# 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

在pycharm中使用‘VCS’->‘share project on Gitee’相当于如下这条命令：git remote add gitee https://gitee.com/\*.\*

\*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()**

批量替换列表的 字符 a 为 b

>>> list\_=['1','a','3','a']

>>> rep\_list = map(lambda x:[x,'b'][x=='a'],list\_)

>>> rep\_list

['1', 'b', '3', 'b']

>>>

语句解析：

[x=='a']表示对前一个列表取下标，如果为0取第一个值，如果为1取第二个值。

**def** triangles():  
 tri=[1]  
 temp=tri  
 j=0  
 **while** j<10:  
 *# print('中断前tri为')* **yield** tri*#* **for** i **in** range(1,len(tri)):  
 *# print('j=', j, 'i=', i)* tri[i]=temp[i]+temp[i-1]  
 *# print('增加元素后tri为',tri)* tri.append(1)  
 *# print('j=',j,'增加末尾1后tri=',tri)  
 # print('j=',j,'暂存前temp=',temp)* temp=tri[:]#注意这里不能用temp=tri，因为python的变量是指针类型，传可变内容不能直接变量名赋值  
 *# print('j=',j,'暂存后temp=',temp)* j=j+1  
  
s=triangles()  
**for** i **in** s:print(a,i)

***什么是多态***  
  
[面向对象](https://www.baidu.com/s?wd=%E9%9D%A2%E5%90%91%E5%AF%B9%E8%B1%A1&tn=SE_PcZhidaonwhc_ngpagmjz&rsv_dl=gh_pc_zhidao)的三大特性：封装、继承、多态。从一定角度来看，封装和继承几乎都是为多态而准备的。这是我们最后一个概念，也是最重要的知识点。  
多态的定义：指允许不同类的对象对同一消息做出响应。即同一消息可以根据发送对象的不同而采用多种不同的行为方式。（发送消息就是函数调用）  
--------------------------

推荐于2017-09-19

多态性是指允许不同类的对象对同一消息作出响应。多态性包括参数化多态性和包含多态性。多态性语言具有灵活、抽象、行为共享、代码共享的优势，很好的解决了应用程序函数同名问题。  
多态有两种表现形式：重载和覆盖  
首先说重载（overload），是发生在同一类中。与什么父类子类、继承毫无关系。  
标识一个函数除了函数名外，还有函数的参数（个数和类型）。也就是说，一个类中可以有两个或更多的函数，叫同一个名字而他们的参数不同。  
他们之间毫无关系，是不同的函数，只是可能他们的功能类似，所以才命名一样，增加可读性，仅此而已！  
  
再说覆盖(override),是发生在子类中！也就是说必须有继承的情况下才有覆盖发生。***Pyqt中也叫Reimplemented***  
  
我们知道继承一个类，也就有了父类了全部方法，如果你感到哪个方法不爽，功能要变，那就把那个函数在子类中重新实现一遍。  
这样再调用这个方法的时候，就是执行子类中的过程了。父类中的函数就被覆盖了。（当然，覆盖的时候函数名和参数要和父类中完全一样,不然你的方法对父类中的方法就不起任何作用，因为两者是两个函数，毫不关系）

 本回答被提问者采纳

# [*python的反射机制*](https://www.cnblogs.com/Guido-admirers/p/6206212.html)

转载自：http://www.cnblogs.com/feixuelove1009/p/5576206.html

      对编程语言比较熟悉的朋友，应该知道“反射”这个机制。Python作为一门动态语言，当然不会缺少这一重要功能。然而，在网络上却很少见到有详细或者深刻的剖析论文。下面结合一个web路由的实例来阐述python的反射机制的使用场景和核心本质。

### 一、前言

|  |  |
| --- | --- |
| 1  2  3  4  5 | def f1():      print("f1是这个函数的名字！")    s = "f1"  print("%s是个字符串" % s) |

　　在上面的代码中，我们必须区分两个概念，f1和“f1"。前者是函数f1的函数名，后者只是一个叫”f1“的字符串，两者是不同的事物。我们可以用f1()的方式调用函数f1，但我们不能用"f1"()的方式调用函数。说白了就是，**不能通过字符串来调用名字看起来相同的函数！**

### 二、web实例

　　考虑有这么一个场景，根据用户输入的url的不同，调用不同的函数，实现不同的操作，也就是一个url路由器的功能，这在web框架里是核心部件之一。下面有一个精简版的示例：

　　首先，有一个commons模块，它里面有几个函数，分别用于展示不同的页面，代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | def login():      print("这是一个登陆页面！")      def logout():      print("这是一个退出页面！")      def home():      print("这是网站主页面！") |

　　其次，有一个visit模块，作为程序入口，接受用户输入，展示相应的页面，代码如下：（这段代码是比较初级的写法）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | import commons      def run():      inp = input("请输入您想访问页面的url：  ").strip()      if inp == "login":          commons.login()      elif inp == "logout":          commons.logout()      elif inp == "home":          commons.home()      else:          print("404")      if \_\_name\_\_ == '\_\_main\_\_':      run() |

　　我们运行visit.py，输入：home，页面结果如下：

|  |  |
| --- | --- |
| 1  2 | 请输入您想访问页面的url：  home  这是网站主页面！ |

　　这就实现了一个简单的WEB路由功能，根据不同的url，执行不同的函数，获得不同的页面。

　　然而，让我们考虑一个问题，如果commons模块里有成百上千个函数呢(这非常正常)?。难道你在visit模块里写上成百上千个elif?显然这是不可能的！那么怎么破？

### 三、反射机制

　　仔细观察visit中的代码，我们会发现用户输入的url字符串和相应调用的函数名好像！如果能用这个字符串直接调用函数就好了！但是，前面我们已经说了字符串是不能用来调用函数的。为了解决这个问题，python为我们提供一个强大的内置函数：getattr!我们将前面的visit修改一下，代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | import commons      def run():      inp = input("请输入您想访问页面的url：  ").strip()      func = getattr(commons,inp)      func()      if \_\_name\_\_ == '\_\_main\_\_':      run() |

　　首先说明一下getattr函数的使用方法：它接收2个参数，前面的是一个对象或者模块，后面的是一个字符串，**注意了！是个字符串！**

　　例子中，用户输入储存在inp中，这个inp就是个字符串，getattr函数让程序去commons这个模块里，寻找一个叫inp的成员（是叫，不是等于），这个过程就相当于我们把一个字符串变成一个函数名的过程。然后，把获得的结果赋值给func这个变量，实际上func就指向了commons里的某个函数。最后通过调用func函数，实现对commons里函数的调用。这完全就是一个动态访问的过程，一切都不写死，全部根据用户输入来变化。

　　执行上面的代码，结果和最开始的是一样的。

**这就是python的反射，它的核心本质其实就是利用字符串的形式去对象（模块）中操作（查找/获取/删除/添加）成员，一种基于字符串的事件驱动！**

　　这段话，不一定准确，但大概就是这么个意思。

### 四、进一步完善

　　上面的代码还有个小瑕疵，那就是如果用户输入一个非法的url，比如jpg，由于在commons里没有同名的函数，肯定会产生运行错误，具体如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | 请输入您想访问页面的url：  jpg  Traceback (most recent call last):    File "F:/Python/pycharm/s13/reflect/visit.py", line 16, in <module>      run()    File "F:/Python/pycharm/s13/reflect/visit.py", line 11, in run      func = getattr(commons,inp)  AttributeError: module 'commons' has no attribute 'jpg' |

　　那怎么办呢？其实，python考虑的很全面了，它同样提供了一个叫hasattr的内置函数，用于判断commons中是否具有某个成员。我们将代码修改一下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | import commons    def run():      inp = input("请输入您想访问页面的url：  ").strip()      if hasattr(commons,inp):          func = getattr(commons,inp)          func()      else:          print("404")    if \_\_name\_\_ == '\_\_main\_\_':      run() |

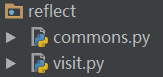
　　通过hasattr的判断，可以防止非法输入错误，并将其统一定位到错误页面。

　　其实，研究过python内置函数的朋友，应该注意到还有delattr和setattr两个内置函数。从字面上已经很好理解他们的作用了。

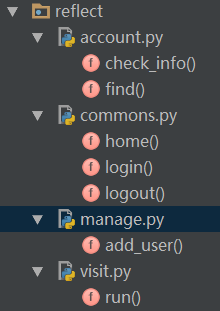
　　python的四个重要内置函数：getattr、hasattr、delattr和setattr较为全面的实现了基于字符串的反射机制。他们都是对内存内的模块进行操作，并不会对源文件进行修改。

### 五、动态导入模块

　　上面的例子是在某个特定的目录结构下才能正常实现的，也就是commons和visit模块在同一目录下，并且所有的页面处理函数都在commons模块内。如下图：



　　但在现实使用环境中，页面处理函数往往被分类放置在不同目录的不同模块中，也就是如下图：



　　难道我们要在visit模块里写上一大堆的import 语句逐个导入account、manage、commons模块吗？要是有1000个这种模块呢？

　　刚才我们分析完了基于字符串的反射，实现了动态的函数调用功能，我们不禁会想那么能不能动态导入模块呢？这完全是可以的！

**python提供了一个特殊的方法：\_\_import\_\_(字符串参数)。通过它，我们就可以实现类似的反射功能。\_\_import\_\_()方法会根据参数，动态的导入同名的模块。**

我们再修改一下上面的visit模块的代码。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | def run():      inp = input("请输入您想访问页面的url：  ").strip()      modules, func = inp.split("/")      obj = \_\_import\_\_(modules)      if hasattr(obj, func):          func = getattr(obj, func)          func()      else:          print("404")    if \_\_name\_\_ == '\_\_main\_\_':      run() |

运行一下：

|  |  |
| --- | --- |
| 1  2  3  4  5 | 请输入您想访问页面的url：  commons/home  这是网站主页面！    请输入您想访问页面的url：  account/find  这是一个查找功能页面！ |

　　我们来分析一下上面的代码：

　　首先，我们并没有定义任何一行import语句；

　　其次，用户的输入inp被要求为类似“commons/home”这种格式，其实也就是模拟web框架里的url地址，斜杠左边指向模块名，右边指向模块中的成员名。

　　然后，modules,func = inp.split("/")处理了用户输入，使我们获得的2个字符串，并分别保存在modules和func变量里。

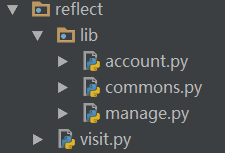
　　接下来，最关键的是obj = \_\_import\_\_(modules)这一行，它让程序去导入了modules这个变量保存的字符串同名的模块，并将它赋值给obj变量。

　　最后的调用中，getattr去modules模块中调用func成员的含义和以前是一样的。

　　总结：通过\_\_import\_\_函数，我们实现了基于字符串的动态的模块导入。

　　同样的，这里也有个小瑕疵！

　　如果我们的目录结构是这样的：



 　　那么在visit的模块调用语句中，必须进行修改，我们想当然地会这么做：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | def run():      inp = input("请输入您想访问页面的url：  ").strip()      modules, func = inp.split("/")      obj = \_\_import\_\_("lib." + modules)      #注意字符串的拼接      if hasattr(obj, func):          func = getattr(obj, func)          func()      else:          print("404")    if \_\_name\_\_ == '\_\_main\_\_':      run( |

　　改了这么一个地方:obj = \_\_import\_\_("lib." + modules)，看起来似乎没什么问题，和import lib.commons的传统方法类似，但实际上运行的时候会有错误。

|  |  |
| --- | --- |
| 1  2  3  4  5 | 请输入您想访问页面的url：  commons/home  404    请输入您想访问页面的url：  account/find  404 |

　　为什么呢？因为**对于lib.xxx.xxx.xxx这一类的模块导入路径，\_\_import\_\_默认只会导入最开头的圆点左边的目录**，也就是“lib”。我们可以做个测试，在visit同级目录内新建一个文件，代码如下：

|  |  |
| --- | --- |
| 1  2 | obj = \_\_import\_\_("lib.commons")  print(obj) |

　　执行结果：

|  |  |
| --- | --- |
| 1 | <module 'lib' (namespace)> |

　　这个问题怎么解决呢？加上fromlist = True参数即可！

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | def run():      inp = input("请输入您想访问页面的url：  ").strip()      modules, func = inp.split("/")      obj = \_\_import\_\_("lib." + modules, fromlist=True)  # 注意fromlist参数      if hasattr(obj, func):          func = getattr(obj, func)          func()      else:          print("404")    if \_\_name\_\_ == '\_\_main\_\_':      run() |

　　至此，动态导入模块的问题基本都解决了，只剩下最后一个，那就是万一用户输入错误的模块名呢？比如用户输入了somemodules/find，由于实际上不存在somemodules这个模块，必然会报错！那有没有类似上面hasattr内置函数这么个功能呢？答案是没有！碰到这种，你只能通过异常处理来解决。

### 六、最后的思考

　　可能有人会问python不是有两个内置函数exec和eval吗？他们同样能够执行字符串。比如：

|  |  |
| --- | --- |
| 1  2  3  4  5 | exec("print('haha')")    结果：    haha |

　　那么直接使用它们不行吗？非要那么费劲地使用getattr，\_\_import\_\_干嘛？

其实，在上面的例子中，围绕的核心主题是如何利用字符串驱动不同的事件，比如导入模块、调用函数等等，这些都是python的反射机制，是一种编程方法、设计模式的体现，凝聚了高内聚、松耦合的编程思想，不能简单的用执行字符串来代替。当然，exec和eval也有它的舞台，在web框架里也经常被使用。

***GET方法说明***

在get和set的过程中你可以控制,譬如合法性检查啊之类的,或者控制只让set不让get或者相反,  
当然,你不set和get也可以,直接存取也可以,只不过没有特别的理由,最好不要这样

# [Python - python里有类似Java的接口(interface)吗？](https://www.cnblogs.com/allen2333/p/9092580.html)

# 参考

1. https://stackoverflow.com/questions/2124190/how-do-i-implement-interfaces-in-python
2. https://stackoverflow.com/questions/372042/difference-between-abstract-class-and-interface-in-python
3. http://www.cnblogs.com/allen2333/p/8878542.html

# 总结

1. 没有。在Python中没必要使用类似Java的interface。因为Python里有多继承和使用鸭子类型。
2. 在Python中，协议就是接口。例如上下文管理协议，只要实现了对应的\_\_enter\_\_, \_\_exit\_\_方法就实现了这个上下文管理协议。
3. Java中的接口是抽象类的特殊情况，抽象类：对一类事物的抽象。接口：对某一行为抽象。Java中的接口里面全部都是抽象方法。
4. 在Python中，可以用抽象类实现接口

from abc import ABCMeta,abstractmethod

class Payment(metaclass=ABCMeta):

@abstractmethod #调用@abstractmethod规定子类必须有pay方法

def pay(self,money):

pass

class Wechatpay(Payment):

def pay(self,money):

print('微信支付了%s元'%money)

obj = Wechatpay()

# [Python - 判断list是否为空](https://www.cnblogs.com/AiyaFocus/p/AiyaFocus.html)

Python中判断list是否为空有以下两种方式：

**方式一：**

1 list\_temp = []

2 if len(list\_temp):

3 # 存在值即为真

4 else:

5 # list\_temp是空的

**方式二：**

1 list\_temp = []

2 if list\_temp:

3 # 存在值即为真

4 else:

5 # list\_temp是空的

以上两种方法均可以判断出 list\_temp 列表是否是空列表，**第二个方法要优于第一个方法**，在Python中，False,0,'',[],{},()都可以视为假。

**json、pickle、shelve库的对比使用**

2018年09月06日 23:59:55 阅读数：15

我们在文件写入和读取时，总会遇到一个问题，就是写入的必须是字符串型（str），读取的也是字符串型。那么，如果我们要写入的是一个字典或者是一个列表之类的，那怎么办呢？没关系，我们可以写入时用str()方法将写入的对象转换成字符串，在读取之后用eval()方法将读取的数据转换成写入前的那个类型。

这种方法没问题，但是是不是觉得有点麻烦呢，或者说是有点low呢。python提供了几个常用的库，来实现数据类型之间的转换。下面介绍一下这些库的使用方法。

一. json库的使用  
import json  
1. json.dumps()-->将json对象转化为文本字符串，例如：

1. a = {"a": 1, "b": {"name": "Jie", "age": 25}, "c": [3, 4, 5]}
2. c = json.dumps(a)
3. print(c, type(c)
4. *#结果： {"a": 1, "b": {"name": "Jie", "age": 25}, "c": [3, 4, 5]} <class 'str'>*

这儿a是一个字典，经过json.dumps()方法之后，结果为一个字符串，也就是说，如果我们要将a写入文件的话，那么可以直接用这种方法，得到一个字符串，然后就可以直接写入，而不用特意的str()了，例如

f.write(json.dumps(a))

可能你会说这和f.write(str(a)有什么不同吗，效果是一样的，但是你不就多学了一种方法吗，何况json用来读写文件只是它 的用途之一而已。json有个专门的写入文件的方法，json.dump(a, f)，注意，这儿时dump，前面说的是dumps，写入的效果是一样的。

2. json.loads()-->将文本字符串转化为json对象

这个方法的一个实际应用就是在读取文件时，正常的读取得到的是字符串，那么既然之前我都用json.dump()写入的，现在我想直接就读取到这个a（字典），那么就可以

1. with open('data.json', 'r', encoding='utf-8') as f:
2. c = json.loads(f.read())
3. print(c, type(c))

运行结果为{'a': 1, 'b': {'name': 'Jie', 'age': 25}, 'c': [3, 4, 5]} <class 'dict'>，这样就实现了写入什么类型，就读取到什么类型。如果说使用str()转化的方法写入的话，那你读取到的也是一个字符串，假如你现在要得到这个字典里的age的值，那你是不是要eval(c)['b']['age']这样，如果读取到的这个数据用到的不多就还好，一旦它在项目中多处用到，那你可能会在其他的地方忘记了它的原始类型，你就直接把它当做一个字典处理（实际上它是一个字符串），这样效率就很低了。

类似上面的json的写入方法，这个读取也有一个专门的方法

c = json.load(f)，结果是一样的。

3. json格式可以理解为就是列表和字典的自由组合形式  
4. 键值对的索引，例如data['姓名']也可以data.get('姓名'),推荐使用第二种，因为如果键不存在，这样也不会报错，返回一个None。get()方法也可以加入第二个参数，如果键没有找到的话，不是返回空，而是返回你加入的那个参数，例如data.get('姓名', '汤姆')  
5. 值得注意的事，JSON的数据需要用双引号，不能用单引号，否则会报json.decoder.JSONecodeError错，这个问题之前我一直没注意到，错了我也半天找不到原因，所以尽量还是用双引号。  
6. json写入文件不能显示中文，在将json对象转为字符串时，会用unicode来编码json中的中文，所以为了在文件中显示的也是中文，可以这样  
    json.dump(data, f, ensure\_ascii=False) ，这儿的ensure\_ascii=False就是控制写入的中文可以正常显示。  
7. dumps()里除了可以传入json对象，ensure\_ascii外，还可以传入一个indent，indent的值就是写入文件时的缩进的长度

二. pickle库的使用

pickle和json的使用方法很像，同样是有dumps,dump,load,和loads方法，特别值得注意的就是pickle时以二进制来操作的，所以不管是文件写入还是文件读取都是以二进制的方式，比如: "rb"，"wb"等。

import pickle  
1. pickle写入文件或打开文件的形式都是二进制类型，比如

with open('data.txt', 'wb') as f:

写入文件同样有两种方法，

    f.write(pickle.dumps(a))  
    pickle.dump(a, f)  
2. pickle写入的文件显示是不正常的，什么叫不正常，就是写入的文件，显示的像是一堆乱码。比如上面定义的那个字典a，写入之后显示的是

€}q (X   aqKX   bq}q(X   nameqX   JieqX   ageqKuX   cq]q(KKKeu.

可能你会以为，既然它是以二进制的方式写入，为什么在文件中不是以二进制显示呢，这个就不深究了，只要知道读取出来的数据是可以正常正常显示的就OK了。

3. pickle没有显示中文的ensure\_ascii参数，但是可以pickle.dumps(data, 3)直接传入一个数字，表示缩进的字符数  
4. 在json中，打开文件时可以使用encoding参数来确定编码格式，但是pickle打开文件不能encoding="utf-8"或是其他，默认就行

上面的json和pickle模块有一个缺点，就是只能写入一个数据，否则会报错。而且用json写入，就必须用json读取，不能用pickle或者是其他的方法。  
           那么有时候确实需要写入多个数据怎么办？下面这个模块就支持多个读取，多个写入

三. shelve库的使用

import shelve  
1. 具体的使用方法，数据写入文件方法如下

1. import shelve
2. import datetime
4. name = ["alan", "rain", "test"]
5. info = {"age": 20, "job": "IT"}
6. f = shelve.open('shelve\_text')
7. f["name"] = name *# 持久化列表（写入）*
8. f["info"] = info
9. f["date"] = datetime.datetime.now() *# 记录当前时间，写入*
10. f.close()

读取文件的方法如下

1. f1 = shelve.open('shelve\_text')
2. print(f1.get("name"))
3. print(f1.get('date'))
4. data = f1.get('info')
5. print(data['age'])
6. print(f1.items())

运行结果

['alan', 'rain', 'test']  
2018-09-07 06:49:06.036216  
20  
2. 注意，这个方法打开文件，写入文件，读取文件的方式均和常规的不一样（是shelve.open()），而且文件没有后缀，原因是因为用这种方法写入的话，就会产生三个文件，后缀是.bak, .dat, .dir，以上面的为例，写入后文件中的内容是

1. 'name', (0, 41)
2. 'info', (512, 39)
3. 'date', (1024, 44)

# QPixmap QImage不能读取jpg

置顶 2018年06月27日 09:25:38 [HuoBaobao小野菊](https://me.csdn.net/kkmdmcgxi) 阅读数：72更多

个人分类： [QT](https://blog.csdn.net/kkmdmcgxi/article/category/7760103)

一、问题

QPixmap/QImage不能读取jpg格式的图像,但是可以读取bmp,png格式的图像

二、解决方法

QPixmap/QImage不能读取jpg图像问题，是因为没有把支持相应格式的动态库放到程序运行目录；

从Qt的安装目录下的plugins文件夹下拷贝文件夹imageformats到程序当前的运行目录下，重新运行你的程序，就能够正常读取jpg图像。

**注意事项：ERIC6*安装完后把安装路径（例如C:\ProgramData\Anaconda3\Lib\site-packages\pyqt5-tools）下的designer.exe 拷贝到 C:\ProgramData\Anaconda3\Lib\site-packages\PyQt5\Qt\bin，否则Eric6运行的时候会报错，说找不到designer***

***ERIC6一定要双击eric6.pyw文件运行进入才能正常！！***

***PYQT从5.6开始就不再提供安装包，需要直接pip安装！***

***ERIC6如果新出版本没有语言包需要从原有版本升级到最新即自带语言包！***

***ANACONDA一定要遵循env隔离的理念来进行使用，不建议使用默认的base环境，自带的包安装不完整！***

# Python类变量，实例变量，类方法，实例方法，静态方法的分析

2017年09月21日 14:37:29 [cling2010](https://me.csdn.net/cling2010) 阅读数：159更多

个人分类： [Python](https://blog.csdn.net/cling2010/article/category/7059639)

Python作为动态语言，跟静态语言如c/c++有很大区别，其中的一个重要的特性就是Python的变量无需声明直接可用。同样，类的成员变量无需声明，直接可用。目的是为了动态语言跟灵活，在思路想到的时候能及时添加成员，而无需在回到类声明的地方，添加成员。但是，同为动态语言，类的成员变量和实例成员变量，生命周期有很大的区别，我们以一个简单的小例子说明：

复制代码

class student(object): '''我们声明一个学生类，希望它有‘姓名’和‘性别’两个属性'''

sex = 'male' '''其中’性别‘这个属性，我们在类中赋值 '''

def \_\_init\_\_(self, name): '''其中‘姓名’这个属性，我们动态赋值，注意观察name属性合适创建

self.\_name = name

stu = student('zhang')

print(dir(stu)) '''我们用Python内置函数dir()查看类的所有属性，方法'''

print(dir(student)) '''我们用Python内置函数dir()查看实例的所有属性，方法'''

通过观察我们发现，实例stu包含2个我们创建的成员，name和sex，这说明实例的属性，不需要声明，在需要用的时候赋值就可以（按照c/c++的思路，student类没有name成员，而Python的理想是，需要用name成员时，直接赋值）

这正是Python动态语言的特性，变量无需声明，应为，Python是解释语言，边解释边创建变量。

通过观察我们发现，类student只有sex成员，没有name成员，这也是动态语言的特性，应为类本质也是个对象（Python一切皆对象），执行类的时候，只声明，不执行\_\_init()\_\_，什么时候执行初始化函数呢？当我们实例一个对象的时候，这说明，动态创建类属性只属于对象，不属于类自己。我们在最后的dir()内置函数输出中再次验证：

['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_doc\_\_', '\_\_format\_\_', '\_\_getattribute\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_module\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', '\_name', 'sex']  
['\_\_class\_\_', '\_\_delattr\_\_', '\_\_dict\_\_', '\_\_doc\_\_', '\_\_format\_\_', '\_\_getattribute\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_module\_\_', '\_\_new\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_weakref\_\_', 'sex']

Process finished with exit code 0

总结：Python是动态语言，类的成员变量无需声明，想到即用，但是，要区分类的属性和实例的属性，动态添加的属性只属于某个实例，不属于类。

另外，对应类变量和实例变量的是：类的三种方法，类方法，实例方法，静态方法。

至于区别，类方法，在声明类可用，实例也可用，但是实例方法只能在实例了一个类之后，才能使用实例方法。

动态给实例绑定方法，得到实例方法，但是该方法只属于实例，不属于类，所以对其他实例不起作用，所以我们开始思考python是否支持，给类绑定方法,确实支持。

利用 \_\_slot\_\_  可以指定哪些属性可以绑定，那些属性不能绑定。

**静态方法**

静态方法是一种普通函数，就位于类定义的命名空间中，它不会对任何实例类型进行操作。使用装饰器@staticmethod定义静态方法。类对象和实例都可以调用静态方法。

码如下:

class Foo:  
    def \_\_init\_\_(self, name):  
        self.name = name  
    def hi(self):  
        print self.name  
    @staticmethod  
    def add(a, b):  
        print a + b

if \_\_name\_\_ == '\_\_main\_\_':  
    foo01 = Foo('letian')  
    foo01.hi()  
    foo01.add(1,2)  
    Foo.add(1, 2)

    运行结果如下：

复制代码代码如下:

letian  
3  
3

1. *# coding=utf-8*
2. class A(object):
3. def \_\_init\_\_(self, x=0):
4. self.x = x
6. *#类实例方法 # 最常见的实例方法*
7. def foo(self, y=0):
8. print self.x, y
10. *#类方法 ＃ cls即为类自身*
11. @classmethod
12. def class\_foo(cls, x=1, y=1):
13. return cls(x) ＃ 因为类方法可以传参一个cls参数，所以可以调用类的相关信息，而静态方法是一个独立的函数，也是可以调用的，这里为了简单，没有调用静态方法
15. *#静态方法 # 不能传递和类或实例相关的参数，如cls或self，但可以传递其他参数*
16. @staticmethod
17. def static\_foo(x=2):
18. return A(x)

测试结果

1. >>> a = A() *# 类实例化*
2. >>> a.x
3. 0
4. >>> a.foo() ＃ 调用类实例方法，可以使用self.x 取出类实例化的(即初始值)
5. 0 0
7. >>> b = A.class\_foo() *# 调用类方法*
8. >>> b.x
9. 1
11. >>> c = A.static\_foo() *# 调用类的静态方法*
12. >>> c.x
13. 2

下面说说区别，实例方法隐含的参数为类实例self，而类方法隐含的参数为类本身cls。   
静态方法无隐含参数，主要为了类实例也可以直接调用静态方法。

所以逻辑上，类方法应当只被类调用，实例方法实例调用，静态方法两者都能调用。主要区别在于参数传递上的区别，实例方法悄悄传递的是self引用作为参数，而类方法悄悄传递的是cls引用作为参数

* **Numpy的轴就是纬度，就是坐标轴的个数**
* **ndarray.ndim**：数组的轴（维度）的个数。在Python世界中，维度的数量被称为rank。
* **ndarray.shape**：数组的维度。这是一个整数的元组，表示每个维度中数组的大小。对于有n行和m列的矩阵，shape将是(n,m)。因此，shape元组的长度就是rank或维度的个数 ndim。
* **ndarray.size**：数组元素的总数。这等于shape的元素的乘积。
* **ndarray.dtype**：一个描述数组中元素类型的对象。可以使用标准的Python类型创建或指定dtype。另外NumPy提供它自己的类型。例如numpy.int32、numpy.int16和numpy.float64。
* **ndarray.itemsize**：数组中每个元素的字节大小。例如，元素为 float64 类型的数组的 itemsize 为8（=64/8），而 complex32 类型的数组的 itemsize 为4（=32/8）。它等于 ndarray.dtype.itemsize 。
* **ndarray.data**：该缓冲区包含数组的实际元素。通常，我们不需要使用此属性，因为我们将使用索引访问数组中的元素。

# numpy 轴的一些理解

2017年12月06日 22:29:02 [bobobe](https://me.csdn.net/u013337743) 阅读数：2559更多

个人分类： [numpy](https://blog.csdn.net/u013337743/article/category/7331224)[python](https://blog.csdn.net/u013337743/article/category/6849286)

numpy中的一些统计函数往往需要指定轴axis   
如:   
np.sum(a,axis = 1)   
那么这个axis=1实际上是说，只对第一轴进行相加操作。比如说：

a = np.array([[1,2,3],[1,2,3]])

print(np.sum(a,axis = 1))

>>>[6 6]

print(np.sum(a,axis = 0))

>>>[2 4 6]

print(np.sum(a))

>>>12

可以看到，当axis=0时，即相加只对第一维度，第一维度的数为[1,2,3]和[1,2,3],所以这两个矩阵相加为[2,4,6]。   
当axis=1时，即相加只对第二维度，第二维度的数相加，第一维不变，即为[1+2+3,1+2+3]=[6,6]。   
当不加axis参数时，把所有元素相加，即为12。

******

# python 找出序列中出现次数最多的元素方法

2017年09月08日 16:32:14 [luoganttcc](https://me.csdn.net/luoganttcc) 阅读数：1380

from collections import Counter

words = [

'look', 'into', 'my', 'eyes', 'look', 'into', 'my', 'eyes',

'the', 'eyes', 'the', 'eyes', 'the', 'eyes', 'not', 'around', 'the',

'eyes', "don't", 'look', 'around', 'the', 'eyes', 'look', 'into',

'my', 'eyes', "you're", 'under'

print (Counter(words))

* 1

Counter({'eyes': 8, 'the': 5, 'look': 4, 'into': 3, 'my': 3, 'around': 2, 'not': 1, "don't": 1, "you're": 1, 'under': 1})

* 1

print (Counter(words).most\_common(4))

[('eyes', 8), ('the', 5), ('look', 4), ('into', 3)]

***Pyinstaller打包經驗：***

***1/如果有問題在cmd窗口運行***

***2/如果彈不出，可以用攝像頭拍下來，慢放來看***

***3/一定要在anaconda相應環境下才能找到對應庫***

***4/pyinstaller本身也有版本，最好安裝到最新版本然後安裝需要的對應包***

***5/如果过期版本包pip找不到，可以下载whl包安装***[***https://www.lfd.uci.edu/~gohlke/pythonlibs/#pyqt4***](https://www.lfd.uci.edu/~gohlke/pythonlibs/#pyqt4)

***Python。Pip安装笔记：***

***Python安装好后，scripts目录下就有pip，如果不能用就是path环境变量没配好！***

## [pipreqs 生成requirements.txt文件时编码错误问题](https://www.cnblogs.com/king123/p/9626537.html)

1，首先安装pipreqs --> pip install pipreqs

2.生成相应项目的路径  --》 pipreqs  e:\a\b

　　在此时可能会遇见

UnicodeDecodeError: 'gbk' codec can't decode byte 0x80 in position 776: illegal multibyte sequence

这个错误.

解决方法：指定编码格式      pipreqs e:\a\b  --encoding=utf8

　　之后会生成requirements.txt 文件,包含了此路径下项目中的依赖项。

　　用法：pip install -r requriements.txt 即可安装所有依赖

***另一输出依赖包命令：***

***Pip freeze > requirements.txt***

# opencv使用VideoCapture无法打开视频

***缺少opencv\_ffmpeg\*.dll文件，64位的系统需要命名为opencv\_ffmpeg\*\_64.dll，\*是版本号。***

把opencv\_ffmpeg2413.dll库拷贝到exe文件路径下就可以了。

***重要提示：***

***如果发现程序在PYCHARM中可以运行，但是打包EXE之后运行不正常，通常是缺少相关DLL文件！***

***如果运行出现Failed to execute 致命错误时：可以在spec编译文件中把console设置为True，然后在CMD中运行即可看见问题！***

***切片：***

前10个数，每两个取一个：

>>> L = list(range(100))

>>> L[:10:2]

[0, 2, 4, 6, 8]

>>> L[:10:-1]#-1表示逆序

[99, 98, 97, 96, 95, 94, 93, 92, 91, 90, 89, 88, 87, 86, 85, 84, 83, 82, 81, 80, 79, 78, 77, 76, 75, 74, 73, 72, 71, 70, 69, 68, 67, 66, 65, 64, 63, 62, 61, 60, 59, 58, 57, 56, 55, 54, 53, 52, 51, 50, 49, 48, 47, 46, 45, 44, 43, 42, 41, 40, 39, 38, 37, 36, 35, 34, 33, 32, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11]

***函数的参数：***

***1、默认参数***可以简化函数的调用。设置默认参数时，有几点要注意：

一是必选参数在前，默认参数在后，否则Python的解释器会报错（思考一下为什么默认参数不能放在必选参数前面）；

二是如何设置默认参数。

当函数有多个参数时，把变化大的参数放前面，变化小的参数放后面。变化小的参数就可以作为默认参数。

***偏函数functools.partial***的作用就是，把一个函数的某些参数给固定住（也就是设置默认值），返回一个新的函数，调用这个新函数会更简单

***2、可变参数***就是传入的参数个数是可变的，是普通位置参数的扩展；

传递参数是也可以使用\*nums表示把nums这个list的所有元素作为可变参数传进去。这种写法相当有用，而且很常见。

可变参数允许你传入0个或任意个参数，这些可变参数在函数调用时自动组装为一个tuple。

**def** **calc**(\*numbers):

sum = 0

**for** n **in** numbers:

sum = sum + n \* n

**return** sum

***3、关键字参数***

关键字参数允许你传入0个或任意个含参数名的参数，这些关键字参数在函数内部自动组装为一个dict。

它可以扩展函数的功能。比如，在person函数里，我们保证能接收到name和age这两个参数，但是，如果调用者愿意提供更多的参数，我们也能收到。试想你正在做一个用户注册的功能，除了用户名和年龄是必填项外，其他都是可选项，利用关键字参数来定义这个函数就能满足注册的需求。

\*\*extra表示把extra这个dict的所有key-value用关键字参数传入到函数的\*\*kw参数，kw将获得一个dict，注意kw获得的dict是extra的一份拷贝，对kw的改动不会影响到函数外的extra。

**def** **person**(name, age, \*\*kw):

print('name:', name, 'age:', age, 'other:', kw)

>>> extra = {'city': 'Beijing', 'job': 'Engineer'}

>>> person('Jack', 24, \*\*extra)

name: Jack age: 24 other: {'city': 'Beijing', 'job': 'Engineer'}

***生成器概念：***

所以，如果列表元素可以按照某种算法推算出来，那我们是否可以在循环的过程中不断推算出后续的元素呢？这样就不必创建完整的list，从而节省大量的空间。在Python中，这种一边循环一边计算的机制，称为生成器：generator。

有两种：

1、第一种方法很简单，只要把一个列表生成式的[]改成()，就创建了一个generator：

>>> L = [x \* x **for** x **in** range(10)]

>>> L

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

>>> g = (x \* x **for** x **in** range(10))

>>> g

<generator object <genexpr> at 0x1022ef630>

2、如果一个函数定义中包含yield关键字，那么这个函数就不再是一个普通函数，而是一个generator。函数是顺序执行，遇到return语句或者最后一行函数语句就返回。而变成generator的函数，在每次调用next()的时候执行，遇到yield语句返回，再次执行时从上次返回的yield语句处继续执行。

***迭代器概念：***

所有的生成器都是Iterator对象，但是迭代器不一定就是生成器，还有可以使用iter()函数把list、dict、str等Iterable变成Iterator。

***高阶函数的概念：***

把函数作为参数传入，这样的函数称为高阶函数，函数式编程就是指这种高度抽象的编程范式。

def \_odd\_iter():  
 n= 1  
 i=1  
 while True:  
 n = n + 2  
 i=i+1  
 yield n  
  
  
def \_not\_divisible(n):  
 result=lambda x: x % n > 0  
 print('返回函数',result)  
 return result#返回的是一个内联函数，x为下次调用时需代入的参数  
  
def primes():  
 yield 2  
 it = \_odd\_iter() # 初始序列  
 while True:  
 n = next(it) # 返回序列的第一个数  
 yield n  
 it = filter(\_not\_divisible(n), it) # 构造新序列，过滤函数，为真保留，为假去除  
  
#打印1000以内的素数:  
for n in primes():  
 if n < 100:  
 print(n)  
 else:  
 break  
#以上等价于以下程序  
m=primes() #返回的m是一个生成器对象  
n = next(m)   
while True:  
 print(n)  
 if n>100:  
 break  
 else:  
 n=next(m)

***装饰器：***

**def** **now**():

print('2015-3-25')

**def** **log**(text):

**def** **decorator**(func):

**def** **wrapper**(\*args, \*\*kw):

print('%s %s():' % (text, func.\_\_name\_\_))

**return** func(\*args, \*\*kw)

**return** wrapper

**return** decorator

@log('execute')

**def** **now**():

print('2015-3-25')

相当于：

now = log('execute')(now)

注意：

**1、这里已经代入了两个参数，3层嵌套相当于已经执行了两层，还剩最后一个wrapper函数**

**2、wrapper的两个参数\*args, \*\*kw是帮助原始now函数进行参数传递的**

以上两种decorator的定义都没有问题，但还差最后一步。因为我们讲了函数也是对象，它有\_\_name\_\_等属性，但你去看经过decorator装饰之后的函数，它们的\_\_name\_\_已经从原来的'now'变成了'wrapper'：

>>> now.\_\_name\_\_

'wrapper'

因为返回的那个wrapper()函数名字就是'wrapper'，所以，需要把原始函数的\_\_name\_\_等属性复制到wrapper()函数中，否则，有些依赖函数签名的代码执行就会出错。

不需要编写wrapper.\_\_name\_\_ = func.\_\_name\_\_这样的代码，Python内置的functools.wraps就是干这个事的，所以，一个完整的decorator的写法如下：

**import** functools

**def** **log**(func):

@functools.wraps(func)

**def** **wrapper**(\*args, \*\*kw):

print('call %s():' % func.\_\_name\_\_)

**return** func(\*args, \*\*kw)

**return** wrapper

或者针对带参数的decorator：

**import** functools

**def** **log**(text):

**def** **decorator**(func):

@functools.wraps(func)

**def** **wrapper**(\*args, \*\*kw):

print('%s %s():' % (text, func.\_\_name\_\_))

**return** func(\*args, \*\*kw)

**return** wrapper

**return** decorator

只需记住在定义wrapper()的前面加上@functools.wraps(func)即可。

**元类：**

1、type()函数既可以返回一个对象的类型，又可以在运行时动态**创建**出新的类型，比如，我们可以通过type()函数创建出Hello类，而无需通过class Hello(object)...的定义：

2、metaclass允许你创建类或者修改类。换句话说，你可以把类看成是metaclass创建出来的“实例”。 先定义metaclass，就可以创建类，最后创建实例。

***装饰器：***

写出一个@log的decorator，使它既支持：

@log

**def** **f**():

**pass**

又支持：

@log('execute')

**def** **f**():

**pass**

def 装饰(参数0):

if isinstance(参数0, str): # 判断参数是否为字符串如果是字符串则是带参数运行

def 第二层(参数1):

@functools.wraps(参数1)

def 第三层(\*args, \*\*kwargs):

开始时间 = perf\_counter() # python3.8中移除time.clock以perf\_counter()代替获取当前系统时间s为单位

print(' begin %s' % 开始时间)

调用 = 参数1(\*args, \*\*kwargs)

print('%s :%s 运行耗费 %f ms \n end \n' % (参数0, 参数1.\_\_name\_\_, perf\_counter()-开始时间))

return 调用

return 第三层

return 第二层

else:

@functools.wraps(参数0)

def 不带参数(\*args, \*\*kwargs):

开始时间 = perf\_counter()

print(' begin %s' % 开始时间)

调用 = 参数0(\*args, \*\*kwargs)

print('%s 运行耗费 %f ms \n end' % (参数0.\_\_name\_\_, perf\_counter()-开始时间))

return 调用

return 不带参数@装饰('带参数')

def 带参():

print('带参数装饰器测试')

@装饰

def 不带参():

print('不带参数装饰器测试')

f1 = 带参()

f2 = 不带参()