

Python 3 教程



Python的3.0版本，常被称为Python 3000，或简称Py3k。相对于Python的早期版本，这是一个较大的升级。为了不带入过多的累赘，Python 3.0在设计的时候没有考虑向下兼容。

Python 介绍及安装教程我们在[Python 2.X版本的教程](#)中已有介绍，这里就不再赘述。

你也可以点击[Python2.x与3.x版本区别](#) 来查看两者的不同。

本教程主要针对Python 3.x版本的学习，如果你使用的是Python 2.x版本请移步至[Python 2.X版本的教程](#)。

查看python版本

我们可以使用以下命令来查看我们使用的Python版本：

```
python -V
```

以上命令执行结果如下：

```
Python 3.3.2
```

你也可以进入Python的交互式编程模式，查看版本：

```
Python 3.3.2 (v3.3.2:d047928ae3f6, May 16 2013, 00:03:43) [MSC v.1600 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

第一个Python3.x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 3.0+)

```
#!/usr/bin/python3 print("Hello, World!");
```

[运行实例 »](#)

你可以将以上代码保存在hello.py文件中并使用python命令执行该脚本文件。

```
$ python3 hello.py
```

以上命令输出结果为：

```
Hello, World!
```

相关内容：

[Python 3.6.3 中文手册](#)

Python3 基础语法

编码

默认情况下，Python 3 源码文件以 **UTF-8** 编码，所有字符串都是 **unicode** 字符串。当然你也可以为源码文件指定不同的编码：

```
# -*- coding: cp-1252 -*-
```

标识符

- 第一个字符必须是字母表中字母或下划线 `_`。
- 标识符的其他部分由字母、数字和下划线组成。
- 标识符对大小写敏感。

在 Python3 中，非 ASCII 标识符也是允许的了。

python保留字

保留字即关键字，我们不能把它们用作任何标识符名称。Python 的标准库提供了一个 `keyword` 模块，可以输出当前版本的所有关键字：

```
>>> import keyword
>>> keyword.kwlist
['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'impor
```

注释

Python 中单行注释以 `#` 开头，实例如下：

```
#!/usr/bin/python3

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

执行以上代码，输出结果为：

```
Hello, Python!
```

多行注释可以用多个 `#` 号，还有 `'''` 和 `"""`：

```
#!/usr/bin/python3

# 第一个注释
# 第二个注释

'''
第三注释
第四注释
'''

"""
第五注释
第六注释
"""
print ("Hello, Python!")
```

执行以上代码，输出结果为：

```
Hello, Python!
```

行与缩进

python 最具特色的就是使用缩进来表示代码块，不需要使用大括号 `{}`。

缩进的空格数是可变的，但是同一个代码块的语句必须包含相同的缩进空格数。实例如下：

```
if True:
    print ("True")
else:
    print ("False")
```

以下代码最后一行语句缩进数的空格数不一致，会导致运行错误：

```
if True:
    print ("Answer")
    print ("True")
else:
    print ("Answer")
    print ("False")    # 缩进不一致，会导致运行错误
```

以上程序由于缩进不一致，执行后会出现类似以下错误：

```
File "test.py", line 6
    print ("False")    # 缩进不一致，会导致运行错误
    ^
```

```
IndentationError: unindent does not match any outer indentation level
```

多行语句

Python 通常是一行写完一条语句，但如果语句很长，我们可以使用反斜杠 `()` 来实现多行语句，例如：

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

在 `[]`, `{}`, 或 `()` 中的多行语句，不需要使用反斜杠 `()`，例如：

```
total = ['item_one', 'item_two', 'item_three',
```

```
'item_four', 'item_five']
```

数字(Number)类型

python中数字有四种类型：整数、布尔型、浮点数和复数。

- **int** (整数), 如 1, 只有一种整数类型 int，表示为长整型，没有 python2 中的 Long。
- **bool** (布尔), 如 true。
- **float** (浮点数), 如 1.23、3E-2
- **complex** (复数), 如 1+2j、1.1+2.2j

字符串(String)

- python中单引号和双引号使用完全相同。
- 使用三引号(“或”)可以指定一个多行字符串。
- 转义符 \
- 反斜杠可以用来转义，使用r可以让反斜杠不发生转义。。如 r'this is a line with \n' 则\n会显示，并不是换行。
- 按字面意义级联字符串，如'this "is "string"会被自动转换为this is string。
- 字符串可以用 + 运算符连接在一起，用 * 运算符重复。
- Python 中的字符串有两种索引方式，从左往右以 0 开始，从右往左以 -1 开始。
- Python中的字符串不能改变。
- Python 没有单独的字符类型，一个字符就是长度为 1 的字符串。
- 字符串的截取的语法格式如下：变量[头下标:尾下标]

```
word = '字符串'
sentence = "这是一个句子。"
paragraph = """这是一个段落，
可以由多行组成"""
```

实例

```
#!/usr/bin/python3

str='Runoob'

print(str)                # 输出字符串
print(str[0:-1])          # 输出第一个到倒数第二个的所有字符
print(str[0])              # 输出字符串第一个字符
print(str[2:5])            # 输出从第三个开始到第五个的字符
print(str[2:])             # 输出从第三个开始的后的所有字符
print(str * 2)             # 输出字符串两次
print(str + '你好')        # 连接字符串

print('-----')

print('hello\nrunoob')     # 使用反斜杠 (\)+n转义特殊字符
print(r'hello\nrunoob')   # 在字符串前面添加一个 r，表示原始字符串，不会发生转义
```

输出结果为：

```
Runoob
Runoo
R
noo
noob
RunoobRunoob
Runoob你好
-----
hello
runoob
hello\nrunoob
```

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

执行下面的程序在按回车键后就会等待用户输入：

```
#!/usr/bin/python3

input("\n\n按下 enter 键后退出。")
```

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 键时，程序将退出。

同一行显示多条语句

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

```
#!/usr/bin/python3

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

执行以上代码，输出结果为：

```
runoob
7
```

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
```

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 **end=""**：

```
#!/usr/bin/python3

x="a"
y="b"
# 换行输出
print( x )
print( y )

print('-----')
# 不换行输出
print( x, end=" " )
print( y, end=" " )
print()
```

以上实例执行结果为：

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

import 与 from...import

在 python 用 import 或者 from...import 来导入相应的模块。

将整个模块(somemodule)导入，格式为：import somemodule

从某个模块中导入某个函数,格式为：from somemodule import somefunction

从某个模块中导入多个函数,格式为：from somemodule import firstfunc, secondfunc, thirdfunc

将某个模块中的全部函数导入，格式为：from somemodule import *

导入 sys 模块

```
import sys print('-----Python import mode-----'); print('命令行参数为:') for i in sys.argv: print(i) print("\n python 路径为",sys.path)
```

导入 sys 模块的 argv,path 成员

```
from sys import argv,path # 导入特定的成员 print('-----python from import-----') print(path',path) # 因为已经导入path成员，所以此处引用时不需要加sys.path
```

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息，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 [Python3 命令行参数](#)。

Python3 基本数据类型

Python 中的变量不需要声明。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。

在 Python 中，变量就是变量，它没有类型，我们所说的"类型"是变量所指的内存中对象的类型。

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 3.0+)

```
#!/usr/bin/python3 counter = 100 # 整型变量 miles = 1000.0 # 浮点型变量 name = "runoob" # 字符串 print (counter) print (miles) print (name)
```

[运行实例 >>](#)

执行以上程序会输出如下结果：

```
100
1000.0
runoob
```

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, "runoob"
```

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 "runoob" 分配给变量 c。

标准数据类型

Python3 中有六个标准的数据类型：

- Number（数字）
- String（字符串）
- List（列表）
- Tuple（元组）
- Sets（集合）
- Dictionary（字典）

Number（数字）

Python3 支持 **int**、**float**、**bool**、**complex**（复数）。

在 Python 3 里，只有一种整数类型 **int**，表示为长整型，没有 python2 中的 **Long**。

像大多数语言一样，数值类型的赋值和计算都是很直观的。

内置的 **type()** 函数可以用来查询变量所指的对象类型。

```
>>> a, b, c, d = 20, 5.5, True, 4+3j
>>> print(type(a), type(b), type(c), type(d))
<class 'int'> <class 'float'> <class 'bool'> <class 'complex'>
```

此外还可以用 `isinstance` 来判断：

实例

```
>>>a = 111 >>> isinstance(a, int) True >>>
```

`isinstance` 和 `type` 的区别在于：

```
class A:
    pass

class B(A):
    pass

isinstance(A(), A)    # returns True
type(A()) == A        # returns True
isinstance(B(), A)    # returns True
type(B()) == A        # returns False
```

区别就是：

- `type()`不会认为子类是一种父类类型。
- `isinstance()`会认为子类是一种父类类型。

注意：在 Python2 中是没有布尔型的，它用数字 0 表示 False，用 1 表示 True。到 Python3 中，把 True 和 False 定义成关键字了，但它们的值还是 1 和 0，它们可以和数字相加。

当你指定一个值时，Number 对象就会被创建：

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

您也可以使用 `del` 语句删除一些对象引用。

`del` 语句的语法是：

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

您可以通过使用 `del` 语句删除单个或多个对象。例如：

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

数值运算

实例

```
>>>5 + 4 # 加法 9 >>> 4.3 - 2 # 减法 2.3 >>> 3 * 7 # 乘法 21 >>> 2 / 4 # 除法，得到一个浮点数 0.5 >>> 2 // 4 # 除法，得到一个整数 0 >>> 17 % 3 # 取余 2 >>> 2 ** 5 # 乘方 32
```

注意：

- 1、Python 可以同时为多个变量赋值，如 `a, b = 1, 2`。
- 2、一个变量可以通过赋值指向不同类型的对象。
- 3、数值的除法 (`/`) 总是返回一个浮点数，要获取整数使用 `//` 操作符。
- 4、在混合计算时，Python 会把整型转换成为浮点数。

数值类型实例

int	float	complex
10	0.0	3.14j
100	15.20	45.j
-786	-21.9	9.322e-36j
080	32.3e+18	.876j
-0490	-90.	-.6545+0J
-0x260	-32.54e100	3e+26J
0x69	70.2E-12	4.53e-7j

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

String（字符串）

Python中的字符串用单引号(')或双引号(")括起来，同时使用反斜杠(\)转义特殊字符。

字符串的截取的语法格式如下：

变量[头下标:尾下标]

索引值以 0 为开始值，-1 为从末尾的开始位置。

加号(+) 是字符串的连接符，星号(*) 表示复制当前字符串，紧跟的数字为复制的次数。实例如下：

实例

```
#!/usr/bin/python3 str = 'Runoob' print (str) # 输出字符串 print (str[0:-1]) # 输出第一个到倒数第二个的所有字符 print (str[0]) # 输出字符串第一个字符 print (str[2:5]) # 输出从第三个开始到第五个的字符 print (str[2:]) # 输出从第三个开始的后的所有字符 print (str * 2) # 输出字符串两次 print (str + "TEST") # 连接字符串
```

执行以上程序会输出如下结果：

```
Runoob
Runoo
R
noo
noob
RunoobRunoob
RunoobTEST
```

Python使用反斜杠(\)转义特殊字符，如果你不想让反斜杠发生转义，可以在字符串前面添加一个r，表示原始字符串：

```
>>> print ('Ru\noob')
Ru
oob
>>> print (r'Ru\noob')
Ru\noob
>>>
```

另外，反斜杠(\)可以作为续行符，表示下一行是上一行的延续。也可以使用 """...""" 或者 "..."" 跨越多行。

注意，Python 没有单独的字符类型，一个字符就是长度为1的字符串。

实例

```
>>>word = 'Python' >>> print(word[0], word[5]) P n >>> print(word[-1], word[-6]) n P
```

与 C 字符串不同的是，Python 字符串不能被改变。向一个索引位置赋值，比如word[0] = 'm'会导致错误。

注意：

- 1、反斜杠可以用来转义，使用r可以让反斜杠不发生转义。
- 2、字符串可以用+运算符连接在一起，用*运算符重复。
- 3、Python中的字符串有两种索引方式，从左往右以0开始，从右往左以-1开始。
- 4、Python中的字符串不能改变。

List（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。列表中元素的类型可以不相同，它支持数字，字符串甚至可以包含列表（所谓嵌套）。

列表是写在方括号[]之间、用逗号分隔开的元素列表。

和字符串一样，列表同样可以被索引和截取，列表被截取后返回一个包含所需元素的新列表。

列表截取的语法格式如下：

变量[头下标:尾下标]

索引值以0为开始值，-1为从末尾的开始位置。

加号(+)是列表连接运算符，星号(*)是重复操作。如下实例：

实例

```
#!/usr/bin/python3 list = ['abcd', 786, 2.23, 'runoob', 70.2] tinylist = [123, 'runoob'] print(list) # 输出完整列表 print(list[0]) # 输出列表第一个元素 print(list[1:3]) # 从第二个开始输出到第三个元素 print(list[2:]) # 输出从第三个元素开始的所有元素 print(tinylist * 2) # 输出两次列表 print(list + tinylist) # 连接列表
```

以上实例输出结果：

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

与Python字符串不一样的是，列表中的元素是可以改变的：

实例

```
>>>a=[1,2,3,4,5,6]>>>a[0]=9>>>a[2:5]=[13,14,15]>>>a[9,2,13,14,15,6]>>>a[2:5]=[] # 将对应的元素值设置为 []>>>a[9,2,6]
```

List内置了有很多方法，例如append()、pop()等等，这在后面会讲到。

注意：

- 1、List写在方括号之间，元素用逗号隔开。
- 2、和字符串一样，list可以被索引和切片。
- 3、List可以使用+操作符进行拼接。
- 4、List中的元素是可以改变的。

Tuple（元组）

元组 (tuple) 与列表类似, 不同之处在于元组的元素不能修改。元组写在小括号()里, 元素之间用逗号隔开。

元组中的元素类型也可以不相同:

实例

```
#!/usr/bin/python3 tuple = ('abcd', 786, 2.23, 'runoob', 70.2) tinytuple = (123, 'runoob') 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]) # 输出从第二个元素开始到第三个元素 print (tuple[2:]) # 输出从第三个元素开始的所有元素 print (tinytuple * 2) # 输出两次元组 print (tuple + tinytuple) # 连接元组
```

以上实例输出结果:

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

元组与字符串类似, 可以被索引且下标索引从0开始, -1 为从末尾开始的位置。也可以进行截取 (看上面, 这里不再赘述)。

其实, 可以把字符串看作一种特殊的元组。

实例

```
>>>tup = (1, 2, 3, 4, 5, 6)>>> print(tup[0]) 1>>> print(tup[1:5]) (2, 3, 4, 5)>>> tup[0] = 11 # 修改元组元素的操作是非法的
Traceback (most recent call last): File "<stdin>", line 1, in <module> TypeError: 'tuple' object does not support item assignment >>>
```

虽然tuple的元素不可改变, 但它可以包含可变的对象, 比如list列表。

构造包含 0 个或 1 个元素的元组比较特殊, 所以有一些额外的语法规则:

```
tup1 = () # 空元组
tup2 = (20,) # 一个元素, 需要在元素后添加逗号
```

string、list和tuple都属于sequence (序列)。

注意:

- 1、与字符串一样, 元组的元素不能修改。
- 2、元组也可以被索引和切片, 方法一样。
- 3、注意构造包含0或1个元素的元组的特殊语法规则。
- 4、元组也可以使用+操作符进行拼接。

Set (集合)

集合 (set) 是一个无序不重复元素的序列。

基本功能是进行成员关系测试和删除重复元素。

可以使用大括号 {} 或者 set() 函数创建集合, 注意: 创建一个空集合必须用 set() 而不是 {}, 因为 {} 是用来创建一个空字典。

创建格式:

```
parame = {value01,value02,...}
或者
set (value)
```

实例

```
#!/usr/bin/python3 student = {'Tom', 'Jim', 'Mary', 'Tom', 'Jack', 'Rose'} print(student) # 输出集合，重复的元素被自动去掉 # 成员测试
if 'Rose' in student: print('Rose 在集合中') else: print('Rose 不在集合中') # set可以进行集合运算 a = set('abracadabra') b = set('alacazam')
print(a) print(a - b) # a和b的差集 print(a | b) # a和b的并集 print(a & b) # a和b的交集 print(a ^ b) # a和b中不同时存在的元素
```

以上实例输出结果：

```
{'Mary', 'Jim', 'Rose', 'Jack', 'Tom'}
Rose 在集合中
{'b', 'a', 'c', 'r', 'd'}
{'b', 'd', 'r'}
{'l', 'r', 'a', 'c', 'z', 'm', 'b', 'd'}
{'a', 'c'}
{'l', 'r', 'z', 'm', 'b', 'd'}
```

Dictionary（字典）

字典（dictionary）是Python中另一个非常有用的内置数据类型。

列表是有序的对象集合，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。

字典是一种映射类型，字典用“{ }”标识，它是一个无序的**键(key): 值(value)**对集合。

键(key)必须使用不可变类型。

在同一个字典中，键(key)必须是唯一的。

实例

```
#!/usr/bin/python3 dict = {} dict['one'] = "1 - 菜鸟教程" dict[2] = "2 - 菜鸟工具" tinydict = {'name': 'runoob', 'code': 1, 'site': 'www.runoob.com'}
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
```

以上实例输出结果：

```
1 - 菜鸟教程
2 - 菜鸟工具
{'name': 'runoob', 'site': 'www.runoob.com', 'code': 1}
dict_keys(['name', 'site', 'code'])
dict_values(['runoob', 'www.runoob.com', 1])
```

构造函数 dict() 可以直接从键值对序列中构建字典如下：

实例

```
>>>dict([('Runoob', 1), ('Google', 2), ('Taobao', 3)]) {'Taobao': 3, 'Runoob': 1, 'Google': 2} >>> {x: x**2 for x in (2, 4, 6)} {2: 4, 4: 16, 6: 36} >>> dict(Runoob=1, Google=2, Taobao=3) {'Taobao': 3, 'Runoob': 1, 'Google': 2}
```

另外，字典类型也有一些内置的函数，例如clear()、keys()、values()等。

注意：

- 1、字典是一种映射类型，它的元素是键值对。
 - 2、字典的关键字必须为不可变类型，且不能重复。
 - 3、创建空字典使用 {}。
-

Python数据类型转换

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

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

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

Python3 解释器

Linux/Unix的系统上，一般默认的 python 版本为 2.x，我们可以将 python3.x 安装在 **/usr/local/python3** 目录中。

安装完成后，我们可以将路径 **/usr/local/python3/bin** 添加到您的 Linux/Unix 操作系统的环境变量中，这样您就可以通过 shell 终端输入下面的命令来启动 Python3。

```
$ PATH=$PATH:/usr/local/python3/bin/python3    # 设置环境变量
$ python3 --version
Python 3.4.0
```

在Window系统下你可以通过以下命令来设置Python的环境变量，假设你的Python安装在 C:\Python34 下：

```
set path=%path%;C:\python34
```

交互式编程

我们可以在命令提示符中输入"Python"命令来启动Python解释器：

```
$ python3
```

执行以上命令后，出现如下窗口信息：

```
$ python3
Python 3.4.0 (default, Apr 11 2014, 13:05:11)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

在 python 提示符中输入以下语句，然后按回车键查看运行效果：

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

以上命令执行结果如下：

```
Hello, Python!
```

当键入一个多行结构时，续行是必须的。我们可以看下如下 if 语句：

```
>>> flag = True
>>> if flag :
...     print("flag 条件为 True!")
...
flag 条件为 True!
```

脚本式编程

将如下代码拷贝至 **hello.py**文件中：

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

通过以下命令执行该脚本：

```
python3 hello.py
```

输出结果为：

```
Hello, Python!
```

在Linux/Unix系统中，你可以在脚本顶部添加以下命令让Python脚本可以像SHELL脚本一样可直接执行：

```
#!/usr/bin/env python3
```

然后修改脚本权限，使其有执行权限，命令如下：

```
$ chmod +x hello.py
```

执行以下命令：

```
./hello.py
```

输出结果为：

```
Hello, Python!
```

Python3 注释

确保对模块, 函数, 方法和行内注释使用正确的风格

Python中的注释有单行注释和多行注释:

Python中单行注释以 # 开头, 例如: :

```
# 这是一个注释  
print("Hello, World!")
```

多行注释用三个单引号 ''' 或者三个双引号 """ 将注释括起来, 例如:

1、单引号 (''')

```
#!/usr/bin/python3  
'''  
这是多行注释, 用三个单引号  
这是多行注释, 用三个单引号  
这是多行注释, 用三个单引号  
'''  
print("Hello, World!")
```

2、双引号 (""")

```
#!/usr/bin/python3  
"""  
这是多行注释, 用三个双引号  
这是多行注释, 用三个双引号  
这是多行注释, 用三个双引号  
"""  
print("Hello, World!")
```

Python3 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为10，变量b为21：

运算符	描述	实例
+	加 - 两个对象相加	a + b 输出结果 31
-	减 - 得到负数或是一个数减去另一个数	a - b 输出结果 -11
*	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 210
/	除 - x 除以 y	b / a 输出结果 2.1
%	取模 - 返回除法的余数	b % a 输出结果 1
**	幂 - 返回x的y次幂	a**b 为10的21次方
//	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：

实例(Python 3.0+)

```
#!/usr/bin/python3 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： 2.1
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。
>	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：

实例(Python 3.0+)

```
#!/usr/bin/python3
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 小于 b")
else: print ("3 - a 大于等于 b")
if (a > b): print ("4 - a 大于 b")
else: print ("4 - a 小于等于 b")
# 修改变量 a 和 b 的值
a = 5
b = 20
if (a <= b): print ("5 - a 小于等于 b")
else: print ("5 - a 大于 b")
if (b >= a): print ("6 - b 大于等于 a")
else: print ("6 - 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 大于等于 b
4 - a 大于 b
5 - a 小于等于 b
6 - 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：

实例(Python 3.0+)

```
#!/usr/bin/python3
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: 52.0
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：对数据的每个二进位取反,即把1变为0,把0变为1。~x 类似于 -x-1	(~a) 输出结果 -61 ，二进制解释： 1100 0011， 在一个有符号二进制的补码形式。
<<	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：

实例(Python 3.0+)

```
#!/usr/bin/python3 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 True, 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:

实例(Python 3.0+)

```
#!/usr/bin/python3 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:

实例(Python 3.0+)

```
#!/usr/bin/python3 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	<code>x is y</code> , 类似 <code>id(x) == id(y)</code> , 如果引用的是同一个对象则返回 True, 否则返回 False
is not	is not 是判断两个标识符是不是引用自不同对象	<code>x is not y</code> , 类似 <code>id(a) != id(b)</code> 。如果引用的不是同一个对象则返回结果 True, 否则返回 False。

注: [id\(\)](#) 函数用于获取对象内存地址。

以下实例演示了Python所有身份运算符的操作:

实例(Python 3.0+)

```
#!/usr/bin/python3 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) == 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
```

is 与 == 区别:

is 用于判断两个变量引用对象是否为同一个, == 用于判断引用变量的值是否相等。

```
>>>a=[1, 2, 3]>>>b=a>>>b is a True>>>b==a True>>>b=a[:]>>>b is a False>>>b==a True
```

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:

实例(Python 3.0+)

```
#!/usr/bin/python3 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: 90.0
((a + b) * c) / d 运算结果为: 90.0
(a + b) * (c / d) 运算结果为: 90.0
a + (b * c) / d 运算结果为: 50.0
```

Python3 数字(Number)

Python 数字数据类型用于存储数值。

数据类型是不允许改变的,这就意味着如果改变数字数据类型的值,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语句删除一些数字对象的引用。

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 支持三种不同的数值类型:

- **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。Python3 整型是没有限制大小的,可以当作 Long 类型使用,所以 Python3 没有 Python2 的 Long 类型。
- **浮点型(float)** - 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 ($2.5e2 = 2.5 \times 10^2 = 250$)
- **复数((complex))** - 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。

我们可以使用十六进制和八进制来代表整数:

```
>>> number = 0xA0F # 十六进制
>>> number
2575
```

```
>>> number=0o37 # 八进制
>>> number
31
```

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

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

Python 数字类型转换

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。

- **int(x)** 将x转换为一个整数。

- **float(x)** 将x转换到一个浮点数。
- **complex(x)** 将x转换到一个复数，实数部分为 x，虚数部分为 0。
- **complex(x, y)** 将 x 和 y 转换到一个复数，实数部分为 x，虚数部分为 y。x 和 y 是数字表达式。

以下实例将浮点数变量 a 转换为整数：

```
>>> a = 1.0
>>> int(a)
1
```

Python 数字运算

Python 解释器可以作为一个简单的计算器，您可以在解释器里输入一个表达式，它将输出表达式的值。

表达式的语法很直白：+, -, * 和 / 和其它语言（如Pascal或C）里一样。例如：

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5 # 总是返回一个浮点数
1.6
```

注意：在不同的机器上浮点运算的结果可能会不一样。

在整数除法中，除法 (/) 总是返回一个浮点数，如果只想得到整数的结果，丢弃可能的分数部分，可以使用运算符 //

```
>>> 17 / 3 # 整数除法返回浮点型
5.666666666666667
>>>
>>> 17 // 3 # 整数除法返回向下取整后的结果
5
>>> 17 % 3 # %操作符返回除法的余数
2
>>> 5 * 3 + 2
17
```

等号(=) 用于给变量赋值。赋值之后，除了下一个提示符，解释器不会显示任何结果。

```
>>> width = 20
>>> height = 5*9
>>> width * height
900
```

Python 可以使用 ** 操作来进行幂运算：

```
>>> 5 ** 2 # 5 的平方
25
>>> 2 ** 7 # 2的7次方
128
```

变量在使用前必须先"定义"(即赋予变量一个值)，否则会出现错误：

```
>>> n # 尝试访问一个未定义的变量
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

不同类型的数混合运算时会将整数转换为浮点数：

```
>>> 3 * 3.75 / 1.5
```

```
7.5
>>> 7.0 / 2
3.5
```

在交互模式中，最后被输出的表达式结果被赋值给变量 `_`。例如：

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
>>> round(_, 2)
113.06
```

此处，`_` 变量应被用户视为只读变量。

数学函数

函数	返回值 (描述)
abs(x)	返回数字的绝对值，如 <code>abs(-10)</code> 返回 10
ceil(x)	返回数字的上入整数，如 <code>math.ceil(4.1)</code> 返回 5
<code>cmp(x, y)</code>	如果 $x < y$ 返回 -1, 如果 $x = y$ 返回 0, 如果 $x > y$ 返回 1。 Python 3 已废弃 。使用 <code>(x > y) - (x < y)</code> 替换。
exp(x)	返回e的x次幂(e^x),如 <code>math.exp(1)</code> 返回2.718281828459045
fabs(x)	返回数字的绝对值，如 <code>math.fabs(-10)</code> 返回10.0
floor(x)	返回数字的下舍整数，如 <code>math.floor(4.9)</code> 返回 4
log(x)	如 <code>math.log(math.e)</code> 返回 1.0, <code>math.log(100,10)</code> 返回2.0
log10(x)	返回以10为基数的x的对数，如 <code>math.log10(100)</code> 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。

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 <code>random.choice(range(10))</code> ，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。

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 $\sqrt{x^2 + y^2}$ 。
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

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）。

Python3 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 = "Runoob"
```

Python 访问字符串中的值

Python 不支持单字符类型，单字符在 Python 中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：

实例(Python 3.0+)

```
#!/usr/bin/python3 var1 = 'Hello World!' var2 = "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]: unoo
```

Python字符串更新

你可以截取字符串的一部分并与其他字段拼接，如下实例：

实例(Python 3.0+)

```
#!/usr/bin/python3 var1 = 'Hello World!' print ("已更新字符串 :", var1[:6] + 'Runoob!')
```

以上实例执行结果

```
已更新字符串 : 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

\转义字符	换页	描述
\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 输出结果: HelloPython
*	重复输出字符串	a*2 输出结果: HelloHello
[]	通过索引获取字符串中字符	a[1] 输出结果 e
[:]	截取字符串中的一部分	a[1:4] 输出结果 ell
in	成员运算符 - 如果字符串中包含给定的字符返回 True	'H' in a 输出结果 1
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 输出结果 1
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') print(R'\n')
%	格式字符串	请看下一节内容。

实例(Python 3.0+)

```
#!/usr/bin/python3 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' )
```

以上实例输出结果为:

```
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。

实例(Python 3.0+)

```
#!/usr/bin/python3 print ("我叫 %s 今年 %d 岁!" % ('小明', 10))
```

以上实例输出结果:

我叫 小明 今年 10 岁!

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

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

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。实例如下

实例(Python 3.0+)

```
#!/usr/bin/python3 para_str = '''这是一个多行字符串的实例 多行字符串可以使用制表符 TAB (\t)。也可以使用换行符 [\n]。''' print (para_str)
```

以上实例执行结果为:

```
这是一个多行字符串的实例
多行字符串可以使用制表符
TAB (    )。
也可以使用换行符 [
]。
```

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

在Python2中，普通字符串是以8位ASCII码进行存储的，而Unicode字符串则存储为16位unicode字符串，这样能够表示更多的字符集。使用的语法是在字符串前面加上前缀 **u**。

在Python3中，所有的字符串都是Unicode字符串。

Python 的字符串内建函数

Python 的字符串常用内建函数如下：

序号	方法及描述
1	<code>capitalize()</code> 将字符串的第一个字符转换为大写
2	<code>center(width, fillchar)</code> 返回一个指定的宽度 <code>width</code> 居中的字符串， <code>fillchar</code> 为填充的字符，默认为空格。
3	<code>count(str, beg= 0,end=len(string))</code> 返回 <code>str</code> 在 <code>string</code> 里面出现的次数，如果 <code>beg</code> 或者 <code>end</code> 指定则返回指定范围内 <code>str</code> 出现的次数
4	<code>bytes.decode(encoding="utf-8", errors="strict")</code> Python3 中没有 <code>decode</code> 方法，但我们可以使用 <code>bytes</code> 对象的 <code>decode()</code> 方法来解码给定的 <code>bytes</code> 对象，这个 <code>bytes</code> 对象可以由 <code>str.encode()</code> 来编码返回。
5	<code>encode(encoding='UTF-8',errors='strict')</code> 以 <code>encoding</code> 指定的编码格式编码字符串，如果出错默认报一个 <code>ValueError</code> 的异常，除非 <code>errors</code> 指定的是 <code>'ignore'</code> 或者 <code>'replace'</code>
6	<code>endswith(suffix, beg=0, end=len(string))</code> 检查字符串是否以 <code>obj</code> 结束，如果 <code>beg</code> 或者 <code>end</code> 指定则检查指定的范围内是否以 <code>obj</code> 结束，如果是，返回 <code>True</code> ，否则返回 <code>False</code> 。
7	<code>expandtabs(tabsize=8)</code> 把字符串 <code>string</code> 中的 <code>tab</code> 符号转为空格， <code>tab</code> 符号默认的空格数是 8。
	<code>find(str, beg=0 end=len(string))</code>

8

检测 str 是否包含在字符串中，如果指定范围 beg 和 end，则检查是否包含在指定范围内，如果包含返回开始的索引值，否则返回-1

[index\(str, beg=0, end=len\(string\)\)](#)

9

跟find()方法一样，只不过如果str不在字符串中会报一个异常。

[isalnum\(\)](#)

10

如果字符串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

[isalpha\(\)](#)

11

如果字符串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

[isdigit\(\)](#)

12

如果字符串只包含数字则返回 True 否则返回 False..

[islower\(\)](#)

13

如果字符串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 True，否则返回 False

[isnumeric\(\)](#)

14

如果字符串中只包含数字字符，则返回 True，否则返回 False

[isspace\(\)](#)

15

如果字符串中只包含空白，则返回 True，否则返回 False.

[istitle\(\)](#)

16

如果字符串是标题化的(见 title())则返回 True，否则返回 False

[isupper\(\)](#)

17

如果字符串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

[join\(seq\)](#)

18

以指定字符串作为分隔符，将 seq 中所有的元素(的字符串表示)合并为一个新的字符串

[len\(string\)](#)

19

返回字符串长度

[ljust\(width\[, fillchar\]\)](#)

20

返回一个原字符串左对齐,并使用 fillchar 填充至长度 width 的新字符串, fillchar 默认为空格。

[lower\(\)](#)

21

转换字符串中所有大写字符为小写。

[lstrip\(\)](#)

22

截掉字符串左边的空格或指定字符。

[maketrans\(\)](#)

23

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\(str\)](#)

24

返回字符串 str 中最大的字母。

[min\(str\)](#)

25

返回字符串 str 中最小的字母。

[replace\(old, new \[, max\]\)](#)

26

把 将字符串中的 str1 替换成 str2,如果 max 指定,则替换不超过 max 次。

[rfind\(str, beg=0,end=len\(string\)\)](#)

27

类似于 find()函数,不过是从右边开始查找。

[rindex\(str, beg=0, end=len\(string\)\)](#)

28

类似于 index(),不过是从右边开始。

[rjust\(width\[, fillchar\]\)](#)

29

返回一个原字符串右对齐,并使用fillchar(默认空格)填充至长度 width 的新字符串

[rstrip\(\)](#)

30

删除字符串字符串末尾的空格。

[split\(str=' ', num=string.count\(str\)\)](#)

31

`num=string.count(str)` 以 `str` 为分隔符截取字符串, 如果 `num` 有指定值, 则仅截取 `num` 个子字符串

[`splitlines\(\[keepends\]\)`](#)

32

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。

[`startswith\(str, beg=0, end=len\(string\)\)`](#)

33

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。

[`strip\(\[chars\]\)`](#)

34

在字符串上执行 `lstrip()`和 `rstrip()`

[`swapcase\(\)`](#)

35

将字符串中大写转换为小写, 小写转换为大写

[`title\(\)`](#)

36

返回"标题化"的字符串,就是说所有单词都是以大写开始, 其余字母均为小写(见 `istitle()`)

[`translate\(table, deletechars=""\)`](#)

37

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 `deletechars` 参数中

[`upper\(\)`](#)

38

转换字符串中的小写字母为大写

[`zfill\(width\)`](#)

39

返回长度为 `width` 的字符串, 原字符串右对齐, 前面填充0

[`isdecimal\(\)`](#)

40

检查字符串是否只包含十进制字符, 如果是返回 `true`, 否则返回 `false`。

Python3 列表

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 = ['Google', 'Runoob', 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：

实例(Python 3.0+)

```
#!/usr/bin/python3 list1 = ['Google', 'Runoob', 1997, 2000]; list2 = [1, 2, 3, 4, 5, 6, 7 ]; print ("list1[0]: ", list1[0]) print ("list2[1:5]: ", list2[1:5])
```

[运行实例 »](#)

以上实例输出结果：

```
list1[0]: Google
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：

实例(Python 3.0+)

```
#!/usr/bin/python3 list = ['Google', 'Runoob', 1997, 2000] print ("第三个元素为 :", list[2]) list[2] = 2001 print ("更新后的第三个元素为 :", 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：

```
第三个元素为 : 1997
更新后的第三个元素为 : 2001
```

删除列表元素

可以使用 del 语句来删除列表的元素，如下实例：

实例(Python 3.0+)

```
#!/usr/bin/python3 list = ['Google', 'Runoob', 1997, 2000] print (list) del list[2] print ("删除第三个元素 :", list)
```

以上实例输出结果：

删除第三个元素 :['Google', 'Runoob', 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, end=" ")	1 2 3	迭代

Python列表截取与拼接

Python的列表截取与字符串操作类型，如下所示：

```
L=['Google', 'Runoob', 'Taobao']
```

操作：

Python 表达式	结果	描述
L[2]	'Taobao'	读取第三个元素
L[-2]	'Runoob'	从右侧开始读取倒数第二个元素: count from the right
L[1:]	['Runoob', 'Taobao']	输出从第二个元素开始后的所有元素

```
>>>L=['Google', 'Runoob', 'Taobao']>>> L[2] 'Taobao'>>> L[-2] 'Runoob'>>> L[1:] ['Runoob', 'Taobao']>>>
```

列表还支持拼接操作：

```
>>>squares = [1, 4, 9, 16, 25]>>> squares + [36, 49, 64, 81, 100] [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

嵌套列表

使用嵌套列表即在列表里创建其它列表，例如：

```
>>>a = ['a', 'b', 'c']>>> n = [1, 2, 3]>>> x = [a, n]>>> x[['a', 'b', 'c'], [1, 2, 3]]>>> x[0] ['a', 'b', 'c']>>> x[0][1] 'b'
```

Python列表函数&方法

Python包含以下函数：

序号	函数
1	<u>len(list)</u> 列表元素个数
2	<u>max(list)</u> 返回列表元素最大值
3	<u>min(list)</u> 返回列表元素最小值
4	<u>list(seq)</u> 将元组转换为列表

Python包含以下方法:

序号	方法
1	<u>list.append(obj)</u> 在列表末尾添加新的对象
2	<u>list.count(obj)</u> 统计某个元素在列表中出现的次数
3	<u>list.extend(seq)</u> 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	<u>list.index(obj)</u> 从列表中找出某个值第一个匹配项的索引位置
5	<u>list.insert(index, obj)</u> 将对象插入列表
6	<u>list.pop(obj=list[-1])</u> 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	<u>list.remove(obj)</u> 移除列表中某个值的第一个匹配项
8	<u>list.reverse()</u> 反向列表中元素
9	<u>list.sort(func)</u> 对原列表进行排序
10	<u>list.clear()</u> 清空列表
11	<u>list.copy()</u> 复制列表

Python3 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 = ('Google', 'Runoob', 1997, 2000);
>>> tup2 = (1, 2, 3, 4, 5 );
>>> tup3 = "a", "b", "c", "d";    # 不需要括号也可以
>>> type(tup3)
<class 'tuple'>
```

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，否则括号会被当作运算符使用：

```
>>> tup1 = (50)
>>> type(tup1)      # 不加逗号，类型为整型
<class 'int'>

>>> tup1 = (50,)
>>> type(tup1)      # 加上逗号，类型为元组
<class 'tuple'>
```

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/python3

tup1 = ('Google', 'Runoob', 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]:  Google
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/python3

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/python3

tup = ('Google', 'Runoob', 1997, 2000)

print (tup)
del tup;
print ("删除后的元组 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:

```
删除后的元组 tup :
Traceback (most recent call last):
  File "test.py", line 8, 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 = ('Google', 'Taobao', 'Runoob')
```

Python 表达式	结果	描述
L[2]	'Runoob'	读取第三个元素
L[-2]	'Taobao'	反向读取；读取倒数第二个元素
L[1:]	('Taobao', 'Runoob')	截取元素，从第二个开始后的所有元素。

运行实例如下:

```
>>> L = ('Google', 'Taobao', 'Runoob')
>>> L[2]
```

```
'Runoob'
>>> L[-2]
'Taobao'
>>> L[1:]
('Taobao', 'Runoob')
```

元组内置函数

Python元组包含了以下内置函数

序号	方法及描述	实例
1	<code>len(tuple)</code> 计算元组元素个数。	<pre>>>> tuple1 = ('Google', 'Runoob', 'Taobao') >>> len(tuple1) 3 >>></pre>
2	<code>max(tuple)</code> 返回元组中元素最大值。	<pre>>>> tuple2 = ('5', '4', '8') >>> max(tuple2) '8' >>></pre>
3	<code>min(tuple)</code> 返回元组中元素最小值。	<pre>>>> tuple2 = ('5', '4', '8') >>> min(tuple2) '4' >>></pre>
4	<code>tuple(seq)</code> 将列表转换为元组。	<pre>>>> list1= ['Google', 'Taobao', 'Runoob', 'Baidu'] >>> tuple1=tuple(list1) >>> tuple1 ('Google', 'Taobao', 'Runoob', 'Baidu')</pre>

Python3 字典

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/python3  
  
dict = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'}  
  
print ("dict['Name']: ", dict['Name'])  
print ("dict['Age']: ", dict['Age'])
```

以上实例输出结果：

```
dict['Name']: Runoob  
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/python3  
  
dict = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'};  
  
print ("dict['Alice']: ", dict['Alice'])
```

以上实例输出结果：

```
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/python3  
  
dict = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'}
```

```
dict['Age'] = 8;           # 更新 Age
dict['School'] = "菜鸟教程" # 添加信息

print ("dict['Age']: ", dict['Age'])
print ("dict['School']: ", dict['School'])
```

以上实例输出结果：

```
dict['Age']: 8
dict['School']: 菜鸟教程
```

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/python3

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

del dict['Name'] # 删除键 'Name'
dict.clear()    # 清空字典
del dict        # 删除字典

print ("dict['Age']: ", dict['Age'])
print ("dict['School']: ", dict['School'])
```

但这会引发一个异常，因为用执行 del 操作后字典不再存在：

```
Traceback (most recent call last):
  File "test.py", line 9, in <module>
    print ("dict['Age']: ", dict['Age'])
TypeError: 'type' object is not subscriptable
```

注：del() 方法后面也会讨论。

字典键的特性

字典值可以是任何的 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/python3

dict = {'Name': 'Runoob', 'Age': 7, 'Name': '小菜鸟'}

print ("dict['Name']: ", dict['Name'])
```

以上实例输出结果：

```
dict['Name']: 小菜鸟
```

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

```
#!/usr/bin/python3

dict = {[ 'Name' ]: 'Runoob', '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': 'Runoob', 'Age': 7}
TypeError: unhashable type: 'list'
```

字典内置函数&方法

Python字典包含了以下内置函数:

序号	函数及描述	实例
1	<code>len(dict)</code> 计算字典元素个数, 即键的总数。	<pre>>>> dict = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'} >>> len(dict) 3</pre>
2	<code>str(dict)</code> 输出字典, 以可打印的字符串表示。	<pre>>>> dict = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'} >>> str(dict) "{'Name': 'Runoob', 'Class': 'First', 'Age': 7}"</pre>
3	<code>type(variable)</code> 返回输入的变量类型, 如果变量是字典就返回字典类型。	<pre>>>> dict = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'} >>> type(dict) <class 'dict'></pre>

Python字典包含了以下内置方法:

序号	函数及描述
1	radiandict.clear() 删除字典内所有元素
2	radiandict.copy() 返回一个字典的浅复制
3	radiandict.fromkeys() 创建一个新字典, 以序列seq中元素做字典的键, val为字典所有键对应的初始值
4	radiandict.get(key, default=None) 返回指定键的值, 如果值不在字典中返回default值
5	key in dict 如果键在字典dict里返回true, 否则返回false
6	radiandict.items() 以列表返回可遍历的(键, 值)元组数组
7	radiandict.keys() 以列表返回一个字典所有的键
8	radiandict.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	radiandict.update(dict2) 把字典dict2的键/值对更新到dict里
10	radiandict.values() 以列表返回字典中的所有值
11	pop(key[, default]) 删除字典给定键 key 所对应的值, 返回值为被删除的值。key值必须给出。否则, 返回default值。 popitem()

12 随机返回并删除字典中的一对键和值(一般删除末尾对)。

Python3 编程第一步

在前面的教程中我们已经学习了一些 Python3 的基本语法知识，下面我们尝试来写一个斐波纳契数列。

实例如下：

```
#!/usr/bin/python3

# Fibonacci series: 斐波纳契数列
# 两个元素的总和确定了下一个数
a, b = 0, 1
while b < 10:
    print(b)
    a, b = b, a+b
```

执行以上程序，输出结果为：

```
1
1
2
3
5
8
```

这个例子介绍了几个新特征。

第一行包含了一个复合赋值：变量 a 和 b 同时得到新值 0 和 1。最后一行再次使用了同样的方法，可以看到，右边的表达式会在赋值变动之前执行。右边表达式的执行顺序是从左往右的。

输出变量值：

```
>>> i = 256*256
>>> print('i 的值为: ', i)
i 的值为: 65536
```

end 关键字

关键字 end 可以用于将结果输出到同一行，或者在输出的末尾添加不同的字符，实例如下：

```
#!/usr/bin/python3

# Fibonacci series: 斐波纳契数列
# 两个元素的总和确定了下一个数
a, b = 0, 1
while b < 1000:
    print(b, end=', ')
    a, b = b, a+b
```

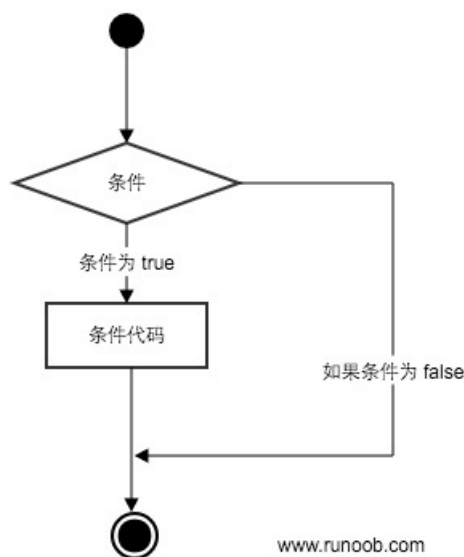
执行以上程序，输出结果为：

```
1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

Python3 条件控制

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：



if 语句

Python中if语句的一般形式如下所示：

```
if condition_1: statement_block_1 elif condition_2: statement_block_2 else: statement_block_3
```

- 如果 "condition_1" 为 True 将执行 "statement_block_1" 块语句
- 如果 "condition_1" 为 False，将判断 "condition_2"
- 如果 "condition_2" 为 True 将执行 "statement_block_2" 块语句
- 如果 "condition_2" 为 False，将执行 "statement_block_3" 块语句

Python 中用 **elif** 代替了 **else if**，所以if语句的关键字为：**if – elif – else**。

注意：

- 1、每个条件后面要使用冒号（:），表示接下来是满足条件后要执行的语句块。
- 2、使用缩进来划分语句块，相同缩进数的语句在一起组成一个语句块。
- 3、在Python中没有switch – case语句。

实例

以下是一个简单的 if 实例：

实例

```
#!/usr/bin/python3 var1 = 100 if var1: print ("1 - if 表达式条件为 true") print (var1) var2 = 0 if var2: print ("2 - if 表达式条件为 true") print (var2) print ("Good bye!")
```

执行以上代码，输出结果为：

```
1 - if 表达式条件为 true
100
Good bye!
```

从结果可以看到由于变量 var2 为 0，所以对应的 if 内的语句没有执行。

以下实例演示了狗的年龄计算判断：

实例

```
#!/usr/bin/python3 age = int(input("请输入你家狗狗的年龄: ")) print("") if age < 0: print("你是在逗我吧!") elif age == 1: print("相当于 14 岁的人。") elif age == 2: print("相当于 22 岁的人。") elif age > 2: human = 22 + (age - 2)*5 print("对应人类年龄: ", human) ### 退出提示 input("点击 enter 键退出")
```

将以上脚本保存在 dog.py 文件中，并执行该脚本：

```
$ python3 dog.py
请输入你家狗狗的年龄: 1
```

```
相当于 14 岁的人。
点击 enter 键退出
```

以下为 Python 中常用的操作运算符：

操作符	描述
<	小于
<=	小于或等于
>	大于
>=	大于或等于
==	等于，比较对象是否相等
!=	不等于

实例

```
#!/usr/bin/python3 # 程序演示了 == 操作符 # 使用数字 print(5 == 6) # 使用变量 x = 5 y = 8 print(x == y)
```

以上实例输出结果：

```
False
False
```

high_low.py 文件演示了数字的比较运算：

实例

```
#!/usr/bin/python3 # 该实例演示了数字猜谜游戏 number = 7 guess = -1 print("数字猜谜游戏!") while guess != number: guess = int(input("请输入你猜的数字: ")) if guess == number: print("恭喜，你猜对了!") elif guess < number: print("猜的数字小了...") elif guess > number: print("猜的数字大了...")
```

执行以上脚本，实例输出结果如下：

```
$ python3 high_low.py
数字猜谜游戏!
请输入你猜的数字: 1
猜的数字小了...
请输入你猜的数字: 9
猜的数字大了...
请输入你猜的数字: 7
```

恭喜，你猜对了！

if 嵌套

在嵌套 if 语句中，可以把 if...elif...else 结构放在另外一个 if...elif...else 结构中。

```
if 表达式1:
    语句
    if 表达式2:
        语句
    elif 表达式3:
        语句
    else:
        语句
elif 表达式4:
    语句
else:
    语句
```

实例

```
#!/usr/bin/python3 num=int(input("输入一个数字： ")) if num%2==0: if num%3==0: print ("你输入的数字可以整除 2 和 3") else:
print ("你输入的数字可以整除 2，但不能整除 3") else: if num%3==0: print ("你输入的数字可以整除 3，但不能整除 2") else:
print ("你输入的数字不能整除 2 和 3")
```

将以上程序保存到 test_if.py 文件中，执行后输出结果为：

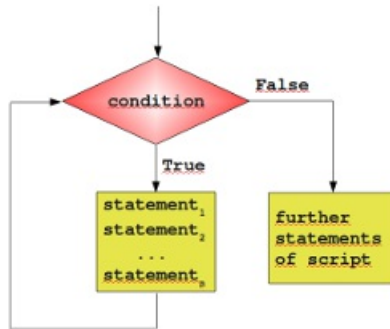
```
$ python3 test.py
输入一个数字： 6
你输入的数字可以整除 2 和 3
```

Python3 循环语句

本章节将为大家介绍Python循环语句的使用。

Python中的循环语句有 for 和 while。

Python循环语句的控制结构图如下所示：



while 循环

Python中while语句的一般形式：

```
while 判断条件:  
    语句
```

同样需要注意冒号和缩进。另外，在Python中没有do..while循环。

以下实例使用了 while 来计算 1 到 100 的总和：

实例

```
#!/usr/bin/env python3 n = 100 sum = 0 counter = 1 while counter <= n: sum = sum + counter counter += 1 print("1 到 %d 之和为:  
%d" % (n, sum))
```

执行结果如下：

```
1 到 100 之和为：5050
```

无限循环

我们可以通过设置条件表达式永远不为 false 来实现无限循环，实例如下：

实例

```
#!/usr/bin/python3 var = 1 while var == 1 : # 表达式永远为 true num = int(input("输入一个数字 :")) print ("你输入的数字是:",  
num) print ("Good bye!")
```

执行以上脚本，输出结果如下：

```
输入一个数字 :5  
你输入的数字是： 5  
输入一个数字 :
```

你可以使用 **CTRL+C** 来退出当前的无限循环。

无限循环在服务器上客户端的实时请求非常有用。

while 循环使用 else 语句

在 while ... else 在条件语句为 false 时执行 else 的语句块:

实例

```
#!/usr/bin/python3 count = 0 while count < 5: print (count, " 小于 5") count = count + 1 else: print (count, " 大于或等于 5")
```

执行以上脚本，输出结果如下：

```
0  小于 5
1  小于 5
2  小于 5
3  小于 5
4  小于 5
5  大于或等于 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：

实例

```
#!/usr/bin/python flag = 1 while (flag): print ('欢迎访问菜鸟教程!') print ("Good bye!")
```

注意： 以上的无限循环你可以使用 CTRL+C 来中断循环。

执行以上脚本，输出结果如下：

```
欢迎访问菜鸟教程！
欢迎访问菜鸟教程！
欢迎访问菜鸟教程！
欢迎访问菜鸟教程！
欢迎访问菜鸟教程！
.....
```

for 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。

for循环的一般格式如下：

```
for <variable> in <sequence>: <statements> else: <statements>
```

Python loop循环实例：

实例

```
>>>languages = ["C", "C++", "Perl", "Python"]>>> for x in languages: ... print (x) ... C C++ Perl Python>>>
```

以下 for 实例中使用了 break 语句，break 语句用于跳出当前循环体：

实例

```
#!/usr/bin/python3 sites = ["Baidu", "Google","Runoob","Taobao"] for site in sites: if site == "Runoob": print("菜鸟教程!") break print("循环数据 " + site) else: print("没有循环数据!") print("完成循环!")
```

执行脚本后，在循环到 "Runoob"时会跳出循环体：

```
循环数据 Baidu  
循环数据 Google  
菜鸟教程！  
完成循环！
```

range()函数

如果你需要遍历数字序列，可以使用内置range()函数。它会生成数列，例如：

实例

```
>>>for i in range(5): ... print(i) ... 0 1 2 3 4
```

你也可以使用range指定区间的值：

实例

```
>>>for i in range(5,9): print(i) 5 6 7 8 >>>
```

也可以使range以指定数字开始并指定不同的增量(甚至可以是负数，有时这也叫做'步长'):

实例

```
>>>for i in range(0, 10, 3): print(i) 0 3 6 9 >>>
```

负数：

实例

```
>>>for i in range(-10, -100, -30): print(i) -10 -40 -70 >>>
```

您可以结合range()和len()函数以遍历一个序列的索引,如下所示:

实例

```
>>>a = ['Google', 'Baidu', 'Runoob', 'Taobao', 'QQ'] >>> for i in range(len(a)): ... print(i, a[i]) ... 0 Google 1 Baidu 2 Runoob 3 Taobao 4 QQ >>>
```

还可以使用range()函数来创建一个列表：

实例

```
>>>list(range(5)) [0, 1, 2, 3, 4] >>>
```

break和continue语句及循环中的else子句

break 语句可以跳出 for 和 while 的循环体。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。实例如下：

实例

```
#!/usr/bin/python3 for letter in 'Runoob': # 第一个实例 if letter == 'b': break print ('当前字母为 .', letter) var = 10 # 第二个实例 while var > 0: print ('当期变量值为 .', var) var = var - 1 if var == 5: break print ("Good bye!")
```


执行以上脚本输出结果为：

```
当前字母为 : R
当前字母为 : u
当前字母为 : n
当前字母为 : o
当前字母为 : o
当期变量值为 : 10
当期变量值为 : 9
当期变量值为 : 8
当期变量值为 : 7
当期变量值为 : 6
Good bye!
```

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。

实例

```
#!/usr/bin/python3 for letter in 'Runoob': # 第一个实例 if letter == 'o': # 字母为 o 时跳过输出 continue print ('当前字母 :', letter) var = 10 # 第二个实例 while var > 0: var = var - 1 if var == 5: # 变量为 5 时跳过输出 continue print ('当前变量值 :', var) print ("Good bye!")
```

执行以上脚本输出结果为：

```
当前字母 : R
当前字母 : u
当前字母 : n
当前字母 : b
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!
```

循环语句可以有 else 子句，它在穷尽列表(以for循环)或条件变为 false (以while循环)导致循环终止时被执行,但循环被 break 终止时不执行。

如下实例用于查询质数的循环例子：

实例

```
#!/usr/bin/python3 for n in range(2, 10): for x in range(2, n): if n % x == 0: print(n, '等于', x, '*', n//x) break else: # 循环中没有找到元素 print(n, '是质数')
```

执行以上脚本输出结果为：

```
2 是质数
3 是质数
4 等于 2 * 2
5 是质数
6 等于 2 * 3
7 是质数
8 等于 2 * 4
9 等于 3 * 3
```

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，如下实例

实例

```
>>>while True: ... pass # 等待键盘中断 (Ctrl+C)
```

最小的类:

实例

```
>>>class MyEmptyClass: ... pass
```

以下实例在字母为 o 时 执行 pass 语句块:

实例

```
#!/usr/bin/python3 for letter in 'Runoob': if letter == 'o': pass print ('执行 pass 块') print ('当前字母 :', letter) print ("Good bye!")
```

执行以上脚本输出结果为:

```
当前字母 : R
当前字母 : u
当前字母 : n
执行 pass 块
当前字母 : o
执行 pass 块
当前字母 : o
当前字母 : b
Good bye!
```

Python3 迭代器与生成器

迭代器

迭代是Python最强大的功能之一，是访问集合元素的一种方式。

迭代器是一个可以记住遍历的位置的对象。

迭代器对象从集合的第一个元素开始访问，直到所有的元素被访问完结束。迭代器只能往前不会后退。

迭代器有两个基本的方法：`iter()` 和 `next()`。

字符串，列表或元组对象都可用于创建迭代器：

实例(Python 3.0+)

```
>>>list=[1,2,3,4]>>> it = iter(list) # 创建迭代器对象>>> print (next(it)) # 输出迭代器的下一个元素 1>>> print (next(it)) 2>>>
```

迭代器对象可以使用常规for语句进行遍历：

实例(Python 3.0+)

```
#!/usr/bin/python3 list=[1,2,3,4] it = iter(list) # 创建迭代器对象 for x in it: print (x, end=" ")
```

执行以上程序，输出结果如下：

```
1 2 3 4
```

也可以使用 `next()` 函数：

实例(Python 3.0+)

```
#!/usr/bin/python3 import sys # 引入 sys 模块 list=[1,2,3,4] it = iter(list) # 创建迭代器对象 while True: try: print (next(it)) except StopIteration: sys.exit()
```

执行以上程序，输出结果如下：

```
1
2
3
4
```

生成器

在 Python 中，使用了 `yield` 的函数被称为生成器（generator）。

跟普通函数不同的是，生成器是一个返回迭代器的函数，只能用于迭代操作，更简单点理解生成器就是一个迭代器。

在调用生成器运行的过程中，每次遇到 `yield` 时函数会暂停并保存当前所有的运行信息，返回 `yield` 的值，并在下一次执行 `next()` 方法时从当前位置继续运行。

调用一个生成器函数，返回的是一个迭代器对象。

以下实例使用 `yield` 实现斐波那契数列：

实例(Python 3.0+)

```
#!/usr/bin/python3 import sys def fibonacci(n): # 生成器函数 - 斐波那契 a, b, counter = 0, 1, 0 while True: if (counter > n): return yield a a, b = b, a + b counter += 1 f = fibonacci(10) # f 是一个迭代器，由生成器返回生成 while True: try: print (next(f), end=" ") except StopIteration: sys.exit()
```

执行以上程序，输出结果如下：

0 1 1 2 3 5 8 13 21 34 55

Python3 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

Python 定义函数使用 def关键字，一般格式如下：

```
def 函数名 (参数列表):  
    函数体
```

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

让我们使用函数来输出"Hello World! "：

```
>>> def hello() :  
    print("Hello World!")
```

```
>>> hello()  
Hello World!  
>>>
```

更复杂点的应用，函数中带上参数变量：

```
#!/usr/bin/python3  
  
# 计算面积函数  
def area(width, height):  
    return width * height  
  
def print_welcome(name):  
    print("Welcome", name)  
  
print_welcome("Runoob")  
w = 4  
h = 5  
print("width =", w, " height =", h, " area =", area(w, h))
```

以上实例输出结果：

```
Welcome Runoob  
width = 4  height = 5  area = 20
```

函数调用

定义一个函数：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 命令提示符执行。

如下实例调用了 `printme()` 函数：

```
#!/usr/bin/python3

# 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），可以是指向 List 类型对象，也可以是指向 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/python3

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

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

```
#!/usr/bin/python3

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

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

传入函数的和在末尾添加新内容的对象用的是同一个引用。故输出结果如下：

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

参数

以下是调用函数时可使用的正式参数类型：

- 必需参数
- 关键字参数
- 默认参数
- 不定长参数

必需参数

必需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/python3

#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 10, in <module>
    printme();
TypeError: printme() missing 1 required positional argument: 'str'
```

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/python3

#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 = "菜鸟教程");
```

以上实例输出结果：

菜鸟教程

以下实例中演示了函数参数的使用不需要使用指定顺序：

```
#!/usr/bin/python3

#可写函数说明
def printinfo( name, age ):
    "打印任何传入的字符串"
    print ("名字: ", name);
    print ("年龄: ", age);
    return;

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

以上实例输出结果：

名字: runoob
年龄: 50

默认参数

调用函数时，如果没有传递参数，则会使用默认参数。以下实例中如果没有传入 age 参数，则使用默认值：

```
#!/usr/bin/python3

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

#调用printinfo函数
printinfo( age=50, name="runoob" );
print ("-----")
printinfo( name="runoob" );
```

以上实例输出结果：

名字: runoob
年龄: 50

名字: runoob
年龄: 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。如果在函数调用时没有指定参数，它就是一个空元组。我们也可以不向函数传递未命名的变量。如下实例：

```
#!/usr/bin/python3

# 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。

所谓匿名，意即不再使用 def 语句这样标准的形式定义一个函数。

- 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，且不能访问自己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/python3

# 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 [表达式] 语句用于退出函数，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，以下实例演示了 return 语句的用法：

```
#!/usr/bin/python3

# 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 );
print ("函数外 :", total)
```

以上实例输出结果:

```
函数内 : 30
函数外 : 30
```

变量作用域

Python 中，程序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可以访问哪个特定的变量名称。Python的作用域一共有4种，分别是：

- L (Local) 局部作用域
- E (Enclosing) 闭包函数外的函数中
- G (Global) 全局作用域
- B (Built-in) 内建作用域

以 L→E→G→B 的规则查找，即：在局部找不到，便会去局部外的局部找（例如闭包），再找不到就会去全局找，再去内建中找。

```
x = int(2.9) # 内建作用域

g_count = 0 # 全局作用域
def outer():
    o_count = 1 # 闭包函数外的函数中
    def inner():
        i_count = 2 # 局部作用域
```

Python 中只有模块（module），类（class）以及函数（def、lambda）才会引入新的作用域，其它的代码块（如 if/elif/else/、try/except、for/while 等）是不会引入新的作用域的，也就是说这些语句内定义的变量，外部也可以访问，如下代码：

```
>>> if True:
...     msg = 'I am from Runoob'
...
>>> msg
'I am from Runoob'
>>>
```

实例中 msg 变量定义在 if 语句块中，但外部还是可以访问的。

如果将 msg 定义在函数中，则它就是局部变量，外部不能访问：

```
>>> def test():
...     msg_inner = 'I am from Runoob'
...
>>> msg_inner
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'msg_inner' is not defined
>>>
```

从报错的信息上看，说明了 msg_inner 未定义，无法使用，因为它是局部变量，只有在函数内可以使用。

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/python3

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
```

global 和 nonlocal关键字

当内部作用域想修改外部作用域的变量时，就要用到global和nonlocal关键字了。

以下实例修改全局变量 num：

```
#!/usr/bin/python3

num = 1
def fun1():
    global num # 需要使用 global 关键字声明
    print(num)
    num = 123
    print(num)
fun1()
```

以上实例输出结果：

```
1
123
```

如果要修改嵌套作用域（enclosing 作用域，外层非全局作用域）中的变量则需要 nonlocal 关键字了，如下实例：

```
#!/usr/bin/python3
```

```
def outer():
    num = 10
    def inner():
        nonlocal num    # nonlocal关键字声明
        num = 100
        print(num)
    inner()
    print(num)
outer()
```

以上实例输出结果:

```
100
100
```

另外有一种特殊情况, 假设下面这段代码被运行:

```
#!/usr/bin/python3
```

```
a = 10
def test():
    a = a + 1
    print(a)
test()
```

以上程序执行, 报错信息如下:

```
Traceback (most recent call last):
  File "test.py", line 7, in <module>
    test()
  File "test.py", line 5, in test
    a = a + 1
UnboundLocalError: local variable 'a' referenced before assignment
```

错误信息为局部作用域引用错误, 因为 test 函数中的 a 使用的是局部, 未定义, 无法修改。

Python3 数据结构

本章节我们主要结合前面所学的知识点来介绍Python数据结构。

列表

Python中列表是可变的，这是它区别于字符串和元组的最重要的特点，一句话概括即：列表可以修改，而字符串和元组不能。

以下是 Python 中列表的方法：

方法	描述
<code>list.append(x)</code>	把一个元素添加到列表的结尾，相当于 <code>a[len(a):] = [x]</code> 。
<code>list.extend(L)</code>	通过添加指定列表的所有元素来扩充列表，相当于 <code>a[len(a):] = L</code> 。
<code>list.insert(i, x)</code>	在指定位置插入一个元素。第一个参数是准备插入到其前面的那个元素的索引，例如 <code>a.insert(0, x)</code> 会插入到整个列表之前，而 <code>a.insert(len(a), x)</code> 相当于 <code>a.append(x)</code> 。
<code>list.remove(x)</code>	删除列表中值为 <code>x</code> 的第一个元素。如果没有这样的元素，就会返回一个错误。
<code>list.pop([i])</code>	从列表的指定位置删除元素，并将其返回。如果没有指定索引， <code>a.pop()</code> 返回最后一个元素。元素随即从列表中被删除。（方法中 <code>i</code> 两边的方括号表示这个参数是可选的，而不是要求你输入一对方括号，你会经常在 Python 库参考手册中遇到这样的标记。）
<code>list.clear()</code>	移除列表中的所有项，等于 <code>del a[:]</code> 。
<code>list.index(x)</code>	返回列表中第一个值为 <code>x</code> 的元素的索引。如果没有匹配的元素就会返回一个错误。
<code>list.count(x)</code>	返回 <code>x</code> 在列表中出现的次数。
<code>list.sort()</code>	对列表中的元素进行排序。
<code>list.reverse()</code>	倒排列表中的元素。
<code>list.copy()</code>	返回列表的浅复制，等于 <code>a[:]</code> 。

下面示例演示了列表的大部分方法：

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print(a.count(333), a.count(66.25), a.count('x'))
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```

注意：类似 `insert`, `remove` 或 `sort` 等修改列表的方法没有返回值。

将列表当做堆栈使用

列表方法使得列表可以很方便的作为一个堆栈来使用，堆栈作为特定的数据结构，最先进入的元素最后一个被释放（后进先出）。用 `append()` 方法可以把一个元素添加到堆栈顶。用不指定索引的 `pop()` 方法可以把一个元素从堆栈顶释放出来。例如：

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

将列表当作队列使用

也可以把列表当做队列用，只是在队列里第一加入的元素，第一个取出来；但是拿列表用作这样的目的效率不高。在列表的最后添加或者弹出元素速度快，然而在列表里插入或者从头部弹出速度却不快（因为所有其他的元素都得一个一个地移动）。

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")          # Terry arrives
>>> queue.append("Graham")        # Graham arrives
>>> queue.popleft()               # The first to arrive now leaves
'Eric'
>>> queue.popleft()               # The second to arrive now leaves
'John'
>>> queue                          # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

列表推导式

列表推导式提供了从序列创建列表的简单途径。通常应用程序将一些操作应用于某个序列的每个元素，用其获得的结果作为生成新列表的元素，或者根据确定的判定条件创建子序列。

每个列表推导式都在 for 之后跟一个表达式，然后有零到多个 for 或 if 子句。返回结果是一个根据表达从其后的 for 和 if 上下文环境中生成出来的列表。如果希望表达式推导出一个元组，就必须使用括号。

这里我们将列表中每个数值乘三，获得一个新的列表：

```
>>> vec = [2, 4, 6]
>>> [3*x for x in vec]
[6, 12, 18]
```

现在我们玩一点小花样：

```
>>> [[x, x**2] for x in vec]
[[2, 4], [4, 16], [6, 36]]
```

这里我们对序列里每一个元素逐个调用某方法：

```
>>> freshfruit = [' banana', ' loganberry ', 'passion fruit ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
```

我们可以用 if 子句作为过滤器：

```
>>> [3*x for x in vec if x > 3]
[12, 18]
>>> [3*x for x in vec if x < 2]
[]
```

以下是一些关于循环和其它技巧的演示：

```
>>> vec1 = [2, 4, 6]
>>> vec2 = [4, 3, -9]
>>> [x*y for x in vec1 for y in vec2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in vec1 for y in vec2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
>>> [vec1[i]*vec2[i] for i in range(len(vec1))]
[8, 12, -54]
```

列表推导式可以使用复杂表达式或嵌套函数：

```
>>> [str(round(355/113, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

嵌套列表解析

Python的列表还可以嵌套。

以下实例展示了3X4的矩阵列表：

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

以下实例将3X4的矩阵列表转换为4X3列表：

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

以下实例也可以使用以下方法来实现：

```
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

另外一种实现方法：

```
>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

del 语句

使用 del 语句可以从一个列表中依索引而不是值来删除一个元素。这与使用 pop() 返回一个值不同。可以用 del 语句从列表中删除一个切片，或清空整个列表（我们以前介绍的方法是给该切片赋一个空列表）。例如：

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
```

```
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

也可以用 del 删除实体变量：

```
>>> del a
```

元组和序列

元组由若干逗号分隔的值组成，例如：

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

如你所见，元组在输出时总是有括号的，以便于正确表达嵌套结构。在输入时可能有或没有括号，不过括号通常是必须的（如果元组是更大的表达式的一部分）。

集合

集合是一个无序不重复元素的集。基本功能包括关系测试和消除重复元素。

可以用大括号({})创建集合。注意：如果要创建一个空集合，你必须用 set() 而不是 {}；后者创建一个空的字典，下一节我们会介绍这个数据结构。

以下是一个简单的演示：

```
>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)          # 删除重复的
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket     # 检测成员
True
>>> 'crabgrass' in basket
False

>>> # 以下演示了两个集合的操作
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                      # a 中唯一的字母
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                  # 在 a 中的字母，但不在 b 中
{'r', 'd', 'b'}
>>> a | b                  # 在 a 或 b 中的字母
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                  # 在 a 和 b 中都有的字母
{'a', 'c'}
>>> a ^ b                  # 在 a 或 b 中的字母，但不同时在 a 和 b 中
{'r', 'd', 'b', 'm', 'z', 'l'}
```

集合也支持推导式：

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

字典

另一个非常有用的 Python 内建数据类型是字典。

序列是以连续的整数为索引，与此不同的是，字典以关键字为索引，关键字可以是任意不可变类型，通常用字符串或数值。

理解字典的最佳方式是把它看做无序的键=>值对集合。在同一个字典之内，关键字必须是互不相同。

一对大括号创建一个空的字典：{ }。

这是一个字典运用的简单例子：

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> list(tel.keys())
['irv', 'guido', 'jack']
>>> sorted(tel.keys())
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

构造函数 dict() 直接从键值对元组列表中构建字典。如果有固定的模式，列表推导式指定特定的键值对：

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

此外，字典推导可以用来创建任意键和值的表达式词典：

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

如果关键字只是简单的字符串，使用关键字参数指定键值对有时候更方便：

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

遍历技巧

在字典中遍历时，关键字和对应的值可以使用 items() 方法同时解读出来：

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

在序列中遍历时，索引位置 and 对应值可以使用 enumerate() 函数同时得到：

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
```

```
1 tac
2 toe
```

同时遍历两个或更多的序列，可以使用 `zip()` 组合：

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}? It is {1}.'.format(q, a))
...
What is your name? It is lancelot.
What is your quest? It is the holy grail.
What is your favorite color? It is blue.
```

要反向遍历一个序列，首先指定这个序列，然后调用 `reversed()` 函数：

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

要按顺序遍历一个序列，使用 `sorted()` 函数返回一个已排序的序列，并不修改原值：

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print(f)
...
apple
banana
orange
pear
```

参阅文档

- [Python3 列表](#)
- [Python3 元组](#)
- [Python3 字典](#)

Python3 模块

在前面的几个章节中我们脚本上是用 python 解释器来编程，如果你从 Python 解释器退出再进入，那么你定义的所有的方法和变量就都消失了。

为此 Python 提供了一个办法，把这些定义存放在文件中，为一些脚本或者交互式的解释器实例使用，这个文件被称为模块。

模块是一个包含所有你定义的函数和变量的文件，其后缀名是.py。模块可以被别的程序引入，以使用该模块中的函数等功能。这也是使用 python 标准库的方法。

下面是一个使用 python 标准库中模块的例子。

```
#!/usr/bin/python3
# 文件名：using_sys.py

import sys

print('命令行参数如下:')
for i in sys.argv:
    print(i)

print('\n\nPython 路径为: ', sys.path, '\n')
```

执行结果如下所示：

```
$ python using_sys.py 参数1 参数2
命令行参数如下:
using_sys.py
参数1
参数2
```

Python 路径为: ['root', '/usr/lib/python3.4', '/usr/lib/python3.4/plat-x86_64-linux-gnu', '/usr/lib/python3.4/lib-dynload', '/usr/local/lib/python3.4/dist-packages', '

- 1、import sys 引入 python 标准库中的 sys.py 模块；这是引入某一模块的方法。
- 2、sys.argv 是一个包含命令行参数的列表。
- 3、sys.path 包含了一个 Python 解释器自动查找所需模块的路径的列表。

import 语句

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

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

当解释器遇到 import 语句，如果模块在当前的搜索路径就会被导入。

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

support.py 文件代码为：

```
#!/usr/bin/python3
# Filename: support.py

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

test.py 引入 support 模块：

```
#!/usr/bin/python3
# Filename: test.py

# 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：

```
$ python3 test.py
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。

当我们使用import语句的时候，Python解释器是怎样找到对应的文件的呢？

这就涉及到Python的搜索路径，搜索路径是由一系列目录名组成的，Python解释器就依次从这些目录中去寻找所引入的模块。

这看起来很像环境变量，事实上，也可以通过定义环境变量的方式来确定搜索路径。

搜索路径是在Python编译或安装的时候确定的，安装新的库应该也会修改。搜索路径被存储在sys模块中的path变量，做一个简单的实验，在交互式解释器中，输入以下代码：

```
>>> import sys
>>> sys.path
['', '/usr/lib/python3.4', '/usr/lib/python3.4/plat-x86_64-linux-gnu', '/usr/lib/python3.4/lib-dynload', '/usr/local/lib/python3.4/dist-packages', '/usr/lib/python3.4/lib2to3']
>>>
```

sys.path 输出是一个列表，其中第一项是空串，代表当前目录（若是从一个脚本中打印出来的话，可以更清楚地看出是哪个目录），亦即我们执行python解释器的目录（对于脚本的话就是运行的脚本所在的目录）。

因此若像我一样在当前目录下存在与要引入模块同名的文件，就会把要引入的模块屏蔽掉。

了解了搜索路径的概念，就可以在脚本中修改sys.path来引入一些不在搜索路径中的模块。

现在，在解释器的当前目录或者 sys.path 中的一个目录里面来创建一个fibonacci.py的文件，代码如下：

```
# 斐波那契 (fibonacci) 数列模块

def fib(n):    # 定义到 n 的斐波那契数列
    a, b = 0, 1
    while b < n:
        print(b, end=' ')
        a, b = b, a+b
    print()
```

```
def fib2(n): # 返回到 n 的斐波那契数列
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

然后进入Python解释器，使用下面的命令导入这个模块：

```
>>> import fibo
```

这样做并没有把直接定义在fibo中的函数名称写入到当前符号表里，只是把模块fibo的名字写到了那里。

可以使用模块名称来访问函数：

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

如果你打算经常使用一个函数，你可以把它赋给一个本地的名称：

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

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 fibo 的 fib 函数，使用如下语句：

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

这个声明不会把整个fibo模块导入到当前的命名空间中，它只会将fibo里的fib函数引入进来。

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。

深入模块

模块除了方法定义，还可以包括可执行的代码。这些代码一般用来初始化这个模块。这些代码只有在第一次被导入时才会被执行。

每个模块有各自独立的符号表，在模块内部为所有的函数当作全局符号表来使用。

所以，模块的作者可以放心大胆的在模块内部使用这些全局变量，而不用担心把其他用户的全局变量搞花。

从另一个方面，当你确实知道你在做什么的话，你也可以通过 modname.itemname 这样的表示法来访问模块内的函数。

模块是可以导入其他模块的。在一个模块（或者脚本，或者其他地方）的最前面使用 import 来导入一个模块，当然这只是一个惯例，而不是强制的。被导入的模块的名称将被放入当前操作的模块的符号表中。

还有一种导入的方法，可以使用 import 直接把模块内（函数，变量的）名称导入到当前操作模块。比如：

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

这种导入的方法不会把被导入的模块的名称放在当前的字符表中（所以在这个例子里面，fibo 这个名称是没有定义的）。

这还有一种方法，可以一次性的把模块中的所有（函数，变量）名称都导入到当前模块的字符表：

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

这会将所有的名字都导入进来，但是那些由单一下划线（_）开头的名字不在此例。大多数情况，Python程序员不使用这种方法，因为引入的其它来源的命名，很可能覆盖了已有的定义。

__name__ 属性

一个模块被另一个程序第一次引入时，其主程序将运行。如果我们想在模块被引入时，模块中的某一程序块不执行，我们可以用 __name__ 属性来使该程序块仅在该模块自身运行时执行。

```
#!/usr/bin/python3
# Filename: using_name.py

if __name__ == '__main__':
    print('程序自身在运行')
else:
    print('我来自另一模块')
```

运行输出如下：

```
$ python using_name.py
```

程序自身在运行

```
$ python
>>> import using_name
我来自另一模块
>>>
```

说明： 每个模块都有一个 `__name__` 属性，当其值是 `'__main__'` 时，表明该模块自身在运行，否则是被引入。

dir() 函数

内置的函数 `dir()` 可以找到模块内定义的所有名称。以一个字符串列表的形式返回：

```
</p>
<pre>
>>> import fibo, sys
>>> dir(fibo)
['_name_', 'fib', 'fib2']
>>> dir(sys)
['_displayhook_', '__doc__', '__excepthook__', '__loader__', '__name__',
'__package__', '__stderr__', '__stdin__', '__stdout__',
'__clear_type_cache__', '__current_frames__', 'debugmallocstats', 'getframe',
'home', 'mercurial', 'xoptions', 'abiflags', 'api_version', 'argv',
'base_exec_prefix', 'base_prefix', 'builtin_module_names', 'byteorder',
'call_tracing', 'callstats', 'copyright', 'displayhook',
'dont_write_bytecode', 'exc_info', 'excepthook', 'exec_prefix',
'executable', 'exit', 'flags', 'float_info', 'float_repr_style',
'getcheckinterval', 'getdefaultencoding', 'getdlopenflags',
'getfilesystemencoding', 'getobjects', 'getprofile', 'getrecursionlimit',
'getrefcount', 'getsizeof', 'getswitchinterval', 'gettotalrefcount',
'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
'intern', 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path',
'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1',
'setcheckinterval', 'setdlopenflags', 'setprofile', 'setrecursionlimit',
'setswitchinterval', 'settrace', 'stderr', 'stdin', 'stdout',
'thread_info', 'version', 'version_info', 'warnoptions']
```

如果没有给定参数，那么 `dir()` 函数会罗列出当前定义的所有名称：

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir() # 得到一个当前模块中定义的属性列表
['_builtins_', '__name__', 'a', 'fib', 'fibo', 'sys']
>>> a = 5 # 建立一个新的变量 'a'
>>> dir()
['_builtins_', '__doc__', '__name__', 'a', 'sys']
>>>
>>> del a # 删除变量名a
>>>
>>> dir()
['_builtins_', '__doc__', '__name__', 'sys']
>>>
```

标准模块

Python 本身带着一些标准的模块库，在 Python 库参考文档中将会介绍到（就是后面的“库参考文档”）。

有些模块直接被构建在解析器里，这些虽然不是一些语言内置的功能，但是他却能很高效的使用，甚至是系统级调用也没问题。

这些组件会根据不同的操作系统进行不同形式的配置，比如 `wireg` 这个模块就只会提供给 Windows 系统。

应该注意到这有一个特别的模块 `sys`，它内置在每一个 Python 解析器中。变量 `sys.ps1` 和 `sys.ps2` 定义了主提示符和副提示符所对应的字符串：

```
>>> import sys
>>> sys.ps1
'>>>'
>>> sys.ps2
'...'
>>> sys.ps1 = 'C>'
C> print('Yuck!')
Yuck!
C>
```

包

包是一种管理 Python 模块命名空间的形式，采用“点模块名称”。

比如一个模块的名称是 `A.B`，那么他表示一个包 `A` 中的子模块 `B`。

就好像使用模块的时候，你不用担心不同模块之间的全局变量相互影响一样，采用点模块名称这种形式也不用担心不同库之间的模块重名的情况。

这样不同的作者都可以提供 NumPy 模块，或者是 Python 图形库。

不妨假设你想设计一套统一处理声音文件和数据的模块（或者称之为一个“包”）。

现存很多种不同的音频文件格式（基本上都是通过后缀名区分的，例如：`.wav`，`.file.aiff`，`.file.au`，），所以你需要有一组不断增加的模块，用来在不同的格式之间转换。

并且针对这些音频数据，还有很多不同的操作（比如混音，添加回声，增加均衡器功能，创建人造立体声效果），所以你还需要一组怎么也写不完的模块来处理这些操作。

这里给出了一种可能的包结构（在分层的文件系统中）：

sound/	顶层包
__init__.py	初始化 sound 包
formats/	文件格式转换子包
__init__.py	
wavread.py	
wavwrite.py	
aiffread.py	
aiffwrite.py	
auread.py	
auwrite.py	
...	
effects/	声音效果子包
__init__.py	

```
    echo.py
    surround.py
    reverse.py
    ...
filters/          filters 子包
    __init__.py
    equalizer.py
    vocoder.py
    karaoke.py
    ...
```

在导入一个包的时候，Python会根据 sys.path 中的目录来寻找这个包中包含的子目录。

目录只有包含一个叫做 `__init__.py` 的文件才会被认作是一个包，主要是为了避免一些滥俗的名字（比如叫做 string）不小心的影响搜索路径中的有效模块。

最简单的情况，放一个空的 `.file: __init__.py` 就可以了。当然这个文件中也可以包含一些初始化代码或者为（将在后面介绍的） `__all__` 变量赋值。

用户可以每次只导入一个包里面的特定模块，比如：

```
import sound.effects.echo
```

这将会导入子模块 `sound.effects.echo`。他必须使用全名去访问：

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

还有一种导入子模块的方法是：

```
from sound.effects import echo
```

这同样会导入子模块 `echo`，并且他不需要那些冗长的前缀，所以他可以这样使用：

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

还有一种变化就是直接导入一个函数或者变量：

```
from sound.effects.echo import echofilter
```

同样的，这种方法会导入子模块 `echo`，并且可以直接使用他的 `echofilter()` 函数：

```
echofilter(input, output, delay=0.7, atten=4)
```

注意当使用 `from package import item` 这种形式的时候，对应的 `item` 既可以是包里面的子模块（子包），或者包里面定义的其他名称，比如函数，类或者变量。

`import` 语法会首先把 `item` 当作一个包定义的名称，如果没找到，再试图按照一个模块去导入。如果还没找到，恭喜，一个 `ImportError` 异常被抛出了。

反之，如果使用形如 `import item.subitem.subsubitem` 这种导入形式，除了最后一项，都必须是包，而最后一项则可以是模块或者是包，但是不可以是类，函数或者变量的名字。

从一个包中导入 *

设想一下，如果我们使用 `from sound.effects import *` 会发生什么？

Python 会进入文件系统，找到这个包里面所有的子模块，一个一个的把它们都导入进来。

但是很不幸，这个方法在 Windows 平台上工作的就不是非常好，因为 Windows 是一个大小写不区分的系统。

在这类平台上，没有人敢担保一个叫做 `ECHO.py` 的文件导入为模块 `echo` 还是 `Echo` 甚至 `ECHO`。

（例如，Windows 95 就很讨厌的把每一个文件的首字母大写显示）而且 DOS 的 8+3 命名规则对长模块名称的处理会把问题搞得更纠结。

为了解决这个问题，只能烦劳包作者提供一个精确的包的索引了。

导入语句遵循如下规则：如果包定义文件 `__init__.py` 存在一个叫做 `__all__` 的列表变量，那么在使用 `from package import *` 的时候就把这个列表中的所有名字作为包内容导入。

作为包的作者，可别忘了在更新包之后保证 `__all__` 也更新了啊。你说我就不这么做，我就不使用导入 `*` 这种用法，好吧，没问题，谁让你老板呢。这里有一个例子，在 `file:sounds/effects/__init__.py` 中包含如下代码：

```
__all__ = ["echo", "surround", "reverse"]
```

这表示当你使用 `from sound.effects import *` 这种用法时，你只会导入包里面这三个子模块。

如果 `__all__` 真的没有定义，那么使用 `from sound.effects import *` 这种语法的时候，就不会导入包 `sound.effects` 里的任何子模块。他只是把包 `sound.effects` 和它里面定义的所有内容导入进来（可能运行 `__init__.py` 里定义的初始化代码）。

这会把 `__init__.py` 里面定义的所有名字导入进来。并且他不会破坏掉我们在这句话之前导入的所有明确指定的模块。看下这部分代码：

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

这个例子中，在执行 `from...import` 前，包 `sound.effects` 中的 `echo` 和 `surround` 模块都被导入到当前的命名空间中了。（当然如果定义了 `__all__` 就更没问题了）

通常我们并不主张使用 `*` 这种方法来导入模块，因为这种方法经常会导致代码的可读性降低。不过这样倒的确是可以省去不少敲键的功夫，而且一些模块都设计成了只能通过特定的方法导入。

记住，使用 `from Package import specific_submodule` 这种方法永远不会有错。事实上，这也是推荐的方法。除非是你导入的子模块有可能和其他包的子模块重名。

如果在结构中包是一个子包（比如这个例子中对于包 `sound` 来说），而你又想导入兄弟包（同级别的包）你就得使用导入绝对的路径来导入。比如，如果模块 `sound.filters.vocoder` 要使用包 `sound.effects` 中的模块 `echo`，你就要写成 `from sound.effects import echo`。

```
from . import echo
from .. import formats
from ..filters import equalizer
```

无论是隐式的还是显式的相对导入都是从当前模块开始的。主模块的名字永远是 `"__main__"`，一个 Python 应用程序的主模块，应当总是使用绝对路径引用。

包还提供一个额外的属性 `__path__`。这是一个目录列表，里面每一个包含的目录都有为这个包服务的 `__init__.py`，你得在其他 `__init__.py` 被执行前定义哦。可以修改这个变量，用来影响包包含在包里面的模块和子包。

这个功能并不常用，一般用来扩展包里面的模块。

Python3 输入和输出

在前面几个章节中，我们其实已经接触了 Python 的输入输出的功能。本章节我们将具体介绍 Python 的输入输出。

输出格式美化

Python 两种输出值的方式: 表达式语句和 `print()` 函数。

第三种方式是使用文件对象的 `write()` 方法，标准输出文件可以用 `sys.stdout` 引用。

如果你希望输出的形式更加多样，可以使用 `str.format()` 函数来格式化输出值。

如果你希望将输出的值转成字符串，可以使用 `repr()` 或 `str()` 函数来实现。

- **str():** 函数返回一个用户易读的表达形式。
- **repr():** 产生一个解释器易读的表达形式。

例如

```
>>> s = 'Hello, Runoob'
>>> str(s)
'Hello, Runoob'
>>> repr(s)
"'Hello, Runoob'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'x 的值为: ' + repr(x) + ', y 的值为: ' + repr(y) + '...'
>>> print(s)
x 的值为: 32.5, y 的值为: 40000...
>>> # repr() 函数可以转义字符串中的特殊字符
... hello = 'hello, runoob\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, runoob\n'
>>> # repr() 的参数可以是 Python 的任何对象
... repr(x, y, ('Google', 'Runoob'))
"(32.5, 40000, ('Google', 'Runoob'))"
```

这里有两种方式输出一个平方与立方的表:

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # 注意前一行 'end' 的使用
...     print(repr(x*x*x).rjust(4))
...
1   1   1
2   4   8
3   9  27
4  16  64
5  25 125
6  36 216
7  49 343
8  64 512
9  81 729
10 100 1000

>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
1   1   1
2   4   8
3   9  27
```

```
4 16 64
5 25 125
6 36 216
7 49 343
8 64 512
9 81 729
10 100 1000
```

注意：在第一个例子中, 每列间的空格由 `print()` 添加。

这个例子展示了字符串对象的 `rjust()` 方法, 它可以将字符串靠右, 并在左边填充空格。

还有类似的方法, 如 `ljust()` 和 `center()`。这些方法并不会写任何东西, 它们仅仅返回新的字符串。

另一个方法 `zfill()`, 它会在数字的左边填充 0, 如下所示:

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

`str.format()` 的基本使用如下:

```
>>> print('{}网址: {}'.format('菜鸟教程', 'www.runoob.com'))
菜鸟教程网址: www.runoob.com!
```

括号及其里面的字符 (称作格式化字段) 将会被 `format()` 中的参数替换。

在括号中的数字用于指向传入对象在 `format()` 中的位置, 如下所示:

```
>>> print('{0} 和 {1}'.format('Google', 'Runoob'))
Google 和 Runoob
>>> print('{1} 和 {0}'.format('Google', 'Runoob'))
Runoob 和 Google
```

如果在 `format()` 中使用了关键字参数, 那么它们的值会指向使用该名字的参数。

```
>>> print('{name}网址: {site}'.format(name='菜鸟教程', site='www.runoob.com'))
菜鸟教程网址: www.runoob.com
```

位置及关键字参数可以任意的结合:

```
>>> print('站点列表 {0}, {1}, 和 {other}'.format('Google', 'Runoob',
                                                    other='Taobao'))
站点列表 Google, Runoob, 和 Taobao。
```

'`a`' (使用 `ascii()`), '`s`' (使用 `str()`) 和 '`r`' (使用 `repr()`) 可以用于在格式化某个值之前对其进行转化:

```
>>> import math
>>> print('常量 PI 的值近似为: {}'.format(math.pi))
常量 PI 的值近似为: 3.141592653589793。
>>> print('常量 PI 的值近似为: {!r}'.format(math.pi))
常量 PI 的值近似为: 3.141592653589793。
```

可选项 '`f`' 和格式标识符可以跟着字段名。这就允许对值进行更好的格式化。下面的例子将 Pi 保留到小数点后三位:

```
>>> import math
>>> print('常量 PI 的值近似为 {:.3f}'.format(math.pi))
常量 PI 的值近似为 3.142。
```

在 '`f`' 后传入一个整数, 可以保证该域至少有这么多的宽度。用于美化表格时很有用。

```
>>> table = {'Google': 1, 'Runoob': 2, 'Taobao': 3}
>>> for name, number in table.items():
...     print('{0:10} ==> {1:10d}'.format(name, number))
...
```



```
Runoob    ==>      2
Taobao    ==>      3
Google    ==>      1
```

如果你有一个很长的格式化字符串, 而你不想将它们分开, 那么在格式化时通过变量名而非位置会是很好的事情。

最简单的就是传入一个字典, 然后使用方括号 '[' 来访问键值:

```
>>> table = {'Google': 1, 'Runoob': 2, 'Taobao': 3}
>>> print('Runoob: {0[Runoob]:d}; Google: {0[Google]:d}; Taobao: {0[Taobao]:d}'.format(table))
Runoob: 2; Google: 1; Taobao: 3
```

也可以通过在 table 变量前使用 '**' 来实现相同的功能:

```
>>> table = {'Google': 1, 'Runoob': 2, 'Taobao': 3}
>>> print('Runoob: {Runoob:d}; Google: {Google:d}; Taobao: {Taobao:d}'.format(**table))
Runoob: 2; Google: 1; Taobao: 3
```

旧式字符串格式化

% 操作符也可以实现字符串格式化。它将左边的参数作为类似 `sprintf()` 式的格式化字符串, 而将右边的代入, 然后返回格式化后的字符串。例如:

```
>>> import math
>>> print('常量 PI 的值近似为: %5.3f.' % math.pi)
常量 PI 的值近似为: 3.142。
```

因为 `str.format()` 比较新的函数, 大多数的 Python 代码仍然使用 % 操作符。但是因为这种旧式的格式化最终会从该语言中移除, 应该更多的使用 `str.format()`。

读取键盘输入

Python 提供了 `input()` 函数从标准输入读入一行文本, 默认的标准输入是键盘。

`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/python3

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: 菜鸟教程
你输入的内容是:  菜鸟教程
```

读和写文件

`open()` 将会返回一个 `file` 对象, 基本语法格式如下:

```
open(filename, mode)
```

- **filename:** `filename` 变量是一个包含了你要访问的文件名称的字符串值。
- **mode:** `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`)。

不同模式打开文件的完全列表:

模式	描述
<code>r</code>	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。

下图很好的总结了这几种模式：

□

模式	r	r+	w	w+	a	a+
读	+	+			+	+
写			+	+	+	+
创建			+	+	+	+
覆盖			+	+		
指针在开始	+	+	+	+		
指针在结尾					+	+

以下实例将字符串写入到文件 foo.txt 中：

```
#!/usr/bin/python3

# 打开一个文件
f = open("/tmp/foo.txt", "w")

f.write( "Python 是一个非常好的语言。\\n是的，的确非常好!!\\n" )

# 关闭打开的文件
f.close()
```

- 第一个参数为要打开的文件名。
- 第二个参数描述文件如何使用的字符。 mode 可以是 'r' 如果文件只读, 'w' 只用于写 (如果存在同名文件则将被删除), 和 'a' 用于追加文件内容; 所写的任何数据都会被自动增加到末尾. 'r+' 同时用于读写。 mode 参数是可选的; 'r' 将是默认值。

此时打开文件 foo.txt,显示如下：

```
$ cat /tmp/foo.txt
Python 是一个非常好的语言。
是的，的确非常好!!
```

文件对象的方法

本节中剩下的例子假设已经创建了一个称为 f 的文件对象。

f.read()

为了读取一个文件的内容，调用 `f.read(size)`，这将读取一定数目的数据，然后作为字符串或字节对象返回。

`size` 是一个可选的数字类型的参数。当 `size` 被忽略了或者为负，那么该文件的所有内容都将被读取并且返回。

以下实例假定文件 `foo.txt` 已存在（上面实例中已创建）：

```
#!/usr/bin/python3

# 打开一个文件
f = open("/tmp/foo.txt", "r")

str = f.read()
print(str)

# 关闭打开的文件
f.close()
```

执行以上程序，输出结果为：

```
Python 是一个非常好的语言。
是的，的确非常好!!
```

f.readline()

`f.readline()` 会从文件中读取单独的一行。换行符为 `\n`。`f.readline()` 如果返回一个空字符串，说明已经已经读取到最后一行。

```
#!/usr/bin/python3

# 打开一个文件
f = open("/tmp/foo.txt", "r")

str = f.readline()
print(str)

# 关闭打开的文件
f.close()
```

执行以上程序，输出结果为：

```
Python 是一个非常好的语言。
```

f.readlines()

`f.readlines()` 将返回该文件中包含的所有行。

如果设置可选参数 `sizehint`，则读取指定长度的字节，并且将这些字节按行分割。

```
#!/usr/bin/python3

# 打开一个文件
f = open("/tmp/foo.txt", "r")

str = f.readlines()
print(str)

# 关闭打开的文件
f.close()
```

执行以上程序，输出结果为：

```
['Python 是一个非常好的语言。\\n', '是的，的确非常好!!\\n']
```

另一种方式是迭代一个文件对象然后读取每行：

```
#!/usr/bin/python3

# 打开一个文件
f = open("/tmp/foo.txt", "r")

for line in f:
    print(line, end='')

# 关闭打开的文件
f.close()
```

执行以上程序，输出结果为：

Python 是一个非常好的语言。
是的，的确非常好!!

这个方法很简单,但是并没有提供一个很好的控制。因为两者的处理机制不同,最好不要混用。

f.write()

f.write(string) 将 string 写入到文件中, 然后返回写入的字符数。

```
#!/usr/bin/python3

# 打开一个文件
f = open("/tmp/foo.txt", "w")

num = f.write( "Python 是一个非常好的语言。\\n是的，的确非常好!!\\n" )
print(num)
# 关闭打开的文件
f.close()
```

执行以上程序，输出结果为：

29

如果要写入一些不是字符串的东西, 那么将需要先进行转换:

```
#!/usr/bin/python3

# 打开一个文件
f = open("/tmp/fool.txt", "w")

value = ('www.runoob.com', 14)
s = str(value)
f.write(s)

# 关闭打开的文件
f.close()
```

执行以上程序，打开 fool.txt 文件：

```
$ cat /tmp/fool.txt
('www.runoob.com', 14)
```

f.tell()

f.tell() 返回文件对象当前所处的位置, 它是从文件开头开始算起的字节数。

f.seek()

如果要改变文件当前的位置, 可以使用 f.seek(offset, from_what) 函数。

from_what 的值, 如果是 0 表示开头, 如果是 1 表示当前位置, 2 表示文件的结尾, 例如:

- seek(x,0): 从起始位置即文件首行首字符开始移动 x 个字符

- `seek(x,1)`: 表示从当前位置往后移动x个字符
- `seek(-x,2)`: 表示从文件的结尾往前移动x个字符

`from_what` 值为默认为0, 即文件开头。下面给出一个完整的例子:

```
>>> f = open('/tmp/foo.txt', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)      # 移动到文件的第六个字节
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2) # 移动到文件的倒数第三字节
13
>>> f.read(1)
b'd'
```

f.close()

在文本文件中 (那些打开文件的模式下没有 `b` 的), 只会相对于文件起始位置进行定位。

当你处理完一个文件后, 调用 `f.close()` 来关闭文件并释放系统的资源, 如果尝试再调用该文件, 则会抛出异常。

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

当处理一个文件对象时, 使用 `with` 关键字是非常好的方式。在结束后, 它会帮你正确的关闭文件。而且写起来也比 `try-finally` 语句块要简短:

```
>>> with open('/tmp/foo.txt', 'r') as f:
...     read_data = f.read()
>>> f.closed
True
```

文件对象还有其他方法, 如 `isatty()` 和 `truncate()`, 但这些通常比较少用。

pickle 模块

python的pickle模块实现了基本的数据序列和反序列化。

通过pickle模块的序列化操作我们能够将程序中运行的对象信息保存到文件中去, 永久存储。

通过pickle模块的反序列化操作, 我们能够从文件中创建上一次程序保存的对象。

基本接口:

```
pickle.dump(obj, file, [,protocol])
```

有了 `pickle` 这个对象, 就能对 `file` 以读取的形式打开:

```
x = pickle.load(file)
```

注解: 从 `file` 中读取一个字符串, 并将它重构为原来的python对象。

file: 类文件对象, 有`read()`和`readline()`接口。

实例1:

```
#!/usr/bin/python3
import pickle
```

```
# 使用pickle模块将数据对象保存到文件
data1 = {'a': [1, 2.0, 3, 4+6j],
        'b': ('string', u'Unicode string'),
        'c': None}

selfref_list = [1, 2, 3]
selfref_list.append(selfref_list)

output = open('data.pkl', 'wb')

# Pickle dictionary using protocol 0.
pickle.dump(data1, output)

# Pickle the list using the highest protocol available.
pickle.dump(selfref_list, output, -1)

output.close()
```

实例2:

```
#!/usr/bin/python3
import pprint, pickle

#使用pickle模块从文件中重构python对象
pkl_file = open('data.pkl', 'rb')

data1 = pickle.load(pkl_file)
pprint.pprint(data1)

data2 = pickle.load(pkl_file)
pprint.pprint(data2)

pkl_file.close()
```

Python3 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	方法及描述
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([sizeint]) 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 sizeint 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]) 从文件的首行首字符开始截断，截断文件为 size 个字符，无 size 表示从当前位置截断；截断之后 V 后面的所有

字符被删除，其中 Windows 系统下的换行代表2个字符大小。

[file.write\(str\)](#)

12

将字符串写入文件，没有返回值。

[file.writelines\(sequence\)](#)

13

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。

Python3 OS 文件/目录方法

os 模块提供了非常丰富的方法用来处理文件和目录。常用的方法如下表所示：

序号	方法及描述
1	os.access(path, mode) 检验权限模式
2	os.chdir(path) 改变当前工作目录
3	os.chflags(path, flags) 设置路径的标记为数字标记。
4	os.chmod(path, mode) 更改权限
5	os.chown(path, uid, gid) 更改文件所有者
6	os.chroot(path) 改变当前进程的根目录
7	os.close(fd) 关闭文件描述符 fd
8	os.closerange(fd_low, fd_high) 关闭所有文件描述符，从 fd_low (包含) 到 fd_high (不包含), 错误会忽略
9	os.dup(fd) 复制文件描述符 fd
10	os.dup2(fd, fd2) 将一个文件描述符 fd 复制到另一个 fd2
11	os.fchdir(fd)

通过文件描述符改变当前工作目录

[os.fchmod\(fd, mode\)](#)

12

改变一个文件的访问权限，该文件由参数fd指定，参数mode是Unix下的文件访问权限。

[os.fchown\(fd, uid, gid\)](#)

13

修改一个文件的所有权，这个函数修改一个文件的用户ID和用户组ID，该文件由文件描述符fd指定。

[os.fdatasync\(fd\)](#)

14

强制将文件写入磁盘，该文件由文件描述符fd指定，但是不强制更新文件的状态信息。

[os.flopen\(fd\[, mode\[, bufsize\]\]\)](#)

15

通过文件描述符 fd 创建一个文件对象，并返回这个文件对象

[os.fpathconf\(fd, name\)](#)

16

返回一个打开的文件的系统配置信息。name为检索的系统配置的值，它也许是一个定义系统值的字符串，这些名字在很多标准中指定（POSIX.1, Unix 95, Unix 98, 和其它）。

[os.fstat\(fd\)](#)

17

返回文件描述符fd的状态，像stat()。

[os.fstatvfs\(fd\)](#)

18

返回包含文件描述符fd的文件的文件系统的信息，像 statvfs()

[os.fsync\(fd\)](#)

19

强制将文件描述符为fd的文件写入硬盘。

[os.ftruncate\(fd, length\)](#)

20

裁剪文件描述符fd对应的文件, 所以它最大不能超过文件大小。

[os.getcwd\(\)](#)

21

返回当前工作目录

[os.getcwdu\(\)](#)

22

返回一个当前工作目录的Unicode对象

[os.isatty\(fd\)](#)

23

如果文件描述符fd是打开的，同时与tty(-like)设备相连，则返回true, 否则False。

[os.lchflags\(path, flags\)](#)

24

设置路径的标记为数字标记，类似 chflags(), 但是没有软链接

[os.lchmod\(path, mode\)](#)

25

修改连接文件权限

[os.lchown\(path, uid, gid\)](#)

26

更改文件所有者，类似 chown, 但是不追踪链接。

[os.link\(src, dst\)](#)

27

创建硬链接，名为参数 dst, 指向参数 src

[os.listdir\(path\)](#)

28

返回path指定的文件夹包含的文件或文件夹的名字的列表。

[os.lseek\(fd, pos, how\)](#)

29

设置文件描述符 fd当前位置为pos, how方式修改: SEEK_SET 或者 0 设置从文件开始的计算的pos; SEEK_CUR或者 1 则从当前位置计算; os.SEEK_END或者2则从文件尾部开始. 在unix, Windows中有效

[os.lstat\(path\)](#)

30

像stat(),但是没有软链接

[os.major\(device\)](#)

31

从原始的设备号中提取设备major号码 (使用stat中的st_dev或者st_rdev field)。

[os.makedev\(major, minor\)](#)

32

以major和minor设备号组成一个原始设备号

[os.makedirs\(path\[, mode\]\)](#)

33

递归文件夹创建函数。像mkdir(), 但创建的所有intermediate-level文件夹需要包含子文件夹。

[os.minor\(device\)](#)

34

从原始的设备号中提取设备minor号码 (使用stat中的st_dev或者st_rdev field)。

[os.mkdir\(path\[, mode\]\)](#)

35

以数字mode的mode创建一个名为path的文件夹.默认的 mode 是 0777 (八进制)。

[os.mkfifo\(path\[, mode\]\)](#)

36

创建命名管道, mode 为数字, 默认为 0666 (八进制)

[os.mknod\(filename\[, mode=0600, device\]\)](#)

37

创建一个名为filename文件系统节点 (文件, 设备特别文件或者命名pipe) 。

[os.open\(file, flags\[, mode\]\)](#)

38

打开一个文件, 并且设置需要的打开选项, mode参数是可选的

[os.openpty\(\)](#)

39

打开一个新的伪终端对。返回 pty 和 tty的文件描述符。

[os.pathconf\(path, name\)](#)

40

返回相关文件的系统配置信息。

[os.pipe\(\)](#)

41

创建一个管道. 返回一对文件描述符(r, w) 分别为读和写

[os.popen\(command\[, mode\[, bufsize\]\]\)](#)

42

从一个 command 打开一个管道

[os.read\(fld, n\)](#)

43

从文件描述符 fld 中读取最多 n 个字节, 返回包含读取字节的字符串, 文件描述符 fld对应文件已达到结尾, 返回一个空字符串。

[os.readlink\(path\)](#)

44

返回软链接所指向的文件

[os.remove\(path\)](#)

45

删除路径为path的文件。如果path是一个文件夹, 将抛出OSError; 查看下面的rmdir()删除一个 directory。

[os.removedirs\(path\)](#)

46

递归删除目录。

[os.rename\(src, dst\)](#)

47

重命名文件或目录，从 src 到 dst

[os.rename\(old, new\)](#)

48

递归地对目录进行更名，也可以对文件进行更名。

[os.rmdir\(path\)](#)

49

删除path指定的空目录，如果目录非空，则抛出一个OSError异常。

[os.stat\(path\)](#)

50

获取path指定的路径的信息，功能等同于C API中的stat()系统调用。

[os.stat_float_times\(\[newvalue\]\)](#)

51

决定stat_result是否以float对象显示时间戳

[os.statvfs\(path\)](#)

52

获取指定路径的文件系统统计信息

[os.symlink\(src, dst\)](#)

53

创建一个软链接

[os.tcgetpgrp\(fd\)](#)

54

返回与终端fd（一个由os.open()返回的打开的文件描述符）关联的进程组

[os.tcsetpgrp\(fd, pg\)](#)

55

设置与终端fd（一个由os.open()返回的打开的文件描述符）关联的进程组为pg。

[os.tmpnam\(\[dir\[, prefix\]\]\)](#)

56

Python3 中已删除。 返回唯一的路径名用于创建临时文件。

[os.tmpfile\(\)](#)

57

Python3 中已删除。 返回一个打开的模式为(w+b)的文件对象。这文件对象没有文件夹入口，没有文件描述符，将会自动删除。

[os.tmpnam\(\)](#)

58

Python3 中已删除。 为创建一个临时文件返回一个唯一的路径

59

[os.ttyname\(fd\)](#)

返回一个字符串，它表示与文件描述符fd 关联的终端设备。如果fd 没有与终端设备关联，则引发一个异常。

60

[os.unlink\(path\)](#)

删除文件路径

61

[os.utime\(path, times\)](#)

返回指定的path文件的访问和修改的时间。

62

[os.walk\(top\[, topdown=True\[, onerror=None\[, followlinks=False\]\]\]\)](#)

输出在文件夹中的文件名通过在树中行走，向上或者向下。

63

[os.write\(fd, str\)](#)

写入字符串到文件描述符 fd中. 返回实际写入的字符串长度

Python3 错误和异常

作为Python初学者，在刚学习Python编程时，经常会看到一些报错信息，在前面我们没有提及，这章节我们会专门介绍。

Python有两种错误很容易辨认：语法错误和异常。

语法错误

Python 的语法错误或者称之为解析错，是初学者经常碰到的，如下实例

```
>>> while True print('Hello world')
      File "<stdin>", line 1, in ?
        while True print('Hello world')
                        ^
SyntaxError: invalid syntax
```

这个例子中，函数 print() 被检查到有错误，是它前面缺少了一个冒号 (:)。

语法分析器指出了出错的一行，并且在最先找到的错误的位置标记了一个小小的箭头。

异常

即便Python程序的语法是正确的，在运行它的时候，也有可能发生错误。运行期检测到的错误被称为异常。

大多数的异常都不会被程序处理，都以错误信息的形式展现在这里：

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: Can't convert 'int' object to str implicitly
```

异常以不同的类型出现，这些类型都作为信息的一部分打印出来：例子中的类型有 ZeroDivisionError, NameError 和 TypeError。

错误信息的前面部分显示了异常发生的上下文，并以调用栈的形式显示具体信息。

异常处理

以下例子中，让用户输入一个合法的整数，但是允许用户中断这个程序（使用 Control-C 或者操作系统提供的方法）。用户中断的信息会引发一个 KeyboardInterrupt 异常。

```
>>> while True:
    try:
        x = int(input("Please enter a number: "))
        break
    except ValueError:
        print("Oops! That was no valid number. Try again ")
```

try语句按照如下方式工作：

- 首先，执行try子句（在关键字try和关键字except之间的语句）
- 如果没有异常发生，忽略except子句，try子句执行后结束。

- 如果在执行try子句的过程中发生了异常，那么try子句余下的部分将被忽略。如果异常的类型和 except 之后的名称相符，那么对应的except子句将被执行。最后执行 try 语句之后的代码。
- 如果一个异常没有与任何的except匹配，那么这个异常将会传递给上层的try中。

一个 try 语句可能包含多个except子句，分别来处理不同的特定的异常。最多只有一个分支会被执行。

处理程序将只针对对应的try子句中的异常进行处理，而不是其他的 try 的处理程序中的异常。

一个except子句可以同时处理多个异常，这些异常将被放在一个括号里成为一个元组，例如：

```
except (RuntimeError, TypeError, NameError):
    pass
```

最后一个except子句可以忽略异常的名称，它将被当作通配符使用。你可以使用这种方法打印一个错误信息，然后再次把异常抛出。

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

try except 语句还有一个可选的else子句，如果使用这个子句，那么必须放在所有的except子句之后。这个子句将在try子句没有发生任何异常的时候执行。例如：

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

使用 else 子句比把所有的语句都放在 try 子句里面要好，这样可以避免一些意想不到的、而except又没有捕获的异常。

异常处理并不仅仅处理那些直接发生在try子句中的异常，而且还能处理子句中调用的函数（甚至间接调用的函数）里抛出的异常。例如：

```
>>> def this_fails():
    x = 1/0

>>> try:
    this_fails()
except ZeroDivisionError as err:
    print('Handling run-time error:', err)

Handling run-time error: int division or modulo by zero
```

抛出异常

Python使用 raise 语句抛出一个指定的异常。例如：

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: HiThere
```


`raise` 唯一的一个参数指定了要被抛出的异常。它必须是一个异常的实例或者是异常的类（也就是 `Exception` 的子类）。

如果你只想知道这是否抛出了一个异常，并不想去处理它，那么一个简单的 `raise` 语句就可以再次把它抛出。

```
>>> try:
    raise NameError('HiThere')
except NameError:
    print('An exception flew by!')
    raise
```

```
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError: HiThere
```

用户自定义异常

你可以通过创建一个新的`exception`类来拥有自己的异常。异常应该继承自 `Exception` 类，或者直接继承，或者间接继承，例如：

```
>>> class MyError(Exception):
    def __init__(self, value):
        self.value = value
    def __str__(self):
        return repr(self.value)

>>> try:
    raise MyError(2*2)
except MyError as e:
    print('My exception occurred, value:', e.value)
```

```
My exception occurred, value: 4
>>> raise MyError('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MyError: 'oops!'
```

在这个例子中，类 `Exception` 默认的 `__init__()` 被覆盖。

当创建一个模块有可能抛出多种不同的异常时，一种通常的做法是为这个包建立一个基础异常类，然后基于这个基础类为不同的错误情况创建不同的子类：

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expression -- input expression in which the error occurred
        message -- explanation of the error
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
    """Raised when an operation attempts a state transition that's not
    allowed.

    Attributes:
        previous -- state at beginning of transition
        next -- attempted new state
        message -- explanation of why the specific transition is not allowed
    """
```

```
def __init__(self, previous, next, message):
    self.previous = previous
    self.next = next
    self.message = message
```

大多数的异常的名字都以"Error"结尾，就跟标准的异常命名一样。

定义清理行为

try 语句还有另外一个可选的子句，它定义了无论在任何情况下都会执行的清理行为。例如：

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt
```

以上例子不管 try 子句里面有没有发生异常，finally 子句都会执行。

如果一个异常在 try 子句里（或者在 except 和 else 子句里）被抛出，而又没有任何的 except 把它截住，那么这个异常会在 finally 子句执行后再次被抛出。

下面是一个更加复杂的例子（在同一个 try 语句里包含 except 和 finally 子句）：

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")

>>> divide(2, 1)
result is 2.0
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

预定义的清理行为

一些对象定义了标准的清理行为，无论系统是否成功的使用了它，一旦不需要它了，那么这个标准的清理行为就会执行。

这面这个例子展示了尝试打开一个文件，然后把内容打印到屏幕上：

```
for line in open("myfile.txt"):
    print(line, end="")
```

以上这段代码的问题是，当执行完毕后，文件会保持打开状态，并没有被关闭。

关键词 `with` 语句就可以保证诸如文件之类的对象在使用完之后一定会正确的执行他的清理方法:

```
with open("myfile.txt") as f:
    for line in f:
        print(line, end="")
```

以上这段代码执行完毕后，就算在处理过程中出问题了，文件 `f` 总是会关闭。

Python3 面向对象

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

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

接下来我们先来简单的了解下面面向对象的一些基本特征。

面向对象技术简介

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

和其它编程语言相比，Python 在尽可能不增加新的语法和语义的情况下加入了类机制。

Python中的类提供了面向对象编程的所有基本功能：类的继承机制允许多个基类，派生类可以覆盖基类中的任何方法，方法中可以调用基类中的同名方法。

对象可以包含任意数量和类型的数据。

类定义

语法格式如下：

```
class ClassName: <statement-1> ... <statement-N>
```

类实例化后，可以使用其属性，实际上，创建一个类之后，可以通过类名访问其属性。

类对象

类对象支持两种操作：属性引用和实例化。

属性引用使用 Python 中所有的属性引用一样的标准语法：**obj.name**。

类对象创建后，类命名空间中所有的命名都是有效属性名。所以如果类定义是这样：

实例(Python 3.0+)

```
#!/usr/bin/python3 class MyClass: """一个简单的类实例""" i = 12345 def f(self): return 'hello world' # 实例化类 x = MyClass() # 访问类的属性和方法 print("MyClass 类的属性 i 为：", x.i) print("MyClass 类的方法 f 输出为：", x.f())
```

以上创建了一个新的类实例并将该对象赋给局部变量 x，x 为空的对象。

执行以上程序输出结果为：

```
MyClass 类的属性 i 为: 12345
MyClass 类的方法 f 输出为: hello world
```

很多类都倾向于将对象创建为有初始状态的。因此类可能会定义一个名为 `__init__()` 的特殊方法（构造方法），像下面这样：

```
def __init__(self): self.data = []
```

类定义了 `__init__()` 方法的话，类的实例化操作会自动调用 `__init__()` 方法。所以在下例中，可以这样创建一个新的实例：

```
x = MyClass()
```

当然，`__init__()` 方法可以有参数，参数通过 `__init__()` 传递到类的实例化操作上。例如：

实例(Python 3.0+)

```
#!/usr/bin/python3 class Complex: def __init__(self, realpart, imagpart): self.r = realpart self.i = imagpart x = Complex(3.0, -4.5)
print(x.r, x.i) # 输出结果: 3.0 -4.5
```

self代表类的实例，而非类

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，按照惯例它的名称是 `self`。

```
class Test: def prt(self): print(self) print(self.__class__) t = Test() t.prt()
```

以上实例执行结果为：

```
<__main__.Test instance at 0x100771878>
__main__.Test
```

从执行结果可以很明显的看出，`self`代表的是类的实例，代表当前对象的地址，而 `self.class` 则指向类。

`self` 不是 python 关键字，我们把他换成 `runoob` 也是可以正常执行的：

```
class Test: def prt(runoob): print(runoob) print(runoob.__class__) t = Test() t.prt()
```

以上实例执行结果为：

```
<__main__.Test instance at 0x100771878>
__main__.Test
```

类的方法

在类地内部，使用 `def` 关键字来定义一个方法，与一般函数定义不同，类方法必须包含参数 `self`，且为第一个参数，`self` 代表的是类的实例。

实例(Python 3.0+)

```
#!/usr/bin/python3 #类定义 class people: #定义基本属性 name = " age = 0 #定义私有属性,私有属性在类外部无法直接进行访问
__weight = 0 #定义构造方法 def __init__(self,n,a,w): self.name = n self.age = a self.__weight = w def speak(self): print("%s 说: 我
%d 岁。"%(self.name,self.age)) # 实例化类 p = people('runoob',10,30) p.speak()
```

执行以上程序输出结果为：

```
runoob 说: 我 10 岁。
```

继承

Python 同样支持类的继承，如果一种语言不支持继承，类就没有什么意义。派生类的定义如下所示：

```
class DerivedClassName(BaseClassName1): <statement-1> ... <statement-N>
```

需要注意圆括号中基类的顺序，若是基类中有相同的方法名，而在子类使用时未指定，python 从左至右搜索 即方法在子类中未找到时，从左到右查找基类中是否包含方法。

BaseClassName（示例中的基类名）必须与派生类定义在一个作用域内。除了类，还可以用表达式，基类定义在另一个模块中时这一点非常有用：

```
class DerivedClassName(modname.BaseClassName):
```

实例(Python 3.0+)

```
#!/usr/bin/python3 #类定义 class people: #定义基本属性 name = " age = 0 #定义私有属性,私有属性在类外部无法直接进行访问
__weight = 0 #定义构造方法 def __init__(self,n,a,w): self.name = n self.age = a self.__weight = w def speak(self): print("%s 说: 我
%d 岁。" %(self.name,self.age)) #单继承示例 class student(people): grade = " def __init__(self,n,a,w,g): #调用父类的构函
people.__init__(self,n,a,w) self.grade = g #覆写父类的方法 def speak(self): print("%s 说: 我 %d 岁了，我在读 %d 年
级" %(self.name,self.age,self.grade)) s = student('ken',10,60,3) s.speak()
```

执行以上程序输出结果为：

```
ken 说：我 10 岁了，我在读 3 年级
```

多继承

Python 同样有限的支持多继承形式。多继承的类定义形如下例：

```
class DerivedClassName(Base1, Base2, Base3): <statement-1> ... <statement-N>
```

需要注意圆括号中父类的顺序，若是父类中有相同的方法名，而在子类使用时未指定，python 从左至右搜索 即方法在子类中未找到时，从左到右查找父类中是否包含方法。

实例(Python 3.0+)

```
#!/usr/bin/python3 #类定义 class people: #定义基本属性 name = " age = 0 #定义私有属性,私有属性在类外部无法直接进行访问
__weight = 0 #定义构造方法 def __init__(self,n,a,w): self.name = n self.age = a self.__weight = w def speak(self): print("%s 说: 我
%d 岁。" %(self.name,self.age)) #单继承示例 class student(people): grade = " def __init__(self,n,a,w,g): #调用父类的构函
people.__init__(self,n,a,w) self.grade = g #覆写父类的方法 def speak(self): print("%s 说: 我 %d 岁了，我在读 %d 年
级" %(self.name,self.age,self.grade)) #另一个类，多重继承之前的准备 class speaker(): topic = " name = " def __init__(self,n,t):
self.name = n self.topic = t def speak(self): print("我叫 %s，我是一个演说家，我演讲的主题是 %s" %(self.name,self.topic)) #多重
继承 class sample(speaker,student): a = " def __init__(self,n,a,w,g,t): student.__init__(self,n,a,w,g) speaker.__init__(self,n,t) test =
sample("Tim",25,80,4,"Python") test.speak() #方法名同，默认调用的是在括号中排前地父类的方法
```

执行以上程序输出结果为：

```
我叫 Tim，我是一个演说家，我演讲的主题是 Python
```

方法重写

如果你的父类方法的功能不能满足你的需求，你可以在子类重写你父类的方法，实例如下：

实例(Python 3.0+)

```
#!/usr/bin/python3 class Parent: # 定义父类 def myMethod(self): print ('调用父类方法') class Child(Parent): # 定义子类 def myMethod(self): print ('调用子类方法') c = Child() # 子类实例 c.myMethod() # 子类调用重写方法 super(Child,c).myMethod() # 用子类对象调用父类已被覆盖的方法
```

[super\(\)](#) 函数是用于调用父类(超类)的一个方法。

执行以上程序输出结果为：

调用子类方法
调用父类方法

类属性与方法

类的私有属性

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

类的方法

在类地内部，使用 `def` 关键字来定义一个方法，与一般函数定义不同，类方法必须包含参数 `self`，且为第一个参数，`self` 代表的是类的实例。

`self` 的名字并不是规定死的，也可以使用 `this`，但是最好还是按照约定是用 `self`。

类的私有方法

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

实例

类的私有属性实例如下：

实例(Python 3.0+)

```
#!/usr/bin/python3 class JustCounter: __secretCount = 0 # 私有变量 publicCount = 0 # 公开变量 def count(self): self.__secretCount += 1 self.publicCount += 1 print (self.__secretCount) counter = JustCounter() counter.count() counter.count() print (counter.publicCount) print (counter.__secretCount) # 报错，实例不能访问私有变量
```

执行以上程序输出结果为：

```
1
2
2
Traceback (most recent call last):
  File "test.py", line 16, in <module>
    print (counter.__secretCount) # 报错，实例不能访问私有变量
AttributeError: 'JustCounter' object has no attribute '__secretCount'
```

类的私有方法实例如下：

实例(Python 3.0+)

```
#!/usr/bin/python3 class Site: def __init__(self, name, url): self.name = name # public self.__url = url # private def who(self): print('name :', self.name) print('url:', self.__url) def __foo(self): # 私有方法 print('这是私有方法') def foo(self): # 公共方法 print('这是公共方法') self.__foo() x = Site('菜鸟教程', 'www.runoob.com') x.who() # 正常输出 x.foo() # 正常输出 x.__foo() # 报错
```

以上实例执行结果：

```
root@iZ23mtq8bs1Z:~/test# python3 test.py
```

```
name : 菜鸟教程
```

```
url : www.runoob.com
```

```
这是公共方法
```

```
这是私有方法
```

外部不能调用私有方法

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

```
File "test.py", line 22, in <module>
```

```
x.__foo()
```

```
AttributeError: 'Site' object has no attribute '__foo'
```

类的专有方法:

- `__init__`: 构造函数, 在生成对象时调用
- `__del__`: 析构函数, 释放对象时使用
- `__repr__`: 打印, 转换
- `__setitem__`: 按照索引赋值
- `__getitem__`: 按照索引获取值
- `__len__`: 获得长度
- `__cmp__`: 比较运算
- `__call__`: 函数调用
- `__add__`: 加运算
- `__sub__`: 减运算
- `__mul__`: 乘运算
- `__div__`: 除运算
- `__mod__`: 求余运算
- `__pow__`: 乘方

运算符重载

Python同样支持运算符重载, 我们可以对类的专有方法进行重载, 实例如下:

实例(Python 3.0+)

```
#!/usr/bin/python3 class Vector: def __init__(self, a, b): self.a = a self.b = b def __str__(self): return 'Vector (%d, %d)' % (self.a, self.b) def __add__(self, other): return Vector(self.a + other.a, self.b + other.b) v1 = Vector(2,10) v2 = Vector(5,-2) print (v1 + v2)
```

以上代码执行结果如下所示:

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


Python3 标准库概览

操作系统接口

os模块提供了不少与操作系统相关联的函数。

```
>>> import os
>>> os.getcwd()          # 返回当前的工作目录
'C:\Python34'
>>> os.chdir('/server/accesslogs')  # 修改当前的工作目录
>>> os.system('mkdir today')  # 执行系统命令 mkdir
0
```

建议使用 "import os" 风格而非 "from os import *". 这样可以保证随操作系统不同而有所变化的 os.open() 不会覆盖内置函数 open()。

在使用 os 这样的大型模块时内置的 dir() 和 help() 函数非常有用:

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

针对日常的文件和目录管理任务, mod.shutil 模块提供了一个易于使用的高级接口:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
>>> shutil.move('/build/executables', 'installdir')
```

文件通配符

glob模块提供了一个函数用于从目录通配符搜索中生成文件列表:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

命令行参数

通用工具脚本经常调用命令行参数。这些命令行参数以链表形式存储于 sys 模块的 argv 变量。例如在命令行中执行 "python demo.py one two three" 后可以得到以下输出结果:

```
>>> import sys
>>> print(sys.argv)
['demo.py', 'one', 'two', 'three']
```

错误输出重定向和程序终止

sys 还有 stdin, stdout 和 stderr 属性, 即使在 stdout 被重定向时, 后者也可以用于显示警告和错误信息。

```
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

大多脚本的定向终止都使用 "sys.exit()"。

字符串正则匹配

re模块为高级字符串处理提供了正则表达式工具。对于复杂的匹配和处理，正则表达式提供了简洁、优化的解决方案：

```
>>> import re
>>> re.findall(r'\b[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

如果只需要简单的功能，应该首先考虑字符串方法，因为它们非常简单，易于阅读和调试：

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

数学

math模块为浮点运算提供了对底层C函数库的访问：

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

random提供了生成随机数的工具。

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10)    # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()                  # random float
0.17970987693706186
>>> random.randrange(6)              # random integer chosen from range(6)
4
```

访问 互联网

有几个模块用于访问互联网以及处理网络通信协议。其中最简单的两个是用于处理从 urls 接收的数据的 urllib.request 以及用于发送电子邮件的 smtplib。

```
>>> from urllib.request import urlopen
>>> for line in urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):
...     line = line.decode('utf-8')    # Decoding the binary data to text.
...     if 'EST' in line or 'EDT' in line: # look for Eastern Time
...         print(line)

<BR>Nov. 25, 09:43:32 PM EST

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

注意第二个例子需要本地有一个在运行的邮件服务器。

日期和时间

datetime模块为日期和时间处理同时提供了简单和复杂的方法。

支持日期和时间算法的同时，实现的重点放在更有效的处理和格式化输出。

该模块还支持时区处理：

```
>>> # dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

>>> # dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

数据压缩

以下模块直接支持通用的数据打包和压缩格式：`zlib`，`gzip`，`bz2`，`zipfile`，以及 `tarfile`。

```
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

性能度量

有些用户对了解解决同一问题的不同方法之间的性能差异很感兴趣。Python 提供了一个度量工具，为这些问题提供了直接答案。

例如，使用元组封装和拆封来交换元素看起来要比使用传统的方法要诱人的多，`timeit` 证明了现代的方法更快一些。

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

相对于 `timeit` 的细粒度，`modprofile` 和 `pstats` 模块提供了针对更大代码块的时间度量工具。

测试模块

开发高质量软件的方法之一是为每一个函数开发测试代码，并且在开发过程中经常进行测试

`doctest` 模块提供了一个工具，扫描模块并根据程序中内嵌的文档字符串执行测试。

测试构造如同简单的将它的输出结果剪切并粘贴到文档字符串中。

通过用户提供的例子，它强化了文档，允许 `doctest` 模块确认代码的结果是否与文档一致：

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.
```

```
>>> print(average([20, 30, 70]))
40.0
"""
return sum(values) / len(values)
```

```
import doctest
doctest.testmod()    # 自动验证嵌入测试
```

unittest模块不像 **doctest**模块那么容易使用，不过它可以在一个独立的文件里提供一个更全面的测试集：

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        self.assertRaises(ZeroDivisionError, average, [])
        self.assertRaises(TypeError, average, 20, 30, 70)

unittest.main() # Calling from the command line invokes all tests
```

Python3 实例

以下实例在 Python3.4.3 版本下测试通过：

- [Python Hello World 实例](#)
- [Python 数字求和](#)
- [Python 平方根](#)
- [Python 二次方程](#)
- [Python 计算三角形的面积](#)
- [Python 随机数生成](#)
- [Python 摄氏温度转华氏温度](#)
- [Python 交换变量](#)
- [Python if 语句](#)
- [Python 判断字符串是否为数字](#)
- [Python 判断奇数偶数](#)
- [Python 判断闰年](#)
- [Python 获取最大值函数](#)
- [Python 质数判断](#)
- [Python 输出指定范围内的素数](#)
- [Python 阶乘实例](#)
- [Python 九九乘法表](#)
- [Python 斐波那契数列](#)
- [Python 阿姆斯特朗数](#)
- [Python 十进制转二进制、八进制、十六进制](#)
- [Python ASCII码与字符相互转换](#)
- [Python 最大公约数算法](#)
- [Python 最小公倍数算法](#)
- [Python 简单计算器实现](#)
- [Python 生成日历](#)
- [Python 使用递归斐波那契数列](#)
- [Python 文件 IO](#)
- [Python 字符串判断](#)
- [Python 字符串大小写转换](#)
- [Python 计算每个月天数](#)
- [Python 获取昨天日期](#)
- [Python list 常用操作](#)

Python3 正则表达式

正则表达式是一个特殊的字符序列，它能帮助你方便的检查一个字符串是否与某种模式匹配。

Python 自1.5版本起增加了re 模块，它提供 Perl 风格的正则表达式模式。

re 模块使 Python 语言拥有全部的正则表达式功能。

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

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

本章节主要介绍Python中常用的正则表达式处理函数。

re.match函数

re.match 尝试从字符串的起始位置匹配一个模式，如果不是起始位置匹配成功的话，match()就返回none。

函数语法：

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

函数参数说明：

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

匹配成功re.match方法返回一个匹配的对象，否则返回None。

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

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

实例

```
#!/usr/bin/python import re print(re.match('www', 'www.runoob.com').span()) # 在起始位置匹配 print(re.match('com', 'www.runoob.com')) # 不在起始位置匹配
```

以上实例运行输出结果为：

```
(0, 3)
None
```

实例

```
#!/usr/bin/python3 import re line = "Cats are smarter than dogs" matchObj = re.match( r'(.*) are (.*) .*', line, re.M|re.I) if matchObj: print ("matchObj.group() :", matchObj.group()) print ("matchObj.group(1) :", matchObj.group(1)) print ("matchObj.group(2) :", matchObj.group(2)) else: print ("No match!!")
```

以上实例执行结果如下：

```
matchObj.group() : Cats are smarter than dogs
matchObj.group(1) : Cats
matchObj.group(2) : smarter
```

re.search方法

re.search 扫描整个字符串并返回第一个成功的匹配。

函数语法：

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

函数参数说明：

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

匹配成功re.search方法返回一个匹配的对象，否则返回None。

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

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

实例

```
#!/usr/bin/python3 import re print(re.search('www', 'www.runoob.com').span()) # 在起始位置匹配 print(re.search('com', 'www.runoob.com').span()) # 不在起始位置匹配
```

以上实例运行输出结果为：

```
(0, 3)
(11, 14)
```

实例

```
#!/usr/bin/python3 import re line = "Cats are smarter than dogs"; searchObj = re.search( r'(.*) are (.*?) .*', line, re.M|re.I) if searchObj: print ("searchObj.group() :", searchObj.group()) print ("searchObj.group(1) :", searchObj.group(1)) print ("searchObj.group(2) :", searchObj.group(2)) else: print ("Nothing found!!")
```

以上实例执行结果如下：

```
searchObj.group() : Cats are smarter than dogs
searchObj.group(1) : Cats
searchObj.group(2) : smarter
```

re.match与re.search的区别

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

实例

```
#!/usr/bin/python3 import re line = "Cats are smarter than dogs"; matchObj = re.match( r'dogs', line, re.M|re.I) if matchObj: print ("match --> matchObj.group() :", matchObj.group()) else: print ("No match!!") matchObj = re.search( r'dogs', line, re.M|re.I) if matchObj: print ("search --> matchObj.group() :", matchObj.group()) else: print ("No match!!")
```

以上实例运行结果如下：

```
No match!!
search --> matchObj.group() : dogs
```

检索和替换

Python 的re模块提供了re.sub用于替换字符串中的匹配项。

语法：

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

参数：

- pattern: 正则中的模式字符串。
- repl: 替换的字符串，也可为一个函数。
- string: 要被查找替换的原始字符串。
- count: 模式匹配后替换的最大次数，默认 0 表示替换所有的匹配。

实例

```
#!/usr/bin/python3 import re phone = "2004-959-559 # 这是一个电话号码" # 删除注释 num = re.sub(r'#[^#]*$', "", phone) print ("电话号码 :", num) # 移除非数字的内容 num = re.sub(r'\D', "", phone) print ("电话号码 :", num)
```

以上实例执行结果如下：

```
电话号码 : 2004-959-559
电话号码 : 2004959559
```

repl 参数是一个函数

以下实例中将字符串中的匹配的数字乘以 2：

实例

```
#!/usr/bin/python import re # 将匹配的数字乘以 2 def double(matched): value = int(matched.group('value')) return str(value * 2) s = 'A23G4HFD567' print(re.sub('(P<value>\d+)', double, s))
```

执行输出结果为：

```
A46G8HFD1134
```

compile 函数

compile 函数用于编译正则表达式，生成一个正则表达式（Pattern）对象，供 match() 和 search() 这两个函数使用。

语法格式为：

```
re.compile(pattern[, flags])
```

参数：

- pattern: 一个字符串形式的正则表达式

- flags 可选，表示匹配模式，比如忽略大小写，多行模式等，具体参数为：
- re.I 忽略大小写
- re.L 表示特殊字符集 \w, \W, \b, \B, \s, \S 依赖于当前环境
- re.M 多行模式
- re.S 即为 '.' 并且包括换行符在内的任意字符（'.' 不包括换行符）
- re.U 表示特殊字符集 \w, \W, \b, \B, \d, \D, \s, \S 依赖于 Unicode 字符属性数据库
- re.X 为了增加可读性，忽略空格和 '#' 后面的注释

实例

实例

```
>>>import re>>> pattern=re.compile(r'd+')#用于匹配至少一个数字>>> m=pattern.match('one12twothree34four')#查找头部，没有匹配>>> print mNone>>> m=pattern.match('one12twothree34four', 2, 10)#从'e'的位置开始匹配，没有匹配>>> print mNone>>> m=pattern.match('one12twothree34four', 3, 10)#从'l'的位置开始匹配，正好匹配>>> print m#返回一个Match对象<_sre.SRE_Match object at 0x10a42aac0>>> m.group(0)#可省略0'12'>>> m.start(0)#可省略03>>> m.end(0)#可省略05>>> m.span(0)#可省略0(3, 5)
```

在上面，当匹配成功时返回一个 Match 对象，其中：

- group([group1, ...]) 方法用于获得一个或多个分组匹配的字符串，当要获得整个匹配的子串时，可直接使用 group() 或 group(0)；
- start([group]) 方法用于获取分组匹配的子串在整个字符串中的起始位置（子串第一个字符的索引），参数默认值为 0；
- end([group]) 方法用于获取分组匹配的子串在整个字符串中的结束位置（子串最后一个字符的索引+1），参数默认值为 0；
- span([group]) 方法返回 (start(group), end(group))。

再看看一个例子：

实例

```
>>>import re>>> pattern=re.compile(r'([a-z]+)([A-Z]+)', re.I)#re.I表示忽略大小写>>> m=pattern.match('Hello World Wide Web')>>> print m#匹配成功，返回一个Match对象<_sre.SRE_Match object at 0x10bea83e8>>> m.group(0)#返回匹配成功的整个子串'Hello World'>>> m.span(0)#返回匹配成功的整个子串的索引(0, 11)>>> m.group(1)#返回第一个分组匹配成功的子串'Hello'>>> m.span(1)#返回第一个分组匹配成功的子串的索引(0, 5)>>> m.group(2)#返回第二个分组匹配成功的子串'World'>>> m.span(2)#返回第二个分组匹配成功的子串的索引(6, 11)>>> m.groups()#等价于(m.group(1), m.group(2), ...)('Hello', 'World')>>> m.group(3)#不存在第三个分组Traceback (most recent call last): File "<stdin>", line 1, in <module> IndexError: no such group
```

findall

在字符串中找到正则表达式所匹配的所有子串，并返回一个列表，如果没有找到匹配的，则返回空列表。

注意：match 和 search 是匹配一次 findall 匹配所有。

语法格式为：

```
findall(string[, pos[, endpos]])
```

参数：

- string 待匹配的字符串。
- pos 可选参数，指定字符串的起始位置，默认为 0。
- endpos 可选参数，指定字符串的结束位置，默认为字符串的长度。

查找字符串中的所有数字：

实例

```
import re
pattern = re.compile(r'\d+') # 查找数字
result1 = pattern.findall('runoob 123 google 456')
result2 = pattern.findall('run88oob123google456', 0, 10)
print(result1)
print(result2)
```

输出结果：

```
['123', '456']
['88', '12']
```

re.finditer

和 findall 类似，在字符串中找到正则表达式所匹配的所有子串，并把它们作为一个迭代器返回。

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

参数：

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

实例

```
import re
it = re.finditer(r"\d+", "12a32bc43jf3")
for match in it:
    print(match.group())
```

输出结果：

```
12
32
43
3
```

re.split

split 方法按照能够匹配的子串将字符串分割后返回列表，它的使用形式如下：

```
re.split(pattern, string[, maxsplit=0, flags=0])
```

参数：

参数	描述
pattern	匹配的正则表达式
string	要匹配的字符串。
maxsplit	分隔次数，maxsplit=1 分隔一次，默认为 0，不限制次数。
flags	标志位，用于控制正则表达式的匹配方式，如：是否区分大小写，多行匹配等等。参见： 正则表达式修饰符 - 可选标志

实例

```
>>>import re
>>> re.split('\W+', 'runoob, runoob, runoob,')
['runoob', 'runoob', 'runoob', '']
>>> re.split('\W+', 'runoob, runoob, runoob,')
['', '', 'runoob', '', 'runoob', '', 'runoob', '', '']
>>> re.split('\W+', 'runoob, runoob, runoob, 1')
['', 'runoob, runoob, runoob,']
>>> re.split('a*', 'hello world')
# 对于一个找不到匹配的字符串而言，split 不会对其作出分割
['hello world']
```

正则表达式对象

re.RegexObject

re.compile() 返回 RegexObject 对象。

re.MatchObject

group() 返回被 RE 匹配的字符串。

- start() 返回匹配开始的位置
- end() 返回匹配结束的位置
- span() 返回一个元组包含匹配 (开始,结束) 的位置

正则表达式修饰符 - 可选标志

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

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

正则表达式模式

模式字符串使用特殊的语法来表示一个正则表达式：

字母和数字表示他们自身。一个正则表达式模式中的字母和数字匹配同样的字符串。

多数字母和数字前加一个反斜杠时会拥有不同的含义。

标点符号只有被转义时才匹配自身，否则它们表示特殊的含义。

反斜杠本身需要使用反斜杠转义。

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

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

模式	描述
^	匹配字符串的开头
\$	匹配字符串的末尾。
.	匹配任意字符，除了换行符，当re.DOTALL标记被指定时，则可以匹配包括换行符的任意字符。
[...]	用来表示一组字符,单独列出：[amk] 匹配 'a', 'm'或'k'
[^...]	不在[]中的字符：[^abc] 匹配除了a,b,c之外的字符。
re*	匹配0个或多个的表达式。
re+	匹配1个或多个的表达式。

<code>re?</code>	匹配0个或1个由前面的正则表达式定义的片段，非贪婪方式
<code>re{ n}</code>	匹配n个前面表达式。例如， <code>"o{2}"</code> 不能匹配"Bob"中的"o"，但是能匹配"food"中的两个o。
<code>re{ n, }</code>	精确匹配n个前面表达式。例如， <code>"o{2,}"</code> 不能匹配"Bob"中的"o"，但能匹配"fooooo"中的所有o。 <code>"o{1,}"</code> 等价于 <code>"o+"</code> 。 <code>"o{0,}"</code> 则等价于 <code>"o*"</code> 。
<code>re{ n, m}</code>	匹配 n 到 m 次由前面的正则表达式定义的片段，贪婪方式
<code>a b</code>	匹配a或b
<code>(re)</code>	G匹配括号内的表达式，也表示一个组
<code>(?imx)</code>	正则表达式包含三种可选标志： <code>i</code> , <code>m</code> 或 <code>x</code> 。只影响括号中的区域。
<code>(?-imx)</code>	正则表达式关闭 <code>i</code> , <code>m</code> 或 <code>x</code> 可选标志。只影响括号中的区域。
<code>(?: re)</code>	类似 (...), 但是不表示一个组
<code>(?imx: re)</code>	在括号中使用 <i>i</i> , <i>m</i> 或 <i>x</i> 可选标志
<code>(?-imx: re)</code>	在括号中不使用 <i>i</i> , <i>m</i> 或 <i>x</i> 可选标志
<code>(?#...)</code>	注释。
<code>(?= re)</code>	前向肯定界定符。如果所含正则表达式，以 ... 表示，在当前位置成功匹配时成功，否则失败。但一旦所含表达式已经尝试，匹配引擎根本没有提高；模式的剩余部分还要尝试界定符的右边。
<code>(?! re)</code>	前向否定界定符。与肯定界定符相反；当所含表达式不能在字符串当前位置匹配时成功。
<code>(?> re)</code>	匹配的独立模式，省去回溯。
<code>\w</code>	匹配数字字母下划线
<code>\W</code>	匹配非数字字母下划线
<code>\s</code>	匹配任意空白字符，等价于 <code>[\t\n\r\f]</code> 。
<code>\S</code>	匹配任意非空字符
<code>\d</code>	匹配任意数字，等价于 <code>[0-9]</code> 。
<code>\D</code>	匹配任意非数字
<code>\A</code>	匹配字符串开始
<code>\Z</code>	匹配字符串结束，如果是存在换行，只匹配到换行前的结束字符串。
<code>\z</code>	匹配字符串结束
<code>\G</code>	匹配最后匹配完成的位置。
<code>\b</code>	匹配一个单词边界，也就是指单词和空格间的位置。例如， <code>'er\b'</code> 可以匹配"never"中的'er'，但不能匹配 <code>"verb"</code> 中的'er'。
<code>\B</code>	匹配非单词边界。 <code>'er\B'</code> 能匹配 <code>"verb"</code> 中的'er'，但不能匹配 <code>"never"</code> 中的'er'。
<code>\n, \t, 等。</code>	匹配一个换行符。匹配一个制表符, 等
<code>\1...\9</code>	匹配第n个分组的内容。
<code>\10</code>	匹配第n个分组的内容，如果它经匹配。否则指的是八进制字符码的表达式。

正则表达式实例

字符匹配

实例	描述
<code>python</code>	匹配 "python".

字符类

实例	描述
<code>[Pp]ython</code>	匹配 "Python" 或 "python"
<code>rub[ye]</code>	匹配 "ruby" 或 "rube"

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

特殊字符类

实例	描述
.	匹配除 "\n" 之外的任何单个字符。要匹配包括 "\n" 在内的任何字符，请使用象 "[\n]" 的模式。
\d	匹配一个数字字符。等价于 [0-9]。
\D	匹配一个非数字字符。等价于 [^0-9]。
\s	匹配任何空白字符，包括空格、制表符、换页符等等。等价于 [\f\n\r\t\v]。
\S	匹配任何非空白字符。等价于 [^\f\n\r\t\v]。
\w	匹配包括下划线的任何单词字符。等价于 "[A-Za-z0-9_]"。
\W	匹配任何非单词字符。等价于 "[^A-Za-z0-9_]"。

Python CGI编程

什么是CGI

CGI 目前由NCSA维护, NCSA定义CGI如下:

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

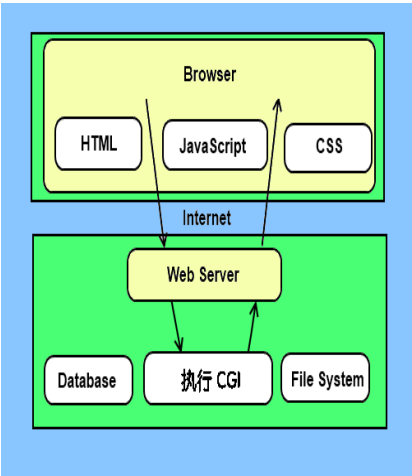
网页浏览

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

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

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

CGI架构图



Web服务器支持及配置

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

Apache 支持CGI 配置:

设置好CGI目录:

```
ScriptAlias /cgi-bin/ /var/www/cgi-bin/
```

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

CGI文件的扩展名为.cgi, python也可以使用.py扩展名。

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

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

```
<Directory "/var/www/cgi-bin">
    AllowOverride None
    Options +ExecCGI
    Order allow,deny
    Allow from all
</Directory>
```

在 AddHandler 中添加 .py 后缀, 这样我们就可以访问 .py 结尾的 python 脚本文件:

```
AddHandler cgi-script .cgi .pl .py
```

第一个CGI程序

我们使用Python创建第一个CGI程序, 文件名为hello.py, 文件位于/var/www/cgi-bin目录中, 内容如下:

```
#!/usr/bin/python3

print ("Content-type:text/html")
print () # 空行, 告诉服务器结束头部
print ('<html>')
print ('<head>')
print ('<meta charset="utf-8">')
print ('<title>Hello Word - 我的第一个 CGI 程序! </title>')
print ('</head>')
print ('<body>')
print ('<h2>Hello Word! 我是来自菜鸟教程的第一CGI程序</h2>')
print ('</body>')
print ('</html>')
```

文件保存后修改 `hello.py`，修改文件权限为 755：

```
chmod 755 hello.py
```

以上程序在浏览器访问显示结果如下：



Hello Word!

我是来自菜鸟教程的第一CGI程序。

这个的hello.py脚本是一个简单的Python脚本，脚本第一行的输出内容"Content-type:text/html"发送到浏览器并告知浏览器显示的内容类型为"text/html"。

用 `print` 输出一个空行用于告诉服务器结束头部信息。

HTTP头部

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

HTTP头部的格式如下：

HTTP 字段名： 字段内容

例如：

```
Content-type: text/html
```

以下表格介绍了CGI程序中HTTP头部经常使用的信息：

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

CGI环境变量

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

变量名	描述
CONTENT_TYPE	这个环境变量的值指示所传递来的信息的MIME类型。目前，环境变量CONTENT_TYPE一般都是：application/x-www-form-urlencoded，他表示数据来自于HTML表单。
CONTENT_LENGTH	如果服务器与CGI程序信息的传递方式是POST，这个环境变量即使从标准输入STDIN中可以读到的有效数据的字节数。这个环境变量在读取所输入的数据时必须使用。
HTTP_COOKIE	客户机内的 COOKIE 内容。
HTTP_USER_AGENT	提供包含了版本号或其他专有数据的客户浏览器信息。
PATH_INFO	这个环境变量的值表示紧接在CGI程序名之后的其他路径信息。它常常作为CGI程序的参数出现。
QUERY_STRING	如果服务器与CGI程序信息的传递方式是GET，这个环境变量的值即使所传递的信息。这个信息紧跟在CGI程序名的后面，两者中间用一个问号"?"分隔。
REMOTE_ADDR	这个环境变量的值是发送请求的客户机的IP地址，例如上面的192.168.1.67。这个值总是存在的。而且它是Web客户机需要提供给Web服务器的唯一标识，可以在CGI程序中用它来区分不同的Web客户机。
REMOTE_HOST	这个环境变量的值包含发送CGI请求的客户机的主机名。如果不支持你想查询，则无需定义此环境变量。
REQUEST_METHOD	提供脚本被调用的方法。对于使用 HTTP/1.0 协议的脚本，仅 GET 和 POST 有意义。
SCRIPT_FILENAME	CGI脚本的完整路径
SCRIPT_NAME	CGI脚本的名称
SERVER_NAME	这是你的 WEB 服务器的主机名、别名或IP地址。
SERVER_SOFTWARE	这个环境变量的值包含了调用CGI程序的HTTP服务器的名称和版本号。例如，上面的值为Apache/2.2.14(Unix)

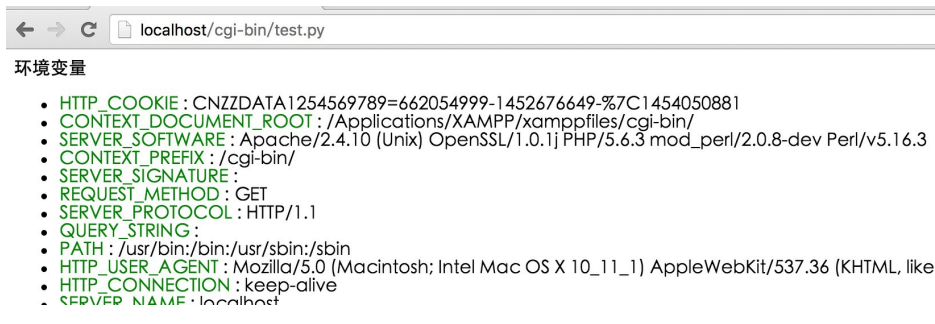
以下是一个简单的CGI脚本输出CGI的环境变量：

```
#!/usr/bin/python3

import os

print ("Content-type: text/html")
print ()
print ("<meta charset='utf-8'>")
print ("<b>环境变量</b><br>")
print ("<ul>")
for key in os.environ.keys():
    print ("<li><span style='color:green'>%30s </span> : %s </li>" % (key,os.environ[key]))
print ("</ul>")
```

将以上点保存为 test.py,并修改文件权限为 755, 执行结果如下:



GET和POST方法

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

使用GET方法传输数据

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

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

有关 GET 请求的其他一些注释:

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

简单的url实例: GET方法

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

```
/cgi-bin/test.py?name=菜鸟教程&url=http://www.runoob.com
```

以下为hello_get.py文件的代码:

```
#!/usr/bin/python3

# CGI处理模块
import cgi, cgitb

# 创建 FieldStorage 的实例化
form = cgi.FieldStorage()

# 获取数据
site_name = form.getvalue('name')
site_url = form.getvalue('url')

print ("Content-type:text/html")
print ()
print (<html>)
print (<head>)
print (<meta charset="utf-8">)
print (<title>菜鸟教程 CGI 测试实例</title>)
print (</head>)
print (<body>)
print (<h2>官网: %s</h2> % (site_name, site_url))
print (</body>)
print (</html>)
```

文件保存后修改 hello_get.py, 修改文件权限为 755:

```
chmod 755 hello_get.py
```

浏览器请求输出结果:



菜鸟教程官网: http://www.runoob.com

简单的表单实例: GET方法

以下是一个通过HTML的表单使用GET方法向服务器发送两个数据, 提交的服务器脚本同样是hello_get.py文件, hello_get.html代码如下:

```
<!DOCTYPE html>
```



```

<html>
<head>
<meta charset="utf-8">
<title>菜鸟教程(runoob.com)</title>
</head>
<body>
<form action="/cgi-bin/hello_get.py" method="get">
站点名称: <input type="text" name="name"> <br />

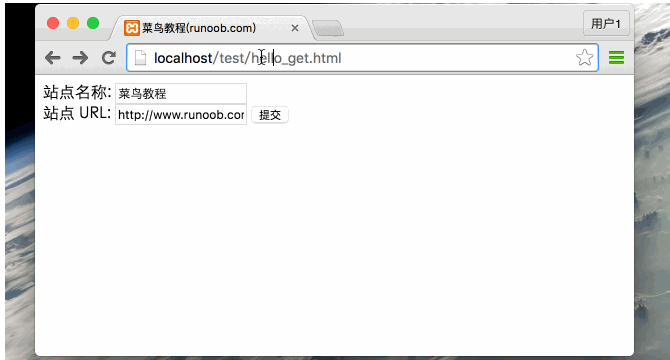
站点 URL: <input type="text" name="url" />
<input type="submit" value="提交" />
</form>
</body>
</html>

```

默认情况下 cgi-bin 目录只能存放脚本文件，我们将 `hello_get.html` 存储在 `test` 目录下，修改文件权限为 755：

```
chmod 755 hello_get.html
```

Gif 演示如下所示：



使用POST方法传递数据

使用POST方法向服务器传递数据是更安全可靠，像一些敏感信息如用户密码等需要使用POST传输数据。

以下同样是 `hello_get.py`，它也可以处理浏览器提交的POST表单数据：

```

#!/usr/bin/python3

# CGI处理模块
import cgi, cgitb

# 创建 FieldStorage 的实例化
form = cgi.FieldStorage()

# 获取数据
site_name = form.getvalue('name')
site_url = form.getvalue('url')

print ("Content-type:text/html")
print ()
print (<html>)
print (<head>)
print (<meta charset=\<utf-8\>)
print (<title>菜鸟教程 CGI 测试实例</title>)
print (</head>)
print (<body>)
print (<h2>官方网站: %s</h2> % (site_name, site_url))
print (</body>)
print (</html>)

```

以下为表单通过POST方法（`method="post"`）向服务器脚本 `hello_get.py` 提交数据：

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>菜鸟教程(runoob.com)</title>
</head>
<body>
<form action="/cgi-bin/hello_get.py" method="post">
站点名称: <input type="text" name="name"> <br />

站点 URL: <input type="text" name="url" />
<input type="submit" value="提交" />
</form>
</body>
</html>
</form>

```

Gif 演示如下所示：



通过CGI程序传递checkbox数据

checkbox用于提交一个或者多个选项数据，HTML代码如下：

```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>菜鸟教程 (runoob.com)</title>
</head>
<body>
<form action="/cgi-bin/checkbox.py" method="POST" target="_blank">
<input type="checkbox" name="runoob" value="on" /> 菜鸟教程
<input type="checkbox" name="google" value="on" /> Google
<input type="submit" value="选择站点" />
</form>
</body>
</html>
```

以下为 checkbox.py 文件的代码：

```
#!/usr/bin/python3

# 引入 CGI 处理模块
import cgi, cgitb

# 创建 FieldStorage的实例
form = cgi.FieldStorage()

# 接收字段数据
if form.getvalue('google'):
    google_flag = "是"
else:
    google_flag = "否"

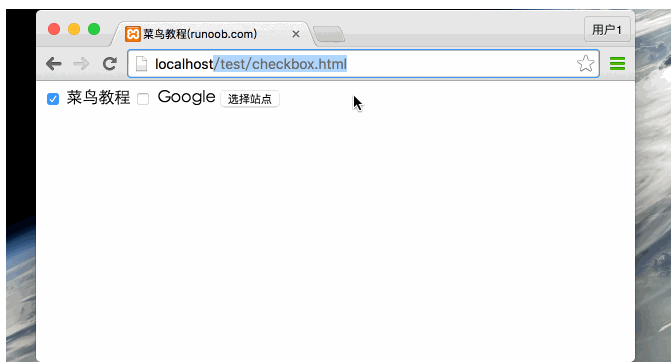
if form.getvalue('runoob'):
    runoob_flag = "是"
else:
    runoob_flag = "否"

print ("Content-type:text/html")
print ()
print (<html>)
print (<head>)
print (<meta charset=\"utf-8\">)
print (<title>菜鸟教程 CGI 测试实例</title>)
print (</head>)
print (<body>)
print (<h2> 菜鸟教程是否选择了 : %s</h2>" % runoob_flag)
print (<h2> Google 是否选择了 : %s</h2>" % google_flag)
print (</body>)
print (</html>)
```

修改 checkbox.py 权限：

```
chmod 755 checkbox.py
```

浏览器访问 Gif 演示图：



通过CGI程序传递Radio数据

Radio 只向服务器传递一个数据，HTML代码如下：

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>菜鸟教程(runoob.com)</title>
</head>
<body>
<form action="/cgi-bin/radiobutton.py" method="post" target="_blank">
<input type="radio" name="site" value="runoob" /> 菜鸟教程
<input type="radio" name="site" value="google" /> Google
<input type="submit" value="提交" />
</form>
</body>
</html>

```

radiobutton.py 脚本代码如下：

```

#!/usr/bin/python3

# 引入 CGI 处理模块
import cgi, cgitb

# 创建 FieldStorage的实例
form = cgi.FieldStorage()

# 接收字段数据
if form.getvalue('site'):
    site = form.getvalue('site')
else:
    site = "提交数据为空"

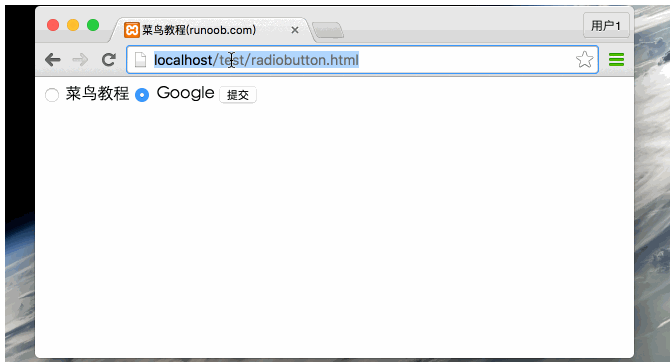
print ("Content-type:text/html")
print ()
print ("<html>")
print ("<head>")
print ("<meta charset=\"utf-8\">")
print ("<title>菜鸟教程 CGI 测试实例</title>")
print ("</head>")
print ("<body>")
print ("<h2> 选中的网站是 %s</h2>" % site)
print ("</body>")
print ("</html>")

```

修改 radiobutton.py 权限：

```
chmod 755 radiobutton.py
```

浏览器访问 Gif 演示图：



通过CGI程序传递 Textarea 数据

Textarea 向服务器传递多行数据，HTML代码如下：

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>菜鸟教程(runoob.com)</title>
</head>
<body>
<form action="/cgi-bin/textarea.py" method="post" target="_blank">
<textarea name="textcontent" cols="40" rows="4">
在这里输入内容...
</textarea>
<input type="submit" value="提交" />
</form>
</body>
</html>

```

textarea.py 脚本代码如下：

```

#!/usr/bin/python3

# 引入 CGI 处理模块
import cgi, cgitb

# 创建 FieldStorage的实例
form = cgi.FieldStorage()

# 接收字段数据
if form.getvalue('textcontent'):
    text_content = form.getvalue('textcontent')
else:
    text_content = "没有内容"

print ("Content-type:text/html")
print ()
print ("<html>")

```

```

print ("<head>")
print ("<meta charset=\"utf-8\">")
print ("<title>菜鸟教程 CGI 测试实例</title>")
print ("</head>")
print ("<body>")
print ("<h2> 输入的内容是: %s</h2>" % text_content)
print ("</body>")
print ("</html>")

```

修改 textarea.py 权限:

```
chmod 755 textarea.py
```

浏览器访问 Gif 演示图:



通过CGI程序传递下拉数据。

HTML 下拉框代码如下:

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>菜鸟教程 (runoob.com)</title>
</head>
<body>
<form action="/cgi-bin/dropdown.py" method="post" target="_blank">
<select name="dropdown">
<option value="runoob" selected>菜鸟教程</option>
<option value="google">Google</option>
</select>
<input type="submit" value="提交"/>
</form>
</body>
</html>

```

dropdown.py 脚本代码如下所示:

```

#!/usr/bin/python3

# 引入 CGI 处理模块
import cgi, cgitb

# 创建 FieldStorage的实例
form = cgi.FieldStorage()

# 接收字段数据
if form.getvalue('dropdown'):
    dropdown_value = form.getvalue('dropdown')
else:
    dropdown_value = "没有内容"

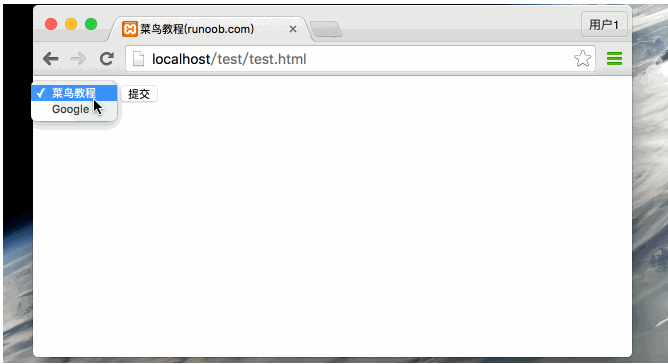
print ("Content-type:text/html")
print ()
print ("<html>")
print ("<head>")
print ("<meta charset=\"utf-8\">")
print ("<title>菜鸟教程 CGI 测试实例</title>")
print ("</head>")
print ("<body>")
print ("<h2> 选中的选项是: %s</h2>" % dropdown_value)
print ("</body>")
print ("</html>")

```

修改 dropdown.py 权限:

```
chmod 755 dropdown.py
```

浏览器访问 Gif 演示图:



CGI中使用Cookie

在 http 协议一个很大的缺点就是不对用户身份的的判断, 这样给编程人员带来很大的不便, 而 cookie 功能的出现弥补了这个不足。

cookie 就是在客户访问脚本的同时, 通过客户的浏览器, 在客户硬盘上写入纪录数据, 当下次客户访问脚本时取回数据信息, 从而达到身份判别的功能, cookie 常用在身份校验中。

cookie的语法

http cookie的发送是通过http头部来实现的, 他早于文件的传递, 头部set-cookie的语法如下:

```
Set-cookie:name=name;expires=date;path=path;domain=domain;secure
```

- **name=name**: 需要设置cookie的值(name不能使用";"和","号), 有多个name值时用 ";" 分隔, 例如: **name1=name1;name2=name2;name3=name3**。
- **expires=date**: cookie的有效期限, 格式: **expires="Wdy,DD-Mon-YYYY HHMM:SS"**
- **path=path**: 设置cookie支持的路径, 如果path是一个路径, 则cookie对这个目录下的所有文件及子目录生效, 例如: **path="/cgi-bin"**, 如果path是一个文件, 则cookie指对这个文件生效, 例如: **path="/cgi-bin/cookie.cgi"**。
- **domain=domain**: 对cookie生效的域名, 例如: **domain="www.runoob.com"**
- **secure**: 如果给出此标志, 表示cookie只能通过SSL协议的https服务器来传递。
- cookie的接收是通过设置环境变量HTTP_COOKIE来实现的, CGI程序可以通过检索该变量获取cookie信息。

Cookie设置

Cookie的设置非常简单, cookie会在http头部单独发送。以下实例在cookie中设置了name 和 expires:

```
#!/usr/bin/python3
#
print ('Content-Type: text/html')
print ('Set-Cookie: name="菜鸟教程";expires=Wed, 28 Aug 2016 18:30:00 GMT')
print ()
print ("")
<html>
<head>
<meta charset="utf-8">
<title>菜鸟教程(runoob.com)</title>
</head>
<body>
<h1>Cookie set OK!</h1>
</body>
</html>
"""
)
```

将以上代码保存至 cookie_set.py, 并修改 cookie_set.py 权限:

```
chmod 755 cookie_set.py
```

以上实例使用了 Set-Cookie 头信息来设置Cookie信息, 可选项中设置了Cookie的其他属性, 如过期时间Expires, 域名Domain, 路径Path。这些信息设置在 "Content-type:text/html"之前。

检索Cookie信息

Cookie信息检索页非常简单, Cookie信息存储在CGI的环境变量HTTP_COOKIE中, 存储格式如下:

```
key1=value1;key2=value2;key3=value3....
```

以下是一个简单的CGI检索cookie信息的程序:

```
#!/usr/bin/python3
# 导入模块
import os
import http.cookies

print ("Content-type: text/html")
print ()

print ("")
<html>
<head>
<meta charset="utf-8">
<title>菜鸟教程(runoob.com)</title>
</head>
<body>
<h1>读取cookie信息</h1>
"""
)

if 'HTTP_COOKIE' in os.environ:
```

```

cookie_string=os.environ.get('HTTP_COOKIE')
c=Cookie.SimpleCookie()
c.load(cookie_string)

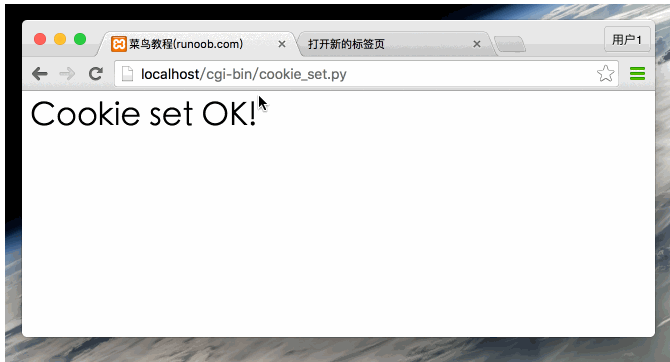
try:
    data=c['name'].value
    print ("cookie data: "+data+"<br>")
except KeyError:
    print ("cookie 没有设置或者已过去<br>")
print ("""
</body>
</html>
""")

```

将以上代码保存到 cookie_get.py, 并修改 cookie_get.py 权限:

```
chmod 755 cookie_get.py
```

以上 cookie 设置颜色 Gif 如下所示:



文件上传实例

HTML设置上传文件的表单需要设置 **enctype** 属性为 **multipart/form-data**, 代码如下所示:

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>菜鸟教程 (runoob.com)</title>
</head>
<body>
<form enctype="multipart/form-data"
      action="/cgi-bin/save_file.py" method="post">
  <p>选中文件: <input type="file" name="filename" /></p>
  <p><input type="submit" value="上传" /></p>
</form>
</body>
</html>

```

save_file.py脚本文件代码如下:

```

#!/usr/bin/python3

import cgi, os
import cgitb; cgitb.enable()

form = cgi.FieldStorage()

# 获取文件名
fileitem = form['filename']

# 检测文件是否上传
if fileitem.filename:
    # 设置文件路径
    fn = os.path.basename(fileitem.filename)
    open('/tmp/' + fn, 'wb').write(fileitem.file.read())

    message = '文件 "' + fn + '" 上传成功'
else:
    message = '文件没有上传'

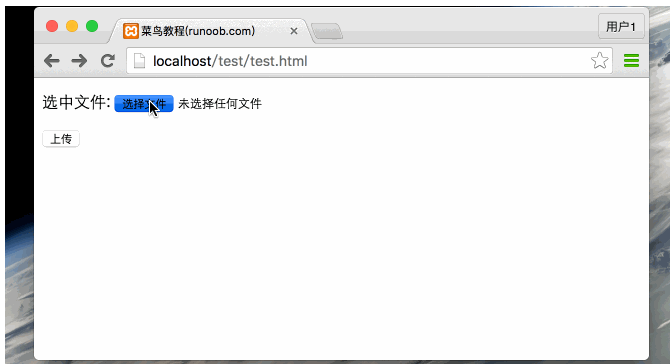
print ("""
Content-Type: text/html\n
<html>
<head>
<meta charset="utf-8">
<title>菜鸟教程 (runoob.com)</title>
</head>
<body>
<p>%s</p>
</body>
</html>
""" % (message,))

```

将以上代码保存到 save_file.py, 并修改 save_file.py 权限:

```
chmod 755 save_file.py
```

以上 cookie 设置颜色 Gif 如下所示:



如果你使用的系统是Unix/Linux，你必须替换文件分隔符，在window下只需要使用open()语句即可：

```
fn = os.path.basename(fileitem.filename.replace("\\", "/" ))
```

文件下载对话框

我们先在当前目录下创建 foo.txt 文件，用于程序的下载。

文件下载通过设置HTTP头信息来实现，功能代码如下：

```
#!/usr/bin/python3

# HTTP 头部
print ("Content-Disposition: attachment; filename=\"foo.txt\"")
print ()
# 打开文件
fo = open("foo.txt", "rb")

str = fo.read();
print (str)

# 关闭文件
fo.close()
```

Python3 MySQL 数据库连接

本文我们为大家介绍 Python3 使用 [PyMySQL](#) 连接数据库，并实现简单的增删改查。

什么是 PyMySQL?

PyMySQL 是在 Python3.x 版本中用于连接 MySQL 服务器的一个库，Python2中则使用mysqldb。

PyMySQL 遵循 Python 数据库 API v2.0 规范，并包含了 pure-Python MySQL 客户端库。

PyMySQL 安装

在使用 PyMySQL 之前，我们需要确保 PyMySQL 已安装。

PyMySQL 下载地址：<https://github.com/PyMySQL/PyMySQL>。

如果还未安装，我们可以使用以下命令安装最新版的 PyMySQL：

```
$ pip install PyMySQL
```

如果你的系统不支持 pip 命令，可以使用以下方式安装：

1、使用 git 命令下载安装包安装(你也可以手动下载)：

```
$ git clone https://github.com/PyMySQL/PyMySQL
$ cd PyMySQL/
$ python3 setup.py install
```

2、如果需要制定版本号，可以使用 curl 命令来安装：

```
$ # X.X 为 PyMySQL 的版本号
$ curl -L https://github.com/PyMySQL/PyMySQL/tarball/pymysql-X.X | tar xz
$ cd PyMySQL*
$ python3 setup.py install
$ # 现在你可以删除 PyMySQL* 目录
```

注意：请确保您有root权限来安装上述模块。

安装的过程中可能会出现"ImportError: No module named setuptools"的错误提示，意思是您没有安装setuptools，您可以访问<https://pypi.python.org/pypi/setuptools> 找到各个系统的安装方法。

Linux 系统安装实例：

```
$ wget https://bootstrap.pypa.io/ez_setup.py
$ python3 ez_setup.py
```

数据库连接

连接数据库前，请先确认以下事项：

- 您已经创建了数据库 TESTDB.
- 在TESTDB数据库中您已经创建了表 EMPLOYEE
- EMPLOYEE表字段为 FIRST_NAME, LAST_NAME, AGE, SEX 和 INCOME。
- 连接数据库TESTDB使用的用户名为 "testuser"，密码为 "test123",您可以自己设定或者直接使用root用户名及其密码，Mysql数据库用户授权请使用Grant命令。
- 在你的机子上已经安装了 Python MySQLdb 模块。
- 如果您对sql语句不熟悉，可以访问我们的 [SQL基础教程](#)

实例：

以下实例链接 Mysql 的 TESTDB 数据库：

实例(Python 3.0+)

```
#!/usr/bin/python3 import pymysql # 打开数据库连接 db = pymysql.connect("localhost","testuser","test123","TESTDB") # 使用
cursor() 方法创建一个游标对象 cursor cursor = db.cursor() # 使用 execute() 方法执行 SQL 查询 cursor.execute("SELECT
VERSION()") # 使用 fetchone() 方法获取单条数据. data = cursor.fetchone() print ("Database version : %s "% data) # 关闭数据库
连接 db.close()
```

执行以上脚本输出结果如下：

```
Database version : 5.5.20-log
```

创建数据库表

如果数据库连接存在我们可以使用execute()方法来为数据库创建表，如下所示创建表EMPLOYEE:

实例(Python 3.0+)

```
#!/usr/bin/python3 import pymysql # 打开数据库连接 db = pymysql.connect("localhost","testuser","test123","TESTDB") # 使用
cursor() 方法创建一个游标对象 cursor cursor = db.cursor() # 使用 execute() 方法执行 SQL，如果表存在则删除
cursor.execute("DROP TABLE IF EXISTS EMPLOYEE") # 使用预处理语句创建表 sql = """CREATE TABLE EMPLOYEE (
FIRST_NAME CHAR(20) NOT NULL, LAST_NAME CHAR(20), AGE INT, SEX CHAR(1), INCOME FLOAT)"""
cursor.execute(sql) # 关闭数据库连接 db.close()
```

数据库插入操作

以下实例使用执行 SQL INSERT 语句向表 EMPLOYEE 插入记录：

实例(Python 3.0+)

```
#!/usr/bin/python3 import pymysql # 打开数据库连接 db = pymysql.connect("localhost","testuser","test123","TESTDB") # 使用
cursor()方法获取操作游标 cursor = db.cursor() # SQL 插入语句 sql = "INSERT INTO EMPLOYEE(FIRST_NAME,
LAST_NAME, AGE, SEX, INCOME) VALUES ('Mac', 'Mohan', 20, 'M', 2000)"" try: # 执行sql语句 cursor.execute(sql) # 提交到
数据库执行 db.commit() except: # 如果发生错误则回滚 db.rollback() # 关闭数据库连接 db.close()
```

以上例子也可以写成如下形式：

实例(Python 3.0+)

```
#!/usr/bin/python3 import pymysql # 打开数据库连接 db = pymysql.connect("localhost","testuser","test123","TESTDB") # 使用
cursor()方法获取操作游标 cursor = db.cursor() # SQL 插入语句 sql = 'INSERT INTO EMPLOYEE(FIRST_NAME, \
LAST_NAME, AGE, SEX, INCOME) \ VALUES ("%s", "%s", "%d", "%c", "%d")' % \ ('Mac', 'Mohan', 20, 'M', 2000) try: # 执行sql
语句 cursor.execute(sql) # 执行sql语句 db.commit() except: # 发生错误时回滚 db.rollback() # 关闭数据库连接 db.close()
```

以下代码使用变量向SQL语句中传递参数：

```
.....
user_id = "test123"
password = "password"

con.execute('insert into Login values ("%s", "%s")' % \
            (user_id, password))
.....
```

数据库查询操作

Python查询Mysql使用 `fetchone()` 方法获取单条数据, 使用`fetchall()` 方法获取多条数据。

- **fetchone():** 该方法获取下一个查询结果集。结果集是一个对象
- **fetchall():** 接收全部的返回结果行。
- **rowcount:** 这是一个只读属性，并返回执行`execute()`方法后影响的行数。

实例：

查询EMPLOYEE表中salary（工资）字段大于1000的所有数据：

实例(Python 3.0+)

```
#!/usr/bin/python3 import pymysql # 打开数据库连接 db = pymysql.connect("localhost","testuser","test123","TESTDB") # 使用
cursor()方法获取操作游标 cursor = db.cursor() # SQL 查询语句 sql = "SELECT * FROM EMPLOYEE \ WHERE INCOME >
'%d'" % (1000) try: # 执行SQL语句 cursor.execute(sql) # 获取所有记录列表 results = cursor.fetchall() for row in results: fname =
row[0] lname = row[1] age = row[2] sex = row[3] income = row[4] # 打印结果 print
("fname=%s,lname=%s,age=%d,sex=%s,income=%d" % (fname, lname, age, sex, income )) except: print ("Error: unable to fetch data")
# 关闭数据库连接 db.close()
```

以上脚本执行结果如下：

```
fname=Mac, lname=Mohan, age=20, sex=M, income=2000
```

数据库更新操作

更新操作用于更新数据表的数据，以下实例将 TESTDB表中的 SEX 字段全部修改为 'M'，AGE 字段递增1：

实例(Python 3.0+)

```
#!/usr/bin/python3 import pymysql # 打开数据库连接 db = pymysql.connect("localhost","testuser","test123","TESTDB") # 使用
cursor()方法获取操作游标 cursor = db.cursor() # SQL 更新语句 sql = "UPDATE EMPLOYEE SET AGE = AGE + 1 WHERE
SEX = '%c'" % ('M') try: # 执行SQL语句 cursor.execute(sql) # 提交到数据库执行 db.commit() except: # 发生错误时回滚
db.rollback() # 关闭数据库连接 db.close()
```

删除操作

删除操作用于删除数据表中的数据，以下实例演示了删除数据表 EMPLOYEE 中 AGE 大于 20 的所有数据：

实例(Python 3.0+)

```
#!/usr/bin/python3 import pymysql # 打开数据库连接 db = pymysql.connect("localhost","testuser","test123","TESTDB") # 使用
cursor()方法获取操作游标 cursor = db.cursor() # SQL 删除语句 sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" %
(20) try: # 执行SQL语句 cursor.execute(sql) # 提交修改 db.commit() except: # 发生错误时回滚 db.rollback() # 关闭连接
db.close()
```

执行事务

事务机制可以确保数据一致性。

事务应该具有4个属性：原子性、一致性、隔离性、持久性。这四个属性通常称为ACID特性。

- 原子性（atomicity）。一个事务是一个不可分割的工作单位，事务中包括的诸操作要么都做，要么都不做。

- 一致性（consistency）。事务必须是使数据库从一个一致性状态变到另一个一致性状态。一致性与原子性是密切相关的。
- 隔离性（isolation）。一个事务的执行不能被其他事务干扰。即一个事务内部的操作及使用的数据对并发的其他事务是隔离的，并发执行的各个事务之间不能互相干扰。
- 持久性（durability）。持续性也称永久性（permanence），指一个事务一旦提交，它对数据库中数据的改变就应该是永久性的。接下来的其他操作或故障不应该对其有任何影响。

Python DB API 2.0 的事务提供了两个方法 `commit` 或 `rollback`。

实例

实例(Python 3.0+)

```
# SQL删除记录语句 sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (20) try: # 执行SQL语句 cursor.execute(sql)
# 向数据库提交 db.commit() except: # 发生错误时回滚 db.rollback()
```

对于支持事务的数据库，在Python数据库编程中，当游标建立之时，就自动开始了一个隐形的数据库事务。

`commit()`方法游标的所有更新操作，`rollback()`方法回滚当前游标的所有操作。每一个方法都开始了一个新的事务。

错误处理

DB API中定义了一些数据库操作的错误及异常，下表列出了这些错误和异常：

异常	描述
Warning	当有严重警告时触发，例如插入数据是被截断等等。必须是 <code>StandardError</code> 的子类。
Error	警告以外所有其他错误类。必须是 <code>StandardError</code> 的子类。
InterfaceError	当有数据库接口模块本身的错误（而不是数据库的错误）发生时触发。必须是 <code>Error</code> 的子类。
DatabaseError	和数据库有关的错误发生时触发。必须是 <code>Error</code> 的子类。
DataError	当有数据处理时的错误发生时触发，例如：除零错误，数据超范围等等。必须是 <code>DatabaseError</code> 的子类。
OperationalError	指非用户控制的，而是操作数据库时发生的错误。例如：连接意外断开、数据库名未找到、事务处理失败、内存分配错误等等操作数据库是发生的错误。必须是 <code>DatabaseError</code> 的子类。
IntegrityError	完整性相关的错误，例如外键检查失败等。必须是 <code>DatabaseError</code> 子类。
InternalError	数据库的内部错误，例如游标（cursor）失效了、事务同步失败等等。必须是 <code>DatabaseError</code> 子类。
ProgrammingError	程序错误，例如数据表（table）没找到或已存在、SQL语句语法错误、参数数量错误等等。必须是 <code>DatabaseError</code> 的子类。
NotSupportedError	不支持错误，指使用了数据库不支持的函数或API等。例如在连接对象上使用 <code>.rollback()</code> 函数，然而数据库并不支持事务或者事务已关闭。必须是 <code>DatabaseError</code> 的子类。

Python3 网络编程

Python 提供了两个级别访问的网络服务。:

- 低级别的网络服务支持基本的 Socket，它提供了标准的 BSD Sockets API，可以访问底层操作系统Socket接口的全部方法。
- 高级别的网络服务模块 SocketServer，它提供了服务器中心类，可以简化网络服务器的开发。

什么是 Socket?

Socket又称"套接字"，应用程序通常通过"套接字"向网络发出请求或者应答网络请求，使主机间或者一台计算机上的进程间可以通讯。

socket()函数

Python 中，我们用 socket（）函数来创建套接字，语法格式如下：

```
socket.socket([family[, type[, proto]]])
```

参数

- family: 套接字家族可以使AF_UNIX或者AF_INET
- type: 套接字类型可以根据是面向连接的还是非连接分为SOCK_STREAM或SOCK_DGRAM
- protocol: 一般不填默认为0.

Socket 对象(内建)方法

函数	描述
服务器端套接字	
s.bind()	绑定地址（host,port）到套接字，在AF_INET下,以元组（host,port）的形式表示地址。
s.listen()	开始TCP监听。backlog指定在拒绝连接之前，操作系统可以挂起的最大连接数量。该值至少为1，大部分应用程序设为5就可以了。
s.accept()	被动接受TCP客户端连接,(阻塞式)等待连接的到来
客户端套接字	
s.connect()	主动初始化TCP服务器连接，。一般address的格式为元组（hostname,port），如果连接出错，返回socket.error错误。
s.connect_ex()	connect()函数的扩展版本,出错时返回出错码,而不是抛出异常
公共用途的套接字函数	
s.recv()	接收TCP数据，数据以字符串形式返回，bufsize指定要接收的最大数据量。flag提供有关消息的其他信息，通常可以忽略。
s.send()	发送TCP数据，将string中的数据发送到连接的套接字。返回值是要发送的字节数量，该数量可能小于string的字节大小。
s.sendall()	完整发送TCP数据，完整发送TCP数据。将string中的数据发送到连接的套接字，但在返回之前会尝试发送所有数据。成功返回None，失败则抛出异常。
s.recvfrom()	接收UDP数据，与recv()类似，但返回值是（data,address）。其中data是包含接收数据的字符串，address是发送数据的套接字地址。
s.sendto()	发送UDP数据，将数据发送到套接字，address是形式为（ipaddr, port）的元组，指定远程地址。返回值是发送的字节数。
s.close()	关闭套接字
s.getpeername()	返回连接套接字的远程地址。返回值通常是元组（ipaddr,port）。

s.getsockname(函数)	返回套接字自己的地址。通常是一个元组(描述 addr,port)
s.setsockopt(level,optname,value)	设置给定套接字选项的值。
s.getsockopt(level,optname[,buflen])	返回套接字选项的值。
s.settimeout(timeout)	设置套接字操作的超时期，timeout是一个浮点数，单位是秒。值为None表示没有超时期。一般，超时期应该在刚创建套接字时设置，因为它们可能用于连接的操作（如connect()）
s.gettimeout()	返回当前超时期的值，单位是秒，如果没有设置超时期，则返回None。
s.fileno()	返回套接字的文件描述符。
s.setblocking(flag)	如果flag为0，则将套接字设为非阻塞模式，否则将套接字设为阻塞模式（默认值）。非阻塞模式下，如果调用recv()没有发现任何数据，或send()调用无法立即发送数据，那么将引起socket.error异常。
s.makefile()	创建一个与该套接字相关连的文件

简单实例

服务端

我们使用 socket 模块的 **socket** 函数来创建一个 socket 对象。socket 对象可以通过调用其他函数来设置一个 socket 服务。

现在我们可以通过调用 **bind(hostname, port)** 函数来指定服务的 *port*(端口)。

接着，我们调用 socket 对象的 *accept* 方法。该方法等待客户端的连接，并返回 *connection* 对象，表示已连接到客户端。

完整代码如下：

```
#!/usr/bin/python3
# 文件名: server.py

# 导入 socket、sys 模块
import socket
import sys

# 创建 socket 对象
serversocket = socket.socket(
    socket.AF_INET, socket.SOCK_STREAM)

# 获取本地主机名
host = socket.gethostname()

port = 9999

# 绑定端口号
serversocket.bind((host, port))

# 设置最大连接数，超过后排队
serversocket.listen(5)

while True:
    # 建立客户端连接
    clientsocket, addr = serversocket.accept()

    print("连接地址: %s" % str(addr))

    msg='欢迎访问菜鸟教程!' + "\r\n"
    clientsocket.send(msg.encode('utf-8'))
    clientsocket.close()
```

客户端

接下来我们写一个简单的客户端实例连接到以上创建的服务。端口号为 9999。

socket.connect(hostname, port) 方法打开一个 TCP 连接到主机为 *hostname* 端口为 *port* 的服务商。连接后我们就可以从服务端后期数据，记住，操作完成后需要关闭连接。

完整代码如下：

```
#!/usr/bin/python3
# 文件名: client.py

# 导入 socket、sys 模块
import socket
import sys

# 创建 socket 对象
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# 获取本地主机名
host = socket.gethostname()

# 设置端口号
port = 9999

# 连接服务，指定主机和端口
s.connect((host, port))

# 接收小于 1024 字节的数据
msg = s.recv(1024)

s.close()

print (msg.decode('utf-8'))
```

现在我们打开两个终端，第一个终端执行 server.py 文件：

```
$ python3 server.py
```

第二个终端执行 client.py 文件：

```
$ python3 client.py
欢迎访问菜鸟教程！
```

这是我们再打开第一个终端，就会看到有以下信息输出：

```
连接地址: ('192.168.0.118', 33397)
```

Python Internet 模块

以下列出了 Python 网络编程的一些重要模块：

协议	功能用处	端口号	Python 模块
HTTP	网页访问	80	httplib, urllib, xmlrpclib
NNTP	阅读和张贴新闻文章，俗称为"帖子"	119	nntplib
FTP	文件传输	20	ftplib, urllib
SMTP	发送邮件	25	smtplib
POP3	接收邮件	110	poplib
IMAP4	获取邮件	143	imaplib
Telnet	命令行	23	telnetlib
Gopher	信息查找	70	gopherlib, urllib

更多内容可以参阅官网的 [Python Socket Library and Modules](#)。

Python3 SMTP发送邮件

SMTP（Simple Mail Transfer Protocol）即简单邮件传输协议,它是一组用于由源地址到目的地址传送邮件的规则，由它来控制信件的中转方式。

python的smtplib提供了一种很方便的途径发送电子邮件。它对smtp协议进行了简单的封装。

Python创建 SMTP 对象语法如下：

```
import smtplib

smtpObj = smtplib.SMTP([host [, port [, local_hostname]]])
```

参数说明：

- host: SMTP 服务器主机。你可以指定主机的ip地址或者域名如:runoob.com，这个是可选参数。
- port: 如果你提供了 host 参数,你需要指定 SMTP 服务使用的端口号，一般情况下SMTP端口号为25。
- local_hostname: 如果SMTP在你的本机上，你只需要指定服务器地址为 localhost 即可。

Python SMTP对象使用sendmail方法发送邮件，语法如下：

```
SMTP.sendmail(from_addr, to_addrs, msg[, mail_options, rcpt_options])
```

参数说明：

- from_addr: 邮件发送者地址。
- to_addrs: 字符串列表，邮件发送地址。
- msg: 发送消息

这里要注意一下第三个参数，msg是字符串，表示邮件。我们知道邮件一般由标题，发信人，收件人，邮件内容，附件等构成，发送邮件的时候，要注意msg的格式。这个格式就是smtp协议中定义的格式。

实例

以下是一个使用Python发送邮件简单的实例：

实例

```
#!/usr/bin/python3 import smtplib from email.mime.text import MIMEText from email.header import Header sender = 'from@runoob.com' receivers = ['429240967@qq.com'] # 接收邮件，可设置为你的QQ邮箱或者其他邮箱 # 三个参数：第一个为文本内容，第二个 plain 设置文本格式，第三个 utf-8 设置编码 message = MIMEText('Python 邮件发送测试...', 'plain', 'utf-8') message['From'] = Header("菜鸟教程", 'utf-8') message['To'] = Header("测试", 'utf-8') subject = 'Python SMTP 邮件测试' message['Subject'] = Header(subject, 'utf-8') try: smtpObj = smtplib.SMTP('localhost') smtpObj.sendmail(sender, receivers, message.as_string()) print ("邮件发送成功") except smtplib.SMTPException: print ("Error: 无法发送邮件")
```

我们使用三个引号来设置邮件信息，标准邮件需要三个头部信息：**From**, **To**, 和 **Subject**，每个信息直接使用空行分割。

我们通过实例化 smtplib 模块的 SMTP 对象 *smtpObj* 来连接到 SMTP 访问，并使用 *sendmail* 方法来发送信息。

执行以上程序，如果你本机安装sendmail，就会输出：

```
$ python3 test.py
邮件发送成功
```

查看我们的收件箱(一般在垃圾箱)，就可以看到邮件信息：


Python SMTP 邮件测试 ☆

发件人： **菜鸟教程@iZ23mtq8bs1Z** <> 

(由 from@runoob.com 代发) ?

时 间：2016年4月5日(星期二) 下午4:30

收件人：测试@iZ23mtq8bs1Z

这是一封垃圾箱中的邮件。请勿轻信中奖、汇款等虚假信息，勿轻易拨打陌生电话。  举报垃圾邮件 移回收件箱

Python 邮件发送测试...

如果我们本机没有 sendmail 访问，也可以使用其他服务商的 SMTP 访问（QQ、网易、Google等）。

实例

```
#!/usr/bin/python3 import smtplib from email.mime.text import MIMEText from email.header import Header # 第三方 SMTP 服务 mail_host='smtp.XXX.com' #设置服务器 mail_user="XXXX" #用户名 mail_pass="XXXXXX" #口令 sender = 'from@runoob.com' receivers = ['429240967@qq.com'] # 接收邮件，可设置为你的QQ邮箱或者其他邮箱 message = MIMEText('Python 邮件发送测试...', 'plain', 'utf-8') message['From'] = Header("菜鸟教程", 'utf-8') message['To'] = Header("测试", 'utf-8') subject = 'Python SMTP 邮件测试' message['Subject'] = Header(subject, 'utf-8') try: smtpObj = smtplib.SMTP() smtpObj.connect(mail_host, 25) # 25 为 SMTP 端口号 smtpObj.login(mail_user, mail_pass) smtpObj.sendmail(sender, receivers, message.as_string()) print ("邮件发送成功") except smtplib.SMTPException: print ("Error: 无法发送邮件")
```

使用Python发送HTML格式的邮件

Python发送HTML格式的邮件与发送纯文本消息的邮件不同之处就是将MIMEText中_subtype设置为html。具体代码如下：

实例

```
#!/usr/bin/python3 import smtplib from email.mime.text import MIMEText from email.header import Header sender = 'from@runoob.com' receivers = ['429240967@qq.com'] # 接收邮件，可设置为你的QQ邮箱或
```




者其他邮箱 mail_msg = ""<p>Python 邮件发送测试...</p> <p>这是一个链接</p> "" message = MIMEText(mail_msg, 'html', 'utf-8') message['From'] = Header("菜鸟教程", 'utf-8') message['To'] = Header("测试", 'utf-8') subject = 'Python SMTP 邮件测试' message['Subject'] = Header(subject, 'utf-8') try: smtpObj = smtplib.SMTP('localhost') smtpObj.sendmail(sender, receivers, message.as_string()) print ("邮件发送成功") except smtplib.SMTPException: print ("Error: 无法发送邮件")

执行以上程序，如果你本机安装sendmail，就会输出：

```
$ python3 test.py
邮件发送成功
```


查看我们的收件箱(一般在垃圾箱)，就可以查看到邮件信息：

Python SMTP 邮件测试 ☆

发件人： **菜鸟教程@iZ23mtq8bs1Z** <> 
(由 from@runoob.com 代发) 

时 间：2016年4月5日(星期二) 下午4:45

收件人：测试@iZ23mtq8bs1Z

这是一封垃圾箱中的邮件。请勿轻信中奖、汇款等虚假信息，勿轻易拨打陌生电话。  举报垃圾邮件 移回收件箱

Python 邮件发送测试...

[这是一个链接](#)

Python 发送带附件的邮件

发送带附件的邮件，首先要创建MIMEmultipart()实例，然后构造附件，如果有多个附件，可依次构造，最后利用smtplib.smtp发送。

实例

```
#!/usr/bin/python3
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
from email.header import Header

sender = "from@runoob.com"
receivers = ["429240967@qq.com"]

# 接收邮件，可设置为你的QQ邮箱或者其他邮箱
# 创建一个带附件的实例
message = MIMEMultipart()
message['From'] = Header("菜鸟教程", 'utf-8')
message['To'] = Header("测试", 'utf-8')
subject = 'Python SMTP 邮件测试'
message['Subject'] = Header(subject, 'utf-8')

# 邮件正文内容
message.attach(MIMEText("这是菜鸟教程Python 邮件发送测试.....", 'plain', 'utf-8'))

# 构造附件1，传送当前目录下的test.txt文件
att1 = MIMEText(open("test.txt", 'rb').read(), 'base64', 'utf-8')
att1["Content-Type"] = 'application/octet-stream'
# 这里的filename可以任意写，写什么名字，邮件中显示什么名字
att1["Content-Disposition"] = 'attachment; filename="test.txt"'
message.attach(att1)

# 构造附件2，传送当前目录下的runoob.txt文件
att2 = MIMEText(open("runoob.txt", 'rb').read(), 'base64', 'utf-8')
att2["Content-Type"] = 'application/octet-stream'
att2["Content-Disposition"] = 'attachment; filename="runoob.txt"'
message.attach(att2)

try:
    smtpObj = smtplib.SMTP('localhost')
    smtpObj.sendmail(sender, receivers, message.as_string())
    print ("邮件发送成功")
except smtplib.SMTPException:
    print ("Error: 无法发送邮件")
```

```
$ python3 test.py
邮件发送成功
```

查看我们的收件箱(一般在垃圾箱)，就可以查看到邮件信息：

Python SMTP 邮件测试 ☆


发件人: 菜鸟教程@iZ23mtq8bs1Z <> 

(由 from@runoob.com 代发) 

时 间: 2016年4月5日(星期二) 下午5:27

收件人: 测试@iZ23mtq8bs1Z

附 件: 2 个 ( test.txt...)

垃圾邮件中的附件可能包含木马病毒等有害内容。为了您的帐号安全, 请勿轻易打开附件。  举报垃圾邮件 移回收件箱

这是菜鸟教程Python 邮件发送测试.....

附件(2 个)

普通附件  全部下载



test.txt (29字节)

[下载](#) [预览](#) [转存](#) ▼



runoob.txt (4字节)

[下载](#) [预览](#) [转存](#) ▼

在 HTML 文本中添加图片

邮件的 HTML 文本中一般邮件服务商添加外链是无效的, 正确添加突破的实例如下所示:

实例

```
#!/usr/bin/python3 import smtplib from email.mime.image import MIMEImage from email.mime.multipart import MIMEMultipart from email.mime.text import MIMEText from email.header import Header sender = 'from@runoob.com' receivers = ['429240967@qq.com'] # 接收邮件, 可设置为你的QQ邮箱或者其他邮箱 msgRoot = MIMEMultipart('related') msgRoot['From'] = Header("菜鸟教程", 'utf-8') msgRoot['To'] = Header("测试", 'utf-8') subject = 'Python SMTP 邮件测试' msgRoot['Subject'] = Header(subject, 'utf-8') msgAlternative = MIMEMultipart('alternative') msgRoot.attach(msgAlternative) mail_msg = "" <p>Python 邮件发送测试...</p><p><a href="http://www.runoob.com">菜鸟教程链接</a></p><p>图片演示: </p><p></p>"" msgAlternative.attach(MIMEText(mail_msg, 'html', 'utf-8')) # 指定图片为当前目录 fp = open('test.png', 'rb') msgImage = MIMEImage(fp.read()) fp.close() # 定义图片 ID, 在 HTML 文本中引用 msgImage.add_header('Content-ID', '<image1>') msgRoot.attach(msgImage) try: smtpObj = smtplib.SMTP('localhost') smtpObj.sendmail(sender, receivers, msgRoot.as_string()) print ("邮件发送成功") except smtplib.SMTPException: print ("Error: 无法发送邮件")
```

```
$ python3 test.py
邮件发送成功
```

查看我们的收件箱(如果在垃圾箱可能需要移动到收件箱才可正常显示), 就可以查看到邮件信息:



Python SMTP 邮件测试

菜鸟教程@iZ23mtq8bs1Z <> 详情

Python 邮件发送测试...

菜鸟教程链接

图片演示：

RUNOOB.COM

使用第三方 SMTP 服务发送

这里使用了 QQ 邮箱(你也可以使用 163, Gmail等)的 SMTP 服务, 需要做以下配置:

□

QQ 邮箱通过生成授权码来设置密码:

POP3/IMAP/SMTP/Exchange/CardDAV/CalDAV服务

开启服务：	POP3/SMTP服务 (如何使用 Foxmail 等软件收发邮件?)	已开启
	IMAP/SMTP服务 (什么是 IMAP, 它又是如何设置?)	已开启
	Exchange服务 (什么是Exchange, 它又是如何设置?)	已开启
	CardDAV/CalDAV服务 (什么是CardDAV/CalDAV, 它又是如何设置?)	已开启
	(POP3/IMAP/SMTP/CardDAV/CalDAV服务均支持SSL连接。如何设置?)	

生成

温馨提示：登录第三方客户端时，密码框请输入“授权码”进行验证?

生成授权码

QQ 邮箱 SMTP 服务器地址: smtp.qq.com, ssl 端口: 465。

以下实例你需要修改: 发件人邮箱 (你的QQ邮箱), 密码, 收件人邮箱 (可发给自己)。

QQ SMTP

```
#!/usr/bin/python3
import smtplib
from email.mime.text import MIMEText
from email.utils import formataddr

my_sender='429240967@qq.com' # 发件人邮箱账号
my_pass='xxxxxxx' # 发件人邮箱密码
my_user='429240967@qq.com' # 收件人邮箱账号, 我这边发送给自己

def mail():
    ret=True
    try:
        msg=MIMEText('填写邮件内容','plain','utf-8')
        msg['From']=formataddr(["FromRunoob",my_sender]) # 括号里的对应发件人邮箱昵称、发件人邮箱账号
        msg['To']=formataddr(["FK",my_user]) # 括号里的对应收件人邮箱昵称、收件人邮箱账号
        msg['Subject']="菜鸟教程发送邮件测试" # 邮件的主题, 也可以说是标题
        server=smtplib.SMTP_SSL("smtp.qq.com", 465) # 发件人邮箱中的SMTP服务器, 端口是25
        server.login(my_sender, my_pass) # 括号中对应的是发件人邮箱账号、邮箱密码
        server.sendmail(my_sender,[my_user,],msg.as_string()) # 括号中对应的是发件人邮箱账号、收件人邮箱账号、发送邮件
        server.quit() # 关闭连接
    except Exception: # 如果try中的语句没有执行, 则会执行下面的
        ret=False
    return ret

ret=mail()
if ret:
    print("邮件发送成功")
else:
    print("邮件发送失败")
```

```
$ python test.py
邮件发送成功
```

发送成功后, 登陆收件人邮箱即可查看:

□

更多内容请参阅: <https://docs.python.org/3/library/email-examples.html>.

Python3 多线程

多线程类似于同时执行多个不同程序，多线程运行有如下优点：

- 使用线程可以把占据长时间的程序中的任务放到后台去处理。
- 用户界面可以更加吸引人，这样比如用户点击了一个按钮去触发某些事件的处理，可以弹出一个进度条来显示处理的进度
- 程序的运行速度可能加快
- 在一些等待的任务实现上如用户输入、文件读写和网络收发数据等，线程就比较有用了。在这种情况下我们可以释放一些珍贵的资源如内存占用等等。

线程在执行过程中与进程还是有区别的。每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。

每个线程都有他自己的一组CPU寄存器，称为线程的上下文，该上下文反映了线程上次运行该线程的CPU寄存器的状态。

指令指针和堆栈指针寄存器是线程上下文中两个最重要的寄存器，线程总是在进程得到上下文中运行的，这些地址都用于标志拥有线程的进程地址空间中的内存。

- 线程可以被抢占（中断）。
- 在其他线程正在运行时，线程可以暂时搁置（也称为睡眠）-- 这就是线程的退让。

线程可以分为：

- **内核线程**：由操作系统内核创建和撤销。
- **用户线程**：不需要内核支持而在用户程序中实现的线程。

Python3 线程中常用的两个模块为：

- **_thread**
- **threading**(推荐使用)

thread 模块已被废弃。用户可以使用 threading 模块代替。所以，在 Python3 中不能再使用"thread"模块。为了兼容性，Python3 将 thread 重命名为 "_thread"。

开始学习Python线程

Python中使用线程有两种方式：函数或者用类来包装线程对象。

函数式：调用 _thread 模块中的start_new_thread()函数来产生新线程。语法如下：

```
_thread.start_new_thread ( function, args[, kwargs] )
```

参数说明：

- function - 线程函数。
- args - 传递给线程函数的参数,他必须是个tuple类型。
- kwargs - 可选参数。

实例：

```
#!/usr/bin/python3

import _thread
import time

# 为线程定义一个函数
```

```
def print_time( threadName, delay):
    count = 0
    while count < 5:
        time.sleep(delay)
        count += 1
        print ("%s: %s" % ( threadName, time.ctime(time.time()) ))

# 创建两个线程
try:
    _thread.start_new_thread( print_time, ("Thread-1", 2, ) )
    _thread.start_new_thread( print_time, ("Thread-2", 4, ) )
except:
    print ("Error: 无法启动线程")

while 1:
    pass
```

执行以上程序输出结果如下：

```
Thread-1: Wed Apr 6 11:36:31 2016
Thread-1: Wed Apr 6 11:36:33 2016
Thread-2: Wed Apr 6 11:36:33 2016
Thread-1: Wed Apr 6 11:36:35 2016
Thread-1: Wed Apr 6 11:36:37 2016
Thread-2: Wed Apr 6 11:36:37 2016
Thread-1: Wed Apr 6 11:36:39 2016
Thread-2: Wed Apr 6 11:36:41 2016
Thread-2: Wed Apr 6 11:36:45 2016
Thread-2: Wed Apr 6 11:36:49 2016
```

执行以上程后可以按下 ctrl-c to 退出。

线程模块

Python3 通过两个标准库 `_thread` 和 `threading` 提供对线程的支持。

`_thread` 提供了低级别的、原始的线程以及一个简单的锁，它相比于 `threading` 模块的功能还是比较有限的。

`threading` 模块除了包含 `_thread` 模块中的所有方法外，还提供的其他方法：

- `threading.currentThread()`: 返回当前的线程变量。
- `threading.enumerate()`: 返回一个包含正在运行的线程的list。正在运行指线程启动后、结束前，不包括启动前和终止后的线程。
- `threading.activeCount()`: 返回正在运行的线程数量，与`len(threading.enumerate())`有相同的结果。

除了使用方法外，线程模块同样提供了Thread类来处理线程，Thread类提供了以下方法：

- **run()**: 用以表示线程活动的方法。
- **start()**: 启动线程活动。
- **join([time])**: 等待至线程中止。这阻塞调用线程直至线程的join()方法被调用中止-正常退出或者抛出未处理的异常-或者是可选的超时发生。
- **isAlive()**: 返回线程是否活动的。
- **getName()**: 返回线程名。
- **setName()**: 设置线程名。

使用 threading 模块创建线程

我们可以通过直接从 `threading.Thread` 继承创建一个新的子类，并实例化后调用 `start()` 方法启动新线程，即它调用了线程的 `run()` 方法：

```
#!/usr/bin/python3
```

```

import threading
import time

exitFlag = 0

class myThread (threading.Thread):
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):
        print ("开始线程: " + self.name)
        print_time(self.name, self.counter, 5)
        print ("退出线程: " + self.name)

def print_time(threadName, delay, counter):
    while counter:
        if exitFlag:
            threadName.exit()
        time.sleep(delay)
        print ("%s: %s" % (threadName, time.ctime(time.time())))
        counter -= 1

# 创建新线程
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# 开启新线程
thread1.start()
thread2.start()
thread1.join()
thread2.join()
print ("退出主线程")

```

以上程序执行结果如下；

```

开始线程: Thread-1
开始线程: Thread-2
Thread-1: Wed Apr 6 11:46:46 2016
Thread-1: Wed Apr 6 11:46:47 2016
Thread-2: Wed Apr 6 11:46:47 2016
Thread-1: Wed Apr 6 11:46:48 2016
Thread-1: Wed Apr 6 11:46:49 2016
Thread-2: Wed Apr 6 11:46:49 2016
Thread-1: Wed Apr 6 11:46:50 2016
退出线程: Thread-1
Thread-2: Wed Apr 6 11:46:51 2016
Thread-2: Wed Apr 6 11:46:53 2016
Thread-2: Wed Apr 6 11:46:55 2016
退出线程: Thread-2
退出主线程

```

线程同步

如果多个线程共同对某个数据修改，则可能出现不可预料的结果，为了保证数据的正确性，需要对多个线程进行同步。

使用 Thread 对象的 Lock 和 Rlock 可以实现简单的线程同步，这两个对象都有 acquire 方法和 release 方法，对于那些需要每次只允许一个线程操作的数据，可以将其操作放到 acquire 和 release 方法之间。如下：

多线程的优势在于可以同时运行多个任务（至少感觉起来是这样）。但是当线程需要共享数据时，可能存在数据不同步的问题。

考虑这样一种情况：一个列表里所有元素都是0，线程"set"从后向前把所有元素改成1，而线程"print"负责从前往后读取列表并打印。

那么，可能线程"set"开始改的时候，线程"print"便来打印列表了，输出就成了一半0一半1，这就是数据的不同步。为了避免这种情况，引入了锁的概念。

锁有两种状态——锁定和未锁定。每当一个线程比如"set"要访问共享数据时，必须先获得锁定；如果已经有别的线程比如"print"获得锁定了，那么就on让线程"set"暂停，也就是同步阻塞；等到线程"print"访问完毕，释放锁以后，再让线程"set"继续。

经过这样的处理，打印列表时要么全部输出0，要么全部输出1，不会再出现一半0一半1的尴尬场面。

实例：

```
#!/usr/bin/python3

import threading
import time

class myThread (threading.Thread):
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):
        print ("开启线程: " + self.name)
        # 获取锁，用于线程同步
        threadLock.acquire()
        print time(self.name, self.counter, 3)
        # 释放锁，开启下一个线程
        threadLock.release()

def print_time(threadName, delay, counter):
    while counter:
        time.sleep(delay)
        print ("%s: %s" % (threadName, time.ctime(time.time())))
        counter -= 1

threadLock = threading.Lock()
threads = []

# 创建新线程
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# 开启新线程
thread1.start()
thread2.start()

# 添加线程到线程列表
threads.append(thread1)
threads.append(thread2)

# 等待所有线程完成
for t in threads:
    t.join()
print ("退出主线程")
```

执行以上程序，输出结果为：

```
开启线程: Thread-1
开启线程: Thread-2
Thread-1: Wed Apr 6 11:52:57 2016
Thread-1: Wed Apr 6 11:52:58 2016
Thread-1: Wed Apr 6 11:52:59 2016
Thread-2: Wed Apr 6 11:53:01 2016
Thread-2: Wed Apr 6 11:53:03 2016
Thread-2: Wed Apr 6 11:53:05 2016
退出主线程
```

线程优先级队列（Queue）

Python 的 Queue 模块中提供了同步的、线程安全的队列类，包括FIFO（先入先出）队列Queue，LIFO（后入先出）队列LifoQueue，和优先级队列 PriorityQueue。

这些队列都实现了锁原语，能够在多线程中直接使用，可以使用队列来实现线程间的同步。

Queue 模块中的常用方法:

- Queue.qsize() 返回队列的大小
- Queue.empty() 如果队列为空，返回True,反之False
- Queue.full() 如果队列满了，返回True,反之False
- Queue.full 与 maxsize 大小对应
- Queue.get([block[, timeout]])获取队列，timeout等待时间
- Queue.get_nowait() 相当Queue.get(False)
- Queue.put(item) 写入队列，timeout等待时间
- Queue.put_nowait(item) 相当Queue.put(item, False)
- Queue.task_done() 在完成一项工作之后，Queue.task_done()函数向任务已经完成的队列发送一个信号
- Queue.join() 实际上意味着等到队列为空，再执行别的操作

实例:

```
#!/usr/bin/python3

import queue
import threading
import time

exitFlag = 0

class myThread (threading.Thread):
    def __init__(self, threadID, name, q):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.q = q
    def run(self):
        print ("开启线程: " + self.name)
        process_data(self.name, self.q)
        print ("退出线程: " + self.name)

def process_data(threadName, q):
    while not exitFlag:
        queueLock.acquire()
        if not workQueue.empty():
            data = q.get()
            queueLock.release()
            print ("%s processing %s" % (threadName, data))
        else:
            queueLock.release()
            time.sleep(1)

threadList = ["Thread-1", "Thread-2", "Thread-3"]
nameList = ["One", "Two", "Three", "Four", "Five"]
queueLock = threading.Lock()
workQueue = queue.Queue(10)
threads = []
threadID = 1

# 创建新线程
for tName in threadList:
    thread = myThread(threadID, tName, workQueue)
    thread.start()
    threads.append(thread)
    threadID += 1
```



```
# 填充队列
queueLock.acquire()
for word in nameList:
    workQueue.put(word)
queueLock.release()

# 等待队列清空
while not workQueue.empty():
    pass

# 通知线程是时候退出
exitFlag = 1

# 等待所有线程完成
for t in threads:
    t.join()
print ("退出主线程")
```

以上程序执行结果:

```
开启线程: Thread-1
开启线程: Thread-2
开启线程: Thread-3
Thread-3 processing One
Thread-1 processing Two
Thread-2 processing Three
Thread-3 processing Four
Thread-1 processing Five
退出线程: Thread-3
退出线程: Thread-2
退出线程: Thread-1
退出主线程
```

Python3 XML解析

什么是XML?

XML 指可扩展标记语言（eXtensible Markup Language），标准通用标记语言的子集，是一种用于标记电子文件使其具有结构性的标记语言。你可以通过本站学习[XML教程](#)

XML 被设计用来传输和存储数据。

XML是一套定义语义标记的规则，这些标记将文档分成许多部件并对这些部件加以标识。

它也是元标记语言，即定义了用于定义其他与特定领域有关的、语义的、结构化的标记语言的句法语言。

python对XML的解析

常见的XML编程接口有DOM和SAX，这两种接口处理XML文件的方式不同，当然使用场合也不同。

python有三种方法解析XML，SAX，DOM，以及ElementTree:

1.SAX (simple API for XML)

python 标准库包含SAX解析器，SAX用事件驱动模型，通过在解析XML的过程中触发一个个的事件并调用用户定义的回调函数来处理XML文件。

2.DOM(Document Object Model)

将XML数据在内存中解析成一个树，通过对树的操作来操作XML。

本章节使用到的XML实例文件movies.xml内容如下：

```
<collection shelf="New Arrivals">
<movie title="Enemy Behind">
  <type>War, Thriller</type>
  <format>DVD</format>
  <year>2003</year>
  <rating>PG</rating>
  <stars>10</stars>
  <description>Talk about a US-Japan war</description>
</movie>
<movie title="Transformers">
  <type>Anime, Science Fiction</type>
  <format>DVD</format>
  <year>1989</year>
  <rating>R</rating>
  <stars>8</stars>
  <description>A schientific fiction</description>
</movie>
<movie title="Trigun">
  <type>Anime, Action</type>
  <format>DVD</format>
  <episodes>4</episodes>
  <rating>PG</rating>
  <stars>10</stars>
  <description>Vash the Stampede!</description>
</movie>
<movie title="Ishtar">
  <type>Comedy</type>
  <format>VHS</format>
  <rating>PG</rating>
  <stars>2</stars>
  <description>Viewable boredom</description>
```

```
</movie>
</collection>
```

python使用SAX解析xml

SAX是一种基于事件驱动的API。

利用SAX解析XML文档牵涉到两个部分:解析器和事件处理器。

解析器负责读取XML文档,并向事件处理器发送事件,如元素开始跟元素结束事件;

而事件处理器则负责对事件作出相应,对传递的XML数据进行处理。

- 1、对大型文件进行处理;
- 2、只需要文件的部分内容,或者只需从文件中得到特定信息。
- 3、想建立自己的对象模型的时候。

在python中使用sax方式处理xml要先引入xml.sax中的parse函数, 还有xml.sax.handler中的ContentHandler。

ContentHandler类方法介绍

characters(content)方法

调用时机:

从行开始, 遇到标签之前, 存在字符, content的值为这些字符串。

从一个标签, 遇到下一个标签之前, 存在字符, content的值为这些字符串。

从一个标签, 遇到行结束符之前, 存在字符, content的值为这些字符串。

标签可以是开始标签, 也可以是结束标签。

startDocument()方法

文档启动的时候调用。

endDocument()方法

解析器到达文档结尾时调用。

startElement(name, attrs)方法

遇到XML开始标签时调用, name是标签的名字, attrs是标签的属性值字典。

endElement(name)方法

遇到XML结束标签时调用。

make_parser方法

以下方法创建一个新的解析器对象并返回。

```
xml.sax.make_parser( [parser_list] )
```

参数说明:

- **parser_list** - 可选参数, 解析器列表
-

parser方法

以下方法创建一个 SAX 解析器并解析xml文档：

```
xml.sax.parse( xmlfile, contenthandler[, errorhandler])
```

参数说明：

- **xmlfile** - xml文件名
 - **contenthandler** - 必须是一个ContentHandler的对象
 - **errorhandler** - 如果指定该参数，errorhandler必须是一个SAX ErrorHandler对象
-

parseString方法

parseString方法创建一个XML解析器并解析xml字符串：

```
xml.sax.parseString(xmlstring, contenthandler[, errorhandler])
```

参数说明：

- **xmlstring** - xml字符串
 - **contenthandler** - 必须是一个ContentHandler的对象
 - **errorhandler** - 如果指定该参数，errorhandler必须是一个SAX ErrorHandler对象
-

Python 解析XML实例

```
#!/usr/bin/python3
```

```
import xml.sax
```

```
class MovieHandler( xml.sax.ContentHandler ) :
    def __init__(self):
        self.CurrentData = ""
        self.type = ""
        self.format = ""
        self.year = ""
        self.rating = ""
        self.stars = ""
        self.description = ""

    # 元素开始调用
    def startElement(self, tag, attributes):
        self.CurrentData = tag
        if tag == "movie":
            print ("*****Movie*****")
            title = attributes["title"]
            print ("Title:", title)

    # 元素结束调用
    def endElement(self, tag):
        if self.CurrentData == "type":
            print ("Type:", self.type)
        elif self.CurrentData == "format":
            print ("Format:", self.format)
        elif self.CurrentData == "year":
            print ("Year:", self.year)
        elif self.CurrentData == "rating":
            print ("Rating:", self.rating)
        elif self.CurrentData == "stars":
            print ("Stars:", self.stars)
        elif self.CurrentData == "description":
            print ("Description:", self.description)
        self.CurrentData = ""
```

```

# 读取字符时调用
def characters(self, content):
    if self.CurrentData == "type":
        self.type = content
    elif self.CurrentData == "format":
        self.format = content
    elif self.CurrentData == "year":
        self.year = content
    elif self.CurrentData == "rating":
        self.rating = content
    elif self.CurrentData == "stars":
        self.stars = content
    elif self.CurrentData == "description":
        self.description = content

if ( __name__ == "__main__" ):

    # 创建一个 XMLReader
    parser = xml.sax.make_parser()
    # 关闭命名空间
    parser.setFeature(xml.sax.handler.feature_namespaces, 0)

    # 重写 ContextHandler
    Handler = MovieHandler()
    parser.setContentHandler( Handler )

    parser.parse("movies.xml")

```

以上代码执行结果如下：

```

*****Movie*****
Title: Enemy Behind
Type: War, Thriller
Format: DVD
Year: 2003
Rating: PG
Stars: 10
Description: Talk about a US-Japan war
*****Movie*****
Title: Transformers
Type: Anime, Science Fiction
Format: DVD
Year: 1989
Rating: R
Stars: 8
Description: A schientific fiction
*****Movie*****
Title: Trigun
Type: Anime, Action
Format: DVD
Rating: PG
Stars: 10
Description: Vash the Stampede!
*****Movie*****
Title: Ishtar
Type: Comedy
Format: VHS
Rating: PG
Stars: 2
Description: Viewable boredom

```

完整的 SAX API 文档请查阅[Python SAX APIs](#)

使用xml.dom解析xml

文件对象模型（Document Object Model，简称DOM），是W3C组织推荐的处理可扩展置标语言的标准编程接口。

一个 DOM 的解析器在解析一个 XML 文档时，一次性读取整个文档，把文档中所有元素保存在内存中的一个树结构里，之后你可以利用DOM提供的不同的函数来读取或修改文档的内容和结构，也可以把修改过的内容写入xml文件。

python中用xml.dom.minidom来解析xml文件，实例如下：

```
#!/usr/bin/python3

from xml.dom.minidom import parse
import xml.dom.minidom

# 使用minidom解析器打开 XML 文档
DOMTree = xml.dom.minidom.parse("movies.xml")
collection = DOMTree.documentElement
if collection.hasAttribute("shelf"):
    print ("Root element : %s" % collection.getAttribute("shelf"))

# 在集合中获取所有电影
movies = collection.getElementsByTagName("movie")

# 打印每部电影的详细信息
for movie in movies:
    print ("*****Movie*****")
    if movie.hasAttribute("title"):
        print ("Title: %s" % movie.getAttribute("title"))

    type = movie.getElementsByTagName('type')[0]
    print ("Type: %s" % type.childNodes[0].data)
    format = movie.getElementsByTagName('format')[0]
    print ("Format: %s" % format.childNodes[0].data)
    rating = movie.getElementsByTagName('rating')[0]
    print ("Rating: %s" % rating.childNodes[0].data)
    description = movie.getElementsByTagName('description')[0]
    print ("Description: %s" % description.childNodes[0].data)
```

以上程序执行结果如下：

```
Root element : New Arrivals
*****Movie*****
Title: Enemy Behind
Type: War, Thriller
Format: DVD
Rating: PG
Description: Talk about a US-Japan war
*****Movie*****
Title: Transformers
Type: Anime, Science Fiction
Format: DVD
Rating: R
Description: A schientific fiction
*****Movie*****
Title: Trigun
Type: Anime, Action
Format: DVD
Rating: PG
Description: Vash the Stampede!
*****Movie*****
Title: Ishtar
Type: Comedy
Format: VHS
Rating: PG
Description: Viewable boredom
```

完整的 DOM API 文档请查阅[Python DOM APIs](#)。

Python3 JSON 数据解析

JSON (JavaScript Object Notation) 是一种轻量级的数据交换格式。它基于ECMAScript的一个子集。

Python3 中可以使用 json 模块来对 JSON 数据进行编解码，它包含了两个函数：

- **json.dumps():** 对数据进行编码。
- **json.loads():** 对数据进行解码。

在json的编解码过程中，python 的原始类型与json类型会相互转换，具体的转化对照如下：

Python 编码为 JSON 类型转换对应表：

Python	JSON
dict	object
list, tuple	array
str	string
int, float, int- & float-derived Enums	number
True	true
False	false
None	null

JSON 解码为 Python 类型转换对应表：

JSON	Python
object	dict
array	list
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

json.dumps 与 json.loads 实例

以下实例演示了 Python 数据结构转换为JSON：

```
#!/usr/bin/python3

import json

# Python 字典类型转换为 JSON 对象
data = {
    'no' : 1,
    'name' : 'Runoob',
    'url' : 'http://www.runoob.com'
}

json_str = json.dumps(data)
print ("Python 原始数据: ", repr(data))
print ("JSON 对象: ", json_str)
```

执行以上代码输出结果为：

Python 原始数据: {'url': 'http://www.runoob.com', 'no': 1, 'name': 'Runoob'}
JSON 对象: {"url": "http://www.runoob.com", "no": 1, "name": "Runoob"}

通过输出的结果可以看出, 简单类型通过编码后跟其原始的repr()输出结果非常相似。

接着以上实例, 我们可以将一个JSON编码的字符串转换回一个Python数据结构:

```
#!/usr/bin/python3

import json

# Python 字典类型转换为 JSON 对象
data1 = {
    'no' : 1,
    'name' : 'Runoob',
    'url' : 'http://www.runoob.com'
}

json_str = json.dumps(data1)
print ("Python 原始数据: ", repr(data1))
print ("JSON 对象: ", json_str)

# 将 JSON 对象转换为 Python 字典
data2 = json.loads(json_str)
print ("data2['name']: ", data2['name'])
print ("data2['url']: ", data2['url'])
```

执行以上代码输出结果为:

Python 原始数据: {'name': 'Runoob', 'no': 1, 'url': 'http://www.runoob.com'}
JSON 对象: {"name": "Runoob", "no": 1, "url": "http://www.runoob.com"}
data2['name']: Runoob
data2['url']: http://www.runoob.com

如果你要处理的是文件而不是字符串, 你可以使用 **json.dump()** 和 **json.load()** 来编码和解码JSON数据。例如:

```
# 写入 JSON 数据
with open('data.json', 'w') as f:
    json.dump(data, f)

# 读取数据
with open('data.json', 'r') as f:
    data = json.load(f)
```

更多资料请参考: <https://docs.python.org/3/library/json.html>

Python3 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/python3

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: 1459996086.7115328

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	<code>tm_year</code>	2008
1	<code>tm_mon</code>	1 到 12
2	<code>tm_mday</code>	1 到 31
3	<code>tm_hour</code>	0 到 23
4	<code>tm_min</code>	0 到 59
5	<code>tm_sec</code>	0 到 61 (60或61 是闰秒)
6	<code>tm_wday</code>	0到6 (0是周一)
7	<code>tm_yday</code>	一年中的第几天, 1 到 366
8	<code>tm_isdst</code>	是否为夏令时, 值有: 1(夏令时)、0(不是夏令时)、-1(未知), 默认 -1

获取当前时间

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

```
#!/usr/bin/python3

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=28, tm_sec=49, 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/python3

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 10:29:13 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/python3

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 10:29:46
Thu Apr 07 10:29:46 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/python3

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	函数及描述	实例
	<code>time.altzone</code> 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。	以下实例展示了 <code>altzone()</code> 函数的使用方法：
1	<code>time.asctime([tupletime])</code> 接受时间元组并返回一个字符串。	<pre>>>> import time >>> print ("time.altzone %d " % time.altzone) time.altzone -28800</pre>

2	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 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。	>>> import time >>> t = time.localtime() >>> print ("time.asctime(t): %s " % time.asctime(t)) time.asctime(t): Thu Apr 7 10:36:20 2016
4	time.ctime([secs]) 作用相当于 asctime(localtime(secs)), 未给参数相当于asctime()	以下实例展示了 ctime()函数的使用方法: >>> import time >>> print ("time.ctime(): %s" % time.ctime()) time.ctime(): Thu Apr 7 10:51:58 2016
5	time.gmtime([secs]) 接收时间戳 (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	以下实例展示了 gmtime()函数的使用方法: >>> import time >>> print ("gmtime :", time.gmtime(1455508609.34375)) gmtime : time.struct_time(tm_year=2016, tm_mon=2, tm_mday=15, tm_hour=3, tm_min=56, tm_sec=49, tm_wday=0, tm_yday=46, tm_isdst=0)
6	time.localtime([secs]) 接收时间戳 (1970纪元后经过的浮点秒数) 并返回当地时间下的时间元组t (t.tm_isdst可取0或1，取决于当地当时是不是夏令时)。 time.mktime(tupletime) 接受时间元组并返回时间戳 (1970纪元后经过的浮点秒数)。	以下实例展示了 localtime()函数的使用方法: >>> import time >>> print ("localtime(): ", time.localtime(1455508609.34375)) localtime(): time.struct_time(tm_year=2016, tm_mon=2, tm_mday=15, tm_hour=11, tm_min=56, tm_sec=49, tm_wday=0, tm_yday=46, tm_isdst=0)
7	time.sleep(secs) 推迟调用线程的运行，secs指秒数。	以下实例展示了 sleep()函数的使用方法: #!/usr/bin/python3 import time print ("Start : %s" % time.ctime()) time.sleep(5) print ("End : %s" % time.ctime())
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。	以下实例展示了 strftime()函数的使用方法: >>> import time >>> print (time.strftime("%Y-%m-%d %H:%M:%S", time.localtime())) 2016-04-07 11:18:05
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。	以下实例展示了 strptime()函数的使用方法: >>> import time >>> struct_time = time.strptime("30 Nov 00", "%d %b %y") >>> print ("返回元组: ", struct_time) 返回元组: time.struct_time(tm_year=2000, tm_mon=11, tm_mday=30, tm_hour=0, tm_min=0, tm_sec=0, tm_wday=3, tm_yday=335, tm_isdst=-1)
11	time.time() 返回当前时间的时间戳 (1970纪元后经过的浮点秒数)。	以下实例展示了 time()函数的使用方法: >>> import time >>> print(time.time()) 1459999336.1963577
12	time.tzset() 根据环境变量TZ重新初始化时间相关设置。	实例

Time模块包含了以下2个非常重要的属性:

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

日历（Calendar）模块

此模块的函数都是日历相关的，例如打印某月的字符月历。

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

序号	函数及描述
1	calendar.calendar(year,w=2,l=1,c=6) 返回一个多行字符串格式的year年年历，3个月一行，间隔距离为c。每日宽度间隔为w字符。每行长度为21* W+18+2* C。l是每星期行数。
2	calendar.firstweekday() 返回当前每周起始日期的设置。默认情况下，首次载入calendar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，返回该时刻的时间戳（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：

- [time 模块](#)
- [datetime模块](#)

Python3 内置函数

注意：有些函数与 Python2.x 变化不大，会直接跳转到 Python2.x 教程下的内置函数说明，大家要注意下哈。

内置函数

abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

Python3 range() 函数用法

[Python3 内置函数](#)

Python3 range() 函数返回的是一个可迭代对象（类型是对象），而不是列表类型，所以打印的时候不会打印列表。

Python3 list() 函数是对象迭代器，可以把range()返回的可迭代对象转为一个列表，返回的变量类型为列表。

Python2 [range\(\) 函数](#)返回的是列表。

函数语法

```
range(stop)
range(start, stop[, step])
```

参数说明：

- start: 计数从 start 开始。默认是从 0 开始。例如range（5）等价于range（0， 5）；
- stop: 计数到 stop 结束，但不包括 stop。例如：range（0， 5）是[0, 1, 2, 3, 4]没有5
- step: 步长，默认为1。例如：range（0， 5）等价于 range(0, 5, 1)

实例

```
>>>range(5) range(0, 5)>>> for i in range(5): ... print(i) ... 0 1 2 3 4>>> list(range(5)) [0, 1, 2, 3, 4]>>> list(range(0)) []>>>
```

有两个参数或三个参数的情况（第二种构造方法）：：

```
>>>list(range(0, 30, 5)) [0, 5, 10, 15, 20, 25]>>> list(range(0, 10, 2)) [0, 2, 4, 6, 8]>>> list(range(0, -10, -1)) [0, -1, -2, -3, -4, -5, -6, -7, -8, -9]>>> list(range(1, 0)) []>>>>>>
```

[Python3 内置函数](#)