创建TCP服务端

from socket import \*

from time import ctime

HOST = ''

PORT = 21567

BUFFER = 1024

ADDR = (HOST, PORT)

tcpSerSock = socket(AF\_INET, SOCK\_STREAM) #创建服务器套接字

tcpSerSock.bind(ADDR) #套接字与端口绑定

tcpSerSock.listen(5) #监听

while True: #服务器无限循环

print 'Waiting for connection...'

tcpCliSock, addr = tcpSerSock.accept() #服务器接受连接

print "...connected from:", addr

while True: #通信循环

data = tcpCliSock.recv(BUFFER) 接受对话

if not data:

break

tcpCliSock.send('[%s] %s' % (ctime(), data)) 发送对话

tcpCliSock.close()

tcpSerSock.close()

创建TCP客户端

**from** socket **import** \*  
  
HOST = **'localhost'**PORT = 21567  
BUFSIZ = 1024  
ADDR = (HOST, PORT)  
  
tcpCliSock = socket(AF\_INET, SOCK\_STREAM) #创建客户端套接字  
tcpCliSock.connect(ADDR) #尝试连接服务器  
  
**while** True: #通信循环  
 data = raw\_input(**'> '**)  
 **if not** data:  
 **break** tcpCliSock.send(data) #对话  
 data = tcpCliSock.recv(BUFSIZ)  
 **if not** data:  
 **break  
 print** data  
tcpCliSock.close()

Python面向对象指南：

抽象是隐藏多余细节的艺术。在面向对象的概念中，抽象的直接表现形式通常为类。虽然Python是解释性语言，但是它是面向对象的，从设计之初就已经是一门面向对象的语言。Python基本上提供了面向对象编程语言的所有元素，如果你已经至少掌握了一门面向对象语言，那么利用Python进行面向对象程序设计将会相当容易。下面就来了解一下如何在Python中进行对象编程。

**一. 如何定义一个类**

　　在进行python面向对象编程之前，**先来了解几个术语：类，类对象，实例对象，属性，函数和方法。**

　　类是对现实世界中一些事物的封装，定义一个类可以采用下面的方式来定义：

1. **class** className:
2. block

　　注意类名后面有个冒号，在block块里面就可以定义属性和方法了。当一个类定义完之后，就产生了一个类对象。类对象支持两种操作：引用和实例化。引用操作是通过类对象去调用类中的属性或者方法，而实例化是产生出一个类对象的实例，称作实例对象。比如定义了一个people类：

1. **class** people:
2. name = 'jack'       #定义了一个属性
3. #定义了一个方法
4. **def** printName(self):
5. **print** self.name

　　people类定义完成之后就产生了一个全局的类对象，可以通过类对象来访问类中的属性和方法了。当通过people.name（至于为什么可以直接这样访问属性后面再解释，这里只要理解类对象这个概念就行了）来访问时，people.name中的people称为类对象，这点和C++中的有所不同。当然还可以进行实例化操作，p=people( )，这样就产生了一个people的实例对象，此时也可以通过实例对象p来访问属性或者方法了(p.name).

　　理解了类、类对象和实例对象的区别之后，我们来了解一下Python中属性、方法和函数的区别。

　　在上面代码中注释的很清楚了，name是一个属性，printName( )是一个方法，与某个对象进行绑定的函数称作为方法。一般在类里面定义的函数与类对象或者实例对象绑定了，所以称作为方法；而在类外定义的函数一般没有同对象进行绑定，就称为函数。

**二. 属性**

　　在类中我们可以定义一些属性，比如：

1. **class** people:
2. name = 'jack'
3. age = 12
5. p = people()
6. **print** p.name,p.age

　　定义了一个people类，里面定义了name和age属性，默认值分别为'jack'和12。在定义了类之后，就可以用来产生实例化对象了，这句p = people( )实例化了一个对象p，然后就可以通过p来读取属性了。这里的name和age都是公有的，可以直接在类外通过对象名访问，如果想定义成私有的，则需在前面加2个下划线 ' \_\_'。

**[python]** [view plain](http://blog.csdn.net/zhoudaxia/article/details/23341261) [copy](http://blog.csdn.net/zhoudaxia/article/details/23341261)

[在CODE上查看代码片](https://code.csdn.net/snippets/295557)

1. **class** people:
2. \_\_name = 'jack'
3. \_\_age = 12
5. p = people()
6. **print** p.\_\_name,p.\_\_age

　　这段程序运行会报错：

1. Traceback (most recent call last):
2. File "C:/PycharmProjects/FirstProject/oop.py", line 6, in <module>
3. print p.\_\_name,p.\_\_age
4. AttributeError: people instance has no attribute '\_\_name

　　提示找不到该属性，因为私有属性是不能够在类外通过对象名来进行访问的。**在Python中没有像C++中public和private这些关键字来区别公有属性和私有属性，它是以属性命名方式来区分，如果在属性名前面加了2个下划线'\_\_'，则表明该属性是私有属性，否则为公有属性（方法也是一样，方法名前面加了2个下划线的话表示该方法是私有的，否则为公有的）。**

**三. 方法**

　　在类中可以根据需要定义一些方法，定义方法采用def关键字，在类中定义的方法至少会有一个参数，，一般以名为'self'的变量作为该参数（用其他名称也可以），而且需要作为第一个参数。下面看个例子：

1. **class** people:
2. \_\_name = 'jack'
3. \_\_age = 12
5. **def** getName(self):
6. **return** self.\_\_name
7. **def** getAge(self):
8. **return** self.\_\_age
10. p = people()
11. **print** p.getName(),p.getAge()

　　如果对self不好理解的话，可以把它当做C++中类里面的this指针一样理解，就是对象自身的意思，在用某个对象调用该方法时，就将该对象作为第一个参数传递给self。

**四. 类中内置的方法**

　　在Python中有一些内置的方法，这些方法命名都有比较特殊的地方（其方法名以2个下划线开始然后以2个下划线结束）。类中最常用的就是构造方法和析构方法。

**构造方法\_\_init\_\_(self,....)：在生成对象时调用，可以用来进行一些初始化操作，不需要显示去调用，系统会默认去执行。**构造方法支持重载，如果用户自己没有重新定义构造方法，系统就自动执行默认的构造方法。

**析构方法\_\_del\_\_(self)：在释放对象时调用，支持重载，可以在里面进行一些释放资源的操作，不需要显示调用。**

　　还有其他的一些内置方法，比如 \_\_cmp\_\_( ), \_\_len( )\_\_等。下面是常用的内置方法：

|  |  |
| --- | --- |
| 内置方法 | 说明 |
| **\_\_init\_\_(self,...)** | 初始化对象，在创建新对象时调用 |
| **\_\_del\_\_(self)** | 释放对象，在对象被删除之前调用 |
| **\_\_new\_\_(cls,\*args,\*\*kwd)** | 实例的生成操作 |
| **\_\_str\_\_(self)** | 在使用print语句时被调用 |
| **\_\_getitem\_\_(self,key)** | 获取序列的索引key对应的值，等价于seq[key] |
| **\_\_len\_\_(self)** | 在调用内联函数len()时被调用 |
| **\_\_cmp\_\_(stc,dst)** | 比较两个对象src和dst |
| **\_\_getattr\_\_(s,name)** | 获取属性的值 |
| **\_\_setattr\_\_(s,name,value)** | 设置属性的值 |
| **\_\_delattr\_\_(s,name)** | 删除name属性 |
| **\_\_getattribute\_\_()** | \_\_getattribute\_\_()功能与\_\_getattr\_\_()类似 |
| **\_\_gt\_\_(self,other)** | 判断self对象是否大于other对象 |
| **\_\_lt\_\_(slef,other)** | 判断self对象是否小于other对象 |
| **\_\_ge\_\_(slef,other)** | 判断self对象是否大于或者等于other对象 |
| **\_\_le\_\_(slef,other)** | 判断self对象是否小于或者等于other对象 |
| **\_\_eq\_\_(slef,other)** | 判断self对象是否等于other对象 |
| **\_\_call\_\_(self,\*args)** | 把实例对象作为函数调用 |

**\_\_init\_\_():\_\_init\_\_方法在类的一个对象被建立时，马上运行。这个方法可以用来对你的对象做一些你希望的初始化。**注意，这个名称的开始和结尾都是双下划线。代码例子:

1. # Filename: class\_init.py
2. **class** Person:
3. **def** \_\_init\_\_(self, name):
4. self.name = name
5. **def** sayHi(self):
6. **print** 'Hello, my name is', self.name
8. p = Person('Swaroop')
9. p.sayHi()
11. 输出：
12. Hello, my name **is** Swaroop

**\_\_new\_\_():\_\_new\_\_()在\_\_init\_\_()之前被调用，用于生成实例对象。**利用这个方法和类属性的特性可以实现设计模式中的单例模式。单例模式是指创建唯一对象吗，单例模式设计的类只能实例化一个对象。

1. # -\*- coding: UTF-8 -\*-
3. **class** Singleton(object):
4. \_\_instance = None                       # 定义实例
6. **def** \_\_init\_\_(self):
7. **pass**
9. **def** \_\_new\_\_(cls, \*args, \*\*kwd):         # 在\_\_init\_\_之前调用
10. **if** Singleton.\_\_instance **is** None:    # 生成唯一实例
11. Singleton.\_\_instance = object.\_\_new\_\_(cls, \*args, \*\*kwd)
12. **return** Singleton.\_\_instance

**\_\_getattr\_\_()、\_\_setattr\_\_()和\_\_getattribute\_\_():**当读取对象的某个属性时，python会自动调用\_\_getattr\_\_()方法。例如，fruit.color将转换为fruit.\_\_getattr\_\_(color)。当使用赋值语句对属性进行设置时，python会自动调用\_\_setattr\_\_()方法。\_\_getattribute\_\_()的功能与\_\_getattr\_\_()类似，用于获取属性的值。但是\_\_getattribute\_\_()能提供更好的控制，代码更健壮。注意，python中并不存在\_\_setattribute\_\_()方法。代码例子：

1. # -\*- coding: UTF-8 -\*-
3. **class** Fruit(object):
4. **def** \_\_init\_\_(self, color="red", price=0):
5. self.\_\_color = color
6. self.\_\_price = price
8. **def** \_\_getattribute\_\_(self, item):              # <span style="font-family:宋体;font-size:12px;">获取属性的方法</span>
9. **return** object.\_\_getattribute\_\_(self, item)
11. **def** \_\_setattr\_\_(self, key, value):
12. self.\_\_dict\_\_[key] = value
14. **if** \_\_name\_\_ == "\_\_main\_\_":
15. fruit = Fruit("blue", 10)
16. **print** fruit.\_\_dict\_\_.get("\_Fruit\_\_color")    # <span style="font-family:宋体;font-size:12px;">获取color属性</span>
17. fruit.\_\_dict\_\_["\_Fruit\_\_price"] = 5
18. **print** fruit.\_\_dict\_\_.get("\_Fruit\_\_price")    # <span style="font-family:宋体;font-size:12px;">获取price属性</span>

　　Python不允许实例化的类访问私有数据，但你可以使用object.\_className\_\_attrName访问这些私有属性。

**\_\_getitem\_\_():**如果类把某个属性定义为序列，可以使用\_\_getitem\_\_()输出序列属性中的某个元素.假设水果店中销售多钟水果，可以通过\_\_getitem\_\_()方法获取水果店中的没种水果。代码例子：

1. # -\*- coding: UTF-8 -\*-
3. **class** FruitShop:
4. **def** \_\_getitem\_\_(self, i):      # 获取水果店的水果
5. **return** self.fruits[i]
7. **if** \_\_name\_\_ == "\_\_main\_\_":
8. shop = FruitShop()
9. shop.fruits = ["apple", "banana"]
10. **print** shop[1]
11. **for** item **in** shop:               # 输出水果店的水果
12. **print** item,

　　输出：

1. banana
2. apple banana

　　\_**\_str\_\_():**\_\_str\_\_()用于表示对象代表的含义，返回一个字符串.实现了\_\_str\_\_()方法后，可以直接使用print语句输出对象，也可以通过函数str()触发\_\_str\_\_()的执行。这样就把对象和字符串关联起来，便于某些程序的实现，可以用这个字符串来表示某个类。代码例子：

1. # -\*- coding: UTF-8 -\*-
3. **class** Fruit:
4. '''''Fruit类'''               #为Fruit类定义了文档字符串
5. **def** \_\_str\_\_(self):          # 定义对象的字符串表示
6. **return** self.\_\_doc\_\_
8. **if** \_\_name\_\_ == "\_\_main\_\_":
9. fruit = Fruit()
10. **print** str(fruit)            # 调用内置函数str()触发\_\_str\_\_()方法，输出结果为:Fruit类
11. **print** fruit                 #直接输出对象fruit,返回\_\_str\_\_()方法的值，输出结果为:Fruit类

**\_\_call\_\_():**在类中实现\_\_call\_\_()方法，可以在对象创建时直接返回\_\_call\_\_()的内容。使用该方法可以模拟静态方法。代码例子:

1. # -\*- coding: UTF-8 -\*-
3. **class** Fruit:
4. **class** Growth:        # 内部类
5. **def** \_\_call\_\_(self):
6. **print** "grow ..."
8. grow = Growth()      # 调用Growth()，此时将类Growth作为函数返回,即为外部类Fruit定义方法grow(),grow()将执行\_\_call\_\_()内的代码
9. **if** \_\_name\_\_ == '\_\_main\_\_':
10. fruit = Fruit()
11. fruit.grow()         # 输出结果：grow ...
12. Fruit.grow()         # 输出结果：grow ...

**五. 类属性、实例属性、类方法、实例方法以及静态方法**

　　在了解了类基本的东西之后，下面看一下python中这几个概念的区别。

　　先来谈一下类属性和实例属性

　　在前面的例子中我们接触到的就是类属性，顾名思义，**类属性就是类对象所拥有的属性，它被所有类对象的实例对象所共有，在内存中只存在一个副本，这个和C++中类的静态成员变量有点类似。**对于公有的类属性，在类外可以通过类对象和实例对象访问。

1. **class** people:
2. name = 'jack'  #公有的类属性
3. \_\_age = 12     #私有的类属性
5. p = people()
7. **print** p.name             #正确
8. **print** people.name        #正确
9. **print** p.\_\_age            #错误，不能在类外通过实例对象访问私有的类属性
10. **print** people.\_\_age       #错误，不能在类外通过类对象访问私有的类属性

**实例属性是不需要在类中显示定义的**，比如：

1. **class** people:
2. name = 'jack'
4. p = people()
5. p.age =12
6. **print** p.name    #正确
7. **print** p.age     #正确
9. **print** people.name    #正确
10. **print** people.age     #错误

　　在类外对类对象people进行实例化之后，产生了一个实例对象p，然后p.age = 12这句给p添加了一个实例属性age，赋值为12。这个实例属性是实例对象p所特有的，注意，类对象people并不拥有它（所以不能通过类对象来访问这个age属性）。当然还可以在实例化对象的时候给age赋值。

1. **class** people:
2. name = 'jack'
4. #\_\_init\_\_()是内置的构造方法，在实例化对象时自动调用
5. **def** \_\_init\_\_(self,age):
6. self.age = age
8. p = people(12)
9. **print** p.name    #正确
10. **print** p.age     #正确
12. **print** people.name    #正确
13. **print** people.age     #错误

**如果需要在类外修改类属性，必须通过类对象去引用然后进行修改。如果通过实例对象去引用，会产生一个同名的实例属性，这种方式修改的是实例属性，不会影响到类属性，并且之后如果通过实例对象去引用该名称的属性，实例属性会强制屏蔽掉类属性，即引用的是实例属性，除非删除了该实例属性。**

1. **class** people:
2. country = 'china'

5. **print** people.country
6. p = people()
7. **print** p.country
8. p.country = 'japan'
9. **print** p.country      #实例属性会屏蔽掉同名的类属性
10. **print** people.country
11. **del** p.country    #删除实例属性
12. **print** p.country

　　下面来看一下类方法、实例方法和静态方法的区别。

**类方法：**是类对象所拥有的方法，需要用修饰器"@classmethod"来标识其为类方法，对于类方法，第一个参数必须是类对象，一般以"cls"作为第一个参数（当然可以用其他名称的变量作为其第一个参数，但是大部分人都习惯以'cls'作为第一个参数的名字，就最好用'cls'了），能够通过实例对象和类对象去访问。

1. **class** people:
2. country = 'china'
4. #类方法，用classmethod来进行修饰
5. @classmethod
6. **def** getCountry(cls):
7. **return** cls.country
9. p = people()
10. **print** p.getCountry()    #可以用过实例对象引用
11. **print** people.getCountry()    #可以通过类对象引用

　　类方法还有一个用途就是可以对类属性进行修改：

1. **class** people:
2. country = 'china'
4. #类方法，用classmethod来进行修饰
5. @classmethod
6. **def** getCountry(cls):
7. **return** cls.country
9. @classmethod
10. **def** setCountry(cls,country):
11. cls.country = country

14. p = people()
15. **print** p.getCountry()    #可以用过实例对象引用
16. **print** people.getCountry()    #可以通过类对象引用
18. p.setCountry('japan')
20. **print** p.getCountry()
21. **print** people.getCountry()

　　运行结果：

1. china
2. china
3. japan
4. japan

　　结果显示在用类方法对类属性修改之后，通过类对象和实例对象访问都发生了改变。

**实例方法：**在类中最常定义的成员方法，它至少有一个参数并且必须以实例对象作为其第一个参数，一般以名为'self'的变量作为第一个参数（当然可以以其他名称的变量作为第一个参数）。**在类外实例方法只能通过实例对象去调用，不能通过其他方式去调用。**

1. **class** people:
2. country = 'china'
4. #实例方法
5. **def** getCountry(self):
6. **return** self.country

9. p = people()
10. **print** p.getCountry()         #正确，可以用过实例对象引用
11. **print** people.getCountry()    #错误，不能通过类对象引用实例方法

**静态方法：**需要通过修饰器"@staticmethod"来进行修饰，静态方法不需要多定义参数。

1. **class** people:
2. country = 'china'
4. @staticmethod
5. #静态方法
6. **def** getCountry():
7. **return** people.country

10. **print** people.getCountry()

　　对于类属性和实例属性，如果在类方法中引用某个属性，该属性必定是类属性，而如果在实例方法中引用某个属性（不作更改），并且存在同名的类属性，此时若实例对象有该名称的实例属性，则实例属性会屏蔽类属性，即引用的是实例属性，若实例对象没有该名称的实例属性，则引用的是类属性；如果在实例方法更改某个属性，并且存在同名的类属性，此时若实例对象有该名称的实例属性，则修改的是实例属性，若实例对象没有该名称的实例属性，则会创建一个同名称的实例属性。想要修改类属性，如果在类外，可以通过类对象修改，如果在类里面，只有在类方法中进行修改。

**从类方法和实例方法以及静态方法的定义形式就可以看出来，类方法的第一个参数是类对象cls，那么通过cls引用的必定是类对象的属性和方法；而实例方法的第一个参数是实例对象self，那么通过self引用的可能是类属性、也有可能是实例属性（这个需要具体分析），不过在存在相同名称的类属性和实例属性的情况下，实例属性优先级更高。静态方法中不需要额外定义参数，因此在静态方法中引用类属性的话，必须通过类对象来引用。**

**六. 继承和多重继承**

　　上面谈到了类的基本定义和使用方法，这只体现了面向对象编程的三大特点之一：封装。下面就来了解一下另外两大特征：继承和多态。

　　在Python中，如果需要的话，可以让一个类去继承一个类，被继承的类称为父类或者超类、也可以称作基类，继承的类称为子类。并且Python支持多继承，能够让一个子类有多个父类。

　　Python中类的继承定义基本形式如下：

1. #父类
2. **class** superClassName:
3. block
5. #子类
6. **class** subClassName(superClassName):
7. block

　　在定义一个类的时候，可以在类名后面紧跟一对括号，在括号中指定所继承的父类，如果有多个父类，多个父类名之间用逗号隔开。以大学里的学生和老师举例，可以定义一个父类UniversityMember，然后类Student和类Teacher分别继承类UniversityMember：

1. # -\*- coding: UTF-8 -\*-
3. **class** UniversityMember:
5. **def** \_\_init\_\_(self,name,age):
6. self.name = name
7. self.age = age
9. **def** getName(self):
10. **return** self.name
12. **def** getAge(self):
13. **return** self.age
15. **class** Student(UniversityMember):
17. **def** \_\_init\_\_(self,name,age,sno,mark):
18. UniversityMember.\_\_init\_\_(self,name,age)     #注意要显示调用父类构造方法，并传递参数self
19. self.sno = sno
20. self.mark = mark
22. **def** getSno(self):
23. **return** self.sno
25. **def** getMark(self):
26. **return** self.mark


30. **class** Teacher(UniversityMember):
32. **def** \_\_init\_\_(self,name,age,tno,salary):
33. UniversityMember.\_\_init\_\_(self,name,age)
34. self.tno = tno
35. self.salary = salary
37. **def** getTno(self):
38. **return** self.tno
40. **def** getSalary(self):
41. **return** self.salary

　　在大学中的每个成员都有姓名和年龄，而学生有学号和分数这2个属性，老师有教工号和工资这2个属性，从上面的代码中可以看到：

　　1）在Python中，如果父类和子类都重新定义了构造方法\_\_init( )\_\_，在进行子类实例化的时候，子类的构造方法不会自动调用父类的构造方法，必须在子类中显示调用。

　　2）**如果需要在子类中调用父类的方法，需要以”父类名.方法“这种方式调用，**以这种方式调用的时候，注意要传递self参数过去。

　　对于继承关系，子类继承了父类所有的公有属性和方法，可以在子类中通过父类名来调用，而对于私有的属性和方法，子类是不进行继承的，因此在子类中是无法通过父类名来访问的。

　　Python支持多重继承。对于多重继承，比如

　　class SubClass(SuperClass1,SuperClass2)

　　此时有一个问题就是如果SubClass没有重新定义构造方法，它会自动调用哪个父类的构造方法？这里记住一点：以第一个父类为中心。如果SubClass重新定义了构造方法，需要显示去调用父类的构造方法，此时调用哪个父类的构造方法由你自己决定；若SubClass没有重新定义构造方法，则只会执行第一个父类的构造方法。**并且若SuperClass1和SuperClass2中有同名的方法，通过子类的实例化对象去调用该方法时调用的是第一个父类中的方法。**

**七. 多态**

　　多态即多种形态，在运行时确定其状态，在编译阶段无法确定其类型，这就是多态。Python中的多态和Java以及C++中的多态有点不同，Python中的变量是弱类型的，在定义时不用指明其类型，它会根据需要在运行时确定变量的类型（个人觉得这也是多态的一种体现），并且Python本身是一种解释性语言，不进行预编译，因此它就只在运行时确定其状态，故也有人说Python是一种多态语言。在Python中很多地方都可以体现多态的特性，比如内置函数len(object)，len函数不仅可以计算字符串的长度，还可以计算列表、元组等对象中的数据个数，**这里在运行时通过参数类型确定其具体的计算过程，正是多态的一种体现。**这有点类似于函数重载（一个编译单元中有多个同名函数，但参数不同），相当于为每种类型都定义了一个len函数。这是典型的多态表现。有些朋友提出Python不支持多态，我是完全不赞同的。

**本质上，多态意味着可以对不同的对象使用同样的操作，但它们可能会以多种形态呈现出结果。**len(object)函数就体现了这一点。在C++、Java、C#这种编译型语言中，由于有编译过程，因此就鲜明地分成了运行时多态和编译时多态。运行时多态是指允许父类指针或名称来引用子类对象，或对象方法，而实际调用的方法为对象的类类型方法，这就是所谓的动态绑定。编译时多态有模板或范型、方法重载（overload）、方法重写（override）等。而Python是动态语言，动态地确定类型信息恰恰体现了多态的特征。在Python中，任何不知道对象到底是什么类型，但又需要对象做点什么的时候，都会用到多态。

　　能够直接说明多态的两段示例代码如下：  
　　**1、方法多态**

1. # -\*- coding: UTF-8 -\*-
3. \_metaclass\_=type # 确定使用新式类
4. **class** calculator:
6. **def** count(self,args):
7. **return** 1
9. calc=calculator() #自定义类型
11. **from** random **import** choice
12. obj=choice(['hello,world',[1,2,3],calc]) #obj是随机返回的 类型不确定
13. **print** type(obj)
14. **print** obj.count('a') #方法多态

　　对于一个临时对象obj，它通过Python的随机函数取出来，不知道具体类型（是字符串、元组还是自定义类型），都可以调用count方法进行计算，至于count由谁（哪种类型）去做怎么去实现我们并不关心。

　　有一种称为”[鸭子类型（duck typing）](http://zh.wikipedia.org/wiki/Duck_typing)“的东西，讲的也是多态：

1. \_metaclass\_=type # 确定使用新式类
2. **class** Duck:
3. **def** quack(self):
4. **print** "Quaaaaaack!"
5. **def** feathers(self):
6. **print** "The duck has white and gray feathers."
8. **class** Person:
9. **def** quack(self):
10. **print** "The person imitates a duck."
11. **def** feathers(self):
12. **print** "The person takes a feather from the ground and shows it."
14. **def** in\_the\_forest(duck):
15. duck.quack()
16. duck.feathers()
18. **def** game():
19. donald = Duck()
20. john = Person()
21. in\_the\_forest(donald)
22. in\_the\_forest(john)
24. game()

　　就in\_the\_forest函数而言，参数对象是一个鸭子类型，它实现了方法多态。但是实际上我们知道，从严格的抽象来讲，Person类型和Duck完全风马牛不相及。  
　　**2、运算符多态**

1. **def** add(x,y):
2. **return** x+y
4. **print** add(1,2) #输出3
6. **print** add("hello,","world") #输出hello,world
8. **print** add(1,"abc") #抛出异常 TypeError: unsupported operand type(s) for +: 'int' and 'str'

　　上例中，显而易见，Python的加法运算符是”多态“的，理论上，我们实现的add方法支持任意支持加法的对象，但是我们不用关心两个参数x和y具体是什么类型。  
　　Python同样支持运算符重载，实例如下：

1. **class** Vector:
2. **def** \_\_init\_\_(self, a, b):
3. self.a = a
4. self.b = b
6. **def** \_\_str\_\_(self):
7. **return** 'Vector (%d, %d)' % (self.a, self.b)
9. **def** \_\_add\_\_(self,other):
10. **return** Vector(self.a + other.a, self.b + other.b)
12. v1 = Vector(2,10)
13. v2 = Vector(5,-2)
14. **print** v1 + v2

一两个示例代码当然不能从根本上说明多态。**普遍认为面向对象最有价值最被低估的特征其实是多态。**我们所理解的多态的实现和子类的虚函数地址绑定有关系，多态的效果其实和函数地址运行时动态绑定有关。在C++, Java, C#中实现多态的方式通常有重写和重载两种，从上面两段代码，我们其实可以分析得出Python中实现多态也可以变相理解为重写和重载。在Python中很多内置函数和运算符都是多态的。

参考文献：

<http://www.cnblogs.com/dolphin0520/archive/2013/03/29/2986924.html>

<http://www.cnblogs.com/jeffwongishandsome/archive/2012/10/06/2713258.html>

摘自：<http://blog.csdn.net/zhoudaxia/article/details/23341261>

### Python正则表达式

**字符匹配**

大多数字母和字符一般都会和自身匹配。例如，正则表达式 test 会和字符串“test”完全匹配。（你也可以使用大小写不敏感模式，它还能让这个 RE 匹配“Test”或“TEST”；稍後会有更多解释。）

这个规则当然会有例外；有些字符比较特殊，它们和自身并不匹配，而是会表明应和一些特殊的东西匹配，或者它们会影响到 RE 其它部分的重复次数。本文很大篇幅专门讨论了各种元字符及其作用。

这里有一个元字符的完整列表；其含义会在本指南馀下部分进行讨论。

. ^ $ \* + ? { [ ] \ | ( )

我们首先考察的元字符是"[" 和 "]"。它们常用来指定一个字符类别，所谓字符类别就是你想匹配的一个字符集。字符可以单个列出，也可以用“-”号分隔的两个给定字符来表示一个字符区间。例如，[abc] 将匹配"a", "b", 或 "c"中的任意一个字符；也可以用区间[a-c]来表示同一字符集，和前者效果一致。如果你只想匹配小写字母，那幺 RE 应写成 [a-z].

元字符在类别里并不起作用。例如，[akm$]将匹配字符"a", "k", "m", 或 "$" 中的任意一个；"$"通常用作元字符，但在字符类别里，其特性被除去，恢复成普通字符。

你可以用补集来匹配不在区间范围内的字符。其做法是把"^"作为类别的首个字符；其它地方的"^"只会简单匹配 "^"字符本身。例如，[^5] 将匹配除 "5" 之外的任意字符。

也许最重要的元字符是反斜杠"""。 做为 Python 中的字符串字母，反斜杠後面可以加不同的字符以表示不同特殊意义。它也可以用于取消所有的元字符，这样你就可以在模式中匹配它们了。举个例子，如果你需要匹配字符 "[" 或 """，你可以在它们之前用反斜杠来取消它们的特殊意义： "[ 或 ""。

一些用 """ 开始的特殊字符所表示的预定义字符集通常是很有用的，象数字集，字母集，或其它非空字符集。下列是可用的预设特殊字符：

\d  匹配任何十进制数；它相当于类 [0-9]。  
  
\D  匹配任何非数字字符；它相当于类 [^0-9]。  
  
\s  匹配任何空白字符；它相当于类  [ "t"n"r"f"v]。  
  
\S  匹配任何非空白字符；它相当于类 [^ "t"n"r"f"v]。  
  
\w  匹配任何字母数字字符；它相当于类 [a-zA-Z0-9\_]。  
  
\W  匹配任何非字母数字字符；它相当于类 [^a-zA-Z0-9\_]。

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

本节最後一个元字符是 . 。它匹配除了换行字符外的任何字符，在 alternate 模式（re.DOTALL）下它甚至可以匹配换行。"." 通常被用于你想匹配“任何字符”的地方。

**重复**

正则表达式第一件能做的事是能够匹配不定长的字符集，而这是其它能作用在字符串上的方法所不能做到的。 不过，如果那是正则表达式唯一的附加功能的话，那么它们也就不那么优秀了。它们的另一个功能就是你可以指定正则表达式的一部分的重复次数。

我们讨论的第一个重复功能的元字符是 \*。\* 并不匹配字母字符 "\*"；相反，它指定前一个字符可以被匹配零次或更多次，而不是只有一次。

举个例子，ca\*t 将匹配 "ct" (0 个 "a" 字符), "cat" (1 个 "a"), "caaat" (3 个 "a" 字符)等等。RE 引擎有各种来自 C 的整数类型大小的内部限制，以防止它匹配超过2亿个 "a" 字符；你也许没有足够的内存去建造那么大的字符串，所以将不会累计到那个限制。

象 \* 这样地重复是“贪婪的”；当重复一个 RE 时，匹配引擎会试着重复尽可能多的次数。如果模式的後面部分没有被匹配，匹配引擎将退回并再次尝试更小的重复。  
 一步步的示例可以使它更加清晰。让我们考虑表达式 a[bcd]\*b。它匹配字母 "a"，零个或更多个来自类 [bcd]中的字母，最後以 "b" 结尾。现在想一想该 RE 对字符串 "abcbd" 的匹配。

|  |  |  |
| --- | --- | --- |
| Step | Matched | Explanation |
| 1 | a | a 匹配模式 |
| 2 | abcbd | 引擎匹配 [bcd]\*，并尽其所能匹配到字符串的结尾 |
| 3 | Failure | 引擎尝试匹配 b，但当前位置已经是字符的最後了，所以失败 |
| 4 | abcb | 退回，[bcd]\*尝试少匹配一个字符。 |
| 5 | Failure | 再次尝次b，但在当前最後一位字符是"d"。 |
| 6 | abc | 再次退回，[bcd]\*只匹配 "bc"。 |
| 7 | abcb | 再次尝试 b ，这次当前位上的字符正好是 "b" |

RE 的结尾部分现在可以到达了，它匹配 "abcb"。这证明了匹配引擎一开始会尽其所能进行匹配，如果没有匹配然後就逐步退回并反复尝试 RE 剩下来的部分。直到它退回尝试匹配 [bcd] 到零次为止，如果随後还是失败，那么引擎就会认为该字符串根本无法匹配 RE 。  
 另一个重复元字符是 +，表示匹配一或更多次。请注意 \* 和 + 之间的不同；＊匹配零或更多次，所以根本就可以不出现，而 + 则要求至少出现一次。用同一个例子，ca+t 就可以匹配 "cat" (1 个 "a")， "caaat" (3 个 "a")， 但不能匹配 "ct"。

还有更多的限定符。问号 ? 匹配一次或零次；你可以认为它用于标识某事物是可选的。例如：home-?brew 匹配 "homebrew" 或 "home-brew"。  
 最复杂的重复限定符是 {m,n}，其中 m 和 n 是十进制整数。该限定符的意思是至少有 m 个重复，至多到 n 个重复。举个例子，a/{1,3}b 将匹配 "a/b"，"a//b" 和 "a///b"。它不能匹配 "ab" 因为没有斜杠，也不能匹配 "a////b" ，因为有四个。  
 你可以忽略 m 或 n；因为会为缺失的值假设一个合理的值。忽略 m 会认为下边界是 0，而忽略 n 的结果将是上边界为无穷大 -- 实际上是先前我们提到的 2 兆，但这也许同无穷大一样。  
 细心的读者也许注意到其他三个限定符都可以用这样方式来表示。 {0,} 等同于 \*，{1,} 等同于 +，而{0,1}则与 ? 相同。如果可以的话，最好使用 \*，+，或?。很简单因为它们更短也再容易懂。

**编译正则表达式**

正则表达式被编译成 `RegexObject` 实例，可以为不同的操作提供方法，如模式匹配搜索或字符串替换。

#!python  
>>> import re  
>>> p = re.compile('ab\*')  
>>> print p  
<re.RegexObject instance at 80b4150>

re.compile() 也接受可选的标志参数，常用来实现不同的特殊功能和语法变更。我们稍後将查看所有可用的设置，但现在只举一个例子：

#!python  
>>> p = re.compile('ab\*', re.IGNORECASE)

RE 被做为一个字符串发送给 re.compile()。REs 被处理成字符串是因为正则表达式不是 Python 语言的核心部分，也没有为它创建特定的语法。（应用程序根本就不需要 REs，因此没必要包含它们去使语言说明变得臃肿不堪。）而 re 模块则只是以一个 C 扩展模块的形式来被 Python 包含，就象 socket 或 zlib 模块一样。  
 将 REs 作为字符串以保证 Python 语言的简洁，但这样带来的一个麻烦就是象下节标题所讲的。

**反斜杠的麻烦**

在早期规定中，正则表达式用反斜杠字符 (""") 来表示特殊格式或允许使用特殊字符而不调用它的特殊用法。这就与 Python 在字符串中的那些起相同作用的相同字符产生了冲突。  
 让我们举例说明，你想写一个 RE 以匹配字符串 ""section"，可能是在一个 LATEX 文件查找。为了要在程序代码中判断，首先要写出想要匹配的字符串。接下来你需要在所有反斜杠和元字符前加反斜杠来取消其特殊意义。

|  |  |
| --- | --- |
| 字符 | 阶段 |
| \section | 要匹配的字符串 |
| \\section | 为 re.compile 取消反斜杠的特殊意义 |
| "\\\\section" | 为字符串取消反斜杠 |

简单地说，为了匹配一个反斜杠，不得不在 RE 字符串中写 '\\'，因为正则表达式中必须是 "\\"，而每个反斜杠按 Python 字符串字母表示的常规必须表示成 "\\"。在 REs 中反斜杠的这个重复特性会导致大量重复的反斜杠，而且所生成的字符串也很难懂。  
 解决的办法就是为正则表达式使用 Python 的 raw 字符串表示；在字符串前加个 "r" 反斜杠就不会被任何特殊方式处理，所以 r"\n" 就是包含"\" 和 "n" 的两个字符，而 "\n" 则是一个字符，表示一个换行。正则表达式通常在 Python 代码中都是用这种 raw 字符串表示。

|  |  |
| --- | --- |
| 常规字符串 | Raw 字符串 |
| "ab\*" | r"ab\*" |
| "\\\\section" | r"\\section" |
| "\\w+\\s+\\1" | r"\w+\s+\1" |

**执行匹配**

一旦你有了已经编译了的正则表达式的对象，你要用它做什么呢？`RegexObject` 实例有一些方法和属性。这里只显示了最重要的几个，如果要看完整的列表请查阅 Python Library Reference

|  |  |
| --- | --- |
| 方法/属性 | 作用 |
| match() | 决定 RE 是否在字符串刚开始的位置匹配 |
| search() | 扫描字符串，找到这个 RE 匹配的位置 |
| findall() | 找到 RE 匹配的所有子串，并把它们作为一个列表返回 |
| finditer() | 找到 RE 匹配的所有子串，并把它们作为一个迭代器返回 |

如果没有匹配到的话，match() 和 search() 将返回 None。如果成功的话，就会返回一个 `MatchObject` 实例，其中有这次匹配的信息：它是从哪里开始和结束，它所匹配的子串等等。

你可以用采用人机对话并用 re 模块实验的方式来学习它。如果你有 Tkinter 的话，你也许可以考虑参考一下 Tools/scripts/redemo.py，一个包含在 Python 发行版里的示范程序。

首先，运行 Python 解释器，导入 re 模块并编译一个 RE：

#!python  
  
Python 2.2.2 (#1, Feb 10 2003, 12:57:01)  
  
>>> import re  
  
>>> p = re.compile('[a-z]+')  
  
>>> p  
  
<\_sre.SRE\_Pattern object at 80c3c28>

现在，你可以试着用 RE 的 [a-z]+ 去匹配不同的字符串。一个空字符串将根本不能匹配，因为 + 的意思是 “一个或更多的重复次数”。 在这种情况下 match() 将返回 None，因为它使解释器没有输出。你可以明确地打印出 match() 的结果来弄清这一点。

#!python  
  
>>> p.match("")  
  
>>> print p.match("")  
  
None

现在，让我们试着用它来匹配一个字符串，如 "tempo"。这时，match() 将返回一个 MatchObject。因此你可以将结果保存在变量里以便后面使用。

#!python  
  
>>> m = p.match( 'tempo')  
  
>>> print m  
  
<\_sre.SRE\_Match object at 80c4f68>

现在你可以查询 `MatchObject` 关于匹配字符串的相关信息了。MatchObject 实例也有几个方法和属性；最重要的那些如下所示：

|  |  |
| --- | --- |
| 方法/属性 | 作用 |
| 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 |

试试这些方法不久就会清楚它们的作用了：

#!python  
  
>>> m.group()  
  
'tempo'  
  
>>> m.start(), m.end()  
  
(0, 5)  
  
>>> m.span()  
  
(0, 5)

group() 返回 RE 匹配的子串。start() 和 end() 返回匹配开始和结束时的索引。span() 则用单个元组把开始和结束时的索引一起返回。因为匹配方法检查到如果 RE 在字符串开始处开始匹配，那么 start() 将总是为零。然而， `RegexObject` 实例的 search 方法扫描下面的字符串的话，在这种情况下，匹配开始的位置就也许不是零了。

#!python  
  
>>> print p.match('::: message')  
  
None  
  
>>> m = p.search('::: message') ; print m  
  
<re.MatchObject instance at 80c9650>  
  
>>> m.group()  
  
'message'  
  
>>> m.span()  
  
(4, 11)

在实际程序中，最常见的作法是将 `MatchObject` 保存在一个变量里，然后检查它是否为 None，通常如下所示：

#!python  
  
p = re.compile( http://files.jb51.net/upload/201105/20110519010141847.gif )  
  
m = p.match( 'string goes here' )  
  
if m:  
  
print 'Match found: ', m.group()  
  
else:  
  
print 'No match'  
  
两个 `RegexObject` 方法返回所有匹配模式的子串。findall()返回一个匹配字符串行表：  
  
#!python  
  
>>> p = re.compile('"d+')  
  
>>> p.findall('12 drummers drumming, 11 pipers piping, 10 lords a-leaping')  
  
['12', '11', '10']

findall() 在它返回结果时不得不创建一个列表。在 Python 2.2中，也可以用 finditer() 方法。

#!python  
  
>>> iterator = p.finditer('12 drummers drumming, 11 http://files.jb51.net/upload/201105/20110519010141847.gif 10 http://files.jb51.net/upload/201105/20110519010141847.gif')  
  
>>> iterator  
  
<callable-iterator object at 0x401833ac>  
  
>>> for match in iterator:  
  
http://files.jb51.net/upload/201105/20110519010141847.gif     print match.span()  
  
http://files.jb51.net/upload/201105/20110519010141847.gif  
  
(0, 2)  
  
(22, 24)  
  
(29, 31)

**模块级函数**

你不一定要产生一个 `RegexObject` 对象然後再调用它的方法；re 模块也提供了顶级函数调用如 match()、search()、sub() 等等。这些函数使用 RE 字符串作为第一个参数，而後面的参数则与相应 `RegexObject` 的方法参数相同，返回则要么是 None 要么就是一个 `MatchObject` 的实例。

#!python  
>>> print re.match(r'From"s+', 'Fromage amk')  
None  
>>> re.match(r'From"s+', 'From amk Thu May 14 19:12:10 1998')  
<re.MatchObject instance at 80c5978>

Under the hood, 这些函数简单地产生一个 RegexOject 并在其上调用相应的方法。它们也在缓存里保存编译後的对象，因此在将来调用用到相同 RE 时就会更快。  
 你将使用这些模块级函数，还是先得到一个 `RegexObject` 再调用它的方法呢？如何选择依赖于怎样用 RE 更有效率以及你个人编码风格。如果一个 RE 在代码中只做用一次的话，那么模块级函数也许更方便。如果程序包含很多的正则表达式，或在多处复用同一个的话，那么将全部定义放在一起，在一段代码中提前编译所有的 REs 更有用。从标准库中看一个例子，这是从 xmllib.py 文件中提取出来的：

#!python  
ref = re.compile( http://files.jb51.net/upload/201105/20110519010141847.gif )  
entityref = re.compile( http://files.jb51.net/upload/201105/20110519010141847.gif )  
charref = re.compile( http://files.jb51.net/upload/201105/20110519010141847.gif )  
starttagopen = re.compile( http://files.jb51.net/upload/201105/20110519010141847.gif )

我通常更喜欢使用编译对象，甚至它只用一次，but few people will be as much of a purist about this as I am。

**编译标志**

编译标志让你可以修改正则表达式的一些运行方式。在 re 模块中标志可以使用两个名字，一个是全名如 IGNORECASE，一个是缩写，一字母形式如 I。（如果你熟悉 Perl 的模式修改，一字母形式使用同样的字母；例如 re.VERBOSE的缩写形式是 re.X。）多个标志可以通过按位 OR-ing 它们来指定。如 re.I | re.M 被设置成 I 和 M 标志：  
这有个可用标志表，对每个标志後面都有详细的说明。

|  |  |
| --- | --- |
| 标志 | 含义 |
| DOTALL, S | 使 . 匹配包括换行在内的所有字符 |
| IGNORECASE, I | 使匹配对大小写不敏感 |
| LOCALE, L | 做本地化识别（locale-aware）匹配 |
| MULTILINE, M | 多行匹配，影响 ^ 和 $ |
| VERBOSE, X | 能够使用 REs 的 verbose 状态，使之被组织得更清晰易懂 |

**I**  
**IGNORECASE**

使匹配对大小写不敏感；字符类和字符串匹配字母时忽略大小写。举个例子，[A-Z]也可以匹配小写字母，Spam 可以匹配 "Spam", "spam", 或 "spAM"。这个小写字母并不考虑当前位置。

**L**  
**LOCALE**

影响 "w, "W, "b, 和 "B，这取决于当前的本地化设置。

locales 是 C 语言库中的一项功能，是用来为需要考虑不同语言的编程提供帮助的。举个例子，如果你正在处理法文文本，你想用 "w+ 来匹配文字，但 "w 只匹配字符类 [A-Za-z]；它并不能匹配 "é" 或 "ç"。如果你的系统配置适当且本地化设置为法语，那么内部的 C 函数将告诉程序 "é" 也应该被认为是一个字母。当在编译正则表达式时使用 LOCALE 标志会得到用这些 C 函数来处理 "w 後的编译对象；这会更慢，但也会象你希望的那样可以用 "w+ 来匹配法文文本。

**M**  
**MULTILINE**  
(此时 ^ 和 $ 不会被解释; 它们将在 4.1 节被介绍.)  
 使用 "^" 只匹配字符串的开始，而 $ 则只匹配字符串的结尾和直接在换行前（如果有的话）的字符串结尾。当本标志指定後， "^" 匹配字符串的开始和字符串中每行的开始。同样的， $ 元字符匹配字符串结尾和字符串中每行的结尾（直接在每个换行之前）。

**S**  
**DOTALL**

使 "." 特殊字符完全匹配任何字符，包括换行；没有这个标志， "." 匹配除了换行外的任何字符。

**X**  
**VERBOSE**  
 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。当该标志被指定时，在 RE 字符串中的空白符被忽略，除非该空白符在字符类中或在反斜杠之後；这可以让你更清晰地组织和缩进 RE。它也可以允许你将注释写入 RE，这些注释会被引擎忽略；注释用 "#"号 来标识，不过该符号不能在字符串或反斜杠之後。  
举个例子，这里有一个使用 re.VERBOSE 的 RE；看看读它轻松了多少？

#!python  
charref = re.compile(r"""  
&[[]]             # Start of a numeric entity reference  
(  
[0-9]+[^0-9]      # Decimal form  
| 0[0-7]+[^0-7]   # Octal form  
| x[0-9a-fA-F]+[^0-9a-fA-F] # Hexadecimal form  
)  
""", re.VERBOSE)  
没有 verbose 设置， RE 会看起来象这样：  
#!python  
charref = re.compile("&#([0-9]+[^0-9]"  
"|0[0-7]+[^0-7]"  
"|x[0-9a-fA-F]+[^0-9a-fA-F])")

在上面的例子里，Python 的字符串自动连接可以用来将 RE 分成更小的部分，但它比用 re.VERBOSE 标志时更难懂。

**更多模式功能**

到目前为止，我们只展示了正则表达式的一部分功能。在本节，我们将展示一些新的元字符和如何使用组来检索被匹配的文本部分。

**更多的元字符**

还有一些我们还没展示的元字符，其中的大部分将在本节展示。  
 剩下来要讨论的一部分元字符是零宽界定符（zero-width assertions）。它们并不会使引擎在处理字符串时更快;相反，它们根本就没有对应任何字符，只是简单的成功或失败。举个例子， "b 是一个在单词边界定位当前位置的界定符（assertions），这个位置根本就不会被 "b 改变。这意味着零宽界定符（zero-width assertions）将永远不会被重复，因为如果它们在给定位置匹配一次，那么它们很明显可以被匹配无数次。

**|**  
 可选项，或者 "or" 操作符。如果 A 和 B 是正则表达式，A|B 将匹配任何匹配了 "A" 或 "B" 的字符串。| 的优先级非常低，是为了当你有多字符串要选择时能适当地运行。Crow|Servo 将匹配"Crow" 或 "Servo", 而不是 "Cro", 一个 "w" 或 一个 "S", 和 "ervo"。  
 为了匹配字母 "|"，可以用 "|，或将其包含在字符类中，如[|]。

**^**  
 匹配行首。除非设置 MULTILINE 标志，它只是匹配字符串的开始。在 MULTILINE 模式里，它也可以直接匹配字符串中的每个换行。  
 例如，如果你只希望匹配在行首单词 "From"，那幺 RE 将用 ^From。

#!python  
>>> print re.search('^From', 'From Here to Eternity')  
<re.MatchObject instance at 80c1520>  
>>> print re.search('^From', 'Reciting From Memory')  
None

**$**  
匹配行尾，行尾被定义为要么是字符串尾，要么是一个换行字符後面的任何位置。

#!python  
>>> print re.search('}$', '{block}')  
<re.MatchObject instance at 80adfa8>  
>>> print re.search('}$', '{block} ')  
None  
>>> print re.search('}$', '{block}"n')  
<re.MatchObject instance at 80adfa8>

匹配一个 "$"，使用 "$ 或将其包含在字符类中，如[$]。

**"A**  
 只匹配字符串首。当不在 MULTILINE 模式，"A 和 ^ 实际上是一样的。然而，在 MULTILINE 模式里它们是不同的；"A 只是匹配字符串首，而 ^ 还可以匹配在换行符之後字符串的任何位置。

**"Z**

Matches only at the end of the string.   
 只匹配字符串尾。

**"b**

单词边界。这是个零宽界定符（zero-width assertions）只用以匹配单词的词首和词尾。单词被定义为一个字母数字序列，因此词尾就是用空白符或非字母数字符来标示的。  
下面的例子只匹配 "class" 整个单词；而当它被包含在其他单词中时不匹配。

#!python  
>>> p = re.compile(r'"bclass"b')  
>>> print p.search('no class at all')  
<re.MatchObject instance at 80c8f28>  
>>> print p.search('the declassified algorithm')  
None  
>>> print p.search('one subclass is')  
None

当用这个特殊序列时你应该记住这里有两个微妙之处。第一个是 Python 字符串和正则表达式之间最糟的冲突。在 Python 字符串里，""b" 是反斜杠字符，ASCII值是8。如果你没有使用 raw 字符串时，那幺 Python 将会把 ""b" 转换成一个回退符，你的 RE 将无法象你希望的那样匹配它了。下面的例子看起来和我们前面的 RE 一样，但在 RE 字符串前少了一个 "r" 。

#!python  
>>> p = re.compile('"bclass"b')  
>>> print p.search('no class at all')  
None  
>>> print p.search('"b' + 'class' + '"b')  
<re.MatchObject instance at 80c3ee0>

第二个在字符类中，这个限定符（assertion）不起作用，"b 表示回退符，以便与 Python 字符串兼容。

**"B**  
 另一个零宽界定符（zero-width assertions），它正好同 "b 相反，只在当前位置不在单词边界时匹配。

**分组**

你经常需要得到比 RE 是否匹配还要多的信息。正则表达式常常用来分析字符串，编写一个 RE 匹配感兴趣的部分并将其分成几个小组。举个例子，一个 RFC-822 的头部用 ":" 隔成一个头部名和一个值，这就可以通过编写一个正则表达式匹配整个头部，用一组匹配头部名，另一组匹配头部值的方式来处理。  
 组是通过 "(" 和 ")" 元字符来标识的。 "(" 和 ")" 有很多在数学表达式中相同的意思；它们一起把在它们里面的表达式组成一组。举个例子，你可以用重复限制符，象 \*, +, ?, 和 {m,n}，来重复组里的内容，比如说(ab)\* 将匹配零或更多个重复的 "ab"。

#!python  
>>> p = re.compile('(ab)\*')  
>>> print p.match('ababababab').span()  
(0, 10)

组用 "(" 和 ")" 来指定，并且得到它们匹配文本的开始和结尾索引；这就可以通过一个参数用 group()、start()、end() 和 span() 来进行检索。组是从 0 开始计数的。组 0 总是存在；它就是整个 RE，所以 `MatchObject` 的方法都把组 0 作为它们缺省的参数。稍後我们将看到怎样表达不能得到它们所匹配文本的 span。

#!python  
>>> p = re.compile('(a)b')  
>>> m = p.match('ab')  
>>> m.group()  
'ab'  
>>> m.group(0)  
'ab'

小组是从左向右计数的，从1开始。组可以被嵌套。计数的数值可以能过从左到右计算打开的括号数来确定。

#!python  
>>> p = re.compile('(a(b)c)d')  
>>> m = p.match('abcd')  
>>> m.group(0)  
'abcd'  
>>> m.group(1)  
'abc'  
>>> m.group(2)  
'b'

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。

#!python  
>>> m.group(2,1,2)  
('b', 'abc', 'b')  
The groups() 方法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。  
#!python  
>>> m.groups()  
('abc', 'b')

模式中的逆向引用允许你指定先前捕获组的内容，该组也必须在字符串当前位置被找到。举个例子，如果组 1 的内容能够在当前位置找到的话，"1 就成功否则失败。记住 Python 字符串也是用反斜杠加数据来允许字符串中包含任意字符的，所以当在 RE 中使用逆向引用时确保使用 raw 字符串。  
例如，下面的 RE 在一个字符串中找到成双的词。

#!python  
>>> p = re.compile(r'("b"w+)"s+"1')  
>>> p.search('Paris in the the spring').group()  
'the the'

象这样只是搜索一个字符串的逆向引用并不常见 -- 用这种方式重复数据的文本格式并不多见 -- 但你不久就可以发现它们用在字符串替换上非常有用。

**无捕获组和命名组**

精心设计的 REs 也许会用很多组，既可以捕获感兴趣的子串，又可以分组和结构化 RE 本身。在复杂的 REs 里，追踪组号变得困难。有两个功能可以对这个问题有所帮助。它们也都使用正则表达式扩展的通用语法，因此我们来看看第一个。  
 Perl 5 对标准正则表达式增加了几个附加功能，Python 的 re 模块也支持其中的大部分。选择一个新的单按键元字符或一个以 """ 开始的特殊序列来表示新的功能，而又不会使 Perl 正则表达式与标准正则表达式产生混乱是有难度的。如果你选择 "&" 做为新的元字符，举个例子，老的表达式认为 "&" 是一个正常的字符，而不会在使用 "& 或 [&] 时也不会转义。  
 Perl 开发人员的解决方法是使用 (?...) 来做为扩展语法。"?" 在括号後面会直接导致一个语法错误，因为 "?" 没有任何字符可以重复，因此它不会产生任何兼容问题。紧随 "?" 之後的字符指出扩展的用途，因此 (?=foo)  
 Python 新增了一个扩展语法到 Perl 扩展语法中。如果在问号後的第一个字符是 "P"，你就可以知道它是针对 Python 的扩展。目前有两个这样的扩展: (?P<name>...) 定义一个命名组，(?P=name) 则是对命名组的逆向引用。如果 Perl 5 的未来版本使用不同的语法增加了相同的功能，那幺 re 模块也将改变以支持新的语法，这是为了兼容性的目的而保持的 Python 专用语法。  
 现在我们看一下普通的扩展语法，我们回过头来简化在复杂 REs 中使用组运行的特性。因为组是从左到右编号的，而且一个复杂的表达式也许会使用许多组，它可以使跟踪当前组号变得困难，而修改如此复杂的 RE 是十分麻烦的。在开始时插入一个新组，你可以改变它之後的每个组号。  
 首先，有时你想用一个组去收集正则表达式的一部分，但又对组的内容不感兴趣。你可以用一个无捕获组: (?:...) 来实现这项功能，这样你可以在括号中发送任何其他正则表达式。

#!python  
>>> m = re.match("([abc])+", "abc")  
>>> m.groups()  
('c',)  
>>> m = re.match("(?:[abc])+", "abc")  
>>> m.groups()  
()

除了捕获匹配组的内容之外，无捕获组与捕获组表现完全一样；你可以在其中放置任何字符，可以用重复元字符如 "\*" 来重复它，可以在其他组（无捕获组与捕获组）中嵌套它。(?:...) 对于修改已有组尤其有用，因为你可以不用改变所有其他组号的情况下添加一个新组。捕获组和无捕获组在搜索效率方面也没什么不同，没有哪一个比另一个更快。  
其次，更重要和强大的是命名组；与用数字指定组不同的是，它可以用名字来指定。  
 命令组的语法是 Python 专用扩展之一： (?P<name>...)。名字很明显是组的名字。除了该组有个名字之外，命名组也同捕获组是相同的。`MatchObject` 的方法处理捕获组时接受的要么是表示组号的整数，要么是包含组名的字符串。命名组也可以是数字，所以你可以通过两种方式来得到一个组的信息：

#!python  
>>> p = re.compile(r'(?P<word>"b"w+"b)')  
>>> m = p.search( '(((( Lots of punctuation )))' )  
>>> m.group('word')  
'Lots'  
>>> m.group(1)  
'Lots'

命名组是便于使用的，因为它可以让你使用容易记住的名字来代替不得不记住的数字。这里有一个来自 imaplib 模块的 RE 示例：

#!python  
InternalDate = re.compile(r'INTERNALDATE "'  
r'(?P<day>[ 123][0-9])-(?P<mon>[A-Z][a-z][a-z])-'  
r'(?P<year>[0-9][0-9][0-9][0-9])'  
r' (?P<hour>[0-9][0-9]):(?P<min>[0-9][0-9]):(?P<sec>[0-9][0-9])'  
r' (?P<zonen>[-+])(?P<zoneh>[0-9][0-9])(?P<zonem>[0-9][0-9])'  
r'"')

很明显，得到 m.group('zonem') 要比记住得到组 9 要容易得多。  
 因为逆向引用的语法，象 (...)"1 这样的表达式所表示的是组号，这时用组名代替组号自然会有差别。还有一个 Python 扩展：(?P=name) ，它可以使叫 name 的组内容再次在当前位置发现。正则表达式为了找到重复的单词，("b"w+)"s+"1 也可以被写成 (?P<word>"b"w+)"s+(?P=word)：

#!python  
>>> p = re.compile(r'(?P<word>"b"w+)"s+(?P=word)')  
>>> p.search('Paris in the the spring').group()  
'the the'

**前向界定符**

另一个零宽界定符（zero-width assertion）是前向界定符。前向界定符包括前向肯定界定符和後向肯定界定符，所下所示：

**(?=...)**

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；模式的剩馀部分还要尝试界定符的右边。

**(?!...)**

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  
 通过示范在哪前向可以成功有助于具体实现。考虑一个简单的模式用于匹配一个文件名，并将其通过 "." 分成基本名和扩展名两部分。如在 "news.rc" 中，"news" 是基本名，"rc" 是文件的扩展名。  
匹配模式非常简单：

.\*[.].\*$

注意 "." 需要特殊对待，因为它是一个元字符；我把它放在一个字符类中。另外注意後面的 $; 添加这个是为了确保字符串所有的剩馀部分必须被包含在扩展名中。这个正则表达式匹配

"foo.bar"、"autoexec.bat"、 "sendmail.cf" 和 "printers.conf"。

现在，考虑把问题变得复杂点；如果你想匹配的扩展名不是 "bat" 的文件名？一些不正确的尝试：

.\*[.][^b].\*$

上面的第一次去除 "bat" 的尝试是要求扩展名的第一个字符不是 "b"。这是错误的，因为该模式也不能匹配 "foo.bar"。

.\*[.]([^b]..|.[^a].|..[^t])$

当你试着修补第一个解决方法而要求匹配下列情况之一时表达式更乱了：扩展名的第一个字符不是 "b"; 第二个字符不是 "a"；或第三个字符不是 "t"。这样可以接受 "foo.bar" 而拒绝 "autoexec.bat"，但这要求只能是三个字符的扩展名而不接受两个字符的扩展名如 "sendmail.cf"。我们将在努力修补它时再次把该模式变得复杂。

.\*[.]([^b].?.?|.[^a]?.?|..?[^t]?)$

在第三次尝试中，第二和第三个字母都变成可选，为的是允许匹配比三个字符更短的扩展名，如 "sendmail.cf"。  
 该模式现在变得非常复杂，这使它很难读懂。更糟的是，如果问题变化了，你想扩展名不是 "bat" 和 "exe"，该模式甚至会变得更复杂和混乱。  
前向否定把所有这些裁剪成：

.\*[.](?!bat$).\*$

前向的意思：如果表达式 bat 在这里没有匹配，尝试模式的其馀部分；如果 bat$ 匹配，整个模式将失败。後面的 $ 被要求是为了确保象 "sample.batch" 这样扩展名以 "bat" 开头的会被允许。  
 将另一个文件扩展名排除在外现在也容易；简单地将其做为可选项放在界定符中。下面的这个模式将以 "bat" 或 "exe" 结尾的文件名排除在外。

.\*[.](?!bat$|exe$).\*$

**修改字符串**

到目前为止，我们简单地搜索了一个静态字符串。正则表达式通常也用不同的方式，通过下面的 `RegexObject` 方法，来修改字符串。

|  |  |
| --- | --- |
| 方法/属性 | 作用 |
| split() | 将字符串在 RE 匹配的地方分片并生成一个列表， |
| sub() | 找到 RE 匹配的所有子串，并将其用一个不同的字符串替换 |
| subn() | 与 sub() 相同，但返回新的字符串和替换次数 |

**将字符串分片**

`RegexObject` 的 split() 方法在 RE 匹配的地方将字符串分片，将返回列表。它同字符串的 split() 方法相似但提供更多的定界符；split()只支持空白符和固定字符串。就象你预料的那样，也有一个模块级的 re.split() 函数。

split(string [, maxsplit = 0])

通过正则表达式将字符串分片。如果捕获括号在 RE 中使用，那么它们的内容也会作为结果列表的一部分返回。如果 maxsplit 非零，那么最多只能分出 maxsplit 个分片。  
 你可以通过设置 maxsplit 值来限制分片数。当 maxsplit 非零时，最多只能有 maxsplit 个分片，字符串的其馀部分被做为列表的最後部分返回。在下面的例子中，定界符可以是非数字字母字符的任意序列。

#!python  
>>> p = re.compile(r'"W+')  
>>> p.split('This is a test, short and sweet, of split().')  
['This', 'is', 'a', 'test', 'short', 'and', 'sweet', 'of', 'split', '']  
>>> p.split('This is a test, short and sweet, of split().', 3)  
['This', 'is', 'a', 'test, short and sweet, of split().']

有时，你不仅对定界符之间的文本感兴趣，也需要知道定界符是什么。如果捕获括号在 RE 中使用，那么它们的值也会当作列表的一部分返回。比较下面的调用：

#!python  
>>> p = re.compile(r'"W+')  
>>> p2 = re.compile(r'("W+)')  
>>> p.split('Thishttp://files.jb51.net/upload/201105/20110519010141847.gif is a test.')  
['This', 'is', 'a', 'test', '']  
>>> p2.split('Thishttp://files.jb51.net/upload/201105/20110519010141847.gif is a test.')  
['This', 'http://files.jb51.net/upload/201105/20110519010141847.gif ', 'is', ' ', 'a', ' ', 'test', '.', '']

模块级函数 re.split() 将 RE 作为第一个参数，其他一样。

#!python  
>>> re.split('["W]+', 'Words, words, words.')  
['Words', 'words', 'words', '']  
>>> re.split('(["W]+)', 'Words, words, words.')  
['Words', ', ', 'words', ', ', 'words', '.', '']  
>>> re.split('["W]+', 'Words, words, words.', 1)  
['Words', 'words, words.']

**搜索和替换**

其他常见的用途就是找到所有模式匹配的字符串并用不同的字符串来替换它们。sub() 方法提供一个替换值，可以是字符串或一个函数，和一个要被处理的字符串。

sub(replacement, string[, count = 0])

返回的字符串是在字符串中用 RE 最左边不重复的匹配来替换。如果模式没有发现，字符将被没有改变地返回。  
 可选参数 count 是模式匹配後替换的最大次数；count 必须是非负整数。缺省值是 0 表示替换所有的匹配。  
 这里有个使用 sub() 方法的简单例子。它用单词 "colour" 替换颜色名。

#!python  
>>> p = re.compile( '(blue|white|red)')  
>>> p.sub( 'colour', 'blue socks and red shoes')  
'colour socks and colour shoes'  
>>> p.sub( 'colour', 'blue socks and red shoes', count=1)  
'colour socks and red shoes'

subn() 方法作用一样，但返回的是包含新字符串和替换执行次数的两元组。

#!python  
>>> p = re.compile( '(blue|white|red)')  
>>> p.subn( 'colour', 'blue socks and red shoes')  
('colour socks and colour shoes', 2)  
>>> p.subn( 'colour', 'no colours at all')  
('no colours at all', 0)

空匹配只有在它们没有紧挨着前一个匹配时才会被替换掉。

#!python  
>>> p = re.compile('x\*')  
>>> p.sub('-', 'abxd')  
'-a-b-d-'

如果替换的是一个字符串，任何在其中的反斜杠都会被处理。""n" 将会被转换成一个换行符，""r"转换成回车等等。未知的转义如 ""j" 则保持原样。逆向引用，如 ""6"，被 RE 中相应的组匹配而被子串替换。这使你可以在替换後的字符串中插入原始文本的一部分。  
 这个例子匹配被 "{" 和 "}" 括起来的单词 "section"，并将 "section" 替换成 "subsection"。

#!python  
>>> p = re.compile('section{ ( [^}]\* ) }', re.VERBOSE)  
>>> p.sub(r'subsection{"1}','section{First} section{second}')  
'subsection{First} subsection{second}'

还可以指定用 (?P<name>...) 语法定义的命名组。""g<name>" 将通过组名 "name" 用子串来匹配，并且 ""g<number>" 使用相应的组号。所以 ""g<2>" 等于 ""2"，但能在替换字符串里含义不清，如 ""g<2>0"。（""20" 被解释成对组 20 的引用，而不是对後面跟着一个字母 "0" 的组 2 的引用。）

#!python  
>>> p = re.compile('section{ (?P<name> [^}]\* ) }', re.VERBOSE)  
>>> p.sub(r'subsection{"1}','section{First}')  
'subsection{First}'  
>>> p.sub(r'subsection{"g<1>}','section{First}')  
'subsection{First}'  
>>> p.sub(r'subsection{"g<name>}','section{First}')  
'subsection{First}'

替换也可以是一个甚至给你更多控制的函数。如果替换是个函数，该函数将会被模式中每一个不重复的匹配所调用。在每个调用时，函数被作为 `MatchObject` 的匹配函属，并可以使用这个信息去计算预期的字符串并返回它。  
在下面的例子里，替换函数将十进制翻译成十六进制：

#!python  
>>> def hexrepl( match ):  
http://files.jb51.net/upload/201105/20110519010141847.gif     "Return the hex string for a decimal number"  
http://files.jb51.net/upload/201105/20110519010141847.gif     value = int( match.group() )  
http://files.jb51.net/upload/201105/20110519010141847.gif     return hex(value)  
http://files.jb51.net/upload/201105/20110519010141847.gif  
>>> p = re.compile(r'"d+')  
>>> p.sub(hexrepl, 'Call 65490 for printing, 49152 for user code.')  
'Call 0xffd2 for printing, 0xc000 for user code.'

当使用模块级的 re.sub() 函数时，模式作为第一个参数。模式也许是一个字符串或一个 `RegexObject`；如果你需要指定正则表达式标志，你必须要么使用 `RegexObject` 做第一个参数，或用使用模式内嵌修正器，如 sub("(?i)b+", "x", "bbbb BBBB") returns 'x x'。

**常见问题**

正则表达式对一些应用程序来说是一个强大的工具，但在有些时候它并不直观而且有时它们不按你期望的运行。本节将指出一些最容易犯的常见错误。

**使用字符串方式**

有时使用 re 模块是个错误。如果你匹配一个固定的字符串或单个的字符类，并且你没有使用 re 的任何象 IGNORECASE 标志的功能，那么就没有必要使用正则表达式了。字符串有一些方法是对固定字符串进行操作的，它们通常快很多，因为都是一个个经过优化的C 小循环，用以代替大的、更具通用性的正则表达式引擎。  
 举个用一个固定字符串替换另一个的例子；如，你可以把 "deed" 替换成 "word"。re.sub() seems like the function to use for this, but consider the replace() method. 注意 replace() 也可以在单词里面进行替换，可以把 "swordfish" 变成 "sdeedfish",不过 RE 也是可以做到的。（为了避免替换单词的一部分，模式将写成 "bword"b，这是为了要求 "word" 两边有一个单词边界。这是个超出替换能力的工作）。  
 另一个常见任务是从一个字符串中删除单个字符或用另一个字符来替代它。你也许可以用象 re.sub('"n',' ',S) 这样来实现，但 translate() 能够实现这两个任务，而且比任何正则表达式操作起来更快。  
 总之，在使用 re 模块之前，先考虑一下你的问题是否可以用更快、更简单的字符串方法来解决。

**match() vs search()**

match() 函数只检查 RE 是否在字符串开始处匹配，而 search() 则是扫描整个字符串。记住这一区别是重要的。记住，match() 只报告一次成功的匹配，它将从 0 处开始；如果匹配不是从 0 开始的，match() 将不会报告它。

#!python  
>>> print re.match('super', 'superstition').span()  
(0, 5)  
>>> print re.match('super', 'insuperable')  
None

另一方面，search() 将扫描整个字符串，并报告它找到的第一个匹配。

#!python  
>>> print re.search('super', 'superstition').span()  
(0, 5)  
>>> print re.search('super', 'insuperable').span()  
(2, 7)

有时你可能倾向于使用 re.match()，只在RE的前面部分添加 .\* 。请尽量不要这么做，最好采用 re.search() 代替之。正则表达式编译器会对 REs 做一些分析以便可以在查找匹配时提高处理速度。一个那样的分析机会指出匹配的第一个字符是什么；举个例子，模式 Crow 必须从 "C" 开始匹配。分析机可以让引擎快速扫描字符串以找到开始字符，并只在 "C" 被发现後才开始全部匹配。

添加 .\* 会使这个优化失败，这就要扫描到字符串尾部，然後回溯以找到 RE 剩馀部分的匹配。使用 re.search() 代替。

**贪婪 vs 不贪婪**

当重复一个正则表达式时，如用 a\*，操作结果是尽可能多地匹配模式。当你试着匹配一对对称的定界符，如 HTML 标志中的尖括号时这个事实经常困扰你。匹配单个 HTML 标志的模式不能正常工作，因为 .\* 的本质是“贪婪”的

#!python  
>>> s = '<html><head><title>Title</title>'  
>>> len(s)  
32  
>>> print re.match('<.\*>', s).span()  
(0, 32)  
>>> print re.match('<.\*>', s).group()  
<html><head><title>Title</title>

RE 匹配 在 "<html>" 中的 "<"，.\* 消耗掉子符串的剩馀部分。在 RE 中保持更多的左，虽然 > 不能匹配在字符串结尾，因此正则表达式必须一个字符一个字符地回溯，直到它找到 > 的匹配。最终的匹配从 "<html" 中的 "<" 到 "</title>" 中的 ">",这并不是你所想要的结果。  
 在这种情况下，解决方案是使用不贪婪的限定符 \*?、+?、?? 或 {m,n}?，尽可能匹配小的文本。在上面的例子里， ">" 在第一个 "<" 之後被立即尝试，当它失败时，引擎一次增加一个字符，并在每步重试 ">"。这个处理将得到正确的结果：

#!python  
>>> print re.match('<.\*?>', s).group()  
<html>

注意用正则表达式分析 HTML 或 XML 是痛苦的。变化混乱的模式将处理常见情况，但 HTML 和 XML 则是明显会打破正则表达式的特殊情况；当你编写一个正则表达式去处理所有可能的情况时，模式将变得非常复杂。象这样的任务用 HTML 或 XML 解析器。

**不用 re.VERBOSE**

现在你可能注意到正则表达式的表示是十分紧凑，但它们非常不好读。中度复杂的 REs 可以变成反斜杠、圆括号和元字符的长长集合，以致于使它们很难读懂。  
 在这些 REs 中，当编译正则表达式时指定 re.VERBOSE 标志是有帮助的，因为它允许你可以编辑正则表达式的格式使之更清楚。  
 re.VERBOSE 标志有这么几个作用。在正则表达式中不在字符类中的空白符被忽略。这就意味着象 dog | cat 这样的表达式和可读性差的 dog|cat 相同，但 [a b] 将匹配字符 "a"、"b" 或 空格。另外，你也可以把注释放到 RE 中；注释是从 "#" 到下一行。当使用三引号字符串时，可以使 REs 格式更加干净：

#!python  
pat = re.compile(r"""  
"s\*                 # Skip leading whitespac  
(?P<header>[^:]+)   # Header name  
"s\* :               # Whitespace, and a colon  
(?P<value>.\*?)      # The header's value -- \*? used to  
# lose the following trailing whitespace  
"s\*$                # Trailing whitespace to end-of-line  
""", re.VERBOSE)

这个要难读得多：

#!python  
pat = re.compile(r""s\*(?P<header>[^:]+)"s\*:(?P<value>.\*?)"s\*$")

### Python网络编程

1. Socket套接字的概念

Socket,是操作系统内核中的一个数据结构，它是网络中的节点进行相互通信的门户。它是网络进程的ID。网络通信，归根到底还是进程间的通信（不同计算机上的进程间通信, 又称进程间通信, IP协议进行的主要是端到端通信）。在网络中，每一个节点（计算机或路由）都有一个网络地址，也就是IP地址。两个进程通信时，首先要确定各自所在的网络节点的网络地址。但是，网络地址只能确定进程所在的计算机，而一台计算机上很可能同时运行着多个进程，所以仅凭网络地址还不能确定到底是和网络中的哪一个进程进行通信，因此套接口中还需要包括其他的信息，也就是端口号（PORT）。在一台计算机中，一个端口号一次只能分配给一个进程，也就是说，在一台计算机中，端口号和进程之间是一一对应关系。  
所以，使用端口号和网络地址的组合可以唯一的确定整个网络中的一个网络进程.

端口号的范围从0~65535，一类是由互联网指派名字和号码公司ICANN负责分配给一些常用的应用程序固定使用的“周知的端口”，其值一般为0~1023, 用户自定义端口号一般大于等于1024, 我比较喜欢用8888

每一个socket都用一个半相关描述{协议、本地地址、本地端口}来表示；一个完整的套接字则用一个相关描述{协议、本地地址、本地端口、远程地址、远程端口}来表示。socket也有一个类似于打开文件的函数调用，该函数返回一个整型的socket描述符，随后的连接建立、数据传输等操作都是通过socket来实现的

* 流式socket（SOCK\_STREAM） 用于TCP通信

流式套接字提供可靠的、面向连接的通信流；它使用TCP协议，从而保证了数据传输的正确性和顺序性

* 数据报socket（SOCK\_DGRAM） 用于UDP通信

数据报套接字定义了一种无连接的服务，数据通过相互独立的报文进行传输，是无序的，并且不保证是可靠、无差错的。它使用数据报协议UDP

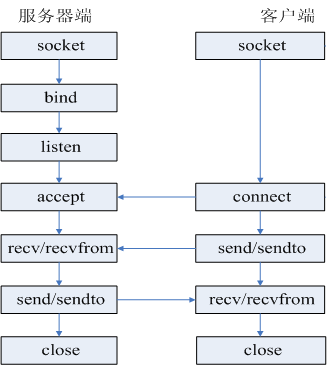
* 原始socket（SOCK\_RAW） 用于新的网络协议实现的测试等

原始套接字，普通的套接字无法处理ICMP、IGMP等网络报文，而SOCK\_RAW可以, 其次，SOCK\_RAW也可以处理特殊的IPv4报文；此外，利用原始套接字，可以通过IP\_HDRINCL套接字选项由用户构造IP头。

**2. Socket编程**

**2.1. TCP通信**

TCP通信的基本步骤如下：  
服务端：socket---bind---listen---while(True){---accept---recv---send----}---close  
客户端：socket----------------------------------connect---send---recv-------close



TCP

**socket函数**  
使用给定的地址族、套接字类型、协议编号（默认为0）来创建套接字

#Linux

int socket(int domain, int type, int protocol);

domain：AF\_INET：Ipv4网络协议 AF\_INET6：IPv6网络协议

type : tcp：SOCK\_STREAM udp：SOCK\_DGRAM

protocol : 指定socket所使用的传输协议编号。通常为0.

返回值：成功则返回套接口描述符，失败返回-1。

#python

socket.socket([family[, type[, proto]]])

family : AF\_INET (默认ipv4), AF\_INET6(ipv6) or AF\_UNIX(Unix系统进程间通信).

type : SOCK\_STREAM (TCP), SOCK\_DGRAM(UDP) .

protocol : 一般为0或者默认

如果socket创建失败会抛出一个socket.error异常

**2.1.1. 服务器端函数**

**bind函数**  
将套接字绑定到地址, python下,以元组（host,port）的形式表示地址, Linux下使用sockaddr\_in结构体指针

#Linux

int bind(int sockfd, struct sockaddr \* my\_addr, int addrlen);

sockfd : 前面socket()的返回值

my\_addr : 结构体指针变量

#####

struct sockaddr\_in //常用的结构体

{

unsigned short int sin\_family; //即为sa\_family AF\_INET

uint16\_t sin\_port; //为使用的port编号

struct in\_addr sin\_addr; //为IP地址

unsigned char sin\_zero[8]; //未使用

};

struct in\_addr

{

uint32\_t s\_addr;

};

####

addrlen : sockaddr的结构体长度。通常是计算sizeof(struct sockaddr);

返回值：成功则返回0，失败返回-1

#python

s.bind(address)

s为socket.socket()返回的套接字对象

address为元组（host,port）

host: ip地址, 为一个字符串

post: 自定义主机号, 为整型

**listen函数**  
使服务器的这个端口和IP处于监听状态，等待网络中某一客户机的连接请求。如果客户端有连接请求，端口就会接受这个连接

#Linux

int listen(int sockfd,int backlog);

sockfd : 为前面socket的返回值.

backlog : 指定同时能处理的最大连接要求，通常为10或者5。最大值可设至128

返回值：成功则返回0，失败返回-1

#python

s.listen(backlog)

s为socket.socket()返回的套接字对象

backlog : 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

**accept函数**  
接受远程计算机的连接请求，建立起与客户机之间的通信连接。服务器处于监听状态时，如果某时刻获得客户机的连接请求，此时并不是立即处理这个请求，而是将这个请求放在等待队列中，当系统空闲时再处理客户机的连接请求。

#Linux

int accept(int s,struct sockaddr \* addr,int \* addrlen);

sockfd : 为前面socket的返回值.

addr : 为结构体指针变量，和bind的结构体是同种类型的，系统会把远程主机的信息（远程主机的地址和端口号信息）保存到这个指针所指的结构体中。

addrlen : 表示结构体的长度，为整型指针

返回值：成功则返回新的socket处理代码new\_fd，失败返回-1

#python

s.accept()

s为socket.socket()返回的套接字对象

返回（conn,address）,其中conn是新的套接字对象，可以用来接收和发送数据。address是连接客户端的地址

**2.1.2. 客户端函数**

**connect函数**  
用来请求连接远程服务器

#Linux

int connect (int sockfd,struct sockaddr \* serv\_addr,int addrlen);

sockfd : 为前面socket的返回值.

serv\_addr : 为结构体指针变量，存储着远程服务器的IP与端口号信息

addrlen : 表示结构体变量的长度

返回值：成功则返回0，失败返回-1

#python

s.connect(address)

s为socket.socket()返回的套接字对象

address : 格式为元组（hostname,port），如果连接出错，返回socket.error错误

**2.1.3. 通用函数**

接收远端主机传来的数据

**recv函数**

#Linux

int recv(int sockfd,void \*buf,int len,unsigned int flags);

sockfd : 为前面accept的返回值.也就是新的套接字。

buf : 表示缓冲区

len : 表示缓冲区的长度

flags : 通常为0

返回值：成功则返回实际接收到的字符数，可能会少于你所指定的接收长度。失败返回-1

#python

s.recv(bufsize[,flag])

s为socket.socket()返回的套接字对象

bufsize : 指定要接收的数据大小

flag : 提供有关消息的其他信息，通常可以忽略

返回值为数据以字符串形式

**send函数**  
发送数据给指定的远端主机

#Linux

int send(int s,const void \* msg,int len,unsigned int flags);

sockfd : 为前面socket的返回值.

msg : 一般为常量字符串

len : 表示长度

flags : 通常为0

返回值：成功则返回实际传送出去的字符数，可能会少于你所指定的发送长度。失败返回-1

#python

s.send(string[,flag])

s为socket.socket()返回的套接字对象

string : 要发送的字符串数据

flag : 提供有关消息的其他信息，通常可以忽略

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(string[,flag])

#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。

返回值 : 成功返回None，失败则抛出异常。

**close函数**  
关闭套接字

#Linux

int close(int fd);

fd : 为前面的sockfd

返回值：若文件顺利关闭则返回0，发生错误时返回-1

#python

s.close()

s为socket.socket()返回的套接字对象

**2.2. 简单的客户端服务器TCP连接**

一个简单的回显服务器和客户端模型, 客户端发出的数据, 服务器会回显到客户端的终端上(只是一个简单的模型, 没考虑错误处理等问题)

#服务器端

#!/usr/bin/env python

# -\*- coding:utf-8 -\*-

import socket #socket模块

import commands #执行系统命令模块

BUF\_SIZE = 1024 #设置缓冲区大小

server\_addr = ('127.0.0.1', 8888) #IP和端口构成表示地址

server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) #生成一个新的socket对象

server.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1) #设置地址复用

server.bind(server\_addr) #绑定地址

server.listen(5) #监听, 最大监听数为5

while True:

client, client\_addr = server.accept() #接收TCP连接, 并返回新的套接字和地址

print 'Connected by', client\_addr

while True :

data = client.recv(BUF\_SIZE) #从客户端接收数据

print data

client.sendall(data) #发送数据到客户端

server.close()

#客户端

#!/usr/bin/env python

# -\*- coding:utf-8 -\*-

import socket

BUF\_SIZE = 1024 #设置缓冲区的大小

server\_addr = ('127.0.0.1', 8888) #IP和端口构成表示地址

client = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) #返回新的socket对象

client.connect(server\_addr) #要连接的服务器地址

while True:

data = raw\_input("Please input some string > ")

client.sendall(data) #发送数据到服务器

data = client.recv(BUF\_SIZE) #从服务器端接收数据

print data

client.close()

**2.2.1. 带错误处理的客户端服务器TCP连接**

在进行网络编程时, 最好使用大量的错误处理, 能够尽量的发现错误, 也能够使代码显得更加严谨

#服务器端

#!/usr/bin/env python

# -\*- coding:utf-8 -\*-

import sys

import socket #socket模块

BUF\_SIZE = 1024 #设置缓冲区大小

server\_addr = ('127.0.0.1', 8888) #IP和端口构成表示地址

try :

server = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) #生成一个新的socket对象

except socket.error, msg :

print "Creating Socket Failure. Error Code : " + str(msg[0]) + " Message : " + msg[1]

sys.exit()

print "Socket Created!"

server.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1) #设置地址复用

try :

server.bind(server\_addr) #绑定地址

except socket.error, msg :

print "Binding Failure. Error Code : " + str(msg[0]) + " Message : " + msg[1]

sys.exit()

print "Socket Bind!"

server.listen(5) #监听, 最大监听数为5

print "Socket listening"

while True:

client, client\_addr = server.accept() #接收TCP连接, 并返回新的套接字和地址, 阻塞函数

print 'Connected by', client\_addr

while True :

data = client.recv(BUF\_SIZE) #从客户端接收数据

print data

client.sendall(data) #发送数据到客户端

server.close()

#客户端

#!/usr/bin/env python

# -\*- coding:utf-8 -\*-

import sys

import socket

BUF\_SIZE = 1024 #设置缓冲区的大小

server\_addr = ('127.0.0.1', 8888) #IP和端口构成表示地址

try :

client = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) #返回新的socket对象

except socket.error, msg :

print "Creating Socket Failure. Error Code : " + str(msg[0]) + " Message : " + msg[1]

sys.exit()

client.connect(server\_addr) #要连接的服务器地址

while True:

data = raw\_input("Please input some string > ")

if not data :

print "input can't empty, Please input again.."

continue

client.sendall(data) #发送数据到服务器

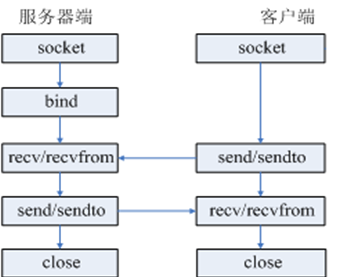
data = client.recv(BUF\_SIZE) #从服务器端接收数据

print data

client.close()

**2.3. UDP通信**

UDP通信流程图如下：  
服务端：socket---bind---recvfrom---sendto---close  
客户端：socket----------sendto---recvfrom---close



UDP

**sendto()函数**

发送UDP数据, 将数据发送到套接字

#Linux

int sendto(int sockfd, const void \*msg,int len,unsigned int flags,const struct sockaddr \*to, int tolen);

sockfd : 为前面socket的返回值.

msg : 一般为常量字符串

len : 表示长度

flags : 通常为0

to : 表示目地机的IP地址和端口号信息, 表示地址的结构体

tolen : 常常被赋值为sizeof (struct sockaddr)

返回值 :　返回实际发送的数据字节长度或在出现发送错误时返回-1。

#Python

s.sendto(string[,flag],address)

s为socket.socket()返回的套接字对象

address : 指定远程地址, 形式为（ipaddr，port）的元组

flag : 提供有关消息的其他信息，通常可以忽略

返回值 : 发送的字节数。

**recvfrom()函数**  
接受UDP套接字的数据, 与recv()类似

#Linux

int recvfrom(int sockfd,void \*buf,int len,unsigned int flags,struct sockaddr \*from,int \*fromlen);

sockfd : 为前面socket的返回值.

msg : 一般为常量字符串

len : 表示长度

flags : 通常为0

from :是一个struct sockaddr类型的变量，该变量保存连接机的IP地址及端口号

fromlen : 常置为sizeof (struct sockaddr)。

返回值 : 返回接收到的字节数或当出现错误时返回-1，并置相应的errno。

#Python

s.recvfrom(bufsize[.flag])

返回值 : （data,address）元组, 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

bufsize : 指定要接收的数据大小

flag : 提供有关消息的其他信息，通常可以忽略

**2.4. 简单的客户端服务器UDP连接**

#服务器端

#!/usr/bin/env python

# -\*- coding:utf-8 -\*-

import socket

BUF\_SIZE = 1024 #设置缓冲区大小

server\_addr = ('127.0.0.1', 8888) #IP和端口构成表示地址

server = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM) #生成新的套接字对象

server.bind(server\_addr) #套接字绑定IP和端口

while True :

print "waitting for data"

data, client\_addr = server.recvfrom(BUF\_SIZE) #从客户端接收数据

print 'Connected by', client\_addr, ' Receive Data : ', data

server.sendto(data, client\_addr) #发送数据给客户端

server.close()

#客户端

#!/usr/bin/env python

# -\*- coding:utf-8 -\*-

import socket

import struct

BUF\_SIZE = 1024 #设置缓冲区

server\_addr = ('127.0.0.1', 8888) #IP和端口构成表示地址

client = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM) #生成新的套接字对象

while True :

data = raw\_input('Please Input data > ')

client.sendto(data, server\_addr) #向服务器发送数据

data, addr = client.recvfrom(BUF\_SIZE) #从服务器接收数据

print "Data : ", data

client.close()

**2.5. 其他**

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

Python多线程编程

**1. thread模块**

* python是支持多线程的, 主要是通过thread和threading这两个模块来实现的。
* python的thread模块是比较底层的模块(或者说轻量级)，python的threading模块是对thread做了一些包装的，可以更加方便的被使用。

简要的看一下thread模块中含函数和常量

import thread

thread.LockType #锁对象的一种, 用于线程的同步

thread.error #线程的异常

thread.start\_new\_thread(function, args[, kwargs]) #创建一个新的线程

function : 线程执行函数

args : 线程执行函数的参数, 类似为tuple,

kwargs : 是一个字典

返回值: 返回线程的标识符

thread.exit() #线程退出函数

thread.allocate\_lock() #生成一个未锁状态的锁对象

返回值: 返回一个锁对象

锁对象的方法

lock.acquire([waitflag]) #获取锁

无参数时, 无条件获取锁, 无法获取时, 会被阻塞, 知道可以锁被释放

有参数时, waitflag = 0 时,表示只有在不需要等待的情况下才获取锁, 非零情况与上面相同

返回值 :　获得锁成功返回True, 获得锁失败返回False

lock.release() #释放锁

lock.locked() #获取当前锁的状态

返回值 : 如果锁已经被某个线程获取,返回True, 否则为False

**1.1. thread多线程**

#!/usr/bin/env python

# -\*- coding:utf-8 -\*-

import thread

import time

def print\_time(thread\_name, delay) :

count = 0

while count < 5 :

time.sleep(delay)

count += 1

print "%s : %s" % (thread\_name, time.ctime(time.time()))

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 the thread"

while True :

pass

**2. threading模块**

python的threading模块是对thread做了一些包装的，可以更加方便的被使用。经常和[Queue](https://docs.python.org/2/library/queue.html)结合使用,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

**2.1. 常用函数和对象**

#函数

threading.active\_count() #返回当前线程对象Thread的个数

threading.enumerate() #返回当前运行的线程对象Thread(包括后台的)的list

threading.Condition() #返回条件变量对象的工厂函数, 主要用户线程的并发

threading.current\_thread() #返回当前的线程对象Thread, 文档后面解释没看懂

threading.Lock() #返回一个新的锁对象, 是在thread模块的基础上实现的 与acquire()和release()结合使用

#类

threading.Thread #一个表示线程控制的类, 这个类常被继承

thraeding.Timer #定时器,线程在一定时间后执行

threading.ThreadError #引发中各种线程相关异常

**2.1.1. Thread对象**

一般来说，使用线程有两种模式, 一种是创建线程要执行的函数, 把这个函数传递进Thread对象里，让它来执行. 另一种是直接从Thread继承，创建一个新的class，把线程执行的代码放到这个新的class里。

常用两种方式运行线程(线程中包含name属性) :

* 在构造函数中传入用于线程运行的函数(这种方式更加灵活)
* 在子类中重写threading.Thread基类中run()方法(只重写\_\_init\_\_()和run()方法)

创建线程对象后, 通过调用start()函数运行线程, 然后会自动调用run()方法.

通过设置｀daemon｀属性, 可以将线程设置为守护线程

threading.Thread(group = None, target = None, name = None, args = () kwars = {})

group : 应该为None

target : 可以传入一个函数用于run()方法调用,

name : 线程名 默认使用"Thread-N"

args : 元组, 表示传入target函数的参数

kwargs : 字典, 传入target函数中关键字参数

属性:

name #线程表示, 没有任何语义

doemon #布尔值, 如果是守护线程为True, 不是为False, 主线程不是守护线程, 默认threading.Thread.damon = False

类方法:

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。

范例:

#!/usr/bin/env python

# -\*- coding:utf-8 -\*-

import threading

import time

def test\_thread(count) :

while count > 0 :

print "count = %d" % count

count = count - 1

time.sleep(1)

def main() :

my\_thread = threading.Thread(target = test\_thread, args = (10, ))

my\_thread.start()

my\_thread.join()

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

main()

**2.2. 常用多线程写法**

* 固定线程运行的函数

#!/usr/bin/env python

# -\*- coding:utf-8 -\*-

import threading, thread

import time

class MyThread(threading.Thread):

"""docstring for MyThread"""

def \_\_init\_\_(self, thread\_id, name, counter) :

super(MyThread, self).\_\_init\_\_() #调用父类的构造函数

self.thread\_id = thread\_id

self.name = name

self.counter = counter

def run(self) :

print "Starting " + self.name

print\_time(self.name, self.counter, 5)

print "Exiting " + self.name

def print\_time(thread\_name, delay, counter) :

while counter :

time.sleep(delay)

print "%s %s" % (thread\_name, time.ctime(time.time()))

counter -= 1

def main():

#创建新的线程

thread1 = MyThread(1, "Thread-1", 1)

thread2 = MyThread(2, "Thread-2", 2)

#开启线程

thread1.start()

thread2.start()

thread1.join()

thread2.join()

print "Exiting Main Thread"

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

main()

* 外部传入线程运行的函数

#/usr/bin/env python

# -\*- coding: utf-8 -\*-

import threading

import time

class MyThread(threading.Thread):

"""

属性:

target: 传入外部函数, 用户线程调用

args: 函数参数

"""

def \_\_init\_\_(self, target, args):

super(MyThread, self).\_\_init\_\_() #调用父类的构造函数

self.target = target

self.args = args

def run(self) :

self.target(self.args)

def print\_time(counter) :

while counter :

print "counter = %d" % counter

counter -= 1

time.sleep(1)

def main() :

my\_thread = MyThread(print\_time, 10)

my\_thread.start()

my\_thread.join()

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

main()

**2.3. 生产者消费者问题**

试着用python写了一个生产者消费者问题(伪生产者消费者), 只是使用简单的锁, 感觉有点不太对, 下面另一个程序会写出正确的生产者消费者问题

#!/usr/bin/env python

# -\*- coding:utf-8 -\*-

import thread, threading

import urllib2

import time, random

import Queue

share\_queue = Queue.Queue() #共享队列

my\_lock = thread.allocate\_lock()

class Producer(threading.Thread) :

def run(self) :

products = range(5)

global share\_queue

while True :

num = random.choice(products)

my\_lock.acquire()

share\_queue.put(num)

print "Produce : ", num

my\_lock.release()

time.sleep(random.random())

class Consumer(threading.Thread) :

def run(self) :

global share\_queue

while True:

my\_lock.acquire()

if share\_queue.empty() : #这里没有使用信号量机制进行阻塞等待,

print "Queue is Empty..."

my\_lock.release()

time.sleep(random.random())

continue

num = share\_queue.get()

print "Consumer : ", num

my\_lock.release()

time.sleep(random.random())

def main() :

producer = Producer()

consumer = Consumer()

producer.start()

consumer.start()

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

main()

杀死多线程程序方法: 使用control + z挂起程序(程序依然在后台, 可以使用ps aux查看), 获得程序的进程号, 然后使用kill -9 进程号杀死进程

参考一篇帖子解决了上述问题,重写了生产者消费者问题程序, 参考链接惯例放在最后.

使用了wait()和notify()解决

当然最简答的方法是直接使用Queue,Queue封装了Condition的行为, 如wait(), notify(), acquire(), 没看文档就这样, 使用了Queue竟然不知道封装了这些函数, 继续滚去看文档了

#!/usr/bin/env python

# -\*- coding:utf-8 -\*-

import threading

import random, time, Queue

MAX\_SIZE = 5

SHARE\_Q = [] #模拟共享队列

CONDITION = threading.Condition()

class Producer(threading.Thread) :

def run(self) :

products = range(5)

global SHARE\_Q

while True :

CONDITION.acquire()

if len(SHARE\_Q) == 5 :

print "Queue is full.."

CONDITION.wait()

print "Consumer have comsumed something"

product = random.choice(products)

SHARE\_Q.append(product)

print "Producer : ", product

CONDITION.notify()

CONDITION.release()

time.sleep(random.random())

class Consumer(threading.Thread) :

def run(self) :

global SHARE\_Q

while True:

CONDITION.acquire()

if not SHARE\_Q :

print "Queue is Empty..."

CONDITION.wait()

print "Producer have producted something"

product = SHARE\_Q.pop(0)

print "Consumer :", product

CONDITION.notify()

CONDITION.release()

time.sleep(random.random())

def main() :

producer = Producer()

consumer = Consumer()

producer.start()

consumer.start()

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

main()

**2.4.简单锁**

如果只是简单的加锁解锁可以直接使用threading.Lock()生成锁对象, 然后使用acquire()和release()方法

例如:

#!/usr/bin/env python

# -\*- coding:utf-8 -\*-

import threading

import time

class MyThread(threading.Thread) :

def \_\_init\_\_(self, thread\_id, name, counter) :

threading.Thread.\_\_init\_\_(self)

self.thread\_id = thread\_id

self.name = name

self.counter = counter

def run(self) :

#重写run方法, 添加线程执行逻辑, start函数运行会自动执行

print "Starting " + self.name

threadLock.acquire() #获取所

print\_time(self.name, self.counter, 3)

threadLock.release() #释放锁

def print\_time(thread\_name, delay, counter) :

while counter :

time.sleep(delay)

print "%s %s" % (thread\_name, time.ctime(time.time()))

counter -= 1

threadLock = threading.Lock()

threads = [] #存放线程对象

thread1 = MyThread(1, "Thread-1", 1)

thread2 = MyThread(2, "Thread-2", 2)

#开启线程

thread1.start()

thread2.start()

for t in threads :

t.join() #等待线程直到终止

print "Exiting Main Thread"

**2.5. Condition**

如果是向生产者消费者类似的情形, 使用Condition类 或者直接使用Queue模块

**Condition**

条件变量中有acquire()和release方法用来调用锁的方法, 有wait(), notify(), notifyAll()方法, 后面是三个方法必须在获取锁的情况下调用, 否则产生RuntimeError错误.

* 当一个线程获得锁后, 发现没有期望的资源或者状态, 就会调用wait()阻塞, 并释放已经获得锁, 知道期望的资源或者状态发生改变
* 当一个线程获得锁, 改变了资源或者状态, 就会调用notify()和notifyAll()去通知其他线程,

#官方文档中提供的生产者消费者模型

# Consume one item

cv.acquire()

while not an\_item\_is\_available():

cv.wait()

get\_an\_available\_item()

cv.release()

# Produce one item

cv.acquire()

make\_an\_item\_available()

cv.notify()

cv.release()

#threading.Condition类

thread.Condition([lock])

可选参数lock: 必须是Lock或者RLock对象, 并被作为underlying锁(悲观锁?), 否则, 会创建一个新的RLock对象作为underlying锁

类方法:

acquire() #获得锁

release() #释放锁

wait([timeout]) #持续等待直到被notify()或者notifyAll()通知或者超时(必须先获得锁),

#wait()所做操作, 先释放获得的锁, 然后阻塞, 知道被notify或者notifyAll唤醒或者超时, 一旦被唤醒或者超时, 会重新获取锁(应该说抢锁), 然后返回

notify() #唤醒一个wait()阻塞的线程.

notify\_all()或者notifyAll() #唤醒所有阻塞的线程

**1. Queue模块**

Queue模块实现了多生产者多消费者队列, 尤其适合多线程编程.Queue类中实现了所有需要的锁原语(这句话非常重要), Queue模块实现了三种类型队列:

 FIFO(先进先出)队列, 第一加入队列的任务, 被第一个取出

 LIFO(后进先出)队列,最后加入队列的任务, 被第一个取出(操作类似与栈, 总是从栈顶取出, 这个队列还不清楚内部的实现)

 PriorityQueue(优先级)队列, 保持队列数据有序, 最小值被先取出(在C++中我记得优先级队列是可以自己重写排序规则的, Python不知道可以吗)

**1.1. 类和异常**

import Queue

#类

Queue.Queue(maxsize = 0) #构造一个FIFO队列,maxsize设置队列大小的上界, 如果插入数据时, 达到上界会发生阻塞, 直到队列可以放入数据. 当maxsize小于或者等于0, 表示不限制队列的大小(默认)

Queue.LifoQueue(maxsize = 0) #构造一LIFO队列,maxsize设置队列大小的上界, 如果插入数据时, 达到上界会发生阻塞, 直到队列可以放入数据. 当maxsize小于或者等于0, 表示不限制队列的大小(默认)

Queue.PriorityQueue(maxsize = 0) #构造一个优先级队列,,maxsize设置队列大小的上界, 如果插入数据时, 达到上界会发生阻塞, 直到队列可以放入数据. 当maxsize小于或者等于0, 表示不限制队列的大小(默认). 优先级队列中, 最小值被最先取出

#异常

Queue.Empty #当调用非阻塞的get()获取空队列的元素时, 引发异常

Queue.Full #当调用非阻塞的put()向满队列中添加元素时, 引发异常

**1.2. Queue对象**

三种队列对象提供公共的方法

Queue.empty() #如果队列为空, 返回True(注意队列为空时, 并不能保证调用put()不会阻塞); 队列不空返回False(不空时, 不能保证调用get()不会阻塞)

Queue.full() #如果队列为满, 返回True(不能保证调用get()不会阻塞), 如果队列不满, 返回False(并不能保证调用put()不会阻塞)

Queue.put(item[, block[, timeout]]) #向队列中放入元素, 如果可选参数block为True并且timeout参数为None(默认), 为阻塞型put(). 如果timeout是正数, 会阻塞timeout时间并引发Queue.Full异常. 如果block为False为非阻塞put

Queue.put\_nowait(item) #等价于put(itme, False)

Queue.get([block[, timeout]]) #移除列队元素并将元素返回, block = True为阻塞函数, block = False为非阻塞函数. 可能返回Queue.Empty异常

Queue.get\_nowait() #等价于get(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

下面是官方文档给多出的多线程模型:

def worker():

while True:

item = q.get()

do\_work(item)

q.task\_done()

q = Queue()

for i in range(num\_worker\_threads):

t = Thread(target=worker)

t.daemon = True

t.start()

for item in source():

q.put(item)

q.join() # block until all tasks are done

**2. Queue模块与线程相结合**

简单#!/usr/bin/env python

# -\*- coding:utf-8 -\*-

import threading

import time

import Queue

SHARE\_Q = Queue.Queue() #构造一个不限制大小的的队列

\_WORKER\_THREAD\_NUM = 3 #设置线程个数

class MyThread(threading.Thread) :

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

super(MyThread, self).\_\_init\_\_()

self.func = func

def run(self) :

self.func()

def worker() :

global SHARE\_Q

while not SHARE\_Q.empty():

item = SHARE\_Q.get() #获得任务

print "Processing : ", item

time.sleep(1)

def main() :

global SHARE\_Q

threads = []

for task in xrange(5) : #向队列中放入任务

SHARE\_Q.put(task)

for i in xrange(\_WORKER\_THREAD\_NUM) :

thread = MyThread(worker)

thread.start()

threads.append(thread)

for thread in threads :

thread.join()

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

main()

**Scrapy框架学习**

**1. Scrapy简介**

Scrapy是一个为了爬取网站数据，提取结构性数据而编写的应用框架。 可以应用在包括数据挖掘，信息处理或存储历史数据等一系列的程序中。  
其最初是为了页面抓取 (更确切来说, 网络抓取 )所设计的， 也可以应用在获取API所返回的数据(例如 Amazon Associates Web Services ) 或者通用的网络爬虫。Scrapy用途广泛，可以用于数据挖掘、监测和自动化测试

Scrapy 使用了 Twisted异步网络库来处理网络通讯。整体架构大致如下



*Scrapy*

Scrapy主要包括了以下组件：

 引擎(Scrapy): 用来处理整个系统的数据流处理, 触发事务(框架核心)

 调度器(Scheduler): 用来接受引擎发过来的请求, 压入队列中, 并在引擎再次请求的时候返回. 可以想像成一个URL（抓取网页的网址或者说是链接）的优先队列, 由它来决定下一个要抓取的网址是什么, 同时去除重复的网址

 下载器(Downloader): 用于下载网页内容, 并将网页内容返回给蜘蛛(Scrapy下载器是建立在twisted这个高效的异步模型上的)

 爬虫(Spiders): 爬虫是主要干活的, 用于从特定的网页中提取自己需要的信息, 即所谓的实体(Item)。用户也可以从中提取出链接,让Scrapy继续抓取下一个页面

 项目管道(Pipeline): 负责处理爬虫从网页中抽取的实体，主要的功能是持久化实体、验证实体的有效性、清除不需要的信息。当页面被爬虫解析后，将被发送到项目管道，并经过几个特定的次序处理数据。

 下载器中间件(Downloader Middlewares): 位于Scrapy引擎和下载器之间的框架，主要是处理Scrapy引擎与下载器之间的请求及响应。

 爬虫中间件(Spider Middlewares): 介于Scrapy引擎和爬虫之间的框架，主要工作是处理蜘蛛的响应输入和请求输出。

 调度中间件(Scheduler Middewares): 介于Scrapy引擎和调度之间的中间件，从Scrapy引擎发送到调度的请求和响应。

**Scrapy运行流程大概如下：**

1. 首先，引擎从调度器中取出一个链接(URL)用于接下来的抓取
2. 引擎把URL封装成一个请求(Request)传给下载器，下载器把资源下载下来，并封装成应答包(Response)
3. 然后，爬虫解析Response
4. 若是解析出实体（Item）,则交给实体管道进行进一步的处理。
5. 若是解析出的是链接（URL）,则把URL交给Scheduler等待抓取

**2. 安装Scrapy**

使用以下命令

sudo pip install virtualenv #安装虚拟环境工具

virtualenv ENV #创建一个虚拟环境目录

source ./ENV/bin/active #激活虚拟环境

pip install Scrapy

#验证是否安装成功

pip list

#输出如下

cffi (0.8.6)

cryptography (0.6.1)

cssselect (0.9.1)

lxml (3.4.1)

pip (1.5.6)

pycparser (2.10)

pyOpenSSL (0.14)

queuelib (1.2.2)

Scrapy (0.24.4)

setuptools (3.6)

six (1.8.0)

Twisted (14.0.2)

w3lib (1.10.0)

wsgiref (0.1.2)

zope.interface (4.1.1)

**3. Scrapy Tutorial**

在抓取之前, 你需要新建一个Scrapy工程. 进入一个你想用来保存代码的目录，然后执行：

$ scrapy startproject tutorial

这个命令会在当前目录下创建一个新目录 tutorial, 它的结构如下:

.

├── scrapy.cfg

└── tutorial

├── \_\_init\_\_.py

├── items.py

├── pipelines.py

├── settings.py

└── spiders

└── \_\_init\_\_.py

这些文件主要是：

* scrapy.cfg: 项目配置文件
* tutorial/: 项目python模块, 之后您将在此加入代码
* tutorial/items.py: 项目items文件
* tutorial/pipelines.py: 项目管道文件
* tutorial/settings.py: 项目配置文件
* tutorial/spiders: 放置spider的目录

**3.1. 定义Item**

Items是将要装载抓取的数据的容器，它工作方式像 python 里面的字典，但它提供更多的保护，比如对未定义的字段填充以防止拼写错误

通过创建scrapy.Item类, 并且定义类型为 scrapy.Field 的类属性来声明一个Item.

我们通过将需要的item模型化，来控制从 dmoz.org 获得的站点数据，比如我们要获得站点的名字，url 和网站描述，我们定义这三种属性的域。在 tutorial 目录下的 items.py 文件编辑

from scrapy.item import Item, Field

class DmozItem(Item):

# define the fields for your item here like:

name = Field()

description = Field()

url = Field()

**3.2. 编写Spider**

Spider 是用户编写的类, 用于从一个域（或域组）中抓取信息, 定义了用于下载的URL的初步列表, 如何跟踪链接，以及如何来解析这些网页的内容用于提取items。

要建立一个 Spider，继承 scrapy.Spider 基类，并确定三个主要的、强制的属性：

 name：爬虫的识别名，它必须是唯一的，在不同的爬虫中你必须定义不同的名字.

 start\_urls：包含了Spider在启动时进行爬取的url列表。因此，第一个被获取到的页面将是其中之一。后续的URL则从初始的URL获取到的数据中提取。我们可以利用正则表达式定义和过滤需要进行跟进的链接。

 parse()：是spider的一个方法。被调用时，每个初始URL完成下载后生成的 Response 对象将会作为唯一的参数传递给该函数。该方法负责解析返回的数据(response data)，提取数据(生成item)以及生成需要进一步处理的URL的 Request 对象。

这个方法负责解析返回的数据、匹配抓取的数据(解析为 item )并跟踪更多的 URL。

在 /tutorial/tutorial/spiders 目录下创建 dmoz\_spider.py

import scrapy

class DmozSpider(scrapy.Spider):

name = "dmoz"

allowed\_domains = ["dmoz.org"]

start\_urls = [

"http://www.dmoz.org/Computers/Programming/Languages/Python/Books/",

"http://www.dmoz.org/Computers/Programming/Languages/Python/Resources/"

]

def parse(self, response):

filename = response.url.split("/")[-2]

with open(filename, 'wb') as f:

f.write(response.body)

**3.3. 爬取**

当前项目结构

├── scrapy.cfg

└── tutorial

├── \_\_init\_\_.py

├── items.py

├── pipelines.py

├── settings.py

└── spiders

├── \_\_init\_\_.py

└── dmoz\_spider.py

到项目根目录, 然后运行命令:

$ scrapy crawl dmoz

运行结果:

2014-12-15 09:30:59+0800 [scrapy] INFO: Scrapy 0.24.4 started (bot: tutorial)

2014-12-15 09:30:59+0800 [scrapy] INFO: Optional features available: ssl, http11

2014-12-15 09:30:59+0800 [scrapy] INFO: Overridden settings: {'NEWSPIDER\_MODULE': 'tutorial.spiders', 'SPIDER\_MODULES': ['tutorial.spiders'], 'BOT\_NAME': 'tutorial'}

2014-12-15 09:30:59+0800 [scrapy] INFO: Enabled extensions: LogStats, TelnetConsole, CloseSpider, WebService, CoreStats, SpiderState

2014-12-15 09:30:59+0800 [scrapy] INFO: Enabled downloader middlewares: HttpAuthMiddleware, DownloadTimeoutMiddleware, UserAgentMiddleware, RetryMiddleware, DefaultHeadersMiddleware, MetaRefreshMiddleware, HttpCompressionMiddleware, RedirectMiddleware, CookiesMiddleware, ChunkedTransferMiddleware, DownloaderStats

2014-12-15 09:30:59+0800 [scrapy] INFO: Enabled spider middlewares: HttpErrorMiddleware, OffsiteMiddleware, RefererMiddleware, UrlLengthMiddleware, DepthMiddleware

2014-12-15 09:30:59+0800 [scrapy] INFO: Enabled item pipelines:

2014-12-15 09:30:59+0800 [dmoz] INFO: Spider opened

2014-12-15 09:30:59+0800 [dmoz] INFO: Crawled 0 pages (at 0 pages/min), scraped 0 items (at 0 items/min)

2014-12-15 09:30:59+0800 [scrapy] DEBUG: Telnet console listening on 127.0.0.1:6023

2014-12-15 09:30:59+0800 [scrapy] DEBUG: Web service listening on 127.0.0.1:6080

2014-12-15 09:31:00+0800 [dmoz] DEBUG: Crawled (200) <GET http://www.dmoz.org/Computers/Programming/Languages/Python/Resources/> (referer: None)

2014-12-15 09:31:00+0800 [dmoz] DEBUG: Crawled (200) <GET http://www.dmoz.org/Computers/Programming/Languages/Python/Books/> (referer: None)

2014-12-15 09:31:00+0800 [dmoz] INFO: Closing spider (finished)

2014-12-15 09:31:00+0800 [dmoz] INFO: Dumping Scrapy stats:

{'downloader/request\_bytes': 516,

'downloader/request\_count': 2,

'downloader/request\_method\_count/GET': 2,

'downloader/response\_bytes': 16338,

'downloader/response\_count': 2,

'downloader/response\_status\_count/200': 2,

'finish\_reason': 'finished',

'finish\_time': datetime.datetime(2014, 12, 15, 1, 31, 0, 666214),

'log\_count/DEBUG': 4,

'log\_count/INFO': 7,

'response\_received\_count': 2,

'scheduler/dequeued': 2,

'scheduler/dequeued/memory': 2,

'scheduler/enqueued': 2,

'scheduler/enqueued/memory': 2,

'start\_time': datetime.datetime(2014, 12, 15, 1, 30, 59, 533207)}

2014-12-15 09:31:00+0800 [dmoz] INFO: Spider closed (finished)

**3.4. 提取Items**

**3.4.1. 介绍Selector**

从网页中提取数据有很多方法。Scrapy使用了一种基于 XPath 或者 CSS 表达式机制： Scrapy Selectors

出XPath表达式的例子及对应的含义:

 /html/head/title: 选择HTML文档中 <head> 标签内的 <title> 元素

 /html/head/title/text(): 选择 <title> 元素内的文本

 //td: 选择所有的 <td> 元素

 //div[@class="mine"]: 选择所有具有class="mine" 属性的 div 元素

等多强大的功能使用可以查看[XPath tutorial](http://www.w3school.com.cn/xpath/)

为了方便使用 XPaths，Scrapy 提供 Selector 类， 有四种方法

 xpath()：返回selectors列表, 每一个selector表示一个xpath参数表达式选择的节点.

 css() : 返回selectors列表, 每一个selector表示CSS参数表达式选择的节点

 extract()：返回一个unicode字符串，该字符串为XPath选择器返回的数据

 re()： 返回unicode字符串列表，字符串作为参数由正则表达式提取出来

**3.4.2. 取出数据**

首先使用谷歌浏览器开发者工具, 查看网站源码, 来看自己需要取出的数据形式(这种方法比较麻烦), 更简单的方法是直接对感兴趣的东西右键审查元素, 可以直接查看网站源码

在查看网站源码后, 网站信息在第二个<ul>内

<ul class="directory-url" style="margin-left:0;">

<li><a href="http://www.pearsonhighered.com/educator/academic/product/0,,0130260363,00%2Ben-USS\_01DBC.html" class="listinglink">Core Python Programming</a>

- By Wesley J. Chun; Prentice Hall PTR, 2001, ISBN 0130260363. For experienced developers to improve extant skills; professional level examples. Starts by introducing syntax, objects, error handling, functions, classes, built-ins. [Prentice Hall]

<div class="flag"><a href="/public/flag?cat=Computers%2FProgramming%2FLanguages%2FPython%2FBooks&url=http%3A%2F%2Fwww.pearsonhighered.com%2Feducator%2Facademic%2Fproduct%2F0%2C%2C0130260363%2C00%252Ben-USS\_01DBC.html"><img src="/img/flag.png" alt="[!]" title="report an issue with this listing"></a></div>

</li>

...省略部分...

</ul>

那么就可以通过一下方式进行提取数据

#通过如下命令选择每个在网站中的 <li> 元素:

sel.xpath('//ul/li')

#网站描述:

sel.xpath('//ul/li/text()').extract()

#网站标题:

sel.xpath('//ul/li/a/text()').extract()

#网站链接:

sel.xpath('//ul/li/a/@href').extract()

#如前所述，每个 xpath() 调用返回一个 selectors 列表，所以我们可以结合 xpath() 去挖掘更深的节点。我们将会用到这些特性，所以:

for sel in response.xpath('//ul/li')

title = sel.xpath('a/text()').extract()

link = sel.xpath('a/@href').extract()

desc = sel.xpath('text()').extract()

print title, link, desc

在已有的爬虫文件中修改代码

import scrapy

class DmozSpider(scrapy.Spider):

name = "dmoz"

allowed\_domains = ["dmoz.org"]

start\_urls = [

"http://www.dmoz.org/Computers/Programming/Languages/Python/Books/",

"http://www.dmoz.org/Computers/Programming/Languages/Python/Resources/"

]

def parse(self, response):

for sel in response.xpath('//ul/li'):

title = sel.xpath('a/text()').extract()

link = sel.xpath('a/@href').extract()

desc = sel.xpath('text()').extract()

print title, link, desc

**3.4.3. 使用item**

Item对象是自定义的python字典,可以使用标准的字典语法来获取到其每个字段的值(字段即是我们之前用Field赋值的属性)

>>> item = DmozItem()

>>> item['title'] = 'Example title'

>>> item['title']

'Example title'

一般来说，Spider将会将爬取到的数据以 Item 对象返回, 最后修改爬虫类，使用 Item 来保存数据，代码如下

from scrapy.spider import Spider

from scrapy.selector import Selector

from tutorial.items import DmozItem

class DmozSpider(Spider):

name = "dmoz"

allowed\_domains = ["dmoz.org"]

start\_urls = [

"http://www.dmoz.org/Computers/Programming/Languages/Python/Books/",

"http://www.dmoz.org/Computers/Programming/Languages/Python/Resources/",

]

def parse(self, response):

sel = Selector(response)

sites = sel.xpath('//ul[@class="directory-url"]/li')

items = []

for site in sites:

item = DmozItem()

item['name'] = site.xpath('a/text()').extract()

item['url'] = site.xpath('a/@href').extract()

item['description'] = site.xpath('text()').re('-\s[^\n]\*\\r')

items.append(item)

return items

**3.5. 使用Item Pipeline**

当Item在Spider中被收集之后，它将会被传递到Item Pipeline，一些组件会按照一定的顺序执行对Item的处理。  
每个item pipeline组件(有时称之为ItemPipeline)是实现了简单方法的Python类。他们接收到Item并通过它执行一些行为，同时也决定此Item是否继续通过pipeline，或是被丢弃而不再进行处理。  
以下是item pipeline的一些典型应用：

* 清理HTML数据
* 验证爬取的数据(检查item包含某些字段)
* 查重(并丢弃)
* 将爬取结果保存，如保存到数据库、XML、JSON等文件中

　编写你自己的item pipeline很简单，每个item pipeline组件是一个独立的Python类，同时必须实现以下方法:

process\_item(item, spider) #每个item pipeline组件都需要调用该方法，这个方法必须返回一个 Item (或任何继承类)对象，或是抛出 DropItem异常，被丢弃的item将不会被之后的pipeline组件所处理。

#参数:

item: 由 parse 方法返回的 Item 对象(Item对象)

spider: 抓取到这个 Item 对象对应的爬虫对象(Spider对象)

open\_spider(spider) #当spider被开启时，这个方法被调用。

#参数:

spider : (Spider object) – 被开启的spider

close\_spider(spider) #当spider被关闭时，这个方法被调用，可以再爬虫关闭后进行相应的数据处理。

#参数:

spider : (Spider object) – 被关闭的spider

为JSON文件编写一个items

from scrapy.exceptions import DropItem

class TutorialPipeline(object):

# put all words in lowercase

words\_to\_filter = ['politics', 'religion']

def process\_item(self, item, spider):

for word in self.words\_to\_filter:

if word in unicode(item['description']).lower():

raise DropItem("Contains forbidden word: %s" % word)

else:

return item

在 settings.py 中设置ITEM\_PIPELINES激活item pipeline，其默认为[]

ITEM\_PIPELINES = {'tutorial.pipelines.FilterWordsPipeline': 1}

**3.6. 存储数据**

使用下面的命令存储为json文件格式

scrapy crawl dmoz -o items.json

<https://github.com/Andrew-liu/scrapy_example/tree/master/douban>

**Scrapy模拟登录**

**1. Cookie原理**

HTTP是无状态的面向连接的协议, 为了保持连接状态, 引入了Cookie机制

Cookie是http消息头中的一种属性，包括：

 Cookie名字（Name）Cookie的值（Value）

 Cookie的过期时间（Expires/Max-Age）

 Cookie作用路径（Path）

 Cookie所在域名（Domain），使用Cookie进行安全连接（Secure）。  
前两个参数是Cookie应用的必要条件，另外，还包括Cookie大小（Size，不同浏览器对Cookie个数及大小限制是有差异的）。

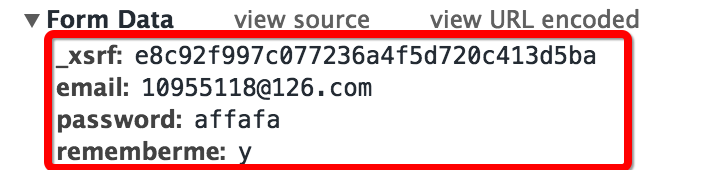
**2. 模拟登陆**

这次主要爬取的网站是知乎

爬取知乎就需要登陆的, 通过之前的python内建库, 可以很容易的实现表单提交

**现在就来看看如何通过Scrapy实现表单提交**

首先查看登陆时的表单结果, 依然像前面使用的技巧一样, 故意输错密码, 方面抓到登陆的网页头部和表单(我使用的Chrome自带的开发者工具中的Network功能)



*表单截图*

查看抓取到的表单可以发现有四个部分:

* 邮箱和密码就是个人登陆的邮箱和密码
* rememberme字段表示是否记住账号
* 第一个字段是\_xsrf,猜测是一种验证机制

现在只有\_xsrf不知道, 猜想这个验证字段肯定会实现在请求网页的时候发送过来, 那么我们查看当前网页的源码(鼠标右键然后查看网页源代码, 或者直接用快捷键)



*查询网页源码*

发现我们的猜测是正确的

那么现在就可以来写表单登陆功能了

def start\_requests(self):

return [Request("https://www.zhihu.com/login", callback = self.post\_login)] #重写了爬虫类的方法, 实现了自定义请求, 运行成功后会调用callback回调函数

#FormRequeset

def post\_login(self, response):

print 'Preparing login'

#下面这句话用于抓取请求网页后返回网页中的\_xsrf字段的文字, 用于成功提交表单

xsrf = Selector(response).xpath('//input[@name="\_xsrf"]/@value').extract()[0]

print xsrf

#FormRequeset.from\_response是Scrapy提供的一个函数, 用于post表单

#登陆成功后, 会调用after\_login回调函数

return [FormRequest.from\_response(response,

formdata = {

'\_xsrf': xsrf,

'email': '123456',

'password': '123456'

},

callback = self.after\_login

)]

**3. Cookie的保存**

为了能使用同一个状态持续的爬取网站, 就需要保存cookie, 使用cookie保存状态, Scrapy提供了cookie处理的中间件, 可以直接拿来使用

[CookiesMiddleware](http://doc.scrapy.org/en/0.24/topics/downloader-middleware.html?highlight=cookie#module-scrapy.contrib.downloadermiddleware.cookies)

这个cookie中间件保存追踪web服务器发出的cookie, 并将这个cookie在接来下的请求的时候进行发送

Scrapy官方的文档中给出了下面的代码范例 :

for i, url in enumerate(urls):

yield scrapy.Request("http://www.example.com", meta={'cookiejar': i},

callback=self.parse\_page)

def parse\_page(self, response):

# do some processing

return scrapy.Request("http://www.example.com/otherpage",

meta={'cookiejar': response.meta['cookiejar']},

callback=self.parse\_other\_page)

那么可以对我们的爬虫类中方法进行修改, 使其追踪cookie

#重写了爬虫类的方法, 实现了自定义请求, 运行成功后会调用callback回调函数

def start\_requests(self):

return [Request("https://www.zhihu.com/login", meta = {'cookiejar' : 1}, callback = self.post\_login)] #添加了meta

#FormRequeset出问题了

def post\_login(self, response):

print 'Preparing login'

#下面这句话用于抓取请求网页后返回网页中的\_xsrf字段的文字, 用于成功提交表单

xsrf = Selector(response).xpath('//input[@name="\_xsrf"]/@value').extract()[0]

print xsrf

#FormRequeset.from\_response是Scrapy提供的一个函数, 用于post表单

#登陆成功后, 会调用after\_login回调函数

return [FormRequest.from\_response(response, #"http://www.zhihu.com/login",

meta = {'cookiejar' : response.meta['cookiejar']}, #注意这里cookie的获取

headers = self.headers,

formdata = {

'\_xsrf': xsrf,

'email': '123456',

'password': '123456'

},

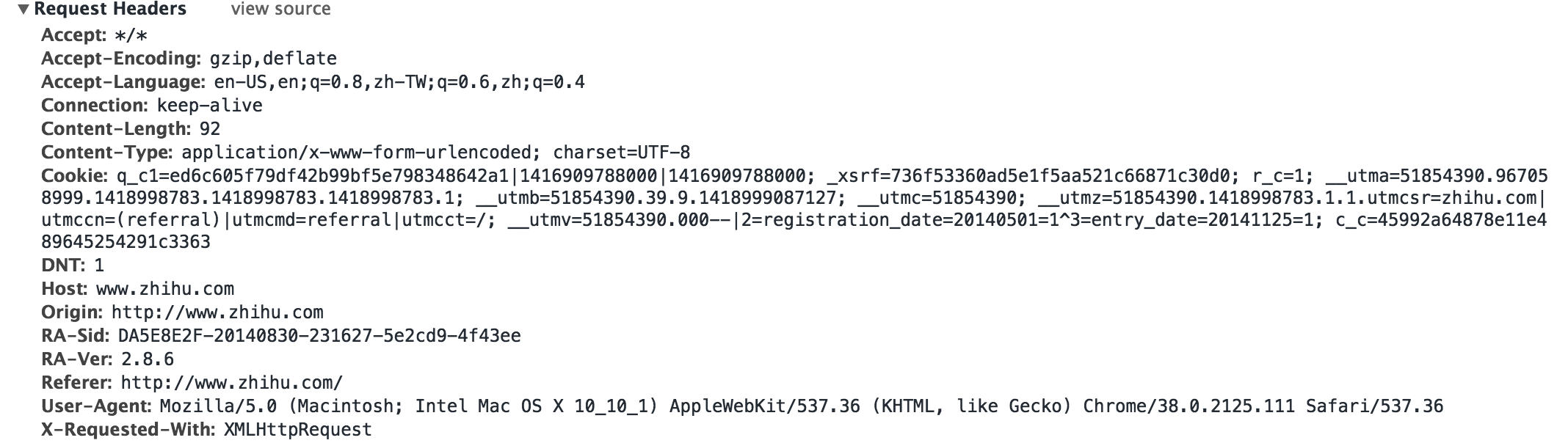
callback = self.after\_login,

dont\_filter = True

)]

**4. 伪装头部**

有时候登陆网站需要进行头部伪装, 比如增加防盗链的头部, 还有模拟服务器登陆, 这些都在前面的爬虫知识中提到过



*Headers*

为了保险, 我们可以在头部中填充更多的字段, 如下

headers = {

"Accept": "\*/\*",

"Accept-Encoding": "gzip,deflate",

"Accept-Language": "en-US,en;q=0.8,zh-TW;q=0.6,zh;q=0.4",

"Connection": "keep-alive",

"Content-Type":" application/x-www-form-urlencoded; charset=UTF-8",

"User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_10\_1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/38.0.2125.111 Safari/537.36",

"Referer": "http://www.zhihu.com/"

}

在scrapy中Request和FormRequest初始化的时候都有一个headers字段, 可以自定义头部, 这样我们可以添加headers字段

形成最终版的登陆函数

#!/usr/bin/env python

# -\*- coding:utf-8 -\*-

from scrapy.contrib.spiders import CrawlSpider, Rule

from scrapy.selector import Selector

from scrapy.contrib.linkextractors.sgml import SgmlLinkExtractor

from scrapy.http import Request, FormRequest

from zhihu.items import ZhihuItem

class ZhihuSipder(CrawlSpider) :

name = "zhihu"

allowed\_domains = ["www.zhihu.com"]

start\_urls = [

"http://www.zhihu.com"

]

rules = (

Rule(SgmlLinkExtractor(allow = ('/question/\d+#.\*?', )), callback = 'parse\_page', follow = True),

Rule(SgmlLinkExtractor(allow = ('/question/\d+', )), callback = 'parse\_page', follow = True),

)

headers = {

"Accept": "\*/\*",

"Accept-Encoding": "gzip,deflate",

"Accept-Language": "en-US,en;q=0.8,zh-TW;q=0.6,zh;q=0.4",

"Connection": "keep-alive",

"Content-Type":" application/x-www-form-urlencoded; charset=UTF-8",

"User-Agent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10\_10\_1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/38.0.2125.111 Safari/537.36",

"Referer": "http://www.zhihu.com/"

}

#重写了爬虫类的方法, 实现了自定义请求, 运行成功后会调用callback回调函数

def start\_requests(self):

return [Request("https://www.zhihu.com/login", meta = {'cookiejar' : 1}, callback = self.post\_login)]

#FormRequeset出问题了

def post\_login(self, response):

print 'Preparing login'

#下面这句话用于抓取请求网页后返回网页中的\_xsrf字段的文字, 用于成功提交表单

xsrf = Selector(response).xpath('//input[@name="\_xsrf"]/@value').extract()[0]

print xsrf

#FormRequeset.from\_response是Scrapy提供的一个函数, 用于post表单

#登陆成功后, 会调用after\_login回调函数

return [FormRequest.from\_response(response, #"http://www.zhihu.com/login",

meta = {'cookiejar' : response.meta['cookiejar']},

headers = self.headers, #注意此处的headers

formdata = {

'\_xsrf': xsrf,

'email': '1095511864@qq.com',

'password': '123456'

},

callback = self.after\_login,

dont\_filter = True

)]

def after\_login(self, response) :

for url in self.start\_urls :

yield self.make\_requests\_from\_url(url)

def parse\_page(self, response):

problem = Selector(response)

item = ZhihuItem()

item['url'] = response.url

item['name'] = problem.xpath('//span[@class="name"]/text()').extract()

print item['name']

item['title'] = problem.xpath('//h2[@class="zm-item-title zm-editable-content"]/text()').extract()

item['description'] = problem.xpath('//div[@class="zm-editable-content"]/text()').extract()

item['answer']= problem.xpath('//div[@class=" zm-editable-content clearfix"]/text()').extract()

return item

**5. Item类和抓取间隔**

from scrapy.item import Item, Field

class ZhihuItem(Item):

# define the fields for your item here like:

# name = scrapy.Field()

url = Field() #保存抓取问题的url

title = Field() #抓取问题的标题

description = Field() #抓取问题的描述

answer = Field() #抓取问题的答案

name = Field() #个人用户的名称

设置抓取间隔, 访问由于爬虫的过快抓取, 引发网站的发爬虫机制, 在setting.py中设置

BOT\_NAME = 'zhihu'

SPIDER\_MODULES = ['zhihu.spiders']

NEWSPIDER\_MODULE = 'zhihu.spiders'

DOWNLOAD\_DELAY = 0.25 #设置下载间隔为250ms

抓取结果(只是截取了其中很少一部分)

...

'url': 'http://www.zhihu.com/question/20688855/answer/16577390'}

2014-12-19 23:24:15+0800 [zhihu] DEBUG: Crawled (200) <GET http://www.zhihu.com/question/20688855/answer/15861368> (referer: http://www.zhihu.com/question/20688855/answer/19231794)

[]

2014-12-19 23:24:15+0800 [zhihu] DEBUG: Scraped from <200 http://www.zhihu.com/question/20688855/answer/15861368>

{'answer': [u'\u9009\u4f1a\u8ba1\u8fd9\u4e2a\u4e13\u4e1a\uff0c\u8003CPA\uff0c\u5165\u8d22\u52a1\u8fd9\u4e2a\u884c\u5f53\u3002\u8fd9\u4e00\u8def\u8d70\u4e0b\u6765\uff0c\u6211\u53ef\u4ee5\u5f88\u80af\u5b9a\u7684\u544a\u8bc9\u4f60\uff0c\u6211\u662f\u771f\u7684\u559c\u6b22\u8d22\u52a1\uff0c\u70ed\u7231\u8fd9\u4e2a\u884c\u4e1a\uff0c\u56e0\u6b64\u575a\u5b9a\u4e0d\u79fb\u5730\u5728\u8fd9\u4e2a\u884c\u4e1a\u4e2d\u8d70\u4e0b\u53bb\u3002',

u'\u4e0d\u8fc7\u4f60\u8bf4\u6709\u4eba\u4ece\u5c0f\u5c31\u559c\u6b22\u8d22\u52a1\u5417\uff1f\u6211\u89c9\u5f97\u51e0\u4e4e\u6ca1\u6709\u5427\u3002\u8d22\u52a1\u7684\u9b45\u529b\u5728\u4e8e\u4f60\u771f\u6b63\u61c2\u5f97\u5b83\u4e4b\u540e\u3002',

u'\u901a\u8fc7\u5b83\uff0c\u4f60\u53ef\u4ee5\u5b66\u4e60\u4efb\u4f55\u4e00\u79cd\u5546\u4e1a\u7684\u7ecf\u8425\u8fc7\u7a0b\uff0c\u4e86\u89e3\u5176\u7eb7\u7e41\u5916\u8868\u4e0b\u7684\u5b9e\u7269\u6d41\u3001\u73b0\u91d1\u6d41\uff0c\u751a\u81f3\u4f60\u53ef\u4ee5\u638c\u63e1\u5982\u4f55\u53bb\u7ecf\u8425\u8fd9\u79cd\u5546\u4e1a\u3002',

u'\u5982\u679c\u5bf9\u4f1a\u8ba1\u7684\u8ba4\u8bc6\u4ec5\u4ec5\u505c\u7559\u5728\u505a\u5206\u5f55\u8fd9\u4e2a\u5c42\u9762\uff0c\u5f53\u7136\u4f1a\u89c9\u5f97\u67af\u71e5\u65e0\u5473\u3002\u5f53\u4f60\u5bf9\u5b83\u7684\u8ba4\u8bc6\u8fdb\u5165\u5230\u6df1\u5c42\u6b21\u7684\u65f6\u5019\uff0c\u4f60\u81ea\u7136\u5c31\u4f1a\u559c\u6b22\u4e0a\u5b83\u4e86\u3002\n\n\n'],

'description': [u'\u672c\u4eba\u5b66\u4f1a\u8ba1\u6559\u80b2\u4e13\u4e1a\uff0c\u6df1\u611f\u5176\u67af\u71e5\u4e4f\u5473\u3002\n\u5f53\u521d\u662f\u51b2\u7740\u5e08\u8303\u4e13\u4e1a\u62a5\u7684\uff0c\u56e0\u4e3a\u68a6\u60f3\u662f\u6210\u4e3a\u4e00\u540d\u8001\u5e08\uff0c\u4f46\u662f\u611f\u89c9\u73b0\u5728\u666e\u901a\u521d\u9ad8\u4e2d\u8001\u5e08\u5df2\u7ecf\u8d8b\u4e8e\u9971\u548c\uff0c\u800c\u987a\u6bcd\u4eb2\u5927\u4eba\u7684\u610f\u9009\u4e86\u8fd9\u4e2a\u4e13\u4e1a\u3002\u6211\u559c\u6b22\u4e0a\u6559\u80b2\u5b66\u7684\u8bfe\uff0c\u5e76\u597d\u7814\u7a76\u5404\u79cd\u6559\u80b2\u5fc3\u7406\u5b66\u3002\u4f46\u4f1a\u8ba1\u8bfe\u4f3c\u4e4e\u662f\u4e3b\u6d41\u3001\u54ce\u3002\n\n\u4e00\u76f4\u4e0d\u559c\u6b22\u94b1\u4e0d\u94b1\u7684\u4e13\u4e1a\uff0c\u6240\u4ee5\u5f88\u597d\u5947\u5927\u5bb6\u9009\u4f1a\u8ba1\u4e13\u4e1a\u5230\u5e95\u662f\u51fa\u4e8e\u4ec0\u4e48\u76ee\u7684\u3002\n\n\u6bd4\u5982\u8bf4\u5b66\u4e2d\u6587\u7684\u4f1a\u8bf4\u4ece\u5c0f\u559c\u6b22\u770b\u4e66\uff0c\u4f1a\u6709\u4ece\u5c0f\u559c\u6b22\u4f1a\u8ba1\u501f\u554a\u8d37\u554a\u7684\u7684\u4eba\u5417\uff1f'],

'name': [],

'title': [u'\n\n', u'\n\n'],

'url': 'http://www.zhihu.com/question/20688855/answer/15861368'}

...

<https://github.com/Andrew-liu/scrapy_example/tree/master/zhihu>

<http://www.jianshu.com/p/b7f41df6202d>

### 扩展Python

**创建Python扩展的步骤**

**1. 创建应用程序代码**

#**include** <stdio.h>  
#**include** <stdlib.h>  
#**include** <string.h>  
  
#**define** BUFSIZE 10  
  
**int** fac(**int** n) {  
**if** (n < 2)  
**return** 1;  
**return** n \* fac(n - 1);  
}  
  
**char** \*reverse(**char** \*s) {  
**register** **char** t;  
**char** \*p = s;  
**char** \*q = (s + (strlen(s) - 1));  
**while** (p < q) {  
        t = \*p;  
        \*p++ = \*q;  
        \*q-- = t;  
    }  
**return** s;  
}  
  
**int** main() {  
**char** s[BUFSIZE];  
    printf("4! == %d\n", fac(4));  
    printf("8! == %d\n", fac(8));  
    printf("12! == %d\n", fac(12));  
    strcpy(s, "abcdef");  
    printf("reversing 'abcdef', we get '%s'\n", reverse(s));  
    strcpy(s, "madam");  
    printf("reversing 'madam', we get '%s'\n", reverse(s));  
**return** 0;  
}

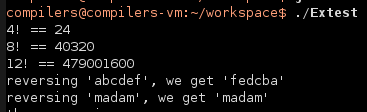
一般是需要写main()函数，用于单元测试

使用gcc进行编译

>gcc Extest.c -o Extest

执行

>./Extest



**2. 利用样板来包装代码**

整个扩展的实现都是围绕"包装"这个概念来进行的。你的设计要尽可能让你的实现语言与Python无缝结合。

接口的代码又被称为"样板"代码，它是你的代码与Python解释器之间进行交互所必不可少的部分：

我们的样板代码分为4步：

**a. 包含python的头文件**

需要找到python的头文件在哪，一般是在/usr/local/include/python2.x中

在上面的C代码中加入#include "Python.h"

**b. 为每个模块的每一个函数增加一个型如PyObject\* Module\_func()的包装函数**

包装函数的用处就是先把python的值传递给c，再把c中函数的计算结果转换成Python对象返回给python。

需要为所有想被Python环境访问到的函数都增加一个静态函数，返回类型为PyObject \*，函数名格式为

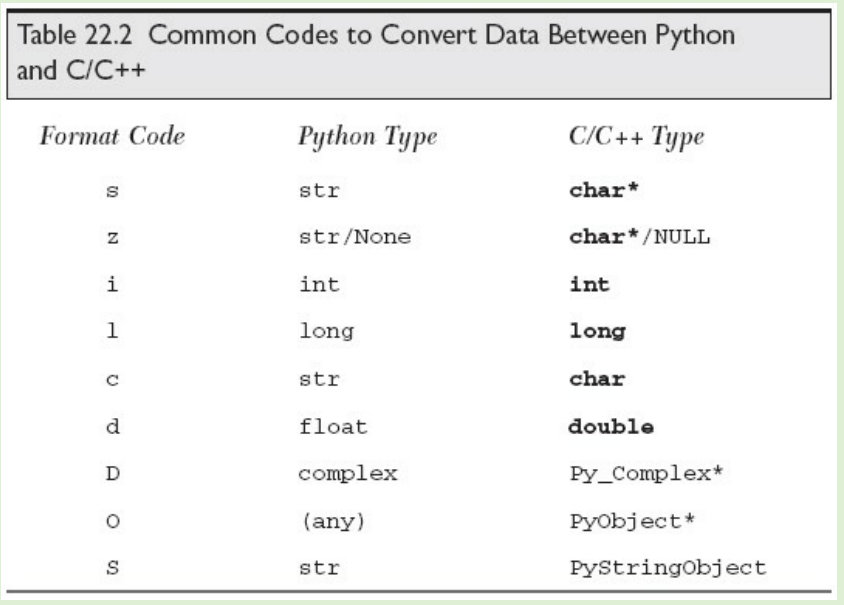
模块名\_函数名；

**static** PyObject \* Extest\_fac(PyObject \*self, PyObject \*args) {  
**int** res;//计算结果值  
**int** num;//参数  
    PyObject\* retval;//返回值  
  
    //i表示需要传递进来的参数类型为整型，如果是，就赋值给num，如果不是，返回NULL；  
    res = PyArg\_ParseTuple(args, "i", &num);   
**if** (!res) {  
        //包装函数返回NULL，就会在Python调用中产生一个TypeError的异常  
**return** NULL;  
    }  
    res = fac(num);  
    //需要把c中计算的结果转成python对象，i代表整数对象类型。  
    retval = (PyObject \*)Py\_BuildValue("i", res);  
**return** retval;  
}

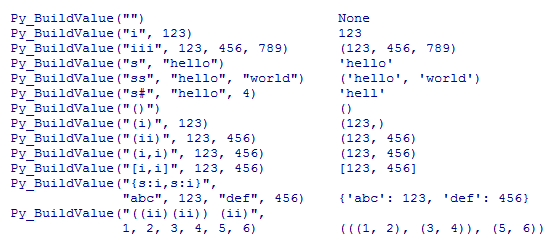
也可以写成更简短，可读性更强的形式：

**static** PyObject \* Extest\_fac(PyObject \*self, PyObject \*args) {  
**int** m;  
**if** (!(PyArg\_ParseTuple(args, "i", &num))) {  
**return** NULL;  
    }  
**return** (PyObject \*)Py\_BuildValue("i", fac(num));  
}

下面是python和c对应的类型转换参数表：



这里还有一个Py\_BuildValue的用法表：



reverse函数的包装也类似：

**static** PyObject \*  
Extest\_reverse(PyObject \*self, PyObject \*args) {  
**char** \*orignal;  
**if** (!(PyArg\_ParseTuple(args, "s", &orignal))) {  
**return** NULL;  
    }  
**return** (PyObject \*)Py\_BuildValue("s", reverse(orignal));  
}

也可以再改造成返回包含原始字串和反转字串的tuple的函数

**static** PyObject \*  
Extest\_doppel(PyObject \*self, PyObject \*args) {  
**char** \*orignal;  
**if** (!(PyArg\_ParseTuple(args, "s", &orignal))) {  
**return** NULL;  
    }  
    //ss,就可以返回两个字符串,应该reverse是在原字符串上进行操作，所以需要先strdup复制一下  
**return** (PyObject \*)Py\_BuildValue("ss", orignal, reverse(strdup(orignal)));  
}

上面的代码有什么问题呢？

和c语言相关的问题，比较常见的就是内存泄露。。。上面的例子中，Py\_BuildValue()函数生成

要返回Python对象的时候，会把转入的数据复制一份。上面的两个字符串都被复制出来。但是

我们申请了用于存放第二个字符串的内存，在退出的时候没有释放掉它。于是内存就泄露了。

正确的做法是：先生成返回的python对象，然后释放在包装函数中申请的内存。

**static** PyObject \*  
Extest\_doppel(PyObject \*self, PyObject \*args) {  
**char** \*orignal;  
**char** \*reversed;  
    PyObject \* retval;  
**if** (!(PyArg\_ParseTuple(args, "s", &orignal))) {  
**return** NULL;  
    }  
    retval = (PyObject \*)Py\_BuildValue("ss", orignal, reversed=reverse(strdup(orignal)));  
    free(reversed);  
**return** retval;  
}

**c. 为每个模块增加一个型如PyMethodDef ModuleMethods[]的数组**

我们已经创建了几个包装函数，需要在某个地方把它们列出来，以便python解释器能够导入并调用它们。

这个就是ModuleMethods[]数组所需要做的事情。

格式如下 ，每一个数组都包含一个函数的信息，最后一个数组放置两个NULL值，代表声明结束

**static** PyMethodDef   
ExtestMethods[] = {  
    {"fac", Extest\_fac, METH\_VARARGS},   
    {"doppel", Extest\_doppel, METH\_VARARGS},  
    {"reverse", Extest\_reverse, METH\_VARARGS},  
    {NULL, NULL},  
};

METH\_VARARGS代表参数以tuple的形式传入。如果我们需要使用PyArg\_ParseTupleAndKeywords()

函数来分析关键字参数的话，这个标志常量应该写成: METH\_VARARGS & METH\_KEYWORDS，进行逻辑与运算。

**d. 增加模块初始化函数void initMethod()**

最后的工作就是模块的初始化工作。这部分代码在模块被python导入时进行调用。

**void** initExtest() {  
    Py\_InitModule("Extest", ExtestMethods);  
}

最终代码如下：

#**include** <stdio.h>  
#**include** <stdlib.h>  
#**include** <string.h>  
#**include** "Python.h"  
  
#**define** BUFSIZE 10  
  
**int** fac(**int** n) {  
**if** (n < 2)  
**return** 1;  
**return** n \* fac(n - 1);  
}  
  
**char** \*reverse(**char** \*s) {  
**register** **char** t;  
**char** \*p = s;  
**char** \*q = (s + (strlen(s) - 1));  
**while** (p < q) {  
        t = \*p;  
       \*p++ = \*q;  
       \*q-- = t;  
    }  
**return** s;  
}  
  
**static** PyObject \*  
Extest\_fac(PyObject \*self, PyObject \*args) {  
**int** res;  
**int** num;  
    PyObject\* retval;  
  
    res = PyArg\_ParseTuple(args, "i", &num);  
**if** (!res) {  
**return** NULL;  
    }  
    res = fac(num);  
    retval = (PyObject \*)Py\_BuildValue("i", res);  
**return** retval;  
}  
  
**static** PyObject \*  
Extest\_reverse(PyObject \*self, PyObject \*args) {  
**char** \*orignal;  
**if** (!(PyArg\_ParseTuple(args, "s", &orignal))) {  
**return** NULL;  
    }  
**return** (PyObject \*)Py\_BuildValue("s", reverse(orignal));  
}  
  
**static** PyObject \*  
Extest\_doppel(PyObject \*self, PyObject \*args) {  
**char** \*orignal;  
**char** \*resv;  
    PyObject \*retval;  
**if** (!(PyArg\_ParseTuple(args, "s", &orignal))) {  
**return** NULL;  
    }  
    retval = (PyObject \*)Py\_BuildValue("ss", orignal, resv=reverse(strdup(orignal)));  
    free(resv);  
**return** retval;  
}  
  
**static** PyMethodDef   
ExtestMethods[] = {  
    {"fac", Extest\_fac, METH\_VARARGS},  
    {"doppel", Extest\_doppel, METH\_VARARGS},  
    {"reverse", Extest\_reverse, METH\_VARARGS},  
    {NULL, NULL},  
};  
  
**void** initExtest() {  
    Py\_InitModule("Extest", ExtestMethods);  
}  
  
**int** main() {  
**char** s[BUFSIZE];  
    printf("4! == %d\n", fac(4));  
    printf("8! == %d\n", fac(8));  
    printf("12! == %d\n", fac(12));  
    strcpy(s, "abcdef");  
    printf("reversing 'abcdef', we get '%s'\n", reverse(s));  
    strcpy(s, "madam");  
    printf("reversing 'madam', we get '%s'\n", reverse(s));  
    test();  
**return** 0;  
}

**3. 编译与测试**

为了让你的新python扩展能够被创建，你需要把它们与python库放在一起编译。python中的distutils包被

用来编译，安装和分发这些模块，扩展和包。步骤如下:

**a. 创建setup.py**

我们在安装python第三方包的时候，很多情况下会用到python setup.py install这个命令，

下面我们来了解一下setup.py文件的内容。

编译的最主要的内容由setup函数完成，你需要为每一个扩展创建一个Extension实例，在这里我们只有一个

扩展，所以只需要创建一个实例。

Extension('Extest', sources=['Extest.c'])，第一个参数是扩展的名字，如果模块是包的一部分，还需要加"."；

第二个参数是源代码文件列表

setup('Extest', ext\_modules=[...])，第一个参数表示要编译哪个东西，第二个参数列出要编译的Extension对象。

#!/usr/bin/env python  
**from** distutils.core **import** setup, Extension  
    MOD = 'Extest'  
    setup(name=MOD, ext\_modules=[Extension(MOD, sources=['Extest.c'])])

setup函数还有很多选项可以设置。详情可见官网。

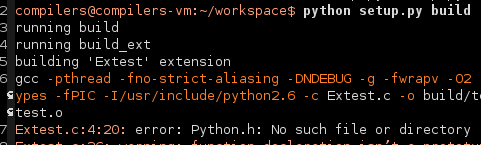
<http://docs.python.org/distutils/setupscript.html>

**b. 通过运行setup.py来编译和连接你的代码**

在shell中运行命令

>python setup.py build

当你报错如：无法找到Python.h文件



那么说明你没有安装python-dev包，需要去官网下载源码包重装自己编译安装一下python。

Python.h文件一般会出现在/usr/include/Python2.X文件夹中，我这里反正是没有的。。。

只有重新编译一个python...

我现在linux系统上的python版本是2.6.6，我下载一个相同版本的源码，也可以下载更高版本。

<http://www.python.org/download/releases/2.6.6/>

解压源码包

> tar xzf Python-2.6.6.tgz

> cd Python-2.6.6.tgz

编译安装Python

> ./configure --prefix=/usr/local/python2.6

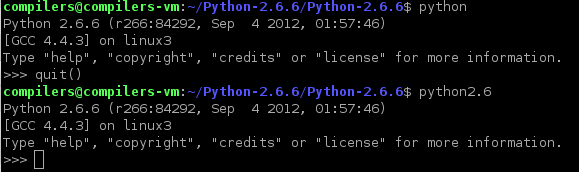
> make

> sudo make install

创建一个新编译python的链接

> sudo ln -sf /usr/local/python2.6/bin/python2.6 /usr/bin/python2.6

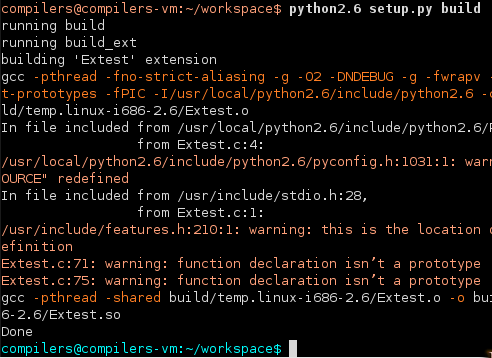
测试一下，可用



使用这种方法可以在Linux上运行不同版本的python.

Python.h文件也在/usr/local/python2.6/include/python2.6路径下找到。

重新运行编译



编译成功后，你的扩展就会被创建在bulid/lib.\*目录下。你会看到一个.so文件，这是linux下的

动态库文件：

http://images.cnblogs.com/cnblogs_com/btchenguang/201209/201209051401008247.png

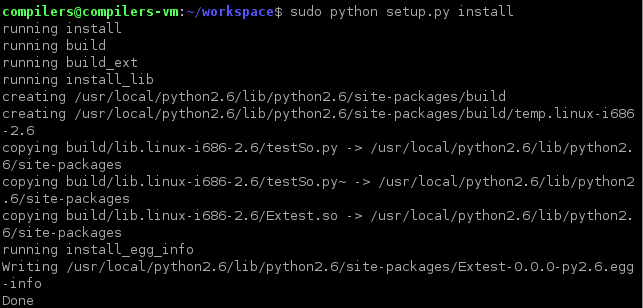
**c. 进行调试**

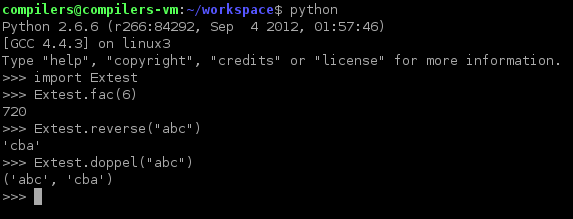
你可以直接用python代码调用进行测试：

#!/usr/bin/python  
**from** ctypes **import** \*  
**import** **os**   
#需要使用绝对路径  
extest = cdll.LoadLibrary(**os**.getcwd() + '/Extest.so')   
**print** extest.fac(4)

也可以在当前目录下执行命令，安装到你的python路径下

> python setup.py install



安装成功的话，直接导入测试：  


最后需要注意一点的是，原来的c文件中有一个main函数，因为一个系统中只能有一个main

函数，所以为了不起冲突，可以把main函数改成test函数，再用Extest\_test()包装函数处理一下，

再加入ExtestMethods数组，这样就可以调用这个测试函数了。

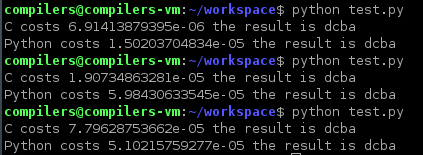
**static** PyObject \*  
Extest\_test(PyObject \*self, PyObject \*args) {  
    test();  
    #返回空的话，就使用下面这一句   
    **return** (PyObject \*)Py\_BuildValue("");  
}

**简单性能比较**

测试代码

**import** Extest  
**import** time  
  
start = time.time()  
a = Extest.reverse("abcd")  
timeC = time.time() - start  
**print** 'C costs', timeC, 'the result is', a  
  
start = time.time()  
b = **list**("abcd")  
b.reverse()  
b = ''.join(b)  
timePython = time.time()-start  
**print** 'Python costs', timePython, 'the result is', b

运行结果



可以看出，python也不是绝对比C慢嘛，还要看情况。

摘自：

<http://www.cnblogs.com/btchenguang/archive/2012/09/04/2670849.html>

### Python WCGI

**概述**

WSGI接口包含两方面：server/gateway 及 application/framework。  
server调用由application提供的可调用对象。  
另外在server和application之间还可能有一种称作middleware的中间件。  
可调用对象是指：函数、方法、类或者带有callable方法的实例。  
**关于application**

函数、方法、类及带有callable方法的实例等可调用对象都可以作为the application object。  
WSGI协议要求：  
the application object接受两个参数且可以被多次调用

这两个参数分别为：  
1.CGI式的字典；  
2.回调函数：application用来向server传递http状态码/消息/http头

另外协议要求可调用对象必须将响应体封装成一个可迭代的strings返回。

# the application object. 可以使用其他名字,

# 但是在使用mod\_wsgi 时必须为 "application"

def application( environ, start\_response):

# 函数接受两个参数:

# environ ：包含有CGI 式环境变量的字典，由server负责提供内容

# start\_response：由server提供的回调函数，其作用是将状态码和响应头返回给server

# 构造响应体，以可迭代字符串形式封装

response\_body = 'The request method was %s' % environ['REQUEST\_METHOD']

# HTTP 响应码及消息

status = '200 OK'

# 提供给客户端的响应头.

# 封装成list of tuple pairs 的形式:

# 格式要求：[(Header name, Header value)].

response\_headers = [('Content-Type', 'text/plain'),

('Content-Length', str(len(response\_body)))]

# 将响应码/消息及响应头通过传入的start\_reponse回调函数返回给server

start\_response(status, response\_headers)

# 响应体作为返回值返回

# 注意这里被封装到了list中.

return [response\_body]

**关于server**

从概述中可以知道，WSGI server必须要调用application，同时，从application的协议要求可知：  
1. WSGI server必须向application提供环境参数，因此，自身也必须能够获取环境参数。  
2. WSGI server接收application的返回值作为响应体。  
最简单的WSGI server为Python自带的wsgiref.simple\_server  
示例如下：

from wsgiref.simple\_server import make\_server

srv = make\_server('localhost', 8080, hello\_world)

srv.serve\_forever()

**关于middleware**

middleware的概念没有appllication和server那么容易理解。  
假设一个符合application标准的可调用对象，它接受可调用对象作为参数，返回一个可调用对象的对象。  
那么对于server来说，它是一个符合标准的可调用对象，因此是application。  
而对于application来说，它可以调用application，因此是server。  
这样的可调用对象称为middleware。

middleware的概念非常接近decorator。

以一个路由的例子示例：

import re

# 这是一个标准的application object

def index(environ, start\_response):

start\_response('200 OK', [('Content-Type', 'text/html')])

return ['index page']

# 这是一个标准的application object

def hello(environ, start\_response):

start\_response('200 OK', [('Content-Type', 'text/html')])

return ['hello page']

# 这是一个标准的application object

def not\_found(environ, start\_response):

start\_response('404 NOT FOUND', [('Content-Type', 'text/plain')])

return ['Not Found Page']

# map urls to functions

urls = [

(r'^$', index),

(r'hello/?$', hello)

]

# 这是一个middleware

# 根据不同的route返回不同的application object

def application(environ, start\_response):

path = environ.get('PATH\_INFO', '').lstrip('/')

for regex, callback in urls:

match = re.search(regex, path)

if match is not None:

### Python CGI、

## CGI是什么?

* 通用网关接口或CGI，是一组定义信息如何在Web服务器和自定义脚本之间交换的标准。
* CGI规范目前保持是由NCSA 和 NCSA 维护和定义如下。
* 通用网关接口或CGI，是外部网关方案，如HTTP服务器的信息服务器的接口标准。
* 目前的版本是CGI/1.1，而CGI/1.2目前正在定制中。

## 网页浏览

要了解CGI的概念，让我们看看当点击一个超链接，浏览某一个网页或URL发生什么情况。

* 浏览器触发HTTP Web服务器和网址也就是文件名的请求。
* Web服务器将解析URL，并寻找，如果找到该文件，然后将其发送回浏览器中的文件名，否则将表示请求了错误文件的错误消息。
* Web浏览器需要从Web服务器和显示器接收到的文件信息或错误信息的响应。

然而，也可以设置在HTTP服务器，以便每当在某个目录中的文件被请求文件不被发送回;相反，它被执行的程序，无论该程序的输出被发送回浏览器显示。此函数被称为公共网关接口CGI或与程序称为CGI脚本。这些CGI程序可以是一个Python脚本，Perl脚本，Shell脚本，C或C++程序等等。

## CGI架构图



## Web服务器支持与配置

在进行CGI编程之前，请确保Web服务器支持CGI，它被配置为处理CGI程序。所有对由HTTP服务器执行的CGI程序保存在一个预先配置的目录。此目录被称为CGI目录，并按照惯例被命名为/var/www/cgi-bin目录。按照惯例，CGI文件具有扩展名为.cgi，但文件扩展名可以为Python语言脚本 .py。

默认情况下，Linux服务器被配置为只运行在在/var/www/cgi-bin目录中的脚本。如果想指定要运行的CGI脚本任何其他目录，在httpd.conf文件中注释以下几行：

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

AllowOverride None

Options ExecCGI

Order allow,deny

Allow from all

</Directory>

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

Options All

</Directory>

在这里，假设Web服务器能成功运行cgi程序，也可以运行Perl或Shell等任何其它的CGI程序

## 第一个CGI程序

下面是一个简单的链接，链接到一个名为hello.py CGI脚本。此文件被保存在/var/www/cgi-bin目录，它有以下内容。运行CGI程序之前，请确保有使用chmod 755 hello.py UNIX命令，使文件可执行文件的更改模式。

#!/usr/bin/python

print "Content-type:text/html

"

print '<html>'

print '<head>'

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

print '</head>'

print '<body>'

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

print '</body>'

print '</html>'

如果单击hello.py，那么这将产生以下的输出：

|  |
| --- |
| Hello Word! This is my first CGI program |

hello.py脚本是一个简单的Python脚本，它在标准输出（即屏幕）输出显示文件。还有一个重要的额外功能，第一行要打印Content-type:text/html 。这行内容被发送回浏览器和指定内容类型可以在浏览器画面上显示出来。

现在，必须了解CGI的基本概念，可以使用Python编写许多复杂的CGI程序。脚本可以与任何其他外部系统交互还向交换信息，如RDBMS（数据库管理系统）。

## HTTP报头

第一行Content-type:text/html 被发送到浏览器，了解内容的HTTP标头的一部分。所有HTTP报头将在下面的表格：

HTTP Field Name: Field Content

For Example

Content-type: text/html

还有其他一些重要的HTTP头，会在CGI程序经常使用。

|  |  |
| --- | --- |
| **报头** | **描述** |
| Content-type: | 返回MIME字符串，定义文件的格式。例如Content-type:text/html |
| Expires: Date | 日期的信息变为无效。这应该被使用的浏览器，以决定当一个页面需要刷新。有效日期字符串应该是格式1998年1月1日12:00:00 GMT。 |
| Location: URL | 应该返回替代请求URL的URL。可以使用此字段来重定向请求到任何文件。 |
| Last-modified: Date | 资源的最后修改的日期。 |
| Content-length: N | 数据的长度，以字节为单位被返回。浏览器使用这个值来报告预计下载时间的文件。 |
| Set-Cookie: String | 通过设置该字符串传递cookie |

## CGI环境变量

所有CGI程序将可使用下面的环境变量。在编写任何CGI程序，这些变量中发挥重要作用。

|  |  |
| --- | --- |
| **Variable Name** | **描述** |
| CONTENT\_TYPE | 内容的数据类型。当客户端发送内容附加到服务器使用。例如，文件上传等。 |
| CONTENT\_LENGTH | 查询信息的长度。它仅适用于POST请求。 |
| HTTP\_COOKIE | 返回键和值对的形式设置Cookie。 |
| HTTP\_USER\_AGENT | 用户代理请求头字段包含有关用户代理发起请求信息。网络浏览器的名称。 |
| PATH\_INFO | TCGI脚本的路径。 |
| QUERY\_STRING | 被发送GET方法请求URL编码的信息。 |
| REMOTE\_ADDR | 远程主机发出请求的IP地址。这可以是用于记录或用于认证的目的是有用的。 |
| REMOTE\_HOST | 发出请求的主机的完全合格的名称。如果该信息不可用，则REMOTE\_ADDR可用于获得IP地址。 |
| REQUEST\_METHOD | 该方法用于制造要求。最常用的方法是GET和POST。 |
| SCRIPT\_FILENAME | CGI脚本的完整路径。 |
| SCRIPT\_NAME | CGI脚本的名称。 |
| SERVER\_NAME | 服务器的主机名或IP地址 |
| SERVER\_SOFTWARE | 软件服务器运行的名称和版本。 |

这里是小CGI程序，列出所有的CGI变量，代码内容详细如下：

#!/usr/bin/python

import os

print "Content-type: text/html

";

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

for param in os.environ.keys():

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

## GET / POST 方法

当需要从浏览器中传递一些信息到Web服务器中的CGI程序。最常见的是浏览器会使用两种方法二传这个信息给Web服务器。这两个方法分别是GET方法和POST方法。

## 使用GET方法传递信息：

GET方法将附加在页面请求所编码的用户信息。页面和编码信息是由 ？字符分开如下：

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

GET方法是默认的方法，从浏览器的信息传递到Web服务器和它所产生出现在浏览器的位置，如果是很长的字符串，或如果有密码或其他敏感信息传递给服务器，切勿使用GET方法。 GET方法有大小限制：仅1024个字符可以在请求字符串被发送。 GET方法将使用QUERY\_STRING头信息，并会通过QUERY\_STRING环境变量的CGI程序访问。

可以通过简单的串联键和值对传递以及任何URL信息，或者可以使用HTML<form>标记使用GET方法来传递信息。

## 简单URL示例：GET方法

下面是一个简单的URL，它会通过两个值使用GET方法传递给hello\_get.py程序。

[/cgi-bin/hello\_get.py?first\_name=ZARA&last\_name=ALI](http://www.yiibai.com/cgi-bin/hello_get.py?first_name=ZARA&last_name=ALI)

下面是hello\_get.py脚本来处理输入Web浏览器显示。要使用CGI模块，这使得它非常容易访问传递信息：

#!/usr/bin/python

# Import modules for CGI handling

import cgi, cgitb

# Create instance of FieldStorage

form = cgi.FieldStorage()

# Get data from fields

first\_name = form.getvalue('first\_name')

last\_name = form.getvalue('last\_name')

print "Content-type:text/html

"

print "<html>"

print "<head>"

print "<title>Hello - Second CGI Program</title>"

print "</head>"

print "<body>"

print "<h2>Hello %s %s</h2>" % (first\_name, last\_name)

print "</body>"

print "</html>"

这将产生以下结果：

|  |
| --- |
| Hello ZARA ALI |

## 简单的表单示例：GET方法

下面是一个简单的例子，通过使用HTML表单和提交按钮两个值。我们将使用相同的CGI脚本hello\_get.py来处理这个输入。

<form action="/cgi-bin/hello\_get.py" method="get">

First Name: <input type="text" name="first\_name"> <br />

Last Name: <input type="text" name="last\_name" />

<input type="submit" value="Submit" />

</form>

下面是上述形式的实际输出，输入您名字和姓氏，然后单击提交按钮来查看结果。

窗体顶端

First Name:   
Last Name:  

窗体底端

## 使用POST方法传递信息：

将信息传递给CGI程序的一般比较可靠的方法是POST方法。这个包中完全相同的方式作为GET方法的信息，但是，代替发送它作为后一个文本字符串？在URL中，它把它作为一个单独的消息。此消息进入在标准输入表单的CGI脚本。

下面是处理GET和POST方法都使用相同的hello\_get.py脚本。

#!/usr/bin/python

# Import modules for CGI handling

import cgi, cgitb

# Create instance of FieldStorage

form = cgi.FieldStorage()

# Get data from fields

first\_name = form.getvalue('first\_name')

last\_name = form.getvalue('last\_name')

print "Content-type:text/html

"

print "<html>"

print "<head>"

print "<title>Hello - Second CGI Program</title>"

print "</head>"

print "<body>"

print "<h2>Hello %s %s</h2>" % (first\_name, last\_name)

print "</body>"

print "</html>"

让我们再上面同样的例子其中通过使用HTML表单两个值和提交按钮。我们将使用相同的CGI脚本hello\_get.py来处理这个输入框。

<form action="/cgi-bin/hello\_get.py" method="post">

First Name: <input type="text" name="first\_name"><br />

Last Name: <input type="text" name="last\_name" />

<input type="submit" value="Submit" />

</form>

这里是在上述形式的实际输出。输入名字和姓氏，然后单击提交按钮来查看结果。

窗体顶端

First Name:   
Last Name:  

窗体底端

## 传递复选框数据给CGI程序

复选框用于当需要多个选项被选中。

下面是两个复选框形式例如HTML代码：

<form action="/cgi-bin/checkbox.cgi" method="POST" target="\_blank">

<input type="checkbox" name="maths" value="on" /> Maths

<input type="checkbox" name="physics" value="on" /> Physics

<input type="submit" value="Select Subject" />

</form>

这段代码的结果如下表：

窗体顶端

 Maths  Physics 

窗体底端

下面是checkbox.cgi脚本来处理Web浏览器中给出的复选框按钮。

#!/usr/bin/python

# Import modules for CGI handling

import cgi, cgitb

# Create instance of FieldStorage

form = cgi.FieldStorage()

# Get data from fields

if form.getvalue('maths'):

math\_flag = "ON"

else:

math\_flag = "OFF"

if form.getvalue('physics'):

physics\_flag = "ON"

else:

physics\_flag = "OFF"

print "Content-type:text/html

"

print "<html>"

print "<head>"

print "<title>Checkbox - Third CGI Program</title>"

print "</head>"

print "<body>"

print "<h2> CheckBox Maths is : %s</h2>" % math\_flag

print "<h2> CheckBox Physics is : %s</h2>" % physics\_flag

print "</body>"

print "</html>"

## 传递单选按钮数据给CGI程序

单选按钮，使用只需要一个选项被选择。

下面是两个单选按钮的形式例如HTML代码：

<form action="/cgi-bin/radiobutton.py" method="post" target="\_blank">

<input type="radio" name="subject" value="maths" /> Maths

<input type="radio" name="subject" value="physics" /> Physics

<input type="submit" value="Select Subject" />

</form>

这段代码的结果如下表：

窗体顶端

 Maths  Physics 

窗体底端

下面是radiobutton.py脚本来处理Web浏览器给出的单选按钮：

#!/usr/bin/python

# Import modules for CGI handling

import cgi, cgitb

# Create instance of FieldStorage

form = cgi.FieldStorage()

# Get data from fields

if form.getvalue('subject'):

subject = form.getvalue('subject')

else:

subject = "Not set"

print "Content-type:text/html

"

print "<html>"

print "<head>"

print "<title>Radio - Fourth CGI Program</title>"

print "</head>"

print "<body>"

print "<h2> Selected Subject is %s</h2>" % subject

print "</body>"

print "</html>"

## Passing Text Area Data to CGI Program

TEXTAREA element is used when multiline text has to be passed to the CGI Program.

Here is example HTML code for a form with a TEXTAREA box:

|  |
| --- |
| <form action="/cgi-bin/textarea.py" method="post" target="\_blank">  <textarea name="textcontent" cols="40" rows="4">  Type your text here...  </textarea>  <input type="submit" value="Submit" />  </form> |

The result of this code is the following form:

窗体顶端

窗体底端

Below is textarea.cgi script to handle input given by web browser:

#!/usr/bin/python

# Import modules for CGI handling

import cgi, cgitb

# Create instance of FieldStorage

form = cgi.FieldStorage()

# Get data from fields

if form.getvalue('textcontent'):

text\_content = form.getvalue('textcontent')

else:

text\_content = "Not entered"

print "Content-type:text/html

"

print "<html>"

print "<head>";

print "<title>Text Area - Fifth CGI Program</title>"

print "</head>"

print "<body>"

print "<h2> Entered Text Content is %s</h2>" % text\_content

print "</body>"

## Passing Drop Down Box Data to CGI Program

Drop Down Box is used when we have many options available but only one or two will be selected.

Here is example HTML code for a form with one drop down box:

<form action="/cgi-bin/dropdown.py" method="post" target="\_blank">

<select name="dropdown">

<option value="Maths" selected>Maths</option>

<option value="Physics">Physics</option>

</select>

<input type="submit" value="Submit"/>

</form>

The result of this code is the following form:

窗体顶端

窗体底端

Below is dropdown.py script to handle input given by web browser.

#!/usr/bin/python

# Import modules for CGI handling

import cgi, cgitb

# Create instance of FieldStorage

form = cgi.FieldStorage()

# Get data from fields

if form.getvalue('dropdown'):

subject = form.getvalue('dropdown')

else:

subject = "Not entered"

print "Content-type:text/html

"

print "<html>"

print "<head>"

print "<title>Dropdown Box - Sixth CGI Program</title>"

print "</head>"

print "<body>"

print "<h2> Selected Subject is %s</h2>" % subject

print "</body>"

print "</html>"

## Using Cookies in CGI

HTTP protocol is a stateless protocol. But for a commercial website, it is required to maintain session information among different pages. For example, one user registration ends after completing many pages. But how to maintain user's session information across all the web pages.

In many situations, using cookies is the most efficient method of remembering and tracking preferences, purchases, commissions, and other information required for better visitor experience or site statistics.

## How It Works?

Your server sends some data to the visitor's browser in the form of a cookie. The browser may accept the cookie. If it does, it is stored as a plain text record on the visitor's hard drive. Now, when the visitor arrives at another page on your site, the cookie is available for retrieval. Once retrieved, your server knows/remembers what was stored.

Cookies are a plain text data record of 5 variable-length fields:

* **Expires :** The date the cookie will expire. If this is blank, the cookie will expire when the visitor quits the browser.
* **Domain :** The domain name of your site.
* **Path :** The path to the directory or web page that sets the cookie. This may be blank if you want to retrieve the cookie from any directory or page.
* **Secure :** If this field contains the word "secure", then the cookie may only be retrieved with a secure server. If this field is blank, no such restriction exists.
* **Name=Value :** Cookies are set and retrieved in the form of key and value pairs.

## Setting up Cookies

It is very easy to send cookies to browser. These cookies will be sent along with HTTP Header before to Content-type field. Assuming you want to set UserID and Password as cookies. So cookies setting will be done as follows:

#!/usr/bin/python

print "Set-Cookie:UserID=XYZ;

"

print "Set-Cookie:Password=XYZ123;

"

print "Set-Cookie:Expires=Tuesday, 31-Dec-2007 23:12:40 GMT";

"

print "Set-Cookie:Domain=www.yiibai.com;

"

print "Set-Cookie:Path=/perl;

"

print "Content-type:text/html

"

...........Rest of the HTML Content....

From this example, you must have understood how to set cookies. We use **Set-Cookie** HTTP header to set cookies.

Here, it is optional to set cookies attributes like Expires, Domain and Path. It is notable that cookies are set before sending magic line **"Content-type:text/html**.

## Retrieving Cookies

It is very easy to retrieve all the set cookies. Cookies are stored in CGI environment variable HTTP\_COOKIE and they will have following form:

key1=value1;key2=value2;key3=value3....

下面是如何获取cookies的例子。

#!/usr/bin/python

# Import modules for CGI handling

from os import environ

import cgi, cgitb

if environ.has\_key('HTTP\_COOKIE'):

for cookie in map(strip, split(environ['HTTP\_COOKIE'], ';')):

(key, value ) = split(cookie, '=');

if key == "UserID":

user\_id = value

if key == "Password":

password = value

print "User ID = %s" % user\_id

print "Password = %s" % password

这将产生以下结果由上面的脚本设置cookie：

User ID = XYZ

Password = XYZ123

## 文件上传例子：

上传文件，HTML表单的enctype属性必须设置为multipart/form-data。文件类型的input标签将创建一个“浏览”按钮。

<html>

<body>

<form enctype="multipart/form-data"

action="save\_file.py" method="post">

<p>File: <input type="file" name="filename" /></p>

<p><input type="submit" value="Upload" /></p>

</form>

</body>

</html>

这段代码的结果如下表：

窗体顶端

File:



窗体底端

上面的例子中编写完成后，可以在你的服务器尝试上面的代码。

下面是脚本save\_file.py处理文件上传：

#!/usr/bin/python

import cgi, os

import cgitb; cgitb.enable()

form = cgi.FieldStorage()

# Get filename here.

fileitem = form['filename']

# Test if the file was uploaded

if fileitem.filename:

# strip leading path from file name to avoid

# directory traversal attacks

fn = os.path.basename(fileitem.filename)

open('/tmp/' + fn, 'wb').write(fileitem.file.read())

message = 'The file "' + fn + '" was uploaded successfully'

else:

message = 'No file was uploaded'

print """

Content-Type: text/html

<html>

<body>

<p>%s</p>

</body>

</html>

""" % (message,)

如果运行是上Unix/Linux上的脚本，那么就必须照顾替换文件分隔符，如下所示，否则在Windows机器上面open()语句应该正常工作。

fn = os.path.basename(fileitem.filename.replace("", "/" ))

## 如何弹出“文件下载”对话框？

有时，期望要提供选项当用户将点击一个链接，它将一个“文件下载”对话框弹出给用户，而不是显示的实际内容。这是很容易的，并通过HTTP头来实现。这个HTTP头跟从上一节提到的报头是一样。

例如，如果想从一个给定链路进行文件名的文件下载，它的语法如下：

#!/usr/bin/python

# HTTP Header

print "**Content-Type:**application/octet-stream; name="FileName"

";

print "**Content-Disposition:** attachment; filename="FileName"

";

# Actual File Content will go hear.

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

str = fo.read();

print str

# Close opend file

fo.close()