# 第八章 Python模块

本节所讲内容：

8.1 模块的概念

8.2 模块导入的原理

8.3 import导入的几种方式

8.4 简单的导入关系

8.5 关于if\_\_name\_\_==“\_\_main\_\_”

8.6 time模块

8.7 os模块

## 8.1 模块的概念

### 8.1.1 模块的概念介绍

什么是模块:

模块是一个包含有一系列数据、函数、类等组成的程序组

模块是一个文件，模块文件名通常以.py结尾.

作用:

让一些相关的数据，函数，类等有逻辑的组织在一起，使逻辑结构更加清晰

模块中的数据,函数，类等可提供给其它模块或程序使用

模块的分类:

内建模块(builtins) 在解析器的内可以直接使用:map()

标准库模块, 安装Python时已安装且可以直接使用os,time

第三方模块(通常开源),需要自己安装

用户自己编写的模块(可以作为其它人的第三方模块)

模块的导入语句

import 语句

语法:

import 模块名1 [as 模块新名1], 模块名2 [as 模块新名2], ....

例：test01.py是一个简单的模块

def print\_func(a):

print('hello ,world')

print(a)

例：imp\_test是一个Python包

\_\_init\_\_.py

imp\_test01.py 是其中的一个模块

def print\_func(a):

print('hello ,world')

print(a)

## 8.2 模块导入的原理

### 8.2.1 原理介绍

1. 模块导入会将要导入的文件执行一遍

2. 导入模块的名称就是我们定义的脚本或包的名称

3. 我们导入模块的过程总结就是：在指定的范围内搜索指定名称对Python脚本或者包，将其运行，获取其中的方法

例：在test02.py中导入上例中test01.py模块，可见其运行结果

import test01

test01.print\_func(18)

运行结果：

hello ,world

18

例：在test02.py中导入上例中imp\_test01.py模块，可见其运行结果

form imp\_test import imp\_test01

imp\_test01.print\_func(10)

运行结果：

hello ,world

10

### 8.2.2 导入的路径范围

由sys模块的sys.path方法来规定

import sys

print(sys.path)

['E:\\workspace\\importTest', 'E:\\workspace\\importTest', 'E:\\xuegod\\Python35.zip', 'E:\\xuegod\\DLLs', 'E:\\xuegod\\lib', 'E:\\xuegod', 'E:\\xuegod\\lib\\site-packages']

因为sys.path返回的数据类型是列表

1、列表可以修改、我们导入的范围也可以通过修改sys.path这个列表获得暂时的修改。例如通过sys.path.append()添加目录，导入额外目录的模块。

2、列表是有序的，当搜索的过程当中，在第一个路径下搜索到了，就停止搜索。而且sys.path第一个路径是脚本的当前路径，所以禁止大家将自己的脚本命名成模块的名称。

常见错误：例：把test01.py改名为os.py，在test02.py中进行导入

improt os

os.print\_func(18)

运行结果：

AttributeError: module 'os' has no attribute 'path'

3、Python导入模块，被导入的脚本在第一次被导入之后会自动形成一个.pyc临时文件，之后，再次被导入时，实际上导入的是这个.pyc的临时文件，发生这样的错误之后，除了要修改文件名称还要把当前目录下同名的.pyc文件删除掉（适用于Python2）。

4、reload() 函数

reload函数是使用模块名称，而不是变量名称，也就是说当你把它改变了名称，它也是加载原来的名称。

reload [ˌri:ˈləʊd]：再装填

当使用reload时，这个模块会被重新编译，新的模块替换原来的模块。如果之前从这个模块创建的实例仍然使用旧的模块实现。特别是使用from-import语句加载的名称，也不会被更新。

简单来说：用于重新载入之前载入的模块

import test01  
from importlib import reload  
reload(test01)  
test01.print\_func(18)

reload 会重新加载已加载的模块，但原来已经使用的方法还是会使用旧的模块，而新生产的方法会使用新的模块；

reload 后还是用原来的内存地址；

reload 不支持 from ××× import ××× 格式的模块进行重新加载。

## 8.3 import导入的几种方式

### 8.3.1 import导入几种方式介绍

|  |  |  |
| --- | --- | --- |
| 导入关键字 | 导入对象 | 导入后的使用 |
| import | 模块 | 模块.功能import sys, sys.path |
| from模块import功能 | 功能 | 功能from sys import path |
| from模块import \* | 功能 | 功能这里同样是导入了所有功能 |
| from模块import功能,功能 | 功能 | 功能可以导入多个功能 |
| as | 给导入的模块或者功能起别名 | |

from os import path

from datetime import \*

from time import time ,localtime

from os import path as pas

注意：

在Python2版本中目录不可以直接被导入，但是3可以。

Python2目录下有\_\_init\_\_.py文件才可以被导入,并且\_\_init\_\_.py导入后自动执行

## 8.4 简单的导入关系

### 8.4.1 导入模块的本质

导入模块的本质就是把Python文件解释一遍

分析：import module\_name🡪就是要解释module\_name.py🡪首先要找到module\_name.py的位置🡪查找系统路径sys.path，当前路径优先。

### 8.4.2 导入包的本质

导入包的本质就是执行该包下的\_\_init\_\_.py文件

导入包下边的imp\_test01.py文件

1. 修改\_\_init\_\_.py文件，导入包下边的py文件

from . import imp\_test01

这样就可以直接在test02导入import 包名（如果不加不能导入包下的模块）。

2. 在py文件中导入包下边的模块imp\_test01.py文件

from imp\_test import imp\_test01

## 8.5 关于if\_\_name\_\_==“\_\_main\_\_”

### 8.5.1 本脚本

在我们脚本创建之初，Python会默认加载一些内建的变量，\_\_name\_\_就是其中一个

当我们脚本自己执行，也就是直接执行这个脚本的时候\_\_name\_\_的值是\_\_main\_\_

print(\_\_name\_\_)

运行结果：

\_\_main\_\_

### 8.5.2 导入的模块

但是我们如果导入执行脚本，该脚本的\_\_name\_\_属性的值就是文件名本身

import os

print(os.\_\_name\_\_)

运行结果

os

下面咱们思考一下？

如果在这个test01中加入if\_\_name\_\_==“\_\_main\_\_”：

def print\_func(a):

print('hello,world')

print(a)

#判断是不是自己调用本身

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

print\_func(10)

如果我们在test02中导入test01

from test01 import print\_func

print\_info(18)

运行结果如下：

hello,world

18

如果上面的代码没有看懂 ，我们可以联想一下生活中的例子，在生活中别人看我，我的代号就是我的名字’for’,那如果我自己看自己就是自己本身。

结合例子看代码：

在test01中我们执行程序，\_\_name\_\_ = ‘\_\_main\_\_’,如果我在test02中导入test01，那么test01的\_\_name\_\_就会变成test01(也就是模块名本身)。

### 8.5.3 Python的相对导入

from . import test01 from .. import ……

包含相对路径import 的Python脚本不能直接运行，只能作为module被引用。

原因正如官方手册中描述的，所谓相对路径其实就是相对于当前module的路径，但如果直接执行脚本这个module的name就是“\_\_main\_\_”, 而不是module原来的name， 这样相对路径也就不是原来的相对路径了，导入就会失败，出现错误“ValueError: Attempted relative import in non-package”或者是：SystemError: Parent module '' not loaded, cannot perform relative import。

### 8.5.4 Python中有趣的导入

1、>>> import \_\_hello\_\_

Hello World...

2、经典的 Python 之禅

import this

运行此命令将显示由 Tim Peters 编写的"Zen Of Python（Python 之禅）"。据传这是 Python 中的『八荣八耻』，每个有追求的 Python 程序员都应该谨记于心。

>>> import this

The Zen of Python, by Tim Peters

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than \*right\* now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!

参考翻译：

优美胜于丑陋。

显式胜于隐式。

简单胜于复杂。

复杂胜于难懂。

扁平胜于嵌套。

分散胜于密集。

可读性应当被重视。

尽管实用性会打败纯粹性，特例也不能凌驾于规则之上。

除非明确地使其沉默，错误永远不应该默默地溜走。

面对不明确的定义，拒绝猜测的诱惑。

用一种方法，最好只有一种方法来做一件事。

虽然一开始这种方法并不是显而易见的，但谁叫你不是 Python 之父呢。

做比不做好，但立马去做有时还不如不做。

如果实现很难说明，那它是个坏想法。

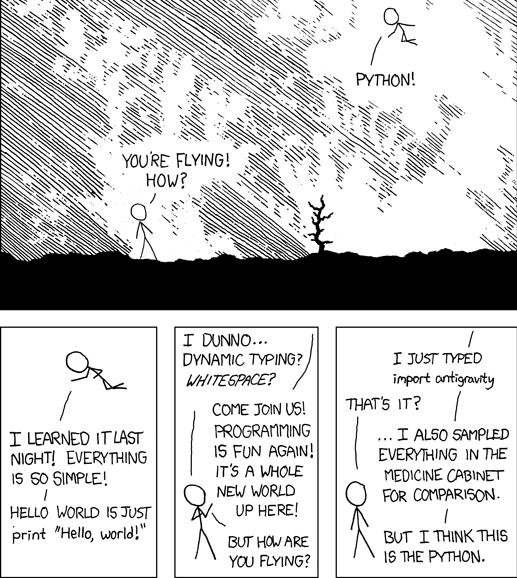
如果实现容易解释，那它有可能是个好想法。

命名空间是个绝妙的想法，让我们多多地使用它们吧！

3、“反重力”漫画

>>>import antigravity

antigravity [ˌæntiˈgrævɪti]：反重力



## 8.6 time模块

### 8.6.1 Python基础的时间模块介绍

时区

时区：时区（Time Zone)是地球上的区域使用同一个时间定义。

零时区：零时区: 7.5°W～7.5°E 时区中心线 0° ，它只是一个时区，就像北京的东8区。在这个时区里面的，就是格林威治标准时间。比如北京是今天早上10点，那么那里就是上午2点。

UTC0时区(中国东八区:上海)

闰秒

闰秒（或称跳秒）是对协调世界时作出加一秒或减一秒的调整，用来调整世界时间。

1998年底调整过一次。61秒(地球自转偏差导致的多出的1秒)

时间戳

时间戳是指格林威治时间1970年01月01日00时00分00秒(北京时间1970年01月01日08时00分00秒)起至现在的总秒数。通俗的讲， 时间戳是一份能够表示一份数据在一个特定时间点已经存在的完整的可验证的数据

1970年1月1日0:00(距离上面这个时间过了多少秒)

夏令时

夏令时日光时为了节省能源，每年夏天人工将时间调早一个小时。

0 : 采用

1 ： 不采用,

-1： 系统判断

### 8.6.2 时间的格式

1. 字符串时间

2. 九元素时间

1、年tm\_year

2、月tm\_mon

3、日tm\_mday

4、时tm\_hour

5、分tm\_min

6、秒tm\_sec1-61

7、一周的第几天tm\_wday 0-6

8、一年的第几天tm\_yday

9、是否夏令时tm\_isdst

1: 启用

0: 不启用

-1: 不知道

8.6.3 时间

|  |  |  |
| --- | --- | --- |
| time.localtime | 返回当前时区的九元素时间 | 九元素时间 |
| time.gmtime | 返回0时区的九元素时间 |
| time.struct\_time | 返回一个九元素元组时间 |
| time.asctime | 将九元素时间转换成字符串格式周月日时分秒年 | 字符串时间 |
| time.ctime | 将时间戳转换为字符串时间周月日时分秒年 |
| time.strftime | 定制指定格式字符串时间 |
| time.time | 返回当前时间的时间戳 | 时间戳时间 |
| time.mktime | 返回九元素时间的时间戳 |
| time.clock | 在Linux下记录进程运行时间，在win下记录是距离上一次调用过了多久。第一次调用，返回的是进程运行的实际时间。而第二次之后的调用是自第一次调用以后到现在的运行时间 | |
| time.sleep | 就是让程序挂起指定秒 | |

>>> import time

返回当前时区的九元素时间

>>> time.localtime()

time.struct\_time(tm\_year=2018, tm\_mon=8, tm\_mday=6, tm\_hour=16, tm\_min=5, tm\_sec=29, tm\_wday=0, tm\_yday=218, tm\_isdst=0)

struct [strʌkt]：结构

返回0时区的九元素时间

>>> time.gmtime()

time.struct\_time(tm\_year=2018, tm\_mon=8, tm\_mday=6, tm\_hour=8, tm\_min=5, tm\_sec=41, tm\_wday=0, tm\_yday=218, tm\_isdst=0)

>>> time.struct\_time(time.localtime())

time.struct\_time(tm\_year=2018, tm\_mon=8, tm\_mday=6, tm\_hour=16, tm\_min=6, tm\_sec=7, tm\_wday=0, tm\_yday=218, tm\_isdst=0)

将九元素时间转换成字符串格式周月日时分秒年

>>> time.asctime()

'Mon Aug 6 16:08:50 2018'

将时间戳转换为字符串时间周月日时分秒年

>>> time.ctime()

'Mon Aug 6 16:09:08 2018'

返回当前时间的时间戳

>>> time.time()

1533542965.5388699

返回九元素时间的时间戳

>>> time.mktime(time.localtime())

1533544213.0

>>>

在Linux下记录进程运行时间，在win下记录是距离上一次调用过了多久

>>> time.clock()

3.4212304455297346e-06

就是让程序挂起指定秒

>>> time.sleep(5)

4. strftime的格式符

%y两位

%Y四位

%m月

%d日

%H24时

%I12小时

%M分

%S秒

>>> time.strftime('%Y-%m-%d',time.localtime())

'2018-08-06'

>>> time.strftime('%y-%m-%d',time.gmtime())

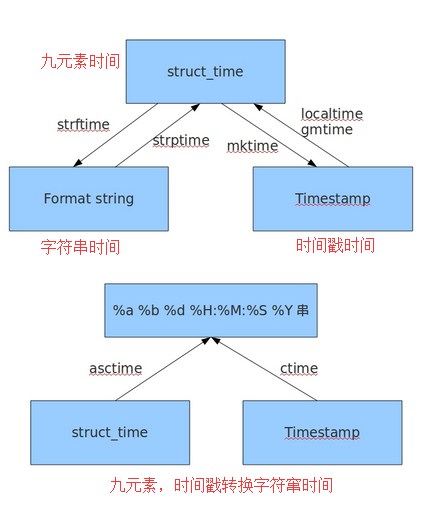
'18-08-06'

>>> data\_str = '2018/5/20'

>>> time.strptime(data\_str,'%Y/%m/%d')

time.struct\_time(tm\_year=2018, tm\_mon=5, tm\_mday=20, tm\_hour=0, tm\_min=0, tm\_sec=0, tm\_wday=6, tm\_yday=140, tm\_isdst=-1)

>>>



## 8.7 os模块

### 8.7.1 Python系统路径和其他一些操作模块

|  |  |
| --- | --- |
| os常用功能 | |
| os.sep | 符合当前系统的路径分割符，Linux/windows\ |
| os.name | 返回操作系统类型windows“nt”Linux“posix” |
| os.getcwd | 返回当前的工作目录 |
| os.listdir | 列出指定目录下的目录和文件 |
| os.chdir | 修改当前的工作路径 |
| os.mkdir | 创建目录 |
| os.makedirs | 递归创建目录 |
| os.remove | 删除文件 |
| os.rmdir | 删除文件夹 |
| os.removedirs | 递归删除文件夹 |
| os.system | 执行系统命令 |
| os.popen | 执行系统命令，会将结果以文件的形式返回(命令，权限，缓存) |

1.os.name——判断现在正在实用的平台，Windows 返回 ‘nt'; Linux 返回’posix'

    rename(需要修改的文件名, 新的文件名) 也可以做剪切。

2. os.getcwd()——得到当前工作的目录。

3. os.listdir()——指定所有目录下所有的文件和目录名。

　　以列表的形式全部列举出来，其中没有区分目录和文件。

4．os.mkdir()——创建目录

注意：这样只能建立一层，要想递归建立可用：os.makedirs(‘x/y/z’)

5. os.chdir()——改变目录到指定目录

7. os.remove()——删除指定文件

8. os.rmdir()——删除指定文件夹

os.removedirs(‘x/y/z’)递归删除

9. os.system()——执行系统命令

import os

a = 'mkdir wdir'

b =os.system(a)

print(b)

10. os.popen()——执行系统命令，会将结果以文件的形式返回

os.popen(command[, mode[, bufsize]])

案例：

import os

a = 'mkdir nwdir'

b =os.popen(a,'r',1)

print(b)

import os

for i in os.popen('dir'):

print(i)

总结：

8.1 模块的概念

8.2 模块导入的原理

8.3 import导入的几种方式

8.4 简单的导入关系

8.5 关于if\_\_name\_\_==“\_\_main\_\_”

8.6 time模块

8.7 os模块