# 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"'。