!')
```

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

无关闭分隔符

任意无符号的对象，以逗号隔开，默认为元组，如下实例：

```
#!/usr/bin/python

print 'abc', -4.24e93, 18+6.6j, 'xyz';

x, y = 1, 2;

print "Value of x , y : ", x,y;
```

以上实例允许结果：

```
abc -4.24e+93 (18+6.6j) xyz

Value of x , y : 1 2
```

元组内置函数

Python 元组包含了以下内置函数

序号	方法及描述
1	<u>cmp(tuple1, tuple2)</u> 比较两个元组元素。
2	<u>len(tuple)</u> 计算元组元素个数。
3	<u>max(tuple)</u> 返回元组中元素最大值。
4	<u>min(tuple)</u> 返回元组中元素最小值。
5	<u>tuple(seq)</u> 将列表转换为元组。

Python 字典(Dictionary)

字典是另一种可变容器模型，且可存储任意类型对象，如其他容器模型。

字典由键和对应值成对组成。字典也被称作关联数组或哈希表。基本语法如下：

```
dict = {'Alice': '2341', 'Beth': '9102', 'Cecil': '3258'}
```

也可如此创建字典：

```
dict1 = { 'abc': 456 };

dict2 = { 'abc': 123, 98.6: 37 };
```

每个键与值用冒号隔开（:），每对用逗号分割，整体放在花括号中（{}）。

键必须独一无二，但值则不必。

值可以取任何数据类型，但必须是不可变的，如字符串，数或元组。

访问字典里的值

把相应的键放入熟悉的方括弧，如下实例：

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

print "dict['Name']: ", dict['Name'];

print "dict['Age']: ", dict['Age'];
```

以上实例输出结果：

```
dict['Name']:  Zara

dict['Age']:   7
```

如果用字典里没有的键访问数据，会输出错误如下：

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'};

print "dict['Alice']: ", dict['Alice'];
```

以上实例输出结果：

```
dict['Zara']:

Traceback (most recent call last):

  File "test.py", line 4, in <module>

    print "dict['Alice']:", dict['Alice'];

KeyError: 'Alice'
```

修改字典

向字典添加新内容的方法是增加新的键/值对，修改或删除已有键/值对如下实例：

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

dict['Age'] = 8; # update existing entry

dict['School'] = "DPS School"; # Add new entry


print "dict['Age']:", dict['Age'];

print "dict['School']:", dict['School'];
```

以上实例输出结果：

```
dict['Age']:  8

dict['School']:  DPS  School
```

删除字典元素

能删单一的元素也能清空字典，清空只需一项操作。

显示删除一个字典用 del 命令，如下实例：

```
#coding=utf-8

#!/usr/bin/python


dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}


del dict['Name']; # 删除键是'Name'的条目

dict.clear();      #清空字典所有条目

del dict ;          删除字典


print "dict['Age']: ", dict['Age'];

print "dict['School']: ", dict['School'];
```

但这会引发一个异常，因为用 del 后字典不再存在：

```
dict['Age']:

Traceback (most recent call last):

  File "test.py", line 8, in <module>

    print "dict['Age']: ", dict['Age'];

TypeError: 'type' object is unsubscriptable
```

注：del (方法后面也会讨论。

删除字典元素

字典键的特性

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

两个重要的点需要记住：

1) 不允许同一个键出现两次。创建时如果同一个键被赋值两次，后一个值会被记住，如下实例：

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'};

print "dict['Name']: ", dict['Name'];
```

以上实例输出结果：

```
dict['Name']:  Manni
```

2) 键必须不可变，所以可以用数，字符串或元组充当，所以用列表就不行，如下实例：

```
#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7};

print "dict['Name']: ", dict['Name'];
```

以上实例输出结果：


```
Traceback (most recent call last):

  File "test.py", line 3, in <module>

    dict = {'Name': 'Zara', 'Age': 7};

TypeError: list objects are unhashable
```

字典内置函数& 方法

Python 字典包含了以下内置函数：

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

Python 字典包含了以下内置函数：

序号	函数及描述
1	<u>radiansdict.clear()</u> 删除字典内所有元素
2	<u>radiansdict.copy()</u> 返回一个字典的浅复制
3	<u>radiansdict.fromkeys()</u> 创建一个新字典，以序列 seq 中元素做字典的键，val为字典所有键对应的初始值
4	<u>radiansdict.get(key, default=None)</u> 返回指定键的值，如果值不在字典中返回 default值

5	<code>radiansdict.has_key(key)</code> 如果键在字典 dict里返回 true 否则返回 false
6	<code>radiansdict.items()</code> 以列表返回可遍历的(键, 值) 元组数组
7	<code>radiansdict.keys()</code> 以列表返回一个字典所有的键
8	<code>radiansdict.setdefault(key, default=None)</code> 和 get(类似, 但如果键不已经存在于字典中, 将会添加键并将值设为 default
9	<code>radiansdict.update(dict2)</code> 把字典 dict2的键/值对更新到 dict里
10	<code>radiansdict.values()</code> 以列表返回字典中的所有值

Python 日期和时间

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

什么是 Tick?

时间间隔是以秒为单位的浮点小数。

每个时间戳都以自从 1970 年 1 月 1 日午夜（历元）经过了多长时间来表示。

Python 附带的受欢迎的 time 模块下有很多函数可以转换常见日期格式。如函数 time.time(用 ticks 计时单位返回从 12:00am, January 1, 1970(epoch)开始的记录的当前操作系统时间, 如下实例:

```
#!/usr/bin/python

import time;  # This is required to include time module.
```

```
ticks = time.time()

print "Number of ticks since 12:00am, January 1, 1970:", ticks
```

以上实例输出结果：

```
Number of ticks since 12:00am, January 1, 1970: 7186862.73399
```

Tick单位最适于做日期运算。但是 1970 年之前的日期就无法以此表示了。太遥远的日期也不行，UNIX 和 Windows 只支持到 2038 年某日。

什么是时间元组？

很多 Python 函数用一个元组装起来的 9 组数字处理时间：

<td0 到 59< td="" style="color: rgb(0, 0, 0); font-family: 'Microsoft Yahei', 'Helvetica Neue', Helvetica, Arial, sans-serif; font-size: 12px; font-style: normal; font-variant: normal; font-weight: normal; letter-spacing: normal; line-height: normal; orphans: auto; text-align: start; text-indent: 0px; text-transform: none; white-space: normal; widows: auto; word-spacing: 0px; -webkit-text-stroke-width: 0px; background-color: rgb(255, 255, 255);"></td0 到 59<>

序号	字段	值
0	4 位数年	2008
1	月	1 到 12
2	日	1 到 31
3	小时	0 到 23
4	分钟	
5	秒	0 到 61（60或 61 是闰秒）
6	一周的第几日	0 到 6（0是周一）
7	一年的第几日	1 到 366（儒略历）
8	夏令时	-1, 0, 1, 是决定是否为夏令时的旗帜

上述也就是 struct_time元组。这种结构具有如下属性：

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

获取当前时间

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

```
#!/usr/bin/python

import time;

localtime = time.localtime(time.time())

print "Local current time :", localtime
```

以上实例输出结果：

```
Local current time : time.struct_time(tm_year=2013, tm_mon=7,

tm_mday=17, tm_hour=21, tm_min=26, tm_sec=3, tm_wday=2, tm_yday=198, t
m_isdst=0)
```

获取格式化的时间

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

```
#!/usr/bin/python

import time;

localtime = time.asctime( time.localtime(time.time()) )

print "Local current time :", localtime
```

以上实例输出结果：

```
Local current time : Tue Jan 13 10:17:09 2009
```

获取某月日历

`Calendar` 模块有很广泛的方法用来处理年历和月历，例如打印某月的月历：

```
#!/usr/bin/python

import calendar

cal = calendar.month(2008, 1)

print "Here is the calendar:"

print cal;
```

以上实例输出结果：

```
Here is the calendar:

    January 2008

Mo Tu We Th Fr Sa Su

    1  2  3  4  5  6

 7  8  9 10 11 12 13

14 15 16 17 18 19 20

21 22 23 24 25 26 27

28 29 30 31
```

Time 模块

Time 模块包含了以下内置函数，既有时间处理相的，也有转换时间格式的：

序 号	函数及描述
1	<u>time.altzone</u> 返回格林威治西部的夏令时地区的偏移秒数。如果该地区在格林威治东部会返回负值（如西欧，包括英国）。 能使用。
2	<u>time.asctime([tupletime])</u> 接受时间元组并返回一个可读的形式为“Tue Dec 11 18:07:14 2008”（2008 年 12 月 11 日 周二 18 时 07 分 14 秒）的字符串。
3	<u>time.clock()</u> 用以浮点数计算的秒数返回当前的 CPU 时间。用来衡量不同程序的耗时，比 time.time 更有用。
4	<u>time.ctime([secs])</u> 作用相当于 asctime(localtime(secs))未给参数相当于 asctime()
5	<u>time.gmtime([secs])</u> 接收时间辍（1970 纪元后经过的浮点秒数）并返回格林威治天文时间下的时间元组 t。注：t.tm_isds始终为 0。
6	<u>time.localtime([secs])</u>

	接收时间戳（1970 纪元后经过的浮点秒数）并返回当地时间下的时间元组 t(t.tm_isdst可取 0 或 1，取决于当地是否处于夏令时）。
7	<u>time.mktime(tupletime)</u> 接受时间元组并返回时间戳（1970 纪元后经过的浮点秒数）。
8	<u>time.sleep(secs)</u> 推迟调用线程的运行，secs 指秒数。
9	<u>time.strftime(fmt[,tupletime])</u> 接收以时间元组，并返回以可读字符串表示的当地时间，格式由 fmt 决定。
10	<u>time.strptime(str,fmt='%a %b %d %H:%M:%S %Y')</u> 根据 fmt 的格式把一个时间字符串解析为时间元组。
11	<u>time.time()</u> 返回当前时间的时间戳（1970 纪元后经过的浮点秒数）。
12	<u>time.tzset()</u> 根据环境变量 TZ 重新初始化时间相关设置。

Time 模块包含了以下 2 个非常重要的属性：

序号	属性及描述
1	<code>time.timezone</code> 属性 <code>time.timezone</code> 是当地时区（未启动夏令时）距离格林威治的偏移秒数（>0，美洲;<=0 大部分欧洲，亚洲）。
2	<code>time.tzname</code> 属性 <code>time.tzname</code> 包含一对根据情况的不同而不同的字符串，分别是带夏令时的本地时区名称，和不带的。

日历（Calendar）模块

此模块的函数都是日历相关的，例如打印某月的字符月历。

星期一是默认的每周第一天，星期天是默认的最后一天。更改设置需调用 `calendar.setfirstweekday()` 函数。模块包含了以下内置函数：

序号	函数及描述
1	<code>calendar.calendar(year, w=2, l=1, c=6)</code> 返回一个多行字符串格式的 year 年年历，3 个月一行，间隔距离为 c。 每日宽度间隔为 w 字符。每行长度为 14w。

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

其他相关模块和函数

在 Python 中，其他处理日期和时间的模块还有：

- `datetime` 模块
- `pytz` 模块

-
- [dateutil](#)模块
-

Python 函数

函数是组织好的，可重复使用的，用来实现单一，或相关联功能的代码段。

函数能提高应用的模块性，和代码的重复利用率。你已经知道 Python 提供了许多内建函数，比如 `print()` 但你也可以自己创建函数，这被叫做用户自定义函数。

定义一个函数

你可以定义一个由自己想要功能的函数，以下是简单的规则：

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

语法

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

默认情况下，参数值和参数名称是按函数声明中定义的的顺序匹配起来的。

实例

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

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

```
"打印传入的字符串到标准显示设备上"
```

```
print str
```

```
return
```

函数调用

定义一个函数只给了函数一个名称，指定了函数里包含的参数，和代码块结构。

这个函数的基本结构完成以后，你可以通过另一个函数调用执行，也可以直接从 Python 提示符执行。

如下实例调用了 printme () 函数：

```
#coding=utf-8

#!/usr/bin/python


# Function definition is here

def printme( str ):

    "打印任何传入的字符串"

    print str;

    return;


# Now you can call printme function

printme("我要调用用户自定义函数!");

printme("再次调用同一函数");
```

以上实例输出结果：

```
我要调用用户自定义函数！
```

按值传递参数和按引用传递参数

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

```
#coding=utf-8

#!/usr/bin/python

# 可写函数说明

def changeme( mylist ):

    "修改传入的列表"

    mylist.append([1,2,3,4]);

    print "函数内取值：", mylist

    return

# 调用 changeme 函数

mylist = [10,20,30];

changeme( mylist );

print "函数外取值：", mylist
```

传入函数的和在末尾添加新内容的对象用的是同一个引用。故输出结果如下：

```
函数内取值： [10, 20, 30, [1, 2, 3, 4]]

函数外取值： [10, 20, 30, [1, 2, 3, 4]]
```

参数

以下是调用函数时可使用的正式参数类型：

- 必备参数
- 命名参数
- 缺省参数
- 不定长参数

必备参数

必备参数须以正确的顺序传入函数。调用时的数量必须和声明时的一样。

调用 printme 函数，你必须传入一个参数，不然会出现语法错误：

```
#coding=utf-8

#!/usr/bin/python

#可写函数说明

def printme( str ):

    "打印任何传入的字符串"

    print str;

    return;


#调用 printme 函数

printme();
```

以上实例输出结果：

```
Traceback (most recent call last):
```

```
File "test.py", line 11, in <module>

    printme();

TypeError: printme() takes exactly 1 argument (0 given)
```

命名参数

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

```
#coding=utf-8

#!/usr/bin/python

#可写函数说明

def printme( str ):

    "打印任何传入的字符串"

    print str;

    return;


#调用 printme 函数

printme( str = "My string");
```

以上实例输出结果：

```
My string
```

下例能将命名参数顺序不重要展示得更清楚：

```
#coding=utf-8

#!/usr/bin/python
```

#可写函数说明

```
def printinfo( name, age ):
```

```
    "打印任何传入的字符串"
```

```
    print "Name: ", name;
```

```
    print "Age ", age;
```

```
    return;
```

#调用 printinfo 函数

```
printinfo( age=50, name="miki" );
```

以上实例输出结果:

```
Name:  miki
```

```
Age   50
```

缺省参数

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

```
#coding=utf-8
```

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

#可写函数说明

```
def printinfo( name, age = 35 ):
```

```
    "打印任何传入的字符串"
```

```
    print "Name: ", name;

    print "Age ", age;

    return;

#调用 printinfo 函数

printinfo( age=50, name="miki" );

printinfo( name="miki" );
```

以上实例输出结果：

```
Name:  miki

Age   50

Name:  miki

Age   35
```

不定长参数

你可能需要一个函数能处理比当初声明时更多的参数。这些参数叫做不定长参数，和上述 2 种参数不同，声明时不会命名。基本语法如下：

```
def functionname([formal_args,] *var_args_tuple ):

    '函数_文档字符串'

    function_suite

    return [expression]
```

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

```
#coding=utf-8

#!/usr/bin/python
```

```
# 可写函数说明

def printinfo( arg1, *vartuple ):

    "打印任何传入的参数"

    print '输出: '

    print arg1

    for var in vartuple:

        print var

    return;


# 调用 printinfo 函数

printinfo( 10 );

printinfo( 70, 60, 50 );
```

以上实例输出结果:

输出:

10

输出:

70

60

50

匿名函数

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

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

语法

lambda 函数的语法只包含一个语句，如下：

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

如下实例：

```
#coding=utf-8

#!/usr/bin/python

#可写函数说明

sum = lambda arg1, arg2: arg1 + arg2;

#调用 sum 函数

print "Value of total : ", sum( 10, 20 )

print "Value of total : ", sum( 20, 20 )
```

以上实例输出结果：

```
Value of total :  30

Value of total :  40
```

return 语句

return语句[表达式]退出函数，选择性地向调用方返回一个表达式。不带参数值的 return语句返回 None 。之前的例子都没有示范如何返回数值，下例便告诉你怎么做：

```
#coding=utf-8

#!/usr/bin/python


# 可写函数说明

def sum( arg1, arg2 ):

    # 返回 2 个参数的和."

    total = arg1 + arg2

    print "Inside the function : ", total

    return total;


# 调用 sum 函数

total = sum( 10, 20 );

print "Outside the function : ", total
```

以上实例输出结果：

```
Inside the function :  30

Outside the function :  30
```

变量作用域

一个程序的所有的变量并不是在哪个位置都可以访问的。访问权限决定于这个变量是在哪里赋值的。

变量的作用域决定了在哪一部分程序你可以访问哪个特定的变量名称。两种最基本的变量作用域如下：

- 全局变量
- 局部变量

变量和局部变量

定义在函数内部的变量拥有一个局部作用域，定义在函数外的拥有全局作用域。

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

```
#coding=utf-8

#!/usr/bin/python


total = 0; # This is global variable.

# 可写函数说明

def sum( arg1, arg2 ):

    #返回 2 个参数的和."

    total = arg1 + arg2; # tota在这里是局部变量.

    print "Inside the function local total : ", total

    return total;


#调用 sum 函数

sum( 10, 20 );

print "Outside the function global total : ", total
```

以上实例输出结果：

```
Inside the function local total :  30

Outside the function global total :  0
```

Python 模块

模块让你能够有逻辑地组织你的 Python 代码段。

把相关的代码分配到一个 模块里能让你的代码更好用，更易懂。

模块也是 Python 对象，具有随机的名字属性用来绑定或引用。

简单地说，模块就是一个保存了 Python 代码的文件。模块能定义函数，类和变量。模块里也能包含可执行的代码。

例子

一个叫做 aname 的模块里的 Python 代码一般都能在一个叫 aname.py 的文件中找到。下例是个简单的模块 support.py。

```
def print_func( par ) :

    print "Hello : ", par

    return
```

import 语句

想使用 Python 源文件，只需在另一个源文件里执行 import 语句，语法如下：

```
import module1[, module2[,... moduleN]
```

当解释器遇到 import 语句，如果模块在当前的搜索路径就会被导入。

搜索路径是一个解释器会先进行搜索的所有目录的列表。如想要导入模块 hello.py 需要把命令放在脚本的顶端：

```
#coding=utf-8

#!/usr/bin/python


# 导入模块

import support


# 现在可以调用模块里包含的函数了

support.print_func("Zara")
```

以上实例输出结果：

```
Hello : Zara
```

一个模块只会被导入一次，不管你执行了多少次 import。这样可以防止导入模块被一遍又一遍地执行。

From...import 语句

Python 的 from 语句让你从模块中导入一个指定的部分到当前命名空间中。语法如下：

```
from modname import name1[, name2[, ... nameN]]
```

例如，要导入模块 fib 的 fibonacc 函数，使用如下语句：

```
from fib import fibonacc
```

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

From...import* 语句

把一个模块的所有内容全都导入到当前的命名空间也是可行的，只需使用如下声明：

```
from modname import *
```

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

定位模块

当你导入一个模块，Python 解析器对模块位置的搜索顺序是：

- 当前目录
- 如果不在当前目录，Python 则搜索在 shell 变量 PYTHONPATH 下的每个目录。
- 如果都找不到，Python 会察看默认路径。UNIX 下，默认路径一般为/usr/local/lib/python/

模块搜索路径存储在 system 模块的 sys.path 变量中。变量里包含当前目录，PYTHONPATH 和由安装过程决定的默认目录。

PYTHONPATH 变量

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

在 Windows 系统，典型的 PYTHONPATH 如下：

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

在 UNIX 系统，典型的 PYTHONPATH 如下：

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

命名空间和作用域

变量是拥有匹配对象的名字（标识符）。命名空间是一个包含了变量名称们（键）和它们各自相应的对象们（值）的字典。

一个 Python 表达式可以访问局部命名空间和全局命名空间里的变量。如果一个局部变量和一个全局变量重名，则局部变量会覆盖全局变量。

每个函数都有自己的命名空间。类的方法的作用域规则和通常函数的一样。

Python 会智能地猜测一个变量是局部的还是全局的，它假设任何在函数内赋值的变量都是局部的。

因此，如果要给全局变量在一个函数里赋值，必须使用 `global` 语句。

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

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

```
#coding=utf-8

#!/usr/bin/python


Money = 2000

def AddMoney():

    # 想改正代码就取消以下注释：

    # global Money

    Money = Money + 1


print Money
```

```
AddMoney()  
  
print Money
```

dir(函数

dir(函数一个排好序的字符串列表，内容是一个模块里定义过的名字。

返回的列表容纳了在一个模块里定义的所有模块，变量和函数。如下一个简单的实例：

```
#coding=utf-8  
  
#!/usr/bin/python  
  
# 导入内置 math 模块  
  
import math  
  
  
content = dir(math)  
  
  
print content;
```

以上实例输出结果：

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

在这里，特殊字符串变量__name__ 指向模块的名字，__file__指向该模块的导入文件名。

globals()和 locals()函数

根据调用地方的不同，globals()和 locals()函数可被用来返回全局和局部命名空间里的名字。

如果在函数内部调用 locals() 返回的是所有能在该函数里访问的命名。

如果在函数内部调用 globals() 返回的是所有在该函数里能访问的全局名字。

两个函数的返回类型都是字典。所以名字们能用 keys()函数摘取。

reload()函数

当一个模块被导入到一个脚本，模块顶层部分的代码只会被执行一次。

因此，如果你想重新执行模块里顶层部分的代码，可以用 reload()函数。该函数会重新导入之前导入过的模块。语法如下：

```
reload(module_name)
```

在这里，module_name 要直接放模块的名字，而不是一个字符串形式。比如想重载 hello模块，如下：

```
reload(hello)
```

Python 中的包

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

考虑一个在 Phone 目录下的 pots.py文件。这个文件有如下源代码：

```
#coding=utf-8

#!/usr/bin/python


def Pots():

    print "I'm Pots Phone"
```

同样地，我们有另外两个保存了不同函数的文件：

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

现在，在 Phone 目录下创建 file __init__.py

- Phone/__init__.py

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

```
from Pots import Pots

from Isdn import Isdn

from G3 import G3
```

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

```
#coding=utf-8

#!/usr/bin/python


# Now import your Phone Package.

import Phone
```

```
Phone.Pots()
```

```
Phone.Isdn()
```

```
Phone.G3()
```

以上实例输出结果：

```
I'm Pots Phone
```

```
I'm 3G Phone
```

```
I'm ISDN Phone
```

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

Python 文件 I/O

本章只讲述所有基本的 I/O 函数，更多函数请参考 Python 标准文档。

打印到屏幕

最简单的输出方法是用 `print` 语句，你可以给它传递零个或多个用逗号隔开的表达式。此函数把你传递的表达式转换成一个字符串表达式，并将结果写到标准输出如下：

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

```
print "Python is really a great language,", "isn't it?";
```

你的标准屏幕上会产生以下结果：

```
Python is really a great language, isn't it?
```

读取键盘输入

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

- raw_input
- input

raw_input 函数

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

```
#!/usr/bin/python

str = raw_input("Enter your input: ");

print "Received input is : ", str
```

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

```
Enter your input: Hello Python

Received input is :  Hello Python
```

input 函数

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

```
#!/usr/bin/python

str = input("Enter your input: ");

print "Received input is : ", str
```

这会产生如下的对应着输入的结果：

```
Enter your input: [x*5 for x in range(2,10,2)]

Recieved input is :  [10, 20, 30, 40]
```

打开和关闭文件

到现在为止，您已经可以向标准输入和输进行读写。现在，来看看怎么读写实际的数据文件。

Python 提供了必要的函数和方法进行默认情况下的文件基本操作。你可以用 `file`对象做大部分的文件操作。

open 函数

你必须先用 Python 内置的 `open()`函数打开一个文件，创建一个 `file`对象，相关的辅助方法才可以调用它进行读写。

语法：

```
file object = open(file_name [, access_mode][, buffering])
```

各个参数的细节如下：

- `file_name`: `file_name` 变量是一个包含了你要访问的文件名称的字符串值。
- `access_mode` : `access_mode` 决定了打开文件的模式：只读，写入，追加等。所有可取值见如下的完全列表。这个参数是非强制的，默认文件访问模式为只读 (`r`)
- `buffering`如果 `buffering` 的值被设为 0，就不会有寄存。如果 `buffering` 的值取 1，访问文件时会寄存行。如果将 `buffering` 的值设为大于 1 的整数，表明了这就是的寄存区的缓冲大小。如果取负值，寄存区的缓冲大小则为系统默认。

不同模式打开文件的完全列表：

模式	描述
r	以只读方式打开文件。文件的指针将会放在文件的开头。这是默认模式。
rb	以二进制格式打开一个文件用于只读。文件指针将会放在文件的开头。这是默认模式。
r+	打开一个文件用于读写。文件指针将会放在文件的开头。
rb+	以二进制格式打开一个文件用于读写。文件指针将会放在文件的开头。
w	打开一个文件只用于写入。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
wb	以二进制格式打开一个文件只用于写入。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文
w+	打开一个文件用于读写。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。

wb+	以二进制格式打开一个文件用于读写。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
a	打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会后。如果该文件不存在，创建新文件进行写入。
ab	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，到已有内容之后。如果该文件不存在，创建新文件进行写入。
a+	打开一个文件用于读写。如果该文件已存在，文件指针将会放在文件的结尾。文件打开时会是追加模式（即只能从文件末尾进行写入）。如果该文件不存在，创建新文件用于读写。
ab+	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。如果该文件用于读写。

File对象的属性

一个文件被打开后，你有一个 `file` 对象，你可以得到有关该文件的各种信息。

以下是和 `file` 对象相关的所有属性的列表：

属性	描述
<code>file.closed</code>	返回 <code>true</code> 如果文件已被关闭，否则返回 <code>false</code>
<code>file.mode</code>	返回被打开文件的访问模式。
<code>file.name</code>	返回文件的名称。
<code>file.softspace</code>	如果用 <code>print</code> 输出后，必须跟一个空格符，则返回 <code>false</code> ，否则返回 <code>true</code>

如下实例：

```
#coding=utf-8

#!/usr/bin/python

# 打开一个文件

fo = open("foo.txt", "wb")

print "Name of the file: ", fo.name

print "Closed or not : ", fo.closed
```

```
print "Opening mode : ", fo.mode

print "Softspace flag : ", fo.softspace
```

以上实例输出结果:

```
Name of the file:  foo.txt

Closed or not :  False

Opening mode :  wb

Softspace flag :  0
```

Close()方法

File对象的 close() 方法刷新缓冲区里任何还没写入的信息，并关闭该文件，这之后便不能再进行写入。

当一个文件对象的引用被重新指定给另一个文件时，Python 会关闭之前的文件。用 close() 方法关闭文件是一个很好的习惯。

语法:

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

例子:

```
#coding=utf-8

#!/usr/bin/python

# 打开一个文件

fo = open("foo.txt", "wb")

print "Name of the file: ", fo.name
```

```
# 关闭打开的文件
```

```
fo.close()
```

以上实例输出结果：

```
Name of the file:  foo.txt
```

读写文件：

file对象提供了一系列方法，能让我们的文件访问更轻松。来看看如何使用 read()和 write()方法来读取和写入文件。

Write()方法

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

Write()方法不在字符串的结尾不添加换行符（'\n'）

语法：

```
fileObject.write(string);
```

在这里，被传递的参数是要写入到已打开文件的内容。

例子：

```
#coding=utf-8
```

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

```
# 打开一个文件
```

```
fo = open("/tmp/foo.txt", "wb")
```

```
fo.write( "Python is a great language.\nYeah its great!!\n");
```



```
# 关闭打开的文件
```

```
fo.close()
```

上述方法会创建 `foo.tx` 文件，并将收到的内容写入该文件，并最终关闭文件。如果你打开这个文件，将看到以下内容：

```
Python is a great language.
```

```
Yeah its great!!
```

`read()`方法

`read()` 方法从一个打开的文件中读取一个字符串。需要重点注意的是，Python 字符串可以是二进制数据，而不是仅仅是文字。

语法：

```
fileObject.read([count]);
```

在这里，被传递的参数是要从已打开文件中读取的字节计数。该方法从文件的开头开始读入，如果没有传入 `count`，它会尝试尽可能多地读取更多的内容，很可能是直到文件的末尾。

例子：

就用我们上面创建的文件 `foo.txt`

```
#coding=utf-8
```

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

```
# 打开一个文件
```

```
fo = open("/tmp/foo.txt", "r+")
```

```
str = fo.read(10);
```

```
print "Read String is : ", str
```

```
# 关闭打开的文件
```

```
fo.close()
```

以上实例输出结果：

```
Read String is : Python is
```

文件位置：

Tell 方法告诉你文件内的当前位置；换句话说，下一次的读写会发生在文件开头这么多字节之后：

seek (offset [,from])方法改变当前文件的位置。Offset变量表示要移动的字节数。From 变量指定开始移动字节的参考位置。

如果 from 被设为 0，这意味着将文件的开头作为移动字节的参考位置。如果设为 1，则使用当前的位置作为参考位置。如果它被设为 2，那么该文件的末尾将作为参考位置。

例子：

就用我们上面创建的文件 foo.txt

```
#coding=utf-8

#!/usr/bin/python


# 打开一个文件

fo = open("/tmp/foo.txt", "r+")

str = fo.read(10);

print "Read String is : ", str


# 查找当前位置

position = fo.tell();

print "Current file position : ", position
```

```
# 把指针再次重新定位到文件开头

position = fo.seek(0, 0);

str = fo.read(10);

print "Again read String is : ", str

# 关闭打开的文件

fo.close()
```

以上实例输出结果：

```
Read String is :  Python is

Current file position :  10

Again read String is :  Python is
```

重命名和删除文件

Python 的 os 模块提供了帮你执行文件处理操作的方法，比如重命名和删除文件。

要使用这个模块，你必须先导入它，然后可以调用相关的各种功能。

rename() 方法：

rename() 方法需要两个参数，当前的文件名和新文件名。

语法：

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

例子：

下例将重命名一个已经存在的文件 test1.txt

```
#coding=utf-8

#!/usr/bin/python
```

```
import os

# 重命名文件 test1.txt 到 test2.txt。

os.rename( "test1.txt", "test2.txt" )
```

remove() 方法

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

语法：

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

例子：

下例将删除一个已经存在的文件 `test2.txt`

```
#coding=utf-8

#!/usr/bin/python

import os

# 删除一个已经存在的文件 test2.txt

os.remove("text2.txt")
```

Python 里的目录：

所有文件都包含在各个不同的目录下，不过 Python 也能轻松处理。os 模块有许多方法能帮你创建，删除和更改目录。

mkdir()方法

可以使用 os 模块的 `mkdir()`方法在当前目录下创建新的目录们。你需要提供一个包含了要创建的目录名称的参数。

语法:

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

例子:

下例将在当前目录下创建一个新目录 test

```
#coding=utf-8

#!/usr/bin/python

import os


# 创建目录 test

os.mkdir("test")
```

chdir()方法

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

语法:

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

例子:

下例将进入"/home/newdir"目录。

```
#coding=utf-8

#!/usr/bin/python

import os
```

```
# 将当前目录改为"/home/newdir"
```

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

getcwd()方法:

getcwd()方法显示当前的工作目录。

语法:

```
os.getcwd()
```

例子:

下例给出当前目录:

```
#coding=utf-8
```

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

```
import os
```

```
# 给出当前的目录
```

```
os.getcwd()
```

rmdir()方法

rmdir()方法删除目录，目录名称以参数传递。

在删除这个目录之前，它的所有内容应该先被清除。

语法:

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

例子:

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

```
#coding=utf-8

#!/usr/bin/python

import os


# 删除"/tmp/test"目录

os.rmdir( "/tmp/test" )
```

文件、目录相关的方法

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

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

Python 异常处理

python 提供了两个非常重要的功能来处理 python 程序在运行中出现的异常和错误。你可以使用该功能来调试 python 程序。

- 异常处理：本站 Python 教程会具体介绍。
- 断言 (Assertions)本站 Python 教程会具体介绍。

python 标准异常

异常名称	描述
BaseException	所有异常的基类
SystemExit	解释器请求退出
KeyboardInterrupt	用户中断执行 (通常是输入^C)
Exception	常规错误的基类

StopIteration	迭代器没有更多的值
GeneratorExit	生成器(generator)发生异常来通知退出
StandardError	所有的内建标准异常的基类
ArithmeticError	所有数值计算错误的基类
FloatingPointError	浮点计算错误
OverflowError	数值运算超出最大限制
ZeroDivisionError	除(或取模)零 (所有数据类型)
AssertionError	断言语句失败
AttributeError	对象没有这个属性
EOFError	没有内建输入,到达 EOF 标记
EnvironmentError	操作系统错误的基类
IOError	输入/输出操作失败
OSError	操作系统错误
WindowsError	系统调用失败
ImportError	导入模块/对象失败
LookupError	无效数据查询的基类
IndexError	序列中没有此索引(index)
KeyError	映射中没有这个键
MemoryError	内存溢出错误(对于 Python 解释器不是致命的)
NameError	未声明/初始化对象 (没有属性)
UnboundLocalError	访问未初始化的本地变量
ReferenceError	弱引用(Weak reference)试图访问已经垃圾回收了的对象
RuntimeError	一般的运行时错误
NotImplementedError	尚未实现的方法

SyntaxError	Python 语法错误
IndentationError	缩进错误
TabError	Tab 和空格混用
SystemError	一般的解释器系统错误
TypeError	对类型无效的操作
ValueError	传入无效的参数
UnicodeError	Unicode 相关的错误
UnicodeDecodeError	Unicode 解码时的错误
UnicodeEncodeError	Unicode 编码时错误
UnicodeTranslateError	Unicode 转换时错误
Warning	警告的基类
DeprecationWarning	关于被弃用的特征的警告
FutureWarning	关于构造将来语义会有改变的警告
OverflowWarning	旧的关于自动提升为长整型(long)的警告
PendingDeprecationWarning	关于特性将会被废弃的警告
RuntimeWarning	可疑的运行时行为(runtime behavior)的警告
SyntaxWarning	可疑的语法的警告
UserWarning	用户代码生成的警告

什么是异常？

异常即是一个事件，该事件会在程序执行过程中发生，影响了程序的正常执行。

一般情况下，在 Python 无法正常处理程序时就会发生一个异常。

异常是 Python 对象，表示一个错误。

当 Python 脚本发生异常时我们需要捕获处理它，否则程序会终止执行。

异常处理

捕捉异常可以使用 try/except 语句。

try/except 语句用来检测 try 语句块中的错误，从而让 except 语句捕获异常信息并处理。

如果你不想在异常发生时结束你的程序，只需在 try 里捕获它。

语法：

以下为简单的 try...except... 的语法：

try:

<语句> 运行别的代码

except <名字>:

<语句> 如果在 try 部份引发了 'name' 异常

except <名字>, <数据>:

<语句> 如果引发了 'name' 异常，获得附加的数据

else:

<语句> 如果没有异常发生

try 的工作原理是，当开始一个 try 语句后，python 就在当前程序的上下文中作标记，这样当异常出现时就可以回到这里，try 子句先执行，接下来会发生什么依赖于执行时是否出现异常。

- 如果当 try 后的语句执行时发生异常，python 就跳回到 try 并执行第一个匹配该异常的 except 子句，异常处理完毕，控制流就通过整个 try 语句（除非在处理异常时又引发新的异常）。
- 如果在 try 后的语句里发生了异常，却没有匹配的 except 子句，异常将被递交到上层的 try 或者到程序的最上层（这样将结束程序，并打印缺省的出错信息）。
- 如果在 try 子句执行时没有发生异常，python 将执行 else 语句后的语句（如果有 else 的话），然后控制流通过整个 try 语句。

实例

下面是简单的例子，它打开一个文件，在该文件中的内容写入内容，且并未发生异常：

```
#!/usr/bin/python

try:

    fh = open("testfile", "w")

    fh.write("This is my test file for exception handling!!")

except IOError:

    print "Error: can\'t find file or read data"

else:

    print "Written content in the file successfully"

    fh.close()
```

以上程序输出结果:

```
Written content in the file successfully
```

实例

下面是简单的例子，它打开一个文件，在该文件中的内容写入内容，但文件没有写入权限，发生了异常：

```
#!/usr/bin/python

try:

    fh = open("testfile", "w")

    fh.write("This is my test file for exception handling!!")

except IOError:

    print "Error: can\'t find file or read data"
```

```
else:

    print "Written content in the file successfully"
```

以上程序输出结果:

```
Error: can't find file or read data
```

使用 except 而不带任何异常类型

你可以不带任何异常类型使用 except，如下实例:

```
try:

    You do your operations here;

    .....

except:

    If there is any exception, then execute this block.

    .....

else:

    If there is no exception then execute this block.
```

以上方式 try-except 语句捕获所有发生的异常。但这不是一个很好的方式，我们不能通过该程序识别出具体的异常信息。因为它捕获所有的异常。

使用 except 而带多种异常类型

你也可以使用相同的 except 语句来处理多个异常信息，如下所示:

```
try:

    You do your operations here;
```

```
.....

except (Exception1[, Exception2[,...ExceptionN]]):

    If there is any exception from the given exception list,

    then execute this block.

    .....

else:

    If there is no exception then execute this block.
```

try-finally语句

try-finally语句无论是否发生异常都将执行最后的代码。

```
try:

<语句>

finally:

<语句>      #退出 try 时总会执行

raise
```

注意: 你可以使用 except 语句或者 finally 语句,但是两者不能同时使用。else 语句也不能与 finally 语句同时使用

实例

```
#!/usr/bin/python

try:

    fh = open("testfile", "w")
```

```
        fh.write("This is my test file for exception handling!!")

finally:

    print "Error: can\'t find file or read data"
```

如果打开的文件没有可写权限，输出如下所示：

```
Error: can't find file or read data
```

同样的例子也可以写成如下方式：

```
#!/usr/bin/python

try:

    fh = open("testfile", "w")

    try:

        fh.write("This is my test file for exception handling!!")

    finally:

        print "Going to close the file"

        fh.close()

except IOError:

    print "Error: can\'t find file or read data"
```

当在 try 块中抛出一个异常，立即执行 final 块代码。

final 块中的所有语句执行后，异常被再次提出，并执行 except 块代码。

参数的内容不同于异常。

异常的参数

一个异常可以带上参数，可作为输出的异常信息参数。

你可以通过 `except` 语句来捕获异常的参数，如下所示：

```
try:

    You do your operations here;

    .....

except ExceptionType, Argument:

    You can print value of Argument here...
```

变量接收的异常值通常包含在异常的语句中。在元组的表单中变量可以接收一个或者多个值。

元组通常包含错误字符串，错误数字，错误位置。

实例

以下为单个异常的实例：

```
#!/usr/bin/python

# Define a function here.

def temp_convert(var):

    try:

        return int(var)

    except ValueError, Argument:

        print "The argument does not contain numbers\n", Argument

# Call above function here.

temp_convert("xyz");
```

以上程序执行结果如下：

```
The argument does not contain numbers

invalid literal for int() with base 10: 'xyz'
```

触发异常

我们可以使用 raise 语句自己触发异常

raise 语法格式如下：

```
raise [Exception [, args [, traceback]]]
```

语句中 Exception 是异常的类型（例如，NameError ）参数是一个异常参数值。该参数是可选的，如果不提供，异常的参数是“None”。

最后一个参数是可选的（在实践中很少使用），如果存在，是跟踪异常对象。

实例

一个异常可以是一个字符串，类或对象。 Python 的内核提供的异常，大多数都是实例化的类，这是一个类的实例的参数。

定义一个异常非常简单，如下所示：

```
def functionName( level ):

    if level < 1:

        raise "Invalid level!", level

        # The code below to this would not be executed

        # if we raise the exception
```

注意：为了能够捕获异常，“except”语句必须有用相同的异常来抛出类对象或者字符串。

例如我们捕获以上异常，“except”语句如下所示：


```
try:

    Business Logic here...

except "Invalid level!":

    Exception handling here...

else:

    Rest of the code here...
```

用户自定义异常

通过创建一个新的异常类，程序可以命名它们自己的异常。异常应该是典型的继承自 `Exception` 类，通过直接或间接的方式。

以下为与 `RuntimeError` 相关的实例,实例中创建了一个类，基类为 `RuntimeError`，用于在异常触发时输出更多的信息。

在 `try` 语句块中，用户自定义的异常后执行 `except` 块语句，变量 `e` 是用于创建 `Networkerror` 类的实例。

```
class Networkerror(RuntimeError):

    def __init__(self, arg):

        self.args = arg
```

在你定义以上类后，你可以触发该异常，如下所示：

```
try:

    raise Networkerror("Bad hostname")

except Networkerror,e:

    print e.args
```

Python 面向对象

Python 从设计之初就已经是一门面向对象的语言，正因为如此，在 Python 中创建一个类和对象是很容易的。本章节我们将详细介绍 Python 的面向对象编程。

如果你以前没有接触过面向对象的编程语言，那你可能需要先了解一些面向对象语言的一些基本特征，在头脑里头形成一个基本的面向对象的概念，这样有助于你更容易的学习 Python 的面向对象编程。

接下来我们先来简单的了解下面面向对象的一些基本特征。

面向对象技术简介

- **类(Class)**:用来描述具有相同的属性和方法的对象的集合。它定义了该集合中每个对象所共有的属性和方法。对象是类的实例。
- **类变量**: 类变量在整个实例化的对象中是公用的。类变量定义在类中且在函数体之外。类变量通常不作为实例变量使用。
- **数据成员**: 类变量或者实例变量用于处理类及其实例对象的相关的数据。
- **方法重载**: 如果从父类继承的方法不能满足子类的需求，可以对其进行改写，这个过程叫方法的覆盖（override），也称为方法的重载。
- **实例变量**: 定义在方法中的变量，只作用于当前实例的类。
- **继承**: 即一个派生类（derived class）继承基类（base class）的字段和方法。继承也允许把一个派生类的对象作为一个基类对象对待。例如，有这样一个设计：一个 Dog 类型的对象派生自 Animal 类,这是模拟“是一个(is-a)”关系(例图,Dog 是一个 Animal)。
- **实例化**: 创建一个类的实例，类的具体对象。
- **方法**: 类中定义的函数。
- **对象**: 通过类定义的数据结构实例。对象包括两个数据成员（类变量和实例变量）和方法。

创建类

使用 class 语句来创建一个新类，class 之后为类的名称并以冒号结尾，如下实例：

```
class ClassName:

    '类的帮助信息'    #类文档字符串

    class_suite    #类体
```

类的帮助信息可以通过 ClassName.__doc__ 查看。

class_suite 由类成员，方法，数据属性组成。

实例

以下是一个简单的 Python 类实例：

```
#coding=utf-8

class Employee:

    '所有员工的基类'

    empCount = 0

    def __init__(self, name, salary):

        self.name = name

        self.salary = salary

        Employee.empCount += 1

    def displayCount(self):

        print "Total Employee %d" % Employee.empCount

    def displayEmployee(self):
```

```
print "Name : ", self.name, ", Salary: ", self.salary
```

- empCount 变量是一个类变量，它的值将在这个类的所有实例之间共享。你可以在内部类或外部类使用 Employee.empCount 访问。
- 第一种方法__init__ 方法是一种特殊的方法，被称为类的构造函数或初始化方法，当创建了这个类的实例时就会调用该方法

创建实例对象

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

"创建 Employee 类的第一个对象"

```
emp1 = Employee("Zara", 2000)
```

"创建 Employee 类的第二个对象"

```
emp2 = Employee("Manni", 5000)
```

访问属性

您可以使用点(.)来访问对象的属性。使用如下类的名称访问类变量：

```
emp1.displayEmployee()
```

```
emp2.displayEmployee()
```

```
print "Total Employee %d" % Employee.empCount
```

完整实例：

```
#coding=utf-8
```

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

```
class Employee:
```

'所有员工的基类'

```
empCount = 0
```

```
def __init__(self, name, salary):
```

```
    self.name = name
```

```
    self.salary = salary
```

```
    Employee.empCount += 1
```

```
def displayCount(self):
```

```
    print "Total Employee %d" % Employee.empCount
```

```
def displayEmployee(self):
```

```
    print "Name : ", self.name, ", Salary: ", self.salary
```

"创建 Employee 类的第一个对象"

```
emp1 = Employee("Zara", 2000)
```

"创建 Employee 类的第二个对象"

```
emp2 = Employee("Manni", 5000)
```

```
emp1.displayEmployee()
```

```
emp2.displayEmployee()
```

```
print "Total Employee %d" % Employee.empCount
```

执行以上代码输出结果如下：

```
Name :   Zara ,Salary:   2000
```

```
Name : Manni ,Salary: 5000
```

```
Total Employee 2
```

你可以添加，删除，修改类的属性，如下所示：

```
empl.age = 7 # 添加一个 'age' 属性
```

```
empl.age = 8 # 修改 'age' 属性
```

```
del empl.age # 删除 'age' 属性
```

你也可以使用以下函数的方式来访问属性：

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

```
hasattr(empl, 'age') # 如果存在 'age' 属性返回 True。
```

```
getattr(empl, 'age') # 返回 'age' 属性的值
```

```
setattr(empl, 'age', 8) # 添加属性 'age' 值为 8
```

```
delattr(empl, 'age') # 删除属性 'age'
```

Python 内置类属性

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

Python 内置类属性调用实例如下：

```
#coding=utf-8
```

```
#!/usr/bin/python

class Employee:

    '所有员工的基类'

    empCount = 0

    def __init__(self, name, salary):

        self.name = name

        self.salary = salary

        Employee.empCount += 1

    def displayCount(self):

        print "Total Employee %d" % Employee.empCount

    def displayEmployee(self):

        print "Name : ", self.name, " , Salary: ", self.salary

print "Employee.__doc__:", Employee.__doc__

print "Employee.__name__:", Employee.__name__

print "Employee.__module__:", Employee.__module__

print "Employee.__bases__:", Employee.__bases__

print "Employee.__dict__:", Employee.__dict__
```

执行以上代码输出结果如下：

```
Employee.__doc__: Common base class for all employees

Employee.__name__: Employee

Employee.__module__: __main__

Employee.__bases__: ()

Employee.__dict__: {'__module__': '__main__', 'displayCount':
<function displayCount at 0xb7c84994>, 'empCount': 2,
'displayEmployee': <function displayEmployee at 0xb7c8441c>,
'__doc__': 'Common base class for all employees',
'__init__': <function __init__ at 0xb7c846bc>}
```

python 对象销毁(垃圾回收)

同 Java 语言一样，Python 使用了引用计数这一简单技术来追踪内存中的对象。

在 Python 内部记录着所有使用中的对象各有多少引用。

一个内部跟踪变量，称为一个引用计数器。

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

```
a = 40      #创建对象  <40>

b = a      #增加引用，  <40> 的计数

c = [b]    #增加引用。  <40> 的计数


del a      #减少引用 <40> 的计数

b = 100    #减少引用 <40> 的计数
```



```
c[0] = -1    #减少引用 <40> 的计数
```

垃圾回收机制不仅针对引用计数为 0 的对象，同样也可以处理循环引用的情况。循环引用指的是，两个对象相互引用，但是没有其他变量引用他们。这种情况下，仅使用引用计数是不够的。Python 的垃圾收集器实际上是一个引用计数器和一个循环垃圾收集器。作为引用计数的补充，垃圾收集器也会留心被分配的总量很大（及未通过引用计数销毁的那些）的对象。在这种情况下，解释器会暂停下来，试图清理所有未引用的循环。

实例

析构函数 `__del__`，`__del__` 在对象消逝的时候被调用，当对象不再被使用时，`__del__` 方法运行：

```
#coding=utf-8

#!/usr/bin/python

class Point:

    def __init__( self, x=0, y=0 ):

        self.x = x

        self.y = y

    def __del__(self):

        class_name = self.__class__.__name__

        print class_name, "destroyed"

pt1 = Point()

pt2 = pt1

pt3 = pt1

print id(pt1), id(pt2), id(pt3) # 打印对象的 id

del pt1
```

```
del pt2
```

```
del pt3
```

以上实例运行结果如下：

```
3083401324 3083401324 3083401324
```

```
Point destroyed
```

注意：通常你需要在单独的文件中定义一个类，

类的继承

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

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

在 python 中继承中的一些特点：

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

如果在继承元组中列了一个以上的类，那么它就被称作“多重继承”。

语法：

派生类的声明，与他们的父类类似，继承的基类列表跟在类名之后，如下所示：

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

实例：

```
#coding=utf-8

#!/usr/bin/python


class Parent:          #定义父类

    parentAttr = 100

    def __init__(self):

        print 调用父类构造函数"

    def parentMethod(self):

        print 调用父类方法'

    def setAttr(self, attr):

        Parent.parentAttr = attr

    def getAttr(self):

        print 父类属性 :", Parent.parentAttr


class Child(Parent): # 定义子类

    def __init__(self):

        print 调用子类构造方法"

    def childMethod(self):

        print 调用子类方法 child method'
```

```
c = Child()          实例化子类

c.childMethod()      #调用子类的方法

c.parentMethod()     #调用父类方法

c.setAttr(200)       #再次调用父类的方法

c.getAttr()          再次调用父类的方法
```

以上代码执行结果如下：

```
调用子类构造方法

调用子类方法  child method

调用父类方法

父类属性  ： 200
```

你可以继承多个类

```
class A:             定义类 A

.....

class B:             定义类 B

.....

class C(A, B):       #继承类 A 和 B

.....
```

你可以使用 `issubclass` 或者 `isinstance` 方法来检测。

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

`issubclass(sub, sup)`

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

方法重写

如果你的父类方法的功能不能满足你的需求，你可以在子类重写你父类的方法：

实例：

```
#coding=utf-8

#!/usr/bin/python

class Parent:      #定义父类

    def myMethod(self):

        print '调用父类方法'

class Child(Parent): # 定义子类

    def myMethod(self):

        print '调用子类方法'

c = Child()         #子类实例

c.myMethod()        #子类调用重写方法
```

执行以上代码输出结果如下：

调用子类方法

基础重载方法

下表列出了一些通用的功能，你可以在自己的类重写：

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

运算符重载

Python 同样支持运算符重载，实例如下：

```
#!/usr/bin/python

class Vector:

    def __init__(self, a, b):

        self.a = a

        self.b = b
```

```
def __str__(self):  
  
    return 'Vector (%d, %d)' % (self.a, self.b)  
  
def __add__(self, other):  
  
    return Vector(self.a + other.a, self.b + other.b)  
  
v1 = Vector(2,10)  
  
v2 = Vector(5,-2)  
  
print v1 + v2
```

以上代码执行结果如下所示：

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

类属性与方法

类的私有属性

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

类的方法

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

类的私有方法

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

实例

```
#coding=utf-8

#!/usr/bin/python

class JustCounter:

    __secretCount = 0  # 私有变量

    publicCount = 0    # 公开变量

    def count(self):

        self.__secretCount += 1

        self.publicCount += 1

        print self.__secretCount

counter = JustCounter()

counter.count()

counter.count()

print counter.publicCount

print counter.__secretCount  # 报错，实例不能访问私有变量
```

Python 通过改变名称来包含类名：

1

2

2


```
Traceback (most recent call last):

  File "test.py", line 17, in <module>

    print counter.__secretCount    #报错，实例不能访问私有变量

AttributeError: JustCounter instance has no attribute '__secretCount'
```

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

```
.....

print counter._JustCounter__secretCount
```

执行以上代码，执行结果如下：

```
1

2

2

2
```

Python 正则表达式

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

`re` 模块使 Python 语言拥有全部的正则表达式功能。

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

`re` 模块也提供了与这些方法功能完全一致的函数，这些函数使用一个模式字符串做为它们的第一个参数。

本章节主要介绍 Python 中常用的正则表达式处理函数。

re.match 函数

re.match 尝试从字符串的开始匹配一个模式。

函数语法：

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

函数参数说明：

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

匹配成功 re.match 方法返回一个匹配的对象， 否则返回 None 。

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

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

实例：

```
#!/usr/bin/python

import re

line = "Cats are smarter than dogs"

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

```
if matchObj:

    print "matchObj.group() : ", matchObj.group()

    print "matchObj.group(1) : ", matchObj.group(1)

    print "matchObj.group(2) : ", matchObj.group(2)

else:

    print "No match!!"
```

以上实例执行结果如下：

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

matchObj.group(1) :  Cats

matchObj.group(2) :  smarter
```

re. search 方法

re.match 尝试从字符串的开始匹配一个模式。

函数语法：

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

函数参数说明：

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

匹配成功 re. search方法方法返回一个匹配的对象，否则返回 None 。

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

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

实例：

```
#!/usr/bin/python

import re

line = "Cats are smarter than dogs";

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

if matchObj:
    print "matchObj.group() : ", matchObj.group()

    print "matchObj.group(1) : ", matchObj.group(1)

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

以上实例执行结果如下：

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

matchObj.group(1) :  Cats

matchObj.group(2) :  smarter
```

re.match 与 re.search 的区别

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

实例:

```
#!/usr/bin/python

import re

line = "Cats are smarter than dogs";

matchObj = re.match( r'dogs', line, re.M|re.I)

if matchObj:

    print "match --> matchObj.group() : ", matchObj.group()

else:

    print "No match!!"


matchObj = re.search( r'dogs', line, re.M|re.I)

if matchObj:

    print "search --> matchObj.group() : ", matchObj.group()

else:

    print "No match!!"
```

以上实例运行结果如下:

```
No match!!
```

```
search --> matchObj.group() : dogs
```

检索和替换

Python 的 re 模块提供了 re.sub 用于替换字符串中的匹配项。

语法：

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

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

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

实例：

```
#!/usr/bin/python

import re

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

# Delete Python-style comments

num = re.sub(r'#.*$', "", phone)

print "Phone Num : ", num

# Remove anything other than digits

num = re.sub(r'\D', "", phone)

print "Phone Num : ", num
```

以上实例执行结果如下：

Phone Num : 2004-959-559

Phone Num : 2004959559

正则表达式修饰符 – 可选标志

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

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

正则表达式模式

模式字符串使用特殊的语法来表示一个正则表达式：

字母和数字表示他们自身。一个正则表达式模式中的字母和数字匹配同样的字符串。

多数字母和数字前加一个反斜杠时会拥有不同的含义。

标点符号只有被转义时才匹配自身，否则它们表示特殊的含义。

反斜杠本身需要使用反斜杠转义。

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

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

模式	描述
<code>^</code>	匹配字符串的开头
<code>\$</code>	匹配字符串的末尾。
<code>.</code>	匹配任意字符，除了换行符，当 <code>re.DOTALL</code> 标记被指定时，则可以匹配包括换行符
<code>[...]</code>	用来表示一组字符,单独列出： <code>[amk]</code> 匹配 'a'，'m'或'k'
<code>[^...]</code>	不在[]中的字符： <code>[^abc]</code> 匹配除了 a, b, c之外的字符。
<code>re*</code>	匹配 0 个或多个的表达式。
<code>re+</code>	匹配 1 个或多个的表达式。
<code>re?</code>	匹配 0 个或 1 个由前面的正则表达式定义的片段，贪婪方式
<code>re{ n}</code>	
<code>re{ n, }</code>	精确匹配 n 个前面表达式。
<code>re{ n, m}</code>	匹配 n 到 m 次由前面的正则表达式定义的片段，贪婪方式
<code>a b</code>	匹配 a 或 b
<code>(re)</code>	G 匹配括号内的表达式，也表示一个组
<code>(?imx)</code>	正则表达式包含三种可选标志：i, m,或 x 。只影响括号中的区域。
<code>(?-imx)</code>	正则表达式关闭 i, m,或 x 可选标志。只影响括号中的区域。
<code>(?: re)</code>	类似 (...)但是不表示一个组
<code>(?imx: re)</code>	在括号中使用 i, m,或 x 可选标志
<code>(?-imx: re)</code>	在括号中不使用 i, m,或 x 可选标志
<code>(?#...)</code>	注释.
<code>(?= re)</code>	前向肯定界定符。如果所含正则表达式，以 ...表示，在当前位置成功匹配时成功 所含表达式已经尝试，匹配引擎根本没有提高；模式的剩余部分还要尝试界定符的
<code>(?! re)</code>	前向否定界定符。与肯定界定符相反；当所含表达式不能在字符串当前位置匹配时

(?> re)	匹配的独立模式，省去回溯。
\w	匹配字母数字
\W	匹配非字母数字
\s	匹配任意空白字符，等价于 [\t\n\r\f].
\S	匹配任意非空字符
\d	匹配任意数字，等价于 [0-9].
\D	匹配任意非数字
\A	匹配字符串开始
\Z	匹配字符串结束，如果是存在换行，只匹配到换行前的结束字符串。c
\z	匹配字符串结束
\G	匹配最后匹配完成的位置。
\b	匹配一个单词边界，也就是指单词和空格间的位置。例如， ’er\b’可以匹配”never”中出现的第一个 ’e’，但不会匹配 ”verb”中的 ’er’。
\B	匹配非单词边界。’er\B’能匹配 ”verb”中的 ’er’，但不能匹配 ”never”中的 ’er’。
\n, \等.	匹配一个换行符。匹配一个制表符。等
\1... \9	比赛第 n 个分组的子表达式。
\10	匹配第 n 个分组的子表达式，如果它经匹配。否则指的是八进制字符码的表达式。

正则表达式实例

字符匹配

实例	描述
python	匹配 ”python”。

字符类

实例	描述
[Pp]ython	匹配 “Python” 或 “python”
rub[ye]	匹配 “ruby” 或 “rube”
[aeiou]	匹配中括号内的任意一个字母
[0-9]	匹配任何数字。类似于 [0123456789]
[a-z]	匹配任何小写字母
[A-Z]	匹配任何大写字母
[a-zA-Z0-9]	匹配任何字母及数字
[^aeiou]	除了 aeiou 字母以外的所有字符
[^0-9]	匹配除了数字外的字符

特殊字符类

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

Python CGI 编程

什么是 CGI

CGI 目前由 NCSA 维护，NCSA 定义 CGI 如下：

CGI(Common Gateway Interface), 通用网关接口,它是一段程序,运行在服务器上如：HTTP 服务器，提供同客户端 HTML 页面的接口。

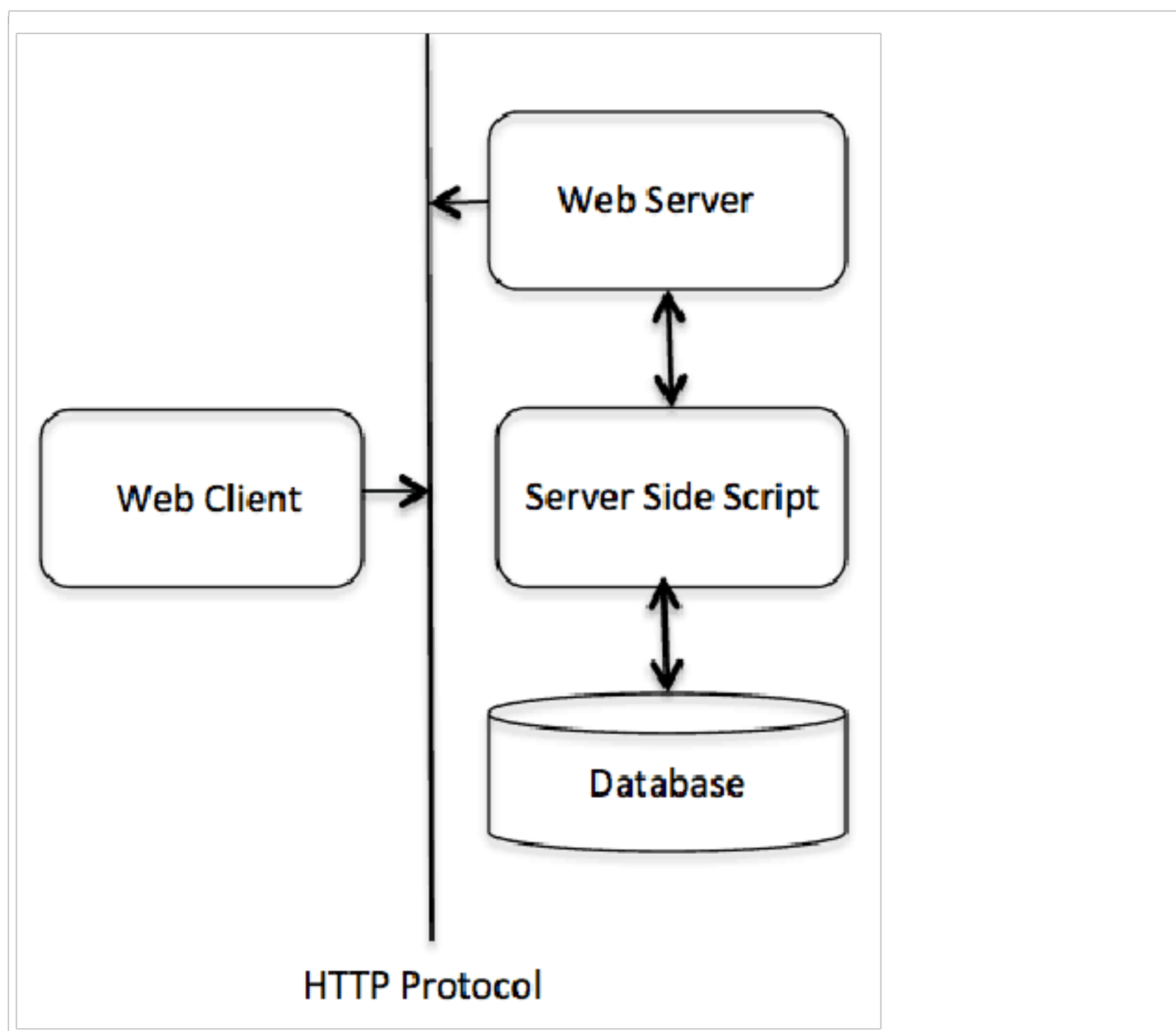
网页浏览

为了更好的了解 CGI 是如何工作的，我们可以从在网页上点击一个链接或 URL 的流程：

- 1、使用你的浏览器访问 URL 并连接到 HTTP web 服务器。
- 2、Web 服务器接收到请求信息后会解析 URL，并查找访问的文件在服务器上是否存在，如果存在返回文件的内容，否则返回错误信息。
- 3、浏览器从服务器上接收信息，并显示接收的文件或者错误信息。

CGI 程序可以是 Python 脚本，PERL 脚本，SHELL 脚本，C 或者 C++ 程序等。

CGI 架构图



Web 服务器支持及配置

在你进行 CGI 编程前，确保您的 Web 服务器支持 CGI 及已经配置了 CGI 的处理程序。

所有的 HTTP 服务器执行 CGI 程序都保存在一个预先配置的目录。这个目录被称为 CGI 目录，并按照惯例，它被命名为 `/var/www/cgi-bin` 目录。

CGI 文件的扩展名为 `.cgi`，python 也可以使用 `.py` 扩展名。

默认情况下，Linux 服务器配置运行的 `cgi-bin` 目录中为 `/var/www`。

如果你想指定其他运行 CGI 脚本的目录，可以修改 `httpd.conf` 配置文件，如下所示：

```
<Directory "/var/www/cgi-bin">

    AllowOverride None

    Options ExecCGI
```

```
Order allow,deny

Allow from all

</Directory>


<Directory "/var/www/cgi-bin">

Options All

</Directory>
```

第一个 CGI 程序

我们使用 Python 创建第一个 CGI 程序，文件名为 `hellp.py` 文件位于 `/var/www/cgi-bin` 目录中，内容如下，修改文件的权限为 755：

```
#coding=utf-8

#!/usr/bin/python


print "Content-type:text/html\r\n\r\n"

print '<html>'

print '<head>'

print '<title>Hello Word - First CGI Program</title>'

print '</head>'

print '<body>'

print '<h2>Hello Word! This is my first CGI program</h2>'

print '</body>'

print '</html>'
```

以上程序在浏览器访问显示结果如下：

```
Hello Word! This is my first CGI program
```

这个的 hello.py脚本是一个简单的 Python 脚本，脚本第一的输出内容
"Content-type:text/html\r\n"发送到浏览器并告知浏览器显示的内容类型为"text/html"

HTTP 头部

hello.py文件内容中的" Content-type:text/html\r\n"即为HTTP 头部的一部分，它会发送给浏览器告诉浏览器文件的内容类型。
HTTP 头部的格式如下：

HTTP 字段名： 字段内容

例如

```
Content-type: text/html\r\n\r\n
```

以下表格介绍了 CGI 程序中 HTTP 头部经常使用的信息：

头	描述
Content-type:	请求的与实体对应的 MIME 信息。例如：Content-type:text/html
Expires: Date	响应过期的日期和时间
Location: URL	用来重定向接收方到非请求 URL 的位置来完成请求或标识新的资源
Last-modified: Date	请求资源的最后修改时间
Content-length: N	请求的内容长度
Set-Cookie: String	设置 Http Cookie

CGI 环境变量

所有的 CGI 程序都接收以下的环境变量，这些变量在 CGI 程序中发挥了重要的作用：

变量名	描述
CONTENT_TYPE	这个环境变量的值指示所传递来的信息的 MIME 类型。目前，环境变量 CONTENT_TYPE 都是：application/x-www-form-urlencoded。他表示数据来自于 HTML 表单。
CONTENT_LENGTH	如果服务器与 CGI 程序信息的传递方式是 POST ，这个环境变量即使从标准输入中读到的有效数据的字节数。这个环境变量在读取所输入的数据时必须使用。
HTTP_COOKIE	客户机内的 COOKIE 内容。
HTTP_USER_AGENT	提供包含了版本数或其他专有数据的客户浏览器信息。
PATH_INFO	这个环境变量的值表示紧接在 CGI 程序名之后的其他路径信息。它常常作为 CGI 程序名的一部分。
QUERY_STRING	如果服务器与 CGI 程序信息的传递方式是 GET ，这个环境变量的值即使所有跟在 CGI 程序名的后面，两者中间用一个问号’?’分隔。
REMOTE_ADDR	这个环境变量的值是发送请求的客户机的 IP 地址，例如上面的 192. 168. 1. 67。而且它是 Web 客户机需要提供给 Web 服务器的唯一标识，可以在 CGI 程序中通过 getenv(“REMOTE_ADDR”) 得到 Web 客户机。
REMOTE_HOST	这个环境变量的值包含发送 CGI 请求的客户机的主机名。如果不支持你想查看这个环境变量。
REQUEST_METHOD	提供脚本被调用的方法。对于使用 HTTP/1.0 协议的脚本，仅 GET 和 POST 是有效的。
SCRIPT_FILENAME	CGI 脚本的完整路径
SCRIPT_NAME	CGI 脚本的的名称
SERVER_NAME	这是你的 WEB 服务器的主机名、别名或 IP 地址。
SERVER_SOFTWARE	这个环境变量的值包含了调用 CGI 程序的 HTTP 服务器的名称和版本号。例如：Apache/2. 2. 14(Unix)

以下是一个简单的 CGI 脚本输出 CGI 的环境变量：

```
#coding=utf-8

#!/usr/bin/python

import os
```

```
print "Content-type: text/html\r\n\r\n";

print "<font size=+1>Environment</font><\br>";

for param in os.environ.keys():

    print "<b>%20s</b>: %s<\br>" % (param, os.environ[param])
```

GET 和 POST 方法

浏览器客户端通过两种方法向服务器传递信息，这两种方法就是 GET 方法和 POST 方法。

使用 GET 方法传输数据

GET 方法发送编码后的用户信息到服务端，数据信息包含在请求页面的 URL 上，以“?”号分割，如下所示：

```
http://www.test.com/cgi-bin/hello.py?key1=value1&key2=value2
```

有关 GET 请求的其他一些注释：

- GET 请求可被缓存
- GET 请求保留在浏览器历史记录中
- GET 请求可被收藏为书签
- GET 请求不应在处理敏感数据时使用
- GET 请求有长度限制
- GET 请求只应当用于取回数据

简单的 url 实例：GET 方法

以下是一个简单的 URL ，使用 GET 方法向 hello_get.py 程序发送两个参数：

```
/cgi-bin/hello_get.py?first_name=ZARA&last_name=ALI
```

以下为 hello_get.py 文件的代码：

```
#coding=utf-8

#!/usr/bin/python


# CGI 处理模块

import cgi, cgitb


# 创建 FieldStorage 的实例化

form = cgi.FieldStorage()


# 获取数据

first_name = form.getvalue('first_name')

last_name = form.getvalue('last_name')


print "Content-type:text/html\r\n\r\n"

print "<html>"

print "<head>"

print "<title>Hello - Second CGI Program</title>"

print "</head>"

print "<body>"

print "<h2>Hello %s %s</h2>" % (first_name, last_name)

print "</body>"

print "</html>"
```

浏览器请求输出结果：

```
Hello ZARA ALI
```

简单的表单实例：GET 方法

以下是一个通过 HTML 的表单使用 GET 方法向服务器发送两个数据，提交的服务器脚本同样是 `hello_get.py` 文件，代码如下：

```
<form action="/cgi-bin/hello_get.py" method="get">

First Name: <input type="text" name="first_name"> <br />

Last Name: <input type="text" name="last_name" />

<input type="submit" value="Submit" />

</form>
```

使用 POST 方法传递数据

使用 POST 方法向服务器传递数据是更安全可靠的，像一些敏感信息如用户密码等需要使用 POST 传输数据。

以下同样是 `hello_get.py`，它也可以处理浏览器提交的 POST 表单数据：

```
#coding=utf-8

#!/usr/bin/python

# 引入 CGI 模块

import cgi, cgitb

# 创建 FieldStorage 实例
```

```
form = cgi.FieldStorage()

# 获取表单数据

first_name = form.getvalue('first_name')

last_name = form.getvalue('last_name')


print "Content-type:text/html\r\n\r\n"

print "<html>"

print "<head>"

print "<title>Hello - Second CGI Program</title>"

print "</head>"

print "<body>"

print "<h2>Hello %s %s</h2>" % (first_name, last_name)

print "</body>"

print "</html>"
```

以下为表单通过 POST 方法向服务器脚本 hello_get.py 提交数据：

```
<form action="/cgi-bin/hello_get.py" method="post">

First Name: <input type="text" name="first_name"><br />

Last Name: <input type="text" name="last_name" />


<input type="submit" value="Submit" />

</form>
```

通过 CGI 程序传递 checkbox 数据

checkbox 用于提交一个或者多个选项数据，HTML 代码如下：

```
<form action="/cgi-bin/checkbox.cgi" method="POST" target="_blank">

<input type="checkbox" name="maths" value="on" /> Maths

<input type="checkbox" name="physics" value="on" /> Physics

<input type="submit" value="Select Subject" />

</form>
```

以下为 checkbox.cgi 文件的代码：

```
#coding=utf-8

#!/usr/bin/python

# 引入 CGI 处理模块

import cgi, cgitb

# 创建 FieldStorage 的实例

form = cgi.FieldStorage()

# 接收字段数据

if form.getvalue('maths'):

    math_flag = "ON"

else:

    math_flag = "OFF"
```

```
if form.getvalue('physics'):

    physics_flag = "ON"

else:

    physics_flag = "OFF"


print "Content-type:text/html\r\n\r\n"

print "<html>"

print "<head>"

print "<title>Checkbox - Third CGI Program</title>"

print "</head>"

print "<body>"

print "<h2> CheckBox Maths is : %s</h2>" % math_flag

print "<h2> CheckBox Physics is : %s</h2>" % physics_flag

print "</body>"

print "</html>"
```

通过 CGI 程序传递 Radio 数据

Radio 只向服务器传递一个数据，HTML 代码如下：

```
<form action="/cgi-bin/radiobutton.py" method="post" target="_blank">

<input type="radio" name="subject" value="maths" /> Maths

<input type="radio" name="subject" value="physics" /> Physics

<input type="submit" value="Select Subject" />
```

```
</form>
```

```
radiobutton.py脚本代码如下:
```

```
#coding=utf-8

#!/usr/bin/python


# Import modules for CGI handling

import cgi, cgitb


# Create instance of FieldStorage

form = cgi.FieldStorage()


# Get data from fields

if form.getvalue('subject'):

    subject = form.getvalue('subject')

else:

    subject = "Not set"


print "Content-type:text/html\r\n\r\n"

print "<html>"

print "<head>"

print "<title>Radio - Fourth CGI Program</title>"

print "</head>"

print "<body>"
```

```
print "<h2> Selected Subject is %s</h2>" % subject

print "</body>"

print "</html>"
```

通过 CGI 程序传递 Textarea 数据

Textarea 向服务器传递多行数据，HTML 代码如下：

```
<form action="/cgi-bin/textarea.py" method="post" target="_blank">

<textarea name="textcontent" cols="40" rows="4">

Type your text here...

</textarea>

<input type="submit" value="Submit" />

</form>
```

textarea.cgi 脚本代码如下：

```
#coding=utf-8

#!/usr/bin/python


# Import modules for CGI handling

import cgi, cgitb


# Create instance of FieldStorage

form = cgi.FieldStorage()


# Get data from fields
```

```
if form.getvalue('textcontent'):

    text_content = form.getvalue('textcontent')

else:

    text_content = "Not entered"


print "Content-type:text/html\r\n\r\n"

print "<html>"

print "<head>";

print "<title>Text Area - Fifth CGI Program</title>"

print "</head>"

print "<body>"

print "<h2> Entered Text Content is %s</h2>" % text_content

print "</body>"
```

通过 CGI 程序传递下拉数据

HTML 下拉框代码如下：

```
<form action="/cgi-bin/dropdown.py" method="post" target="_blank">

<select name="dropdown">

<option value="Maths" selected>Maths</option>

<option value="Physics">Physics</option>

</select>

<input type="submit" value="Submit"/>

</form>
```

dropdown.py 脚本代码如下所示：

```
#coding=utf-8

#!/usr/bin/python


# Import modules for CGI handling

import cgi, cgitb


# Create instance of FieldStorage

form = cgi.FieldStorage()


# Get data from fields

if form.getvalue('dropdown'):

    subject = form.getvalue('dropdown')

else:

    subject = "Not entered"


print "Content-type:text/html\r\n\r\n"

print "<html>"

print "<head>"

print "<title>Dropdown Box - Sixth CGI Program</title>"

print "</head>"

print "<body>"

print "<h2> Selected Subject is %s</h2>" % subject
```

```
print "</body>"

print "</html>"
```

CGI 中使用 Cookie

在 http 协议一个很大的缺点就是不作用户身份的判断，这样给编程人员带来很大的不便，而 cookie 功能的出现弥补了这个缺憾。

所有 cookie 就是在客户访问脚本的同时，通过客户的浏览器，在客户硬盘上写入纪录数据，当下次客户访问脚本时取回数据信息，从而达到身份判别的功能，cookie 常用在密码判断中。

cookie 的语法

http cookie 的发送是通过 http 头部来实现的，他早于文件的传递，头部 set-cookie 的语法如下：

```
Set-cookie:name=name;expires=date;path=path;domain=domain;secure
```

- name=name: 需要设置 cookie 的值 (name 不能使用";"和",,"号), 有多个 name 值时用";"分隔例如: name1=name1;name2=name2;name3=name3。
- expires=date: cookie 的有效期限, 格式: expires="Wdy, DD-Mon-YYYY HH:MM:SS"
- path=path: 设置 cookie 支持的路径, 如果 path 是一个路径, 则 cookie 对这个目录下的所有文件及子目录生效, 例如: path="/cgi-bin/" 如果 path 是一个文件, 则 cookie 指对这个文件生效, 例如: path="/cgi-bin/cookie.cgi"
- domain=domain: 对 cookie 生效的域名, 例如: domain="www.chinalb.com"
- secure: 如果给出此标志, 表示 cookie 只能通过 SSL 协议的 https 服务器来传递。
- cookie 的接收是通过设置环境变量 HTTP_COOKIE 来实现的, CGI 程序可以通过检索该变量获取 cookie 信息。

Cookie 设置

Cookie 的设置非常简单, cookie 会在 http 头部单独发送。以下实例在 cookie 中设置了 UserID 和 Password :

```
<pre>

#coding=utf-8

#!/usr/bin/python


print "Set-Cookie:UserID=XYZ;\r\n"

print "Set-Cookie:Password=XYZ123;\r\n"

print "Set-Cookie:Expires=Tuesday, 31-Dec-2007 23:12:40 GMT";\r\n"

print "Set-Cookie:Domain=www.w3cschool.cc;\r\n"

print "Set-Cookie:Path=/perl;\r\n"

print "Content-type:text/html\r\n\r\n"

.....Rest of the HTML Content.....
```

以上实例使用了 Set-Cookie 头信息来设置 Cookie 信息,可选项中设置了 Cookie 的其他属性,如过期时间 Expires, 域名 Domain , 路径 Path。这些信息设置在 "Content-type:text/html\r\n\r\n" 之前。

检索 Cookie 信息

Cookie 信息检索页非常简单, Cookie 信息存储在 CGI 的环境变量 HTTP_COOKIE 中, 存储格式如下:

```
key1=value1;key2=value2;key3=value3....
```

以下是一个简单的 CGI 检索 cookie 信息的程序:

```
#coding=utf-8

#!/usr/bin/python
```

```
# Import modules for CGI handling

from os import environ

import cgi, cgitb

if environ.has_key('HTTP_COOKIE'):

    for cookie in map(strip, split(environ['HTTP_COOKIE'], ';')):

        (key, value ) = split(cookie, '=');

        if key == "UserID":

            user_id = value

        if key == "Password":

            password = value

print "User ID  = %s" % user_id

print "Password = %s" % password
```

以上脚本输出结果如下：

```
User ID = XYZ

Password = XYZ123
```

文件上传实例：

HTML 设置上传文件的表单需要设置 enctype 属性为 multipart/form-data代码如下所示：

```
<html>

<body>
```

```
<form enctype="multipart/form-data"

        action="save_file.py" method="post">

<p>File: <input type="file" name="filename" /></p>

<p><input type="submit" value="Upload" /></p>

</form>

</body>

</html>
```

save_file.py脚本文件代码如下：

```
#coding=utf-8

#!/usr/bin/python


import cgi, os

import cgitb; cgitb.enable()


form = cgi.FieldStorage()


# 获取文件名

fileitem = form['filename']


# 检测文件是否上传

if fileitem.filename:

    # 设置文件路径

    fn = os.path.basename(fileitem.filename)
```

```
open('/tmp/' + fn, 'wb').write(fileitem.file.read())

message = 'The file "' + fn + '" was uploaded successfully'

else:

    message = 'No file was uploaded'

print """\n
Content-Type: text/html\n

<html>

<body>

    <p>%s</p>

</body>

</html>

""" % (message,)
```

如果你使用的系统是 Unix/Linux，你必须替换文件分隔符，在 window 下只需要使用 open() 语句即可：

```
fn = os.path.basename(fileitem.filename.replace("\\", "/" ))
```

文件下载对话框

如果我们需要为用户提供文件下载链接，并在用户点击链接后弹出文件下载对话框，我们通过设置 HTTP 头信息来实现这些功能，功能代码如下：

```
#coding=utf-8
```

```
#!/usr/bin/python

# HTTP Header

print "Content-Type:application/octet-stream; name=\"FileName\"\\r\\n";

print "Content-Disposition: attachment; filename=\"FileName\"\\r\\n\\n";


# Actual File Content will go hear.

fo = open("foo.txt", "rb")


str = fo.read();

print str


# Close opened file

fo.close()
```

python 操作 mysql 数据库

Python 标准数据库接口为 Python DB-API，Python DB-API 为开发人员提供了数据库应用编程接口。

Python 数据库接口支持非常多的数据库，你可以选择适合你项目的数据库：

- GadFly
- mSQL
- MySQL
- PostgreSQL
- Microsoft SQL Server 2000
- Informix
- Interbase
- Oracle
- Sybase

你可以访问 [Python 数据库接口及 API](#) 查看详细的支持数据库列表。

不同的数据库你需要下载不同的 DB API 模块，例如你需要访问 Oracle 数据库和 Mysql 数据，你需要下载 Oracle 和 MySQL 数据库模块。

DB-API 是一个规范。它定义了一系列必须的对象和数据存取方式，以便为各种各样的底层数据库系统和多种多样的数据库接口程序提供一致的访问接口。

Python 的 DB-API，为大多数的数据库实现了接口，使用它连接各数据库后，就可以用相同的方式操作各数据库。

Python DB-API 使用流程：

- 引入 API 模块。
- 获取与数据库的连接。
- 执行 SQL 语句和存储过程。
- 关闭数据库连接。

什么是 MySQLdb？

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

如何安装 MySQLdb？

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

```
# encoding: utf-8

#!/usr/bin/python

import MySQLdb
```

如果执行后的输出结果如下所示，意味着你没有安装 MySQLdb 模块：

```
Traceback (most recent call last):
```



```
File "test.py", line 3, in <module>
```

```
import MySQLdb
```

```
ImportError: No module named MySQLdb
```

安装 MySQLdb ，请访问 <http://sourceforge.net/projects/mysql-python>Linux 平台可以访问：
<https://pypi.python.org/pypi/MySQL-python> 从这里可选择适合您的平台的安装包，分为预编译的二进制文件和源代码安装包。

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

```
$ gunzip MySQL-python-1.2.2.tar.gz
```

```
$ tar -xvf MySQL-python-1.2.2.tar
```

```
$ cd MySQL-python-1.2.2
```

```
$ python setup.py build
```

```
$ python setup.py install
```

注意：请确保您有 root 权限来安装上述模块。

数据库连接

连接数据库前，请先确认以下事项：

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

实例：

以下实例链接 Mysql 的 TESTDB 数据库：

```
# encoding: utf-8

#!/usr/bin/python


import MySQLdb


# 打开数据库连接

db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )


# 使用 cursor() 方法获取操作游标

cursor = db.cursor()


# 使用 execute 方法执行 SQL 语句

cursor.execute("SELECT VERSION() ")


# 使用 fetchone() 方法获取一条数据库。

data = cursor.fetchone()


print "Database version : %s " % data


# 关闭数据库连接

db.close()
```

执行以上脚本输出结果如下：

```
Database version : 5.0.45
```

创建数据库表

如果数据库连接存在我们可以使用 `execute()`方法来为数据库创建表，如下所示创建表 `EMPLOYEE` ：

```
# encoding: utf-8

#!/usr/bin/python

import MySQLdb

# 打开数据库连接

db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# 使用 cursor() 方法获取操作游标

cursor = db.cursor()

# 如果数据表已经存在使用 execute() 方法删除表。

cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")

# 创建数据表 SQL 语句

sql = """CREATE TABLE EMPLOYEE (

            FIRST_NAME  CHAR(20) NOT NULL,

            LAST_NAME   CHAR(20),

            AGE  INT,
```

```
SEX CHAR(1),  
  
INCOME FLOAT )"""
```

```
cursor.execute(sql)
```

```
# 关闭数据库连接
```

```
db.close()
```

数据库插入操作

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

```
# encoding: utf-8
```

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

```
import MySQLdb
```

```
# 打开数据库连接
```

```
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )
```

```
# 使用 cursor() 方法获取操作游标
```

```
cursor = db.cursor()
```

```
# SQL 插入语句
```

```
sql = """INSERT INTO EMPLOYEE(FIRST_NAME,
```

```
        LAST_NAME, AGE, SEX, INCOME)

        VALUES ('Mac', 'Mohan', 20, 'M', 2000)"""

try:

    # 执行 sql 语句

    cursor.execute(sql)

    # 提交到数据库执行

    db.commit()

except:

    # Rollback in case there is any error

    db.rollback()


# 关闭数据库连接

db.close()
```

以上例子也可以写成如下形式：

```
# encoding: utf-8

#!/usr/bin/python


import MySQLdb


# 打开数据库连接

db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )


# 使用 cursor() 方法获取操作游标
```

```

cursor = db.cursor()

# SQL 插入语句

sql = "INSERT INTO EMPLOYEE(FIRST_NAME, \

        LAST_NAME, AGE, SEX, INCOME) \

        VALUES ('%s', '%s', '%d', '%c', '%d' )" % \

        ('Mac', 'Mohan', 20, 'M', 2000)

try:

    # 执行 sql 语句

    cursor.execute(sql)

    # 提交到数据库执行

    db.commit()

except:

    # 发生错误时回滚

    db.rollback()

# 关闭数据库连接

db.close()

```

实例：

以下代码使用变量向 SQL 语句中传递参数：

```

.....

user_id = "test123"

password = "password"

```

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

            (user_id, password))

.....
```

数据库查询操作

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

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

实例：

查询 EMPLOYEE 表中 salary (工资) 字段大于 1000 的所有数据：

```
# encoding: utf-8

#!/usr/bin/python

import MySQLdb

# 打开数据库连接

db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# 使用 cursor() 方法获取操作游标

cursor = db.cursor()
```

```
# SQL 查询语句

sql = "SELECT * FROM EMPLOYEE \

      WHERE INCOME > '%d'" % (1000)

try:

    # 执行 SQL 语句

    cursor.execute(sql)

    # 获取所有记录列表

    results = cursor.fetchall()

    for row in results:

        fname = row[0]

        lname = row[1]

        age = row[2]

        sex = row[3]

        income = row[4]

        #打印结果

        print "fname=%s,lname=%s,age=%d,sex=%s,income=%d" % \

              (fname, lname, age, sex, income )

except:

    print "Error: unable to fecth data"


# 关闭数据库连接

db.close()
```

以上脚本执行结果如下：


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

数据库更新操作

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

```
# encoding: utf-8

#!/usr/bin/python

import MySQLdb

# 打开数据库连接

db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# 使用 cursor() 方法获取操作游标

cursor = db.cursor()

# SQL 更新语句

sql = "UPDATE EMPLOYEE SET AGE = AGE + 1

        WHERE SEX = '%c'" % ('M')

try:

    # 执行 SQL 语句

    cursor.execute(sql)

    # 提交到数据库执行
```

```
db.commit()

except:

    # 发生错误时回滚

    db.rollback()

# 关闭数据库连接

db.close()
```

执行事务

事务机制可以确保数据一致性。

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

- 原子性 (atomicity)。一个事务是一个不可分割的工作单位，事务中包括的诸操作要么都做，要么都不做。
- 一致性 (consistency)。事务必须是使数据库从一个一致性状态变到另一个一致性状态。一致性与原子性是密切相关的。
- 隔离性 (isolation)。一个事务的执行不能被其他事务干扰。即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。
- 持久性 (durability)。持续性也称永久性 (permanence)，指一个事务一旦提交，它对数据库中数据的改变就应该是永久性的。接下来的其他操作或故障不应该对其有任何影响。

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

实例：

```
# SQL 删除记录语句

sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (20)

try:

    # 执行 SQL 语句

    cursor.execute(sql)
```

```
        # 向数据库提交

        db.commit()

except:

        # 发生错误时回滚

        db.rollback()
```

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

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

错误处理

DB API 中定义了一些数据库操作的错误及异常，下表列出了这些错误和异常：

异常	描述
Warning	当有严重警告时触发，例如插入数据是被截断等等。必须是 StandardError 的子类。
Error	警告以外所有其他错误类。必须是 StandardError 的子类。
InterfaceError	当有数据库接口模块本身的错误（而不是数据库的错误）发生时触发。 必须是 Error的子类。
DatabaseError	和数据库有关的错误发生时触发。 必须是 Error的子类。
DataError	当有数据处理时的错误发生时触发，例如：除零错误，数据超范围等等。 必须是 DatabaseError的子类。
OperationalError	指非用户控制的，而是操作数据库时发生的错误。例如：连接意外断开、 数据库名未找到、分配错误等等操作数据库是发生的错误。 必须是 DatabaseError 的子类。
IntegrityError	完整性相关的错误，例如外键检查失败等。必须是 DatabaseError 子类。
InternalError	数据库的内部错误，例如游标（cursor）失效了、事务同步失败等等。 必须是 DatabaseError的子类。
ProgrammingError	程序错误，例如数据表（table）没找到或已存在、SQL 语句语法错误、 参数数量错误等等。必须是 DatabaseError 的子类。
NotSupportedError	不支持错误，指使用了数据库不支持的函数或 API 等。例如在连接对象上 使用.rollback()函数支持事务或者事务已关闭。 必须是 DatabaseError 的子类。

Python 使用 SMTP 发送邮件

SMTP (Simple Mail Transfer Protocol)即简单邮件传输协议,它是一组用于由源地址到目的地址传送邮件的规则，由它来控制信件的中转方式。

python 的 smtplib提供了一种很方便的途径发送电子邮件。它对 smtp 协议进行了简单的封装。

Python 创建 SMTP 对象语法如下：

```
import smtplib

smtpObj = smtplib.SMTP( [host [, port [, local_hostname]]] )
```

参数说明：

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

Python SMTP 对象使用 sendmail 方法发送邮件，语法如下：

```
SMTP.sendmail(from_addr, to_addrs, msg[, mail_options, rcpt_options])
```

参数说明：

- from_addr: 邮件发送者地址。
- to_addrs: 字符串列表，邮件发送地址。
- msg: 发送消息

这里要注意一下第三个参数，msg 是字符串，表示邮件。我们知道邮件一般由标题，发信人，收件人，邮件内容，附件等构成，发送邮件的时候，要注意 msg 的格式。这个格式就是 smtp 协议中定义的格式。

实例

以下是一个使用 Python 发送邮件简单的实例：

```
#!/usr/bin/python

import smtplib

sender=''

receivers=['']

message="""From:FromPerson<>

To:ToPerson<>

Subject: SMTP e-mail test


This is a test e-mail message.

"""

try:

    smtpObj = smtplib.SMTP('localhost')

    smtpObj.sendmail(sender, receivers, message)

    print "Successfully sent email"

except SMTPException:

    print "Error: unable to send email"
```

使用 Python 发送 HTML 格式的邮件

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

```
import smtplib

from email.mime.text import MIMEText

mailto_list=[""]

mail_host="smtp.XXX.com"  #设置服务器

mail_user="XXX"  #用户名

mail_pass="XXXX"  #口令

mail_postfix="XXX.com"  #发件箱的后缀


def send_mail(to_list,sub,content):  #to_list: 收件人; sub: 主题; content:
    邮件内容

    me="hello"+"<"+mail_user+"@"+mail_postfix+">"  这里的 hello 可以任意
    设置, 收到信后, 将按照设置显示

    msg = MIMEText(content,_subtype='html',_charset='gb2312')  创建
    一个实例, 这里设置为 html 格式邮件

    msg['Subject'] = sub  设置主题

    msg['From'] = me

    msg['To'] = ";".join(to_list)

    try:

        s = smtplib.SMTP()

        s.connect(mail_host)  连接 smtp 服务器

        s.login(mail_user,mail_pass)  登陆服务器

        s.sendmail(me, to_list, msg.as_string())  发送邮件

        s.close()
```

```
        return True

    except Exception, e:

        print str(e)

        return False

if __name__ == '__main__':

    if send_mail(mailto_list,"hello","<a href='http://www.cnblogs.com/xiaowuyi'>小五义</a>"):

        print 发送成功"

    else:

        print 发送失败"
```

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

```
#!/usr/bin/python

import smtplib

message="""From:FromPerson<>

To:ToPerson<>

MIME-Version: 1.0

Content-type: text/html

Subject: SMTP HTML e-mail test


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

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

<h1>This is headline.</h1>

"""

try:

    smtpObj = smtplib.SMTP('localhost')

    smtpObj.sendmail(sender, receivers, message)

    print "Successfully sent email"

except SMTPException:

    print "Error: unable to send email"
```

Python 发送带附件的邮件

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

```
from email.mime.text import MIMEText

from email.mime.multipart import MIMEMultipart

import smtplib


#创建一个带附件的实例

msg = MIMEMultipart()


#构造附件 1

att1 = MIMEText(open('d:\\123.rar', 'rb').read(), 'base64', 'gb2312')
```



```
att1["Content-Type"] = 'application/octet-stream'

att1["Content-Disposition"] = 'attachment; filename="123.doc"' #这里的 filename 可以任意写，写什么名字，邮件中显示什么名字

msg.attach(att1)


#构造附件 2

att2 = MIMEText(open('d:\\123.txt', 'rb').read(), 'base64', 'gb2312')

att2["Content-Type"] = 'application/octet-stream'

att2["Content-Disposition"] = 'attachment; filename="123.txt"'

msg.attach(att2)


#加邮件头

msg['to']=''

msg['from']=''

msg['subject'] = 'hello world'


#发送邮件

try:

    server = smtplib.SMTP()

    server.connect('smtp.XXX.com')

    server.login('XXX','XXXXX') #XXX为用户名，XXXXX 为密码

    server.sendmail(msg['from'], msg['to'],msg.as_string())

    server.quit()

    print '发送成功'

except Exception, e:
```

```
print str(e)
```

以下实例指定了 Content-type header 为 multipart/mixed 并发送/tmp/test.tx文本文件：

```
#!/usr/bin/python

import smtplib

import base64

filename = "/tmp/test.txt"

# 读取文件内容并使用 base64 编码

fo = open(filename, "rb")

filecontent = fo.read()

encodedcontent = base64.b64encode(filecontent) # base64

sender=''

reciever=''

marker = "AUNIQUEMARKER"

body ="""

This is a test email to send an attachement.

"""

# 定义头部信息
```

```
part1="""From:FromPerson<>

To:ToPerson<>

Subject: Sending Attachement

MIME-Version: 1.0

Content-Type: multipart/mixed; boundary=%s

--%s

""" % (marker, marker)


# 定义消息动作

part2 = """Content-Type: text/plain

Content-Transfer-Encoding:8bit


%s

--%s

""" % (body,marker)


# 定义附近部分

part3 = """Content-Type: multipart/mixed; name=\"%s\"

Content-Transfer-Encoding:base64

Content-Disposition: attachment; filename=%s


%s

--%s--

""" %(filename, filename, encodedcontent, marker)
```

```
message = part1 + part2 + part3

try:

    smtpObj = smtplib.SMTP('localhost')

    smtpObj.sendmail(sender, reciever, message)

    print "Successfully sent email"

except Exception:

    print "Error: unable to send email"
```

Python 多线程

多线程类似于同时执行多个不同程序，多线程运行有如下优点：

- 使用线程可以把占据长时间的程序中的任务放到后台去处理。
- 用户界面可以更加吸引人，这样比如用户点击了一个按钮去触发某些事件的处理，可以弹出一个进度条来显示处理的进度
- 程序的运行速度可能加快
- 在一些等待的任务实现上如用户输入、文件读写和网络收发数据等，线程就比较有用了。在这种情况下我们可以释放一些珍贵的资源如内存占用等等。

线程在执行过程中与进程还是有区别的。每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

每个线程都有他自己的一组 CPU 寄存器，称为线程的上下文，该上下文反映了线程上次运行该线程的 CPU 寄存器的状态。

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

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

开始学习 Python 线程

Python 中使用线程有两种方式：函数或者用类来包装线程对象。

函数式：调用 thread 模块中的 start_new_thread() 函数来产生新线程。语法如下：

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

参数说明：

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

实例：

```
#coding=utf-8

#!/usr/bin/python

import thread

import time

# 为线程定义一个函数

def print_time( threadName, delay):

    count = 0

    while count < 5:

        time.sleep(delay)

        count += 1

        print "%s: %s" % ( threadName, time.ctime(time.time()) )

# 创建两个线程

try:
```

```
        thread.start_new_thread( print_time, ("Thread-1", 2, ) )

        thread.start_new_thread( print_time, ("Thread-2", 4, ) )

except:

    print "Error: unable to start thread"

while 1:

    pass
```

执行以上程序输出结果如下：

```
Thread-1: Thu Jan 22 15:42:17 2009

Thread-1: Thu Jan 22 15:42:19 2009

Thread-2: Thu Jan 22 15:42:19 2009

Thread-1: Thu Jan 22 15:42:21 2009

Thread-2: Thu Jan 22 15:42:23 2009

Thread-1: Thu Jan 22 15:42:23 2009

Thread-1: Thu Jan 22 15:42:25 2009

Thread-2: Thu Jan 22 15:42:27 2009

Thread-2: Thu Jan 22 15:42:31 2009

Thread-2: Thu Jan 22 15:42:35 2009
```

线程的结束一般依靠线程函数的自然结束；也可以在线程函数中调用 `thread.exit()` 他抛出 `SystemExit exception` 达到退出线程的目的。

线程模块

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

`thread` 模块提供的其他方法：

- `threading.currentThread()` 返回当前的线程变量。
- `threading.enumerate()` 返回一个包含正在运行的线程的 `list` 正在运行指线程启动后、结束前，不包括启动前和终止后的线程。
- `threading.activeCount()` 返回正在运行的线程数量，与 `len(threading.enumerate())` 有相同的结果。

除了使用方法外，线程模块同样提供了 `Thread` 类来处理线程，`Thread` 类提供了以下方法：

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

使用 Threading 模块创建线程

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

```
#coding=utf-8

#!/usr/bin/python

import threading

import time

exitFlag = 0

class myThread (threading.Thread):    #继承父类 threading.Thread
```

```
def __init__(self, threadID, name, counter):

    threading.Thread.__init__(self)

    self.threadID = threadID

    self.name = name

    self.counter = counter

    def run(self):                                把要执行的代码写到 run 函数里面 线程在创建
    后会直接运行 run 函数

        print "Starting " + self.name

        print_time(self.name, self.counter, 5)

        print "Exiting " + self.name

def print_time(threadName, delay, counter):

    while counter:

        if exitFlag:

            thread.exit()

        time.sleep(delay)

        print "%s: %s" % (threadName, time.ctime(time.time()))

        counter -= 1

# 创建新线程

thread1 = myThread(1, "Thread-1", 1)

thread2 = myThread(2, "Thread-2", 2)

# 开启线程
```

```
thread1.start()  
  
thread2.start()  
  
print "Exiting Main Thread"
```

以上程序执行结果如下;

```
Starting Thread-1  
  
Starting Thread-2  
  
Exiting Main Thread  
  
Thread-1: Thu Mar 21 09:10:03 2013  
  
Thread-1: Thu Mar 21 09:10:04 2013  
  
Thread-2: Thu Mar 21 09:10:04 2013  
  
Thread-1: Thu Mar 21 09:10:05 2013  
  
Thread-1: Thu Mar 21 09:10:06 2013  
  
Thread-2: Thu Mar 21 09:10:06 2013  
  
Thread-1: Thu Mar 21 09:10:07 2013  
  
Exiting Thread-1  
  
Thread-2: Thu Mar 21 09:10:08 2013  
  
Thread-2: Thu Mar 21 09:10:10 2013  
  
Thread-2: Thu Mar 21 09:10:12 2013  
  
Exiting Thread-2
```

线程同步

如果多个线程共同对某个数据修改，则可能出现不可预料的结果，为了保证数据的正确性，需要对多个线程进行同步。

使用 Thread 对象的 Lock 和 Rlock 可以实现简单的线程同步，这两个对象都有 acquire方法和 release方法，对于那些需要每次只允许一个线程操作的数据，可以将其操作放到 acquire和 release方法之间。如下：

多线程的优势在于可以同时运行多个任务(至少感觉起来是这样)。但是当线程需要共享数据时，可能存在数据不同步的问题。

考虑这样一种情况：一个列表里所有元素都是 0，线程”set”从后向前把所有元素改成 1，而线程”prin负责从前往后读取列表并打印。

那么，可能线程”set开始改的时候，线程”prin便来打印列表了，输出就成了一半 0 一半 1，这就是数据的不同步。为了避免这种情况，引入了锁的概念。

锁有两种状态——锁定和未锁定。每当一个线程比如”set要访问共享数据时，必须先获得锁定；如果已经有别的线程比如”prin获得锁定了，那么就on让线程”set暂停，也就是同步阻塞；等到线程”prin访问完毕，释放锁以后，再让线程”set继续。

经过这样的处理，打印列表时要么全部输出 0，要么全部输出 1，不会再出现一半 0 一半 1 的尴尬场面。

实例：

```
#coding=utf-8

#!/usr/bin/python

import threading

import time

class myThread (threading.Thread):

    def __init__(self, threadID, name, counter):

        threading.Thread.__init__(self)

        self.threadID = threadID

        self.name = name
```

```
        self.counter = counter

def run(self):

    print "Starting " + self.name

    #获得锁，成功获得锁定后返回 True

    #可选的 timeout 参数不填时将一直阻塞直到获得锁定

    #否则超时后将返回 False

    threadLock.acquire()

    print_time(self.name, self.counter, 3)

    #释放锁

    threadLock.release()


def print_time(threadName, delay, counter):

    while counter:

        time.sleep(delay)

        print "%s: %s" % (threadName, time.ctime(time.time()))

        counter -= 1


threadLock = threading.Lock()

threads = []


# 创建新线程

thread1 = myThread(1, "Thread-1", 1)

thread2 = myThread(2, "Thread-2", 2)
```

```
# 开启新线程

thread1.start()

thread2.start()


# 添加线程到线程列表

threads.append(thread1)

threads.append(thread2)


# 等待所有线程完成

for t in threads:

    t.join()

print "Exiting Main Thread"
```

线程优先级队列（ Queue ）

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

Queue 模块中的常用方法：

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

实例:

```
#coding=utf-8

#!/usr/bin/python

import Queue

import threading

import time

exitFlag = 0

class myThread (threading.Thread):

    def __init__(self, threadID, name, q):

        threading.Thread.__init__(self)

        self.threadID = threadID

        self.name = name

        self.q = q

    def run(self):

        print "Starting " + self.name

        process_data(self.name, self.q)

        print "Exiting " + self.name

def process_data(threadName, q):

    while not exitFlag:
```

```
        queueLock.acquire()

        if not workQueue.empty():

            data = q.get()

            queueLock.release()

            print "%s processing %s" % (threadName, data)

        else:

            queueLock.release()

        time.sleep(1)

threadList = ["Thread-1", "Thread-2", "Thread-3"]

nameList = ["One", "Two", "Three", "Four", "Five"]

queueLock = threading.Lock()

workQueue = Queue.Queue(10)

threads = []

threadID = 1


# 创建新线程

for tName in threadList:

    thread = myThread(threadID, tName, workQueue)

    thread.start()

    threads.append(thread)

    threadID += 1


# 填充队列
```

```
queueLock.acquire()

for word in nameList:

    workQueue.put(word)

queueLock.release()


# 等待队列清空

while not workQueue.empty():

    pass


# 通知线程是时候退出

exitFlag = 1


# 等待所有线程完成

for t in threads:

    t.join()

print "Exiting Main Thread"
```

以上程序执行结果:

```
Starting Thread-1

Starting Thread-2

Starting Thread-3

Thread-1 processing One

Thread-2 processing Two

Thread-3 processing Three
```

```
Thread-1 processing Four
```

```
Thread-2 processing Five
```

```
Exiting Thread-3
```

```
Exiting Thread-1
```

```
Exiting Thread-2
```

```
Exiting Main Thread
```

Python XML 解析

什么是 XML ？

XML 指可扩展标记语言（eXtensible Markup Language）。你可以通过本站学习 [XML 教程](#)

XML 被设计用来传输和存储数据。

XML 是一套定义语义标记的规则，这些标记将文档分成许多部件并对这些部件加以标识。

它也是元标记语言，即定义了用于定义其他与特定领域有关的、语义的、结构化的标记语言的句法语言。

python 对 XML 的解析

常见的 XML 编程接口有 DOM 和 SAX，这两种接口处理 XML 文件的方式不同，当然使用场合也不同。

python 有三种方法解析 XML，SAX，DOM，以及 ElementTree：

1. SAX (simple API for XML)

python 标准库包含 SAX 解析器，SAX 用事件驱动模型，通过在解析 XML 的过程中触发一个个的事件并调用用户定义的回调函数来处理 XML 文件。

2. DOM (Document Object Model)

将 XML 数据在内存中解析成一个树，通过对树的操作来操作 XML。

3. ElementTree (元素树)

ElementTree 就像一个轻量级的 DOM，具有方便友好的 API。代码可用性好，速度快，消耗内存少。

注：因 DOM 需要将 XML 数据映射到内存中的树，一是比较慢，二是比较耗内存，而 SAX 流式读取 XML 文件，比较快，占用内存少，但需要用户实现回调函数（handler）。

本章节使用到的 XML 实例文件 movies.xml 内容如下：

```
<collection shelf="New Arrivals">

<movie title="Enemy Behind">

    <type>War, Thriller</type>

    <format>DVD</format>

    <year>2003</year>

    <rating>PG</rating>

    <stars>10</stars>

    <description>Talk about a US-Japan war</description>

</movie>

<movie title="Transformers">

    <type>Anime, Science Fiction</type>

    <format>DVD</format>

    <year>1989</year>

    <rating>R</rating>

    <stars>8</stars>
```

```
<description>A schientific fiction</description>

</movie>

<movie title="Trigun">

  <type>Anime, Action</type>

  <format>DVD</format>

  <episodes>4</episodes>

  <rating>PG</rating>

  <stars>10</stars>

  <description>Vash the Stampede!</description>

</movie>

<movie title="Ishtar">

  <type>Comedy</type>

  <format>VHS</format>

  <rating>PG</rating>

  <stars>2</stars>

  <description>Viewable boredom</description>

</movie>

</collection>
```

python 使用 SAX 解析 xml

SAX 是一种基于事件驱动的 API。

利用 SAX 解析 XML 文档牵涉到两个部分:解析器和事件处理器。

解析器负责读取 XML 文档,并向事件处理器发送事件,如元素开始跟元素结束事件;

而事件处理器则负责对事件作出相应,对传递的 XML 数据进行处理。

<psax 适于处理下面的问题: < p="" style="color: rgb(0, 0, 0); font-family: 'Microsoft Yahei', 'Helvetica Neue', Helvetica, Arial, sans-serif; font-size: 12px; font-style: normal; font-variant: normal; font-weight: normal; letter-spacing: normal; line-height: normal; orphans: auto; text-align: start; text-indent: 0px; text-transform: none; white-space: normal; widows: auto; word-spacing: 0px; -webkit-text-stroke-width: 0px; background-color: rgb(255, 255, 255);">

- 1、对大型文件进行处理;
- 2、只需要文件的部分内容,或者只需从文件中得到特定信息。
- 3、想建立自己的对象模型的时候。

在 python 中使用 sax 方式处理 xml 要先引入 xml.sax 中的 parse 函数,还有 xml.sax.handler 中的 ContentHandler。

ContentHandler 类方法介绍

characters(content)方法

调用时机:

从行开始,遇到标签之前,存在字符,content的值为这些字符串。

从一个标签,遇到下一个标签之前,存在字符,content的值为这些字符串。

从一个标签,遇到行结束符之前,存在字符,content的值为这些字符串。

标签可以是开始标签,也可以是结束标签。

startDocument() 方法

文档启动的时候调用。

endDocument() 方法

解析器到达文档结尾时调用。

startElement(name, attrs)方法

遇到 XML 开始标签时调用,name 是标签的名字,attrs是标签的属性值字典。

endElement(name) 方法

遇到 XML 结束标签时调用。

make_parser 方法

以下方法创建一个新的解析器对象并返回。

```
xml.sax.make_parser( [parser_list] )
```

参数说明：

- parser_list-可选参数，解析器列表

parser 方法

以下方法创建一个 SAX 解析器并解析 xml 文档：

```
xml.sax.parse( xmlfile, contenthandler[, errorhandler])
```

参数说明：

- xmlfile- xml文件名
- contenthandler -必须是一个 ContentHandler 的对象
- errorhandler -如果指定该参数，errorhandler必须是一个 SAX ErrorHandler 对象

parseString 方法

parseString方法创建一个 XML 解析器并解析 xml 字符串：

```
xml.sax.parseString(xmlstring, contenthandler[, errorhandler])
```

参数说明：

- xmlstring - xml字符串
 - contenthandler -必须是一个 ContentHandler 的对象
 - errorhandler -如果指定该参数，errorhandler必须是一个 SAX ErrorHandler 对象
-

Python 解析 XML 实例

```
#coding=utf-8

#!/usr/bin/python

import xml.sax

class MovieHandler( xml.sax.ContentHandler ):

    def __init__(self):

        self.CurrentData = ""

        self.type = ""

        self.format = ""

        self.year = ""

        self.rating = ""

        self.stars = ""

        self.description = ""

    # 元素开始事件处理

    def startElement(self, tag, attributes):

        self.CurrentData = tag

        if tag == "movie":

            print "*****Movie*****"

            title = attributes["title"]

            print "Title:", title
```

元素结束事件处理

```
def endElement(self, tag):  
  
    if self.CurrentData == "type":  
  
        print "Type:", self.type  
  
    elif self.CurrentData == "format":  
  
        print "Format:", self.format  
  
    elif self.CurrentData == "year":  
  
        print "Year:", self.year  
  
    elif self.CurrentData == "rating":  
  
        print "Rating:", self.rating  
  
    elif self.CurrentData == "stars":  
  
        print "Stars:", self.stars  
  
    elif self.CurrentData == "description":  
  
        print "Description:", self.description  
  
    self.CurrentData = ""
```

内容事件处理

```
def characters(self, content):  
  
    if self.CurrentData == "type":  
  
        self.type = content  
  
    elif self.CurrentData == "format":  
  
        self.format = content  
  
    elif self.CurrentData == "year":
```

```
        self.year = content

    elif self.CurrentData == "rating":

        self.rating = content

    elif self.CurrentData == "stars":

        self.stars = content

    elif self.CurrentData == "description":

        self.description = content

if ( __name__ == "__main__" ):

    # 创建一个 XMLReader

    parser = xml.sax.make_parser()

    # turn off namespaces

    parser.setFeature(xml.sax.handler.feature_namespaces, 0)

    # 重写 ContextHandler

    Handler = MovieHandler()

    parser.setContentHandler( Handler )

    parser.parse("movies.xml")
```

以上代码执行结果如下：

```
*****Movie*****

Title: Enemy Behind
```

Type: War, Thriller

Format: DVD

Year: 2003

Rating: PG

Stars: 10

Description: Talk about a US-Japan war

*****Movie*****

Title: Transformers

Type: Anime, Science Fiction

Format: DVD

Year: 1989

Rating: R

Stars: 8

Description: A schientific fiction

*****Movie*****

Title: Trigun

Type: Anime, Action

Format: DVD

Rating: PG

Stars: 10

Description: Vash the Stampede!

*****Movie*****

Title: Ishtar

Type: Comedy

Format: VHS

Rating: PG

Stars: 2

Description: Viewable boredom

完整的 SAX API 文档请查阅 [Python SAX APIs](#)

使用 xml.dom 解析 xml

文件对象模型（Document Object Model，简称 DOM），是 W3C 组织推荐的处理可扩展置标语言的标准编程接口。

一个 DOM 的解析器在解析一个 XML 文档时，一次性读取整个文档，把文档中所有元素保存在内存中的一个树结构里，之后你可以利用 DOM 提供的不同的函数来读取或修改文档的内容和结构，也可以把修改过的内容写入 xml 文件。

python 中用 xml.dom.minidom 来解析 xml 文件，实例如下：

```
#coding=utf-8

#!/usr/bin/python


from xml.dom.minidom import parse

import xml.dom.minidom


# 使用 minidom 解析器打开 XML 文档

DOMTree = xml.dom.minidom.parse("movies.xml")

collection = DOMTree.documentElement

if collection.hasAttribute("shelf"):

    print "Root element : %s" % collection.getAttribute("shelf")
```

```
# 在集合中获取所有电影

movies = collection.getElementsByTagName("movie")


# 打印每部电影的详细信息

for movie in movies:

    print "*****Movie*****"

    if movie.hasAttribute("title"):

        print "Title: %s" % movie.getAttribute("title")


    type = movie.getElementsByTagName('type')[0]

    print "Type: %s" % type.childNodes[0].data

    format = movie.getElementsByTagName('format')[0]

    print "Format: %s" % format.childNodes[0].data

    rating = movie.getElementsByTagName('rating')[0]

    print "Rating: %s" % rating.childNodes[0].data

    description = movie.getElementsByTagName('description')[0]

    print "Description: %s" % description.childNodes[0].data
```

以上程序执行结果如下：

```
Root element : New Arrivals

*****Movie*****

Title: Enemy Behind

Type: War, Thriller
```

```
Format: DVD

Rating: PG

Description: Talk about a US-Japan war

*****Movie*****

Title: Transformers

Type: Anime, Science Fiction

Format: DVD

Rating: R

Description: A schientific fiction

*****Movie*****

Title: Trigun

Type: Anime, Action

Format: DVD

Rating: PG

Description: Vash the Stampede!

*****Movie*****

Title: Ishtar

Type: Comedy

Format: VHS

Rating: PG

Description: Viewable boredom
```

完整的 DOM API 文档请查阅 [Python DOM APIs](#)。

</psax 适于处理下面的问题: <>

python GUI 编程(Tkinter)

python 提供了多个图形开发界面的库，几个常用 Python GUI 库如下：

- **Tkinter:** Tkinter模块("Tk 接口")是 Python 的标准 Tk GUI 工具包的接口. Tk 和 Tkinter 可以在大多数的 Unix 平台下使用,同样可以应用在 Windows 和 Macintosh 系统里., Tk8.0 的后续版本可以实现本地窗口风格,并良好地运行在绝大多数平台中。
- **wxPython :** wxPython 是一款开源软件,是 Python 语言的一套优秀的 GUI 图形库,允许 Python 程序员很方便的创建完整的、功能键全的 GUI 用户界面。
- **Jython:** Jython 程序可以和 Java 无缝集成。除了一些标准模块, Jython 使用 Java 的模块。Jython 几乎拥有标准的 Python 中不依赖于 C 语言的全部模块。比如, Jython 的用户界面将使用 Swing, AWT 或者 SWT 。 Jython 可以被动态或静态地编译成 Java 字节码。

Tkinter 编程

Tkinter 是 Python 的标准 GUI 库。Python 使用 Tkinter 可以快速的创建 GUI 应用程序。

由于 Tkinter 是内置到 python 的安装包中、只要安装好 Python 之后就能 import Tkinter 库、而且 IDLE 也是用 Tkinter 编写而成、对于简单的图形界面 Tkinter 还是能应付自如。

创建一个 GUI 程序

- 1、导入 Tkinter 模块
- 2、创建控件
- 3、指定这个控件的 master, 即这个控件属于哪一个
- 4、告诉 GM(geometry manager) 有一个控件产生了。

实例:

```
#!/usr/bin/python

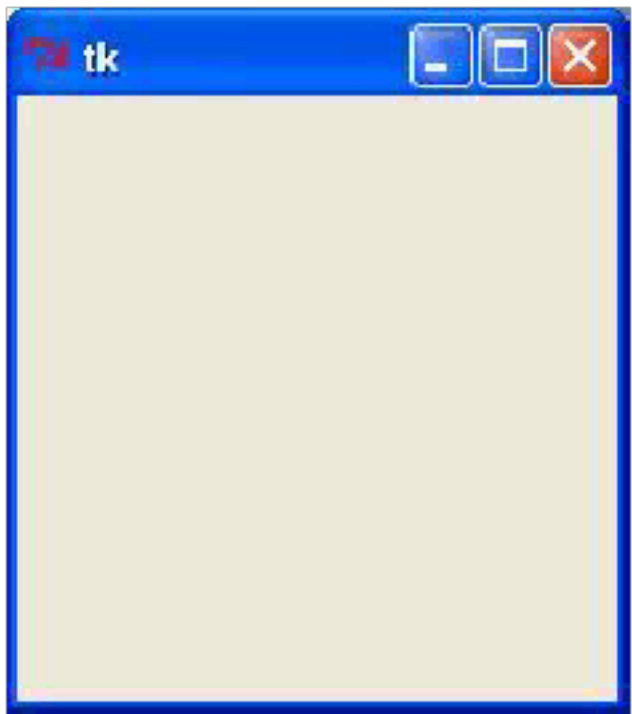
import Tkinter

top = Tkinter.Tk()

# 进入消息循环

top.mainloop()
```

以上代码执行结果如下图：



Tkinter 组件

Tkinter的提供各种控件，如按钮，标签和文本框，一个 GUI 应用程序中使用。这些控件通常被称为控件或者部件。

目前有 15 种 Tkinter的部件。我们提出这些部件以及一个简短的介绍，在下面的表：

控件	描述
Button	按钮控件；在程序中显示按钮。
Canvas	画布控件；显示图形元素如线条或文本
Checkbutton	多选框控件；用于在程序中提供多项选择框
Entry	输入控件；用于显示简单的文本内容
Frame	框架控件；在屏幕上显示一个矩形区域，多用来作为容器
Label	标签控件；可以显示文本和位图
Listbox	列表框控件；在 Listbox窗口小部件是用来显示一个字符串列表给用户
Menubutton	菜单按钮控件，由于显示菜单项。
Menu	菜单控件；显示菜单栏,下拉菜单和弹出菜单
Message	消息控件；用来显示多行文本，与 label比较类似

Radiobutton	单选按钮控件；显示一个单选的按钮状态
Scale	范围控件；显示一个数值刻度，为输出限定范围的数字区间
Scrollbar	滚动条控件，当内容超过可视化区域时使用，如列表框。
Text	文本控件；用于显示多行文本
Toplevel	容器控件；用来提供一个单独的对话框，和 Frame 比较类似
Spinbox	输入控件；与 Entry类似，但是可以指定输入范围值
PanedWindow	PanedWindow 是一个窗口布局管理的插件，可以包含一个或者多个子控件。
LabelFrame	labelframe 是一个简单的容器控件。常用与复杂的窗口布局。
tkMessageBox	用于显示你应用程序的消息框。

标准属性

标准属性也就是所有控件的共同属性，如大小，字体和颜色等等。

属性	描述
Dimension	控件大小；
Color	控件颜色；
Font	控件字体；
Anchor	锚点；
Relief	控件样式；
Bitmap	位图；
Cursor	光标；

几何管理

Tkinter控件有特定的几何状态管理方法，管理整个控件区域组织，一下是 Tkinter公开的几何管理类：包、网格、位置

几何方法	描述
pack()	包装；
grid()	网格；
place()	位置；

Python2.x 与 3.x 版本区别

Python 的 3.0 版本，常被称为 Python 3000，或简称 Py3k。相对于 Python 的早期版本，这是一个较大的升级。

为了不带入过多的累赘，Python 3.0 在设计的时候没有考虑向下相容。

许多针对早期 Python 版本设计的程式都无法在 Python 3.0 上正常执行。

为了照顾现有程式，Python 2.6 作为一个过渡版本，基本使用了 Python 2.x 的语法和库，同时考虑了向 Python 3.0 的迁移，允许使用部分 Python 3.0 的语法与函数。

新的 Python 程式建议使用 Python 3.0 版本的语法。

除非执行环境无法安装 Python 3.0 或者程式本身使用了不支援 Python 3.0 的第三方库。目前不支援 Python 3.0 的第三方库有 Twisted, py2exe, PI 等。

大多数第三方库都正在努力地相容 Python 3.0 版本。即使无法立即使用 Python 3.0，也建议编写相容 Python 3.0 版本的程式，然后使用 Python 2.6, Python 2.x 来执行。

主要变化

Python 3.0 的变化主要在以下几个方面：

print 语句没有了，取而代之的是 print 函数。Python 2.6 与 Python 2.7 部分地支持这种形式的 print 语法。在 Python 2.6 与 Python 2.7 里面，以下三种形式是等价的：

```
print "fish"

print ("fish") #注意 print 后面有个空格

print("fish") #print() 不能带有任何其它参数
```

然而，Python 2.6 实际已经支持新的 print 语法：

```
from __future__ import print_function

print("fish", "panda", sep=', ')
```

新的 str 类别表示一个 Unicode 字符串，相当于 Python 2.x 版本的 unicode 类别。而位元组序列则用类似 b"abc" 的语法表示，用 bytes 类表示，相当于 Python 2.x 的 str 类别。

现在两种类别不能再隐式地自动转换，因此在 Python 3.x 里面 "fish"+b"panda" 是错误的。正确的写法是 "fish"+b"panda".decode("utf-8") Python 2.6 可以自动地将位元组序列识别为 Unicode 字符串，方法是：

```
from __future__ import unicode_literals

print(repr("fish"))
```

除法运算符 "/" 在 Python 3.x 内总是返回浮点数。而在 Python 2.6 内会判断被除数与除数是否是整数。如果是整数会返回整数值，相当于整除；浮点数则返回浮点数值。

为了让 Python 2.6 统一返回浮点数值，可以：

```
from __future__ import division

print(3/2)
```

- 捕获异常的语法由 except exc, var 改为 except exc as var 使用语法 except (exc1, exc2) as var 可以同时捕获多种类别的异常。 Python 2.6 已经支援这两种语法。
- 集合(set)的新写法：{1, 2, 3, 4} 注意 {} 仍然表示空的字典(dict)。
- 字典推导式(Dictionary comprehensions) {expr1: expr2 for k, v in stu} 这个语法等价于

```
result={}

for k, v in d.items():

    result[expr1]=expr2

return result
```

集合推导式(Set Comprehensions) {expr1 for x in stu} 这个语法等价于：

```
result = set()
```



```
for x in stuff:

    result.add(expr1)

return result
```

- 八进制数必须写成 0o777 ，原来的形式 0777 不能用了；二进制必须写成 0b111 。新增了一个 bin()函数用于将一个整数转换成二进制字符串。Python 2.6已经支援这两种语法。
- dict.keys(), dict.values(), dict.items(), map(), filter(),不再返回列表ip而是迭代器。
- 如果两个物件之间没有定义明确的有意义的顺序。使用<, >, <=, 比较它们会投掷异常。比如 1 < "在Python 2.6里面会返回 True,而在Python 3.0里面会投掷异常。现在 cmp(), instance.__cmp__()函数已经被删除。
- 可以注释函数的参数与返回值。此特性可方便 IDE 对原始码进行更深入的分析。例如给参数增加类别讯息：

```
def sendMail(from_ : str, to: str, title: str, body: str) -> bool:

    pass
```

- 合并 int与 long 类型。
- 多个模块被改名（根据 PEP8 ）：

旧的名字	新的名字
_winreg	winreg
ConfigParser	configparser
copy_reg	copyreg
Queue	queue
SocketServer	socketserver
repr	reprlib

- StringIO模块现在被合并到新的 io模组内。 new, md5, gopherlib等模块被删除。Python 2.6已经支援新的 io模组。
- httplib, BaseHTTPServer, CGIHTTPServer, SimpleHTTPServer, Cookie, cookielib被合并到 http包内。
- 取消了 exec 语句，只剩下 exec()函数。 Python 2.6已经支援 exec()函数。

Python IDE

本文为大家推荐几款不错的 Python IDE（集成开发环境），比较推荐 PyCharm，当然你可以根据自己的喜好来选择适合自己的 Python IDE。

PyCharm

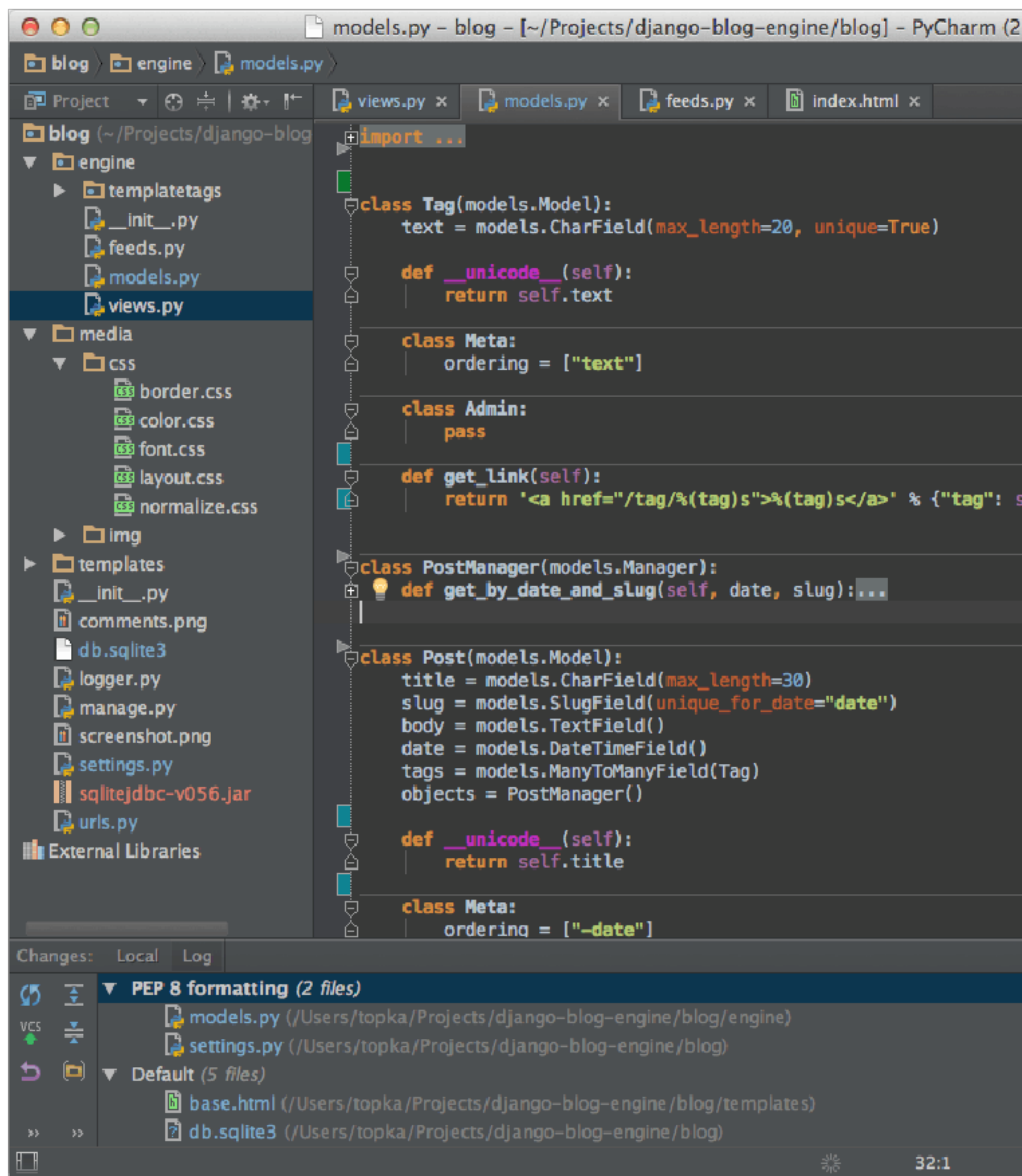
PyCharm 是由 JetBrains 打造的一款 Python IDE。

PyCharm 具备一般 Python IDE 的功能，比如：调试、语法高亮、项目管理、代码跳转、智能提示、自动完成、单元测试、版本控制等。

另外，PyCharm 还提供了一些很好的功能用于 Django 开发，同时支持 Google App Engine，更酷的是，PyCharm 支持 IronPython。

PyCharm 官方下载地址：<http://www.jetbrains.com/pycharm/download/>

效果图查看：

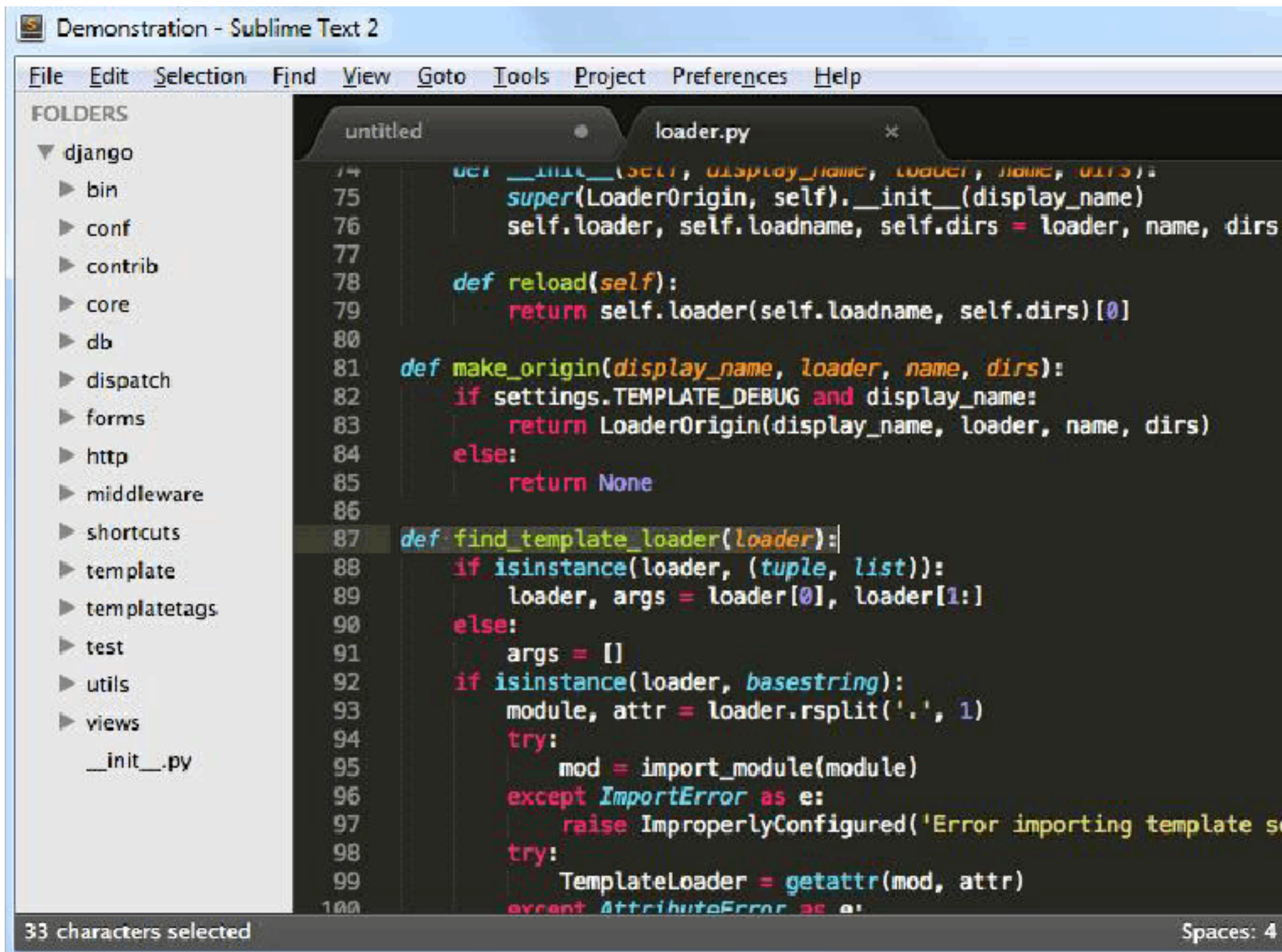


Sublime Text 2

Sublime Text 具有漂亮的用户界面和强大的功能，例如代码缩略图，Python 的插件，代码段等。还可自定义键绑定，菜单和工具栏。

Sublime Text 的主要功能包括：拼写检查，书签，完整的 Python API ， Goto 功能，即时项目切换，多选择，多窗口等等。

Sublime Text 是一个跨平台的编辑器，同时支持 Windows 、 Linux、 Mac OS X 等操作系统。



使用 Sublime Text 2 的插件扩展功能，你可以轻松的打造一款不错的 Python IDE，以下推荐几款插件（你可以找到更多）：

- CodeIntel: 自动补全+成员/方法提示（强烈推荐）
- SublimeREPL : 用于运行和调试一些需要交互的程序（E.G. 使用了 Input 的程序）
- Bracket Highlighter 括号匹配及高亮
- SublimeLinter 代码 pep8 格式检查

Eclipse+Pydev

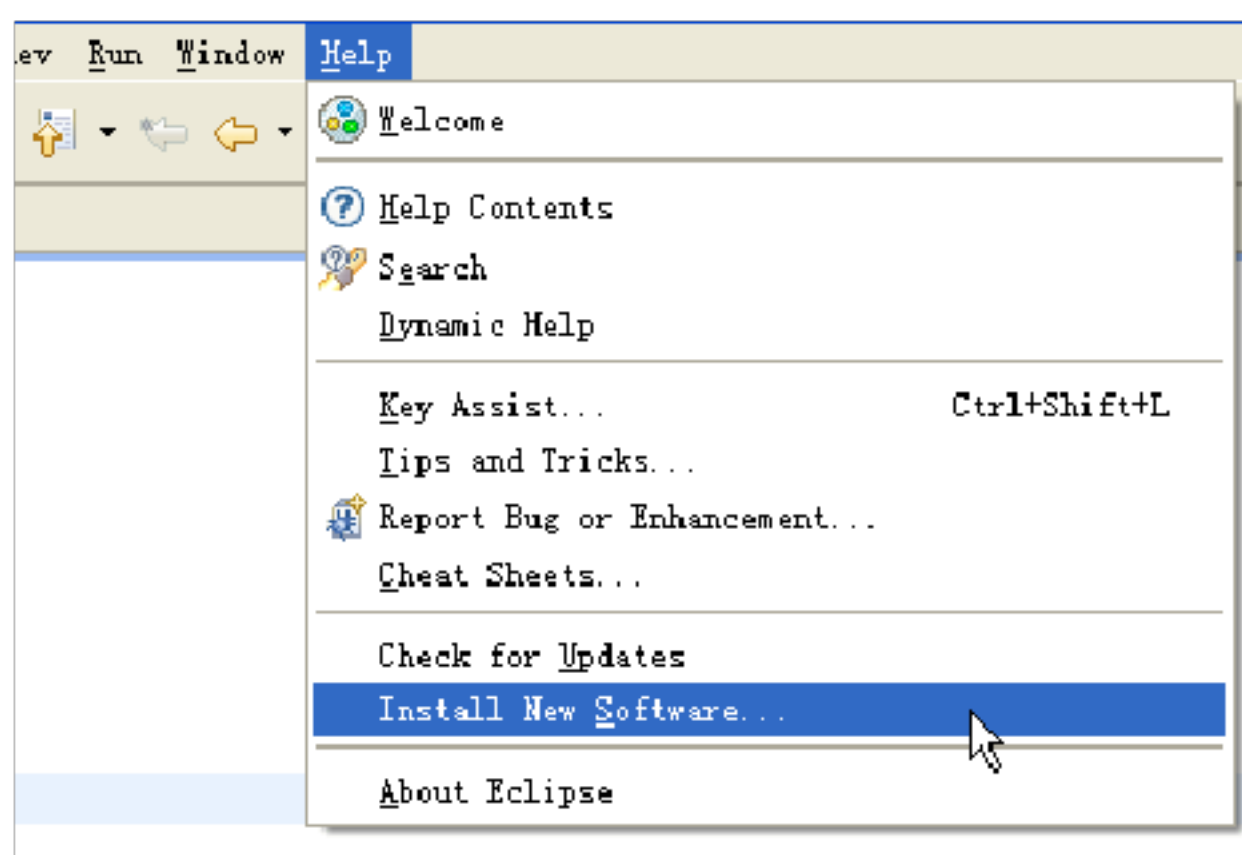
1、安装 Eclipse

Eclipse可以在它的官方网站 [Eclipse.org](http://eclipse.org)找到并下载，通常我们可以选择适合自己的 Eclipse版本，比如 Eclipse Classic 下载完成后解压到你想安装的目录中即可。

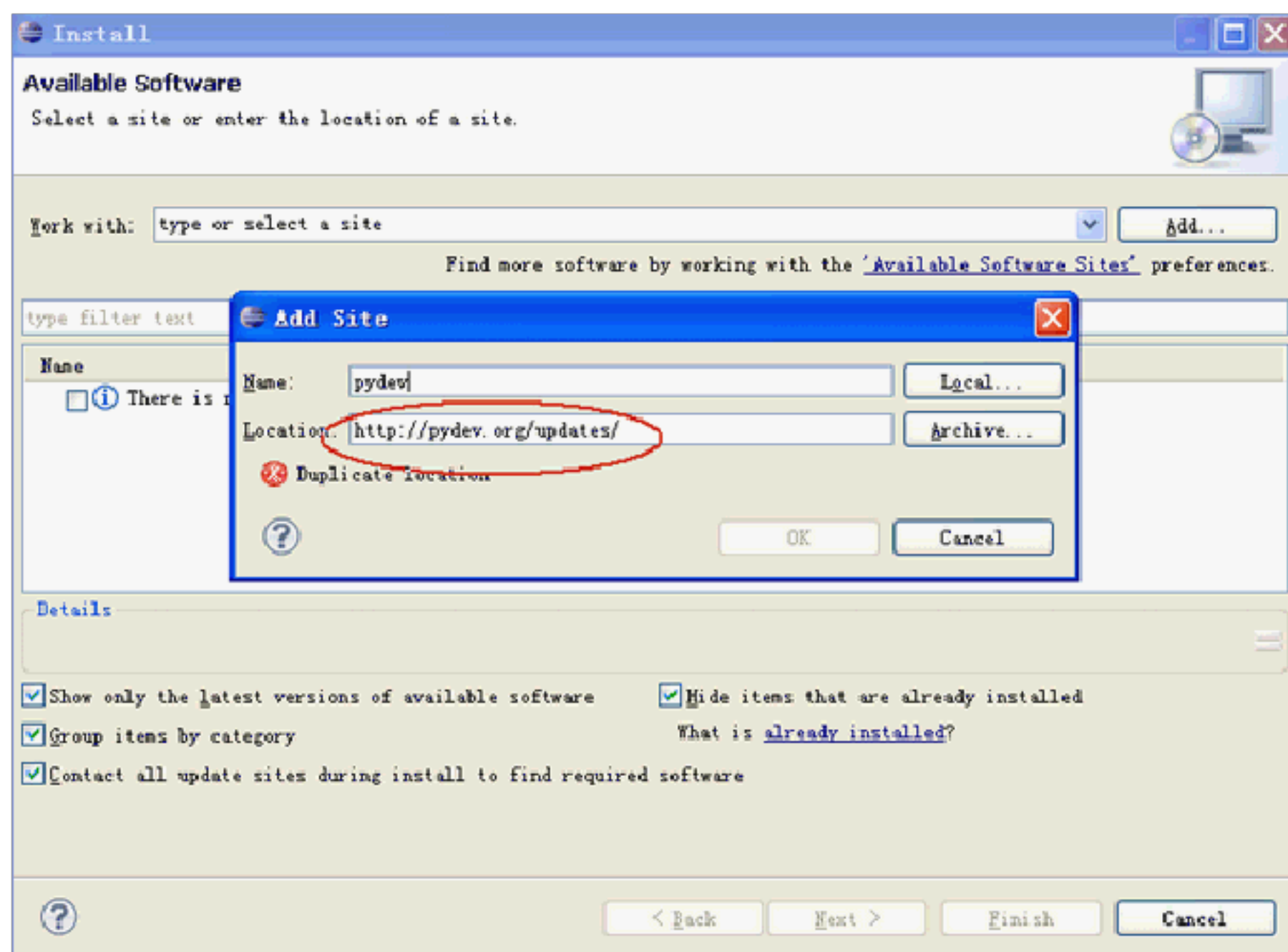
当然在执行 Eclipse之前，你必须确认安装了 Java 运行环境,即必须安装 JRE 或 JDK ，你可以到 (<http://www.java.com/en/download/manual.jsp>)找到 JRE 下载并安装。

2、安装 Pydev

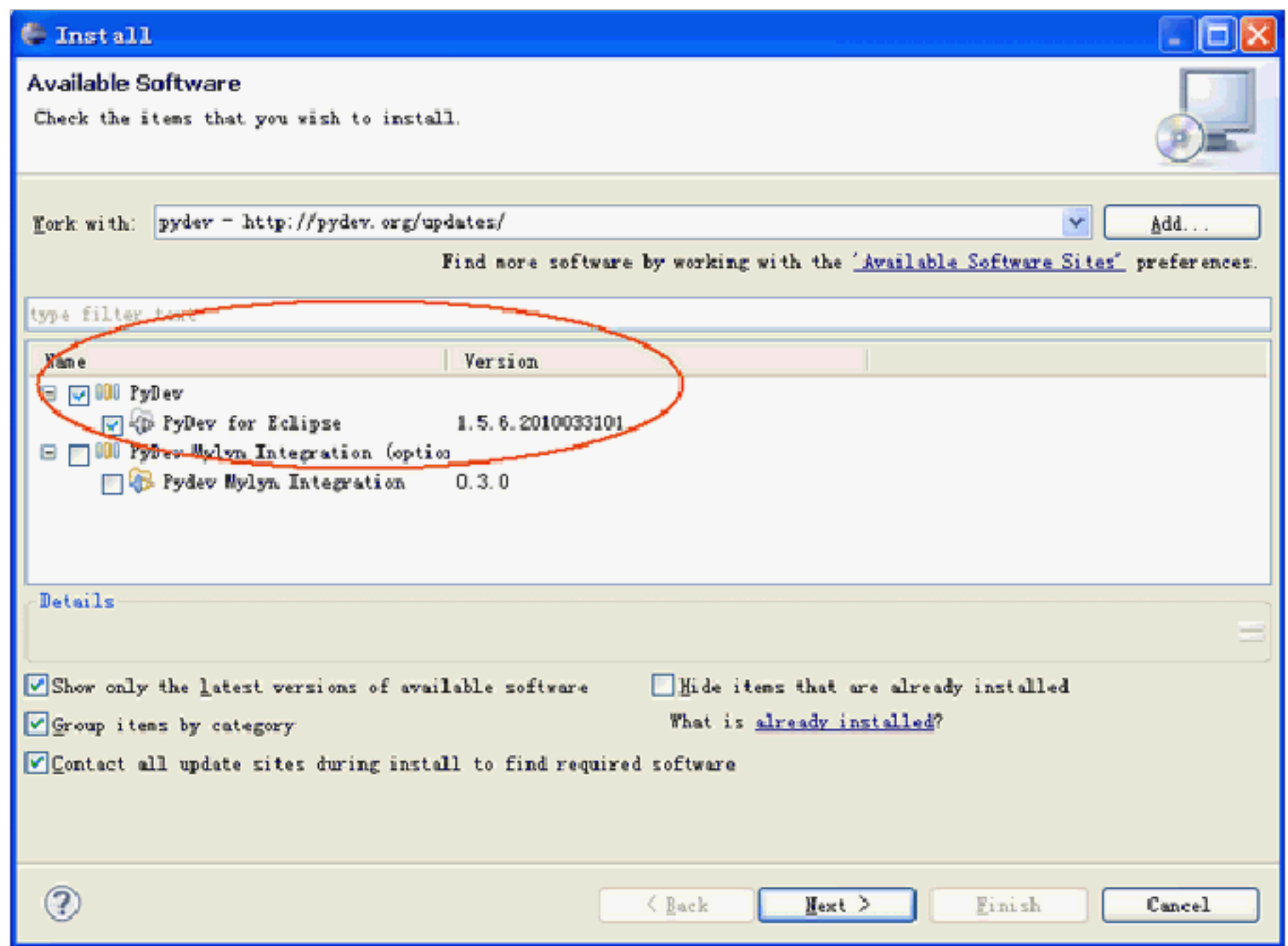
运行 Eclipse之后，选择 help-->Install new Software如下图所示。



点击 Add ，添加 pydev 的安装地址：<http://pydev.org/updates>如下图所示。



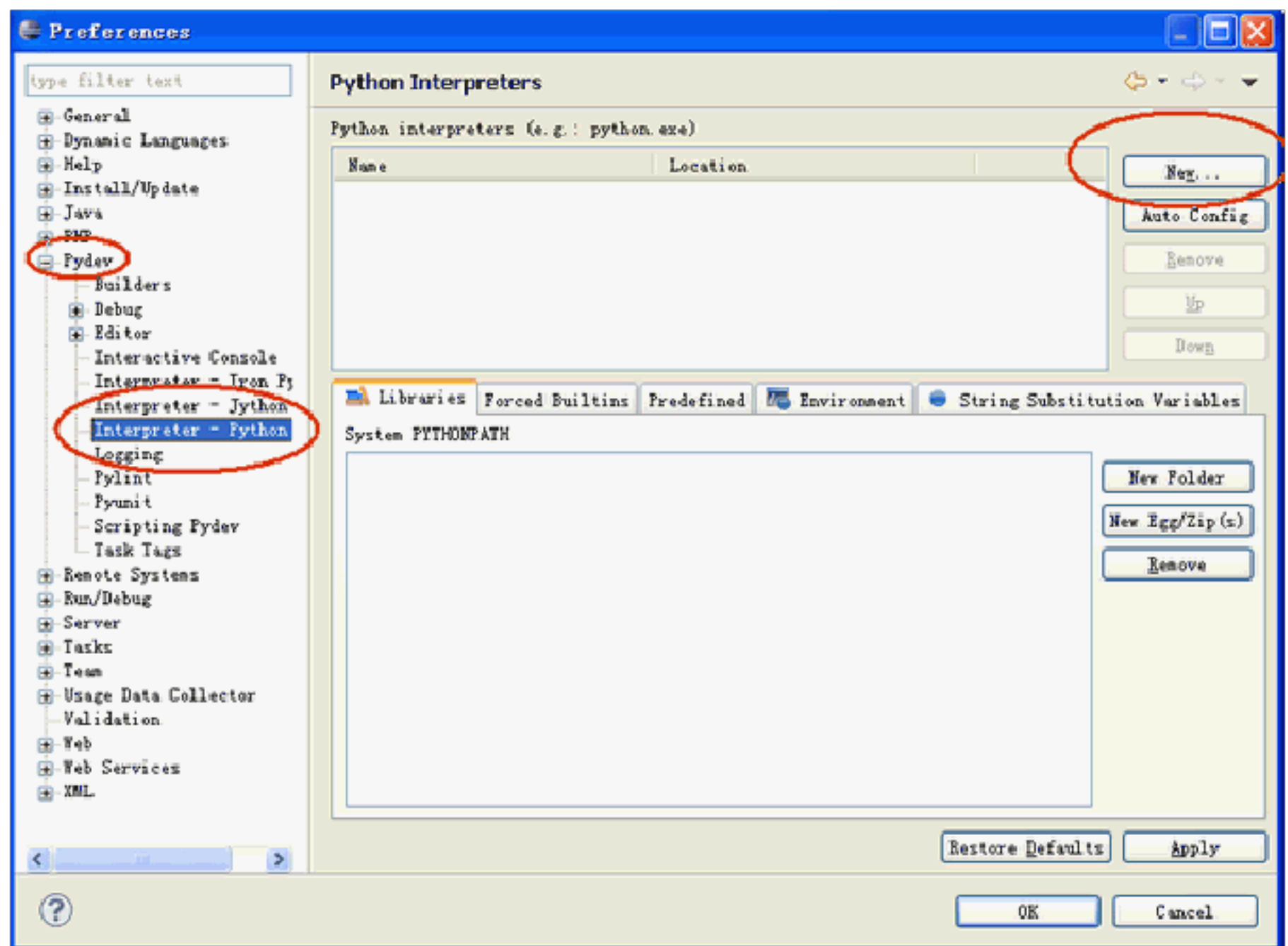
完成后点击“ok”，接着点击PyDev 的“+”，展开PyDev 的节点，要等一小段时间，让它从网上获取PyDev 的相关套件，当完成后会多出PyDev 的相关套件在子节点里，勾选它们然后按 next 进行安装。如下图所示。



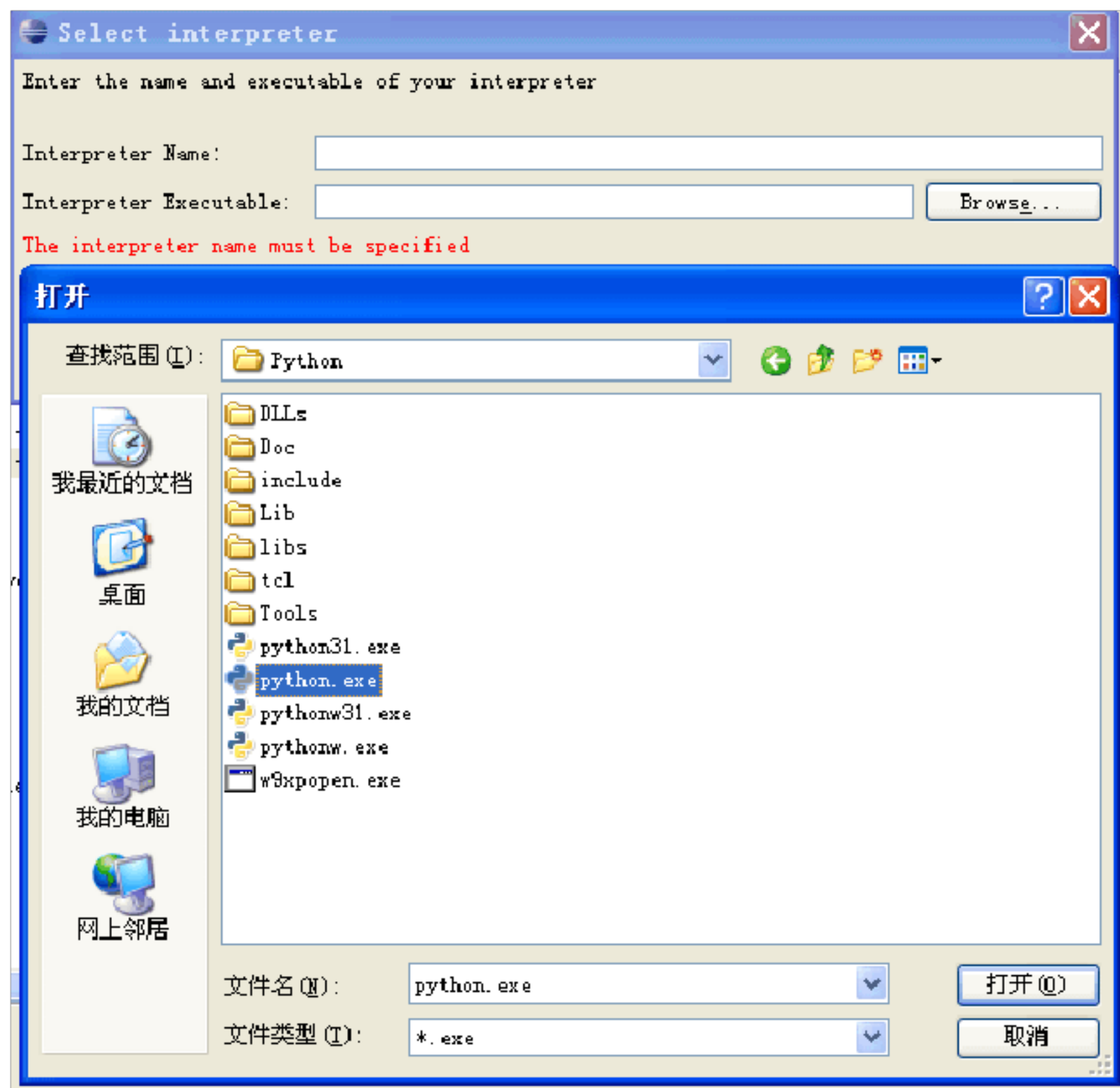
安装完成后，重启 Eclipse即可

3、设置 Pydev

安装完成后，还需要设置一下 PyDev，选择 Window -> Preferences 来设置 PyDev。设置 Python 的路径，从 Pydev 的 Interpreter - Python 页面选择 New



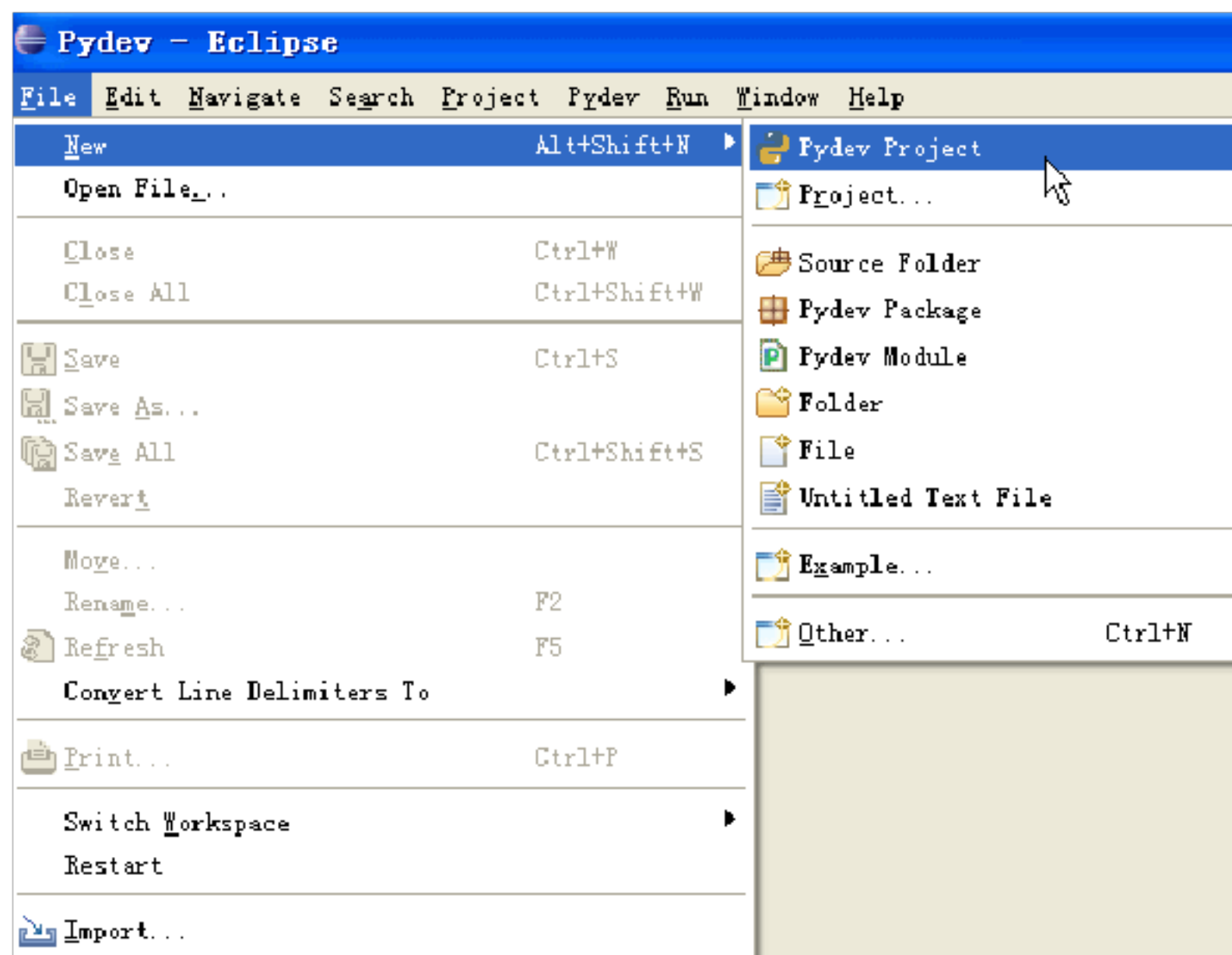
会弹出一个窗口让你选择 Python 的安装位置，选择你安装 Python 的所在位置。



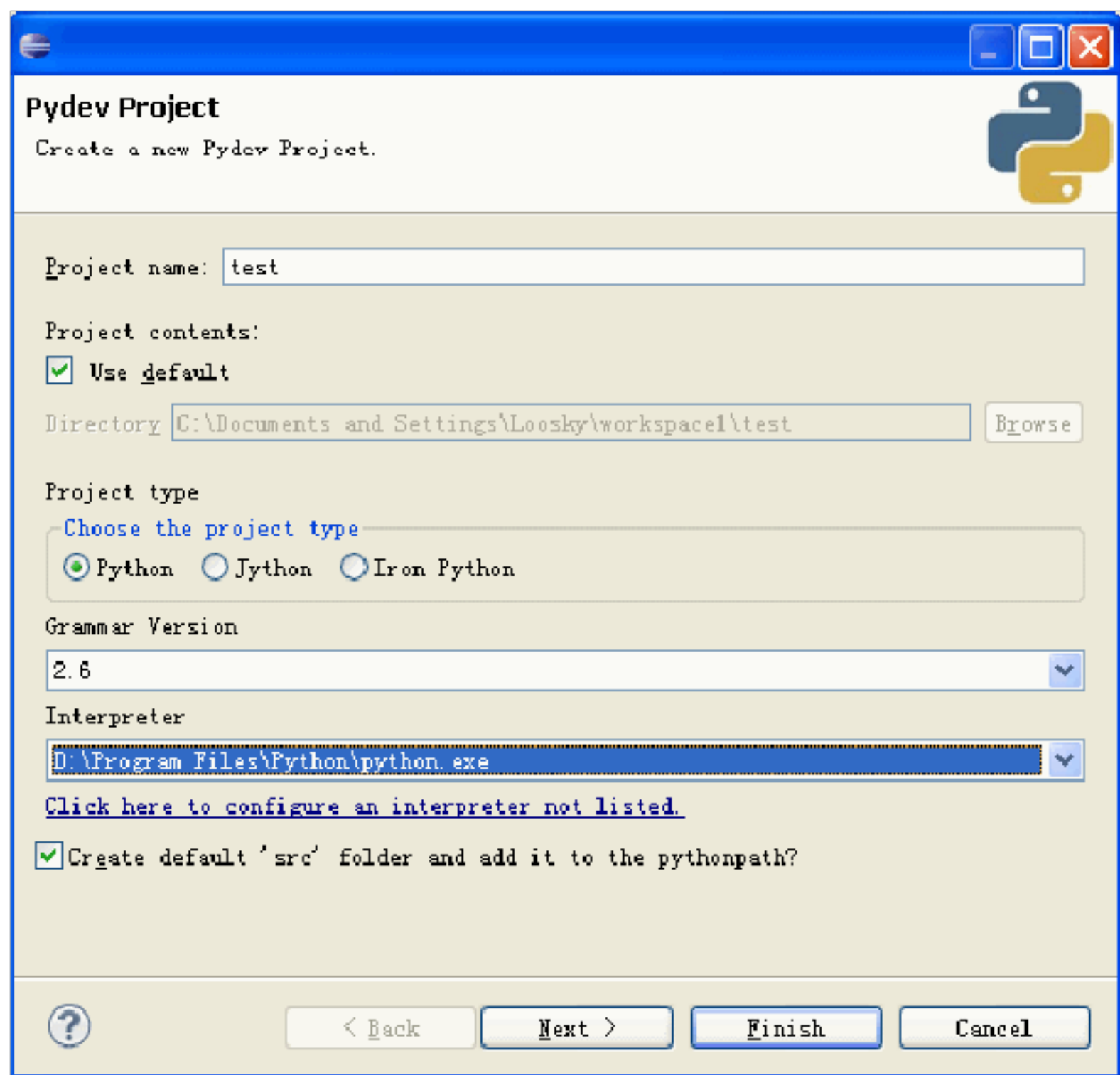
完成之后 PyDev 就设置完成，可以开始使用。

4、建立 Python Project:

安装好 Eclipse+PyDev 以后，我们就可以开始使用它来开发项目了。首先要创建一个项目，选择 File -> New -> Pydev Project

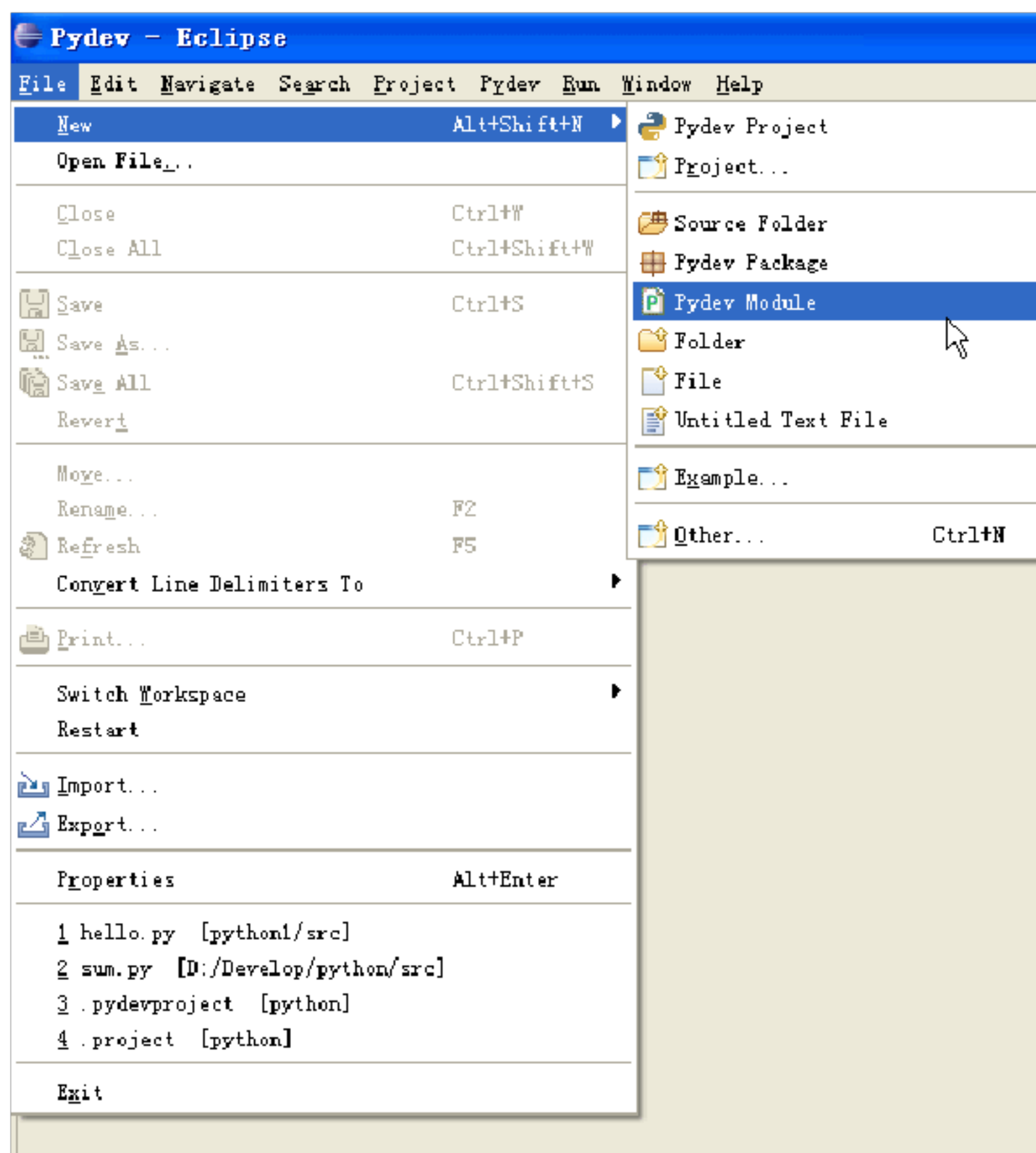


会弹出一个新窗口，填写 Project Name，以及项目保存地址，然后点击 next 完成项目的创建。

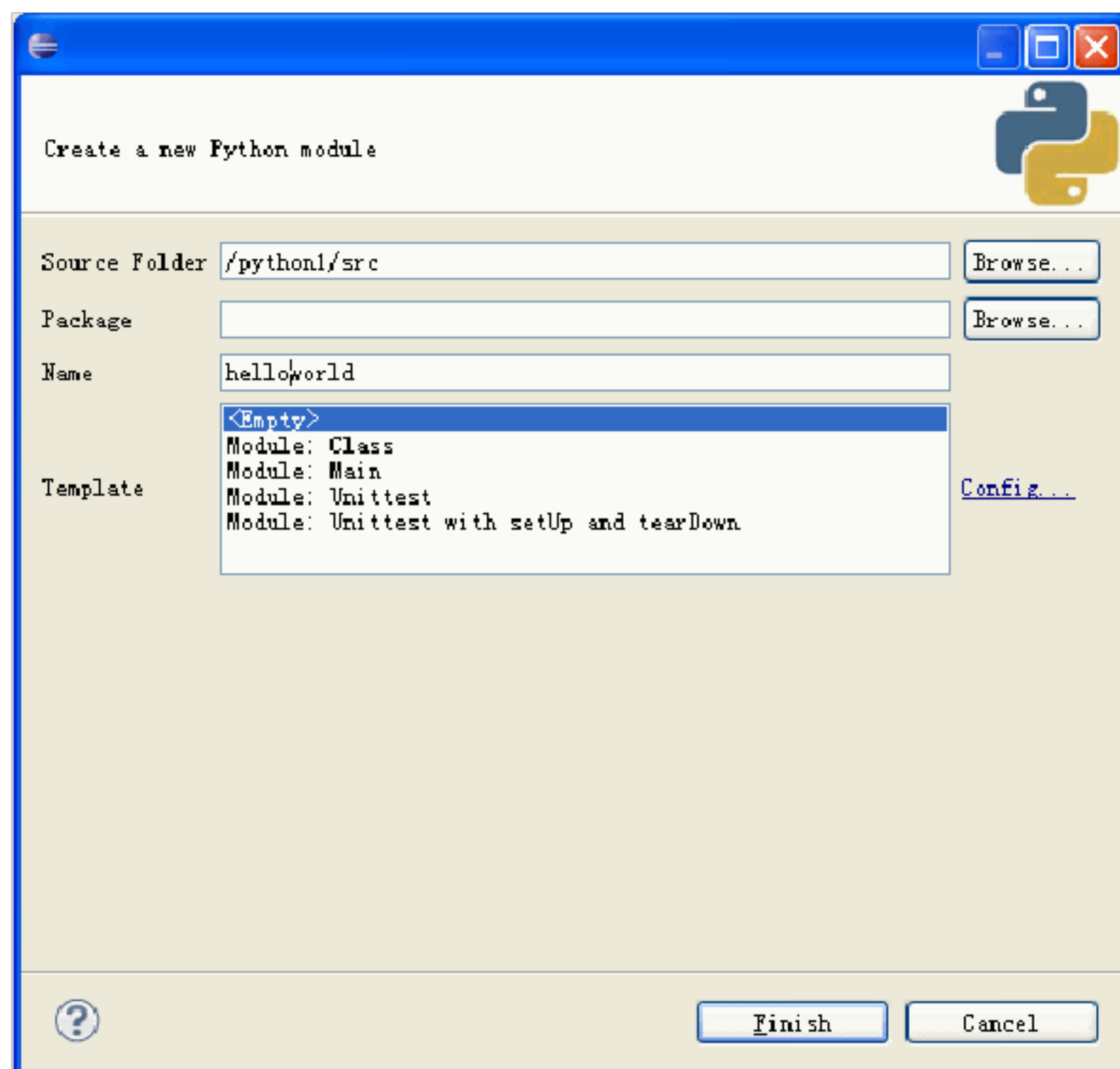


5、创建新的 Pydev Module

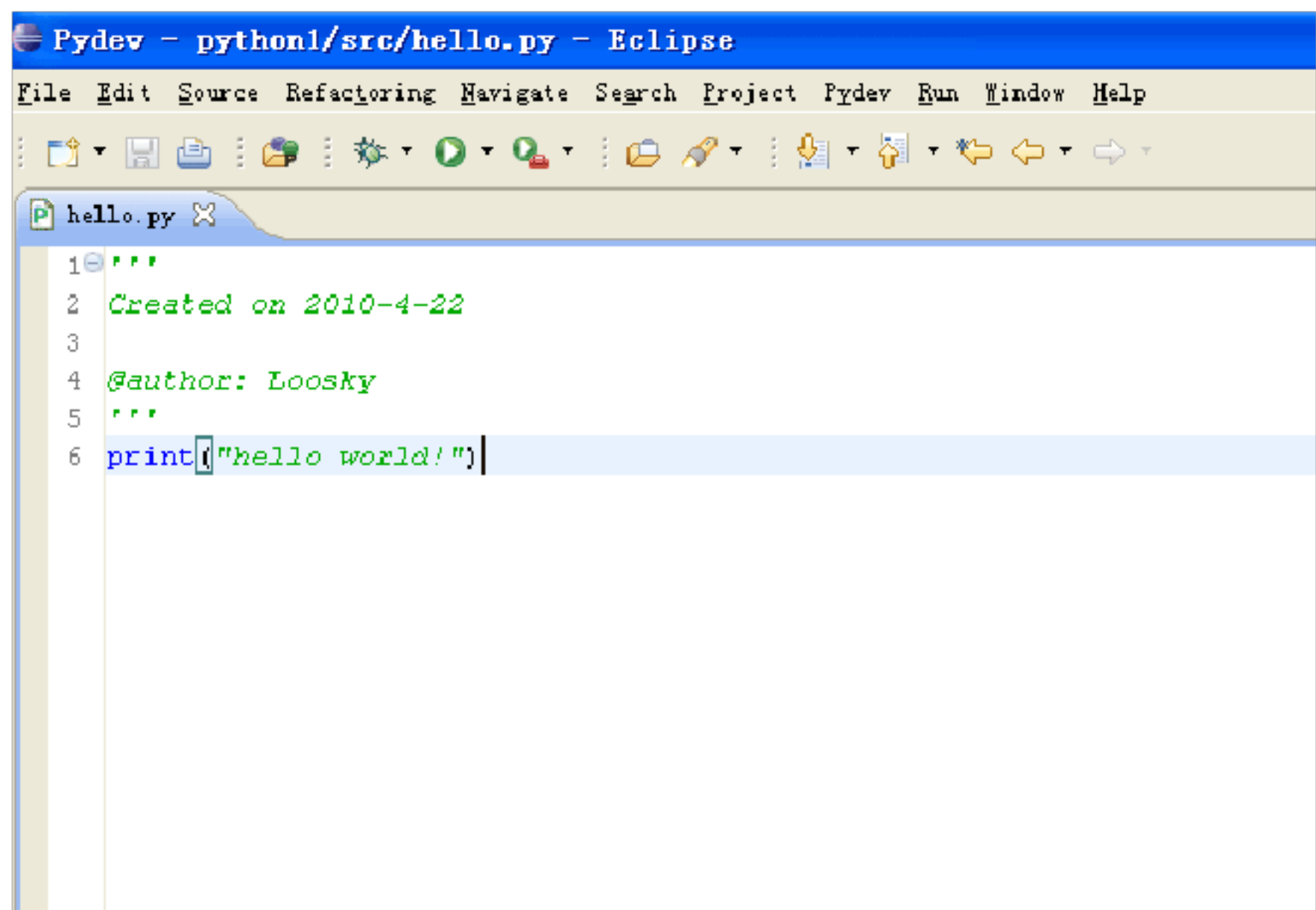
光有项目是无法执行的，接着必须创建新的 Pydev Moudle ，选择 File -> New -> Pydev Module



在弹出的窗口中选择文件存放位置以及 Module Name，注意 Name 不用加 .py，它会自动帮助我们添加。然后点击 Finish 完成创建。

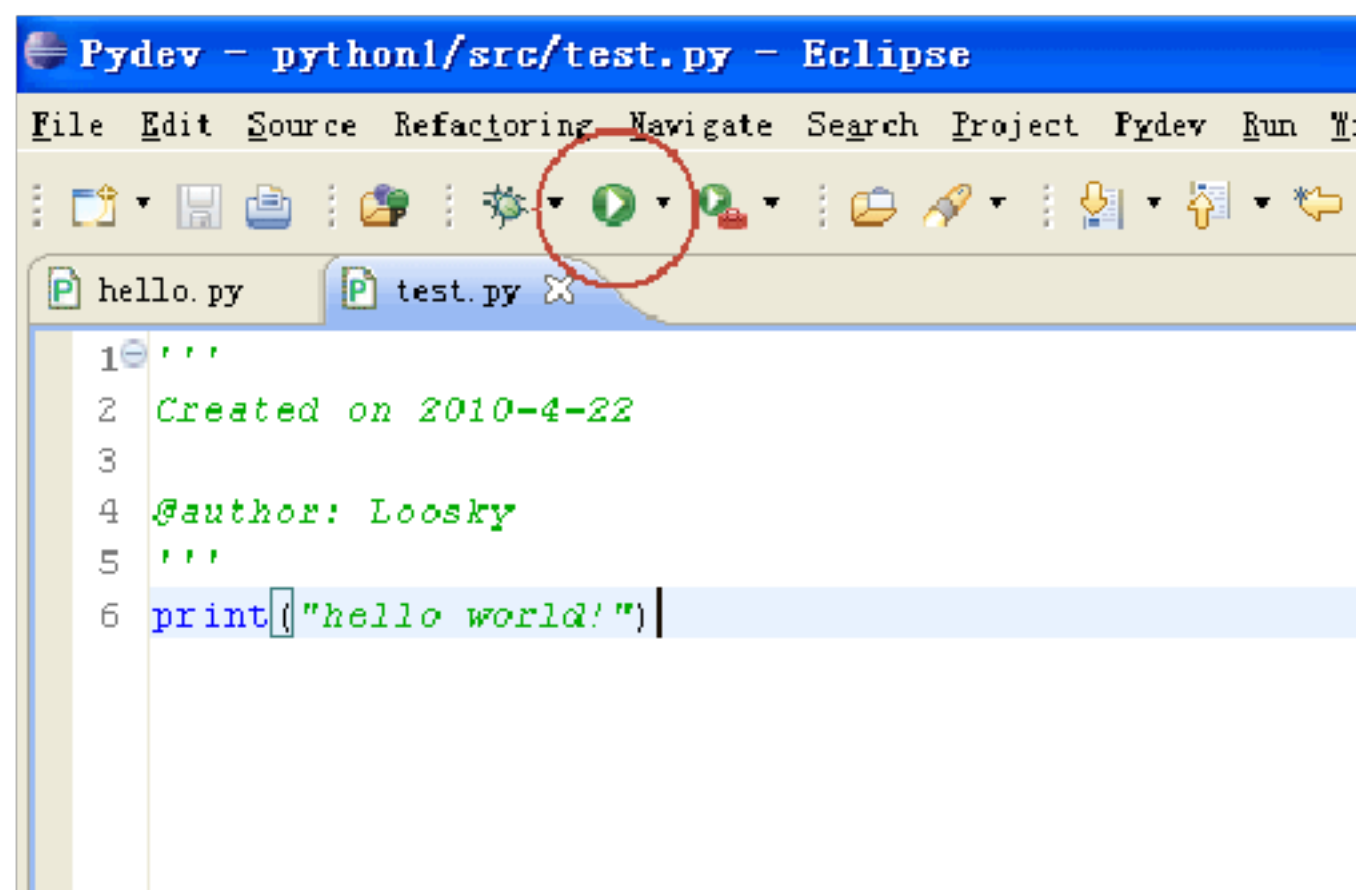


输入"hello world"的代码。

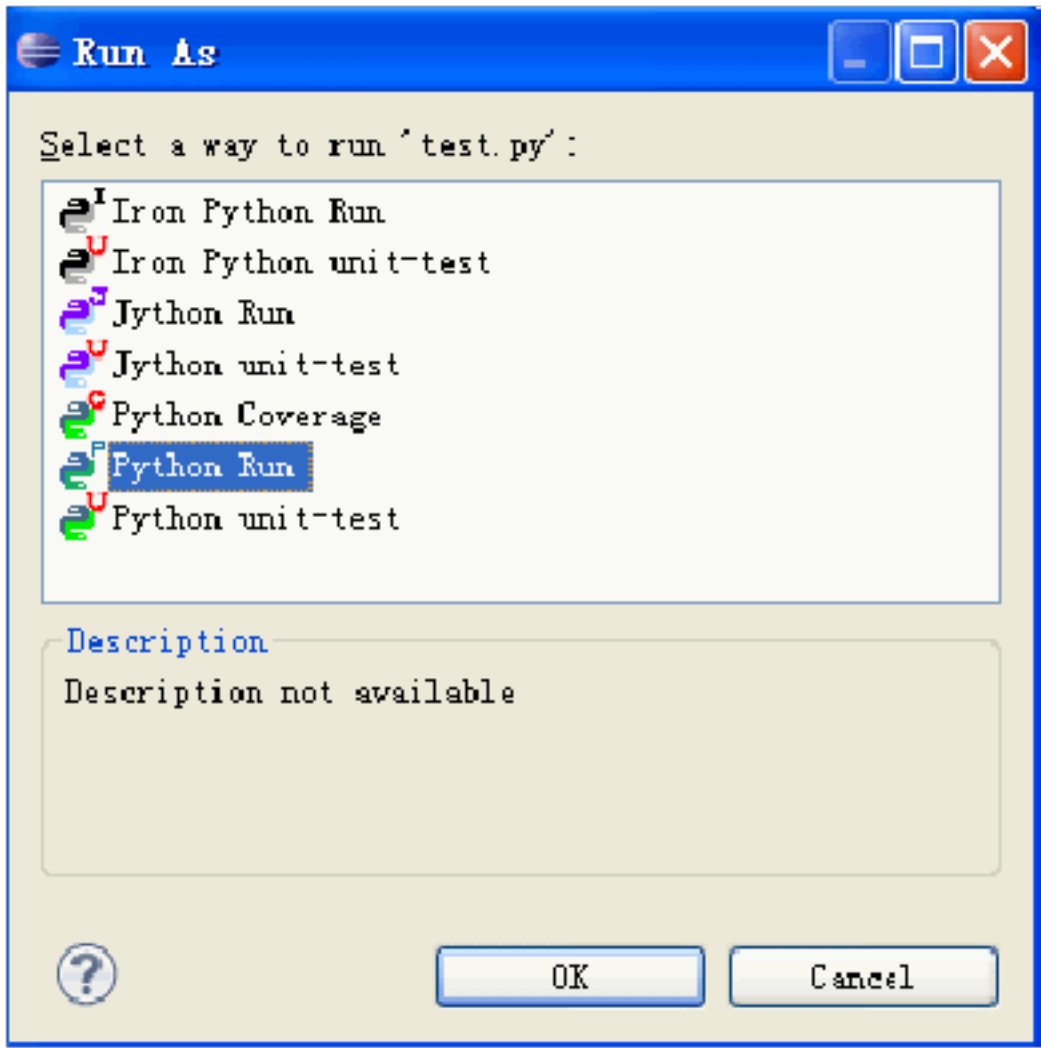


6、执行程序

程序写完后，我们可以开始执行程序,在上方的工具栏上面找到执行的按钮。



之后会弹出一个让你选择执行方式的窗口，通常我们选择 Python Run，开始执行程序。



更多 Python IDE

推荐 10 款最好的 Python IDE：
<http://www.w3school.cc/w3cnote/best-python-ide-for-developers.html>

当然还有非常多很棒的 Python IDE，你可以自由的选择，更多 Python IDE 请参阅：
<http://wiki.python.org/moin/PythonEditors>

Python JSON

本章节我们将为大家介绍如何使用 Python 语言来编码和解码 JSON 对象。

环境配置

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

```
$tar xvfz demjson-1.6.tar.gz
```

```
$cd demjson-1.6
```

```
$python setup.py install
```

JSON 函数

函数	描述
encode	将 Python 对象编码成 JSON 字符串
decode	将已编码的 JSON 字符串解码为 Python 对象

encode

Python encode() 函数用于将 Python 对象编码成 JSON 字符串。

语法

```
demjson.encode(self, obj, nest_level=0)
```

实例

以下实例将数组编码为 JSON 格式数据：

```
#!/usr/bin/python

import demjson

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

json = demjson.encode(data)

print json
```

以上代码执行结果为：


```
[{"a":1,"b":2,"c":3,"d":4,"e":5}]
```

decode

Python 可以使用 `demjson.decode()` 函数解码 JSON 数据。该函数返回 Python 字段的数据类型。

语法

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

实例

以下实例展示了 Python 如何解码 JSON 对象：

```
#!/usr/bin/python

import demjson

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

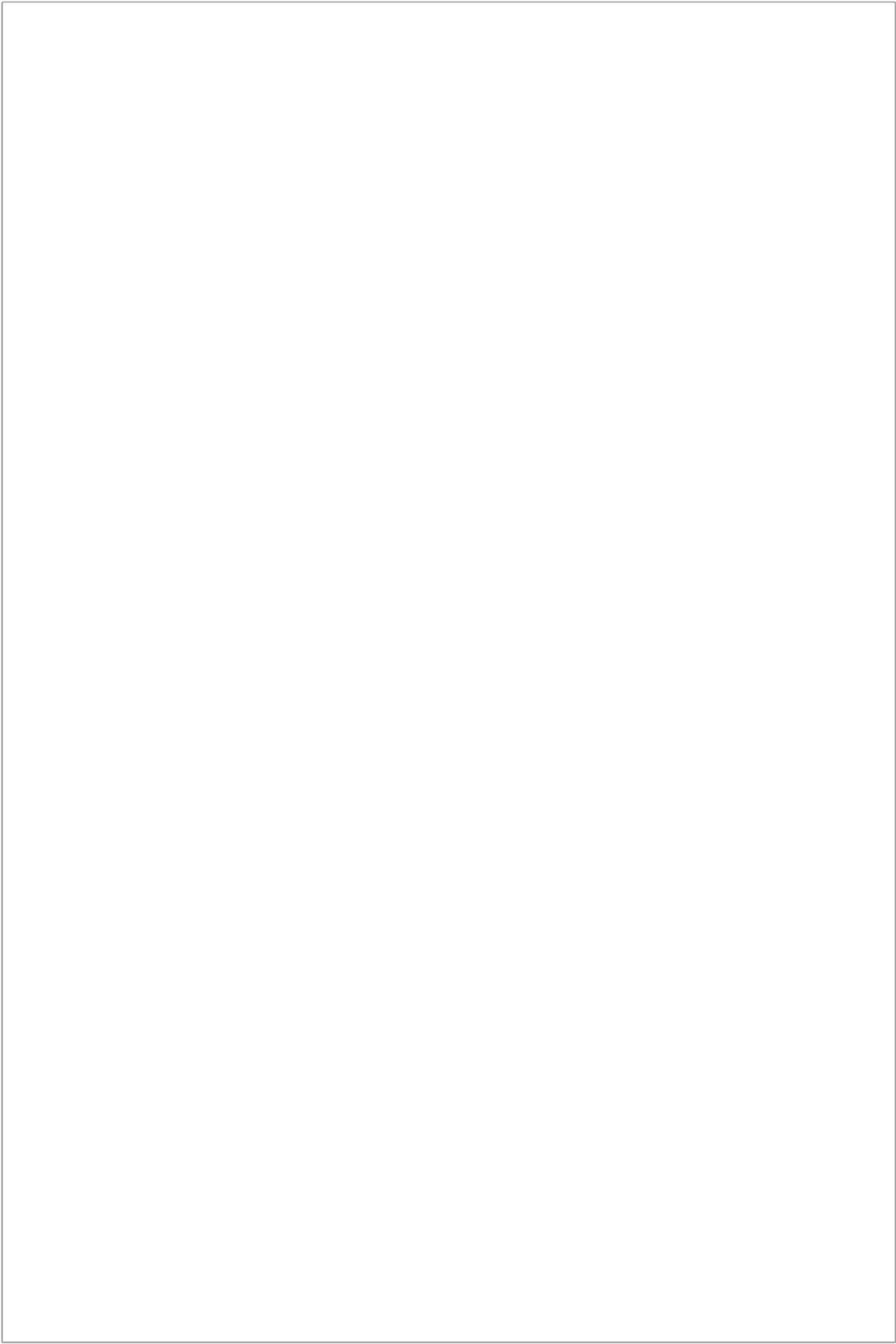
text = demjson.decode(json)

print text
```

以上代码执行结果为：

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





--

