如何简单地理解Python中的if \_\_name\_\_ == '\_\_main\_\_'

2017年09月09日 22:35:42

阅读数：39132

**1. 摘要**

通俗的理解\_\_name\_\_ == '\_\_main\_\_'：假如你叫小明.py，在朋友眼中，你是小明(\_\_name\_\_ == '小明')；在你自己眼中，你是你自己(\_\_name\_\_ == '\_\_main\_\_')。

if \_\_name\_\_ == '\_\_main\_\_'的意思是：当.py文件被直接运行时，if \_\_name\_\_ == '\_\_main\_\_'之下的代码块将被运行；当.py文件以模块形式被导入时，if \_\_name\_\_ == '\_\_main\_\_'之下的代码块不被运行。

**2. 程序入口**

对于很多编程语言来说，程序都必须要有一个入口，比如C，C++，以及完全面向对象的编程语言Java，C#等。如果你接触过这些语言，对于程序入口这个概念应该很好理解，C，C++都需要有一个main函数作为程序的入口，也就是程序的运行会从main函数开始。同样，Java，C#必须要有一个包含Main方法的主类，作为程序入口。

而Python则不同，它属于脚本语言，不像编译型语言那样先将程序编译成二进制再运行，而是动态的逐行解释运行。也就是从脚本第一行开始运行，没有统一的入口。

一个Python源码文件（.py）除了可以被直接运行外，还可以作为模块（也就是库），被其他.py文件导入。不管是直接运行还是被导入，.py文件的最顶层代码都会被运行（Python用缩进来区分代码层次），而当一个.py文件作为模块被导入时，我们可能不希望一部分代码被运行。

**2.1 一个.py文件被其他.py文件引用**

假设我们有一个const.py文件，内容如下：

PI = 3.14

def main():

print("PI:", PI)

main()

# 运行结果：PI: 3.14

现在，我们写一个用于计算圆面积的area.py文件，area.py文件需要用到const.py文件中的PI变量。从const.py中，我们把PI变量导入area.py：

from const import PI

def calc\_round\_area(radius):

return PI \* (radius \*\* 2)

def main():

print("round area: ", calc\_round\_area(2))

main()

'''

运行结果：

PI: 3.14

round area: 12.56

'''

**2.2 修改const.py，添加if \_\_name\_\_ == "\_\_main\_\_"**

我们看到const.py中的main函数也被运行了，实际上我们不希望它被运行，因为const.py提供的main函数只是为了测试常量定义。这时if \_\_name\_\_ == '\_\_main\_\_'派上了用场，我们把const.py改一下，添加if \_\_name\_\_ == "\_\_main\_\_"：

PI = 3.14

def main():

print("PI:", PI)

if \_\_name\_\_ == "\_\_main\_\_":

main()

运行const.py，输出如下：

PI: 3.14

运行area.py，输出如下：

round area: 12.56

如上，我们可以看到if \_\_name\_\_ == '\_\_main\_\_'相当于Python模拟的程序入口，Python本身并没有这么规定，这只是一种编码习惯。由于模块之间相互引用，不同模块可能有这样的定义，而程序入口只有一个。到底哪个程序入口被选中，这取决于\_\_name\_\_的值。

**3. \_\_name\_\_**

**3.1 \_\_name\_\_反映一个包的结构**

\_\_name\_\_是内置变量，可用于反映一个包的结构。假设我们有一个包a，包的结构如下：

a

├── b

│ ├── c.py

│ └── \_\_init\_\_.py

└── \_\_init\_\_.py

在包a中，文件c.py，\_\_init\_\_.py，\_\_init\_\_.py的内容都为：

print(\_\_name\_\_)

当一个.py文件（模块）被其他.py文件（模块）导入时，我们在命令行执行

python -c "import a.b.c"

输出结果：

a

a.b

a.b.c

由此可见，\_\_name\_\_可以清晰地反映一个模块在包中的层次。

**3.2 \_\_name\_\_表示当前模块的名字**

\_\_name\_\_是内置变量，可用于表示当前模块的名字。我们直接运行一个.py文件（模块）

python a/b/c.py

输出结果：

\_\_main\_\_

由此我们可知：如果一个.py文件（模块）被直接运行时，则其没有包结构，其\_\_name\_\_值为\_\_main\_\_，即模块名为\_\_main\_\_。

所以，if \_\_name\_\_ == '\_\_main\_\_'的意思是：当.py文件被直接运行时，if \_\_name\_\_ == '\_\_main\_\_'之下的代码块将被运行；当.py文件以模块形式被导入时，if \_\_name\_\_ == '\_\_main\_\_'之下的代码块不被运行。

**4. \_\_main\_\_.py文件与python -m**

Python的-m参数用于将一个模块或者包作为一个脚本运行，而\_\_main\_\_.py文件相当于是一个包的“入口程序“。

**4.1 运行Python程序的两种方式**

* python xxx.py，直接运行xxx.py文件
* python -m xxx.py，把xxx.py当做模块运行

假设我们有一个文件run.py，内容如下：

import sys

print(sys.path)

我们用直接运行的方式启动

python run.py

输出结果(为了说明问题，输出结果只截取了重要部分，下同)：

['/home/huoty/aboutme/pythonstudy/main', ...]

然后以模块的方式运行:

python -m run.py

输出内容

['', ...]

/usr/bin/python: No module named run.py

由于输出结果只列出了关键的部分，应该很容易看出他们之间的差异：

* 直接运行方式是把run.py文件所在的目录放到了sys.path属性中
* 以模块方式运行是把你输入命令的目录（也就是当前工作路径），放到了 sys.path 属性中。

以模块方式运行还有一个不同的地方：多出了一行No module named run.py的错误。实际上以模块方式运行时，Python先对run.py执行一遍 import，所以print(sys.path)被成功执行，然后Python才尝试运行run.py模块，但是在path变量中并没有run.py这个模块，所以报错。正确的运行方式，应该是python -m run。

**4.2 \_\_main\_\_.py的作用**

仍然先看例子，假设我们有如下一个包package：

package

├── \_\_init\_\_.py

└── \_\_main\_\_.py

其中，文件\_\_init\_\_.py的内容

import sys

print("\_\_init\_\_")

print(sys.path)

其中，文件\_\_main\_\_.py的内容

import sys

print("\_\_main\_\_")

print(sys.path)

接下来，我们运行这个package，使用python -m package运行，输出结果：

\_\_init\_\_

['', ...]

\_\_main\_\_

['', ...]

使用python package运行，输出结果：

\_\_main\_\_

['package', ...]

总结一下

* 当加上-m参数时，Python会把当前工作目录添加到sys.path中；而不加-m时，Python则会把脚本所在目录添加到sys.path中。
* 当加上-m参数时，Python会先将模块或者包导入，然后再执行。
* \_\_main\_\_.py文件是一个包或者目录的入口程序。不管是用python package还是用python -m package运行，\_\_main\_\_.py文件总是被执行。

**5. 参考文章**

[1. Python 中的 if **name** == ‘**main**’ 该如何理解](http://blog.konghy.cn/2017/04/24/python-entry-program/)