**目录**

[Python的特色 5](#_Toc482720354)

[有两种使用Python运行你的程序的方式 6](#_Toc482720355)

[1.使用交互式的带提示符的解释器 6](#_Toc482720356)

[2.使用源文件 6](#_Toc482720357)

[Python命令行选项 6](#_Toc482720358)

[专门为Python设计的IDE软件： 7](#_Toc482720359)

[系统环境设置 8](#_Toc482720360)

[设置环境变量 8](#_Toc482720361)

[判断操作系统 8](#_Toc482720362)

[查询装载模块 9](#_Toc482720363)

[easy\_install 10](#_Toc482720364)

[版本升级2.x to 3.x 11](#_Toc482720365)

[部分函数和语句的改变 11](#_Toc482720366)

[2 to 3 问题 12](#_Toc482720367)

[语法 12](#_Toc482720368)

[注释 12](#_Toc482720369)

[数据类型 13](#_Toc482720370)

[标识符的命名 14](#_Toc482720371)

[逻辑行与物理行 14](#_Toc482720372)

[语法规则 16](#_Toc482720373)

[1.缩进规则 16](#_Toc482720374)

[2.变量没有类型 16](#_Toc482720375)

[3.单语句块 16](#_Toc482720376)

[运算符与表达式: 16](#_Toc482720377)

[运算符优先级 19](#_Toc482720378)

[计算顺序 21](#_Toc482720379)

[结合规律 21](#_Toc482720380)

[字符串 21](#_Toc482720381)

[控制台输入 26](#_Toc482720382)

[1.输入字符串 26](#_Toc482720383)

[2.输入整数 26](#_Toc482720384)

[3.输入浮点型 26](#_Toc482720385)

[4.输入16进制数据 26](#_Toc482720386)

[5.输入8进制数据 26](#_Toc482720387)

[控制流: 27](#_Toc482720388)

[if 语句 27](#_Toc482720389)

[while 语句 28](#_Toc482720390)

[for 循环 29](#_Toc482720391)

[break 语句 30](#_Toc482720392)

[continue 语句 30](#_Toc482720393)

[函数: 31](#_Toc482720394)

[定义函数 31](#_Toc482720395)

[函数形参 32](#_Toc482720396)

[局部变量 32](#_Toc482720397)

[global语句 33](#_Toc482720398)

[默认参数值 33](#_Toc482720399)

[关键参数 34](#_Toc482720400)

[return语句 35](#_Toc482720401)

[None 35](#_Toc482720402)

[DocStrings 36](#_Toc482720403)

[函数属性 func\_\* 37](#_Toc482720404)

[模块: 38](#_Toc482720405)

[sys模块(标准库模块) 38](#_Toc482720406)

[字节编译的.pyc文件 40](#_Toc482720407)

[from ... import 语句 40](#_Toc482720408)

[import ... as 40](#_Toc482720409)

[模块的 \_\_name\_\_ 40](#_Toc482720410)

[创建自己的模块 41](#_Toc482720411)

[dir()函数 43](#_Toc482720412)

[数据结构 44](#_Toc482720413)

[列表(list) 44](#_Toc482720414)

[元组 46](#_Toc482720415)

[字典 47](#_Toc482720416)

[序列 49](#_Toc482720417)

[参考 50](#_Toc482720418)

[列表综合 51](#_Toc482720419)

[集合 52](#_Toc482720420)

[面向对象的编程 54](#_Toc482720421)

[self参数 54](#_Toc482720422)

[类的变量和对象的变量 55](#_Toc482720423)

[继承 57](#_Toc482720424)

[特殊的方法 58](#_Toc482720425)

[try ... except (处理异常) 60](#_Toc482720426)

[使用 try ... except 语句来处理异常。 60](#_Toc482720427)

[try ... finally 60](#_Toc482720428)

[raise 语句 60](#_Toc482720429)

[生成器的 throw 方法 61](#_Toc482720430)

[lambda 形式 63](#_Toc482720431)

[exec 和 eval 64](#_Toc482720432)

[execfile 语句 65](#_Toc482720433)

[assert 语句 66](#_Toc482720434)

[repr 函数 66](#_Toc482720435)

[zip() and \_ 66](#_Toc482720436)

[正则表达式 67](#_Toc482720437)

[匹配符： 67](#_Toc482720438)

[元字符: 69](#_Toc482720439)

[无捕获组: 70](#_Toc482720440)

[命名组: 70](#_Toc482720441)

[松散正则表达式: 71](#_Toc482720442)

[match() vs search() 76](#_Toc482720443)

[Python标准库: 76](#_Toc482720444)

[范例: 77](#_Toc482720445)

[运行系统命令行和调用外部程序 77](#_Toc482720446)

[获取系统时间 78](#_Toc482720447)

[文件和目录操作 79](#_Toc482720448)

[创建目录 79](#_Toc482720449)

[获取当前路径 80](#_Toc482720450)

[获取特定路径下的文件名和子文件夹名 80](#_Toc482720451)

[复制文件或文件夹 81](#_Toc482720452)

[文件操作(读写txt文件) 81](#_Toc482720453)

[文件操作(遍历目录和文件名) 85](#_Toc482720454)

[文件操作(分割路径和文件名) 85](#_Toc482720455)

[储存器 86](#_Toc482720456)

[数据库连接 87](#_Toc482720457)

[编码问题 88](#_Toc482720458)

[得到系统当前编码 88](#_Toc482720459)

[中文字符串按字分开Python2.x 88](#_Toc482720460)

[判断是否为中文字符 88](#_Toc482720461)

[编码转换 89](#_Toc482720462)

[检测未知编码 91](#_Toc482720463)

[UnicodeDecodeError 91](#_Toc482720464)

[得到中文字符的GB2312编码 93](#_Toc482720465)

[转换文件编码 93](#_Toc482720466)

[Unicode介绍 93](#_Toc482720467)

[查看Python内部Unicode格式 93](#_Toc482720468)

[各场景下的编码 94](#_Toc482720469)

[Tkinter 94](#_Toc482720470)

[随机数 96](#_Toc482720471)

[睡眠 97](#_Toc482720472)

[扩展阅读 97](#_Toc482720473)

# Python的特色

1.简单

2.易学

3.免费、开源

4.高层语言: 封装内存管理等

5.可移植性: 程序如果避免使用依赖于系统的特性，那么无需修改就可以在任何平台上运行

6.解释性: 直接从源代码运行程序,不再需要担心如何编译程序,使得程序更加易于移植。

7.面向对象: 支持面向过程的编程也支持面向对象的编程。

8.可扩展性: 需要保密或者高效的代码，可以用C或C++编写，然后在Python程序中使用它们。

9.可嵌入性: 可以把Python嵌入C/C++程序，从而向你的程序用户提供脚本功能。

10.丰富的库: 包括正则表达式、文档生成、单元测试、线程、数据库、网页浏览器、CGI、FTP、

电子邮件、XML、XML-RPC、HTML、WAV文件、密码系统、GUI（图形用户界面）、Tk和其他与系统有关的操作。

除了标准库以外，还有许多其他高质量的库，如wxPython、Twisted和Python图像库等等。

11.概括: Python确实是一种十分精彩又强大的语言。它合理地结合了高性能与使得编写程序简单有趣的特色。

12.规范的代码: Python采用强制缩进的方式使得代码具有极佳的可读性。

## 有两种使用Python运行你的程序的方式

### 1.使用交互式的带提示符的解释器

直接双击运行“python.exe”，在里面输入内容，如： print 'haha...'

### 2.使用源文件

在Python的安装目录下，建一个批处理(test.bat)，写入：

@echo off

python.exe test.py

pause

而“test.py”里面的内容是需要执行的程序

## Python命令行选项

选项 作用

-c cmd 在命令行直接执行python代码。如python -c 'print "hello world"'。

-d 脚本编译后从解释器产生调试信息。同PYTHONDEBUG=1。

-E 忽略环境变量。

-h 显示python命令行选项帮助信息。

-i 脚本执行后马上进入交互命令行模式。同PYTHONINSPECT=1。

-O 在执行前对解释器产生的字节码进行优化。同 PYTHONOPTIMIZE=1。

-OO 在执行前对解释器产生的字节码进行优化，并删除优化代码中的嵌入式文档字符串。

-Q arg 除法规则选项，-Qold(default)，-Qwarn，-Qwarnall，-Qnew。

-S 解释器不自动导入site.py模块。

-t 当脚本的tab缩排格式不一致时产生警告。

-u 不缓冲stdin、stdout和stderr，默认是缓冲的。同PYTHONUNBUFFERED=1。

-v 产生每个模块的信息。如果两个-v选项，则产生更详细的信息。同PYTHONVERBOSE=x。

-V 显示Python的版本信息。

-W arg 出错信息控制。(arg is action:message:category:module:lineno)

-x 忽略源文件的首行。要在多平台上执行脚本时有用。

file 执行file里的代码。

- 从stdin里读取执行代码。

## 专门为Python设计的IDE软件：

　　● IDLE：Python“标准”IDE

　　● Komodo和Komodo Edit：后者是前者的免费精简版

　　● PythonWin：ActivePython软件包的IDE，仅适用于Windows

　　● SPE（Stani's Python Editor）：功能较多的自由软件，依赖wxPython

　　● Ulipad：功能较全的自由软件，依赖wxPython；作者是中国Python高手limodou

　　● WingIDE：可能是功能最全的IDE，但不是自由软件

　　● Eric：基于PyQt的自由软件，功能强大

　　● 有相应插件的通用IDE软件（使用难度往往较大）：

　　● eclipse + pydev插件：稳定性欠佳

　　● emacs + 插件

　　● Vim + 插件

　　● Visual Studio .NET 2003 + VisualPython：仅适用Windows，已停止维护，功能较差

　　● SlickEdit

　　● TextMate

　　● Netbeans IDE

　　另外，诸如 EditPlus、UltraEdit、PSPad 等通用的程序员文本编辑器软件也能对Python代码文本作出一定解释，

但是否够得上集成开发环境的水平，尚有待评估。

## 系统环境设置

### 设置环境变量

假设python安装在C:\Python25目录下，设置环境变量方法如下：

方法一、我的电脑->属性->高级->环境变量->系统变量

在系统变量里找到PATH，双击PATH，在结尾加上 ";C:\Python25"(不要引号)

方法二、运行->cmd

输入set PATH=%PATH%;C:\Python25

接下来，再在当前的 cmd下输入python，即可运行

### 判断操作系统

import **sys**  
print(sys.platform)

### 查询装载模块

**1，pip安装一个模块**

pip install whatever.whl

**2，查询一个模块是否安装成功。**

看自己有没有安装好，最简单的办法在可以再控制台下:

C:\Users\sony>python

Python 2.7.6 (default, Nov 10 2013, 19:24:18) [MSC v.1500 32 bit (Intel)] on win

32

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

>>> import numpy

>>> import scipy

>>> import matplotlib

>>> import wrong\_package

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ImportError: No module named wrong\_package

如果import报错，一般就是没装上，前面三个都是安装成功了。

**3，只列出用户自己安装的模块。**

import **pip**

installed\_packages = pip.get\_installed\_distributions()

installed\_packages\_list = sorted(["%s==%s" % (i.key, i.version) for i in installed\_packages])

print(installed\_packages\_list)

**4，列出所有可供import的module，包括系统本身包含的模块。**

可以这样调用help函数，然后再help控制台里输入modules（整个过程必须在交互式界面下完成，无法写成脚本）

>>> help()

help> modules

稍等一会后，就会列出所有的模块

**5，列出当前程序已经import导入的模块。**

import sys

print(sys.modules.keys())

**6，import模块时，列出系统所有搜索路径。**

import sys

print(sys.path)

### easy\_install

安装 easy\_install

wget -q http://peak.telecommunity.com/dist/ez\_setup.py  
python ez\_setup.py

安装python包

sudo easy\_install <包名>

升级python包

sudo easy\_install -U <包名>

删除python包

sudo easy\_install -m <包名>

# 版本升级2.x to 3.x

python3.0版本较之前的有很大变动，而且不向下兼容。

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。即2.6版本兼容2.x和3.0的语法

Python 2.6保持了对之前版本的全兼容，而且还包含了Python 3.0的新玩意(一些新特性需要通过“from \_\_future\_\_ import”来启用)。

如，在Python2.6要使用3.0的打印,得写上“ from \_\_future\_\_ import print\_function”

基于早期Python版本而能正常运行于Python 2.6并无警告的程序可以通过一个2 to 3的转换工具无缝迁移到Python 3.0。

## 部分函数和语句的改变

最引人注意的改变是print语句没有了，取而代之的是print函数

同样的还有exec语句，已经改为exec()函数。去除了<>，全部改用!=。

在python2.x版本中

#!/usr/bin/env python

# 或者上句写: #!/usr/bin/python

print "Hello, world!"

或者：

import sys

sys.stdout.write("Hello, world\n")

在python3.x中

print('Hello world!')

用迭代器来替代列表

一些知名的API将不再返回列表。

而字典的dict.iterkeys()、dict.itervalues()和dict.iteritems()方法将会移除，而你可以使用.keys()、.values()和.items()，它们会返回更轻量级的、类似于集合的容器对象，而不是返回一个列表来复制键值。

这样做的优点是，可以直接在键和条目上进行集合操作，而不需要再复制一次。

整型数

移除了含糊的除法符号('/')，而只返回浮点数。

在以前的版本中，如果参数是int或者是long的话，就会返回相除后结果的向下取整（floor），而如果参数是float或者是complex的话，那么就会返回相除后结果的一个恰当的近似。

在2.6版本中可以通过from \_\_future\_\_ import division来启用这项特性。

## 2 to 3 问题

1.print 语句

2.x 3.x 说明

① print print() # 输出一个空白行

② print 1 print(1) # 输出一个单独的值

③ print 1, 2 print(1, 2) # 输出多个值，以空格分割

④ print 1, 2, print(1, 2, end=' ') # 输出时取消在末尾输出回车符。

⑤ print >>sys.stderr, 1, 2 print(1, 2, file=sys.stderr) # 把输出重定向到一个管道

# 语法

## 注释

“#”后面的内容

## 数据类型

共4种: 整数、长整数、浮点数和复数。

1.整数,如:2

2.长整数,如:22L # 长整数不过是大一些的整数。Python 3已经取消这种类型,被int取代了。

3.浮点数,如:3.23 和 52.3E-4 # E标记表示10的幂。在这里，52.3E-4表示52.3 \* 10-4。

4.复数,如:(-5+4j) 和 (2.3-4.6j)

在Python 2和Python 3的变化:

1.八进制(octal)数:

Python 2: x = 0755 # 0开头

Python 3: x = 0o755 # 0o开头

2.long 类型

Python 2有为非浮点数准备的 int 和 long 类型。 int 类型的最大值不能超过 sys.maxint,而且这个最大值是平台相关的。

整数可以通过在数字的末尾附上一个L来定义长整型，显然，它比 int 类型表示的数字范围更大。

Python 3里，只有一种整数类型 int,大多数情况下，它很像Python 2里的长整型。

由于已经不存在两种类型的整数，所以就没有必要使用特殊的语法去区别他们。

由于 long 类型在Python 3的取消,引起以下改变

Python 2 Python 3 说明

① x = 1000000000000L x = 1000000000000 # 十进制的普通整数

② x = 0xFFFFFFFFFFFFL x = 0xFFFFFFFFFFFF # 十六进制的普通整数

③ long(x) int(x) # long()函数没有了。可以使用int()函数强制转换一个变量到整型。

④ type(x) is long type(x) is int # 检查一个变量是否是整型

⑤ isinstance(x, long) isinstance(x, int) # 也可以使用 isinstance()函数来检查数据类型

3.sys.maxint(sys.maxsize)

由于长整型和整型被整合在一起了, sys.maxint常量不再精确。

因为这个值对于检测特定平台的能力还是有用处的，所以它被Python 3保留，并且重命名为 sys.maxsize。

Python 2 Python 3

① from sys import maxint from sys import maxsize # maxint变成了maxsize。

② a\_function(sys.maxint) a\_function(sys.maxsize) # 所有的sys.maxint都变成了sys.maxsize。

## 标识符的命名

变量是标识符的例子。 标识符 是用来标识 某样东西 的名字。在命名标识符的时候，你要遵循这些规则：

1.标识符的第一个字符必须是字母表中的字母（大写或小写）或者一个下划线（‘\_’）。

2.标识符名称的其他部分可以由字母（大写或小写）、下划线（‘\_’）或数字（0-9）组成。

3.标识符名称是对大小写敏感的。例如，myname和myName不是一个标识符。

有效 标识符名称的例子有i、\_\_my\_name、name\_23和a1b2\_c3。

无效 标识符名称的例子有2things、this is spaced out和my-name。

## 逻辑行与物理行

物理行是在编写程序时文本的一行。逻辑行是程序的一个语句。

Python假定每个 物理行 对应一个 逻辑行 。 他希望每行都只使用一个语句，这样使得代码更加易读。

1. 如果你想要在一个物理行中使用多于一个逻辑行，那么你需要使用分号（;）来特别地标明这种用法。

分号表示一个逻辑行/语句的结束。

如: i = 5; print i; # 强烈建议你坚持在每个物理行只写一句逻辑行。 让程序见不到分号，而更容易阅读。

2. 明确的行连接

在多个物理行中写一个逻辑行,行结尾用反斜杠标明

如: s = 'This is a string. \

This continues the string.'

# 上面这两行是一个逻辑行,打印是: This is a string. This continues the string.

print \

i

# 上面这两行也是一个逻辑行, 等同于: print i

3. 暗示的行连接

在多个物理行中写一个逻辑行,行结尾不需要使用反斜杠标明。

这种情况出现在逻辑行中使用了圆括号、方括号或波形括号的时候。

4. 缩进

行首的空白是重要的。在逻辑行首的空白（空格和tab符）用来决定逻辑行的缩进层次，从而用来决定语句的分组。

同一层次的语句必须有相同的缩进。每一组这样的语句称为一个块。

不要混合使用tab符和空格来缩进，因为这在跨越不同的平台的时候，无法正常工作。强烈建议只使用一种风格来缩进。

## 语法规则

### 1.缩进规则

一个模块的界限，完全是由每行的首字符在这一行的位置来决定的(而不是花括号{})。这一点曾经引起过争议。

不过不可否认的是，通过强制程序员们缩进，Python确实使得程序更加清晰和美观。

在逻辑行首的空白（空格和tab）用来决定逻辑行的缩进层次，从而用来决定语句的分组。错误的缩进会引发错误。

注意: 强制缩进的问题,最常见的情况是tab符和空格的混用会导致错误，而这是用肉眼无法分别的。

### 2.变量没有类型

使用变量时只需要给它们赋值。 不需要声明或定义数据类型。

### 3.单语句块

如果你的语句块只包含一句语句，那么你可以在条件语句或循环语句的同一行指明它。如: if 1!=2: print('Yes')

强烈建议不要使用这种缩略方法。这会破坏Python清晰美观的代码风格,违背设计者的初衷。

如果是在Python解释器输入,它的把提示符会改变为...以表示语句还没有结束。这时按回车键用来确认语句已经完整了。然后，Python完成整个语句的执行，并且返回原来的提示符来等待下一句输入。

## 运算符与表达式:

运算符

运算符 名称 说明

+ 加 两个对象相加,也可以字符串拼接

- 减 得到负数或是一个数减去另一个数

\* 乘 两个数相乘 或是返回一个被重复若干次的字符串

\*\* 幂 返回x的y次幂

/ 除 x除以y

// 取整除 返回商的整数部分

% 取模 返回除法的余数 # 8%3得到2。 -25.5%2.25得到1.5

<< 左移 把一个数的二进制向左移一定数目 # 2 << 2得到8

>> 右移 把一个数的二进制向右移一定数目 # 11 >> 1得到5

& 按位与 数的按位与 # 5 & 3得到1。

| 按位或 数的按位或 # 5 | 3得到7。

^ 按位异或 数的按位异或 # 5 ^ 3得到6

~ 按位翻转 x的按位翻转是-(x+1) # ~5得到6。

< 小于 返回x是否小于y

> 大于 返回x是否大于y

<= 小于等于 返回x是否小于等于y

>= 大于等于 返回x是否大于等于y

== 等于 比较对象是否相等

!= 不等于 比较两个对象是否不相等(python3删除了“<>”符号)

not 布尔“非” 如果x为True，返回False。如果x为False，它返回True。 # x = True; not x返回False。

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。 # x=False; y=True; x and y返回False。

or 布尔“或” 如果x是True，它返回True，否则它返回y的计算值。# x = True; y = False; x or y返回True。

说明:

1.加号“+”:有数学相加，也有字符串拼接作用,注意:不能字符串和数字相加。如: 3 + 5得到8; 'a' + 'b'得到'ab'。

2.乘号“\*”:两个数相乘，也可以把字符串重复拼接若干次,如: 2 \* 3得到6; 'la' \* 3得到'lalala'。

3.幂“\*\*” :这种写法,其他语言好像没见到过,如: 3 \*\* 4得到81(即3 \* 3 \* 3 \* 3)

4.除号“/”:整数的除法得到整数结果,浮点数的得到浮点数,如:4/3得到1(返回相除后结果的向下取整(floor)); 4.0/3或4/3.0得到1.333...

注意:Python 3.0开始,移除了含糊的除法符号（'/'），而只返回浮点数。如:4/3得到1.333...

5.取整除“//”:将两数相除,然后对结果取整,如: 7 // 3得到2; 4 // 3.0得到1.0

6.比较运算符: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。

如果两个操作数都是数字，它们首先被转换为一个共同的类型(如double)。否则，它总是返回False。

5 < 3返回0(即False); 而3 < 5返回1(即True)。比较可以被任意连接: 3 < 5 < 7返回True。

大于、小于、小于等于、大于等于时:数字跟数字可以比较，字符串跟字符串可以比较，但数字不能跟字符串比较。

等于、不等于时: 数字跟数字可以比较，字符串跟字符串可以比较，数字跟字符串比较返回 False (表示不相等)

等于: Python 使用“==”来做比较，用“=”来赋值。但不允许内嵌的赋值，所以不会出现你本以为在做比较而意外的写成赋值的情况。

7.布尔运算: and 和 or 都是短路运算,没有非短路运算的运算符。

短路运算:当前面一个表达式可以决定结果时，后面的语句不用再判断。非短路运算时，还照样判断后面的。

注意：在 and or 运算中，空字符串'',数字0,空列表[],空字典{},空元组(), None,在逻辑运算中都被当作假来处理。

8.or 的特殊用法:

由于语言的松散性,用 or 返回值时有判断作用。

如:ss = False or None or 0 or '' or -1 or 'sss'; print(ss) # 打印:-1

设定预设值的写法: edittype = edittype or "text"; # 如果 edittype 之前有值,则取之前的值; 之前为空,则取默认值

9.自增,自减:

Python 没有“++”和“--”两个语法,自增自减时只能写: i = i + 1 或者 i += 1, 不能用 i++

这在一定程度上避免出错，因为新手经常搞错“++”放前面还是放后面

10.三目运算符：

Python 没有三目运算符“cond ? a : b”,但可以使用 and or 来代替这个运算，如下：

1) c = cond and a or b # 这在大多数情况下都是正确的，但当你需要的值 a 是空字符串'',数字0等逻辑运算为假的情况下会出错。

2) c = (cond and [a] or [b])[0] # 即使 a或者b为一个逻辑假的值，将他放入集合中后，就为真了，也就是[False] [None]都不为假。

3) c = (b, a)[cond and 1 or 0] # 注意 a和b的位置是颠倒的,将表达式结果转成1和0来作为元组下标而选择结果。

11.switch/case 语句

Python现在不支持这语句，但可以用 range(N) 生成一个 列表

## 运算符优先级

下面这个表给出Python的运算符优先级，从最低的优先级（最松散地结合）到最高的优先级（最紧密地结合）。

在一个表达式中，Python会首先计算下表中较下面的运算符，然后在计算列在下表上部的运算符。

在下表中列在同一行的运算符具有 相同优先级 。例如，+和-有相同的优先级。

建议使用圆括号来分组运算符和操作数，以便能够明确地指出运算的先后顺序，使程序尽可能地易读。例如，2 + (3 \* 4)显然比2 + 3 \* 4清晰。

运算符 描述

lambda Lambda表达式

or 布尔“或”

and 布尔“与”

not x 布尔“非”

in，not in 成员测试

is，is not 同一性测试

<，<=，>，>=，!=，== 比较

| 按位或

^ 按位异或

& 按位与

<<，>> 移位

+，- 加法与减法

\*，/，% 乘法、除法与取余

+x，-x 正负号

~x 按位翻转

\*\* 指数

x.attribute 属性参考

x[index] 下标

x[index:index] 寻址段

f(arguments...) 函数调用

(experession,...) 绑定或元组显示

[expression,...] 列表显示

{key:datum,...} 字典显示

'expression,...' 字符串转换

## 计算顺序

默认地，运算符优先级表决定了哪个运算符在别的运算符之前计算。然而，如果要改变它们的计算顺序，得使用圆括号。

例如，你想要在一个表达式中让加法在乘法之前计算，那么你就得写成类似(2 + 3) \* 4的样子。

## 结合规律

运算符通常由左向右结合，即具有相同优先级的运算符按照从左向右的顺序计算。例如，2 + 3 + 4被计算成(2 + 3) + 4。

一些如赋值运算符那样的运算符是由右向左结合的，即a = b + c被处理为a = (b + c)。

## 字符串

1.使用单引号“'”引起来: 'Quote me on this'

2.使用双引号“"”引起来: "What's your name?"

3.使用三引号（'''或"""）: 可以指示一个多行的字符串。你可以在三引号中自由的使用单引号和双引号。 /'''

如:

"""This is a multi-line string. This is the first line.

"What's your name?," I asked.

He said "Bond, James Bond."

"""

4.转义符“\”

\\ 指示反斜杠本身

\' 指示单引号

\" 指示双引号

注意: 行末的单独一个反斜杠表示字符串在下一行继续，而不是开始一个新的行。

5.自然字符串

自然字符串通过给字符串加上前缀r或R来指定，取消转义符的功能。例如: r"Newlines are indicated by \n"。

三引号的字符串也可以同样用法，如：R'''Newlines are indicated by \n'''

6.Unicode字符串

Python允许你处理Unicode文本(超过拉丁文字范围的)——只需要在字符串前加上前缀u或U。

例如，u"This is a Unicode string.哈哈.."。(Python3.x之后不需要这样了,可以直接写中文;而这样写会报错)

Python 3.0开始对unicode全面支持，所有的文本(str)都是Unicode的；并引入了一个叫做bytes的新类型来处理字节序列。而编码过的Unicode会以二进制的数据来表示。

因为在2.x的世界里，大量的bug都是因为已编码的文本和未编码的文本混杂在一起而产生的。

7.按字面意义级连字符串

如果你把两个字符串按字面意义相邻放着，他们会被Python自动级连。

例如，"What's" ' your name?'会被自动转为"What's your name?"。

即是说，两个字符串放在一起，会有字符拼接的效果。加号“+”也有字符拼接的效果。

8.字符串拼接

可以使用“str1.\_\_add\_\_(str2)”或者“str1 + str2”或者直接两个字符串放一起,来拼接字符串

但字符串与其它类型拼接时，得先把其它类型转成字符串类型，否则会出错。如“str1 + 2”就会出错，需要“str1 + str(2)”

9.格式化

使用“%”可以格式化字符串,非常方便。如: str1 = "Swaroop's age is %d, weight is %f" % (5, 65.5)

格式化的符号用法参考下面的“字符串格式化控制表”

10.字符串序列(索引和切片)

字符串可以使用下标来获取字符串中某个项目，以及截取字符串。详情参考“序列”

用法如: name = 'swaroop'; name[1]; name[1:3]; name[1:-1]

11.str(anything)函数和 unicode(anything)函数

Python 2有两个全局函数可以把对象强制转换成字符串:unicode()把对象转换成Unicode字符串，还有 str()把对象转换为非Unicode字符串。

Python 3只有一种字符串类型，Unicode字符串，所以 str()函数即可完成所有的功能。(unicode()函数在Python 3里不再存在了。)

另外:

没有专门的char数据类型，确实没有需要有这个类型。

单引号和双引号字符串是完全相同的——它们没有在任何方面有不同。

正则表达式: 一定要用自然字符串处理正则表达式。否则会需要使用很多的反斜杠。

使用 help(str) 可查看字符串对象定义的所有方法及属性。

字符串格式化控制：(未参考帮助文档，只是个人猜测)

转义符 (Escape Sequence)：

\ddd 1到3位8进制数指定Unicode字符输出(如：“\127”显示“W”)

\uxxxx 1到4位16进制数指定Unicode字符输出(Python3.x开始支持此写法,如: \u54C8 显示“哈”字)

\xhh 16进制数指定Unicode字符输出(如：“\xe5\x93\x88”显示“哈”)

\\ \

\ \ (单独的一个斜杠也显示斜杠,即不后接有转移作用的字符时，作为斜杠使用)

\' '

\" "

\a 字符：0x07 响铃(ASCII控制字符)

\b 字符：0x08 退格(光标向左走一格)(ASCII控制字符)

\f 字符：0x0c Formfeed(FF)(走纸转页,换页)(ASCII控制字符)

\n 字符：0x0a 换行(ASCII控制字符)

\N{name} Unicode字符 只能针对Unicode

\r 字符：0x0d 回车

\t 字符：0x09 跳格(tab符号),水平制表符

\v 字符：0x0b 垂直制表符

%% %

%d 输出10进制整数，只能是数字类型，输出字符串类型会出错；浮点类型的数字将被取整(直接删除小数部分)。

%f,%F 以10进制输出浮点数，只能是数字类型，输出字符串类型会出错。

%e,%E 以科学计数法输出10进制的浮点数，大小写的“e”反应在显示时科学计数法的“e/E”上，只能是数字类型。

%a Python3.0开始支持此写法，原样输出结果，字符串类型会加上单引号引起来。

%o (字母o)以8进制整数方式输出，只能是数字类型；浮点类型的数字将被取整(直接删除小数部分)。

%x,%X 将数字以16进制方式输出，只能是数字类型；浮点类型的数字将被取整(直接删除小数部分)。

%s 将字符串格式化输出(可输出任何类型)

%c 以字符方式输出，提供的类型必须是 char 或 int 。

注：布尔类型的 True 或 False,用数字类型输出是 1或0,字符串输出是 True 或 False。

字符串转换成数字

float(str) 转换成浮点数,如, float("1e-1") 结果：0.1

int(str) 转换成整数,如, int("12") 结果：12

int(str,base) 转换成base进制的整数,如, int("11",2) 转换成2进制的整数,结果：3

long(str) 转换成长整数,Python3取消此语法,如, long("12L") 结果：12L

long(str,base) 转换成base进制的长整数,Python3取消此语法,如, long("11L",2) 结果：3L

字符串用例

name = 'Swaroop' # This is a string object

# 检查字符串的开头部分

if name.startswith('Swa'): # 类似函数如 endswith()

print('Yes, the string starts with "Swa"')

# 检查是否包含有此内容

if 'a' in name:

print('Yes, it contains the string "a"')

# 找出给定字符串的位置,找不到则返回-1

if name.find('war') != -1:

print('Yes, it contains the string "war"', 's')

# join()函数把列表拼接起来

delimiter = '; '

mylist = ['Brazil', 'Russia', 'India', 'China']

print(delimiter.join(mylist)) # 打印: Brazil; Russia; India; China

# 大小写转换

print("THIS IS TEST".lower()) # 转换成小写,打印：this is test

print("this is test".upper()) # 转换成大写,打印：THIS IS TEST

print("This Is Test".swapcase()) # 大小写互换,打印：tHIS iS tEST

## 控制台输入

使用 raw\_input()函数 或者 input()函数 能够很方便的控从制台读入数据，得到的是字符串。

Python2.x时,raw\_input()和 input()的区别:

当输入为纯数字时:input()返回的是数值类型，如:int,float; raw\_input()返回的是字符串类型

input()会计算在字符串中的数字表达式，而 raw\_input()不会。如输入“57+3”: input()得到整数60,raw\_input()得到字符串'57+3'

注:Python3.0将 raw\_input()函数去除了,而用 input() 取代它的功能。另外,input()获取的字符串会包括结尾的换行符。

例:(此处是Python2.6的写法，3.x时应该把 raw\_input() 改成 input())

### 1.输入字符串

nID = raw\_input("Input your id plz:\n"); print('your id is %s' % (nID))

### 2.输入整数

nAge = int(raw\_input("input your age plz:\n")); print('your age is %d\n' % nAge)

### 3.输入浮点型

fWeight = float(raw\_input("input your weight:\n")); print('your weight is %f' % fWeight)

### 4.输入16进制数据

nHex = int(raw\_input('input hex value(like 0x20):\n'),16); print('nHex = %x,nOct = %d\n' %(nHex,nHex))

### 5.输入8进制数据

nOct = int(raw\_input('input oct value(like 020):\n'),8); print('nOct = %o,nDec = %d\n' % (nOct,nOct))

注:打印字符串时，“%”作为特殊符号，两个百分号才能打印出一个百分号

Python 2 与 Python 3 的比较

Python 2 Python 3

① raw\_input() input() # 最简单的形式，raw\_input()被替换成input()。

② raw\_input('prompt') input('prompt') # 可以指定一个提示符作为参数

③ input() eval(input()) # 如果想要请求用户输入一个Python表达式，计算结果

## 控制流:

### if 语句

写法: if ... elif ... else ... # if后面不用圆括号

注:在Python中没有switch语句。你可以使用 if..elif..else 语句来完成同样的工作（在某些场合，使用字典会更加快捷。）

在C/C++里面可以使用 else if ,但这里不行，得写成: else :\n\t if,故此增加关键字 elif

例:

number = 23

# int是一个类，不过这里它只是把一个字符串转换为一个整数（假设这个字符串含有一个有效的整数文本信息）。

guess = int(raw\_input('Enter an integer : '))

if guess == number:

print('Congratulations, you guessed it.')

elif guess < number:

print('No, it is a little higher than that') # Another block

else:

print('No, it is a little lower than that')

### while 语句

只要条件为真，while语句允许你重复执行一块语句。

注:while语句有一个可选的else从句。else块事实上是多余的，因为可以把其中的语句跟在while语句里面。

例:

number = 23

running = True

while running:

guess = int(raw\_input('Enter an integer : '))

if guess == number:

print('Congratulations, you guessed it.')

running = False # this causes the while loop to stop

elif guess < number:

print('No, it is a little higher than that')

else:

print('No, it is a little lower than that')

else:

# Do anything else you want to do here

print('The while loop is over.')

### for 循环

for..in 是另外一个循环语句，它在一序列的对象上 递归 即逐一使用队列中的每个项目。

else 部分是可选的。如果包含 else,它总是在 for 循环结束后执行一次，除非遇到 break 语句。

例:

for i in range(1, 5):

print(i)

else:

print('The for loop is over')

# 打印结果: 1 至 4 以及 else 的内容

# range(1,5)给出序列[1, 2, 3, 4]。range默认步长为1,第三个参数是步长。如，range(1,5,2)给出[1,3]。

# 记住，range包含第一个数,但不包含第二个数。

注:

Python的 for 循环从根本上不同于C/C++的 for 循环。类似 foreach 循环。

在C/C++中，如果你想要写 for (int i = 0; i < 5; i++)，那么用Python，你写成 for i in range(0,5)。

# 范例：九九乘法表

# 由于Python2 与Python3的 print 语法不相同，改用string来打印，保证两个版本的输出结果相同。

str = ''

for i in range(1,10):

for j in range(1, i+1):

str += ('%d \* %d = %d \t' % (j, i, i\*j))

str += '\n'

print(str)

### break 语句

break 语句是用来 终止 循环语句的，即哪怕循环条件没有称为 False 或序列还没有被完全递归，也停止执行循环语句。

一个重要的注释是，如果你从 for 或 while 循环中 终止 ，任何对应的循环 else 块将不执行。

### continue 语句

continue 语句被用来告诉Python跳过当前循环块中的剩余语句，然后 继续 进行下一轮循环。

break 语句 和 continue 语句 对于 while 循环 和 for 循环 都有效。

例(2.x写法):

while True:

s = raw\_input('Enter something : ')

if s == 'quit':

break

if len(s) < 3:

print 'Input is not of sufficient length'

continue

# Do other kinds of processing here...

print 'Length of the string is', len(s)

例(3.x写法):

while True:

s = input('Enter something : ') # 3.x用input()代替raw\_input(),且会获取结尾输入的换行符

s = s[:-1] # 去掉结尾的换行符

if s == 'quit':

break

if len(s) < 3:

print('Input is not of sufficient length')

continue

# Do other kinds of processing here...

print('Length of the string is', len(s))

## 函数:

### 定义函数

函数通过def关键字定义。

def关键字后跟一个函数的 标识符 名称，然后跟一对圆括号。圆括号之中可以包括一些变量名，该行以冒号结尾。

接下来是一块语句，它们是函数体。

例:

def sayHello():

print('Hello World!') # block belonging to the function

sayHello() # call the function

### 函数形参

函数中的参数名称为 形参 而你提供给函数调用的值称为 实参 。

### 局部变量

当你在函数定义内声明变量的时候，它们与函数外具有相同名称的其他变量没有任何关系，即变量名称对于函数来说是 局部 的。

这称为变量的 作用域 。所有变量的作用域是它们被定义的块，从它们的名称被定义的那点开始。

例:

x = 50

def func(x):

print('x is', x)

x = 2

print('Changed local x to', x) # 打印: 2

func(x)

print('x is still', x) # 打印: 50, 值没有变

### global语句

如果要为一个定义在函数外的变量赋值，那么你就得告诉Python这个变量名不是局部的，而是 全局 的。使用global语句完成这一功能。

没有global语句，是不可能为定义在函数外的变量赋值的。

你可以使用定义在函数外的变量的值（假设在函数内没有同名的变量）。然而，应避免这样做，因为这降低程序的可读性,不清楚变量在哪里定义的。

使用global语句可以清楚地表明变量是在外面的块定义的。

注:可以使用同一个global语句指定多个全局变量。例如 global x, y, z。

例:

def func():

global x

print('x is', x)

x = 2

print('Changed local x to', x) # 打印: 2

x = 50

func()

print('Value of x is', x) # 打印: 2, 值被改变了

### 默认参数值

如果希望一些参数是 可选 的，这些参数可使用默认值。

可以在函数定义的形参名后加上赋值运算符（=）和默认值，从而给形参指定默认参数值。

注意，默认参数值应该是一个参数。

例:

def say(message, times = 2):

print(message \* times)

say('Hello ') # 打印:Hello Hello

say('World ', 5) # 打印:World World World World World

重要:

只有在形参表末尾的那些参数可以有默认参数值，即不能在声明函数形参的时候，先声明有默认值的形参而后声明没有默认值的形参。

这是因为赋给形参的值是根据位置而赋值的。例如，def func(a, b=5)是有效的，但是def func(a=5, b)是 无效 的。

### 关键参数

如果某个函数有许多参数，而你只想指定其中的一部分，那么可以通过命名来为这些参数赋值

——这被称作 关键参数 ——使用名字（关键字）而不是位置来给函数指定实参。

这样做有两个优势:

一、由于我们不必担心参数的顺序，使用函数变得更加简单了。

二、假设其他参数都有默认值，我们可以只给我们想要的那些参数赋值。

例:

def func(a, b=5, c=10):

print('a is', a, 'and b is', b, 'and c is', c)

func(3, 7) # 参数a得到值3，参数b得到值7，而参数c使用默认值10。

func(25, c=24) # 根据实参的位置,变量a得到值25。根据命名，即关键参数，参数c得到值24。变量b根据默认值，为5。

func(c=50, a=100) # 使用关键参数来完全指定参数值。a得到值100,c得到值50。变量b根据默认值，为5。

### return语句

return语句用来从一个函数 返回 即跳出函数。我们也可选从函数 返回一个值 。

例:

def maximum(x, y):

if x > y:

return x

else:

return y

print(maximum(2, 3)) # 打印 3

### None

None 是Python中表示没有任何东西的特殊类型(相当于java的 null)。例如，如果一个变量的值为None，可以表示它没有值。

注意:函数没有返回值的,等价于最后返回return None。通过运行print someFunction()，你可以明白这一点。

例:

def someFunction():

# pass语句在Python中表示一个空的语句块。

pass

print(someFunction())

### DocStrings

DocStrings:文档字符串。它是一个重要的工具，帮助你的程序文档更加简单易懂，应该尽量使用它。甚至可以在程序运行的时候，从函数恢复文档字符串！

在函数的第一个逻辑行的字符串是这个函数的 文档字符串 。注意，DocStrings也适用于模块和类。

文档字符串的惯例是一个多行字符串，它的首行以大写字母开始，句号结尾。第二行是空行，从第三行开始是详细的描述。 强烈建议遵循这个惯例。

例:

def printMax(x, y):

'''Prints the maximum of two numbers.

The two values must be integers.'''

x = int(x) # convert to integers, if possible

y = int(y)

if x > y:

print(x, 'is maximum')

else:

print(y, 'is maximum')

printMax(3, 5) # 打印: 5 is maximum

print(printMax.\_\_doc\_\_) # 打印: Prints the maximum ... must be integers.

注:

使用\_\_doc\_\_（注意是两个下划线）调用printMax函数的文档字符串属性。请记住Python把 每一样东西 都作为对象，包括这个函数。

Python中help()函数即是使用DocStings的了,它只是抓取函数的\_\_doc\_\_属性，然后整洁地展示给你。可以对上面的函数尝试一下: help(printMax)。记住按q退出help。

自动化工具也可以以同样的方式从你的程序中提取文档。因此强烈建议你对你所写的任何正式函数编写文档字符串。

### 函数属性 func\_\*

在Python 2里，函数的里的代码可以访问到函数本身的特殊属性。在Python 3里，为了一致性，这些特殊属性被重新命名了。

Python 2 与 Python 3 的比较

Python 2 Python 3 说明

① a\_function.func\_name a\_function.\_\_name\_\_ # 包含了函数的名字。

② a\_function.func\_doc a\_function.\_\_doc\_\_ # 包含了在函数源代码里定义的文档字符串(docstring)。

③ a\_function.func\_defaults a\_function.\_\_defaults\_\_ # 是一个保存参数默认值的元组。

④ a\_function.func\_dict a\_function.\_\_dict\_\_ # 一个支持任意函数属性的名字空间。

⑤ a\_function.func\_closure a\_function.\_\_closure\_\_ # 一个由cell对象组成的元组，包含了函数对自由变量(free variable)的绑定。

⑥ a\_function.func\_globals a\_function.\_\_globals\_\_ # 一个对模块全局名字空间的引用，函数本身在这个名字空间里被定义。

⑦ a\_function.func\_code a\_function.\_\_code\_\_ # 一个代码对象，表示编译后的函数体。

## 模块:

如果要在其他程序中重用很多函数，那么你该使用模块。

模块基本上就是一个包含了所有你定义的函数和变量的文件。

为了在其他程序中重用模块，模块的文件名必须以.py为扩展名。

### sys模块(标准库模块)

sys模块包含了与Python解释器和它的环境有关的函数。

例:

import sys # 输入 sys模块。基本上，这句语句告诉Python，我们想要使用这个模块。

print('The command line arguments are:')

# 打印调用文件的命令行参数

for i in sys.argv:

print(i)

print('\nThe PYTHONPATH is', sys.path)

输出:

$ python using\_sys.py we are arguments

The command line arguments are:

using\_sys.py

we

are

arguments

The PYTHONPATH is ['/home/swaroop/byte/code', '/usr/lib/python23.zip',

'/usr/lib/python2.3', '/usr/lib/python2.3/plat-linux2',

'/usr/lib/python2.3/lib-tk', '/usr/lib/python2.3/lib-dynload',

'/usr/lib/python2.3/site-packages', '/usr/lib/python2.3/site-packages/gtk-2.0']

注:

执行 import sys 语句的时候，它在 sys.path 变量中所列目录中寻找 sys.py 模块。

如果找到了这个文件，这个模块的主块中的语句将被运行，然后这个模块将能够被你使用。

注意，初始化过程仅在我们 第一次 输入模块的时候进行。另外，“sys”是“system”的缩写。

脚本的名称总是sys.argv列表的第一个参数。所以，在这里，'using\_sys.py'是sys.argv[0]、'we'是sys.argv[1]。

sys.path包含输入模块的目录名列表。

可以观察到sys.path的第一个字符串是空的——这个空的字符串表示当前目录也是sys.path的一部分，这与PYTHONPATH环境变量是相同的。

这意味着你可以直接输入位于当前目录的模块。否则，你得把你的模块放在sys.path所列的目录之一。

另外:

sys.exit() # 程序结束

sys.stdin、 sys.stdout 和 sys.stderr 分别对应你的程序的标准输入、标准输出和标准错误流。

### 字节编译的.pyc文件

输入一个模块相对来说是一个比较费时的事情，所以Python做了一些技巧，以便使输入模块更加快一些。

一种方法是创建 字节编译的文件，这些文件以.pyc作为扩展名。另外，这些字节编译的文件也是与平台无关的。

当你在下次从别的程序输入这个模块的时候，.pyc文件是十分有用的——它会快得多，因为一部分输入模块所需的处理已经完成了。

### from ... import 语句

如果你想要直接输入 argv 变量到你的程序中（避免在每次使用它时打sys.），那么你可以使用 from sys import argv 语句。

如果你想要输入所有 sys 模块使用的名字，那么你可以使用 from sys import \*语句。

这对于所有模块都适用。

一般说来，应该避免使用 from...import 而使用 import 语句，因为这样可以使你的程序更加易读，也可以避免名称的冲突。

### import ... as

为 import 的模块起一个简称。如: import cPickle as p

起简称后,下面的语句即可使用简称,如原本的 cPickle.dump() 可写成 p.dump()

### 模块的 \_\_name\_\_

每个模块都有一个名称，在模块中可以通过语句来找出模块的名称。

这在一个场合特别有用——就如前面所提到的，当一个模块被第一次输入的时候，这个模块的主块将被运行。

假如我们只想在程序本身被使用的时候运行主块，而在它被别的模块输入的时候不运行主块，我们该怎么做呢？这可以通过模块的\_\_name\_\_属性完成。

每个Python模块都有它的\_\_name\_\_，如果它是'\_\_main\_\_'，这说明这个模块被用户单独运行，我们可以进行相应的恰当操作。

例:

# Filename: using\_name.py

if \_\_name\_\_ == '\_\_main\_\_':

print('This program is being run by itself')

else:

print('I am being imported from another module')

输出:

$ python using\_name.py

This program is being run by itself

$ python

>>> import using\_name

I am being imported from another module

### 创建自己的模块

每个Python程序也是一个模块。

模块,例:

# Filename: mymodule.py

def sayhi():

print('Hi, this is mymodule speaking.')

version = '0.1'

# End of mymodule.py

上面是一个 模块 的例子。你已经看到，它与我们普通的Python程序相比并没有什么特别之处。

记住这个模块应该被放置在我们输入它的程序的同一个目录中，或者在 sys.path 所列目录之一。

用例1:

import mymodule

mymodule.sayhi()

print('Version', mymodule.version)

注:函数和成员都以点号来使用。

用例2: 使用from..import语法的版本。

from mymodule import sayhi, version # 或者写: from mymodule import \*

sayhi()

print('Version', version)

当引用模块时，编译器会先在当前目录，接着去sys.path、Python的安装目录去寻找你引用的模块，如果没有的话，就会报错。

第一种情况，你的模块和你要使用的模块在同一目录下。

第二种情况，模块在别的地方。

方法一：先引用sys，对sys.path进行修改，其实就是将模块的路径加到sys.path中。

test.py

import sys

sys.path.append('e:/workspace/Modules')

import Hello

Hello.hello()

方法二：就是将路径加入到环境变量中去，这个是永久的，你可以新建一个专门存放模块的文件夹，把这个文件夹的路径加入到环境变量中去，可以一次劳动，永不麻烦，同时也便于管理。

### dir()函数

使用内建的dir函数来列出模块定义的标识符。标识符有函数、类和变量。

当你为dir()提供一个模块名的时候，它返回模块定义的名称列表。如果不提供参数，它返回当前模块中定义的名称列表。

$ python

>>> import sys

>>> dir(sys) # get list of attributes for sys module

['\_\_displayhook\_\_', '\_\_doc\_\_', '\_\_excepthook\_\_', '\_\_name\_\_', '\_\_stderr\_\_',

'\_\_stdin\_\_', '\_\_stdout\_\_', '\_getframe', 'api\_version', 'argv',

'builtin\_module\_names', 'byteorder', 'call\_tracing', 'callstats',

'copyright', 'displayhook', 'exc\_clear', 'exc\_info', 'exc\_type',

'excepthook', 'exec\_prefix', 'executable', 'exit', 'getcheckinterval',

'getdefaultencoding', 'getdlopenflags', 'getfilesystemencoding',

'getrecursionlimit', 'getrefcount', 'hexversion', 'maxint', 'maxunicode',

'meta\_path','modules', 'path', 'path\_hooks', 'path\_importer\_cache',

'platform', 'prefix', 'ps1', 'ps2', 'setcheckinterval', 'setdlopenflags',

'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout',

'version', 'version\_info', 'warnoptions']

>>> dir() # get list of attributes for current module

['\_\_builtins\_\_', '\_\_doc\_\_', '\_\_name\_\_', 'sys']

>>>

>>> a = 5 # create a new variable 'a'

>>> dir()

['\_\_builtins\_\_', '\_\_doc\_\_', '\_\_name\_\_', 'a', 'sys']

>>>

>>> del a # delete/remove a name; 这个得留意

>>>

>>> dir()

['\_\_builtins\_\_', '\_\_doc\_\_', '\_\_name\_\_', 'sys']

>>>

## 数据结构

可以处理一些 数据 的 结构 。或者说，它们是用来存储一组相关数据的。

在Python中有三种内建的数据结构——列表、元组和字典。

### 列表(list)

是处理一组有序项目的数据结构，即你可以在一个列表中存储一个 序列 的项目。

列表中的项目应该包括在方括号中，每个项目之间用逗号分割。

可以添加、删除或是搜索列表中的项目。列表是 可变的 数据类型。

列表对象定义的所有方法可以通过 help(list) 获得完整的知识。我比较习惯称它为“数组”。

例:

shoplist = ['apple', 'mango', 'carrot', 'banana']

#查看长度

print('I have', len(shoplist),'items to purchase.')

#遍历

print('These items are:', end=' ') # 注意这行的结尾,打印时可以不换行,python 2.x应该用逗号结尾

for item in shoplist:

print(item, end=' ') # python 2.x 此行应该写：“print item,”

#添加

print('\nI also have to buy rice.')

shoplist.append('rice')

print('My shopping list is now:', shoplist)

#排序

print('I will sort my list now')

shoplist.sort()

print('Sorted shopping list is:', shoplist)

#删除,以及使用下标

print('The first item I will buy is:', shoplist[0])

olditem = shoplist[0]

del shoplist[0]

print('I bought the', olditem)

print('My shopping list is now:', shoplist)

#多维列表时,保存的对象只是引用

newlist = ['waa','dd']

shoplist.append(newlist)

print('My shopping list is now:', shoplist)

del newlist[0]

print('My shopping list is now', shoplist)

### 元组

元组和列表十分类似，只不过元组和字符串一样是 不可变的 即你不能修改元组。

元组通过圆括号中用逗号分割的项目定义。

元组通常用在使语句或用户定义的函数能够安全地采用一组值的时候，即被使用的元组的值不会改变。

如果你想要知道元组对象定义的所有方法，可以通过 help(tuple) 获得完整的知识。

例:

#一维元组

zoo = ('wolf', 'elephant', 'penguin')

print('Number of animals in the zoo is %s' % len(zoo)) # 打印: 3

newlist = ['waa','dd']

#多维元组

new\_zoo = ('monkey', 'dolphin', zoo, newlist)

print('Number of animals in the new zoo is %s' % len(new\_zoo)) # 打印: 3

print('All animals in new zoo are %s' % str(new\_zoo)) # 打印: ('monkey','dolphin',('wolf','elephant','penguin'),['waa','dd'])

print('Animals brought from old zoo are %s' % str(new\_zoo[2])) # 打印: ('wolf', 'elephant', 'penguin')

print('Last animal brought from old zoo is %s' % new\_zoo[2][2]) # 打印: penguin

#多维元组时,保存的对象只是引用

del newlist[0]

print('new\_zoo is now:' + str(new\_zoo) ) # 打印: ('monkey','dolphin',('wolf','elephant','penguin'),['dd'])

注意:含有0个或1个项目的元组

一个空的元组(含有0个项目)由一对空的圆括号组成，如myempty = ()。

含有单个元素的元组必须在第一个（唯一一个）项目后跟一个逗号。如: singleton = (2 , )。

如果小括号里面只有唯一一个项目,而这个项目后面又没有跟一个逗号的话,Python会认为它是一个表达式。

### 字典

字典把键(名字)和值(详细情况)联系在一起。

注意，键必须是唯一的，且只能使用不可变的对象(比如字符串)来作为字典的键，但字典的值没有此限制。应该说只能使用简单的对象作为键。

键值对在字典中以这样的方式标记：d = {key1 : value1, key2 : value2 }。

键值对用冒号分割，而各个对用逗号分割，所有这些都包括在花括号中。

记住字典中的键/值对是没有顺序的。如果要一个特定的顺序，那么应该在使用前自己对它们排序。

字典是dict类的实例/对象，可以用 help(dict) 来查看其属性和函数。

例:

ab = { 'Swaroop': 'swar',

'Larry' : 'larry',

'Spammer': 'spammer'

}

print(ab) # 打印: {'Swaroop':'swar', 'Larry':'larry', 'Spammer':'spammer'}

print("Swaroop's address is %s" % ab['Swaroop']) # 打印: swar

# Adding a key/value pair

ab['Guido'] = 'guido'

# Deleting a key/value pair

del ab['Spammer']

print('\nThere are %d contacts in the address-book\n' % len(ab)) # 打印: 3

# 遍历(这写法得留意)

for name, address in ab.items():

print('Contact %s at %s' % (name, address))

# 包含key

if 'Guido' in ab: # 或者写： if ab.has\_key('Guido'):

print("\nGuido's address is %s" % ab['Guido'])

# 原字典上创建新字典

print(ab) # 打印: {'Swaroop':'swar', 'Larry':'larry', 'Guido':'guido'}

dd = dict(ab, slug\_field='slug', test=5) # 创建新字典,字典作为参数的只能放第一个，其余不能再是字典；字典参数可省略

print(dd) # 打印: {'Swaroop':'swar', 'test':5, 'slug\_field':'slug', 'Larry':'larry', 'Guido':'guido'}

### 序列

列表、元组和字符串都是序列，序列的两个主要特点是“索引”操作符和“切片”操作符。

索引操作符让我们可以从序列中抓取一个特定项目。(即使用下标)

切片操作符让我们能够获取序列的一个切片，即一部分序列。(即在下标的中括号里面使用冒号)

例:

shoplist = ['apple', 'mango', 'carrot', 'banana']

# Indexing or 'Subscription' operation

print('Item 0 is %s' % shoplist[0])

print('Item 3 is %s' % shoplist[3])

print('Item -1 is %s' % shoplist[-1]) # 打印:banana 即倒数第一个

print('Item -2 is %s' % shoplist[-2]) # 打印:carrot 即倒数第二个

# Slicing on a list

print('Item 1 to 3 is %s' % shoplist[1:3]) # 打印:['mango', 'carrot'] 即下标[1]到[3],包括开始但不包括结束

print('Item 2 to end is %s' % shoplist[2:]) # 打印:['carrot', 'banana'] 即下标[2]到结束,包括最后一个

print('Item 1 to -1 is %s' % shoplist[1:-1]) # 打印:['mango', 'carrot'] 即下标[1]到[-1],包括开始但不包括结束

print('Item start to end is %s' % shoplist[:]) # 打印整个列表,跟直接写“shoplist”效果一样

# Slicing on a string (string与列表有同样的操作,)

name = 'swaroop'

print('characters 1 to 3 is %s' % name[1:3]) # 打印:wa

print('characters 2 to end is %s' % name[2:]) # 打印:aroop

print('characters 1 to -1 is %s' % name[1:-1]) # 打印:waroo

print('characters start to end is %s' % name[:]) # 打印:swaroop 跟直接写这个字符串一样

### 参考

当你创建一个对象并给它赋一个变量的时候，这个变量仅仅 参考 那个对象，而不是表示这个对象本身！

也就是说，变量名指向你计算机中存储那个对象的内存。

这被称作名称到对象的绑定。

例:

shoplist = ['apple', 'mango', 'carrot', 'banana']

mylist = shoplist # mylist 只是对象的另一个名称,他们指向相同的内存空间

del shoplist[0]

# 他们此时打印相同的内容,都少了'apple'

print('shoplist is', shoplist)

print('mylist is', mylist)

# 深拷贝,复制成另一个对象(得记住用切片操作符来取得拷贝)

mylist = shoplist[:] # make a copy by doing a full slice

del mylist[0] # remove first item

# 注意，现在他们打印出不同的内容

print('shoplist is', shoplist)

print('mylist is', mylist)

### 列表综合

通过列表综合，可以从一个已有的列表导出一个新的列表。

# 例如，你有一个数的列表，而你想要得到一个对应的列表，使其中所有大于2的数都是原来的2倍。对于这种应用，列表综合是最理想的方法。

listone = [2, 3, 4]

listtwo = [2\*i for i in listone if i > 2] # 为满足条件（if i > 2）的数指定了一个操作（2\*i），从而导出一个新的列表。

print(listtwo) # 打印: [6, 8]

a\_dict = {'a': 1, 'b': 2, 'c': 3}

b\_dict = {value:key for key, value in a\_dict.items()} # 这语法仅Python3可以，之前的版本不行

print(b\_dict) # key与value翻转，打印: {1:'a', 2:'b', 3:'c'}

说明:

注意原来的列表并没有发生变化。

在很多时候，我们都是使用循环来处理列表中的每一个元素，而使用列表综合可以用一种更加精确、简洁、清楚的方法完成相同的工作。

### 集合

Python3 开始有这写法,跟之前的差不多,只是用大括号括起来,如： a = {1, 'aa', 3, 5, 6}

集合同样可以使用综合计算，如： a = {x for x in range(10) if x % 2 == 0}

成员测试 in, not in

检查是否包含有此内容,返回 True 或者 False, 例如：

# 1.对字符串

if 'a' in 'Swaroop':

print('Yes, it contains the string "a"')

# 2.对集合(列表、元组和字典)

if 'genre' in ('genre', 'jazz'):

print('Yes, it contains the genre')

print('genre' in ('genre', 'jazz')) # 元组,打印： True

print('genre' in ['genre', 'jazz']) # 列表,打印： True

print('genre' in {'genre':'sss', 'jazz':'dddd'}) # 字典,检查key，打印： True

print('sss' in {'genre':'sss', 'jazz':'dddd'}) # 字典,打印： False

在函数中接收元组和列表(函数的参数数量可以变动,即可变长参数)

当要使函数接收元组或字典形式的参数的时候，有一种特殊的方法，它分别使用\*和\*\*前缀。

这种方法在函数需要获取可变数量的参数的时候特别有用。

例:

# 由于在args变量前有\*前缀，所有多余的函数参数都会作为一个元组存储在args中

def sum(message, \*args):

'''Return the sum of each argument.'''

total = 0

# 除了用循环,也可以用下标来读取参数,如: args[0]

for i in args:

total += i

print (message + ":" + str(total))

sum('hight', 3, 5.5) # 打印: hight:8.5

sum('weight', 10) # 打印: weight:10

# 函数参数接收字典用法。使用的是\*\*前缀，多余的参数则会被认为是一个字典的键/值对。

def printDict(message, \*\*args):

print(message + ':' + str(args))

# 注意:参数为字典时,参数里面必须使用等号,否则运行出错

printDict('hight', a=3, b='dd') # 打印: hight:{'a':3, 'b':'dd'}

## 面向对象的编程

面向过程的编程:根据操作数据的函数或语句块来设计程序的。

面向对象的编程:把数据和功能结合起来，用称为对象的东西包裹起来组织程序的方法。

类和对象是面向对象编程的两个主要方面。“类”创建一个新类型，而“对象”是这个类的实例。

域:属于一个对象或类的变量。

方法:属于类的函数，被称为类的方法。

域和方法可以合称为类的属性。

域有两种类型——属于每个实例/类的对象或属于类本身。它们分别被称为实例变量和类变量。

类使用class关键字创建。类的域和方法被列在一个缩进块中。

### self参数

类的方法与普通的函数只有一个特别的区别——它们“必须”有一个额外的第一个参数名称，但是在调用这个方法的时候你不为这个参数赋值，Python会提供这个值。这个特别的变量指对象本身，按照惯例它的名称是self。

虽然你可以给这个参数任何名称，但是“强烈建议”使用self这个名称——其他名称都是不赞成使用的。

使用一个标准的名称有很多优点——1.方便别人阅读；2.有些IDE（集成开发环境）也可以帮助你。

Python中的self等价于C++中的self指针和Java、C#中的this参考。

例:

class Person:

def sayHi(self): # self参数必须写

print('Hello, how are you?')

p = Person()

p.sayHi() # self参数不需赋值

print(p) # 打印: <\_\_main\_\_.Person instance at 0xf6fcb18c> (已经在\_\_main\_\_模块中有了一个Person类的实例)

### 类的变量和对象的变量

类的变量: 由一个类的所有对象(实例)共享使用。当某个对象对类的变量做了改动的时候，这个改动会反映到所有其他的实例上。

对象的变量: 由类的每个对象/实例拥有。它们不是共享的，在同一个类的不同实例中，虽然对象的变量有相同的名称，但是是互不相关的。

使用的数据成员名称以“双下划线前缀”且不是双下划线后缀,比如\_\_privatevar，Python的名称管理体系会有效地把它作为私有变量。

惯例: 如果某个变量只想在类或对象中使用，就应该以单下划线前缀。而其他的名称都将作为公共的，可以被其他类/对象使用。

例:

class Person:

'''Represents a person.'''

population = 0 # 类的变量

def \_\_init\_\_(self, name):

'''Initializes the person's data.'''

# 每创建一个对象都增加1

Person.population += 1 # 调用类的变量,必须用 类名.变量名,如果写 self.变量名 则是对象的变量了

self.name = name # 对象的变量,每个对象独立的

print('(Initializing %s) We have %d persons here.' % (self.name, Person.population))

def \_\_del\_\_(self):

'''I am dying.'''

print('%s says bye.' % self.name)

Person.population -= 1

def sayHi(self):

self.\_\_sayHi2() # 调用私有方法,外部不能调用的

# 以双下划线开头(但没有双下划线结尾),则变成私有,仅供内部调用

def \_\_sayHi2(self): # 使用 self.population 也可以读取类的变量,只是改变的时候却只改变对象的变量

print('Hi, my name is %s. We have %d persons here.' % (self.name, self.population))

swaroop = Person('Swaroop')

swaroop.sayHi() # 打印: Swaroop, 1

kalam = Person('Abdul Kalam')

kalam.sayHi() # 打印: Abdul Kalam, 2

swaroop.sayHi() # 打印: Swaroop, 2

print(Person.population) # 打印: 2

del swaroop # 调用对象的 \_\_del\_\_ 方法

print(Person.population) # 打印: 1

print(Person.\_\_doc\_\_) # 打印类的docstring

print(Person.\_\_init\_\_.\_\_doc\_\_) # 打印类的方法的docstring

### 继承

多态现象:一个子类型在任何需要父类型的场合可以被替换成父类型，即对象可以被视作是父类的实例。

被继承的类被称为“基本类”或“超类”、“父类”。继承的类被称为“导出类”或“子类”。

例:

# 父类

class Member:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

print('(Initialized Member: %s)' % self.name)

def tell(self):

print('Member Name:"%s" Age:"%s"' % (self.name, self.age))

def tell2(self):

print('Member haha...')

# 子类

class Student(Member): # 继承的父类写括号里面;多继承则写多个,这括号的称为继承元组

def \_\_init\_\_(self, name, age, marks):

Member.\_\_init\_\_(self, name, age) # 父类的初始化,需手动写；Python不会自动调用父类的constructor

self.marks = marks

print('(Initialized Student: %s)' % self.name)

def tell(self):

Member.tell(self) # 调用父类的方法,注意:方法调用之前要加上父类名称前缀，然后把self变量及其他参数传递给它。

print('Marks: "%d"' % self.marks)

s = Student('Swaroop', 22, 75)

s.tell() # 会调用子类的方法

s.tell2() # 子类没有的，则使用父类的；如果多继承,且父类都有这个方法,则使用继承元组中排前面的

### 特殊的方法

\_\_init\_\_ 方法

\_\_init\_\_ 方法在类的一个对象被建立时，马上运行。用来对你的对象做初始化。

注意，这个名称的开始和结尾都是双下划线。( \_\_init\_\_ 方法类似于C++、C#和Java中的 constructor )

例:

class Person:

def \_\_init\_\_(self, name):

self.test\_name = name

def sayHi(self):

print('Hello, my name is ' + self.test\_name)

self.test = 'sss' # 属性可以随处定义,不需事先定义

print('the test is ' + self.test)

p = Person('Swaroop')

p.sayHi() # 打印: Swaroop , sss

print('the Person test is ' + p.test) # 打印: sss

p.test2 = 'haha...'

print('the Person test2 is ' + p.test2) # 打印: haha...

名称 说明

\_\_init\_\_(self,...) 这个方法在新建对象恰好要被返回使用之前被调用。

\_\_del\_\_(self) 在对象要被删除之前调用。如使用 del 删除时。

\_\_str\_\_(self) 在我们对对象使用 print 语句或是使用 str() 的时候调用。

\_\_lt\_\_(self,other) 当使用 小于 运算符 (<) 的时候调用。

\_\_gt\_\_(self,other) 当使用 大于 运算符 (>) 的时候调用。

\_\_eq\_\_(self,other) 当使用 等于 运算符 (==) 的时候调用。

\_\_ne\_\_(self,other) 当使用 不等于 运算符 (!=) 的时候调用。

\_\_le\_\_(self,other) 当使用 小于等于 运算符 (<=) 的时候调用。

\_\_ge\_\_(self,other) 当使用 大于等于 运算符 (>=) 的时候调用。

\_\_add\_\_(self,other)当使用 加 运算符 (+) 的时候调用。

\_\_getitem\_\_(self,key) 使用x[key]索引操作符的时候调用。

\_\_len\_\_(self) 对序列对象使用内建的 len() 函数的时候调用。

## try ... except (处理异常)

### 使用 try ... except 语句来处理异常。

except 从句可以专门处理单一的错误或异常，或者一组包括在圆括号内的错误/异常。没有给出错误或异常的名称，则处理所有的错误和异常。

如果某个错误或异常没有被处理，默认的Python处理器就会被调用。它会终止程序的运行，并且打印一个消息。

还可以关联上一个 else 从句,当没有异常发生的时候执行。

常见异常(可避免的):

使用不存在的字典关键字 将引发 KeyError 异常。

搜索列表中不存在的值 将引发 ValueError 异常。

调用不存在的方法 将引发 AttributeError 异常。

引用不存在的变量 将引发 NameError 异常。

未强制转换就混用数据类型 将引发 TypeError 异常。

导入一个并不存在的模块将引发一个 ImportError 异常。

### try ... finally

假如希望在无论异常发生与否的情况下都执行一段代码,可以使用 finally 块来完成。

注意，在一个 try 块下，你可以同时使用 except 从句和 finally 块。

如果在 finally 前面的 try 或者 except, else 等里面有 return 语句,会先跳去执行 finally 再执行 return

### raise 语句

可以使用 raise 语句引发异常。你还得指明错误/异常的名称和伴随异常触发的异常对象。

可以引发 Error 或 Exception 类的直接或间接导出类。

在Python 3里，抛出自定义异常的语法有细微的变化。

Python 2 Python 3

① raise MyException MyException

② raise MyException, 'error message' raise MyException('error message')

③ raise MyException, 'error message', a\_traceback raise MyException('error message').with\_traceback(a\_traceback)

④ raise 'error message' unsupported(不支持)

说明:

① 抛出不带自定义错误信息的异常，这种最简单的形式下，语法没有改变。

② 抛出带自定义错误信息的异常时:Python 2用一个逗号来分隔异常类和错误信息；Python 3把错误信息作为参数传递给异常类。

③ 抛出一个带用户自定义回溯(stack trace,堆栈追踪)的异常。在Python 2和3里这语法完全不同。

④ 在Python 2里，可以仅仅抛出一个异常信息。在Python 3里，这种形式不再被支持。2to3将会警告你它不能自动修复这种语法。

### 生成器的 throw 方法

在Python 2里，生成器有一个 throw()方法。

调用 a\_generator.throw()会在生成器被暂停的时候抛出一个异常，然后返回由生成器函数获取的下一个值。

Python 2 Python 3

① a\_generator.throw(MyException) a\_generator.throw(MyException) # 没有变化

② a\_generator.throw(MyException, 'error message') a\_generator.throw(MyException('error message'))

③ a\_generator.throw('error message') unsupported(不支持)

说明:

① 最简单的形式下，生成器抛出不带用户自定义错误信息的异常。这种情况下，从Python 2到Python 3语法上没有变化 。

② 如果生成器抛出一个带用户自定义错误信息的异常，你需要将这个错误信息字符串(error string)传递给异常类来以实例化它。

③ Python 2还支持抛出只有异常信息的异常。Python 3不支持这种语法，并且2to3会显示一个警告信息，告诉你需要手动地来修复这处代码。

例(3.x)语法:

# 定义一个异常类,继承 Exception

class ShortInputException(Exception):

'''A user-defined exception class.'''

def \_\_init\_\_(self, length, atleast):

Exception.\_\_init\_\_(self)

self.length = length

self.atleast = atleast

try:

s = input('Enter something --> ') # Python 2 的输入是 raw\_input()

if len(s) < 3:

raise ShortInputException(len(s), 3) # 引发异常;Python 2可以写：raise ShortInputException,(len(s), 3)

# 捕获 EOFError 异常

except EOFError:

print('\nWhy did you do an EOF on me?')

# 捕获一组错误/异常,Python 2 时应该写: “except (RuntimeError, ImportError), e:”

except (RuntimeError, ImportError) as e:

pass

# Python 2 时应该写: “except ShortInputException, x:”

except ShortInputException as x:

print('ShortInputException: The input was of length %d,\

was expecting at least %d' % (x.length, x.atleast))

# 捕获所有异常

except:

print('\nWhy did you do an Exception on me?')

# 没有任何异常时执行

else:

print('No exception was raised.')

# 不管是否有异常,都会执行

finally:

print('finally .....')

## lambda 形式

lambda 语句被用来创建新的函数对象，并且在运行时返回它们。

注意, lambda 形式中，只能使用表达式。

例:

def make\_repeater(n):

return lambda s: s\*n # 注意: lambda 返回的是函数,而不是表达式的值

# 注意, twice 接收的是函数, 而不是表达式的值, 所以 twice 是一个函数,而不是值

twice = make\_repeater(2)

print(twice('word ')) # 因为 twice 是一个函数,这里是调用这个函数,打印结果: word word

print(make\_repeater(3)(5)) # 这里的“make\_repeater(3)”可以认为是匿名函数,打印结果: 15

# 上面例子貌似太过复杂,下面是简单点的写法

# 记住, twice2 是一个函数

twice2 = lambda s: s\*2

print(twice2('word ')) # 打印: word word

print(twice2(5)) # 打印: 10

## exec 和 eval

exec 用来执行储存在字符串或文件中的Python语句。

eval 用来计算存储在字符串中的有效Python表达式。

exec 跟 eval 是相似的，但是 exec 更加强大并更具有技巧性。

eval 只能执行单独一条表达式；但是 exec 能够执行多条语句，导入(import)，函数声明

实际上 exec 能执行整个Python程序的字符串。

Python 2 与 Python 3 的比较

Python 2 Python 3

① exec codeString exec(codeString)

② exec codeString in global\_namespace exec(codeString, global\_namespace)

③ exec codeString in global\_namespace, local\_namespace exec(codeString, global\_namespace, local\_namespace)

说明:

① 就像 print 语句在Python 3里变成了一个函数一样, exec 语句在Python 3里也变成一个函数。

② exec 可以指定名字空间，代码将在这个由全局对象组成的私有空间里执行。

③ exec 还可以指定一个本地名字空间(比如一个函数里声明的变量)。

例：

exec('print("Hello World")') # 执行打印语句

print(eval('2\*3')) # 打印：6

## execfile 语句

Python 2里的 execfile 语句也可以像执行Python代码那样使用字符串。不同的是 exec 使用字符串，而 execfile 则使用文件。

在Python 3里,execfile 语句已经被去掉了。如果你真的想要执行一个文件里的Python代码(但是你不想导入它)，你可以通过打开这个文件，读取它的内容，然后调用 compile()全局函数强制Python解释器编译代码，然后调用 exec() 函数。

Python 2 写的： execfile('a\_filename')

Python 3 写的： exec(compile(open('a\_filename', 'rb').read(), 'a\_filename', 'exec'))

## assert 语句

assert 语句用来声明某个条件是真的。

当 assert 语句失败的时候，会引发一个 AssertionError 错误。

比较常用于检验错误。

例:

assert 2 >= 1 # 正常运行

assert 0 >= 1 # 出现错误

## repr 函数

repr 函数用来取得对象的规范字符串表示。反引号（也称转换符）可以完成相同的功能。

注意，在大多数时候有 eval(repr(object)) == object。

基本上, repr 函数和反引号用来获取对象的可打印的表示形式。

你可以通过定义类的 \_\_repr\_\_ 方法来控制你的对象在被repr函数调用的时候返回的内容。

例：

i = ["item"]

print(repr(i)) # 打印：['item']

## zip() and \_

>>> a = [1, 2]

>>> b = ['a', 'b']

>>> zip(a,b)

[(1, 'a'), (2, 'b')]

>>> zip(a,b,a)

[(1, 'a', 1), (2, 'b', 2)]

>>> zip ([1], b)

[(1, 'a')]

>>> a = [(1, [2, 3]), (4, [5, 6])]

>>> **for** i, (x, [\_, y]) **in** enumerate(a):

... **print** i, x, y

...

0 1 3

1 4 6

## 正则表达式

正则表达式有强大并且标准化的方法来处理字符串查找、替换以及用复杂模式来解析文本。

正则表达式的语法比程序代码更紧凑，格式更严格，比用组合调用字符串处理函数的方法更具有可读性。

还可以在正则表达式中嵌入注释信息，这样就可以使它有自文档化的功能。

### 匹配符：

^ 匹配字符串开始位置。在多行字符串模式匹配每一行的开头。

$ 匹配字符串结束位置。在多行字符串模式匹配每一行的结尾。

. 匹配除了换行符外的任何字符，在 alternate 模式（re.DOTALL）下它甚至可以匹配换行。

\A 匹配字符串开头

\Z 匹配字符串结尾

\b 匹配一个单词边界。即 \w 与 \W 之间。

\B 匹配一个非单词边界；相当于类 [^\b]。

\d 匹配一个数字。

\D 匹配一个任意的非数字字符。

\s 匹配任何空白字符；它相当于类 [ \t\n\r\f\v]。

\S 匹配任何非空白字符；它相当于类 [^ \t\n\r\f\v]。

\w 匹配任何字母数字字符；它相当于类 [a-zA-Z0-9\_]。

\W 匹配任何非字母数字字符；它相当于类 [^a-zA-Z0-9\_]。

x? 匹配可选的x字符。即是0个或者1个x字符。

x\* 匹配0个或更多的x。

x+ 匹配1个或者更多x。

x{n,m} 匹配n到m个x，至少n个，不能超过m个。

(a|b|c) 匹配单独的任意一个a或者b或者c。

(x) 捕获组，小括号括起来即可，它会记忆它匹配到的字符串。

可以用 re.search() 返回的匹配对象的 groups()函数来获取到匹配的值。

\1 记忆组，它表示记住的第一个分组；如果有超过一个的记忆分组，可以使用 \2 和 \3 等等。

记忆组的内容也要小括号括起来。

(?iLmsux) iLmsux的每个字符代表一种匹配模式

re.I(re.IGNORECASE): 忽略大小写（括号内是完整写法，下同）

re.M(re.MULTILINE): 多行模式，改变'^'和'$'的行为（参见上图）

re.S(re.DOTALL): 点任意匹配模式，改变'.'的行为

re.L(re.LOCALE): 使预定字符类 \w \W \b \B \s \S 取决于当前区域设定

re.U(re.UNICODE): 使预定字符类 \w \W \b \B \s \S \d \D 取决于unicode定义的字符属性

re.X(re.VERBOSE): 松散正则表达式模式。这个模式下正则表达式可以是多行，忽略空白字符，并可以加入注释。

(?:表达式) 无捕获组。与捕获组表现一样，只是没有内容。

(?P<name>表达式) 命名组。与记忆组一样，只是多了个名称。

(?P=name) 命名组的逆向引用。

(?#...) “#”后面的将会作为注释而忽略掉。例如：“ab(?#comment)cd”匹配“abcd”

(?=...) 之后的字符串需要匹配表达式才能成功匹配。不消耗字符串内容。 例：“a(?=\d)”匹配“a12”中的“a”

(?!...) 之后的字符串需要不匹配表达式才能成功匹配。不消耗字符串内容。 例：“a(?!\d)”匹配“abc”中的“a”

(?<=...) 之前的字符串需要匹配表达式才能成功匹配。不消耗字符串内容。 例：“(?<=\d)a”匹配“2a”中的“a”

(?<!...) 之前的字符串需要不匹配表达式才能成功匹配。不消耗字符串内容。 例：“(?<!\d)a”匹配“sa”中的“a”

注：上面4个表达式的里面匹配的内容只能是一个字符，多个则报错。

(?(id/name)yes-pattern|no-pattern) 如果编号为 id 或者别名为 name 的组匹配到字符串，则需要匹配yes-pattern，否则需要匹配no-pattern。 “|no-pattern”可以省略。如：“(\d)ab(?(1)\d|c)”匹配到“1ab2”和“abc”

### 元字符:

"[" 和 "]"

它们常用来匹配一个字符集。字符可以单个列出，也可以用“-”号分隔的两个给定字符来表示一个字符区间。

例如，[abc] 将匹配"a", "b", 或 "c"中的任意一个字符；也可以用区间[a-c]来表示同一字符集，和前者效果一致。

元字符在类别里并不起作用。例如，[a$]将匹配字符"a" 或 "$" 中的任意一个；"$" 在这里恢复成普通字符。

也可以用补集来匹配不在区间范围内的字符。其做法是把"^"作为类别的首个字符；其它地方的"^"只会简单匹配 "^"字符本身。

例如，[^5] 将匹配除 "5" 之外的任意字符。

特殊字符都可以包含在一个字符类中。如，[\s,.]字符类将匹配任何空白字符或","或"."。

反斜杠 "\"。

做为 Python 中的字符串字母，反斜杠后面可以加不同的字符以表示不同特殊意义。

它也可以用于取消所有的元字符，这样你就可以在模式中匹配它们了。

例如，需要匹配字符 "[" 或 "\"，可以在它们之前用反斜杠来取消它们的特殊意义： \[ 或 \\。

建议使用原始字符串:

建议在处理正则表达式的时候总是使用原始字符串。如： r'\bROAD$', 而不要写成 '\\bROAD$'

否则，会因为理解正则表达式而消耗大量时间(正则表达式本身就已经够让人困惑的了)。

### 无捕获组:

有时你想用一个组去收集正则表达式的一部分，但又对组的内容不感兴趣。你可以用一个无捕获组“(?:...)”来实现这项功能。

除了捕获匹配组的内容之外，无捕获组与捕获组表现完全一样；可以在其中放置任何字符、匹配符，可以在其他组(无捕获组与捕获组)中嵌套它。

无捕获组对于修改已有组尤其有用，因为可以不用改变所有其他组号的情况下添加一个新组。

捕获组和无捕获组在搜索效率方面也一样。

### 命名组:

与用数字指定组不同的是，它可以用名字来指定。除了该组有个名字之外，命名组也同捕获组是相同的。

(?P<name>...) 定义一个命名组，(?P=name) 则是对命名组的逆向引用。

MatchObject 的方法处理捕获组时接受的要么是表示组号的整数，要么是包含组名的字符串。所以命名组可以通过数字或者名称两种方式来得到一个组的信息。

### 松散正则表达式:

为了方便阅读和维护，可以使用松散正则表达式，它与普通紧凑的正则表达式有两点不同：

1, 空白符被忽略。空格、制表符(tab)和回车会被忽略。如果需要匹配他们，可以在前面加一个“\”来转义。

2, 注释被忽略。注释以“#”开头直到行尾，与python代码中的一样。

使用松散正则表达式，需要传递一个叫 re.VERBOSE的 参数。详细见下面例子。

例如:

# 必须引入 re 标准库

import re

# 字符串替换: sub() 与 subn()

s = '100 NORTH MAIN ROAD'

# 将字符串结尾的单词“ROAD”替换成“RD.”；该 re.sub() 函数执行基于正则表达式的字符串替换。

print(re.sub(r'\bROAD$', 'RD.', s)) # 打印： 100 NORTH MAIN RD.

## subn() 与 sub() 作用一样，但返回的是包含新字符串和替换执行次数的两元组。

print(re.subn(r'\bROAD$', 'RD.', s)) # 打印： ('100 NORTH MAIN RD.', 1)

# 字符串分割, split()

# 在正则表达式匹配的地方将字符串分片，将返回列表。只支持空白符和固定字符串。可指定最大分割次数，不指定将全部分割。

print(re.split(r'\s+', 'this is a test')) # 打印： ['this', 'is', 'a', 'test']

print(re.split(r'\W+', 'This is a test.', 2)) # 指定分割次数,打印：['this', 'is', 'a test']

# 如果你不仅对定界符之间的文本感兴趣，也需要知道定界符是什么。在 RE 中使用捕获括号，就会同时传回他们的值。

print(re.split(r'(\W+)', 'This is a test.', 2)) # 捕获定界符,打印：['this', ' ', 'is', ' ', 'a test']

## `MatchObject` 实例的几个方法

r = re.search(r'\bR(OA)(D)\b', s)

print(r.groups()) # 返回一个包含字符串的元组,可用下标取元组的内容，打印： ('OA', 'D')

print(r.group()) # 返回正则表达式匹配的字符串，打印： ROAD

print(r.group(2)) # 返回捕获组对应的内容(用数字指明第几个捕获组)，打印： D

print(r.start()) # 返回匹配字符串开始的索引, 打印： 15

print(r.end()) # 返回匹配字符串结束的索引，打印： 19

print(r.span()) # 返回一个元组包含匹配字符串 (开始,结束) 的索引，打印： (15, 19)

# 匹配多个内容, findall() 返回一个匹配字符串行表

p = re.compile('\d+')

s0 = '12 drummers drumming, 11 pipers piping, 10 lords a-leaping'

print(p.findall(s0)) # 打印： [12, 11, 10]

print(re.findall(r'\d+', s0)) # 也可这样写，打印： [12, 11, 10]

# 匹配多个内容, finditer() 以迭代器返回

iterator = p.finditer(s0)

# iterator = re.finditer(r'\d+', s0) # 上句也可以这样写

for match in iterator:

print(match.group()) # 三次分别打印：12、 11、 10

# 记忆组

print(re.sub('([^aeiou])y$', 'ies', 'vacancy')) # 将匹配的最后两个字母替换掉，打印： vacanies

print(re.sub('([^aeiou])y$', r'\1ies', 'vacancy')) # 将匹配的最后一个字母替换掉，记忆住前一个(小括号那部分)，打印： vacancies

print(re.search('([^aeiou])y$', 'vacancy').group(1)) # 使用 group() 函数获取对应的记忆组内容，打印： c

# 记忆组(匹配重复字符串)

p = re.compile(r'(?P<word>\b\w+)\s+\1') # 注意, re.match() 函数不能这样用，会返回 None

p = p.search('Paris in the the spring')

# p = re.search(r'(?P<word>\b\w+)\s+\1', 'Paris in the the spring') # 这一句可以替换上面两句

print(p.group()) # 返回正则表达式匹配的所有内容，打印： the the

print(p.groups()) # 返回一个包含字符串的元组，打印： ('the',)

# 捕获组

r = re.search(r'\bR(OA)(D)\b', s) # 如过能匹配到，返回一个 SRE\_Match 类(正则表达式匹配对象)；匹配不到则返回“None”

# `MatchObject` 实例的几个方法

if r: # 如果匹配不到，则 r 为 None,直接执行下面语句则会报错；这里先判断一下，避免这错误

print(r.groups()) # 返回一个包含字符串的元组,可用下标取元组的内容，打印： ('OA', 'D')

print(r.group()) # 返回正则表达式匹配的字符串，打印： ROAD

print(r.group(2)) # 返回捕获组对应的内容(用数字指明第几个捕获组)，打印： D

# 无捕获组

print(re.match("([abc])+", "abcdefab").groups()) # 正常捕获的结果： ('c',)

print(re.match("(?:[abc])+", "abcdefab").groups()) # 无捕获组的结果： ()

# 命名组

m = re.match(r'(?P<word>\b\w+\b) \*(?P<word2>\b\w+\b)', 'Lots of punctuation')

print(m.groups()) # 返回正则表达式匹配的所有内容，打印：('Lots', 'of')

print(m.group(1)) # 通过数字得到对应组的信息，打印： Lots

print(m.group('word2')) # 通过名称得到对应组的信息，打印： of

# 命名组 逆向引用

p = re.compile(r'(?P<word>\b\w+)\s+(?P=word)') # 与记忆组一样用法, re.match() 函数同样不能这样用，会返回 None

p = p.search('Paris in the the spring') # r'(?P<word>\b\w+)\s+(?P=word)' 与 r'(?P<word>\b\w+)\s+\1' 效果一样

print(p.group()) # 返回正则表达式匹配的所有内容，打印： the the

print(p.groups()) # 返回一个包含字符串的元组，打印： ('the',)

# 使用松散正则表达式,以判断罗马数字为例

pattern = '''

^ # beginning of string

(M{0,3}) # thousands - 0 to 3 Ms

(CM|CD|D?C{0,3}) # hundreds - 900 (CM), 400 (CD), 0-300 (0 to 3 Cs),

# or 500-800 (D, followed by 0 to 3 Cs)

(XC|XL|L?X{0,3}) # tens - 90 (XC), 40 (XL), 0-30 (0 to 3 Xs),

# or 50-80 (L, followed by 0 to 3 Xs)

(IX|IV|V?I{0,3}) # ones - 9 (IX), 4 (IV), 0-3 (0 to 3 Is),

# or 5-8 (V, followed by 0 to 3 Is)

$ # end of string

'''

print(re.search(pattern, 'M')) # 这个没有申明为松散正则表达式，按普通的来处理了，打印： None

print(re.search(pattern, 'M', re.VERBOSE).groups()) # 打印： ('M', '', '', '')

# (?iLmsux) 用法

# 以下这三句的写法都是一样的效果，表示忽略大小写，打印： ['aa', 'AA']

print(re.findall(r'(?i)(aa)', 'aa kkAAK s'))

print(re.findall(r'(aa)', 'aa kkAAK s', re.I))

print(re.findall(r'(aa)', 'aa kkAAK s', re.IGNORECASE))

# 可以多种模式同时生效

print(re.findall(r'(?im)(aa)', 'aa kkAAK s')) # 直接在正则表达式里面写

print(re.findall(r'(aa)', 'aa kkAAK s', re.I | re.M)) # 在参数里面写

print(re.findall(r'(aa)', 'aa kkAAK s', re.I or re.M))

# 预编译正则表达式解析的写法

# romPattern = re.compile(pattern) # 如果不是松散正则表达式,则这样写,即少写 re.VERBOSE 参数

romPattern = re.compile(pattern, re.VERBOSE)

print(romPattern.search('MCMLXXXIX').groups()) # 打印： ('M', 'CM', 'LXXX', 'IX')

print(romPattern.search('MMMDCCCLXXXVIII').groups()) # 打印： ('MMM', 'DCCC', 'LXXX', 'VIII')

# match()、search()、sub()、findall() 等等都可以这样用

### match() vs search()

match() 函数只检查 RE 是否在字符串开始处匹配，而 search() 则是扫描整个字符串。记住这一区别是重要的。

match() 只报告一次成功的匹配，它将从 0 处开始；如果匹配不是从 0 开始的， match() 将不会报告它。

search() 将扫描整个字符串，并报告它找到的第一个匹配。

例：

print(re.match('super', 'superstition').span()) # 打印： (0, 5)

print(re.match('super', 'insuperable')) # 打印： None

print(re.search('super', 'superstition').span()) # 打印： (0, 5)

print(re.search('super', 'insuperable').span()) # 打印： (2, 7)

## Python标准库:

os 模块

这个模块包含普遍的操作系统功能。如果你希望你的程序能够与平台无关的话，这个模块是尤为重要的。

os.sep 获取操作系统特定的路径分割符。比如在Linux、Unix下它是'/'，在Windows下它是'\\'，而在Mac OS下它是':'。

os.name 字符串指示你正在使用的平台。比如对于Windows，它是'nt'，而对于Linux/Unix用户，它是'posix'。

os.getcwd() 函数得到当前工作目录，即当前Python脚本工作的目录路径。

os.getenv(key) 函数用来读取环境变量。

os.putenv(key, value) 函数用来设置环境变量。

os.listdir(path) 返回指定目录下的所有文件和目录名。

os.remove(filePath) 函数用来删除一个文件。

os.system(shellStr) 函数用来运行shell命令。

os.linesep 字符串给出当前平台使用的行终止符。例如，Windows使用'\r\n'，Linux使用'\n'而Mac使用'\r'。

os.path.split(pathname) 函数返回一个路径的目录名和文件名。

os.path.isfile(path) 函数检验给出的路径是否一个文件。

os.path.isdir(path) 函数分别检验给出的路径是否目录。

os.path.existe(path) 函数用来检验给出的路径是否真地存在。

#####################################################

# 范例:

## 运行系统命令行和调用外部程序

import os

os\_command = 'echo haha...'

# 运行命令行,返回运行结果(成功时返回0,失败返回1或以上的出错数字)

result = os.system(os\_command)

if result == 0:

print('run Successful')

else:

print('run FAILED')

# 注:os.system()函数不推荐使用,它容易引发严重的错误。(可能是因为不具备可移植性)

windows下，调用其它应用程序

os.startfile(path)

推荐

os.popen()

## 获取系统时间

import time

print(time.strftime('%Y-%m-%d %H:%M:%S')) # 打印: 2011-02-15 16:50:57

print(time.time()) # 以浮点数形式返回自Linux新世纪以来经过的秒数。

print(time.ctime(1150269086.6630149)) #time.ctime([sec]) 把秒数转换成日期格式，如果不带参数，则显示当前的时间。

# 计算系统日期的前一天

import datetime

# 得到今天的日期

print(datetime.date.today())

# 得到前一天的日期

print(datetime.date.today() + datetime.timedelta(days=-1))

# 或者

print(datetime.date.today() - datetime.timedelta(days=1))

其中各字符含义:(举例数字以上面的时间为准)

%A: 星期(全称),如:Tuesday %a: 星期(简称),如:Tue

%B: 月份(全称),如:February %b: 月份(简称),如:Feb

%c: 月日年 时分秒,如:02/15/11 16:50:57

%d: 日期,如:15

%H: 小时,如:16

%I: (未知),如:05

%j: (未知),如:046

%M: 分钟,如:50 %m: 月份(数字显示),如:02

%p: 上下午,如:PM

%S: 秒钟,如:57

%U: (未知),如:07

%W: (未知),如:07 %w: 星期(数字显示),如:2 (星期天为0)

%X: 时分秒,如:16:50:57 %x: 月日年,如:02/15/11

%Y: 公历年(4位),如:2011 %y: 年(后两位),如:11

%Z: 如:中国标准时间 %z: 如:中国标准时间

## 文件和目录操作

### 创建目录

import os

pathDir = r'D:\Work' # 不同系统的目录写法有所不同

if not os.path.exists(pathDir):

os.mkdir(pathDir) # 创建目录, os.makedirs(pathDir) 创建多个不存在的目录

target = pathDir + os.sep + 'test.txt'

print(target)

# 注意os.sep变量的用法, os.sep 是目录分隔符,这样写方便移植。即在Linux、Unix下它是'/'，在Windows下它是'\\'，而在Mac OS下它是':'。

### 获取当前路径

**import** os

***# 方法一***

**print** os.getcwd()

***# 方法二***

**print** os.path.abspath(os.curdir)

***# 方法三***

**print** os.path.abspath('.')

### 获取特定路径下的文件名和子文件夹名

# 返回一个目录下的文件名及子文件夹名

# Return a list containing the names of the entries in the directory. The list is in arbitrary order.

# It does not include the special entries '.' and '..' even if they are present in the

# directory. Availability: Macintosh, Unix, Windows.

os.listdir

# 获取给定目录下所有文件，包含子文件夹的，递归进行

# walk() generates the file names in a directory tree, by walking the tree either top down or bottom up.

# For each directory in the tree rooted at directory *top* (including *top* itself), it yields a 3-tuple

# (*dirpath*, *dirnames*, *filenames*).

os.walk

os.path.isfile # 是否为文件

os.path.isdir # 是否为文件夹

### 复制文件或文件夹

import shutil

# 复制file1 到 file2

shutil.copy(file1, file2)

# 复制目录dir1 到 dir2

shutil.copytree(dir1, dir2)

# 还可参考copy, copy2等函数

### 文件操作(读写txt文件)

filePath = 'poem.txt'

f = open(filePath, 'w') # 以写的模式打开文件,Python 2.x 需将 open() / io.open() 改成 file()

for a in range( 0, 10 ):

s = "%5d %5d\n" % (a, a\*a)

f.write( s ) # 把文本写入文件

f.close() # 关闭io流

f2 = open(filePath) # 没有提供模式，则默认是读取,即 'r'

while True:

line = f2.readline()

if len(line) == 0: # 读取结束

break

print(line, end=' ') # 避免print自动换行, 此行Python2.x应该写：“print line,”

f2.close() # close the file

# 删除文件

import os

os.remove(filePath)

#### 说明:

一、在pythony 3.0 已经废弃了 file 类。

二、pythony 3.0 内置 open() 函数的构造函数是:

open(file, mode="r", buffering=None, encoding=None, errors=None, newline=None, closefd=True)

1.mode(模式):

r: 读，只能读文件，如果文件不存在，会发生异常

w: 写，只能写文件，如果文件不存在，创建该文件；如果文件已存在，先清空，再打开文件

a: 打开供追加

b: 二进制模式；一般是组合写法,如: rb 以二进制读方式打开；wb 以二进制写方式打开

t: 文本模式

+: 打开一个磁盘文件供更新,一般是组合使用,如:

rb+: 以二进制读方式打开，可以读、写文件，如果文件不存在，会发生异常

wb+: 以二进制写方式打开，可以读、写文件，如果文件不存在，创建该文件；如果文件已存在，先清空，再打开文件

u: 通用换行模式

默认的模式是 rt，即打开供读取的文本模式。

2.buffering 关键字参数的期望值是以下三个整数中的一个以决定缓冲策略：

0: 关闭缓冲

1: 行缓冲

> 1: 所填的 int 数=缓冲区大小

默认: 完全缓冲

3.encoding 默认的编码方式独立于平台。

4.关闭文件描述符 closefd 可以是 True 或 False 。

如果是 False,此文件描述符会在文件关闭后保留。若文件名无法奏效的话，那么必须设为 True 。

三、清空文件内容

f.truncate()

注意：当以 "r+","rb+","w","wb","wb+"等模式时可以执行该功能，即具有可写模式时才可以。

四、文件的指针定位与查询

（1）文件指针：

文件被打开后，其对象保存在 f 中， 它会记住文件的当前位置,以便于执行读、写操作，

这个位置称为文件的指针( 一个从文件头部开始计算的字节数 long 类型 )。

（2）文件打开时的位置:

以"r","r+","rb+" 读方式, "w","w+","wb+"写方式 打开的文件，

一开始，文件指针均指向文件的头部。

（3）获取文件指针的值:

L = f.tell()

（4）移动文件的指针

f.seek(偏移量, 选项) # 偏移量 是 long 或者 int 类型，计算偏移量时注意换行符是2,汉字可能是2或3

选项 =0 时， 表示将文件指针指向从文件头部到 "偏移量"字节处。

选项 =1 时， 表示将文件指针指向从文件的当前位置，向后移动 "偏移量"字节。

选项 =2 时， 表示将文件指针指向从文件的尾部，，向前移动 "偏移量"字节。

五、从文件读取指内容

1.文本文件（以"rt"方式打开的文件）的读取

s = f.readline()

返回值： s 是字符串，从文件中读取的一行，含行结束符。

说明: (1)如果 len(s) = 0 表示已到文件尾(换行符也是有长度的,长度为2)

(2)如果是文件的最后一行，有可能没有行结束符

2.二进制文件（以"rb"、"rb+"、"wb+" 方式打开的文件）的读取

s = f.read(n)

说明: (1)如果 len( s ) =0 表示已到文件尾

(2)文件读取后，文件的指针向后移动 len(s) 字节。

(3)如果磁道已坏，会发生异常。

六、向文件写入一个字符串

f.write( s )

参数: s 要写入的字符串

说明: (1)文件写入后，文件的指针向后移动 len(s) 字节。

(2)如果磁道已坏，或磁盘已满会发生异常。

### 文件操作(遍历目录和文件名)

import os

import os.path

rootdir = r"D:\Holemar\1.notes\28.Python\test"

# os.walk 返回一个三元组，其中parent表示所在目录, dirnames是所有目录名字的列表, filenames是所有文件名字的列表

for parent,dirnames,filenames in os.walk(rootdir):

# 所在目录

print("parent is:" + parent)

# 遍历此目录下的所有目录(不包含子目录)

for dirname in dirnames:

print(" dirname is:" + dirname)

# 遍历此目录下的所有文件

for filename in filenames:

print(" filename with full path:" + os.path.join(parent, filename))

### 文件操作(分割路径和文件名)

import os.path

#常用函数有三种：分隔路径，找出文件名，找出盘符(window系统)，找出文件的扩展名。

spath = "d:/test/test.7z"

# 下面三个分割都返回二元组

# 分隔目录和文件名

p,f = os.path.split(spath) # 注意二元组的接收

print("dir is:" + p) # 打印: d:/test

print(" file is:" + f) # 打印: test.7z

# 分隔盘符和文件名

drv,left = os.path.splitdrive(spath)

print(" driver is:" + drv) # 打印: d:

print(" left is:" + left) # 打印: /test/test.7z

# 分隔文件和扩展名

f,ext = os.path.splitext(spath)

print(" f is: " + f) # 打印: d:/test/test

print(" ext is:" + ext) # 打印: 7z

### 储存器

pickle标准模块。它可以在一个文件中储存任何Python对象，之后又可以把它完整无缺地取出来。这被称为 持久地 储存对象。

在pythony 3.0 已经移除了 cPickle 模块，可以使用 pickle 模块代替。

import pickle as p # 这里使用 as 简称,方便更改模块时只需改一行代码

# import cPickle as p # Python 2.x 有这个模块（比pickle快1000倍）

# 将会把资料保存在这个文件里面

shoplistfile = 'shoplist.data'

# 需要保存的资料

shoplist = ['apple', 'mango', 'carrot', 2, 5]

# 写入文件

f = open(shoplistfile, "wb") # 以二进制写入,Python2.x时可不用二进制,但3.x必须

p.dump(shoplist, f) # dump the object to a file

f.close()

# 取出资料

f = open(shoplistfile, "rb") # 以二进制读取

storedlist2 = p.load(f)

print(storedlist2)

f.close()

# 删除文件

import os

os.remove(shoplistfile)

## 数据库连接

cx\_Oracle : 是一个用来连接并操作 Oracle 数据库的 Python 扩展模块， 支持包括 Oracle 9.2 10.2 以及 11.1 等版本。

MySQLdb : MySQL 数据库的 Python 扩展模块

## 编码问题

### 得到系统当前编码

>>> **import** locale

>>> locale.getpreferredencoding()

### 中文字符串按字分开Python2.x

*# encoding 是你字符串的编码,如 gbk,cp936, gb18030, utf-8等*  
list(unicode(*YourString*, '*encoding*'))

例如：utf-8编码的字符串“你好”

>>>list(unicode('你好', 'utf-8')  
['你', '好']

### 判断是否为中文字符

**方法一：**利用utf-8 编码长度不同判断

>>> s = list(unicode('你好nihao', 'utf-8'))  
>>> s  
[u'**\u4f60**', u'**\u597d**', u'n', u'i', u'h', u'a', u'o']  
>>> [ len(i.encode('utf8')) **for** i **in** s]  
[3, 3, 1, 1, 1, 1, 1]

**方法二：**

>>> a = unicode('中华人民共和国', 'utf-8')  
>>> a  
u'**\u4e2d\u534e\u4eba\u6c11\u5171\u548c\u56fd**'  
  
>>> c = repr(a)  
>>> c  
"u'**\\**u4e2d**\\**u534e**\\**u4eba**\\**u6c11**\\**u5171**\\**u548c**\\**u56fd'"  
>>> c = c[2:-1]  
>>> c  
'**\\**u4e2d**\\**u534e**\\**u4eba**\\**u6c11**\\**u5171**\\**u548c**\\**u56fd'  
>>> c.split('**\\**u')  
['', '4e2d', '534e', '4eba', '6c11', '5171', '548c', '56fd']  
# 对于每个字符，用函数：

int(x, 16) 转化为数字

然后用以下函数进行判断

具体编码可以查看http://www.unicode.org/charts/

**def** isch(x):  
    *# Punct & Radicals*  
    **if** x >= 0x2e80 **and** x <= 0x33ff:  
        **return** 1  
  
    *# Fullwidth Latin Characters*  
    **if** x >= 0xff00 **and** x <= 0xffef:  
        **return** 1  
  
    *# CJK Unified Ideographs &*  
    *# CJK Unified Ideographs Extension A*  
    **if** x >= 0x4e00 **and** x <= 0x9fbb:  
        **return** 1  
    *# CJK Compatibility Ideographs*  
    **if** x >= 0xf900 **and** x <= 0xfad9:  
        **return** 1  
  
    *# CJK Unified Ideographs Extension B*  
    **if** x >= 0x20000 **and** x <= 0x2a6d6:  
        **return** 1  
  
    *# CJK Compatibility Supplement*  
    **if** x >= 0x2f8000 **and** x <= 0x2fa1d:  
        **return** 1  
  
    **return** 0

### 编码转换

**unicode 🡪 其它编码**

例如：a为unicode编码 要转为gb2312。

a.encode('gb2312')

**其它编码 🡪 unicode**

例如：a为gb2312编码，要转为unicode.

unicode(a, 'gb2312')

*# or*  
a.decode('gb2312')

**编码1 🡪 编码2**

可以先转为unicode再转为编码2

如gb2312转big5

unicode(a, 'gb2312').encode('big5')

**UTF8 🡨🡪Unicode**

bts = b"hello" #bytes对象

str = "hello" #字符串

decode可以把bytes对象解码成unicode

bts.decode("utf8") -> str

encode可以把unicode字符串编码成utf8等各种编码的bytes对象

str.encode("utf8") -> bts

**判断字符串的编码**

*# 用来判断是否为一般字符串*  
isinstance(yourstring, str)   
  
*# 用来判断是否为unicode*  
isinstance(yourstring, unicode)

如果一个字符串已经是unicode了，再执行unicode转换有时会出错(并不都出错)

可以写一个通用的转成unicode函数：

**def** u(s, encoding):  
    **if** isinstance(s, unicode):  
        **return** s  
    **else**:  
        **return** unicode(s, encoding)

*除上以上的编码方法*，在读写文件时还可以使用codecs的open方法在读写时进行转换。

**import** codecs

data = open("Test.txt").read()

**if** data[:3] == codecs.BOM\_UTF8: *# UTF8 BOM*

data = data[3:]

**print** data.decode("utf-8")

### 检测未知编码

**import** chardet

chardet.detect(*your string*)

### UnicodeDecodeError

在Python开发过程中，常常会遇到这样的问题，很让人头疼：

......

print s

......

Traceback (most recent call last):

  File "<stdin>", line 1, in ?

UnicodeDecodeError: 'ascii' codec can't decode byte 0xe4 in position 0: ordinal not in range(128)

究其原因，是想要在屏幕上输出的字符，不符合当前设定终端编码。在上面这个例子中，当前终端的编码设置为ascii，而欲输出的字符为utf-8编码，ascii的每个字符的十进制编码都在0-128以内，当然无法识别utf-8的编码。

要解决这个问题，首先修改终端的字符编码设置：

>>> import sys

>>> sys.getdefaultencoding()

'ascii'

可以看到，当前的默认编码是ascii，要修改这个设置，可以在Python安装目录下的Lib/site-packages目录中，新建一个sitecustomize.py文件（也可以建在其它地方，然后手工导入，建在这里，每次启动Python的时候设置将自动生效），内容如下：

import sys

sys.setdefaultencoding('utf-8') #set default encoding to utf-8

然后可以查看到改变已经生效

>>> import sys

>>> sys.getdefaultencoding()

'utf-8'

此时运行程序，如果仍然报告之前的错误，只需要显示地设定输出的编码

print s.encode('utf-8')

就可以看到正确显示。

### 得到中文字符的GB2312编码

以中文字符‘啊’为例：

**print** ''.join(`'啊'`.split('**\\**x')).replace("'","")

### 转换文件编码

import codecs

# 用codecs提供的open方法来指定打开的文件的语言编码，它会在读取的时候自动转换为内部Unicode

bfile = codecs.open("dddd.txt", 'r', "gbk")

ss = bfile.read()  
bfile.close()

# 输出，这个时候看到的就是转换后的结果。如果使用语言内建的open函数来打开文件，这里看到的必定是乱码print ss, type(ss)

### Unicode介绍

为什么不所有的文件都使用unicode，还要用GBK，utf-8等编码呢？

unicode可以称为抽象编码，也就是它只是一种内部表示，一般不能直接保存。保存到磁盘上时，需要把它转换为对应的编码，如utf-8和utf-16。

Windows 的unicode指utf-16

### 查看Python内部Unicode格式

    python的内部是使用unicode来处理的，但是unicode的使用需要考虑的是它的编码格式有两种，一是UCS-2，它一共有65536个码位，另一种是UCS-4，它有2147483648g个码位。对于这两种格式，python都是支持的，这个是在编译时通过--enable- unicode=ucs2或--enable-unicode=ucs4来指定的。那么我们自己默认安装的python有的什么编码怎么来确定呢？有一个办法，就是通过sys.maxunicode的值来判断：

import sys  
print sys.maxunicode

    如果输出的值为65535,那么就是UCS-2,如果输出是1114111就是UCS-4编码。

### 各场景下的编码

#coding:utf-8 #由于.py文件是utf-8的，所以必须有这一句

#系统编码： 默认编码，正常情况下window系统默认是gbk,linux系统默认是utf-8，可用locale.getdefaultlocale()和locale.setdefaultlocale()来控制，与encode有关

import locale

print(locale.getpreferredencoding())

#代码编码：python2代码中的编码，默认是ascii，可用"# -\*- coding: utf-8 -\*-"这种方式指定。

import sys

print(sys.getdefaultencoding())

#文件编码：

import sys

print(sys.getfilesystemencoding())

#终端输入输出编码：sys.stdout.encoding，必须与locale编码保持一致，才能print出正确str

import sys

print(sys.stdout.encoding)

Unicode只是在内存中计算时使用，真正显示在屏幕上或者存储在文件中的还是UTF8或者GBK等编码。

## Tkinter

打开文件对话框

**import** Tkinter

**import** tkFileDialog

**def** FileOpen():

filename = tkFileDialog.askopenfilename(title = 'Python Tkinter',

filetypes=[('Python', '\*.py \*.pyw'), ('All files', '\*')])

**print** filename

改变窗口标题

**from** Tkinter **import** \*

root = Tk()

root.title("Hello, World!")

root.mainloop()

程序窗口居中显示

# 放在其他widget定义的后面

**from** Tkinter **import** \*

**def** center\_window(w=300, h=200):

# get screen width and height

ws = root.winfo\_screenwidth()

hs = root.winfo\_screenheight()

# calculate position x, y

x = (ws/2) - (w/2)

y = (hs/2) - (h/2)

root.geometry('%dx%d+%d+%d' % (w, h, x, y))

root = Tk()

center\_window(500, 300)

root.mainloop()

固定程序窗口大小

# 放在其他widget定义的后面

**from** Tkinter **import** \*

root=Tk()

# 下两行起作用

root.minsize(50,50)

root.maxsize(150,150)

Label(root,text="I'm lly365").pack()

root.mainloop()

## 随机数

**随机整数：**

>>> **import** **random**  
>>> random.randint(0,99)  
21

**随机选取0到100间的偶数：**

>>> **import** **random**  
>>> random.randrange(0, 101, 2)  
42  
  
**随机浮点数：**>>> **import** **random**  
>>> random.random()   
0.85415370477785668  
>>> random.uniform(1, 10)  
5.4221167969800881  
 **随机字符：**>>> **import** **random**  
>>> random.choice('abcdefg&#%^\*f')  
'd'  
  
**多个字符中选取特定数量的字符：**>>> **import** **random**  
random.sample('abcdefghij',3)   
['a', 'd', 'b']  
  
**多个字符中选取特定数量的字符组成新字符串：**>>> **import** **random**  
>>> **import** **string**  
>>> string.join(random.sample(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j'], 3)).replace(" ","")  
'fih'  
  
**随机选取字符串：**>>> **import** **random**  
>>> random.choice(['apple', 'pear', 'peach', 'orange', 'lemon'])  
'lemon'  
  
**洗牌：**>>> **import** **random**  
>>> items = [1, 2, 3, 4, 5, 6]  
>>> random.shuffle(items)  
>>> items  
[3, 2, 5, 6, 4, 1]

## 睡眠

import time

time.sleep(2) # 暂停两秒

## 扩展阅读

http://hao.jobbole.com/?catid=144

https://github.com/jobbole/awesome-python-cn