模块化

|  |  |
| --- | --- |
| 语句 | 含义 |
| import 模块1[,模块2,...] | 完全导入 |
| import ... as ... | 模块别名 |

一般来说，编程语言中，库、包、模块是同一种概念，是代码组织方式。

Python中只有一种模块对象类型，但是为了模块化组织模块的便利，提供了"包"的概念。

模块module，指的是Python的源代码文件。

包package，指的是模块组织在一起的和包名同名的目录及其相关文件。

导入语句

import语句

1、找到指定的模块，加载和初始化它，生成模块对象。找不到，抛出异常

2、在import所在的作用域的局部命名空间中，增加名称和上一步创建的对象关联

单独运行下面例子，体会区别

import  functools  #  导入模块

print(dir())  #  [...,  'functools']

print(functools)  #  <module  'functools'  from  'path/to/functools.py'>

print(functools.wraps)  #  <function  wraps  at  0x00000000010FB400>

import  os.path  #  导入os.path，os加入当前名词空间

print(dir())  #  [...,  'os']

print(os)  #  <module  'os'  from  'path/to/os.py'>

print(os.path)  #  完全限定名称访问path

import  os.path  as  osp  #  导入os.path并赋给osp

print(dir())  #[...,  'osp']

print(osp)  #  <module  'ntpath'  from  'path/to/path.py'>

总结

导入顶级模块，其名称会加入到本地名词空间中，并绑定到其模块对象。

导入非顶级模块，只将其顶级模块名称加入到本地名词空间中。导入的模块必须使用完全限定名称来访问。

如果使用了as，as后的名称直接绑定到导入的模块对象，并将该名称加入到本地名词空间中。

from语句

|  |  |
| --- | --- |
| 语句 | 含义 |
| from ... import ... | 部分导入 |
| from ... import ... as ... | 别名 |

from  pathlib  import  Path,  PosixPath  #  在当前名词空间导入该模块指定的成员

print(dir())  #  [...,  'Path',  'PosixPath']

from  pathlib  import  \*  #  在当前名词空间导入该模块所有公共成员（非下划线开头成员）或指定成员

print(dir())  #  [...,  'Path',  'PosixPath',  'PurePath',  'PurePosixPath',  'PureWindowsPath',

'WindowsPath']

from  functools  import  wraps  as  wr,  partial  #  别名

print(dir())  #  [...,  'wr',  'partial']

from  os.path  import  exists  #  加载、初始化os、os.path模块，exists加入本地名词空间并绑定

if  exists('o:/t'):

print('Found')

else:

print('Not  Found')

print(dir())

print(exists)

import  os

#  4种方式获得同一个对象exists

print(os.path.exists)

print(exists)

print(os.path.\_\_dict\_\_['exists'])

print(getattr(os.path,  'exists'))

总结

找到from子句中指定的模块，加载并初始化它（注意不是导入）

对于import子句后的名称

1. 先查from子句导入的模块是否具有该名称的属性

2. 如果不是，则尝试导入该名称的子模块

3. 还没有找到，则抛出ImportError异常

4. 这个名称保存到本地名词空间中，如果有as子句，则使用as子句后的名称

from  pathlib  import  Path  #  导入类Path

print(Path,  id(Path))

import  pathlib  as  pl  #  导入模块使用别名

print(dir())

print(pl)

print(pl.Path,  id(pl.Path))

#  可以看出导入的名词Path和pl.Path是同一个对象

自定义模块

自定义模块：.py文件就是一个模块

#  test1.py文件

print('This  is  test1  module')

class  A:

def  showmodule(self):

print(1,  self.\_\_module\_\_,  self)

print(2,  self.\_\_dict\_\_)

print(3,  self.\_\_class\_\_.\_\_dict\_\_)

print(4,  self.\_\_class\_\_.\_\_name\_\_)

a  =  A()

a.showmodule()

#  test2.py文件

import  test1

a  =  test1.A()

a.showmodule()

#  test3.py文件

from  test1  import  A  as  cls

a  =  cls()

a.showmodule()

自定义模块命名规范

1. 模块名就是文件名

2. 模块名必须符合标识符的要求，是非数字开头的字母数字和下划线的组合。test-module.py这样的文件名不

能作为模块名。也不要使用中文。

3. 不要使用系统模块名来避免冲突，除非你明确知道这个模块名的用途

4. 通常模块名为全小写，下划线来分割

模块搜索顺序

使用 sys.path 查看搜索顺序

import  sys

#  print(\*sys.path,  sep='\n')

for  p  in  sys.path:

print(p)

显示结果为，python模块的路径搜索顺序

当加载一个模块的时候，需要从这些搜索路径中从前到后依次查找，并不搜索这些目录的子目录。

搜索到模块就加载，搜索不到就抛异常

路径也可以为字典、zip文件、egg文件。

.egg文件，由setuptools库创建的包，第三方库常用的格式。添加了元数据（版本号、依赖项等）信息的zip文件

路径顺序为

程序主目录，程序运行的主程序脚本所在的目录

PYTHONPATH目录，环境变量PYTHONPATH设置的目录也是搜索模块的路径

标准库目录，Python自带的库模块所在目录

sys.path可以被修改，增加新的目录

模块的重复导入

#  test1.py文件

print('This  is  test1  module')

class  A:

def  showmodule(self):

print(1,  self.\_\_module\_\_,  self)

print(2,  self.\_\_dict\_\_)

print(3,  self.\_\_class\_\_.\_\_dict\_\_)

print(4,  self.\_\_class\_\_.\_\_name\_\_)

a  =  A()

a.showmodule()

#  test2.py文件

import  test1

print('local  module')

import  test1

import  test1

从执行结果来看，不会产生重复导入的现象。

所有加载的模块都会记录在sys.modules中，sys.modules是存储已经加载过的所有模块的字典。

打印sys.modules可以看到os、os.path都已经加载了。

模块运行

|  |  |
| --- | --- |
| 属性 | 含义 |
| \_\_file\_\_ | 字符串，源文件路径 |
| \_\_cached\_\_ | 字符串，编译后的字节码文件路径 |
| \_\_spec\_\_ | 显示模块的规范 |
| \_\_name\_\_ | 模块名 |
| \_\_package\_\_ | 当模块是包，同  \_\_name\_\_  ;否则，可以设置为顶级模块的空字符串 |

\_\_name\_\_  ，每个模块都会定义一个  \_\_name\_\_  特殊变量来存储当前模块的名称，如果不指定，则默认为源代码文

件名，如果是包则有限定名。

解释器初始化的时候，会初始化sys.modules字典（保存已加载的模块），加载builtins（全局函数、常量）模

块、  \_\_main\_\_  模块、sys模块，以及初始化模块搜索路径sys.path

Python是脚本语言，任何一个脚本都可以直接执行，也可以作为模块被导入。

当从标准输入（命令行方式敲代码）、脚本（$ python test.py）或交互式读取的时候，会将模块的  \_\_name\_\_  设置

为  \_\_main\_\_  ，模块的顶层代码就在  \_\_main\_\_  这个作用域中执行。顶层代码：模块中缩进最外层的代码。

如果是import导入的，其  \_\_name\_\_  默认就是模块名

#  test1.py文件

import  test2

#  test2.py文件

#  判断模块是否以程序的方式运行  $python  test.py

if  \_\_name\_\_  ==  '\_\_main\_\_':

print('in  \_\_main\_\_')  #  程序的方式运行的代码

else:

print('in  import  module')  #  模块导入的方式运行的代码

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

1. 本模块的功能测试

对于非主模块，测试本模块内的函数、类

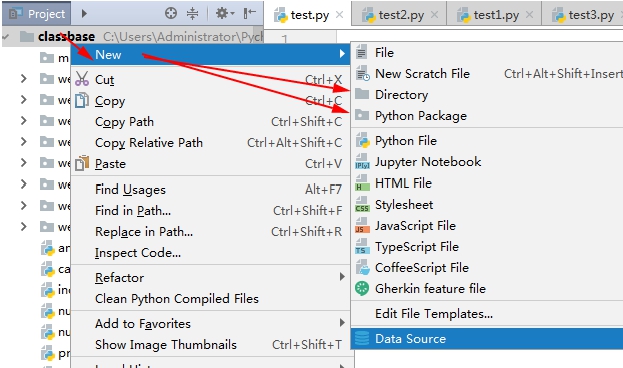
2. 避免主模块变更的副作用

顶层代码，没有封装，主模块使用时没有问题。但是，一旦有了新的主模块，老的主模块成了被导入模块，

由于原来代码没有封装，一并执行了。

模块的属性

import  t3



for  k,v  in  t3.\_\_dict\_\_.items():

print(k,  str(v)[:80])

print(dir(t3))

for  name  in  dir(t3):

print(getattr(t3,  name))

包

包，特殊的模块

Python模块支持目录吗？

实验

项目中新建一个目录m，使用下面的代码

import  m

print(m)

print(type(m))

print(dir(m))  #  没有\_\_file\_\_

竟然可以导入目录m，目录也是文件，所以可以导入，不过问题是，目录模块怎么写入代码？

为了解决这个问题，Python要求在目录下建立一个特殊文件  \_\_init\_\_.py  ，在其中写入代码

pycharm中，创建Directory和创建Python package不同，前者是创建普通的目录，后者是创建一个带有

\_\_init\_\_.py   文件的目录即包

Python中，目录可以作为模块，这就是包，不过代码需要写在该目录下  \_\_init\_\_.py  中

子模块

包目录下的py文件、子目录都是其子模块

m

|--  \_\_init\_\_.py

|--  m1.py

|--  m2

|--  \_\_init\_\_.py

|--  m21

|--  \_\_init\_\_.py

|--  m22.py

如上建立子模块目录和文件，所有的py文件中就写一句话   print(\_\_name\_\_)

#  注意观察已经加载的模块、当前名词空间的名词

#import  m

#import  m.m1

#from  m  import  m1

#from  m.m2  import  m21

import  m.m2.m21

print('-'\*30)

print(dir())

print('-'\*30)

import  sys

print(sorted(filter(lambda  x:x.startswith('m'),  sys.modules.keys())))

删除  \_\_init\_\_.py  试一试，可以发现删除并不影响导入，但是这不是良好的习惯，请保留  \_\_init\_\_.py  文件

模块和包的总结

包能够更好的组织模块，尤其是大的模块，其代码行数很多，可以把它拆分成很多子模块，便于使用某些功能就加

载相应的子模块。

包目录中  \_\_init\_\_.py  是在包第一次导入的时候就会执行，内容可以为空，也可以是用于该包初始化工作的代码，

最好不要删除它（低版本不可删除  \_\_init\_\_.py  文件）

导入子模块一定会加载父模块，但是导入父模块一定不会导入子模块

包目录之间只能使用.点号作为间隔符，表示模块及其子模块的层级关系

模块也是封装，如同类、函数，不过它能够封装变量、类、函数。

模块就是命名空间，其内部的顶层标识符，都是它的属性，可以通过  \_\_dict\_\_  或dir(module)查看。

包也是模块，但模块不一定是包，包是特殊的模块，是一种组织方式，它包含  \_\_path\_\_  属性

问题

from  json  import  encoder  之后，  json.dump  函数用不了，为什么？

import  json.encoder  之后呢？  json.dump  函数能用吗？

原因是from json import encoder之后，当前名词空间没有json，但是json模块已经加载过了，没有json的引用，

无法使用dump函数。

import json.encoder也加载json模块，但是当前名词空间有json，因此可以调用json.dump。

绝对导入、相对导入

绝对导入

在import语句或者from导入模块，模块名称最前面不是以.点开头的

绝对导入总是去模块搜索路径中找，当然会查看一下该模块是否已经加载

相对导入

只能在包内使用，且只能用在from语句中

使用.点号，表示当前目录内

..表示上一级目录

不要在顶层模块中使用相对导入

举例a.b.c模块，c是模块c.py，c的代码中，使用

from . import d # imports a.b.d

from .. import e # imports a.e

from .d import x # a.b.d.x

from ..e import x # a.e.x

...  三点表示上上一级

使用下面结构的包，体会相对导入的使用

m

|--  \_\_init\_\_.py

|--  m1.py

|--  m2

|--  \_\_init\_\_.py

|--  m21

|--  \_\_init\_\_.py

|--  m22.py

测试一下有相对导入语句的模块，能够直接运行吗？

不能了，很好理解，使用相对导入的模块就是为了内部互相的引用资源的，不是为了直接运行的，对于包来说，正

确的使用方式还是在顶级模块使用这些包。

注意：一旦一个模块中使用相对导入，就不可以作为主模块运行了

访问控制

下划线开头的模块名

\_ 或者 \_\_ 开头的模块是否能够被导入呢？

创建文件名为 \_xyz.py 或者 \_\_xyz.py 测试

都可以成功的导入，因为它们都是合法的标识符，就可以用作模块名

模块内的标识符

#  xyz.py

print(\_\_name\_\_)

A  =  5

\_B  =  6

\_\_C  =  7

\_\_my\_\_  =  8

#  test.py中

import  xyz

import  sys

print(sorted(sys.modules.keys()))

print(dir())

print(xyz.A,  xyz.\_B,  xyz.\_\_C,  xyz.\_\_my\_\_)

普通变量、保护变量、私有变量、特殊变量，都没有被隐藏，也就是说模块内没有私有的变量，在模块中定义不做

特殊处理。

#  from语句

from  xyz  import  A,  \_B  as  B,  \_\_my\_\_,  \_\_C  as  C

import  sys

print(sorted(sys.modules.keys()))

print(dir())

print(A,  B,  \_\_my\_\_,  C)

依然可以使用from语句，访问所有变量

from ...  import \* 和  \_\_all\_\_

使用from ...  import \* 导入

#  xyz.py

print(\_\_name\_\_)

A  =  5

\_B  =  6

\_\_C  =  7

\_\_my\_\_  =  8

#  test.py中

from  xyz  import  \*

import  sys

print(sorted(sys.modules.keys()))

print(dir())

print(locals()['A'])

A  =  55

print(locals()['A'])  #  思考这个A是谁的A了

结果是只导入了A，下划线开头的都没有导入

使用 \_\_all\_\_

\_\_all\_\_  是一个列表，元素是字符串，每一个元素都是一个模块内的变量名

#  xyz.py中

\_\_all\_\_  =  ["X",  "Y"]

print(\_\_name\_\_)

A  =  5

\_B  =  6

\_\_C  =  7

\_\_my\_\_  =  8

X  =  10

Y  =  20

#  test.py中

from  xyz  import  \*

import  sys

print(sorted(sys.modules.keys()))

print(dir())

#print(locals()['A'])

print(locals()['X'])

print(locals()['Y'])

修改  \_\_all\_\_  列表，加入下划线开头变量，看看什么效果



#  xyz.py中

\_\_all\_\_  =  ["X",  "Y",  "\_B",  "\_\_C"]

print(\_\_name\_\_)

A  =  5

\_B  =  6

\_\_C  =  7

\_\_my\_\_  =  8

X  =  10

Y  =  20

#  test.py中

from  xyz  import  \*

import  sys

print(sorted(sys.modules.keys()))

print(dir())

#print(locals()['A'])

print(locals()['X'])

print(locals()['Y'])

print(locals()['\_B'])

print(locals()['\_\_C'])

可以看到使用from xyz import \*导入  \_\_all\_\_  列表中的名称

包和子模块

m

|--  \_\_init\_\_.py

|--  m1.py

#  \_\_init\_\_.py中

print(\_\_name\_\_)

x  =  1

#  m1.py中

print(\_\_name\_\_)

y  =  5

#  test.py中

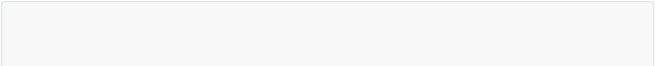
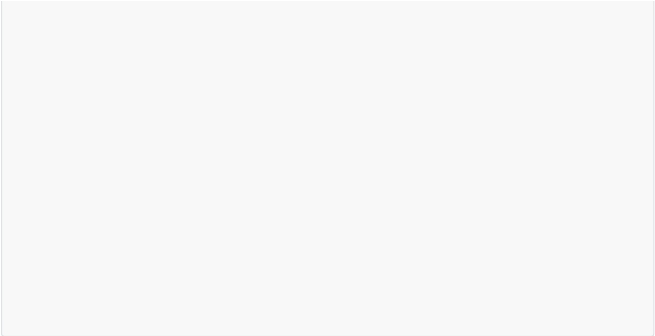
#  如何访问到m1.py中的变量y

#  访问到m.m1的变量y的几种实现

#  方法1，直接导入m1模块

import  m.m1

print(m.m1.y)



#  方法2，直接导入m.m1的属性y

from  m.m1  import  y

print(y)

#  方法3，from  m  import  \*

#  print(dir())

#  该方法导入后，无法看到子模块m1，无法访问y

#  在\_\_init\_\_.py增加\_\_all\_\_  =  ['x',  'm1']，使用\_\_all\_\_提供导出的名称

from  m  import  \*

print(m1.y)

#  方法4，不使用\_\_all\_\_

#  在\_\_init\_\_.py增加from  .  import  m1

from  m  import  \*

print(m1.y)

\_\_init\_\_.py  中有什么变量，则使用from m import \*加载什么变量，这依然符合模块的访问控制

#  \_\_init\_\_.py文件

print(\_\_name\_\_)

x  =  1

from  .m1  import  y  as  \_z

print(dir())

总结

一、使用  from  xyz  import  \*   导入

1. 如果模块没有  \_\_all\_\_  ，  from  xyz  import  \*   只导入非下划线开头的该模块的变量。如果是包，子模块也不

会导入，除非在  \_\_all\_\_  中设置，或  \_\_init\_\_.py  中导入它们

2. 如果模块有  \_\_all\_\_  ，  from  xyz  import  \*   只导入  \_\_all\_\_  列表中指定的名称，哪怕这个名词是下划线开头

的，或者是子模块

3.   from  xyz  import  \*  方式导入，使用简单，但是其副作用是导入大量不需要使用的变量，甚至有可能造成名

称的冲突。而  \_\_all\_\_  可以控制被导入模块在这种导入方式下能够提供的变量名称，就是为了阻止from xyz

import \*导入过多的模块变量，从而避免冲突。因此，编写模块时，应该尽量加入  \_\_all\_\_

二、from module import name1, name2 导入

这种方式的导入是明确的，哪怕是导入子模块，或者导入下划线开头的名称

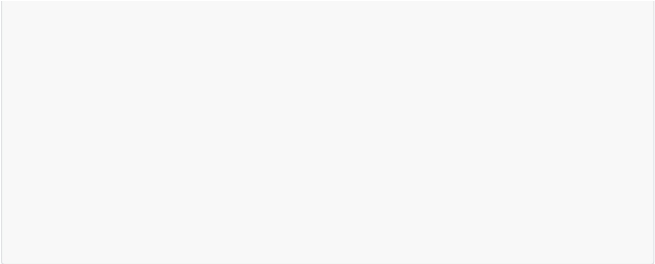
程序员可以有控制的导入名称和其对应的对象

模块变量的修改

#  xyz.py

print(\_\_name\_\_)

X  =  10



#  test2.py

import  xyz

print(xyz.X)

#  test.py

import  xyz

print(xyz.X)

xyz.X  =  50

import  test2

模块对象是同一个，因此模块的变量也是同一个，对模块变量的修改，会影响所有使用者。

除非万不得已，或明确知道自己在做什么，否则不要修改模块的变量。

前面学习过的猴子补丁，也可以通过打补丁的方式，修改模块的变量、类、函数等内容。