1.模块与包

为什么要学习“模块和包”?

- 因为随着代码量的增加，我们需要更有逻辑地组织代码，以及代码文件之间的关系。
- 让代码更容易被定位、归类和引用。
- 让我们可以重复使用自己的和他人的代码，可以多人协作更大型的项目。

1.1 模块 （Module）

1.1.1 模块的概念

Python 模块(Module)，是一个 Python 文件，以 .py 结尾，包含了 Python 对象定义和Python语句。

对于Python中包、模块、函数或者类的理解，可以参照下表：

概念	状态
包	文件夹
模块	文件
函数或者类	文件中的内容

1.1.2 模块的导入

模块导入，相当于把别人的代码拿到自己的文件中使用。

语法：*import 模块名*

（1）导入系统的内置模块

例如:math是python语言中内置的模块，提供了很多数学中的常量与操作。常量math.pi表示圆周率Π;math.pow(2,3)表示2的3次方。

要引用模块 math，就可以在文件最开始的地方用 **import math** 来引入:

```
import math
print(math.pi)          #输出：3.141592653589793
print(type(math))       #输出：<class 'module'>
```

> 在模块名上 按住“Ctrl + 鼠标左键”，可以打开并跳转到引入的模块源文件中。

（2）导入自定义模块

定义一个模块，名为myModule1.py，其中代码为：

```
def add(a,b):
    print("mymodule1:add....")
    return f"{a}+{b}={a+b}"

def divid(a,b):
    print("mymodule1:divid....")
    return f"{a}/{b}={a/b}"

def multiply(a,b):
    print("mymodule1:multiply....")
    return f"{a}*{b}={a*b}"
```

引用自定义的myModule1模块，并调用其中的函数。

调用模块中的函数，语法：*模块名.函数名([实际参数])*

```
import myModule1

print(myModule1.add(3,2))       #输出：mymodule1:add....
                                #      3+2=5
print(myModule1.divid(3,2))     #输出：mymodule1:divid....
                                #      3/2=1.5
```

1.1.3 模块的运行方式

定义一个模块，名为myModule2.py，其中代码为：

```
#可执行语句
print("mymodule2...")
m = 10
print(f"mymodule2:m={m}")
```

定义模块test2.py，导入自定义模块myModule2时，被导入模块的"可执行语句"会立即执行。

```
import myModule2        #输出：mymodule2...
                        #输出：mymodule2:m=10
import myModule2
```

注意：执行多次import，一个模块只会被导入一次。上面的打印语句只会输出1次。

> **不提倡上面这种直接在模块中编写功能性语句的做法**
>
> 因为模块被外界引入时，这些代码会默认执行。会对调用当前模块的外部程序造成影响。

既不想影响外部代码，又想要保持本模块内相关功能代码的运行。我们该怎么做呢？

我们可以通过模块中的默认属性__name__来判断。

我们在module2.py 中录入下面代码并执行

```
print(__name__,type(__name__))      #输出：__main__ <class 'str'>
```

我们再执行test2.py，我们会看到上面语句的输出结果为：

```
print(__name__,type(__name__))      #输出：myModule2 <class 'str'>
```

Python中对于模块的调用有两种模式，

- **脚本模式**：自身模块开发时，作为独立程序由解释器直接运行
  __name__ 的内容为字符串：__main__
- **模块模式**：被其他模块导入，为其他模块提供资源（变量、函数、类的定义）
  __name__ 的内容为字符串：myModule2（模块的名字）

在myModule2中编写：

```
if __name__ == '__main__':
    m = 10
    print(f"mymodule2:m={m}")
    print(add(3,2))
```

更常见的一种方式：将函数放在主函数外部上方。

- 以保证被别人调用函数的时候，不会调用自身的内部函数。
- 还可以保证通过这种脚本方式的判断更加安全的实现函数的测试。

```
def main():
    m = 10
    print(f"mymodule2:m={m}")
    print(add(3,2))

#快捷方式：输入m，根据代码提示回车，生成下面代码
#以脚本程序运行，以保证被其他模块调用的时候自动运行自身的可执行代码
if __name__ == '__main__':
    # m = 10
    # print(f"mymodule2:m={m}")
    # print(add(3,2))

    #更常见的写法：
    main()
```

注意：自定义模块的执行只会有一次。

当再次导入自定义模块时，将不会有作用。

```
import myModule2        #导入自定义模块时，被导入模块的”可执行语句“会立即执行
import myModule2        #你执行多次import，一个模块只会被导入一次。
```

1.1.4 模块搜索路径

整个搜索过程顺序：

（1）内置模块（例如：math）

首先搜索系统内部的模块。

（2）当前模块所在的目录

如果在内置模块中没有找到，那就搜索当前所在目录。

如果没有找到，模块名下会有红线提示。运行结果会有如下报错：

```
ModuleNotFoundError:No module named 'myModule3'
```

新建myModule3后，再次运行成功调用自定义myModule3：

```
import myModule3     #输出：myModule3...
```

（3）环境变量PYTHONPATH（默认包含python的安装路径）

将原myModule3.py通过refactor重命名为myModule3_.py，重新将此文件放在别的路径下，并复制路径到环境变量下。（打开我的电脑 - 属性 - 高级 - 环境变量 - 系统变量 - 新建）

变量名为“PYTHONPATH”，变量值为刚才复制路径+“ \”，选择确定。

运行下方代码后依然报错：

```
import myModule3
```

测试是否环境变量配置错误：

右键点击windows按钮 - 选择“运行” - 输入“cmd” - 界面中输入“python”。

然后输入“import myModule3”，界面返回结果“myModule3...”，说明myModule3是可以正常被访问并调用。

由于Pycharm在启动时会将环境变量加载到缓存中并使用，所以在使用中修改环境变量导致修改的环境变量不能及时被反应出来，所以需要重启Pycharm。

重启后正常运行。

（4）Python安装路径下的Lib文件夹

Python安装路径下的Lib，代表库，是Python默认安装的库。

将myModule3_.py文件放在当下Lib文件夹路径下，运行代码依然可以成功访问它。

（5）lib文件夹下的site-packages文件夹（第三方模块）

该文件夹下保存着安装的第三方的工具，将文件myModule3_.py粘贴到该文件夹下，运行代码仍然可以访问到。

（6）sys.path.append()追加的目录

运行以下代码，输出结果为一系列路径。这些路径为系统在进行查找时搜索的路径。

```
import sys
print(sys.path)
```

尝试在该路径中追加一些路径：

先将之前第三方文件路径下的myModule3_.py删除，在该项目下选择“New Directory” - 命名"lib".

将文件复制到该“lib”文件夹下，运行下方代码进行追加路径：

```
import myModule3

import sys
# print(sys.path)

sys.path.append("D:\XX..")      # ""内为新建lib所在路径
print(sys.path)                 # 查看结果是否添加了新的路径，结果中已经打印
                                # 出“myModule3...”
```

提问：**为什么在Lib文件夹下找不到time、math和sys模块?**

因为这些为系统内置模块，想要查看它们，可通过运行下列代码：

```
import sys
print(sys.builtin_module_names)
```

【扩展】 指定搜索路径

以后代码需要放到其他电脑上运行，如何保证仍然能够找到这个文件夹呢?

绝对路径：从盘符出发的路径
相对路径：不是从盘符出发的路径

```
import os

print(__file__)      #输出当前文件的全路径：
D:/itsishu/python_workspace/demo4/demo1.py

print(os.path.dirname(__file__))    #获取指定文件所在的目录（文件夹）
                                    #输出：D:/itsishu/python_workspace/demo4
```

注意：获取文件路径不能使用字符串截取，因为不同操作系统的路径表示不同

```
import sys
import os

print(os.path.dirname(__file__) + r"/lib")       #获取当前项目路径下的lib文件夹
sys.path.append(os.path.dirname(__file__) + r"/lib")
print(sys.path)          #查看当前系统中的路径
```

输出结果：

> D:/itsishu/python_workspace/demo4/lib

```
['D:\itsishu\python_workspace\demo4', 'D:\itsishu\python_workspace\demo4',
'D:\itsishu\python_workspace\resource', 'D:\Program Files\JetBrains\PyCharm
2020.2.3\plugins\python\helpers\pycharm_display', 'D:\soft\python3.8.6\python38.zip',
'D:\soft\python3.8.6\DLLs', 'D:\soft\python3.8.6\lib', 'D:\soft\python3.8.6',
'D:\soft\python3.8.6\lib\site-packages', 'D:\Program Files\JetBrains\PyCharm
2020.2.3\plugins\python\helpers\pycharm_matplotlib_backend',
'D:/itsishu/python_workspace/demo4/lib']
```

1.1.5 其他方式导入模块

（1）从模块中引入指定的函数　（from ... import ...)

```
from myModule1 import add,divid        #引入模块中多个函数，用逗号分隔

print(add(3,2))                  #直接使用 函数名（） 进行调用
print(divid(3,2))
print(myModule1.add(3,2))      #没有被引入的函数，不能被调用
```

（2）一次性引入模块中全部函数　（from ... import *)

```
from myModule1 import *
print(add(3,2))           #引入myModule1中的全部函数
print(divid(3,2))
print(multiply(3,2))
```

（3）import 和 from ... import ...的区别

```
import myModule1
print(myModule1.add(3,2))        #需要使用模块名来调用函数

from myModule1 import *
def add(x,y):
   return f"10*{x}+10{y}={10*(x+y)}"

print(add(3,2))                  #可以直接调用函数，默认优先调用自身定义的函数
```

（4）使用别名，解决变量或函数重名问题

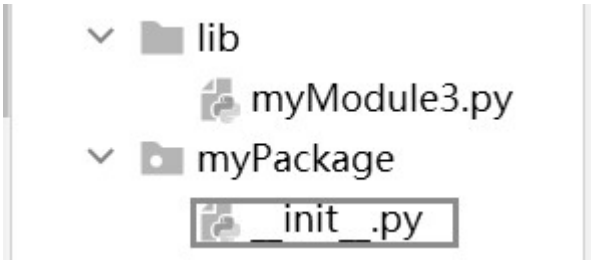
```
#给函数起别名，避免命名冲突
from myModule1 import add as m1add
def add(x,y):
   return f"10*{x}+10{y}={10*(x+y)}"

print(add(3,2))
print(m1add(3,2))

#给模块起别名，简化书写
import myModule1 as m
print(m.add(3,2))
```

1.2 包 （Package）

1.2.1 包的概念

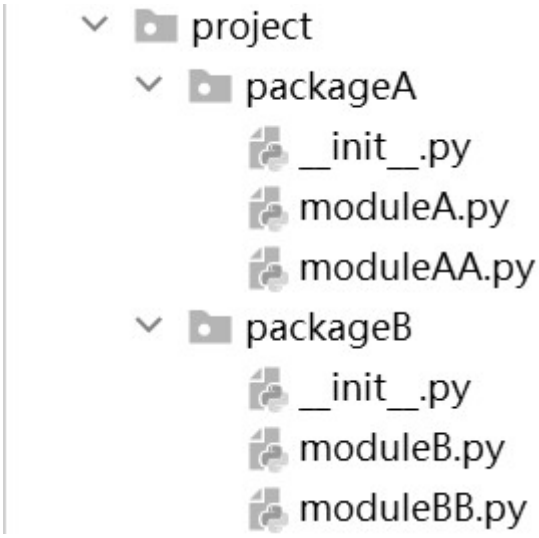
包的本质就是一个文件夹，有__init__.py文件作为标识，不能删除。该标识可以自己创建。

包下可以建立不同模块，方便大型项目管理文件。



在demo4项目文件夹下新建project文件夹，并在project下新建两个包packageA和packageB。

在packageA下新建moduleA.py和moduleAA，



moduleA.py中的内容:

```
info = "moduleA..."
print(info)

def testA():
   print("moduleA:testA()....")
```

moduleAA.py中的内容:

```
info = "moduleAA..."
print(info)

def testAA():
   print("moduleA:testAA()....")
```

在demo4下新建python文件demo2.py，

（1） 调用moduleA中“info”:

```
import project.packageA

print(project.packageA.moduleA.info)  #不能直接调用moduleA.info,需要使用全名
project.packageA.moduleA.testA()      # 运行结果：moduleA...
                                      #          moduleA...
                                      #          moduleA:testA()....
                                      # 出现两次打印moduleA，是因为第一次打印是在
                                      # moduleA里面的print(info),第二次打印是调用
                                      # moduleA时打印的。
```

（2）更简单的调用方式：

```
from project.packageA import moduleA
print(moduleA.info)
moduleA.testA()
```

（3）模块内的变量或函数的直接导入：

```
from project.packageA.moduleA import testA,info
print(info)
testA()
```

输出结果：

> moduleA...
> moduleA...
> moduleA:testA()...

运行结果依然不受影响

（4）包间的模块导入

在packageB中新建Python文件moduleB.py

```
from project.packageA.moduleA import testA,info
print(info)
testA()
```

运行结果一样：

> moduleA...
> moduleA...
> moduleA:testA()....

1.2.2 包和模块的导入

本质：引入了__init__.py文件

示例（1）

在packageA中的__init__.py编辑：

```
print("packageA__init__被调用了...")
```

在demo2.py中调用：

```
import project.packageA.moduleA
```

运行结果表明__init__.py模块被执行了：

> packageA**init**被调用了...
> moduleA...

示例（2）：将包内模块一次性先导入，最后直接引用包来引用所有模

块在packageA的__init__.py文件中：

```
import project.packageA.moduleA
print("packageA__init__被调用了...")
```

在demo2.py中：

```
import project.packageA
project.packageA.moduleA.testA()
```

运行结果：

> moduleA...
> packageA**init**被调用了...
> moduleA:testA()....

但是在demo2.py中调用moduleAA失败

在__init__.py中引用moduleAA：

```
import project.packageA.moduleAA
```

运行成功。

```

```