

## Table of Contents

Python 高级教程	0
Python 面向对象	1
Python 正则表达式	2
Python CGI编程	3
Python MySQL	4
Python 网络编程	5
Python SMTP	6
Python 多线程	7
Python XML解析	8
Python GUI编程(Tkinter)	9
Python 2.x与Python 3.x版本区别	10
Python IDE	11
Python JSON	12
Python 100例	13

# Python 高级教程

# Python 面向对象

Python从设计之初就已经是一门面向对象的语言，正因为如此，在Python中创建一个类和对象是很容易的。本章节我们将详细介绍Python的面向对象编程。如果你以前没有接触过面向对象的编程语言，那你可能需要先了解一些面向对象语言的一些基本特征，在头脑里头形成一个基本的面向对象的概念，这样有助于你更容易的学习Python的面向对象编程。接下来我们先来简单的了解下面面向对象的一些基本特征。

## 面向对象技术简介

- **类(Class):** 用来描述具有相同的属性和方法的对象的集合。它定义了该集合中每个对象所共有的属性和方法。对象是类的实例。
- **类变量:** 类变量在整个实例化的对象中是公用的。类变量定义在类中且在函数体之外。类变量通常不作为实例变量使用。
- **数据成员:** 类变量或者实例变量用于处理类及其实例对象的相关的数据。
- **方法重写:** 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。

## 创建类

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

```
class ClassName:
    '类的帮助信息'    #类文档字符串
    class_suite    #类体
```

类的帮助信息可以通过**ClassName.doc**查看。 **class\_suite** 由类成员，方法，数据属性组成。

## 实例

以下是一个简单的Python类实例:

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

class Employee:
    '所有员工的基类'
    empCount = 0

    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.empCount += 1

    def displayCount(self):
```

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

def displayEmployee(self):
    print "Name : ", self.name, " , Salary: ", self.salary
```

- `empCount` 变量是一个类变量，它的值将在这个类的所有实例之间共享。你可以在内部类或外部类使用 `Employee.empCount` 访问。
- 第一种方法 `__init__()` 方法是一种特殊的方法，被称为类的构造函数或初始化方法，当创建了这个类的实例时就会调用该方法
- `self` 代表类的实例，`self` 在定义类的方法时是必须有的，虽然在调用时不必传入相应的参数。

## 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 0x10d066878>
__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 0x10d066878>
__main__.Test
```

## 创建实例对象

实例化类其他编程语言中一般用关键字 `new`，但是在 `Python` 中并没有这个关键字，类的实例化类似函数调用方式。以下使用类的名称 `Employee` 来实例化，并通过 `__init__` 方法接受参数。

```
"创建 Employee 类的第一个对象"
emp1 = Employee("Zara", 2000)
"创建 Employee 类的第二个对象"
emp2 = Employee("Manni", 5000)
```

## 访问属性

您可以使用点(.)来访问对象的属性。使用如下类的名称访问类变量:

```
emp1.displayEmployee()  
emp2.displayEmployee()  
print "Total Employee %d" % Employee.empCount
```

完整实例:

```
#!/usr/bin/python  
# -*- coding: UTF-8 -*-  
  
class Employee:  
    '所有员工的基类'  
    empCount = 0  
  
    def __init__(self, name, salary):  
        self.name = name  
        self.salary = salary  
        Employee.empCount += 1  
  
    def displayCount(self):  
        print "Total Employee %d" % Employee.empCount  
  
    def displayEmployee(self):  
        print "Name : ", self.name, " , Salary: ", self.salary  
  
"创建 Employee 类的第一个对象"  
emp1 = Employee("Zara", 2000)  
"创建 Employee 类的第二个对象"  
emp2 = Employee("Manni", 5000)  
emp1.displayEmployee()  
emp2.displayEmployee()  
print "Total Employee %d" % Employee.empCount
```

执行以上代码输出结果如下:

```
Name :  Zara ,Salary:  2000  
Name :  Manni ,Salary:  5000  
Total Employee 2
```

你可以添加, 删除, 修改类的属性, 如下所示:

```
emp1.age = 7  # 添加一个 'age' 属性  
emp1.age = 8  # 修改 'age' 属性  
del emp1.age  # 删除 'age' 属性
```

你也可以使用以下函数的方式来访问属性:

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

```
hasattr(emp1, 'age')    # 如果存在 'age' 属性返回 True。  
getattr(emp1, 'age')    # 返回 'age' 属性的值  
setattr(emp1, 'age', 8) # 添加属性 'age' 值为 8
```

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

## Python内置类属性

- `__dict__` : 类的属性（包含一个字典，由类的数据属性组成）
- `__doc__` : 类的文档字符串
- `__name__` : 类名
- `__module__` : 类定义所在的模块（类的全名是 '`__main__.className`'，如果类位于一个导入模块 `mymod` 中，那么 `className.__module__` 等于 `mymod`）
- `__bases__` : 类的所有父类构成元素（包含了一个由所有父类组成的元组）

Python内置类属性调用实例如下：

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

class Employee:
    '所有员工的基类'
    empCount = 0

    def __init__(self, name, salary):
        self.name = name
        self.salary = salary
        Employee.empCount += 1

    def displayCount(self):
        print "Total Employee %d" % Employee.empCount

    def displayEmployee(self):
        print "Name : ", self.name, ", Salary: ", self.salary

print "Employee.__doc__:", Employee.__doc__
print "Employee.__name__:", Employee.__name__
print "Employee.__module__:", Employee.__module__
print "Employee.__bases__:", Employee.__bases__
print "Employee.__dict__:", Employee.__dict__
```

执行以上代码输出结果如下：

```
Employee.__doc__: 所有员工的基类
Employee.__name__: Employee
Employee.__module__: __main__
Employee.__bases__: ()
Employee.__dict__: {'__module__': '__main__', 'displayCount': <function displayCount at 0x10a939c80>, 'empCount': 0}
```

## python对象销毁(垃圾回收)

Python 使用了引用计数这一简单技术来跟踪和回收垃圾。在 Python 内部记录着所有使用中的对象各有多少引用。一个内部跟踪变量，称为一个引用计数器。当对象被创建时，就创建了一个引用计数，当这个对象不再需要时，也就是说，这个对象的引用计数变为0时，它被垃圾回收。但是回收不是“立即”的，由解释器在适当的时机，将垃圾对象占用的内存空间回收。

```
a = 40      # 创建对象  <40>
b = a       # 增加引用， <40> 的计数
c = [b]     # 增加引用， <40> 的计数
```

```
del a      # 减少引用 <40> 的计数
b = 100    # 减少引用 <40> 的计数
c[0] = -1  # 减少引用 <40> 的计数
```

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

实例

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

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

class Point:
    def __init__( self, x=0, y=0):
        self.x = x
        self.y = y
    def __del__(self):
        class_name = self.__class__.__name__
        print class_name, "销毁"

pt1 = Point()
pt2 = pt1
pt3 = pt1
print id(pt1), id(pt2), id(pt3) # 打印对象的id
del pt1
del pt2
del pt3
```

以上实例运行结果如下：

```
3083401324 3083401324 3083401324
Point 销毁
```

注意：通常你需要在单独的文件中定义一个类，

## 类的继承

面向对象的编程带来的主要好处之一是代码的重用，实现这种重用的方法之一是通过继承机制。继承完全可以理解成类之间的类型和子类型关系。需要注意的地方：继承语法 **class** 派生类名（基类名）：`//... 基类名`写在括号里，基本类是在类定义的时候，在元组之中指明的。在**python**中继承中的一些特点：

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

如果在继承元组中列了一个以上的类，那么它就被称作"多重继承"。

语法：

派生类的声明，与他们的父类类似，继承的基类列表跟在类名之后，如下所示：

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

```
class_suite
```

实例：

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

class Parent:      # 定义父类
    parentAttr = 100
    def __init__(self):
        print "调用父类构造函数"

    def parentMethod(self):
        print '调用父类方法'

    def setAttr(self, attr):
        Parent.parentAttr = attr

    def getAttr(self):
        print "父类属性 :", Parent.parentAttr

class Child(Parent): # 定义子类
    def __init__(self):
        print "调用子类构造方法"

    def childMethod(self):
        print '调用子类方法 child method'

c = Child()      # 实例化子类
c.childMethod()  # 调用子类的方法
c.parentMethod() # 调用父类方法
c.setAttr(200)   # 再次调用父类的方法
c.getAttr()      # 再次调用父类的方法
```

以上代码执行结果如下：

```
调用子类构造方法
调用子类方法 child method
调用父类方法
父类属性 : 200
```

你可以继承多个类

```
class A:          # 定义类 A
    ....

class B:          # 定义类 B
    ....

class C(A, B):    # 继承类 A 和 B
    ....
```

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

- `issubclass()` - 布尔函数判断一个类是另一个类的子类或者子孙类，语法：`issubclass(sub,sup)`
- `isinstance(obj, Class)` 布尔函数如果obj是Class类的实例对象或者是一个Class子类的实例对象则返回true。

## 方法重写



如果你的父类方法的功能不能满足你的需求，你可以在子类重写你父类的方法：

实例：

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

class Parent:          # 定义父类
    def myMethod(self):
        print '调用父类方法'

class Child(Parent):    # 定义子类
    def myMethod(self):
        print '调用子类方法'

c = Child()             # 子类实例
c.myMethod()            # 子类调用重写方法
```

执行以上代码输出结果如下：

```
调用子类方法
```

## 基础重载方法

下表列出了一些通用的功能，你可以在自己的类重写：

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

## 运算符重载

Python同样支持运算符重载，实例如下：

```
#!/usr/bin/python

class Vector:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def __str__(self):
        return 'Vector (%d, %d)' % (self.a, self.b)

    def __add__(self,other):
        return Vector(self.a + other.a, self.b + other.b)

v1 = Vector(2,10)
v2 = Vector(5,-2)
print v1 + v2
```

以上代码执行结果如下所示:

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

## 类属性与方法

### 类的私有属性

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

### 类的方法

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

### 类的私有方法

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

实例

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

class JustCounter:
    __secretCount = 0 # 私有变量
    publicCount = 0   # 公开变量

    def count(self):
        self.__secretCount += 1
        self.publicCount += 1
        print self.__secretCount

counter = JustCounter()
counter.count()
counter.count()
print counter.publicCount
print counter.__secretCount # 报错, 实例不能访问私有变量
```

Python 通过改变名称来包含类名:

```
1
2
2
Traceback (most recent call last):
  File "test.py", line 17, in <module>
    print counter.__secretCount # 报错, 实例不能访问私有变量
AttributeError: JustCounter instance has no attribute '__secretCount'
```

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

```
.....
print counter._JustCounter__secretCount
```

执行以上代码, 执行结果如下:

1  
2  
2  
2

# Python正则表达式

正则表达式是一个特殊的字符序列，它能帮助你方便的检查一个字符串是否与某种模式匹配。Python 自1.5版本起增加了re 模块，它提供 Perl 风格的正则表达式模式。re 模块使 Python 语言拥有全部的正则表达式功能。compile 函数根据一个模式字符串和可选的标志参数生成一个正则表达式对象。该对象拥有一系列方法用于正则表达式匹配和替换。re 模块也提供了与这些方法功能完全一致的函数，这些函数使用一个模式字符串做为它们的第一个参数。 本章节主要介绍Python中常用的正则表达式处理函数。

## re.match函数

re.match 尝试从字符串的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。

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 1：

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

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 2：

```
#!/usr/bin/python
import re

line = "Cats are smarter than dogs"

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

if matchObj:
```

```
print "matchObj.group() : ", matchObj.group()
print "matchObj.group(1) : ", matchObj.group(1)
print "matchObj.group(2) : ", matchObj.group(2)
else:
    print "No match!!"
```

以上实例执行结果如下：

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

## re.search方法

**re.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。

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
<b>group(num=0)</b>	匹配的整个表达式的字符串， <b>group()</b> 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。
<b>groups()</b>	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 1：

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

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 2：

```
#!/usr/bin/python
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/python
import re

line = "Cats are smarter than dogs";

matchObj = re.match( r'dogs', line, re.M|re.I)
if matchObj:
    print "match --> matchObj.group() : ", matchObj.group()
else:
    print "No match!!"

matchObj = re.search( r'dogs', line, re.M|re.I)
if matchObj:
    print "search --> matchObj.group() : ", matchObj.group()
else:
    print "No match!!"
```

以上实例运行结果如下：

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

## 检索和替换

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

```
re.sub(pattern, repl, string, count=0, flags=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/python
```

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

import re

phone = "2004-959-559 # 这是一个国外电话号码"

# 删除字符串中的 Python注释
num = re.sub(r'#.*$', "", phone)
print "电话号码是: ", num

# 删除非数字(-)的字符串
num = re.sub(r'\D', "", phone)
print "电话号码是 : ", num
```

以上实例执行结果如下：

```
电话号码是： 2004-959-559
电话号码是 : 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：

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

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
```

## 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
<code>re.I</code>	使匹配对大小写不敏感
<code>re.L</code>	做本地化识别（ <code>locale-aware</code> ）匹配
<code>re.M</code>	多行匹配，影响 <code>^</code> 和 <code>\$</code>
<code>re.S</code>	使 <code>.</code> 匹配包括换行在内的所有字符
<code>re.U</code>	根据 <code>Unicode</code> 字符集解析字符。这个标志影响 <code>\w</code> , <code>\W</code> , <code>\b</code> , <code>\B</code> .
<code>re.X</code>	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
<code>^</code>	匹配字符串的开头
<code>\$</code>	匹配字符串的末尾。
<code>.</code>	匹配任意字符，除了换行符，当 <code>re.DOTALL</code> 标记被指定时，则可以匹配包括换行符的任意字符。
<code>[...]</code>	用来表示一组字符,单独列出： <code>[amk]</code> 匹配 <code>'a'</code> , <code>'m'</code> 或 <code>'k'</code>
<code>...</code>	不在 <code>[]</code> 中的字符： <code>abc</code> 匹配除了 <code>a,b,c</code> 之外的字符。
<code>re*</code>	匹配0个或多个的表达式。
<code>re+</code>	匹配1个或多个的表达式。
<code>re?</code>	匹配0个或1个由前面的正则表达式定义的片段，非贪婪方式
<code>re{ n}</code>	精确匹配 <code>n</code> 个前面表达式。
<code>re{ n, }</code>	匹配 <code>n</code> 到结尾表达式。
<code>re{ n, m}</code>	匹配 <code>n</code> 到 <code>m</code> 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 <code>a</code> 或 <code>b</code>
<code>(re)</code>	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 <code>(...)</code> , 但是不表示一个组
<code>(? imx: re)</code>	在括号中使用 <code>i</code> , <code>m</code> , 或 <code>x</code> 可选标志
<code>(?- imx: re)</code>	在括号中不使用 <code>i</code> , <code>m</code> , 或 <code>x</code> 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>(?&gt; re)</code>	匹配的独立模式，省去回溯。
<code>\w</code>	匹配字母数字及下划线
<code>\W</code>	匹配非字母数字及下划线
<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



\Z	匹配字符串结束，如果是存在换行，只匹配到换行前的结束字符串。
\z	匹配字符串结束
\G	匹配最后匹配完成的位置。
\b	匹配一个单词边界，也就是指单词和空格间的位置。例如， 'er\b' 可以匹配"never" 中的 'er'，但不能匹配 "verb" 中的 'er'。
\B	匹配非单词边界。'er\B' 能匹配 "verb" 中的 'er'，但不能匹配 "never" 中的 'er'。
\n, \t, 等.	匹配一个换行符。匹配一个制表符。等
\1...\9	匹配第n个分组的子表达式。
\10	匹配第n个分组的子表达式，如果它经匹配。否则指的是八进制字符码的表达式。

## 正则表达式实例

### 字符匹配

实例	描述
python	匹配 "python".

### 字符类

实例	描述
[Pp]ython	匹配 "Python" 或 "python"
rub[ye]	匹配 "ruby" 或 "rube"
[aeiou]	匹配中括号内的任意一个字母
[0-9]	匹配任何数字。类似于 [0123456789]
[a-z]	匹配任何小写字母
[A-Z]	匹配任何大写字母
[a-zA-Z0-9]	匹配任何字母及数字
<a href="#">aeiou</a>	除了aeiou字母以外的所有字符
<a href="#">0-9</a>	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 <a href="#">0-9</a> 。
\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 <a href="#">\f\n\r\t\v</a> 。
\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 href="#">A-Za-z0-9_</a> 。

# 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。

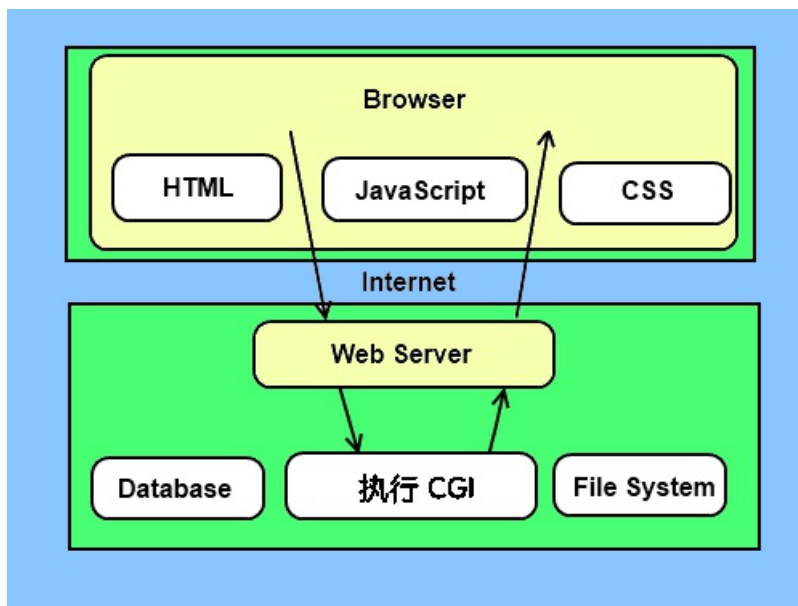
## 网页浏览

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>
```

```
</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/python
# -*- coding: UTF-8 -*-

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。这个信息经跟在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/python
# -*- coding: UTF-8 -*-
# filename:test.py

import os

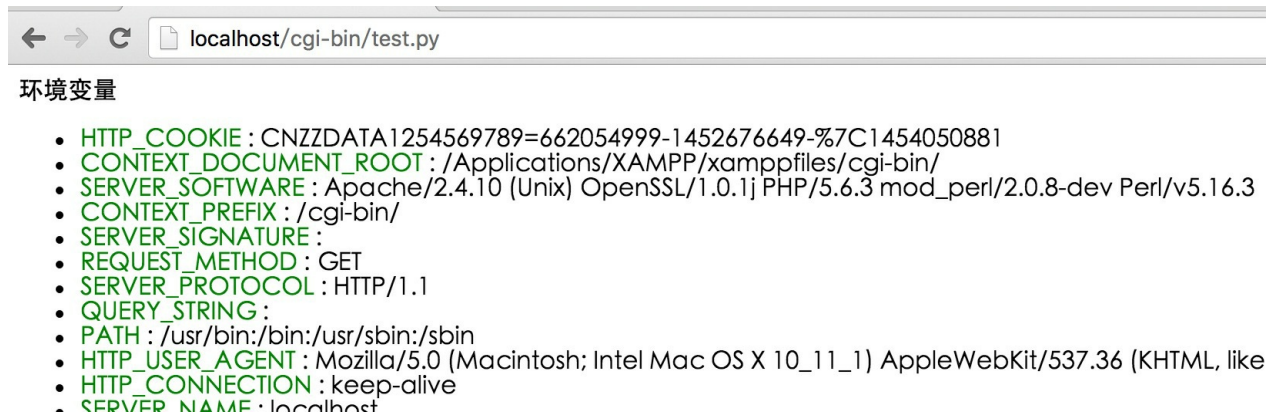
print "Content-type: text/html"
```

```

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'" % (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/python
# -*- coding: UTF-8 -*-

# filename: test.py

# 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>%s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:

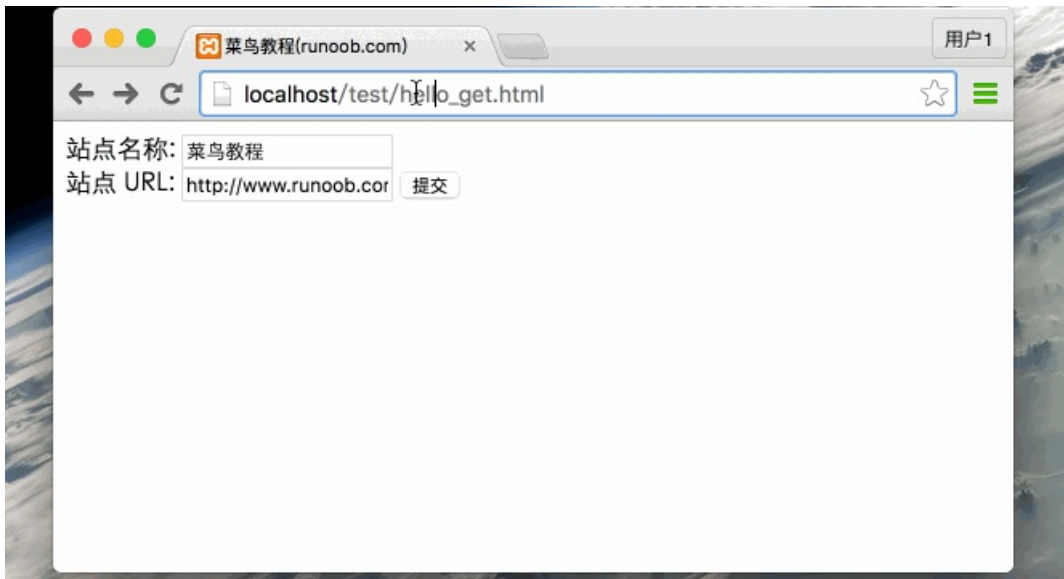
```
<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>菜鸟教程(runoob.com)</title>
</head>
<body>
<form action="/cgi-bin/hello_get.py" method="get">
  站点名称: <input type="text" name="name"> <br />

  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/python
# -*- coding: UTF-8 -*-

# 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>%s官网: %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>
```

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/python
# -*- coding: UTF-8 -*-

# 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"

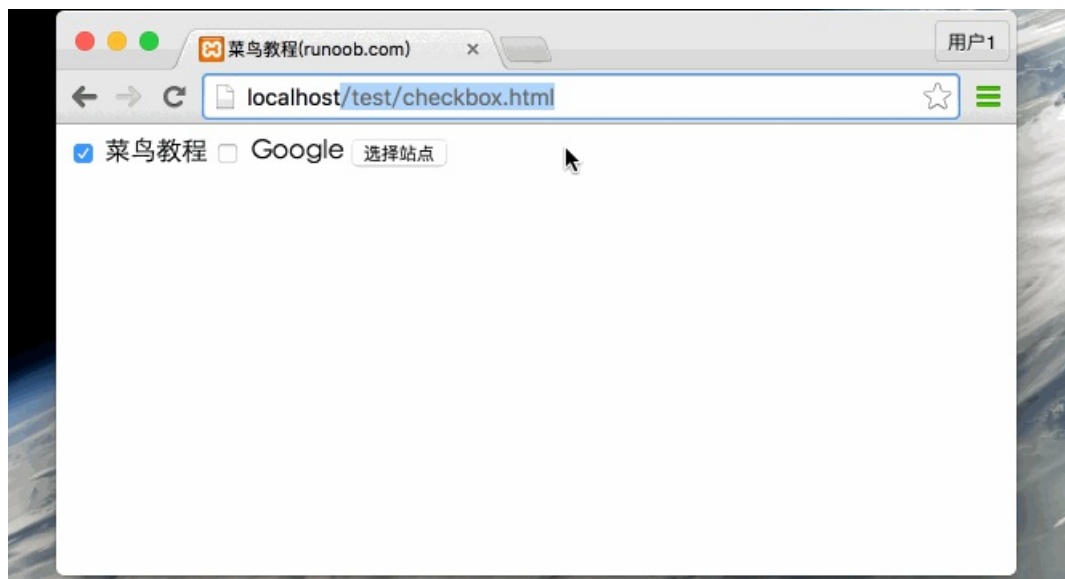
print "Content-type:text/html"
print
print "<html>"
print "<head>"
print "<meta charset=\\"utf-8\\">"
print "<title>菜鸟教程 CGI 测试实例</title>"
print "</head>"
print "<body>"
print "<h2> 菜鸟教程是否选择了 : %s</h2>" % runoob_flag
print "<h2> Google 是否选择了 : %s</h2>" % google_flag
print "</body>"
print "</html>"

```

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/python
# -*- coding: UTF-8 -*-

# 引入 CGI 处理模块
import cgi, cgitb

# 创建 FieldStorage的实例
form = cgi.FieldStorage()

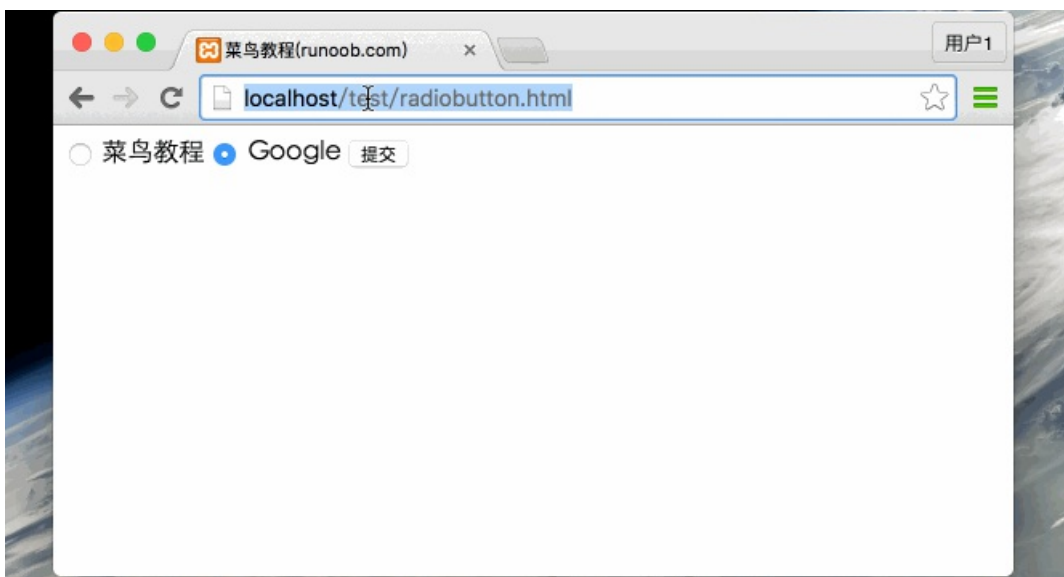
# 接收字段数据
if form.getvalue('site'):
    site = form.getvalue('site')
else:
    site = "提交数据为空"

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/python
# -*- coding: UTF-8 -*-

# 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/python
# -*- coding: UTF-8 -*-

# 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"

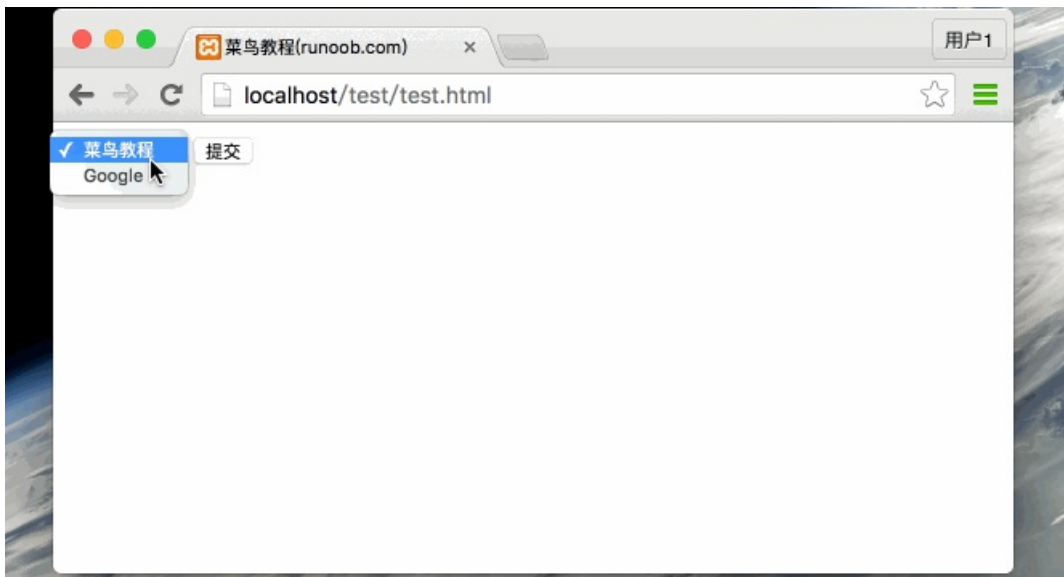
print "Content-type:text/html"
print
print "<html>"
print "<head>"
```

```
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进行判断，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 HH:MM:SS"
- path=path: 设置cookie支持的路径,如果path是一个路径，则cookie对这个目录下的所有文件及子目录生效，例如： path="/cgi-bin/"，如果path是一个文件，则cookie指对这个文件生效，例如： path="/cgi-bin/cookie.cgi"。
- domain=domain: 对cookie生效的域名，例如： -domain="www.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/python
# -*- coding: UTF-8 -*-
#
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>
"""
```

将以上代码保存到 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/python
# -*- coding: UTF-8 -*-

# 导入模块
import os
import Cookie

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>

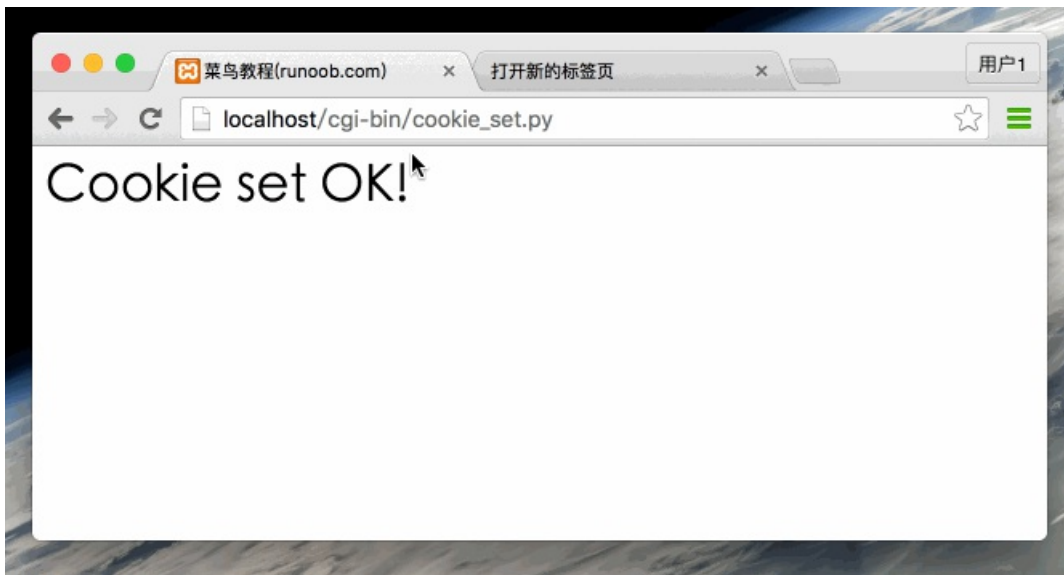
"""

```

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/python
# -*- coding: UTF-8 -*-

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())

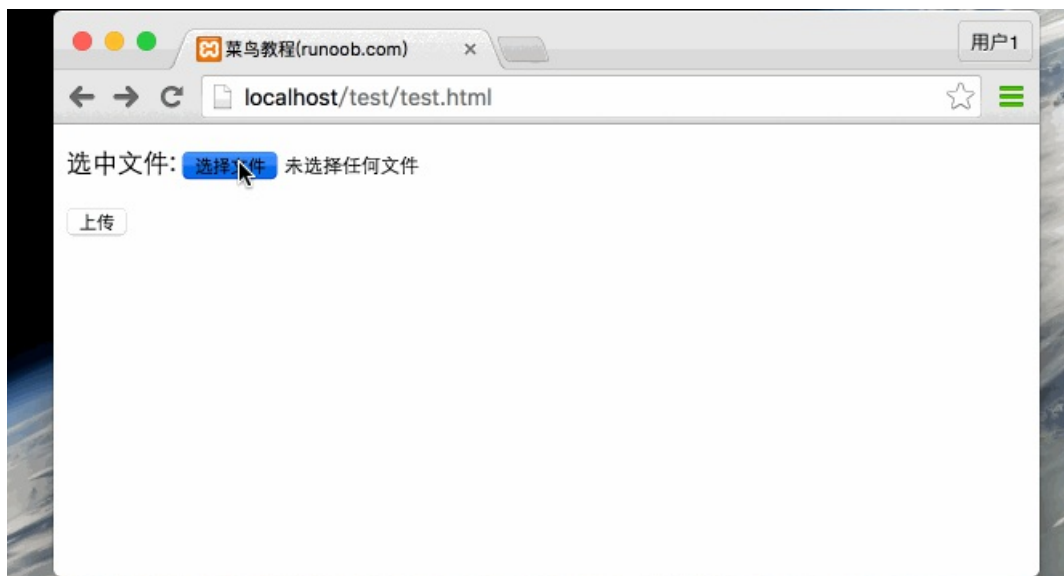
    message = '文件 "' + fn + '" 上传成功'
else:
    message = '文件没有上传'

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/python
# -*- coding: UTF-8 -*-

# 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()
```

# python操作mysql数据库

Python 标准数据库接口为 Python DB-API, Python DB-API为开发人员提供了数据库应用编程接口。Python 数据库接口支持非常多的数据库, 你可以选择适合你项目的数据库:

- GadFly
- mSQL
- MySQL
- PostgreSQL
- Microsoft SQL Server 2000
- Informix
- Interbase
- Oracle
- Sybase

你可以访问[Python数据库接口及API](#)查看详细的支持数据库列表。不同的数据库你需要下载不同的DB API模块, 例如你需要访问Oracle数据库和Mysql数据, 你需要下载Oracle和MySQL数据库模块。DB-API 是一个规范. 它定义了一系列必须的对象和数据库存取方式, 以便为各种各样的底层数据库系统和多种多样的数据库接口程序提供一致的访问接口。Python的DB-API, 为大多数的数据库实现了接口, 使用它连接各数据库后, 就可以用相同的方式操作各数据库。Python DB-API使用流程:

- 引入 API 模块。
- 获取与数据库的连接。
- 执行SQL语句和存储过程。
- 关闭数据库连接。

## 什么是MySQLdb?

MySQLdb 是用于Python链接Mysql数据库的接口, 它实现了 Python 数据库 API 规范 V2.0, 基于 MySQL C API 上建立的。

## 如何安装MySQLdb?

为了用DB-API编写MySQL脚本, 必须确保已经安装了MySQL。复制以下代码, 并执行:

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

import MySQLdb
```

如果执行后的输出结果如下所示, 意味着你没有安装 MySQLdb 模块:

```
Traceback (most recent call last):
  File "test.py", line 3, in <module>
    import MySQLdb
ImportError: No module named MySQLdb
```

安装MySQLdb, 请访问 <http://sourceforge.net/projects/mysql-python>, (Linux平台可以访问: <https://pypi.python.org/pypi/MySQL-python>) 从这里可选择适合您的平台的安装包, 分为预编译的二进制文件和源代码安装包。如果您选择二进制文件发行版本的话, 安装过程基本安装提示即可完成。如果从源代码进行安装的话, 则需要切换到MySQLdb发行版本的顶级目录, 并键入下列命令:

```
$ gunzip MySQL-python-1.2.2.tar.gz
$ tar -xvf MySQL-python-1.2.2.tar
$ cd MySQL-python-1.2.2
$ python setup.py build
$ python setup.py install
```

注意：请确保您有root权限来安装上述模块。

## 数据库连接

连接数据库前，请先确认以下事项：

- 您已经创建了数据库 **TESTDB**.
- 在**TESTDB**数据库中您已经创建了表 **EMPLOYEE**
- **EMPLOYEE**表字段为 **FIRST\_NAME**, **LAST\_NAME**, **AGE**, **SEX** 和 **INCOME**。
- 连接数据库**TESTDB**使用的用户名为 "**testuser**"，密码为 "**test123**",你可以可以自己设定或者直接使用**root**用户名及其密码，**Mysql**数据库用户授权请使用**Grant**命令。
- 在你的机子上已经安装了 **Python MySQLdb** 模块。
- 如果您对**sql**语句不熟悉，可以访问我们的 **SQL基础教程**

实例：

```
以下实例链接Mysql的TESTDB数据库：
#!/usr/bin/python
# -*- coding: UTF-8 -*-

import MySQLdb

# 打开数据库连接
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# 使用execute方法执行SQL语句
cursor.execute("SELECT VERSION()")

# 使用 fetchone() 方法获取一条数据库。
data = cursor.fetchone()

print "Database version : %s " % data

# 关闭数据库连接
db.close()
```

执行以上脚本输出结果如下：

```
Database version : 5.0.45
```

## 创建数据库表

如果数据库连接存在我们可以使用**execute()**方法来为数据库创建表，如下所示创建表**EMPLOYEE**：

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

```

import MySQLdb

# 打开数据库连接
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# 如果数据表已经存在使用 execute() 方法删除表。
cursor.execute("DROP TABLE IF EXISTS EMPLOYEE")

# 创建数据表SQL语句
sql = """CREATE TABLE EMPLOYEE (
            FIRST_NAME  CHAR(20) NOT NULL,
            LAST_NAME   CHAR(20),
            AGE INT,
            SEX CHAR(1),
            INCOME FLOAT )"""

cursor.execute(sql)

# 关闭数据库连接
db.close()

```

## 数据库插入操作

以下实例使用执行 SQL INSERT 语句向表 EMPLOYEE 插入记录：

```

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

import MySQLdb

# 打开数据库连接
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# SQL 插入语句
sql = """INSERT INTO EMPLOYEE(FIRST_NAME,
            LAST_NAME, AGE, SEX, INCOME)
            VALUES ('Mac', 'Mohan', 20, 'M', 2000)"""
try:
    # 执行sql语句
    cursor.execute(sql)
    # 提交到数据库执行
    db.commit()
except:
    # Rollback in case there is any error
    db.rollback()

# 关闭数据库连接
db.close()

```

以上例子也可以写成如下形式：

```

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

```

```

import MySQLdb

# 打开数据库连接
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# SQL 插入语句
sql = "INSERT INTO EMPLOYEE(FIRST_NAME, \
        LAST_NAME, AGE, SEX, INCOME) \
        VALUES ('%s', '%s', '%d', '%c', '%d' )" % \
        ('Mac', 'Mohan', 20, 'M', 2000)
try:
    # 执行sql语句
    cursor.execute(sql)
    # 提交到数据库执行
    db.commit()
except:
    # 发生错误时回滚
    db.rollback()

# 关闭数据库连接
db.close()

```

实例：

以下代码使用变量向SQL语句中传递参数：

```

.....
user_id = "test123"
password = "password"

con.execute('insert into Login values("%s", "%s")' % \
            (user_id, password))
.....

```

## 数据库查询操作

Python查询Mysql使用 `fetchone()` 方法获取单条数据, 使用`fetchall()` 方法获取多条数据。

- `fetchone()`: 该方法获取下一个查询结果集。结果集是一个对象
- `fetchall()`:接收全部的返回结果行。
- `rowcount`: 这是一个只读属性，并返回执行`execute()`方法后影响的行数。

实例：

查询EMPLOYEE表中salary（工资）字段大于1000的所有数据：

```

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

import MySQLdb

# 打开数据库连接
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# 使用cursor()方法获取操作游标
cursor = db.cursor()

```

```
# SQL 查询语句
sql = "SELECT * FROM EMPLOYEE \
      WHERE INCOME > '%d'" % (1000)
try:
    # 执行SQL语句
    cursor.execute(sql)
    # 获取所有记录列表
    results = cursor.fetchall()
    for row in results:
        fname = row[0]
        lname = row[1]
        age = row[2]
        sex = row[3]
        income = row[4]
        # 打印结果
        print "fname=%s,lname=%s,age=%d,sex=%s,income=%d" % \
              (fname, lname, age, sex, income )
except:
    print "Error: unable to 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 **EMPLOYEE** 表中的 **SEX** 字段为 'M' 的 **AGE** 字段递增 1：

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

import MySQLdb

# 打开数据库连接
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# SQL 更新语句
sql = "UPDATE EMPLOYEE SET AGE = AGE + 1 WHERE SEX = '%c'" % ('M')
try:
    # 执行SQL语句
    cursor.execute(sql)
    # 提交到数据库执行
    db.commit()
except:
    # 发生错误时回滚
    db.rollback()

# 关闭数据库连接
db.close()
```

## 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：

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

import MySQLdb

# 打开数据库连接
db = MySQLdb.connect("localhost","testuser","test123","TESTDB" )

# 使用cursor()方法获取操作游标
cursor = db.cursor()

# SQL 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：

```
# SQL删除记录语句
sql = "DELETE FROM EMPLOYEE WHERE AGE > '%d'" % (20)
try:
    # 执行SQL语句
    cursor.execute(sql)
    # 向数据库提交
    db.commit()
except:
    # 发生错误时回滚
    db.rollback()
```

对于支持事务的数据库，在Python数据库编程中，当游标建立之时，就自动开始了一个隐形的数据库事务。**commit()**方法游标的所有更新操作，**rollback()**方法回滚当前游标的所有操作。每一个方法都开始了一个新的事务。

## 错误处理

DB API中定义了一些数据库操作的错误及异常，下表列出了这些错误和异常：

--	--

异常	描述
Warning	当有严重警告时触发，例如插入数据是被截断等等。必须是StandardError 的子类。
Error	警告以外所有其他错误类。必须是 StandardError 的子类。
InterfaceError	当有数据库接口模块本身的错误（而不是数据库的错误）发生时触发。 必须是Error的子类。
DatabaseError	和数据库有关的错误发生时触发。 必须是Error的子类。
DataError	当有数据处理时的错误发生时触发，例如：除零错误，数据超范围等等。 必须是DatabaseError的子类。
OperationalError	指非用户控制的，而是操作数据库时发生的错误。例如：连接意外断开、数据库名未找到、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DatabaseError的子类。
IntegrityError	完整性相关的错误，例如外键检查失败等。必须是DatabaseError子类。
InternalError	数据库的内部错误，例如游标（cursor）失效了、事务同步失败等等。 必须是DatabaseError子类。
ProgrammingError	程序错误，例如数据表（table）没找到或已存在、SQL语句语法错误、参数数量错误等等。必须是DatabaseError的子类。
NotSupportedError	不支持错误，指使用了数据库不支持的函数或API等。例如在连接对象上 使用.rollback()函数，然而数据库并不支持事务或者事务已关闭。 必须是DatabaseError的子类。



# Python 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

### 服务器端套接字

函数	描述
<b>s.bind()</b>	绑定地址（ <b>host,port</b> ）到套接字， 在 <b>AF_INET</b> 下,以元组（ <b>host,port</b> ）的形式表示地址。
<b>s.listen()</b>	开始 <b>TCP</b> 监听。 <b>backlog</b> 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 <b>1</b> ，大部分应用程序设为 <b>5</b> 就可以了。
<b>s.accept()</b>	被动接受 <b>TCP</b> 客户端连接,(阻塞式)等待连接的到来

### 客户端套接字

函数	描述
<b>s.connect()</b>	主动初始化 <b>TCP</b> 服务器连接，。一般 <b>address</b> 的格式为元组（ <b>hostname,port</b> ）， 如果连接出错，返回 <b>socket.error</b> 错误。
<b>s.connect_ex()</b>	<b>connect()</b> 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

函数	描述
<b>s.recv()</b>	接收 <b>TCP</b> 数据，数据以字符串形式返回， <b>bufsize</b> 指定要接收的最大数据量。 <b>flag</b> 提供有关消息的其他信息，通常可以忽略。
<b>s.send()</b>	发送 <b>TCP</b> 数据，将 <b>string</b> 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 <b>string</b> 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()	返回套接字自己的地址。通常是一个元组(ipaddr,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/python
# -*- coding: UTF-8 -*-
# 文件名: server.py

import socket                # 导入 socket 模块

s = socket.socket()          # 创建 socket 对象
host = socket.gethostname() # 获取本地主机名
port = 12345                 # 设置端口
s.bind((host, port))         # 绑定端口

s.listen(5)                  # 等待客户端连接
while True:
    c, addr = s.accept()      # 建立客户端连接。
    print '连接地址: ', addr
    c.send('欢迎访问菜鸟教程! ')
    c.close()                 # 关闭连接
```

## 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 `12345`。`socket.connect(hosname, 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/python
```

```
# -*- coding: UTF-8 -*-
# 文件名: client.py

import socket          # 导入 socket 模块

s = socket.socket()     # 创建 socket 对象
host = socket.gethostname() # 获取本地主机名
port = 12345           # 设置端口号

s.connect((host, port))
print s.recv(1024)
s.close()
```

现在我们打开两个终端，第一个终端执行 `server.py` 文件：

```
$ python server.py
```

第二个终端执行 `client.py` 文件：

```
$ python client.py
欢迎访问菜鸟教程！
```

这时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', 62461)
```

## 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	smtpplib
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](#)。

# Python 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smtpplib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 以下执行实例需要你本机已安装了支持 SMTP 的服务，如：sendmail。 以下是一个使用 Python 发送邮件简单的实例：

实例

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

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`（邮件传输代理程序），就会输出：

```
$ python 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4:30

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，也可以使用其他邮件服务商的 `SMTP` 访问（QQ、网易、Google等）。

实例

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

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/python
# -*- coding: UTF-8 -*-

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><a href="http://www.runoob.com">这是一个链接</a></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：

```
$ python 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/python
# -*- coding: UTF-8 -*-

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"
```

```
$ python 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/python
# -*- coding: UTF-8 -*-

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'))
http://www.runoob.com/wp-content/uploads/2014/01/qqmail-set.jpg
# 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"
```

```
$ python 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 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:



QQ 邮箱 SMTP 服务器地址: `smtp.qq.com`, 端口: 25。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/python
# -*- coding: UTF-8 -*-

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 = 'xxxxxxxxxx'            # 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("smtp.qq.com", 25) # 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
        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:



这是菜鸟教程的邮件测试内容

更多内容请参阅: <https://docs.python.org/2/library/email-examples.html>。

# Python 多线程

多线程类似于同时执行多个不同程序，多线程运行有如下优点：

- 使用线程可以把占据长时间的程序中的任务放到后台去处理。
- 用户界面可以更加吸引人，这样比如用户点击了一个按钮去触发某些事件的处理，可以弹出一个进度条来显示处理的进度
- 程序的运行速度可能加快
- 在一些等待的任务实现上如用户输入、文件读写和网络收发数据等，线程就比较有用了。在这种情况下我们可以释放一些珍贵的资源如内存占用等等。

线程在执行过程中与进程还是有区别的。每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口。但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制。每个线程都有他自己的一组CPU寄存器，称为线程的上下文，该上下文反映了线程上次运行该线程的CPU寄存器的状态。指令指针和堆栈指针寄存器是线程上下文中两个最重要的寄存器，线程总是在进程得到上下文中运行的，这些地址都用于标志拥有线程的进程地址空间中的内存。

- 线程可以被抢占（中断）。
- 在其他线程正在运行时，线程可以暂时搁置（也称为睡眠）-- 这就是线程的退让。

## 开始学习Python线程

Python中使用线程有两种方式：函数或者用类来包装线程对象。函数式：调用thread模块中的start\_new\_thread()函数来产生新线程。语法如下: thread.start\_new\_thread ( function, args[, kwargs] )

参数说明：

- function - 线程函数。
- args - 传递给线程函数的参数,他必须是个tuple类型。
- kwargs - 可选参数。

实例：

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

import thread
import time

# 为线程定义一个函数
def print_time( threadName, delay):
    count = 0
    while count < 5:
        time.sleep(delay)
        count += 1
        print "%s: %s" % ( threadName, time.ctime(time.time()) )

# 创建两个线程
try:
    thread.start_new_thread( print_time, ("Thread-1", 2, ) )
    thread.start_new_thread( print_time, ("Thread-2", 4, ) )
except:
    print "Error: unable to start thread"

while 1:
    pass
```

执行以上程序输出结果如下：

```
Thread-1: Thu Jan 22 15:42:17 2009
Thread-1: Thu Jan 22 15:42:19 2009
Thread-2: Thu Jan 22 15:42:19 2009
Thread-1: Thu Jan 22 15:42:21 2009
Thread-2: Thu Jan 22 15:42:23 2009
Thread-1: Thu Jan 22 15:42:23 2009
Thread-1: Thu Jan 22 15:42:25 2009
Thread-2: Thu Jan 22 15:42:27 2009
Thread-2: Thu Jan 22 15:42:31 2009
Thread-2: Thu Jan 22 15:42:35 2009
```

线程的结束一般依靠线程函数的自然结束；也可以在线程函数中调用`thread.exit()`，他抛出`SystemExit exception`，达到退出线程的目的。

## 线程模块

Python通过两个标准库`thread`和`threading`提供对线程的支持。`thread`提供了低级别的、原始的线程以及一个简单的锁。`thread`模块提供的其他方法：

- `threading.currentThread()`: 返回当前的线程变量。
- `threading.enumerate()`: 返回一个包含正在运行的线程的list。正在运行指线程启动后、结束前，不包括启动前和终止后的线程。
- `threading.activeCount()`: 返回正在运行的线程数量，与`len(threading.enumerate())`有相同的结果。

除了使用方法外，线程模块同样提供了`Thread`类来处理线程，`Thread`类提供了以下方法：

- `run()`: 用以表示线程活动的方法。
- `start()`: 启动线程活动。
- `join([time])`: 等待至线程中止。这阻塞调用线程直至线程的`join()`方法被调用中止-正常退出或者抛出未处理的异常-或者是可选的超时发生。
- `isAlive()`: 返回线程是否活动的。
- `getName()`: 返回线程名。
- `setName()`: 设置线程名。

## 使用Threading模块创建线程

使用`Threading`模块创建线程，直接从`threading.Thread`继承，然后重写 `__init__` 方法和`run`方法：

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

import threading
import time

exitFlag = 0

class myThread (threading.Thread):    #继承父类threading.Thread
    def __init__(self, threadID, name, counter):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):                    #把要执行的代码写到run函数里面 线程在创建后会直接运行run函数
        print "Starting " + self.name
        print_time(self.name, self.counter, 5)
```

```

        print "Exiting " + self.name

def print_time(threadName, delay, counter):
    while counter:
        if exitFlag:
            thread.exit()
        time.sleep(delay)
        print "%s: %s" % (threadName, time.ctime(time.time()))
        counter -= 1

# 创建新线程
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# 开启线程
thread1.start()
thread2.start()

print "Exiting Main Thread"

```

以上程序执行结果如下：

```

Starting Thread-1
Starting Thread-2
Exiting Main Thread
Thread-1: Thu Mar 21 09:10:03 2013
Thread-1: Thu Mar 21 09:10:04 2013
Thread-2: Thu Mar 21 09:10:04 2013
Thread-1: Thu Mar 21 09:10:05 2013
Thread-1: Thu Mar 21 09:10:06 2013
Thread-2: Thu Mar 21 09:10:06 2013
Thread-1: Thu Mar 21 09:10:07 2013
Exiting Thread-1
Thread-2: Thu Mar 21 09:10:08 2013
Thread-2: Thu Mar 21 09:10:10 2013
Thread-2: Thu Mar 21 09:10:12 2013
Exiting Thread-2

```

## 线程同步

如果多个线程共同对某个数据修改，则可能出现不可预料的结果，为了保证数据的正确性，需要对多个线程进行同步。使用 **Thread** 对象的 **Lock** 和 **Rlock** 可以实现简单的线程同步，这两个对象都有 **acquire** 方法和 **release** 方法，对于那些需要每次只允许一个线程操作的数据，可以将其操作放到 **acquire** 和 **release** 方法之间。如下：多线程的优势在于可以同时运行多个任务（至少感觉起来是这样）。但是当线程需要共享数据时，可能存在数据不同步的问题。考虑这样一种情况：一个列表里所有元素都是0，线程 **"set"** 从后向前把所有元素改成1，而线程 **"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/python
# -*- coding: UTF-8 -*-

import threading
import time

class myThread (threading.Thread):
    def __init__(self, threadID, name, counter):

```

```

        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.counter = counter
    def run(self):
        print "Starting " + self.name
        # 获得锁，成功获得锁定后返回True
        # 可选的timeout参数不填时将一直阻塞直到获得锁定
        # 否则超时后将返回False
        threadLock.acquire()
        print_time(self.name, self.counter, 3)
        # 释放锁
        threadLock.release()

def print_time(threadName, delay, counter):
    while counter:
        time.sleep(delay)
        print "%s: %s" % (threadName, time.ctime(time.time()))
        counter -= 1

threadLock = threading.Lock()
threads = []

# 创建新线程
thread1 = myThread(1, "Thread-1", 1)
thread2 = myThread(2, "Thread-2", 2)

# 开启新线程
thread1.start()
thread2.start()

# 添加线程到线程列表
threads.append(thread1)
threads.append(thread2)

# 等待所有线程完成
for t in threads:
    t.join()
print "Exiting Main Thread"

```

## 线程优先级队列（ Queue ）

Python的Queue模块中提供了同步的、线程安全的队列类，包括FIFO（先入先出）队列Queue，LIFO（后入先出）队列LifoQueue，和优先级队列PriorityQueue。这些队列都实现了锁原语，能够在多线程中直接使用。可以使用队列来实现线程间的同步。 Queue模块中的常用方法:

- Queue.qsize() 返回队列的大小
- Queue.empty() 如果队列为空，返回True,反之False
- Queue.full() 如果队列满了，返回True,反之False
- Queue.full 与 maxsize 大小对应
- Queue.get([block[, timeout]])获取队列，timeout等待时间
- Queue.get\_nowait() 相当Queue.get(False)
- Queue.put(item) 写入队列，timeout等待时间
- Queue.put\_nowait(item) 相当Queue.put(item, False)
- Queue.task\_done() 在完成一项工作之后，Queue.task\_done()函数向任务已经完成的队列发送一个信号

- Queue.join() 实际上意味着等到队列为空，再执行别的操作

实例:

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

import Queue
import threading
import time

exitFlag = 0

class myThread (threading.Thread):
    def __init__(self, threadID, name, q):
        threading.Thread.__init__(self)
        self.threadID = threadID
        self.name = name
        self.q = q
    def run(self):
        print "Starting " + self.name
        process_data(self.name, self.q)
        print "Exiting " + self.name

def process_data(threadName, q):
    while not exitFlag:
        queueLock.acquire()
        if not workQueue.empty():
            data = q.get()
            queueLock.release()
            print "%s processing %s" % (threadName, data)
        else:
            queueLock.release()
            time.sleep(1)

threadList = ["Thread-1", "Thread-2", "Thread-3"]
nameList = ["One", "Two", "Three", "Four", "Five"]
queueLock = threading.Lock()
workQueue = Queue.Queue(10)
threads = []
threadID = 1

# 创建新线程
for tName in threadList:
    thread = myThread(threadID, tName, workQueue)
    thread.start()
    threads.append(thread)
    threadID += 1

# 填充队列
queueLock.acquire()
for word in nameList:
    workQueue.put(word)
queueLock.release()

# 等待队列清空
while not workQueue.empty():
    pass

# 通知线程是时候退出
exitFlag = 1

# 等待所有线程完成
```

```
for t in threads:  
    t.join()  
print "Exiting Main Thread"
```

以上程序执行结果:

```
Starting Thread-1  
Starting Thread-2  
Starting Thread-3  
Thread-1 processing One  
Thread-2 processing Two  
Thread-3 processing Three  
Thread-1 processing Four  
Thread-2 processing Five  
Exiting Thread-3  
Exiting Thread-1  
Exiting Thread-2  
Exiting Main Thread
```



# Python XML解析

## 什么是XML？

XML 指可扩展标记语言（eXtensible Markup Language）。你可以通过本站学习XML教程 XML 被设计用来传输和存储数据。XML是一套定义语义标记的规则，这些标记将文档分成许多部件并对这些部件加以标识。它也是元标记语言，即定义了用于定义其他与特定领域有关的、语义的、结构化的标记语言的句法语言。

## python对XML的解析

常见的XML编程接口有DOM和SAX，这两种接口处理XML文件的方式不同，当然使用场合也不同。python有三种方法解析XML，SAX，DOM，以及ElementTree：

### 1.SAX (simple API for XML )

python 标准库包含SAX解析器，SAX用事件驱动模型，通过在解析XML的过程中触发一个个的事件并调用用户定义的回调函数来处理XML文件。

### 2.DOM(Document Object Model)

将XML数据在内存中解析成一个树，通过对树的操作来操作XML。

### 3.ElementTree(元素树)

ElementTree就像一个轻量级的DOM，具有方便友好的API。代码可用性好，速度快，消耗内存少。注：因DOM需要将XML数据映射到内存中的树，一是比较慢，二是比较耗内存，而SAX流式读取XML文件，比较快，占用内存少，但需要用户实现回调函数（handler）。本章节使用到的XML实例文件movies.xml内容如下：

```
<collection shelf="New Arrivals">
<movie title="Enemy Behind">
  <type>War, Thriller</type>
  <format>DVD</format>
  <year>2003</year>
  <rating>PG</rating>
  <stars>10</stars>
  <description>Talk about a US-Japan war</description>
</movie>
<movie title="Transformers">
  <type>Anime, Science Fiction</type>
  <format>DVD</format>
  <year>1989</year>
  <rating>R</rating>
  <stars>8</stars>
  <description>A schientific fiction</description>
</movie>
  <movie title="Trigun">
    <type>Anime, Action</type>
    <format>DVD</format>
    <episodes>4</episodes>
    <rating>PG</rating>
    <stars>10</stars>
    <description>Vash the Stampede!</description>
  </movie>
  <movie title="Ishtar">
    <type>Comedy</type>
    <format>VHS</format>
```

```
<rating>PG</rating>
<stars>2</stars>
<description>Viewable boredom</description>
</movie>
</collection>
```

## python使用SAX解析xml

SAX是一种基于事件驱动的API。利用SAX解析XML文档牵涉到两个部分:解析器和事件处理器。解析器负责读取XML文档,并向事件处理器发送事件,如元素开始跟元素结束事件;而事件处理器则负责对事件作出相应,对传递的XML数据进行处理。

- 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/python
# -*- coding: UTF-8 -*-

import xml.sax

class MovieHandler( xml.sax.ContentHandler ):
    def __init__(self):
        self.CurrentData = ""
        self.type = ""
        self.format = ""
        self.year = ""
        self.rating = ""
        self.stars = ""
        self.description = ""

    # 元素开始事件处理
    def startElement(self, tag, attributes):
        self.CurrentData = tag
        if tag == "movie":
            print "*****Movie*****"
            title = attributes["title"]
            print "Title:", title

    # 元素结束事件处理
    def endElement(self, tag):
        if self.CurrentData == "type":
            print "Type:", self.type
        elif self.CurrentData == "format":
            print "Format:", self.format
        elif self.CurrentData == "year":
            print "Year:", self.year
        elif self.CurrentData == "rating":
            print "Rating:", self.rating
        elif self.CurrentData == "stars":
            print "Stars:", self.stars
        elif self.CurrentData == "description":
            print "Description:", self.description
        self.CurrentData = ""

    # 内容事件处理
    def characters(self, content):
        if self.CurrentData == "type":
            self.type = content
        elif self.CurrentData == "format":
            self.format = content
        elif self.CurrentData == "year":
            self.year = content
        elif self.CurrentData == "rating":
            self.rating = content
        elif self.CurrentData == "stars":
```

```

        self.stars = content
    elif self.CurrentData == "description":
        self.description = content

if ( __name__ == "__main__"):

    # 创建一个 XMLReader
    parser = xml.sax.make_parser()
    # turn off namepsaces
    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/python
```

```

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

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](#)。

# python GUI编程(Tkinter)

python提供了多个图形开发界面的库，几个常用Python GUI库如下：

- **Tkinter:** Tkinter模块("Tk 接口")是Python的标准Tk GUI工具包的接口.Tk和Tkinter可以在大多数的Unix平台下使用,同样可以应用在 Windows和Macintosh系统里.,Tk8.0的后续版本可以实现本地窗口风格,并良好地运行在绝大多数平台中。
- **wxPython:** wxPython 是一款开源软件, 是 Python 语言的一套优秀的 GUI 图形库, 允许 Python 程序员很方便的创建完整的、功能健全的 GUI 用户界面。
- **Jython:** Jython程序可以和Java无缝集成。除了一些标准模块, Jython使用Java的模块。Jython几乎拥有标准的Python中不依赖于C语言的全部模块。比如, Jython的用户界面将使用Swing, AWT或者SWT。Jython可以被动态或静态地编译成Java字节码。

## Tkinter 编程

Tkinter 是Python的标准GUI库。Python使用Tkinter可以快速的创建GUI应用程序。由于Tkinter是内置到python的安装包中、只要安装好Python之后就能import Tkinter库、而且IDLE也是用Tkinter编写而成、对于简单的图形界面Tkinter还是能应付自如。创建一个GUI程序

- 1、导入Tkinter模块
- 2、创建控件
- 3、指定这个控件的master，即这个控件属于哪一个
- 4、告诉GM(geometry manager)有一个控件产生了。

实例:

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

import Tkinter
top = Tkinter.Tk()
# 进入消息循环
top.mainloop()
```

以上代码执行结果如下图:



实例2:

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

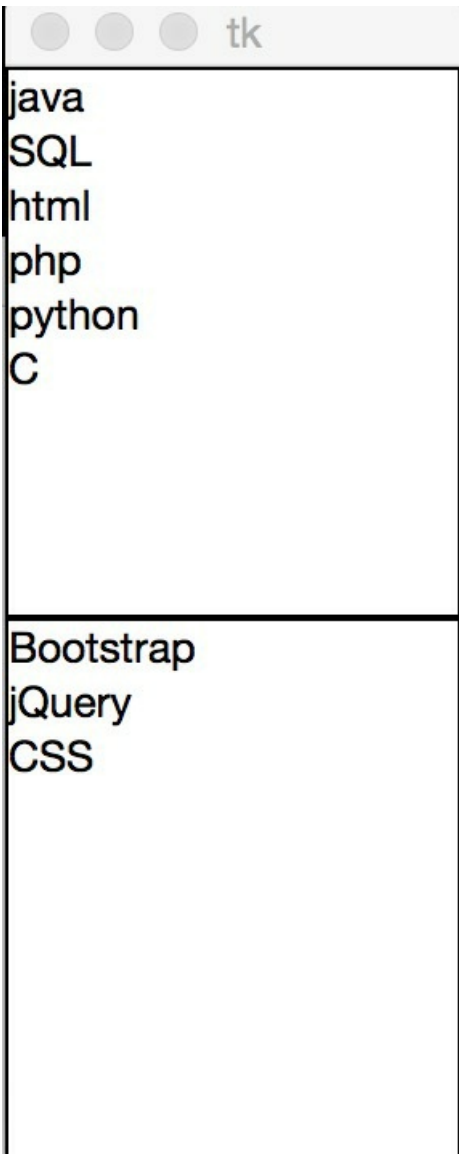
from Tkinter import *      # 导入 Tkinter 库
root = Tk()                # 创建窗口对象的背景色
                           # 创建两个列表
```

```
li      = ['C','python','php','html','SQL','java']
movie   = ['CSS','jQuery','Bootstrap']
listb   = Listbox(root)           # 创建两个列表组件
listb2  = Listbox(root)
for item in li:                   # 第一个小部件插入数据
    listb.insert(0,item)

for item in movie:               # 第二个小部件插入数据
    listb2.insert(0,item)

listb.pack()                    # 将小部件放置到主窗口中
listb2.pack()
root.mainloop()                # 进入消息循环
```

以上代码执行结果如下图:



## Tkinter 组件

Tkinter的提供各种控件，如按钮，标签和文本框，一个GUI应用程序中使用。这些控件通常被称为控件或者部件。目前有15种Tkinter的部件。我们提出这些部件以及一个简短的介绍，在下面的表:

控件	描述
Button	按钮控件；在程序中显示按钮。

Canvas	画布控件；显示图形元素如线条或文本
Checkbutton	多选框控件；用于在程序中提供多项选择框
Entry	输入控件；用于显示简单的文本内容
Frame	框架控件；在屏幕上显示一个矩形区域，多用来作为容器
Label	标签控件；可以显示文本和位图
Listbox	列表框控件；在 <b>Listbox</b> 窗口小部件是用来显示一个字符串列表给用户
Menubutton	菜单按钮控件，由于显示菜单项。
Menu	菜单控件；显示菜单栏,下拉菜单和弹出菜单
Message	消息控件；用来显示多行文本，与 <b>label</b> 比较类似
Radiobutton	单选按钮控件；显示一个单选的按钮状态
Scale	范围控件；显示一个数值刻度，为输出限定范围的数字区间
Scrollbar	滚动条控件，当内容超过可视化区域时使用，如列表框。.
Text	文本控件；用于显示多行文本
Toplevel	容器控件；用来提供一个单独的对话框，和 <b>Frame</b> 比较类似
Spinbox	输入控件；与 <b>Entry</b> 类似，但是可以指定输入范围值
PanedWindow	<b>PanedWindow</b> 是一个窗口布局管理的插件，可以包含一个或者多个子控件。
LabelFrame	<b>labelframe</b> 是一个简单的容器控件。常用与复杂的窗口布局。
tkMessageBox	用于显示你应用程序的消息框。

## 标准属性

标准属性也就是所有控件的共同属性，如大小，字体和颜色等等。

属性	描述
Dimension	控件大小；
Color	控件颜色；
Font	控件字体；
Anchor	锚点；
Relief	控件样式；
Bitmap	位图；
Cursor	光标；

## 几何管理

**Tkinter**控件有特定的几何状态管理方法，管理整个控件区域组织，一下是**Tkinter**公开的几何管理类：包、网格、位置

几何方法	描述
pack()	包装；
grid()	网格；
place()	位置；



## Python2.x与3.x版本区别

Python的3.0版本，常被称为Python 3000，或简称Py3k。相对于Python的早期版本，这是一个较大的升级。为了不带入过多的累赘，Python 3.0在设计的时候没有考虑向下相容。许多针对早期Python版本设计的程式都无法在Python 3.0上正常执行。为了照顾现有程式，Python 2.6作为一个过渡版本，基本使用了Python 2.x的语法和库，同时考虑了向Python 3.0的迁移，允许使用部分Python 3.0的语法与函数。新的Python程式建议使用Python 3.0版本的语法。除非执行环境无法安装Python 3.0或者程式本身使用了不支援Python 3.0的第三方库。目前不支援Python 3.0的第三方库有Twisted, py2exe, PIL等。大多数第三方库都正在努力地相容Python 3.0版本。即使无法立即使用Python 3.0，也建议编写相容Python 3.0版本的程式，然后使用Python 2.6, Python 2.7来执行。Python 3.0的变化主要在以下几个方面：

### print 函数

print语句没有了，取而代之的是print()函数。Python 2.6与Python 2.7部分地支持这种形式的print语法。在Python 2.6与Python 2.7里面，以下三种形式是等价的：

```
print "fish"
print ("fish") #注意print后面有个空格
print("fish") #print()不能带有任何其它参数
```

然而，Python 2.6实际已经支持新的print()语法：

```
from __future__ import print_function
print("fish", "panda", sep=', ')
```

### Unicode

Python 2 有 ASCII str() 类型，unicode() 是单独的，不是 byte 类型。现在，在 Python 3，我们最终有了 Unicode (utf-8) 字符串，以及一个字节类：byte 和 bytearray。由于 Python3.X 源码文件默认使用utf-8编码，这就使得以下代码是合法的：

```
>>> 中国 = 'china'
>>> print(中国)
china
```

#### Python 2.x

```
>>> str = "我爱北京天安门"
>>> str
'\xe6\x88\x91\xe7\x88\xb1\xe5\x8c\x97\xe4\xba\xac\xe5\xa4\xa9\xe5\xae\x89\xe9\x97\xa8'
>>> str = u"我爱北京天安门"
>>> str
u'\u6211\u7231\u5317\u4eac\u5929\u5b89\u95e8'
```

#### Python 3.x

```
>>> str = "我爱北京天安门"
>>> str
'我爱北京天安门'
```

### 除法运算

Python中的除法较其它语言显得非常高端，有套很复杂的规则。Python中的除法有两个运算符，/和// 首先来说/除法: 在python 2.x中/除法就跟我们熟悉的大多数语言，比如Java啊C啊差不多，整数相除的结果是一个整数，把小数部分完全忽略掉，浮点数除法会保留小数点的部分得到一个浮点数的结果。在python 3.x中/除法不再这么做了，对于整数之间的相除，结果也会是浮点数。Python 2.x:

```
>>> 1 / 2
0
>>> 1.0 / 2.0
0.5
```

Python 3.x:

```
>>> 1/2
0.5
```

而对于//除法，这种除法叫做floor除法，会对除法的结果自动进行一个floor操作，在python 2.x和python 3.x中是一致的。python 2.x:

```
>>> -1 // 2
-1
```

python 3.x:

```
>>> -1 // 2
-1
```

注意的是并不是舍弃小数部分，而是执行floor操作，如果要截取小数部分，那么需要使用math模块的trunc函数 python 3.x:

```
>>> import math
>>> math.trunc(1 / 2)
0
>>> math.trunc(-1 / 2)
0
```

## 异常

在 Python 3 中处理异常也轻微的改变，在 Python 3 中我们现在使用 as 作为关键词。捕获异常的语法由 except exc, var 改为 except exc as var。使用语法except (exc1, exc2) as var可以同时捕获多种类别的异常。Python 2.6已经支持这两种语法。

- 1. 在2.x时代，所有类型的对象都是可以直接被抛出的，在3.x时代，只有继承自BaseException的对象才可以被抛出。
- 1. 2.x raise语句使用逗号将抛出对象类型和参数分开，3.x取消了这种奇葩的写法，直接调用构造函数抛出对象即可。

在2.x时代，异常在代码中除了表示程序错误，还经常做一些普通控制结构应该做的事情，在3.x中可以看出，设计者让异常变的更加专一，只有在错误发生的情况才能去用异常捕获语句来处理。

## xrange

在 Python 2 中 xrange() 创建迭代对象的用法是非常流行的。比如：for 循环或者是列表/集合/字典推导式。这个表现十分像生成器（比如。“惰性求值”）。但是这个 xrange-iterable 是无穷的，意味着你可以无限遍历。由于它的惰性求值，如果你不得不仅仅不遍历它一次，xrange() 函数比 range() 更快（比如 for 循环）。尽管如此，对比迭代一次，不建议你重复迭代多次，因为生成器每次都从头开始。在 Python 3 中，range() 是像 xrange() 那样实现以至于一个专门的 xrange() 函数都不再存在（在 Python 3 中 xrange() 会抛出命名异常）。

```
import timeit

n = 10000
def test_range(n):
    return for i in range(n):
        pass

def test_xrange(n):
    for i in xrange(n):
        pass
```

## Python 2

```
print 'Python', python_version()

print '\ntiming range()'
%timeit test_range(n)

print '\n\ntiming xrange()'
%timeit test_xrange(n)

Python 2.7.6

timing range()
1000 loops, best of 3: 433 µs per loop

timing xrange()
1000 loops, best of 3: 350 µs per loop
```

## Python 3

```
print('Python', python_version())

print('\ntiming range()')
%timeit test_range(n)

Python 3.4.1

timing range()
1000 loops, best of 3: 520 µs per loop
```

```
print(xrange(10))
-----
NameError                                Traceback (most recent call last)
<ipython-input-5-5d8f9b79ea70> in <module>()
----> 1 print(xrange(10))

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

## 八进制字面量表示

八进制数必须写成**0o777**，原来的形式**0777**不能用了；二进制必须写成**0b111**。新增了一个**bin()**函数用于将一个整数转换成二进制字符串。Python 2.6已经支持这两种语法。在Python 3.x中，表示八进制字面量的方式只有一种，就是**0o1000**。python 2.x

```
>>> 0o1000
512
>>> 01000
```

python 3.x

```
>>> 01000
File "<stdin>", line 1
    01000
      ^
SyntaxError: invalid token
>>> 0o1000
512
```

## 不等运算符

Python 2.x中不等于有两种写法 `!=` 和 `<>` Python 3.x中去掉了`<>`, 只有`!=`一种写法, 还好, 我从来没有使用`<>`的习惯

## 去掉了`repr`表达式``

Python 2.x 中反引号``相当于`repr`函数的作用 Python 3.x 中去掉了这种写法, 只允许使用`repr`函数, 这样做的目的是为了使代码看上去更清晰么? 不过我感觉用`repr`的机会很少, 一般只在`debug`的时候才用, 多数时候还是用`str`函数来用字符串描述对象。

```
def sendMail(from_: str, to: str, title: str, body: str) -> bool:
    pass
```

## 多个模块被改名（根据PEP8）

旧的名字	新的名字
<code>_winreg</code>	<code>winreg</code>
<code>ConfigParser</code>	<code>configparser</code>
<code>copy_reg</code>	<code>copyreg</code>
<code>Queue</code>	<code>queue</code>
<code>SocketServer</code>	<code>socketserver</code>
<code>repr</code>	<code>reprlib</code>

`StringIO`模块现在被合并到新的`io`模组内。 `new`, `md5`, `gopherlib`等模块被删除。 Python 2.6已经支援新的`io`模组。 `httplib`, `BaseHTTPServer`, `CGIHTTPServer`, `SimpleHTTPServer`, `Cookie`, `cookielib`被合并到`http`包内。 取消了`exec`语句, 只剩下`exec()`函数。 Python 2.6已经支援`exec()`函数。

## 数据类型

1) Py3.X去除了`long`类型, 现在只有一种整型——`int`, 但它的行为就像2.X版本的`long` 2) 新增了`bytes`类型, 对应于2.X版本的八位串, 定义一个`bytes`字面量的方法如下:

```
>>> b = b'china'
>>> type(b)
<type 'bytes'>
```

`str`对象和`bytes`对象可以使用`.encode()` (`str -> bytes`) or `.decode()` (`bytes -> str`)方法相互转化。

```
>>> s = b.decode()
>>> s
'china'
```

```
>>> b1 = s.encode()
>>> b1
b'china'
```

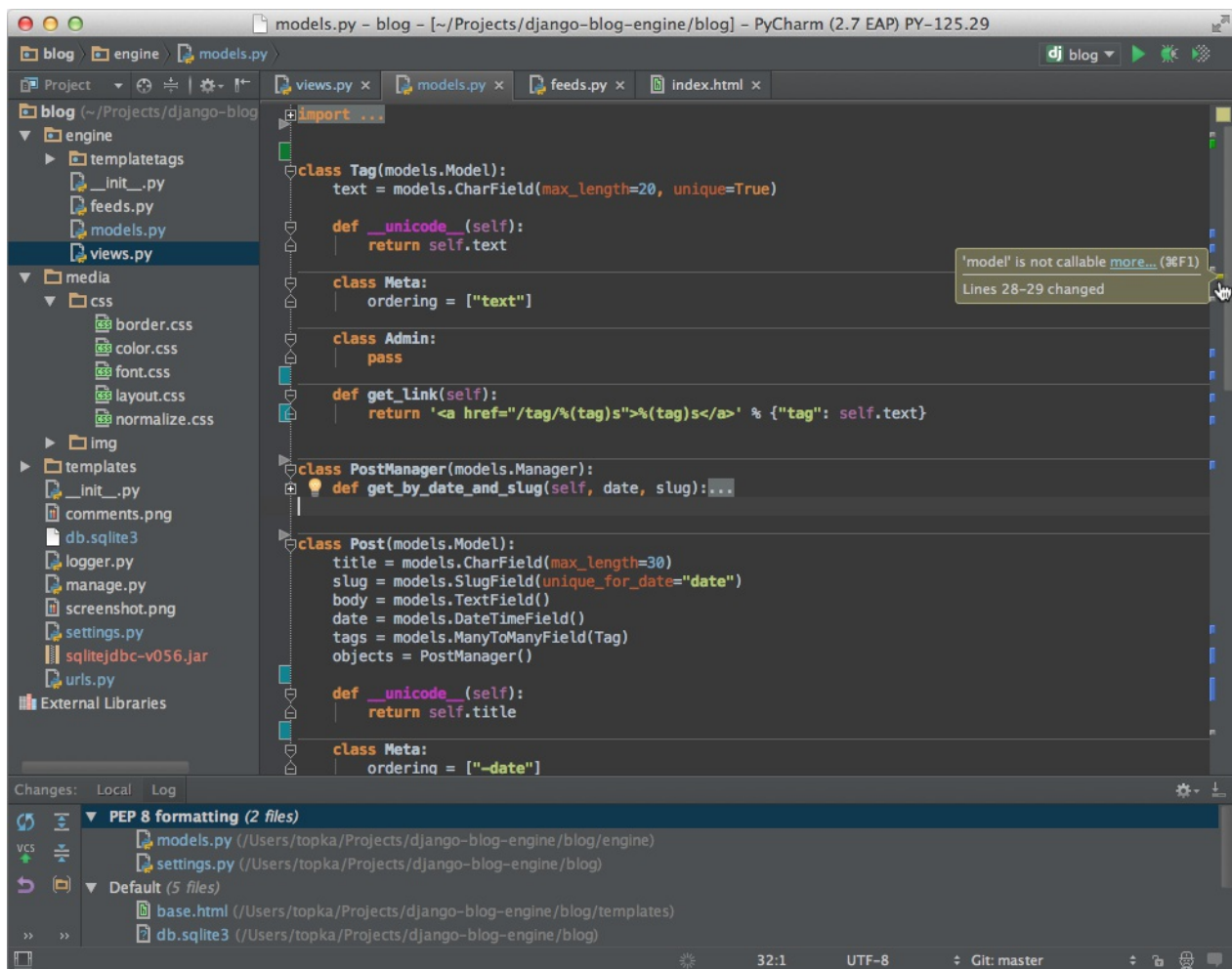
3) dict的`.keys()`、`.items` 和`.values()`方法返回迭代器，而之前的`iterkeys()`等函数都被废弃。同时去掉的还有 `dict.has_key()`，用 `in` 替代它吧。

# Python IDE

本文为大家推荐几款不错的 Python IDE（集成开发环境），比较推荐 PyCharm，当然你可以根据自己的喜好来选择适合自己的 Python IDE。

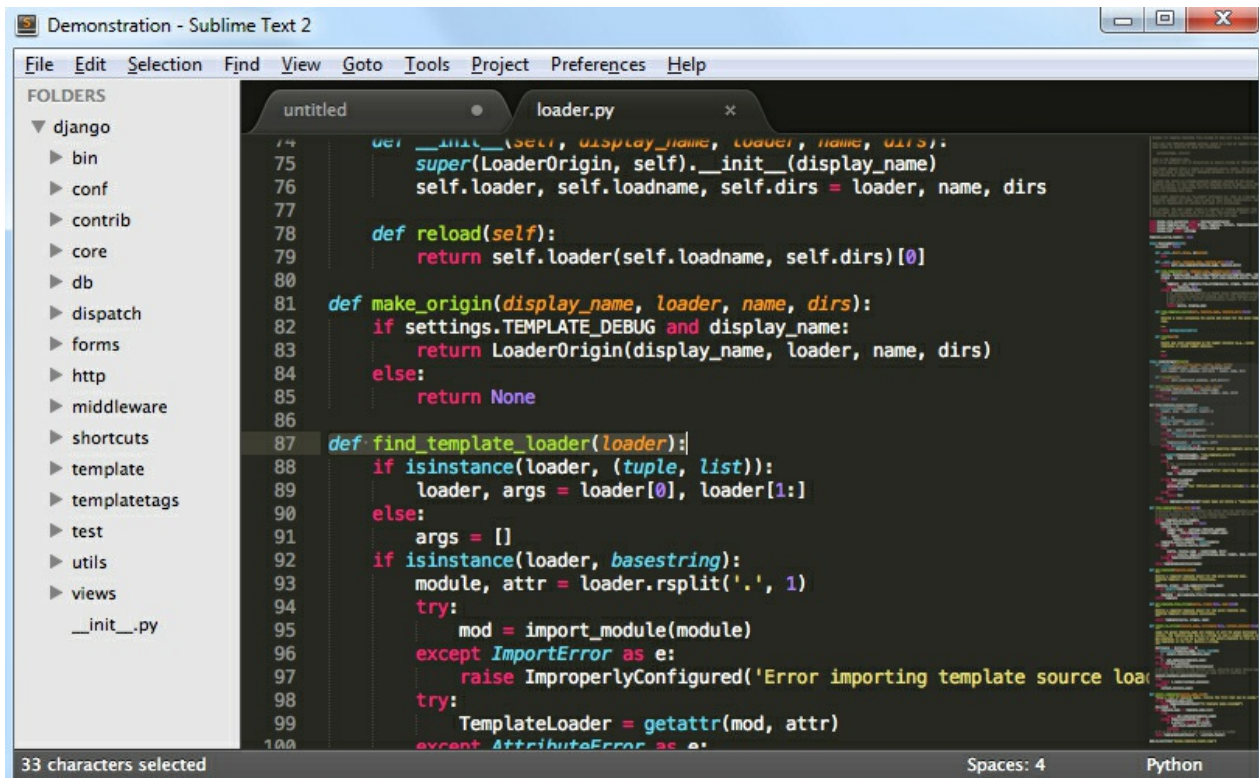
## PyCharm

PyCharm 是由 JetBrains 打造的一款 Python IDE。PyCharm 具备一般 Python IDE 的功能，比如：调试、语法高亮、项目管理、代码跳转、智能提示、自动完成、单元测试、版本控制等。另外，PyCharm 还提供了一些很好的功能用于 Django 开发，同时支持 Google App Engine，更酷的是，PyCharm 支持 IronPython。PyCharm 官方下载地址：<http://www.jetbrains.com/pycharm/download/> 效果图查看：



## Sublime Text

Sublime Text 具有漂亮的用户界面和强大的功能，例如代码缩略图，Python 的插件，代码段等。还可自定义键绑定，菜单和工具栏。Sublime Text 的主要功能包括：拼写检查，书签，完整的 Python API，Goto 功能，即时项目切换，多选择，多窗口等等。Sublime Text 是一个跨平台的编辑器，同时支持 Windows、Linux、Mac OS X 等操作系统。



使用Sublime Text 2的插件扩展功能，你可以轻松的打造一款不错的 Python IDE，以下推荐几款插件（你可以找到更多）：

- CodeIntel: 自动补全+成员/方法提示（强烈推荐）
- SublimeREPL: 用于运行和调试一些需要交互的程序（E.G. 使用了Input()的程序）
- Bracket Highlighter: 括号匹配及高亮
- SublimeLinter: 代码pep8格式检查

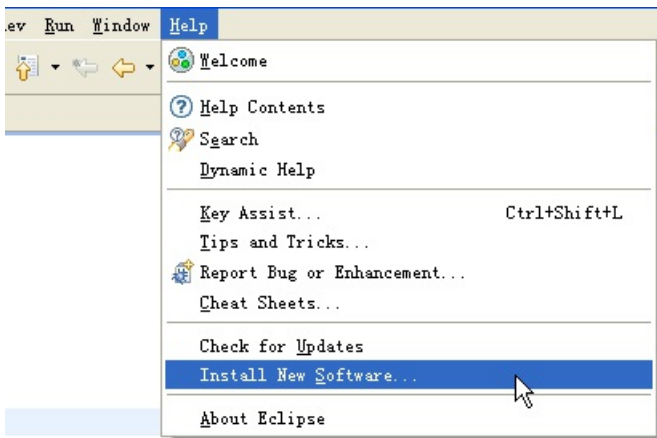
## Eclipse+Pydev

### 1、安装Eclipse

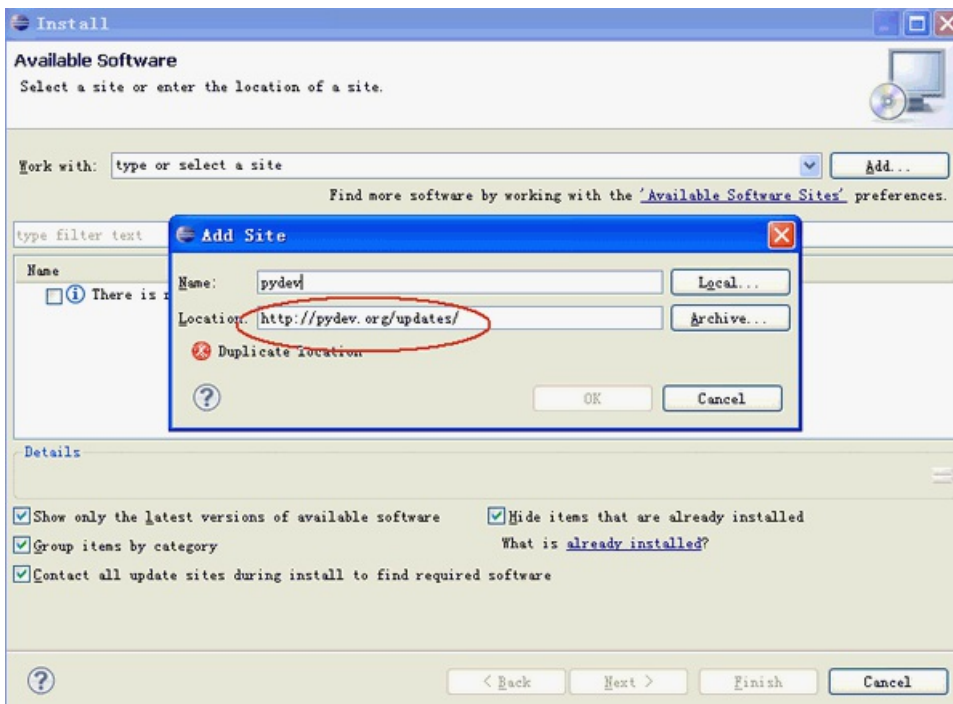
Eclipse可以在它的官方网站[Eclipse.org](http://eclipse.org)找到并下载，通常我们可以选择适合自己的Eclipse版本，比如Eclipse Classic。下载完成后解压到你想安装的目录中即可。当然在执行Eclipse之前，你必须确认安装了Java运行环境,即必须安装JRE或JDK，你可以到（<http://www.java.com/en/download/manual.jsp>）找到JRE下载并安装。

### 2、安装Pydev

运行Eclipse之后，选择help-->Install new Software，如下图所示。

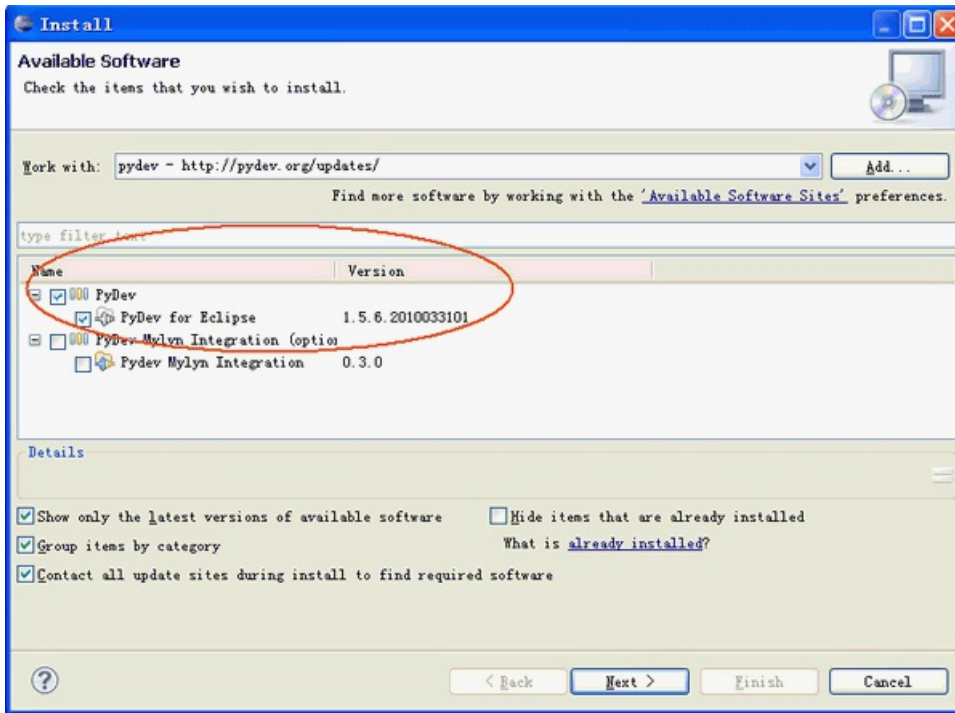


点击Add，添加pydev的安装地址：<http://pydev.org/updates/>，如下图所示。



完成后点击"ok"，接着点击PyDev的"+", 展开PyDev的节点，要等一小段时间，让它从网上获取PyDev的相关套件，当完成后会多出PyDev的相关套件在子节点里，勾选它们然后按next进行安装。如下图所示。

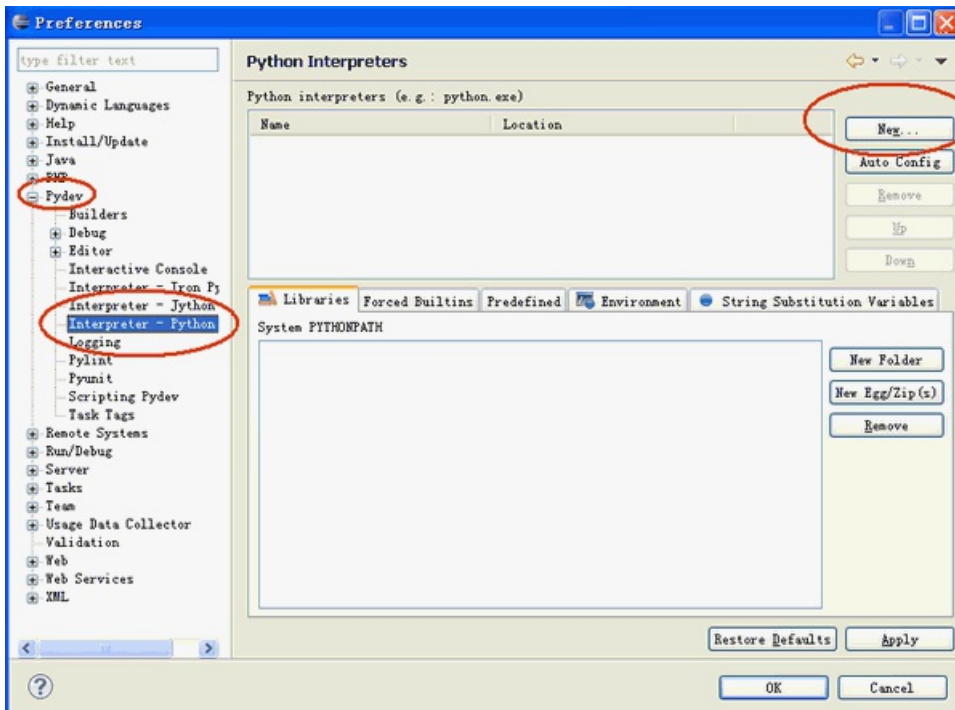




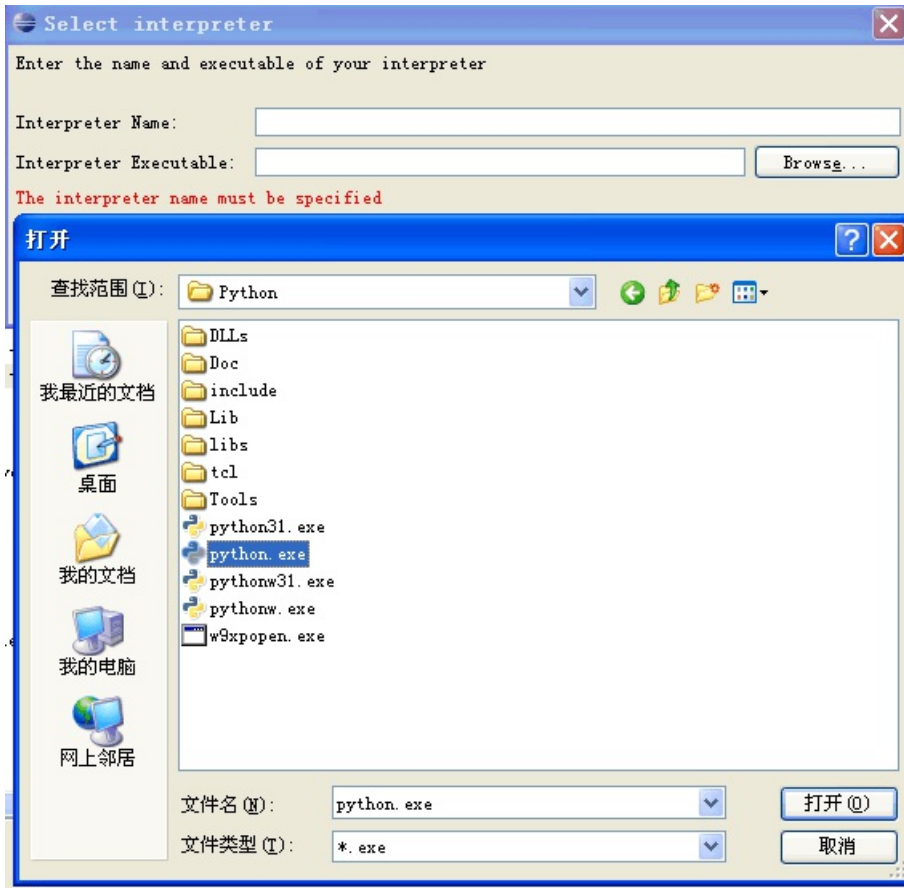
安装完成后，重启Eclipse即可

### 3、设置Pydev

安装完成后，还需要设置一下PyDev，选择Window -> Preferences来设置PyDev。设置Python的路径，从Pydev的Interpreter - Python页面选择New



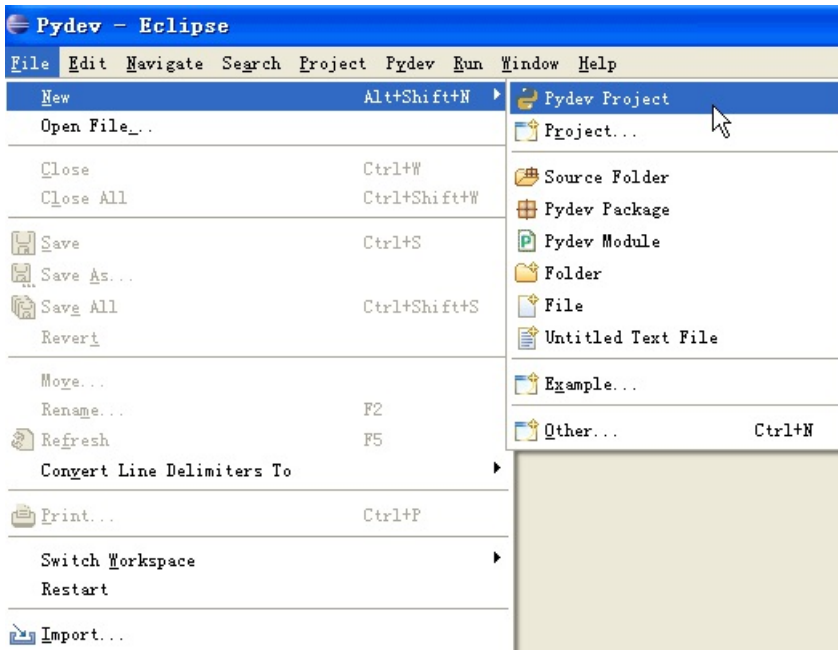
会弹出一个窗口让你选择Python的安装位置，选择你安装Python的所在位置。



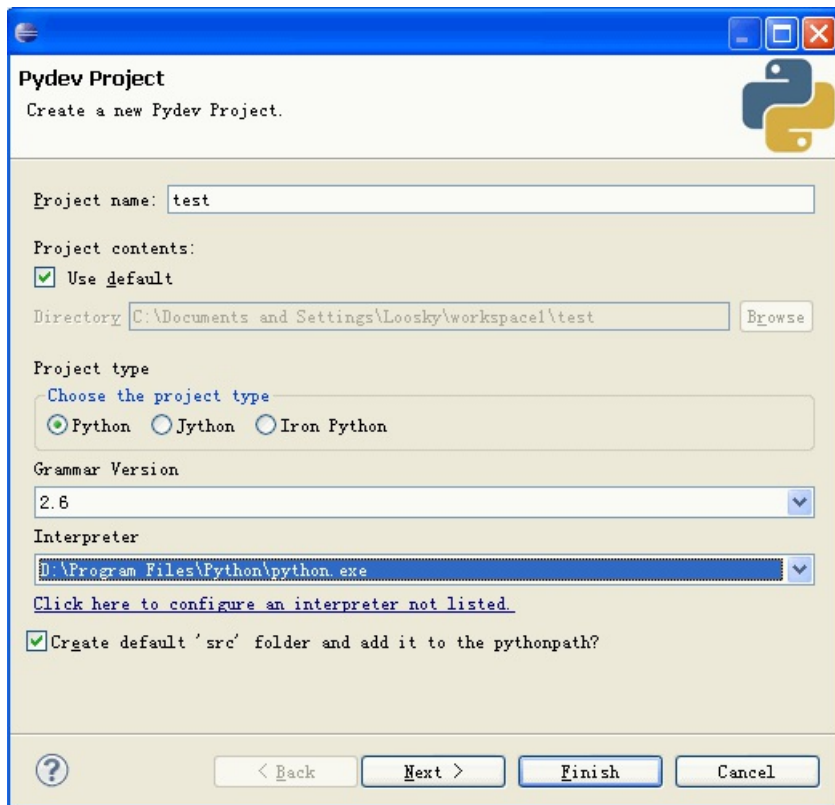
完成之后PyDev就设置完成，可以开始使用。

## 4、建立Python Project:

安装好Eclipse+PyDev以后，我们就可以开始使用它来开发项目了。首先要创建一个项目，选择File -> New -> Pydev Project

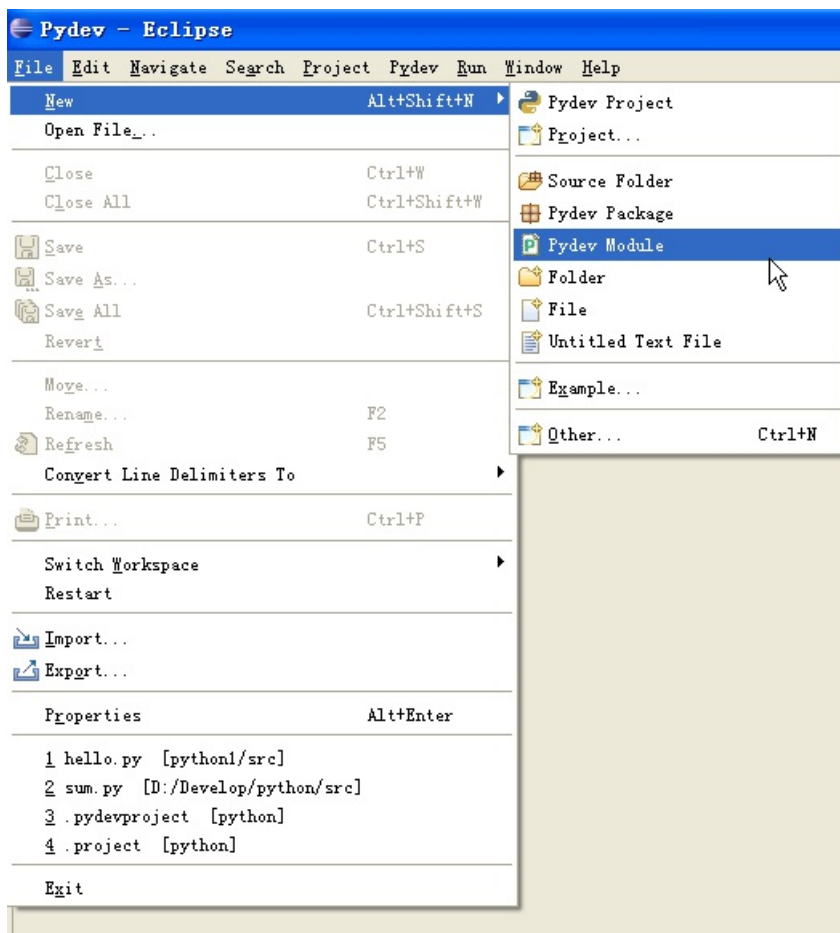


会弹出一个新窗口，填写Project Name，以及项目保存地址，然后点击next完成项目的创建。

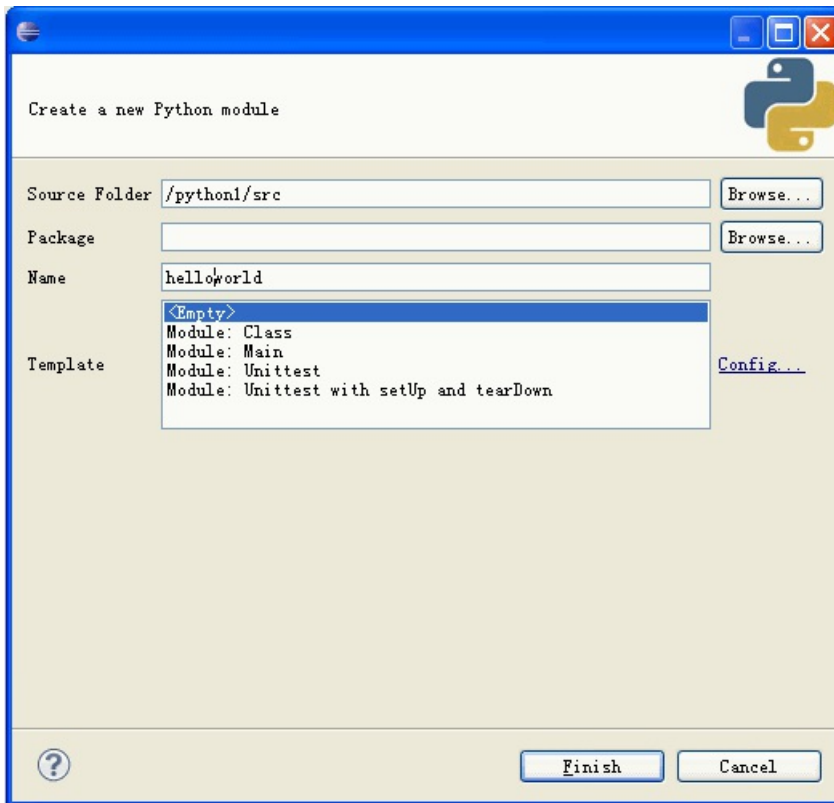


## 5、创建新的Pydev Module

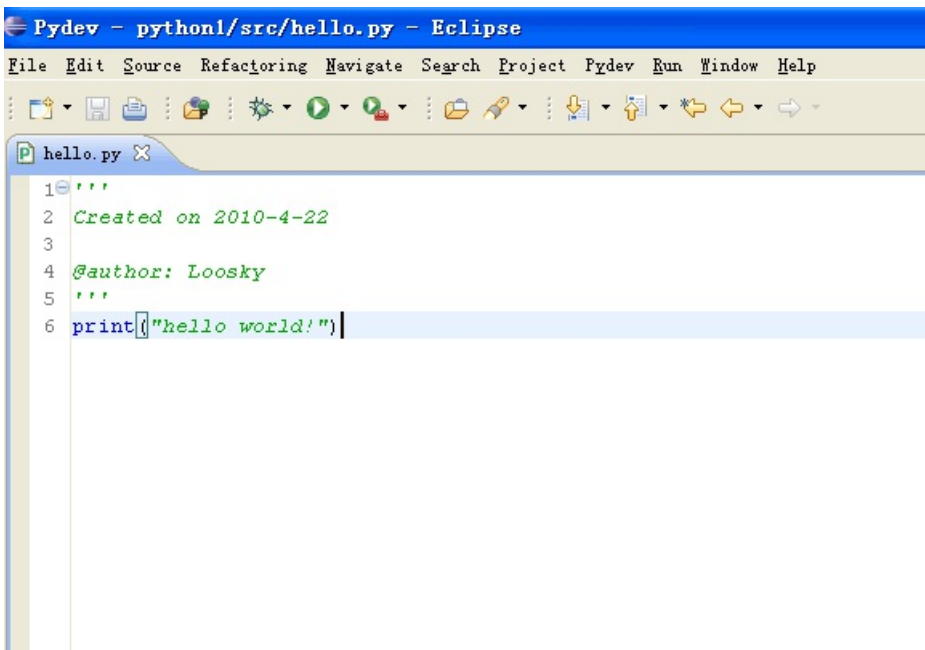
光有项目是无法执行的，接着必须创建新的Pydev Moudle，选择File -> New -> Pydev Module



在弹出的窗口中选择文件存放位置以及Moudle Name，注意Name不用加.py，它会自动帮助我们添加。然后点击Finish完成创建。

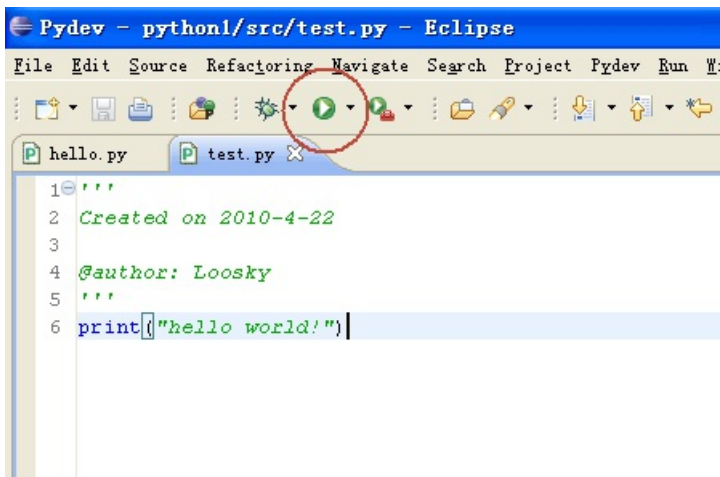


输入"hello world"的代码。

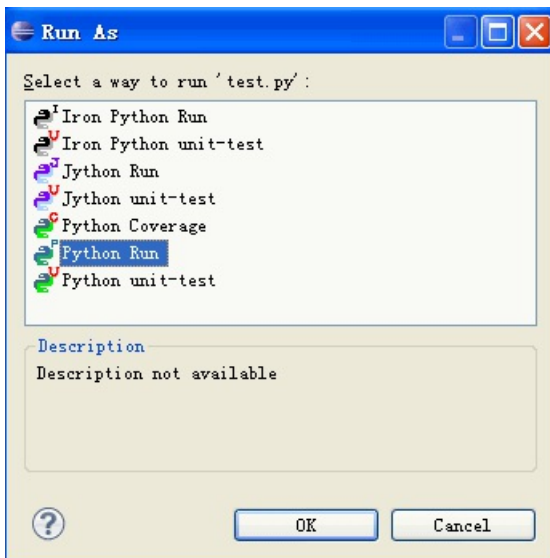


## 6、执行程序

程序写完后，我们可以开始执行程序,在上方的工具栏上面找到执行的按钮。



之后会弹出一个让你选择执行方式的窗口，通常我们选择Python Run，开始执行程序。



## 更多 Python IDE

推荐10款最好的 Python IDE: <http://www.runoob.com/w3cnote/best-python-ide-for-developers.html> 当然还有非常多很棒的 Python IDE，你可以自由的选择，更多 Python IDE 请参阅: <http://wiki.python.org/moin/PythonEditors>

# Python JSON

本章节我们将为大家介绍如何使用 Python 语言来编码和解码 JSON 对象。JSON(JavaScript Object Notation) 是一种轻量级的数据交换格式，易于人阅读和编写。

## JSON 函数

使用 JSON 函数需要导入 json 库：import json。

函数	描述
json.dumps	将 Python 对象编码成 JSON 字符串
json.loads	将已编码的 JSON 字符串解码为 Python 对象

## json.dumps

json.dumps 用于将 Python 对象编码成 JSON 字符串。

语法

```
json.dumps(obj, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None,
```

实例

以下实例将数组编码为 JSON 格式数据：

```
#!/usr/bin/python
import json

data = [ { 'a' : 1, 'b' : 2, 'c' : 3, 'd' : 4, 'e' : 5 } ]

json = json.dumps(data)
print json
```

以上代码执行结果为：

```
[{"a": 1, "c": 3, "b": 2, "e": 5, "d": 4}]
```

使用参数让 JSON 数据格式化输出：

```
>>> import json
>>> print json.dumps({'a': 'Runoob', 'b': 7}, sort_keys=True, indent=4, separators=(',', ': '))
{
    "a": "Runoob",
    "b": 7
}
```

python 原始类型向 json 类型的转化对照表：

Python	JSON
dict	object
list, tuple	array
str, unicode	string

int, long, float	number
True	true
False	false
None	null

## json.loads

json.loads 用于解码 JSON 数据。该函数返回 Python 字段的数据类型。

语法

```
json.loads(s[, encoding[, cls[, object_hook[, parse_float[, parse_int[, parse_constant[, object_pairs_hook[, *args]]]]]]])
```

实例

以下实例展示了Python 如何解码 JSON 对象：

```
#!/usr/bin/python
import json

jsonData = '{"a":1,"b":2,"c":3,"d":4,"e":5}';

text = json.loads(jsonData)
print text
```

以上代码执行结果为：

```
{u'a': 1, u'c': 3, u'b': 2, u'e': 5, u'd': 4}
```

json 类型转换到 python 的类型对照表：

JSON	Python
object	dict
array	list
string	unicode
number (int)	int, long
number (real)	float
true	True
false	False
null	None

更多内容参考：<https://docs.python.org/2/library/json.html>。

## 使用第三方库：Demjson

Demjson 是 python 的第三方模块库，可用于编码和解码 JSON 数据，包含了 JSONLint 的格式化及校验功能。Github 地址：<https://github.com/dmeranda/demjson> 官方地址：<http://deron.meranda.us/python/demjson/>

## 环境配置

在使用 Demjson 编码或解码 JSON 数据前，我们需要先安装 Demjson 模块。本教程我们会下载 Demjson 并安装：

```
$ tar -xvzf demjson-2.2.3.tar.gz
$ cd demjson-2.2.3
$ python setup.py install
```

更多安装介绍查看: <http://deron.meranda.us/python/demjson/install>

## JSON 函数

函数	描述
encode	将 Python 对象编码成 JSON 字符串
decode	将已编码的 JSON 字符串解码为 Python 对象

### encode

Python `encode()` 函数用于将 Python 对象编码成 JSON 字符串。

语法

```
demjson.encode(self, obj, nest_level=0)
```

实例

以下实例将数组编码为 JSON 格式数据:

```
#!/usr/bin/python
import demjson

data = [ { 'a' : 1, 'b' : 2, 'c' : 3, 'd' : 4, 'e' : 5 } ]

json = demjson.encode(data)
print json
```

以上代码执行结果为:

```
[{"a":1,"b":2,"c":3,"d":4,"e":5}]
```

### decode

Python 可以使用 `demjson.decode()` 函数解码 JSON 数据。该函数返回 Python 字段的数据类型。

语法

```
demjson.decode(self, txt)
```

实例

以下实例展示了 Python 如何解码 JSON 对象:

```
#!/usr/bin/python
import demjson

json = '{"a":1,"b":2,"c":3,"d":4,"e":5}';

text = demjson.decode(json)
print text
```



以上代码执行结果为：

```
{u'a': 1, u'c': 3, u'b': 2, u'e': 5, u'd': 4}
```

# Python 100例

以下实例在Python2.7下测试通过：

- [Python 练习实例1 4个不重复数字](#)
- [Python 练习实例2 企业奖金发放](#)
- [Python 练习实例3 正整数开方](#)
- [Python 练习实例4 日期判断](#)
- [Python 练习实例5 3个数字排序](#)
- [Python 练习实例6 斐波那契数列](#)
- [Python 练习实例7 列表复制](#)
- [Python 练习实例8 九九乘法表](#)
- [Python 练习实例9 暂停一秒输出](#)
- [Python 练习实例10 格式化当前时间](#)
- [Python 练习实例11 兔子问题](#)
- [Python 练习实例12 200以内素数](#)
- [Python 练习实例13 水仙花数](#)
- [Python 练习实例14 正整数分解质因数](#)
- [Python 练习实例15 条件运算符](#)
- [Python 练习实例16 输出指定格式的日期](#)
- [Python 练习实例17 统计字符串](#)
- [Python 练习实例18 求 \$s=a+aa+...+aa...a\$ 的值](#)
- [Python 练习实例19 1000以内完数](#)
- [Python 练习实例20 球体反弹](#)
- [Python 练习实例21 猴子吃桃问题](#)
- [Python 练习实例22 乒乓球抽签](#)
- [Python 练习实例23 打印菱形](#)
- [Python 练习实例24 分数序列求和](#)
- [Python 练习实例25 累乘](#)
- [Python 练习实例26 递归阶乘](#)
- [Python 练习实例27 递归排序](#)
- [Python 练习实例28 岁数问题](#)
- [Python 练习实例29 判断位数](#)
- [Python 练习实例30 回文数](#)
- [Python 练习实例31 星期判断](#)
- [Python 练习实例32 倒序输出](#)
- [Python 练习实例33 分隔列表](#)
- [Python 练习实例34 函数调用](#)
- [Python 练习实例35 文本颜色设置](#)
- [Python 练习实例36 100内的素数](#)
- [Python 练习实例37 数字排序](#)
- [Python 练习实例38 矩阵对角线元素之和](#)
- [Python 练习实例39 数组添加元素](#)
- [Python 练习实例40 数组逆序输出](#)
- [Python 练习实例41 静态变量](#)
- [Python 练习实例42 auto定义变量](#)
- [Python 练习实例43 python作用域](#)
- [Python 练习实例44 两个矩阵相加](#)
- [Python 练习实例45  \$1+...+100\$ 求和](#)
- [Python 练习实例46 数字的平方](#)
- [Python 练习实例47 两个变量值互换](#)
- [Python 练习实例48 数字比较](#)
- [Python 练习实例49 匿名函数](#)
- [Python 练习实例50 随机数](#)

- [Python 练习实例51 位运算与](#)
- [Python 练习实例52 位运算非](#)
- [Python 练习实例53 位运算或](#)
- [Python 练习实例54 整数截取](#)
- [Python 练习实例55 按位取反](#)
- [Python 练习实例56 circle画圆形](#)
- [Python 练习实例57 line画直线](#)
- [Python 练习实例58 rectangle画方形](#)
- [Python 练习实例59 画图综合例子](#)
- [Python 练习实例60 计算字符串长度](#)
- [Python 练习实例61 杨辉三角形](#)
- [Python 练习实例62 查找字符串](#)
- [Python 练习实例63 画椭圆ellipse](#)
- [Python 练习实例64 画图](#)
- [Python 练习实例65 一个最优美的图案](#)
- [Python 练习实例66 按大小顺序输出](#)
- [Python 练习实例67 数组元素交换](#)
- [Python 练习实例68 整数顺序移动](#)
- [Python 练习实例69 报数问题](#)
- [Python 练习实例70 计算字符串长度](#)
- [Python 练习实例71 输入输出](#)
- [Python 练习实例72 创建一个链表](#)
- [Python 练习实例73 反向输出一个链表](#)
- [Python 练习实例74 连接两个链表](#)
- [Python 练习实例75 简单的题目](#)
- [Python 练习实例76 指针函数](#)
- [Python 练习实例77 循环输出列表](#)
- [Python 练习实例78 找到年龄最大的人](#)
- [Python 练习实例79 字符串排序](#)
- [Python 练习实例80 桃子问题](#)
- [Python 练习实例81 数学问题](#)
- [Python 练习实例82 八进制转换为十进制](#)
- [Python 练习实例83 奇数组合](#)
- [Python 练习实例84 连接字符串](#)
- [Python 练习实例85 判断素数被9整除](#)
- [Python 练习实例86 两个字符串连接](#)
- [Python 练习实例87 结构体变量传递](#)
- [Python 练习实例88 数学计算](#)
- [Python 练习实例89 数据传递加密](#)
- [Python 练习实例90 列表使用实例](#)
- [Python 练习实例91 时间函数举例1](#)
- [Python 练习实例92 时间函数举例2](#)
- [Python 练习实例93 时间函数举例3](#)
- [Python 练习实例94 时间函数举例4](#)
- [Python 练习实例95 日期格式转换](#)
- [Python 练习实例96 字符串计算](#)
- [Python 练习实例97 键盘输入](#)
- [Python 练习实例98 大小写转换](#)
- [Python 练习实例99 文件操作](#)
- [Python 练习实例100 列表转换为字典](#)