# 理解Python的With语句

转载 2014年04月09日 11:44:50

* 121778

## [With语句是什么？](http://python.42qu.com/11155501" \l "h20" \t "_blank)

Python’s with statement provides a very convenient way of dealing with the situation where you have to do a setup and teardown to make something happen. A very good example for this is the situation where you want to gain a handler to a file, read data from the file and the close the file handler.

有一些任务，可能事先需要设置，事后做清理工作。对于这种场景，Python的with语句提供了一种非常方便的处理方式。一个很好的例子是文件处理，你需要获取一个文件句柄，从文件中读取数据，然后关闭文件句柄。

Without the with statement, one would write something along the lines of:

如果不用with语句，代码如下：

|  |  |
| --- | --- |
| 1  2  3 | file = open("/tmp/foo.txt")  data = file.read()  file.close() |

There are two annoying things here. First, you end up forgetting to close the file handler. The second is how to handle exceptions that may occur once the file handler has been obtained. One could write something like this to get around this:

这里有两个问题。一是可能忘记关闭文件句柄；二是文件读取数据发生异常，没有进行任何处理。下面是处理异常的加强版本：

|  |  |
| --- | --- |
| 1  2  3  4  5 | file = open("/tmp/foo.txt")  try:      data = file.read()  finally:      file.close() |

While this works well, it is unnecessarily verbose. This is where with is useful. The good thing about with apart from the better syntax is that it is very good handling exceptions. The above code would look like this, when using with:

虽然这段代码运行良好，但是太冗长了。这时候就是with一展身手的时候了。除了有更优雅的语法，with还可以很好的处理上下文环境产生的异常。下面是with版本的代码：

|  |  |
| --- | --- |
| 1  2 | with open("/tmp/foo.txt") as file:      data = file.read() |

## [with如何工作？](http://python.42qu.com/11155501" \l "h21" \t "_blank)

while this might look like magic, the way Python handles with is more clever than magic. The basic idea is that the statement after with has to evaluate an object that responds to an \_\_enter\_\_() as well as an \_\_exit\_\_() function.

这看起来充满魔法，但不仅仅是魔法，Python对with的处理还很聪明。基本思想是with所求值的对象必须有一个\_\_enter\_\_()方法，一个\_\_exit\_\_()方法。

After the statement that follows with is evaluated, the \_\_enter\_\_() function on the resulting object is called. The value returned by this function is assigned to the variable following as. After every statement in the block is evaluated, the \_\_exit\_\_() function is called.

紧跟with后面的语句被求值后，返回对象的\_\_enter\_\_()方法被调用，这个方法的返回值将被赋值给as后面的变量。**当with后面的代码块全部被执行完之后，将调用前面返回对象的\_\_exit\_\_()方法。**

This can be demonstrated with the following example:

下面例子可以具体说明with如何工作：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | #!/usr/bin/env python  # with\_example01.py      class Sample:      def \_\_enter\_\_(self):          print "In \_\_enter\_\_()"          return "Foo"        def \_\_exit\_\_(self, type, value, trace):          print "In \_\_exit\_\_()"      def get\_sample():      return Sample()      with get\_sample() as sample:      print "sample:", sample |

When executed, this will result in:

运行代码，输出如下

|  |  |
| --- | --- |
| 1  2  3  4 | bash-3.2$ ./with\_example01.py  In \_\_enter\_\_()  sample: Foo  In \_\_exit\_\_() |

As you can see,

The \_\_enter\_\_() function is executed

The value returned by it - in this case "Foo" is assigned to sample

The body of the block is executed, thereby printing the value of sample ie. "Foo"

The \_\_exit\_\_() function is called.

What makes with really powerful is the fact that it can handle exceptions. You would have noticed that the \_\_exit\_\_() function for Sample takes three arguments - val, type and trace. These are useful in exception handling. Let’s see how this works by modifying the above example.

正如你看到的，

1. \_\_enter\_\_()方法被执行

2. \_\_enter\_\_()方法返回的值 - 这个例子中是"Foo"，赋值给变量'sample'

3. 执行代码块，打印变量"sample"的值为 "Foo"

4. \_\_exit\_\_()方法被调用

with真正强大之处是它可以处理异常。可能你已经注意到Sample类的\_\_exit\_\_方法有三个参数- val, type 和 trace。 这些参数在异常处理中相当有用。我们来改一下代码，看看具体如何工作的。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | #!/usr/bin/env python  # with\_example02.py      class Sample:      def \_\_enter\_\_(self):          return self        def \_\_exit\_\_(self, type, value, trace):          print "type:", type          print "value:", value          print "trace:", trace        def do\_something(self):          bar = 1/0          return bar + 10    with Sample() as sample:      sample.do\_something() |

Notice how in this example, instead of get\_sample(), with takes Sample(). It does not matter, as long as the statement that follows with evaluates to an object that has an \_\_enter\_\_() and \_\_exit\_\_() functions. In this case, Sample()’s \_\_enter\_\_() returns the newly created instance of Sample and that is what gets passed to sample.

这个例子中，with后面的get\_sample()变成了Sample()。这没有任何关系，只要紧跟with后面的语句所返回的对象有\_\_enter\_\_()和\_\_exit\_\_()方法即可。此例中，Sample()的\_\_enter\_\_()方法返回新创建的Sample对象，并赋值给变量sample。

When executed:

代码执行后：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | bash-3.2$ ./with\_example02.py  type: <type 'exceptions.ZeroDivisionError'>  value: integer division or modulo by zero  trace: <traceback object at 0x1004a8128>  Traceback (most recent call last):    File "./with\_example02.py", line 19, in <module>      sample.do\_something()    File "./with\_example02.py", line 15, in do\_something      bar = 1/0  ZeroDivisionError: integer division or modulo by zero |

Essentially, if there are exceptions being thrown from anywhere inside the block, the \_\_exit\_\_() function for the object is called. As you can see, the type, value and the stack trace associated with the exception thrown is passed to this function. In this case, you can see that there was a ZeroDivisionError exception being thrown. People implementing libraries can write code that clean up resources, close files etc. in their \_\_exit\_\_() functions.

实际上，在with后面的代码块抛出任何异常时，\_\_exit\_\_()方法被执行。正如例子所示，异常抛出时，与之关联的type，value和stack trace传给\_\_exit\_\_()方法，因此抛出的ZeroDivisionError异常被打印出来了。开发库时，清理资源，关闭文件等等操作，都可以放在\_\_exit\_\_方法当中。

Thus, Python’s with is a nifty construct that makes code a little less verbose and makes cleaning up during exceptions a bit easier.

因此，Python的with语句是提供一个有效的机制，让代码更简练，同时在异常产生时，清理工作更简单。

I have put the code examples given here on [Github](https://github.com/sdqali/python_dojo/tree/master/with).

示例代码可以在[Github](https://github.com/sdqali/python_dojo/tree/master/with)上面找到。

**python tkinter教程-事件绑定**

一个Tkinter主要跑在mainloop进程里。Events可能来自多个地方，比如按键，鼠标，或是系统事件。   
Tkinter提供了丰富的方法来处理这些事件。对于每一个控件Widget，你都可以为其绑定方法function。

widget.bind(event,handler)

* 1

如果相应的event发生了，就会调用handler处理事件。举个例子：   
捕获鼠标点击事件：

from Tkinter import \*

root = Tk()

def callback(event):

print "clicked at", event.x, event.y

frame = Frame(root, width=100, height=100)

frame.bind("<Button-1>", callback)

frame.pack()

root.mainloop()

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10

在这里，我们使用frame的bind方法将一个callback方法绑定到一个事件，也就是点击鼠标左键，每点击一次，控制台打印出当前点击的坐标x，y。   
键盘事件被发送到当前拥有焦点的widget，你可以用focus\_set方法来设置widget们的焦点：   
捕获键盘事件：

from Tkinter import \*

root = Tk()

def key(event):

print "pressed", repr(event.char)

def callback(event):

print "clicked at", event.x, event.y

frame = Frame(root, width=100, height=100)

frame.bind("<Key>", key)

frame.bind("<Button-1>", callback)

frame.pack()

root.mainloop()

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8
* 9
* 10
* 11
* 12
* 13
* 14
* 15
* 16

运行这个程序，你会发现，只有在当前窗口获取焦点的情况下，你按键盘键它才会捕获到并打印。

事件

事件用字符串定义，有一个特殊的语法规则：

<modifier-type-detail>

* 1

type字段是最重要的，它指出了事件的种类，可以指定为Button，Key或者Enter，Configure等等。modifier和detail字段可以提供一些附加信息，在大多数情况下可以不指定。还有很多方法可以简化事件字符串，比如：为了匹配一个键盘键，你可以省略尖角括号，直接用 键 即可。除非它是空格 ， 或本身就是尖括号。   
让我们来看看最常用的事件格式：   
事件格式：

<Button-1>

* 1

一个鼠标点击事件。1代表左键，2代表中键，3代表右键。当你在一个widget上点击鼠标按键，tkinter会自动捕获并触发event，注意，当按键被抬起时才会执行handler。鼠标的位置（相对于widge）x，y会被发往event对象传入handler。你也可以这样:，<1>，它们是等价的。我比较喜欢这种方式。

<B1-Motion>

* 1

鼠标拖动事件。1代表按下左键拖动，2代表中键，3代表右键。同样的，鼠标的x，y会以event对象方式被送往handler。

<ButtonRelease-1>

* 1

鼠标按下之后释放

<Double-Button-1>

* 1

双击鼠标

<Enter>

* 1

注意，这里是鼠标指针进入到widget里，并不是代表按下键盘上的Enter键。

<Leave>

* 1

和上面的进入对应，鼠标离开widget。

<FocusIn> <FocusOut>

* 1

<Return> <Cancel> <BackSpace> <Tab> <Shift\_L> <Control\_L>

<Alt\_L> <Home> <Left> <Up> <Right> <Down> <Delete> <F1> <F2>

* 1
* 2

这些按键都和键盘上的一一对应。

<Key>

* 1

随便一个按键，键值会以char的格式放入event对象。

a b c ... 1 2 ...

* 1

对应键盘上的按键

<Configure>

* 1

这个关键了，如果widget的大小改变了，或者是位置，新的大小（width和height）会打包到event发往handler。

事件对象

事件对象是独立的python实例，有很多属性。   
对象属性：

widget 产生event的实例，不是名字，所有对象拥有

x，y 鼠标位置，单位：像素

x\_root，y\_root 鼠标相对于屏幕左上角的位置，像素

char 仅键盘事件，string

num 按钮num，仅鼠标事件

width，height widget新大小

type 事件类型

* 1
* 2
* 3
* 4
* 5
* 6
* 7

实例绑定和类绑定

上面我们用的绑定方法是绑定到一个实例对象上，这就意味着，如果新建一个实例，它是没有绑定事件的。   
实际上，tkinter允许你将事件绑定到类上，甚至是绑定到程序本身。你可以创建四个层面的绑定：

* 绑定到widget instance 使用bind方法
* 绑定到widget的toplevel windows，顶层窗口，也是用bind
* 绑定到widget class，使用bind\_class

比如，你可以使用bind\_all来创建一个F1键的绑定，这样你可以在任何地方打开帮助。

origin 是默认的远程版本库名称  
你可以在 .git/config 之中进行修改  
  
事实上 git push origin master 的意思是 git push origin master:master （将本地的 master 分支推送至远端的 master 分支，如果没有就[新建](https://www.baidu.com/s?wd=%E6%96%B0%E5%BB%BA&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d9mhcdPhnsuWbkPHTLuADY0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPj64nWbknWfdn1RsnW6dnHTz)一个）

git fetch:相当于是从远程获取最新版本到本地。  
git pull:命令的作用是,取回远程主机某个分支的更新,再与本地的指定分支合并。  
git fetch获取最新版本后，会覆盖本地计算机的版本，旧版本的信息会被删除。  
git pull从远程主机获得更新信息后，与本地信息合并，旧版本的信息不会被删除。  
这是两个计算机网络程序命令：  
git：在这里指“服务器”。  
fetch：在这里是"获取; 取数据; 读取"的意思。  
pull：在这里是“下拉”的意思

\*nix命令都是如此，一个横线开头的接参数的缩写，两个横线开头的接参数的全称git -h git --helporgit remote [-v | --verbose]对比win下参数用/dir /? format /Q

三种引号都表示字符串  
  
单引号表示的字符串里可包含双引号，当然不能包含单引号  
双引号表示的字符串里可以包含单引号，字符串都只能有一行  
三个引号能包含多行字符串，同时常常出现在函数的声明的下一行，来注释函数的功能，与众不同的地方在于，这个注释作为函数的一个默认属性，可以通过 函数名.\_\_doc\_\_ 来访问

[python单引号和双引号的区别](http://www.cnblogs.com/hellochenchen/p/5516501.html)

今天在网上爬虫的时候，很奇怪的发现python的字符串既可以用双引号又可以用单引号，于是就上网百度了一下原因。

原来由于字符串中有可能既有双引号又有单引号，例如：

字符串：demo'1'。

这时候就可以：str　　= "demo'1'"；这样看更直观，当然也可以使用转义字符\'代替单引号：str　　= 'demo\'1\''。

字符串：demo"1"。

这时候就可以：str　　= 'demo"1"'。

python中如何从字符串中提取数字？

1. [使用正则表达式](https://www.baidu.com/s?wd=%E4%BD%BF%E7%94%A8%E6%AD%A3%E5%88%99%E8%A1%A8%E8%BE%BE%E5%BC%8F&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dbuW-Buhw-rAn1n1nkuj0Y0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3En1c3rH0dPHD1PW0sPW0sPW0d)，用法如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | ## 总结  ## ^ 匹配字符串的开始。  ## $ 匹配字符串的结尾。  ## \b 匹配一个单词的边界。  ## \d 匹配任意数字。  ## \D 匹配任意非数字字符。  ## x? 匹配一个可选的 x 字符 (换言之，它匹配 1 次或者 0 次 x 字符)。  ## x\* 匹配0次或者多次 x 字符。  ## x+ 匹配1次或者多次 x 字符。  ## x{n,m} 匹配 x 字符，至少 n 次，至多 m 次。  ## (a|b|c) 要么匹配 a，要么匹配 b，要么匹配 c。  ## (x) 一般情况下表示一个记忆组 (remembered group)。你可以利用 re.search 函数返回对象的 groups() 函数获取它的值。  ## 正则表达式中的点号通常意味着 “匹配任意单字符” |

1. 解题思路：

2.1 既然是提取数字，那么数字的形式一般是：整数，小数，整数加小数；

2.2 所以一般是形如：----.-----；

2.3 根据上述正则表达式的含义，可写出如下的表达式："\d+\.?\d\*"；

2.4 \d+匹配1次或者多次数字，注意这里不要写成\*，因为即便是小数，[小数点](https://www.baidu.com/s?wd=%E5%B0%8F%E6%95%B0%E7%82%B9&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dbuW-Buhw-rAn1n1nkuj0Y0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3En1c3rH0dPHD1PW0sPW0sPW0d)之前也得有一个数字；\.?这个是匹配[小数点](https://www.baidu.com/s?wd=%E5%B0%8F%E6%95%B0%E7%82%B9&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dbuW-Buhw-rAn1n1nkuj0Y0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3En1c3rH0dPHD1PW0sPW0sPW0d)的，可能有，也可能没有；\d\*这个是匹配[小数点](https://www.baidu.com/s?wd=%E5%B0%8F%E6%95%B0%E7%82%B9&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dbuW-Buhw-rAn1n1nkuj0Y0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3En1c3rH0dPHD1PW0sPW0sPW0d)之后的数字的，所以是0个或者多个；

1. 代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | # -\*- coding: cp936 -\*-  import re    string="A1.45，b5，6.45，8.82"  print re.findall(r"\d+\.?\d\*",string)    # ['1.45', '5', '6.45', '8.82'] |

Python中字符串前面加上 r 表示原生字符串，  
与大多数编程语言相同，正则表达式里使用"\"作为[转义字符](https://www.baidu.com/s?wd=%E8%BD%AC%E4%B9%89%E5%AD%97%E7%AC%A6&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1d-PANWuyPbnAfkPWf4Phfd0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3En10dPHbdPHckPHmzrjDdnHcY)，这就可能造成反斜杠困扰。假如你需要匹配文本中的字符"\"，那么使用编程语言表示的正则表达式里将需要4个反斜杠"\\\\"：前两个和后两个分别用于在编程语言里转义成反斜杠，转换成两个反斜杠后再在正则表达式里转义成一个反斜杠。Python里的原生字符串很好地解决了这个问题，这个例子中的正则表达式可以使用r"\\"表示。同样，匹配一个数字的"\\d"可以写成r"\d"。有了原生字符串，你再也不用担心是不是漏写了反斜杠，写出来的表达式也更直观。

# python 正则表达式 groups和group有什么区别 5

为什么   
m = re.match("([abc])+", "abc")  
print m.groups()  
返回的只有('c',) 而不是 abc  
而 print m.group() 就能返回整个的字符串  
他们到底有什么区别呢 网上的解释我看的不是太明白

最佳答案

group和groups是两个不同的函数。一般，m.group(N) 返回第N组括号匹配的字符。而m.group() == m.group(0) == 所有匹配的字符，与括号无关，这个是[API](https://www.baidu.com/s?wd=API&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1dbuWK9myu-nhnvrADkrjcY0ZwV5Hcvrjm3rH6sPfKWUMw85HfYnjn4nH6sgvPsT6KdThsqpZwYTjCEQLGCpyw9Uz4Bmy-bIi4WUvYETgN-TLwGUv3EPjbLPHf4nWcY)规定的。m.groups() 返回所有括号匹配的字符，以tuple格式。m.groups() == (m.group(0), m.group(1), ...)

正则表达式中，group（）用来提取分组截获的字符串，（）用来分组。

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"。

如果不引入括号，整个个表达式作为一个组，是group(0)

对于题目中的例子：m = re.match("([abc])+", "abc")+号在括号外面。括号最多匹配到一个字符，要么是a， 要么是c，这个python引擎匹配的是末尾的c。而m.group() == m.group(0) 这个返回的是整个匹配的字符串"abc".

***总的来说就是unicode是python解释器的内码，所有代码文件在导入并执行时，python解释器会先将字符串使用你指定的编码形式解码成unicode，然后再进行各种操作。所以不管是对字符串的操作，还是正则表达式，还是读写文件等等最好都通过unicode来进行。***

1.python3.x默认的编码是utf-8..

2.unicode问题请参考:<https://www.zhihu.com/question/23374078>

3.py3.x编码问题引入了byte和unicode概念,对字符串类型的区别更加清晰明确.

5.WINDOWS系统默认编码是GBK.常用的LINUX默认编码是UNICODE.

\*\*1.Python2中默认的字符编码是ASCII码。 2.Python2中字符串有str和unicode两种类型。str有各种编码的区别，unicode是没有编码的标准形式。 3.Python2中可以直接查看到unicode的字节串。\*\*

\*\*\*decode()方法将其他编码字符转化为Unicode编码字符。 encode()方法将Unicode编码字符转化为其他编码字符。\*

1.Python 3的源码.py文件 的默认编码方式为UTF-8，所以在Python3中你可以不用在py脚本中写coding声明，并且系统传递给python的字符不再受系统默认编码的影响，统一为unicode编码。

2.将字符串和字节序列做了区别，字符串str是字符串标准形式与2.x中unicode类似，bytes类似2.x中的str有各种编码区别。bytes通过解码转化成str,str通过编码转化成bytes。

***在最新的Python 3版本中，字符串是以Unicode编码的，由于Python的字符串类型是str，在内存中以Unicode表示，一个字符对应若干个字节。如果要在网络上传输，或者保存到磁盘上，就需要把str变为以字节为单位的bytes。***

在操作字符串时，我们经常遇到str和bytes的互相转换。为了避免乱码问题，应当始终坚持使用UTF-8编码对str和bytes进行转换。

Python源代码也是一个文本文件，所以，当你的源代码中包含中文的时候，在保存源代码时，就需要务必指定保存为UTF-8编码。当Python解释器读取源代码时，为了让它按UTF-8编码读取，我们通常在文件开头写上这两行

#!/usr/bin/env python3# -\*- coding: utf-8 -\*-

第二行注释是为了告诉Python解释器，按照UTF-8编码读取源代码，否则，你在源代码中写的中文输出可能会有乱码。

**在将unicode存储到文本的过程中，还有一种存储方式，不需要将unicode转换为实际的文本存储字符集，而是将unicode的内存编码值进行存储，读取文件的时候再反向转换回来，是采用：unicode-escape的转换方式。**

unicode到unicode-escape转换方法：unicodestr.encode(‘unicode-escape’)，示例：

|  |  |
| --- | --- |
| 1  2 | >>> u'中文测试'.encode('unicode-escape')  '\\u4e2d\\u6587\\u6d4b\\u8bd5' |

unicode-escape到unicode的转换方法：unicodeescapestr.decode(‘unicode-escape’)，示例：

|  |  |
| --- | --- |
| 1  2 | >>> '\\u4e2d\\u6587\\u6d4b\\u8bd5'.decode('unicode-escape')  u'\u4e2d\u6587\u6d4b\u8bd5' |

从英文意思上看，encode和decode分别指编码和解码。在python中，Unicode类型是作为编码的基础类型，即：

decode encode

str ---------> str(Unicode) ---------> str

>>> u = '中文' # 指定字符串类型对象u

>>> str1 = u.encode('gb2312') # 以gb2312编码对u进行编码，获得bytes类型对象

>>> print(str1)

b'\xd6\xd0\xce\xc4'

>>> str2 = u.encode('gbk') # 以gbk编码对u进行编码，获得bytes类型对象

>>> print(str2)

b'\xd6\xd0\xce\xc4'

>>> str3 = u.encode('utf-8') # 以utf-8编码对u进行编码，获得bytes类型对象

>>> print(str3)

b'\xe4\xb8\xad\xe6\x96\x87'

>>> u1 = str1.decode('gb2312') # 以gb2312编码对字符串str进行解码，获得字符串类型对象

>>> print('u1')

'中文'

>>> u2 = str1.decode('utf-8') # 报错，因为str1是gb2312编码的

UnicodeDecodeError: 'utf-8' codec can't decode byte 0xd6 in position 0: invalid continuation byt

# [python下 help()使用方法](http://www.cnblogs.com/laifu/p/3633034.html)

查看python所有的modules：help("modules")

单看python所有的modules中包含指定字符串的modules： help("modules yourstr")

查看python中常见的topics： help("topics")

查看python标准库中的module：import os.path + help("os.path")

查看python内置的类型：help("list")

查看python类型的成员方法：help("str.find")

查看python内置函数：help("open")

**字符串和编码**

1. Python3 字符串是以Unicode编码
2. 字符的表示转换函数   
   * ord()
   * chr()
3. str变为bytes方法   
   * ’中文’.encode(‘utf-8’)
   * ‘abc’.encode(‘ascii’)
4. bytes变为str方法   
   * b’ABC’.decode(‘ascii’)
   * b’\xe4\xb8\xad\xe6\x96\x87’.decode(‘utf-8’)
5. 注意！中文不能转为ascii编码
6. len()函数   
   * 计算str包含多少个字符
   * len(‘abc’)
   * len(‘中文’)
7. 保存源代码时，通常要在文件开头加上两行：   
   #!/usr/bin/env python3   
   # -\*- coding: utf-8 -\*-
8. 格式化   
   * 与C语言一致
   * ‘hello,%s’ % ‘world’
   * ‘hi,%s,you have $%d’ % (‘lily’,100)
   * 如果不确定数据类型是什么，%s永远起作用

*求一个汉字的unicode码？*

*hex(ord('中'))*

# [Python调用（运行）外部程序](http://www.cnblogs.com/IPYQ/p/5672922.html)

在Python中可以方便地使用os模块运行其他的脚本或者程序，这样就可以在脚本中直接使用其他脚本，或者程序提供的功能，而不必再次编写实现该功能的代码。为了更好地控制运行的进程，可以使用win32process模块中的函数。如果想进一步控制进程，则可以使用ctype模块，直接调用kernel32.dll中的函数。  
  
1 使用os.system函数运行其他程序  
2 使用ShellExecute函数运行其他程序  
3 使用CreateProcess函数运行其他程序  
4 使用ctypes调用kernel32.dll中的函数

**1 使用os.system函数运行其他程序**  
os模块中的system()函数可以方便地运行其他程序或者脚本。其函数原型如下所示。  
os.system(command)  
其参数含义如下所示。  
command 要执行的命令，相当于在Windows的cmd窗口中输入的命令。如果要向程序或者脚本传递参数，可以使用空格分隔程序及多个参数。  
以下实例实现通过os.system()函数打开系统的记事本程序。

>>> import os

# 使用os.system()函数打开记事本程序

>>> os.system('notepad')

0 # 关闭记事本后的返回值

# 向记事本传递参数，打开python.txt文件

>>> os.system('notepad python.txt')

**2 使用ShellExecute函数运行其他程序**  
除了使用os模块中的os.system()函数以外，还可以使用win32api模块中的ShellExecute()函数。其函数如下所示。  
ShellExecute(hwnd, op , file , params , dir , bShow )  
其参数含义如下所示。  
hwnd：父窗口的句柄，如果没有父窗口，则为0。  
op：要进行的操作，为“open”、“print”或者为空。  
file：要运行的程序，或者打开的脚本。  
params：要向程序传递的参数，如果打开的为文件，则为空。  
dir：程序初始化的目录。  
bShow：是否显示窗口。  
以下实例使用ShellExecute函数运行其他程序。

>>> import win32api

# 打开记事本程序，在后台运行，即显示记事本程序的窗口

>>> win32api.ShellExecute(0, 'open', 'notepad.exe', '','',0)

# 打开记事本程序，在前台运行

>>> win32api.ShellExecute(0, 'open', 'notepad.exe', '','',1)

# 向记事本传递参数，打开python.txt

>>> win32api.ShellExecute(0, 'open', 'notepad.exe', 'python.txt','',1)

# 在默认浏览器中打开http://www.python.org网站

>>> win32api.ShellExecute(0, 'open', 'http://www.python.org', '','',1)

# 在默认的媒体播放器中播放E:\song.wma

>>> win32api.ShellExecute(0, 'open', 'E:\\song.wma', '','',1)

# 运行位于E:\book\code目录中的MessageBox.py脚本

>>> win32api.ShellExecute(0, 'open', 'E:\\book\\code\\MessageBox.py', '','',1)

可以看出，使用ShellExecute函数，就相当于在资源管理器中双击文件图标一样，系统会打开相应的应用程序执行操作。

**3 使用CreateProcess函数运行其他程序**  
为了便于控制通过脚本运行的程序，可以使用win32process模块中的CreateProcess()函数。其函数原型如下所示。  
CreateProcess(appName, commandLine , processAttributes , threadAttributes , bInheritHandles ,dwCreationFlags , newEnvironment , currentDirectory , startupinfo )  
其参数含义如下。  
appName：可执行的文件名。  
commandLine：命令行参数。  
processAttributes：进程安全属性，如果为None，则为默认的安全属性。  
threadAttributes：线程安全属性，如果为None，则为默认的安全属性。  
bInheritHandles：继承标志。  
dwCreationFlags：创建标志。  
newEnvironment：创建进程的环境变量。  
currentDirectory：进程的当前目录。  
startupinfo ：创建进程的属性。  
以下实例使用win32process.CreateProcess函数运行记事本程序。

>>> import win32process

>>> win32process.CreateProcess('c:\\windows\\notepad.exe', '',

None , None , 0 ,win32process. CREATE\_NO\_WINDOW , None , None ,

win32process.STARTUPINFO())

(<?XML:NAMESPACE PREFIX = PYHANDLE />, , 280, 3076)

# 函数返回进程句柄、线程句柄、进程ID，以及线程ID

有了已创建进程的句柄就可以使用win32process.TerminateProcess函数结束进程，或者使用win32event.WaitForSingleObject等待创建的线程结束。其函数原型分别如下。  
TerminateProcess(handle, exitCode)  
WaitForSingleObject(handle, milliseconds )  
对于TerminateProcess参数含义分别如下。  
handle：要操作的进程句柄。  
exitCode：进程退出代码。  
对于WaitForSingleObject参数含义分别如下。  
handle：要操作的进程句柄。  
milliseconds：等待的时间，如果为1，则一直等待。  
以下实例实现创建进程后并对其进行操作。

>>> import win32process

# 打开记事本程序，获得其句柄

>>> handle = win32process.CreateProcess('c:\\windows\\notepad.exe',

'', None , None , 0 ,win32process. CREATE\_NO\_WINDOW , None , None ,

win32process.STARTUPINFO())

# 使用TerminateProcess函数终止记事本程序

>>> win32process.TerminateProcess(handle[0],0)

# 导入win32event模块

>>> import win32event

# 创建进程获得句柄

>>> handle = win32process.CreateProcess('c:\\windows\\notepad.exe',

'', None , None , 0 ,win32process. CREATE\_NO\_WINDOW , None , None ,

win32process.STARTUPINFO())

# 等待进程结束

>>> win32event.WaitForSingleObject(handle[0], -1)

0 # 进程结束的返回值

**4 使用ctypes调用kernel32.dll中的函数**  
使用ctypes模块可以使Python调用位于动态链接库中的函数。在Python 2.5版中已经包含了ctypes模块。如果使用其他版本的Python，可以到http://python.net/crew/theller/ctypes网站下载安装。ctypes适用于Python 2.3版本及以上。  
  
1．ctypes简介  
ctypes为Python提供了调用动态链接库中函数的功能。使用ctypes可以方便地调用由C语言编写的动态链接库，并向其传递参数。ctypes定义了C语言中的基本数据类型，并且可以实现C语言中的结构体和联合体。ctypes可以工作在Windows、Windows CE、Mac OS X、Linux、Solaris、FreeBSD、OpenBSD等平台上，基本上实现了跨平台。  
以下的实例使用ctypes实现了在Windows下直接调用user32.dll中的MessageBoxA函数。运行后如图10-6所示。

>>> from ctypes import \*

>>> user32 = windll.LoadLibrary('user32.dll') # 加载动态链接库

>>> user32.MessageBoxA(0, 'Ctypes is cool!', 'Ctypes', 0)

# 调用MessageBoxA函数.

# 重写、覆盖、重载、多态几个概念的区别分析

2009年02月01日 21:04:00

阅读数：116326

override->重写(=覆盖)、overload->重载、polymorphism -> 多态

override是重写（覆盖）了一个方法，以实现不同的功能。一般是用于子类在继承父类时，重写（重新实现）父类中的方法。  
重写（覆盖）的规则：  
   1、重写方法的参数列表必须完全与被重写的方法的相同,否则不能称其为重写而是重载.  
   2、重写方法的访问修饰符一定要大于被重写方法的访问修饰符（public>protected>default>private）。  
   3、重写的方法的返回值必须和被重写的方法的返回一致；  
   4、重写的方法所抛出的异常必须和被重写方法的所抛出的异常一致，或者是其子类；  
   5、被重写的方法不能为private，否则在其子类中只是新定义了一个方法，并没有对其进行重写。

   6、静态方法不能被重写为非静态的方法（会编译出错）。

overload是重载，一般是用于在一个类内实现若干重载的方法，这些方法的名称相同而参数形式不同。  
重载的规则：  
   1、在使用重载时只能通过相同的方法名、不同的参数形式实现。不同的参数类型可以是不同的参数类型，不同的参数个数，不同的参数顺序（参数类型必须不一样）；  
   2、不能通过访问权限、返回类型、抛出的异常进行重载；  
   3、方法的异常类型和数目不会对重载造成影响；

多态的概念比较复杂，有多种意义的多态，一个有趣但不严谨的说法是：继承是子类使用父类的方法，而多态则是父类使用子类的方法。  
一般，我们使用多态是为了避免在父类里大量重载引起代码臃肿且难于维护。

举个例子：  
public class Shape   
{  
   public static void main(String[] args){  
     Triangle tri = new Triangle();  
     System.out.println("Triangle is a type of shape? " + tri.isShape());// 继承

     Shape shape = new Triangle();  
     System.out.println("My shape has " + shape.getSides() + " sides."); // 多态

     Rectangle Rec = new Rectangle();  
     Shape shape2 = Rec;  
     System.out.println("My shape has " + shape2.getSides(Rec) + " sides."); //重载  
   }  
   public boolean isShape(){  
     return true;  
   }  
   public int getSides(){  
     return 0 ;  
   }  
   public int getSides(Triangle tri){ //重载  
     return 3 ;  
   }  
   public int getSides(Rectangle rec){ //重载  
    return 4 ;  
   }  
}

class Triangle extends Shape   
{  
   public int getSides() { //重写,实现多态  
     return 3;  
   }  
}  
class Rectangle extends Shape   
{  
   public int getSides(int i) { //重载  
    return i;  
   }  
}

注意Triangle类的方法是重写，而Rectangle类的方法是重载。对两者比较，可以发现多态对重载的优点：  
如果用重载，则在父类里要对应每一个子类都重载一个取得边数的方法；  
如果用多态，则父类只提供取得边数的接口，至于取得哪个形状的边数，怎样取得，在子类里各自实现(重写)。

**按照对多态“同样的操作，得到不同的结果”的观点，重载(overload)和覆盖(override)都是实现多态的手段。其中，重载是静态多态实现，在程序编译时实现；覆盖是动态多态实现，在程序运行时实现。  
 重载：重载函数是定义在全局或某个类中。其要求是同名但参数不同的函数。而父类与子类中实现出现同名但参数不同的函数，不管是否有virtual，其基类函数将被隐藏，注意与重载区别，其调用时需在父**[**类的成员函数**](https://www.baidu.com/s?wd=%E7%B1%BB%E7%9A%84%E6%88%90%E5%91%98%E5%87%BD%E6%95%B0&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)**前加T::进行区分（其中T为父类名）。  
 覆盖：函数必须是virtual，这是首要条件。其要求在不同域中（如子父类中），且要求函数名，参数，返回值都相同。**

# 多态是基于对抽象方法的覆盖来实现的，用统一的对外接口来完成不同的功能。重载也是用统一的对外接口来完成不同的功能。那么两者有什么区别呢？ 重载，是指允许存在多个同名方法，而这些方法的参数不同。重载的实现是：编译器根据方法不同的参数表，对同名方法的名称做修饰。对于编译器而言，这些同名方法就成了不同的方法。它们的调用地址在编译期就绑定了。 多态：是指子类重新定义父类的虚方法（virtual,abstract）。当子类重新定义了父类的虚方法后，父类根据赋给它的不同的子类，动态调用属于子类的该方法，这样的方法调用在编译期间是无法确定的。 不难看出，两者的区别在于编译器何时去寻找所要调用的具体方法，对于重载而言，在方法调用之前，编译器就已经确定了所要调用的方法，这称为“早绑定”或“静态绑定”；而对于多态，只有等到方法调用的那一刻，编译器才会确定所要调用的具体方法，这称为“晚绑定”或“动态绑定”。

# 重写、覆盖、重载、多态几个概念的区别分析

2009年02月01日 21:04:00

阅读数：116778

override->重写(=覆盖)、overload->重载、polymorphism -> 多态

override是重写（覆盖）了一个方法，以实现不同的功能。一般是用于子类在继承父类时，重写（重新实现）父类中的方法。  
重写（覆盖）的规则：  
   1、重写方法的参数列表必须完全与被重写的方法的相同,否则不能称其为重写而是重载.  
   2、重写方法的访问修饰符一定要大于被重写方法的访问修饰符（public>protected>default>private）。  
   3、重写的方法的返回值必须和被重写的方法的返回一致；  
   4、重写的方法所抛出的异常必须和被重写方法的所抛出的异常一致，或者是其子类；  
   5、被重写的方法不能为private，否则在其子类中只是新定义了一个方法，并没有对其进行重写。

   6、静态方法不能被重写为非静态的方法（会编译出错）。

overload是重载，一般是用于在一个类内实现若干重载的方法，这些方法的名称相同而参数形式不同。  
重载的规则：  
   1、在使用重载时只能通过相同的方法名、不同的参数形式实现。不同的参数类型可以是不同的参数类型，不同的参数个数，不同的参数顺序（参数类型必须不一样）；  
   2、不能通过访问权限、返回类型、抛出的异常进行重载；  
   3、方法的异常类型和数目不会对重载造成影响；

多态的概念比较复杂，有多种意义的多态，一个有趣但不严谨的说法是：继承是子类使用父类的方法，而多态则是父类使用子类的方法。  
一般，我们使用多态是为了避免在父类里大量重载引起代码臃肿且难于维护。

举个例子：  
public class Shape   
{  
   public static void main(String[] args){  
     Triangle tri = new Triangle();  
     System.out.println("Triangle is a type of shape? " + tri.isShape());// 继承

     Shape shape = new Triangle();  
     System.out.println("My shape has " + shape.getSides() + " sides."); // 多态

     Rectangle Rec = new Rectangle();  
     Shape shape2 = Rec;  
     System.out.println("My shape has " + shape2.getSides(Rec) + " sides."); //重载  
   }  
   public boolean isShape(){  
     return true;  
   }  
   public int getSides(){  
     return 0 ;  
   }  
   public int getSides(Triangle tri){ //重载  
     return 3 ;  
   }  
   public int getSides(Rectangle rec){ //重载  
    return 4 ;  
   }  
}

class Triangle extends Shape   
{  
   public int getSides() { //重写,实现多态  
     return 3;  
   }  
}  
class Rectangle extends Shape   
{  
   public int getSides(int i) { //重载  
    return i;  
   }  
}

注意Triangle类的方法是重写，而Rectangle类的方法是重载。对两者比较，可以发现多态对重载的优点：  
如果用重载，则在父类里要对应每一个子类都重载一个取得边数的方法；  
如果用多态，则父类只提供取得边数的接口，至于取得哪个形状的边数，怎样取得，在子类里各自实现(重写)。

**Python 2.x中默认都是经典类，只有显式继承了object才是新式类**

**Python 3.x中默认都是新式类，不必显式的继承object**

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

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

判断基本数据类型可以直接写int，str等，但如果要判断一个对象是否是函数怎么办？可以使用types模块中定义的常量：

>>> **import** types

>>> **def** **fn**():

... **pass**

...

>>> type(fn)==types.FunctionType

True

>>> type(abs)==types.BuiltinFunctionType

True

>>> type(**lambda** x: x)==types.LambdaType

True

>>> type((x **for** x **in** range(10)))==types.GeneratorType

True

**def** trim(s):  
 *# 必须加上s!='' 因为如果字符串为空，则第一个字符是S[0]是不存在的* **while** s!=**'' and** s[0] == **' '**:  
 s = s[1:]  
 **while** s!=**''and** s[-1] == **' '**:  
 s = s[:-1]  
 **return** s  
  
**def** trim(s):  
 **while** s[:1] == **' '**:  
 s = s[1:]  
 **while** s[-1:] == **' '**:  
 s = s[:-1]  
 **return** s\

**s[0]和s[:1]是不一样的**

**python -m pip install docutils pygments pypiwin32 kivy.deps.sdl2 kivy.deps.glew**

**datas=[('dish\_menu.txt', '.'), ('dish\_print1.xsl', '.'), ('dish\_print2.xsl', '.')],**

**\*[Tree(p) for p in (sdl2.dep\_bins + glew.dep\_bins)],**

**import os**

**os.environ['KIVY\_NO\_CONSOLELOG'] = '1'**

**os.environ['KIVY\_USE\_DEFAULTCONFIG']='1'**

**os.environ['KIVY\_HOME']='e:\\'#要使用带\\的完整路径**

**from kivy.config import Config**

**Config.read('e:\\config.ini')**

**Config.set('graphics', 'borderless', '1')**

**Config.set('graphics', 'fullscreen', '1')**

**Config.set('graphics', 'height', '200')**

**Config.write()**