Table of Contents

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！

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！"：

```
实例(Python 2.0+)
#!/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：

```
实例(Python 3.0+)
#!/usr/bin/python3

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，您可以从终端输入执行代码并获得结果的语言，互动的测试和调试代码片断。
- **6.**可移植：基于其开放源代码的特性，Python已经被移植（也就是使其工作）到许多平台。
- **7.**可扩展：如果你需要一段运行很快的关键代码，或者是想要编写一些不愿开放的算法，你可以使用C或C++完成那部分程序，然后从你的Python程序中调用。
- **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.**可嵌入: 你可以将Python嵌入到C/C++程序，让你的程序的用户获得"脚本化"的能力。

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/

你可以在以下链接中下载 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：http://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/
- 选择适用于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 ./configure 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/pythonXX，XX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 为你要安装的版本号。
- 要使用安装程序 python-XYZ.msi, Windows系统必须支持Microsoft Installer 2.0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系统都自带有Python环境，你也可以在链接 http://www.python.org/download/ 上下载最新版安装。

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：

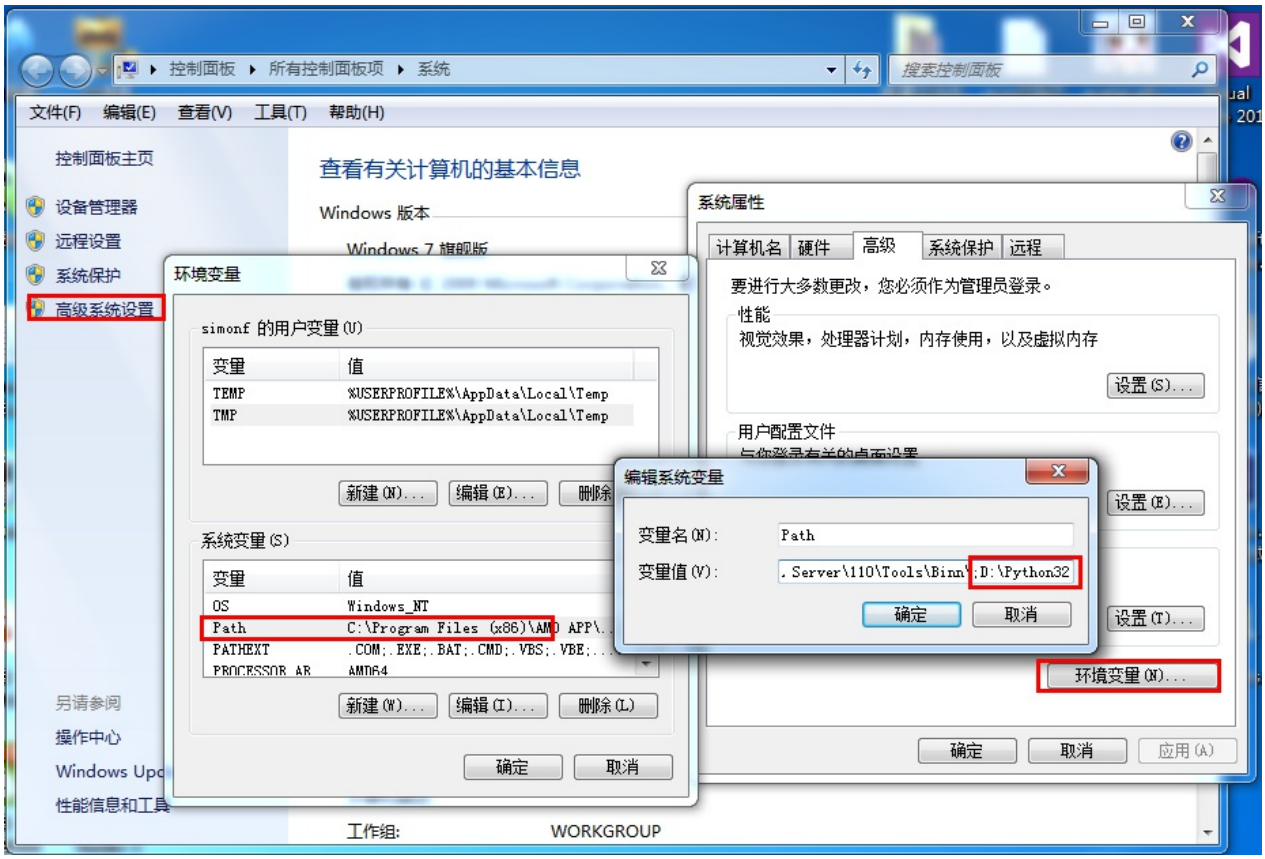
在命令提示框中(cmd) : 输入

```
path=%path%;C:\Python 按下"Enter"。
```

注意: C:\Python 是Python的安装目录。

也可以通过以下方式设置：

- 右键点击"计算机"，然后点击"属性"
- 然后点击"高级系统设置"
- 选择"系统变量"窗口下面的"Path",双击即可！
- 然后在"Path"行，添加python安装路径即可(我的D:\Python32)，所以在后面，添加该路径即可。 ps：记住，路径直接用分号"；"隔开！
- 最后设置成功以后，在cmd命令行，输入命令"python"，就可以有相关显示。



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

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

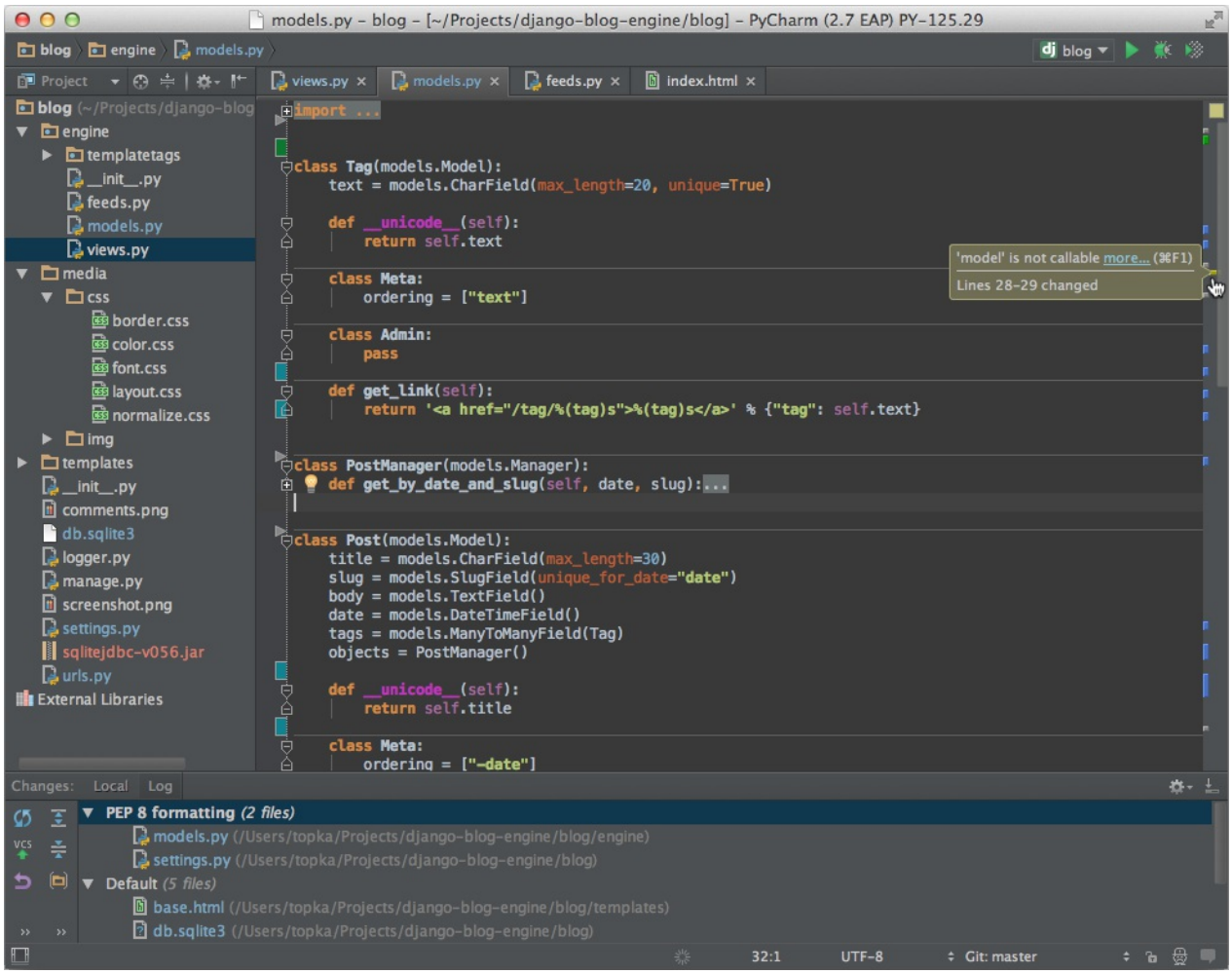
在你的应用程序中通过引入解释器可以在命令行中执行Python脚本，如下所示：

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

或者

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）：PyCharm

PyCharm 是由 JetBrains 打造的一款 Python IDE，支持 macOS、 Windows、 Linux 系统。 PyCharm 功能 : 调试、语法高亮、Project管理、代码跳转、智能提示、自动完成、单元测试、版本控制..... PyCharm 下载地址 : https://www.jetbrains.com/pycharm/download/



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 Python2.7.6 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
```

Python中默认的编码格式是 ASCII 格式，在没修改编码格式时无法正确打印汉字，所以在读取中文时会报错。 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。

```
实例(Python 2.0+)
#!/usr/bin/python
# -*- coding: UTF-8 -*-

print "你好，世界";
```

输出结果为：

```
你好，世界
```

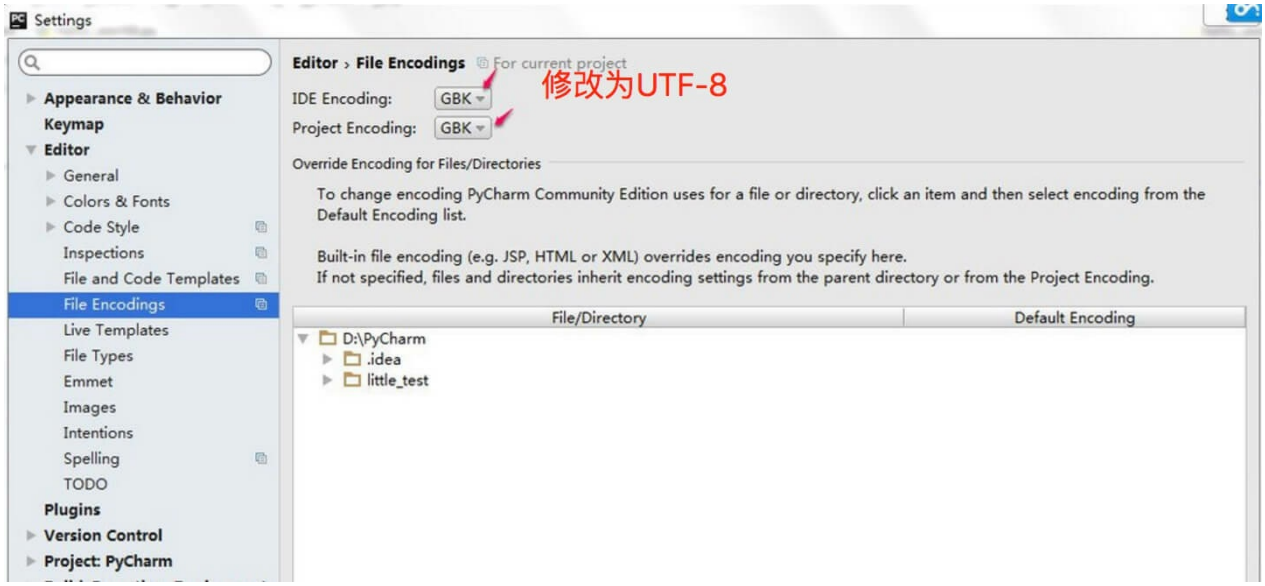
所以如果大家在学习过程中，代码中包含中文，就需要在头部指定编码。

注意：Python3.X 源码文件默认使用utf-8编码，所以可以正常解析中文，无需指定 UTF-8 编码。 注意：如果你使用编辑器，同时需要设置 py 文件存储的格式为 UTF-8，否则会出现类似以下错误信息：

```
SyntaxError: (unicode error) 'utf-8' codec can't decode byte 0xc4 in position 0:
invalid continuation byte
```

Pycharm 设置步骤：

- 进入 file > Settings，在输入框搜索 encoding。
- 找到 Editor > File encodings，将 IDE Encoding 和 Project Encoding 设置为utf-8。



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。

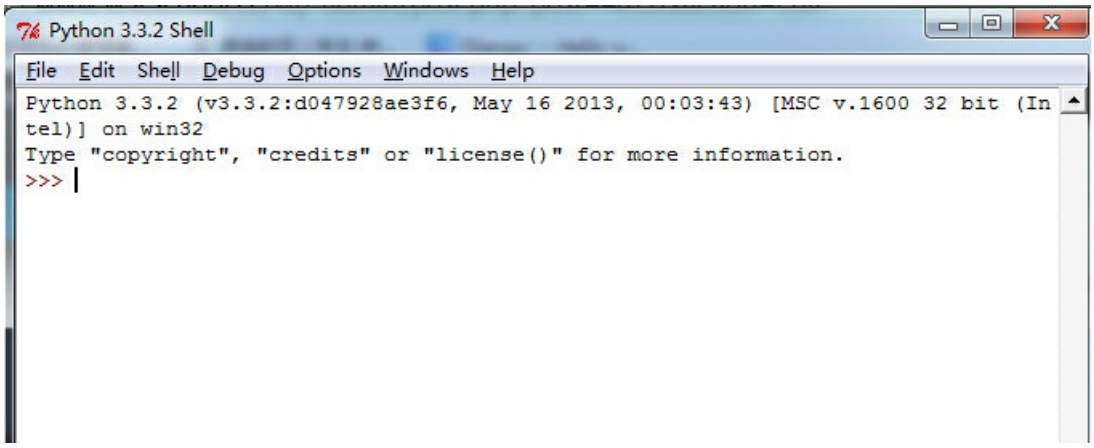
第一个Python程序

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 2.7.6 (default, Sep  9 2014, 15:04:36)
[GCC 4.2.1 Compatible Apple LLVM 6.0 (clang-600.0.39)] on darwin
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 2.7.6 版本中,以上实例输出结果如下：

```
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 *"*而导入； 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。 所有Python的关键字只包含小写字母。

保留字符	保留字符	保留字符
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

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：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
# 文件名：test.py

 if True:
    print "Answer"
    print "True"
else:
    print "Answer"
    # 没有严格缩进，在执行时会报错
  print "False"
```

执行以上代码，会出现如下错误提醒：

```
$ python test.py
  File "test.py", line 5
    if True:
    ^
IndentationError: unexpected indent
```

IndentationError: unexpected indent 错误是python编译器是在告诉你"Hi，老兄，你的文件里格式不对了，可能是tab和空格没对齐的问题"，所有python对格式要求非常严格。 如果是 **IndentationError: unindent does not match any outer indentation level**错误表明，你使用的缩进方式不一致，有的是 tab 键缩进，有的是空格缩进，改为一致即可。 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。 建议你在每个缩进层次使用 单个制表符 或 两个空格 或 四个空格 , 切记不能混用

多行语句

Python语句中一般以新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 可以使用引号(')、双引号(")、三引号(''' 或 """) 来表示字符串，引号的开始与结束必须的相同类型的。 其中三引号可以由多行组成，编写多行文本的快捷语法，常用语文档字符串，在文件的特定地点，被当做注释。

```
word = 'word'
sentence = "这是一个句子。"
paragraph = """这是一个段落。
包含了多个语句"""
```

Python注释

python中单行注释采用 # 开头。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
# 文件名：test.py

# 第一个注释
print "Hello, Python!";  # 第二个注释
```

输出结果：

```
Hello, Python!
```

注释可以在语句或表达式行末：

```
name = "Madisetti" # 这是一个注释
```

python 中多行注释使用三个单引号(''')或三个双引号(""")。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
# 文件名：test.py


'''
这是多行注释，使用单引号。
这是多行注释，使用单引号。
这是多行注释，使用单引号。
```

```
'''

"""
这是多行注释，使用双引号。
这是多行注释，使用双引号。
这是多行注释，使用双引号。
"""
```

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 enter(回车) 键退出，其它键显示。

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：

```
#!/usr/bin/python

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

执行以上代码，输入结果为：

```
$ python test.py
runoob
```

Print 输出

print 默认输出是换行的，如果要实现不换行需要在变量末尾加上逗号：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

x="a"
y="b"
# 换行输出
print x
print y

print '---------'
# 不换行输出
print x,
print y,
```

以上实例执行结果为：

```
a
b
---------
a b
```

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. ]
```

我们在使用脚本形式执行 Python 时，可以接收命令行输入的参数，具体使用可以参照 Python 命令行参数。

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 中的变量赋值不需要类型声明。 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：

实例(Python 2.0+)

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

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，一个人的年龄可以用数字来存储，他的名字可以用字符来存储。 Python 定义了一些标准类型，用于存储各种类型的数据。 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语句删除一些对象的引用。 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的引用。例如：

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

Python支持四种不同的数字类型：

- 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	0xDEFABCECBDAECBFBAEl	32.3e+18	.876j
-0490	535633629843L	-90.	-.6545+0J
-0x260	-052318172735L	-32.54e100	3e+26J
0x69	-4721885298529L	70.2E-12	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 + bj,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

如果你要实现从字符串中获取一段子字符串的话，可以使用变量 **[**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：

实例(Python 2.0+)

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

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：

实例(Python 2.0+)

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

list = [ 'runoob', 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：

```
['runoob', 786, 2.23, 'john', 70.2]
runoob
[786, 2.23]
[2.23, 'john', 70.2]
[123, 'john', 123, 'john']
['runoob', 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。但是元组不能二次赋值，相当于只读列表。

实例(Python 2.0+)

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

tuple = ( 'runoob', 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：

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

```
('runoob', 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：

实例(Python 2.0+)

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

tuple = ( 'runoob', 786 , 2.23, 'john', 70.2 )
list = [ 'runoob', 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 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。

实例(Python 2.0+)

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

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
int(x [,base])	将x转换为一个整数
long(x [,base])	将x转换为一个长整数
float(x)	将x转换到一个浮点数
complex(real [,imag])	创建一个复数
str(x)	将对象 x 转换为字符串
repr(x)	将对象 x 转换为表达式字符串
eval(str)	用来计算在字符串中的有效Python表达式,并返回一个对象
tuple(s)	将序列 s 转换为一个元组
list(s)	将序列 s 转换为一个列表
set(s)	转换为可变集合
dict(d)	创建一个字典。d 必须是一个序列 (key,value)元组。
frozenset(s)	转换为不可变集合

chr(x)	将一个整数转换为一个字符
unichr(x)	将一个整数转换为Unicode字符
ord(x)	将一个字符转换为它的整数值
hex(x)	将一个整数转换为一个十六进制字符串
oct(x)	将一个整数转换为一个八进制字符串

Python 运算符

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，"+" 称为运算符。 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=10，b=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
# -*- coding: UTF-8 -*-

a = 21
b = 10
c = 0

c = a + b
print "1 - c 的值为：", c

c = a - b
print "2 - c 的值为：", c

c = a * b
print "3 - c 的值为：", c

c = a / b
print "4 - c 的值为：", c

c = a % b
print "5 - c 的值为：", c

# 修改变量 a 、b 、c
a = 2
b = 3
c = a**b
print "6 - c 的值为：", c

a = 10
b = 5
```

```
c = a//b
print "7 - c 的值为：", c
```

以上实例输出结果：

```
1 - c 的值为： 31
2 - c 的值为： 11
3 - c 的值为： 210
4 - c 的值为： 2
5 - c 的值为： 1
6 - c 的值为： 8
7 - c 的值为： 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
# -*- coding: UTF-8 -*-

a = 21
b = 10
c = 0

if ( a == b ):
   print "1 - a 等于 b"
else:
   print "1 - a 不等于 b"

if ( a != b ):
   print "2 - a 不等于 b"
else:
   print "2 - a 等于 b"

if ( a <> b ):
   print "3 - a 不等于 b"
else:
   print "3 - a 等于 b"

if ( a < b ):
   print "4 - a 小于 b"
else:
   print "4 - a 大于等于 b"

if ( a > b ):
   print "5 - a 大于 b"
else:
   print "5 - a 小于等于 b"

# 修改变量 a 和 b 的值
a = 5;
b = 20;
if ( a <= b ):
   print "6 - a 小于等于 b"
else:
   print "6 - a 大于  b"

if ( b >= a ):
```

```
   print "7 - b 大于等于 a"
else:
   print "7 - b 小于 a"
```

以上实例输出结果：

```
1 - a 不等于 b
2 - a 不等于 b
3 - a 不等于 b
4 - a 大于等于 b
5 - a 大于 b
6 - a 小于等于 b
7 - b 大于等于 a
```

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
# -*- coding: UTF-8 -*-

a = 21
b = 10
c = 0

c = a + b
print "1 - c 的值为：", c

c += a
print "2 - c 的值为：", c

c *= a
print "3 - c 的值为：", c

c /= a
print "4 - c 的值为：", c

c = 2
c %= a
print "5 - c 的值为：", c

c **= a
print "6 - c 的值为：", c

c //= a
print "7 - c 的值为：", c
```

以上实例输出结果：

```
1 - c 的值为： 31
2 - c 的值为： 52
3 - c 的值为： 1092
4 - c 的值为： 52
5 - c 的值为： 2
6 - c 的值为： 2097152
7 - c 的值为： 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： 下表中变量 a 为 60，b 为 13，二进制格式如下：

```
a = 0011 1100

b = 0000 1101

-----------------

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a  = 1100 0011
```

运算符	描述	实例
&	按位与运算符：参与运算的两个值,如果两个相应位都为1,则该位的结果为1,否则为0	(a & b) 输出结果 12 ，二进制解释： 0000 1100
|	按位或运算符：只要对应的二个二进位有一个为1时，结果位就为1。	(a | b) 输出结果 61 ，二进制解释： 0011 1101
^	按位异或运算符：当两对应的二进位相异时，结果为1	(a ^ b) 输出结果 49 ，二进制解释： 0011 0001
~	按位取反运算符：对数据的每个二进制位取反,即把1变为0,把0变为1	(~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：运算数的各二进位全部左移若干位，由"<<"右边的数指定移动的位数，高位丢弃，低位补0。	a << 2 输出结果 240 ，二进制解释： 1111 0000
>>	右移动运算符：把">>"左边的运算数的各二进位全部右移若干位，">>"右边的数指定移动的位数	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
# -*- coding: UTF-8 -*-

a = 60            # 60 = 0011 1100
b = 13            # 13 = 0000 1101
c = 0

c = a & b;        # 12 = 0000 1100
print "1 - c 的值为：", c

c = a | b;        # 61 = 0011 1101
print "2 - c 的值为：", c

c = a ^ b;        # 49 = 0011 0001
print "3 - c 的值为：", c

c = ~a;           # -61 = 1100 0011
print "4 - c 的值为：", c

c = a << 2;       # 240 = 1111 0000
print "5 - c 的值为：", c

c = a >> 2;       # 15 = 0000 1111
print "6 - c 的值为：", c
```

以上实例输出结果：

```
1 - c 的值为： 12
2 - c 的值为： 61
3 - c 的值为： 49
4 - c 的值为： -61
5 - c 的值为： 240
6 - c 的值为： 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, b为 20:

运算符	逻辑表达式	描述	实例
and	x and y	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 20。
or	x or y	布尔"或" - 如果 x 是非 0，它返回 x 的值，否则它返回 y 的计算值。	(a or b) 返回 10。
not	not x	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

以上实例输出结果：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

a = 10
b = 20

if ( a and b ):
   print "1 - 变量 a 和 b 都为 true"
else:
   print "1 - 变量 a 和 b 有一个不为 true"

if ( a or b ):
   print "2 - 变量 a 和 b 都为 true，或其中一个变量为 true"
else:
   print "2 - 变量 a 和 b 都不为 true"

# 修改变量 a 的值
a = 0
if ( a and b ):
   print "3 - 变量 a 和 b 都为 true"
else:
   print "3 - 变量 a 和 b 有一个不为 true"

if ( a or b ):
   print "4 - 变量 a 和 b 都为 true，或其中一个变量为 true"
else:
   print "4 - 变量 a 和 b 都不为 true"

if not( a and b ):
   print "5 - 变量 a 和 b 都为 false，或其中一个变量为 false"
else:
   print "5 - 变量 a 和 b 都为 true"
```

以上实例输出结果：

```
1 - 变量 a 和 b 都为 true
2 - 变量 a 和 b 都为 true，或其中一个变量为 true
3 - 变量 a 和 b 有一个不为 true
4 - 变量 a 和 b 都为 true，或其中一个变量为 true
5 - 变量 a 和 b 都为 false，或其中一个变量为 false
```

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
# -*- coding: UTF-8 -*-

a = 10
b = 20
list = [1, 2, 3, 4, 5 ];

if ( a in list ):
   print "1 - 变量 a 在给定的列表中 list 中"
else:
   print "1 - 变量 a 不在给定的列表中 list 中"

if ( b not in list ):
   print "2 - 变量 b 不在给定的列表中 list 中"
```

```
else:
   print "2 - 变量 b 在给定的列表中 list 中"

# 修改变量 a 的值
a = 2
if ( a in list ):
   print "3 - 变量 a 在给定的列表中 list 中"
else:
   print "3 - 变量 a 不在给定的列表中 list 中"
```

以上实例输出结果：

```
1 - 变量 a 不在给定的列表中 list 中
2 - 变量 b 不在给定的列表中 list 中
3 - 变量 a 在给定的列表中 list 中
```

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
# -*- coding: UTF-8 -*-

a = 20
b = 20

if ( a is b ):
   print "1 - a 和 b 有相同的标识"
else:
   print "1 - a 和 b 没有相同的标识"

if ( id(a) is not id(b) ):
   print "2 - a 和 b 有相同的标识"
else:
   print "2 - a 和 b 没有相同的标识"

# 修改变量 b 的值
b = 30
if ( a is b ):
   print "3 - a 和 b 有相同的标识"
else:
   print "3 - a 和 b 没有相同的标识"

if ( a is not b ):
   print "4 - a 和 b 没有相同的标识"
else:
   print "4 - a 和 b 有相同的标识"
```

以上实例输出结果：

```
1 - a 和 b 有相同的标识
2 - a 和 b 有相同的标识
3 - a 和 b 没有相同的标识
4 - a 和 b 没有相同的标识
```

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：

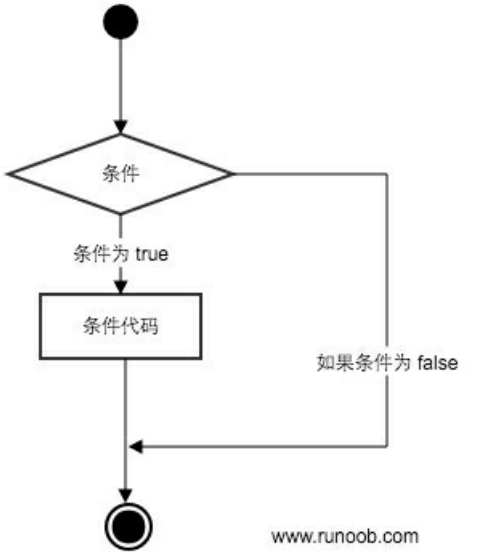
```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

a = 20
b = 10
c = 15
d = 5
e = 0

e = (a + b) * c / d       #( 30 * 15 ) / 5
print "(a + b) * c / d 运算结果为：",  e

e = ((a + b) * c) / d     # (30 * 15 ) / 5
print "((a + b) * c) / d 运算结果为：",  e

e = (a + b) * (c / d);    # (30) * (15/5)
print "(a + b) * (c / d) 运算结果为：",  e

e = a + (b * c) / d;      #  20 + (150/5)
print "a + (b * c) / d 运算结果为：",  e
```

以上实例输出结果：

```
(a + b) * c / d 运算结果为：  90
((a + b) * c) / d 运算结果为：  90
(a + b) * (c / d) 运算结果为：  90
a + (b * c) / d 运算结果为：  5
```

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：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

# 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时，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：

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

```
# -*- coding: UTF-8 -*-
# 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。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

# 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if条件判断语句，如下实例：

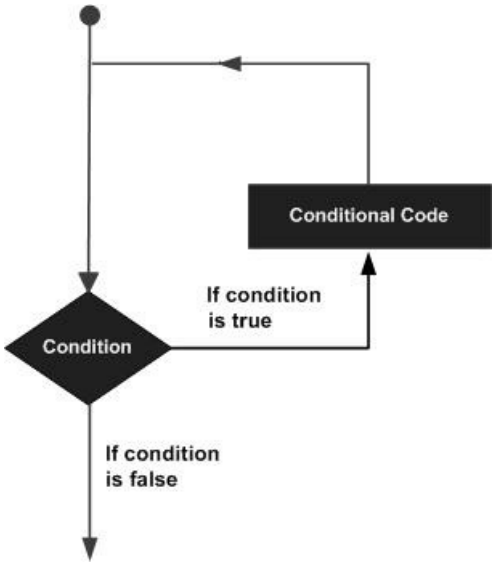
```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

var = 100

if ( var  == 100 ) : print "变量 var 的值为100"

print "Good bye!"
```

以上代码执行输出结果如下：

```
变量 var 的值为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
while 循环	在给定的判断条件为 true 时执行循环体，否则退出循环体。
for 循环	重复执行语句
嵌套循环	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：

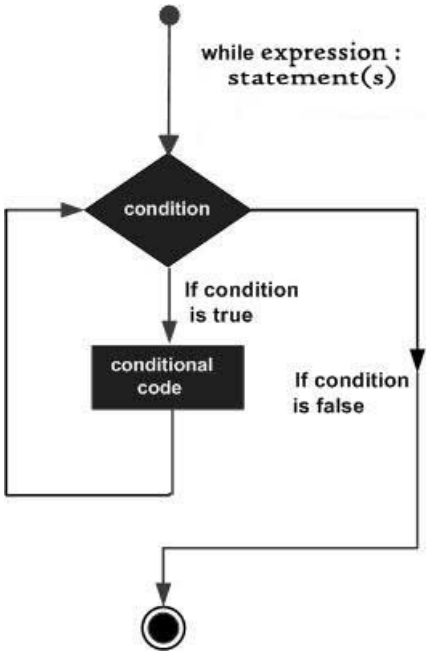
控制语句	描述
break 语句	在语句块执行过程中终止循环，并且跳出整个循环
continue 语句	在语句块执行过程中终止当前循环，跳出该次循环，执行下一次循环。
pass 语句	pass是空语句，是为了保持程序结构的完整性。

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：



Gif 演示 Python while 语句执行过程

```
numbers = [12, 37, 5, 42, 8, 3]
even = []
odd = []
while len(numbers) > 0 :
    number = numbers.pop()
    if(number % 2 == 0):
        even.append(number)
    else:
        odd.append(number)
```

www.penjee.com

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：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

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，while … else 在循环条件为 false 时执行 else 语句块：

```
#!/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

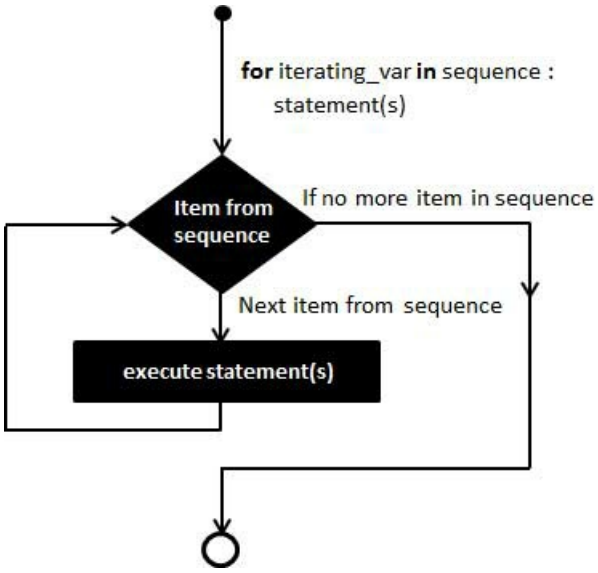
Python for循环可以遍历任何序列的项目，如一个列表或者一个字符串。

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
# -*- coding: UTF-8 -*-

for letter in 'Python':     # 第一个实例
   print '当前字母 :', letter

fruits = ['banana', 'apple',  'mango']
for fruit in fruits:        # 第二个实例
   print '当前水果 :', fruit

print "Good bye!"
```

以上实例输出结果:

```
当前字母 : P
当前字母 : y
当前字母 : t
当前字母 : h
当前字母 : o
当前字母 : n
当前水果 : banana
当前水果 : apple
当前水果 : 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
# -*- coding: UTF-8 -*-

fruits = ['banana', 'apple',  'mango']
for index in range(len(fruits)):
```

```
   print '当前水果 :', fruits[index]

print "Good bye!"
```

以上实例输出结果：

```
当前水果 : banana
当前水果 : apple
当前水果 : 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。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

for num in range(10,20):  # 迭代 10 到 20 之间的数字
   for i in range(2,num): # 根据因子迭代
      if num%i == 0:      # 确定第一个因子
         j=num/i          # 计算第二个因子
         print '%d 等于 %d * %d' % (num,i,j)
         break            # 跳出当前循环
   else:                  # 循环的 else 部分
      print num, '是一个质数'
```

以上实例输出结果：

```
10 等于 2 * 5
11 是一个质数
12 等于 2 * 6
13 是一个质数
14 等于 2 * 7
15 等于 3 * 5
16 等于 2 * 8
17 是一个质数
18 等于 2 * 9
19 是一个质数
```

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：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

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!
```

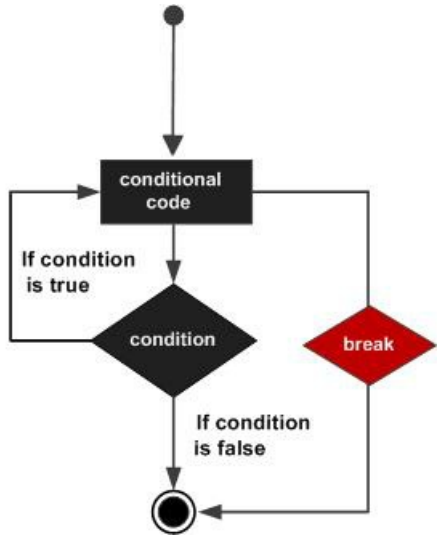
Python break 语句

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!
```

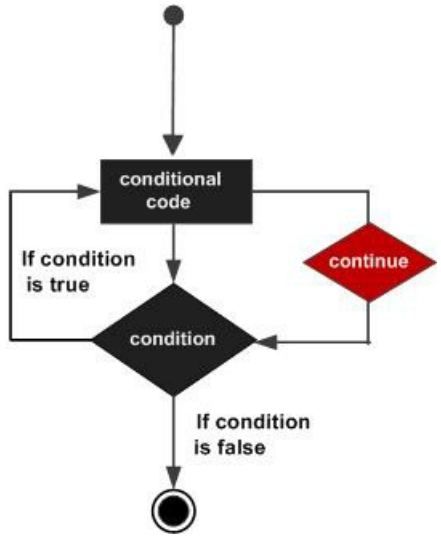
Python continue 语句

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
# -*- coding: UTF-8 -*-

for letter in 'Python':     # 第一个实例
   if letter == 'h':
      continue
   print '当前字母 :', letter

var = 10                    # 第二个实例
while var > 0:
   var = var -1
   if var == 5:
      continue
   print '当前变量值 :', var
print "Good bye!"
```

以上实例执行结果：

```
当前字母 : P
当前字母 : y
当前字母 : t
当前字母 : o
当前字母 : n
当前变量值 : 9
当前变量值 : 8
当前变量值 : 7
当前变量值 : 6
当前变量值 : 4
当前变量值 : 3
当前变量值 : 2
当前变量值 : 1
当前变量值 : 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。 pass 不做任何事情，一般用做占位语句。

Python 语言 pass 语句语法格式如下：

```
pass
```

实例：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

# 输出 Python 的每个字母
for letter in 'Python':
   if letter == 'h':
      pass
      print '这是 pass 块'
   print '当前字母 :', letter

print "Good bye!"
```

以上实例执行结果：

```
当前字母 : P
当前字母 : y
当前字母 : t
这是 pass 块
当前字母 : h
当前字母 : o
当前字母 : n
Good bye!
```

Python Number(数字)

Python Number 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 Number 数据类型的值，将重新分配内存空间。 以下实例在变量赋值时 Number 对象将被创建：

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

您也可以使用del语句删除一些 Number 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。

int	long	float	complex
10	51924361L	0.0	3.14j
100	-0x19323L	15.20	45.j
-786	0122L	-21.9	9.322e-36j
080	0xDEFABCECBDAECBFBAEl	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 + bj,或者complex(a,b)表示， 复数的实部a和虚部b都是浮点型

Python Number 类型转换

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

Python数学函数

函数	返回值 (描述)
abs(x)	返回数字的绝对值，如abs(-10) 返回 10

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

Python随机数函数

随机数可以用于数学，游戏，安全等领域中，还经常被嵌入到算法中，用以提高算法效率，并提高程序的安全性。 Python包含以下常用随机数函数：

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

Python三角函数

Python包括以下三角函数：

函数	描述
acos(x)	返回x的反余弦弧度值。
asin(x)	返回x的反正弦弧度值。
atan(x)	返回x的反正切弧度值。
atan2(y, x)	返回给定的 X 及 Y 坐标值的反正切值。
cos(x)	返回x的弧度的余弦值。
hypot(x, y)	返回欧几里德范数 sqrt(*xx* + *yy*)。
sin(x)	返回的x弧度的正弦值。
tan(x)	返回x弧度的正切值。
degrees(x)	将弧度转换为角度,如degrees(math.pi/2) ， 返回90.0
radians(x)	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('或")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 Runoob"
```

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 Runoob"

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
# -*- coding: UTF-8 -*-

var1 = 'Hello World!'

print "更新字符串 :- ", var1[:6] + 'Runoob!'
```

以上实例执行结果

```
更新字符串 :-  Hello Runoob!
```

Python转义字符

在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
\'	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 + b </p>'HelloPython'
*	重复输出字符串	>>>a * 2</p>'HelloHello'
[]	通过索引获取字符串中字符	>>>a[1]</p>'e'
[:]	截取字符串中的一部分	>>>a[1:4]</p>'ell'
in	成员运算符 - 如果字符串中包含给定的字符返回 True	>>>"H" in a</p>True
not in	成员运算符 - 如果字符串中不包含给定的字符返回 True	>>>"M" not in a</p>True
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。	>>>print r'\n'</p>\n</p>>>> print R'\n'</p>\n
%	格式字符串	请看下一章节

实例如下：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

a = "Hello"
b = "Python"

print "a + b 输出结果：", a + b
print "a * 2 输出结果：", a * 2
print "a[1] 输出结果：", a[1]
print "a[1:4] 输出结果：", a[1:4]

if( "H" in a) :
    print "H 在变量 a 中"
else :
    print "H 不在变量 a 中"

if( "M" not in a) :
    print "M 不在变量 a 中"
else :
    print "M 在变量 a 中"

print r'\n'
print R'\n'
```

以上程序执行结果为：

```
a + b 输出结果： HelloPython
a * 2 输出结果： HelloHello
a[1] 输出结果： e
a[1:4] 输出结果： ell
H 在变量 a 中
M 不在变量 a 中
\n
\n
```

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:

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(+)
	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)

Python2.6 开始，新增了一种格式化字符串的函数 str.format()，它增强了字符串格式化的功能。

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
string.capitalize()	把字符串的第一个字符大写
string.center(width)	返回一个原字符串居中,并使用空格填充至长度 width 的新字符串
string.count(str, beg=0, end=len(string))	返回 str 在 string 里面出现的次数，如果 beg 或者 end 指定则返回指定范围内 str 出现的次数
[string.decode(encoding='UTF-8', errors='strict')	以 encoding 指定的编码格式解码 string，如果出错默认报一个 ValueError 的 异 常 ， 除 非 errors 指 定 的 是 'ignore' 或 者'replace'
string.encode(encoding='UTF-8', errors='strict')	以 encoding 指定的编码格式编码 string，如果出错默认报一个ValueError 的异常，除非 errors 指定的是'ignore'或者'replace'
string.endswith(obj, beg=0, end=len(string))	检查字符串是否以 obj 结束，如果beg 或者 end 指定则检查指定的范围内是否以 obj 结束，如果是，返回 True,否则返回 False.
string.expandtabs(tabsize=8)	把字符串 string 中的 tab 符号转为空格，tab 符号默认的空格数是 8。
string.find(str, beg=0, end=len(string))	检测 str 是否包含在 string 中，如果 beg 和 end 指定范围，则检查是否包含在指定范围内，如果是返回开始的索引值，否则返回-1
string.format()	格式化字符串
string.index(str, beg=0, end=len(string))	跟find()方法一样，只不过如果str不在 string中会报一个异常.
string.isalnum()	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
string.isalpha()	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
string.isdecimal()	如果 string 只包含十进制数字则返回 True 否则返回 False.
string.isdigit()	如果 string 只包含数字则返回 True 否则返回 False.
string.islower()	如果 string 中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是小写，则返回

string.islower()	True，否则返回 False
string.isnumeric()	如果 string 中只包含数字字符，则返回 True，否则返回 False
string.isspace()	如果 string 中只包含空格，则返回 True，否则返回 False.
string.istitle()	如果 string 是标题化的(见 title())则返回 True，否则返回 False
string.isupper()	如果 string 中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是大写，则返回 True，否则返回 False
string.join(seq)	以 string 作为分隔符，将 seq 中所有的元素(的字符串表示)合并为一个新的字符串
string.ljust(width)	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
string.lower()	转换 string 中所有大写字符为小写.
string.lstrip()	截掉 string 左边的空格
string.maketrans(intab, outtab])	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简单的调用方式，第一个参数是字符串，表示需要转换的字符，第二个参数也是字符串表示转换的目标。
max(str)	返回字符串 str 中最大的字母。
min(str)	返回字符串 str 中最小的字母。
string.partition(str)	有点像 find()和 split()的结合体,从 str 出现的第一个位置起,把字符串 string 分成一个 3 元素的元组(string_pre_str,str,string_post_str),如果 string 中不包含str 则 string_pre_str == string.
string.replace(str1, str2, num=string.count(str1))	把 string 中的 str1 替换成 str2,如果 num 指定，则替换不超过 num 次.
string.rfind(str, beg=0,end=len(string))	类似于 find()函数，不过是从右边开始查找.
string.rindex(str, beg=0,end=len(string))	类似于 index()，不过是从右边开始.
string.rjust(width)	返回一个原字符串右对齐,并使用空格填充至长度 width 的新字符串
string.rpartition(str)	类似于 partition()函数,不过是从右边开始查找.
string.rstrip()	删除 string 字符串末尾的空格.
string.split(str="", num=string.count(str))	以 str 为分隔符切片 string，如果 num有指定值，则仅分隔 num 个子字符串
string.splitlines([keepends])	按照行('\r', '\r\n', \n')分隔，返回一个包含各行作为元素的列表，如果参数 keepends 为 False，不包含换行符，如果为 True，则保留换行符。
string.startswith(obj, beg=0,end=len(string))	检查字符串是否是以 obj 开头，是则返回 True，否则返回 False。如果beg 和 end 指定值，则在指定范围内检查.
string.strip([obj])	在 string 上执行 lstrip()和 rstrip()
string.swapcase()	翻转 string 中的大小写
string.title()	返回"标题化"的 string,就是说所有单词都是以大写开始，其余字母均为小写(见 istitle())
string.translate(str, del="")	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
string.upper()	转换 string 中的小写字母为大写
string.zfill(width)	返回长度为 width 的字符串，原字符串 string 右对齐，前面填充0

Python 列表(List)

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 :
2001
```

删除列表元素

可以使用 del 语句来删除列表的的元素，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元素 |L[1:] |['Spam', 'SPAM!'] |从第二个元素开始截取列表

Python列表函数&方法

Python包含以下函数:

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

Python包含以下方法:

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

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:

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

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；读取倒数第二个元素
L[1:]	('Spam', 'SPAM!')	截取元素

无关闭分隔符

任意无符号的对象，以逗号隔开，默认为元组，如下实例：

```
#!/usr/bin/python

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

以上实例运行结果：

```
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	描述
1	cmp(tuple1, tuple2)	比较两个元组元素。

2	len(tuple)	计算元组元素个数。
3	max(tuple)	返回元组中元素最大值。
4	min(tuple)	返回元组中元素最小值。
5	tuple(seq)	将列表转换为元组。

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。 字典的每个键值(key=>value)对用冒号(:)分割，每个对之间用逗号(,)分割，整个字典包括在花括号({})中 ,格式如下所示：

```
d = {key1 : value1, key2 : value2 }
```

键必须是唯一的，但值则不必。 值可以取任何数据类型，但键必须是不可变的，如字符串，数字或元组。 一个简单的字典实例：

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

也可如此创建字典：

```
dict1 = { 'abc': 456 };
dict2 = { 'abc': 123, 98.6: 37 };
```

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['Alice']:
Traceback (most recent call last):
  File "test.py", line 5, 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：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

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

del dict['Name']; # 删除键是'Name'的条目
dict.clear();     # 清空词典所有条目
del dict ;        # 删除词典

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。

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

Python字典包含了以下内置方法：

序号	函数	描述
1	dict.clear()	删除字典内所有元素
2	dict.copy()	返回一个字典的浅复制
3	dict.fromkeys(seq[, val]))	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	dict.get(key, default=None)	返回指定键的值，如果值不在字典中返回default值
5	dict.has_key(key)	如果键在字典dict里返回true，否则返回false
6	dict.items()	以列表返回可遍历的(键, 值) 元组数组
7	dict.keys()	以列表返回一个字典所有的键
8	dict.setdefault(key, default=None)	和get()类似, 但如果键不存在于字典中，将会添加键并将值设为default
9	dict.update(dict2)	把字典dict2的键/值对更新到dict里
10	dict.values()	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功能。 Python 提供了一个 time 和 calendar 模块可以用于格式化日期和时间。 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 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()用于获取当前时间戳, 如下实例:

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

import time;  # 引入time模块

ticks = time.time()
print "当前时间戳为:", ticks
```

以上实例输出结果：

```
当前时间戳为: 1459994552.51
```

时间戳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。

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:

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	0到59
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, -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, -1是决定是否为夏令时的旗帜

获取当前时间

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

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

```
# -*- coding: UTF-8 -*-

import time

localtime = time.localtime(time.time())
print "本地时间为 :", localtime
```

以上实例输出结果：

```
本地时间为 : time.struct_time(tm_year=2016, tm_mon=4, tm_mday=7, tm_hour=10, tm_min=3, tm_sec=27, tm_wday=3, tm_yday=98, tm_isdst=0)
```

获取格式化的时间

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

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

import time

localtime = time.asctime( time.localtime(time.time()) )
print "本地时间为 :", localtime
```

以上实例输出结果：

```
本地时间为 : Thu Apr  7 10:05:21 2016
```

格式化日期

我们可以使用 time 模块的 strftime 方法来格式化日期，：

```
time.strftime(format[, t])
```

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

import time

# 格式化成2016-03-20 11:45:39形式
print time.strftime("%Y-%m-%d %H:%M:%S", time.localtime())

# 格式化成Sat Mar 28 22:24:24 2016形式
print time.strftime("%a %b %d %H:%M:%S %Y", time.localtime())

# 将格式字符串转换为时间戳
a = "Sat Mar 28 22:24:24 2016"
print time.mktime(time.strptime(a,"%a %b %d %H:%M:%S %Y"))
```

以上实例输出结果：

```
2016-04-07 10:25:09
Thu Apr 07 10:25:09 2016
1459175064.0
```

python中时间日期格式化符号：

- %y 两位数的年份表示（00-99）
- %Y 四位数的年份表示（000-9999）
- %m 月份（01-12）
- %d 月内中的一天（0-31）
- %H 24小时制小时数（0-23）
- %I 12小时制小时数（01-12）
- %M 分钟数（00=59）
- %S 秒（00-59）
- %a 本地简化星期名称
- %A本地完整星期名称
- %b 本地简化的月份名称

- %B 本地完整的月份名称
- %c 本地相应的日期表示和时间表示
- %j 年内的一天（001-366）
- %p 本地A.M.或P.M.的等价符
- %U 一年中的星期数（00-53）星期天为星期的开始
- %w 星期（0-6），星期天为星期的开始
- %W 一年中的星期数（00-53）星期一为星期的开始
- %x 本地相应的日期表示
- %X 本地相应的时间表示
- %Z 当前时区的名称
- %% %号本身

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
# -*- coding: UTF-8 -*-

import calendar

cal = calendar.month(2016, 1)
print "以下输出2016年1月份的日历:"
print cal;
```

以上实例输出结果：

```
以下输出2016年1月份的日历:
    January 2016
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	描述
1	time.altzone	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）。对夏令时启用地区才能使用。
2	time.asctime([tupletime])	接受时间元组并返回一个可读的形式为"Tue Dec 11 18:07:14 2008"（2008年12月11日 周二18时07分14秒）的24个字符的字符串。
3	time.clock( )	用以浮点数计算的秒数返回当前的CPU时间。用来衡量不同程序的耗时，比time.time()更有用。
4	time.ctime([secs])	作用相当于asctime(localtime(secs))，未给参数相当于asctime()
5	time.gmtime([secs])	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_isdst始终为0
6	time.localtime([secs])	接收时间辍（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，取决于当地当时是不是夏令时）。
7	time.mktime(tupletime)	接受时间元组并返回时间辍（1970纪元后经过的浮点秒数）。
8	time.sleep(secs)	推迟调用线程的运行，secs指秒数。
9	time.strftime(fmt[,tupletime])	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	time.strptime(str,fmt='%a %b %d %H:%M:%S %Y')	根据fmt的格式把一个时间字符串解析为时间元组。
11	time.time( )	返回当前时间的时间戳（1970纪元后经过的浮点秒数）。
12	time.tzset()	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	描述
1	time.timezone	属性time.timezone是当地时区（未启动夏令时）距离格林威治的偏移秒数（>0，美洲;<=0大部分欧洲，亚洲，非洲）。
2	time.tzname	属性time.tzname包含一对根据情况的不同而不同的字符串，分别是带夏令时的本地时区名称，和不带的。

日历（**Calendar**）模块

此模块的函数都是日历相关的，例如打印某月的字符月历。

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

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

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 [**表达式**]** 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：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

# 定义函数
def printme( str ):
   "打印任何传入的字符串"
   print str;
   return;

# 调用函数
printme("我要调用用户自定义函数!");
printme("再次调用同一函数");
```

以上实例输出结果：

```
我要调用用户自定义函数!
再次调用同一函数
```

参数传递

在 python 中，类型属于对象，变量是没有类型的：

```
a=[1,2,3]
```

```
a="Runoob"
```

以上代码中，[1,2,3] 是 List 类型，"Runoob" 是 String 类型，而变量 a 是没有类型，她仅仅是一个对象的引用（一个指针），可以是 List 类型对象，也可以指向 String 类型对象。

可更改(mutable)与不可更改(immutable)对象

在 python 中，strings, tuples, 和 numbers 是不可更改的对象，而 list,dict 等则是可以修改的对象。

- 不可变类型：变量赋值 a=5 后再赋值 a=10，这里实际是新生成一个 int 值对象 10，再让 a 指向它，而 5 被丢弃，不是改变a的值，相当于新生成了a。
- 可变类型：变量赋值 la=[1,2,3,4] 后再赋值 la[2]=5 则是将 list la 的第三个元素值更改，本身la没有动，只是其内部的一部分值被修改了。

python 函数的参数传递：

- 不可变类型：类似 c++ 的值传递，如 整数、字符串、元组。如fun（a），传递的只是a的值，没有影响a对象本身。比如在 fun（a）内部修改 a 的值，只是修改另一个复制的对象，不会影响 a 本身。
- 可变类型：类似 c++ 的引用传递，如 列表，字典。如 fun（la），则是将 la 真正的传过去，修改后fun外部的la也会受影响

python 中一切都是对象，严格意义我们不能说值传递还是引用传递，我们应该说传不可变对象和传可变对象。

python 传不可变对象实例

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

def ChangeInt( a ):
    a = 10

b = 2
ChangeInt(b)
print b # 结果是 2
```

实例中有 int 对象 2，指向它的变量是 b，在传递给 ChangeInt 函数时，按传值的方式复制了变量 b，a 和 b 都指向了同一个 Int 对象，在 a=10 时，则新生成一个 int 值对象 10，并让 a 指向它。

传可变对象实例

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

# 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
```

实例中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
- 关键字参数
- 默认参数
- 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：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

#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)
```

关键字参数

关键字参数和函数调用关系紧密，函数调用使用关键字参数来确定传入的参数值。 使用关键字参数允许函数调用时参数的顺序与声明时不一致，因为 Python 解释器能够用参数名匹配参数值。 以下实例在函数 printme() 调用时使用参数名：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

#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
```

下例能将关键字参数顺序不重要展示得更清楚：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

#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：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
```

```
#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：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

# 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

python 使用 lambda 来创建匿名函数。

- lambda只是一个表达式，函数体比def简单很多。
- lambda的主体是一个表达式，而不是一个代码块。仅仅能在lambda表达式中封装有限的逻辑进去。
- lambda函数拥有自己的命名空间，且不能访问自有参数列表之外或全局命名空间里的参数。
- 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：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

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

# 调用sum函数
print "相加后的值为 : ", sum( 10, 20 )
print "相加后的值为 : ", sum( 20, 20 )
```

以上实例输出结果：

```
相加后的值为 :  30
相加后的值为 :  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：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

# 可写函数说明
def sum( arg1, arg2 ):
   # 返回2个参数的和."
   total = arg1 + arg2
   print "函数内 : ", total
   return total;

# 调用sum函数
total = sum( 10, 20 );
```

以上实例输出结果：

```
函数内 :  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

全局变量和局部变量

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：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

total = 0; # 这是一个全局变量
# 可写函数说明
def sum( arg1, arg2 ):
   #返回2个参数的和."
   total = arg1 + arg2; # total在这里是局部变量.
   print "函数内是局部变量 : ", total
   return total;

#调用sum函数
sum( 10, 20 );
print "函数外是全局变量 : ", total
```

以上实例输出结果：

```
函数内是局部变量 :  30
函数外是全局变量 :  0
```

Python 模块

Python 模块(Module)，是一个 Python 文件，以 .py 结尾，包含了 Python 对象定义和Python语句。 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。 模块能定义函数，类和变量，模块里也能包含可执行的代码。

例子 下例是个简单的模块 support.py：

support.py 模块：

```
def print_func( par ):
   print "Hello : ", par
   return
```

import 语句

模块的引入

模块定义好后，我们可以使用 import 语句来引入模块，语法如下：

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

比如要引用模块 math，就可以在文件最开始的地方用 import math 来引入。在调用 math 模块中的函数时，必须这样引用：

```
模块名.函数名
```

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 support.py，需要把命令放在脚本的顶端：

test.py 文件代码：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

# 导入模块
import support

# 现在可以调用模块里包含的函数了
support.print_func("Runoob")
```

以上实例输出结果：

```
Hello : Runoob
```

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 fibonacci 函数，使用如下语句：

```
from fib import fibonacci
```

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

From…import* 语句

把一个模块的所有内容全都导入到当前的命名空间也是可行的，只需使用如下声明：

```
from modname import *
```

这提供了一个简单的方法来导入一个模块中的所有项目。然而这种声明不该被过多地使用。 例如我们想一次性引入 math 模块中所有的东西，语句如下：

```
from math import *
```

搜索路径

当你导入一个模块，Python 解析器对模块位置的搜索顺序是：

- 1、当前目录
- 2、如果不在当前目录，Python 则搜索在 shell 变量 PYTHONPATH 下的每个目录。
- 3、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:\python27\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。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

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：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

# 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。 简单来说，包就是文件夹，但该文件夹下必须存在 `__init__.py` 文件, 该文件的内容可以为空。 `__int__.py` 用于标识当前文件夹是一个包。 考虑一个在 package*runoob* 目录下的 *runoob1.py*、*runoob2.py*、`*_init*.py` 文件，test.py 为测试调用包的代码，目录结构如下：

```
test.py
package_runoob
|-- __init__.py
|-- runoob1.py
|-- runoob2.py
```

源代码如下：

package_runoob/runoob1.py

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

def runoob1():
   print "I'm in runoob1"
```

package_runoob/runoob2.py

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

def runoob2():
   print "I'm in runoob2"
```

现在，在 package*runoob* 目录下创建 `*_init*.py`：

```
package_runoob/__init__.py
#!/usr/bin/python
# -*- coding: UTF-8 -*-

if __name__ == '__main__':
```

```
    print '作为主程序运行'
else:
    print 'package_runoob 初始化'
```

然后我们在 package_runoob 同级目录下创建 test.py 来调用 package_runoob 包 test.py

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

# 导入 Phone 包
from package_runoob.runoob1 import runoob1
from package_runoob.runoob2 import runoob2

runoob1()
runoob2()
```

以上实例输出结果：

```
package_runoob 初始化
I'm in runoob1
I'm in runoob2
```

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
# -*- coding: UTF-8 -*-

print "Python 是一个非常棒的语言，不是吗？";
```

你的标准屏幕上会产生以下结果： Python 是一个非常棒的语言，不是吗？

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
# -*- coding: UTF-8 -*-

str = raw_input("请输入：");
print "你输入的内容是: ", 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：

```
请输入：Hello Python!
你输入的内容是:  Hello Python!
```

input函数

input([prompt]) 函数和 raw_input([prompt]) 函数基本类似，但是 input 可以接收一个Python表达式作为输入，并将运算结果返回。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

str = input("请输入：");
print "你输入的内容是: ", str
```

这会产生如下的对应着输入的结果：

```
请输入：[x*5 for x in range(2,10,2)]
你输入的内容是:  [10, 20, 30, 40]
```

打开和关闭文件

现在，您已经可以向标准输入和输出进行读写。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，创建新文件。
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，新的内容将会被写入到已有内容之后。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，新的内容将会被写入到已有内容之后。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。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。如果该文件不存在，创建新文件用于读写。

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

如下实例：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

# 打开一个文件
fo = open("foo.txt", "wb")
print "文件名: ", fo.name
print "是否已关闭 : ", fo.closed
print "访问模式 : ", fo.mode
print "末尾是否强制加空格 : ", fo.softspace
```

以上实例输出结果：

```
文件名:  foo.txt
是否已关闭 :  False
访问模式 :  wb
```

```
末尾是否强制加空格 :  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：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

# 打开一个文件
fo = open("foo.txt", "wb")
print "文件名: ", fo.name

# 关闭打开的文件
fo.close()
```

以上实例输出结果：

```
文件名:  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()方法不会在字符串的结尾添加换行符('\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：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

# 打开一个文件
fo = open("foo.txt", "wb")
fo.write( "www.runoob.com!\nVery good site!\n");

# 关闭打开的文件
fo.close()
```

上述方法会创建foo.txt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:

```
$ cat foo.txt
www.runoob.com!
Very good site!
```

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：

这里我们用到以上创建的 foo.txt 文件。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
```

```
# 打开一个文件
fo = open("foo.txt", "r+")
str = fo.read(10);
print "读取的字符串是 : ", str
# 关闭打开的文件
fo.close()
```

以上实例输出结果：

```
读取的字符串是 :  www.runoob
```

文件位置：

文件定位

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。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

# 打开一个文件
fo = open("foo.txt", "r+")
str = fo.read(10);
print "读取的字符串是 : ", str

# 查找当前位置
position = fo.tell();
print "当前文件位置 : ", 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 "重新读取字符串 : ", str
# 关闭打开的文件
fo.close()
```

以上实例输出结果：

```
读取的字符串是 :  www.runoob
当前文件位置 :  10
重新读取字符串 :  www.runoob
```

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，然后才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。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

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。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

import os

# 删除一个已经存在的文件test2.txt
os.remove("test2.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。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

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。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

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：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

import os

# 给出当前的目录
print 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。

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

import os

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

文件、目录相关的方法

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

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

Python File(文件) 方法

file 对象使用 open 函数来创建，下表列出了 file 对象常用的函数：

序号	方法	描述
1	file.close()	关闭文件。关闭后文件不能再进行读写操作。
2	file.flush()	刷新文件内部缓冲，直接把内部缓冲区的数据立刻写入文件, 而不是被动的等待输出缓冲区写入。
3	file.fileno()	返回一个整型的文件描述符(file descriptor FD 整型), 可以用在如os模块的read方法等一些底层操作上。
4	file.isatty()	如果文件连接到一个终端设备返回 True，否则返回 False。
5	file.next()	返回文件下一行。
6	file.read([size])	从文件读取指定的字节数，如果未给定或为负则读取所有。
7	file.readline([size])	读取整行，包括 "\n" 字符。
8	file.readlines([sizehint])	读取所有行并返回列表，若给定sizeint>0，返回总和大约为sizeint字节的行, 实际读取值可能比sizhint较大, 因为需要填充缓冲区。
9	file.seek(offset[, whence])	设置文件当前位置
10	file.tell()	返回文件当前位置。
11	file.truncate([size])	截取文件，截取的字节通过size指定，默认为当前文件位置。
12	file.write(str)	将字符串写入文件，没有返回值。
13	file.writelines(sequence)	向文件写入一个序列字符串列表，如果需要换行则要自己加入每行的换行符。

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...else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
# -*- coding: UTF-8 -*-

try:
    fh = open("testfile", "w")
    fh.write("这是一个测试文件，用于测试异常!!")
except IOError:
    print "Error: 没有找到文件或读取文件失败"
else:
    print "内容写入文件成功"
    fh.close()
```

以上程序输出结果：

```
$ python test.py
内容写入文件成功
$ cat testfile       # 查看写入的内容
这是一个测试文件，用于测试异常!!
```

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
# -*- coding: UTF-8 -*-

try:
    fh = open("testfile", "w")
    fh.write("这是一个测试文件，用于测试异常!!")
except IOError:
    print "Error: 没有找到文件或读取文件失败"
else:
    print "内容写入文件成功"
    fh.close()
```

在执行代码前为了测试方便，我们可以先去掉 testfile 文件的写权限，命令如下：

```
chmod -w testfile
```

再执行以上代码：

```
$ python test.py
Error: 没有找到文件或读取文件失败
```

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:
    正常的操作
   ......................
except:
    发生异常，执行这块代码
   ......................
else:
    如果没有异常执行这块代码
```

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:
    正常的操作
   ......................
except(Exception1[, Exception2[,...ExceptionN]]):
   发生以上多个异常中的一个，执行这块代码
   ......................
else:
    如果没有异常执行这块代码
```

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
```

实例

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

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

```
    fh.write("这是一个测试文件，用于测试异常!!")
finally:
    print "Error: 没有找到文件或读取文件失败"
```

如果打开的文件没有可写权限，输出如下所示：

```
$ python test.py
Error: 没有找到文件或读取文件失败
```

同样的例子也可以写成如下方式：

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

try:
    fh = open("testfile", "w")
    try:
        fh.write("这是一个测试文件，用于测试异常!!")
    finally:
        print "关闭文件"
        fh.close()
except IOError:
    print "Error: 没有找到文件或读取文件失败"
```

当在try块中抛出一个异常，立即执行finally块代码。 finally块中的所有语句执行后，异常被再次触发，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:
    正常的操作
   ......................
except ExceptionType, Argument:
    你可以在这输出 Argument 的值...
```

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
# -*- coding: UTF-8 -*-

# 定义函数
def temp_convert(var):
    try:
        return int(var)
    except ValueError, Argument:
        print "参数没有包含数字\n", Argument

# 调用函数
temp_convert("xyz");
```

以上程序执行结果如下：

```
$ python test.py
参数没有包含数字
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 Exception("Invalid level!", level)
        # 触发异常后，后面的代码就不会再执行
```

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:
    正常逻辑
except "Invalid level!":
    触发自定义异常
else:
    其余代码
```

实例

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-

# 定义函数
def mye( level ):
    if level < 1:
        raise Exception("Invalid level!", level)
        # 触发异常后，后面的代码就不会再执行

try:
    mye(0)               // 触发异常
except "Invalid level!":
    print 1
else:
    print 2
```

执行以上代码，输出结果为：

```
$ python test.py
Traceback (most recent call last):
  File "test.py", line 11, in <module>
    mye(0)
  File "test.py", line 7, in mye
    raise Exception("Invalid level!", level)
Exception: ('Invalid level!', 0)
```

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
```