1. **Python中的模块**

**第一部分**

**一、模块的基础**

模块其实就是一些函数和类的集合文件，它能实现一些相应的功能，当我们需要使用这些功能的时候，直接把相应的模块导入到我们的程序中，其后缀名是.py。

模块可以被别的程序引入，以使用该模块中的函数等功能。想使用 Python 源文件(模块），只需在另一个源文件里执行 import 语句，语法如下：

import module1[, module2[,... moduleN]

当解释器遇到 import 语句，如果模块在当前的搜索路径就会被导入。搜索路径是一个解释器会先进行搜索的所有目录的列表。当我们使用import语句的时候，Python解释器是怎样找到对应的文件的呢？

这就涉及到Python的搜索路径，搜索路径是由一系列目录名组成的，Python解释器就依次从这些目录中去寻找所引入的模块。sys.path是python的搜索模块的路径集，是一个list。



如果执行导入某个模块的部分内容，可以使用from…import 语句，Python的from语句让你从模块中导入一个指定的部分到当前命名空间中，语法如下：

from modname import name1[, name2[, ... nameN]]

例如，要导入模块 math 的 sqrt 函数，使用如下语句：

>>> from math import sqrt

>>> sqrt(16)

这个声明不会把整个matho模块导入到当前的命名空间中，它只会将math里的 sqrt函数引入进来。

可以一次性的把模块中的所有（函数，变量）名称都导入到当前模块中:

from 模块名 import \*

这将把所有的名字都导入进来，但是那些由单一下划线（\_）开头的名字不在此内。例如：

from math import \*

查看math模块中所有的方法：

>>> import math

>>> dir(math)

['\_\_doc\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']

**二、模块的分类**

1、自定义模块

先自写一个文件，取名为a.py,在a.py中编写代码：

def sum2(n):

count = 0

for t in range(n+1):

count=count+t

return count

把a.py放在C:\Python27\Lib\site-packages路径中。

在b.py文件中引入a模块,再调用a模块中的sum2()方法。

import a

print(a.sum2(100))

2、系统模块

Python 本身带着一些标准的模块库。

1. 第三方模块

安装：

使用pip install 模块名

卸载：

使用 pip uninstall模块名

比如使用chardet模块判断文件的编码格式等

**第二部分**

**三、常见系统模块**

3.1、sys模块

1、sys.argv

sys.argv其实就是一个元组，里边的项为用户输入的参数，关键就是要明白这参数是从程序外部输入的，而非代码本身的什么地方，要想看到它的效果就应该将程序保存了,其中第一个参数是指本表示代码本身文件路径，后面的参数表示从外部接收的数据。

写代码如下，并保存为e.py

import sys

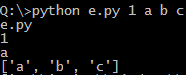
print(sys.argv[0])

print(sys.argv[1])

print(sys.argv[2])

print(sys.argv[2:])

得到的结果如下：



2、sys.path

sys.path是python的搜索模块的路径集，是一个list。



3、错误输出重定向和程序终止

sys 还有 stdin，stdout 和 stderr 属性，即使在 stdout 被重定向时，后者也可以用于显示警告和错误信息。我们通常用的print，其实调用的就是sys.stdout.write方法，print 将需要的内容打印到控制台，然后追加一个换行符，以下两行代码等价：

sys.stdout.write(‘hello‘ + ‘\n‘)

print(‘hello‘)。

而sys.stderr.write一般用于打印错误信息。

import sys

#接收来自键盘的输入,参数为字符数

str=sys.stdin.read(20)

#接收来自键盘的输入行数,参数为字符数 ，返回值包含字符的行,并以列表的形式返回

str2=sys.stdin.readlines(2)

#接收来自键盘的输入行数,参数为行数，返回值字符串

str3=sys.stdin.readline(1)

print(str)

print(str2)

print(str3)

3.2、os模块

1、内置的函数 dir() 可以找到模块内定义的所有名称。以一个字符串列表的形式返回:

import a

print dir(a)

执行结果：

['\_\_builtins\_\_', '\_\_doc\_\_', '\_\_file\_\_', '\_\_name\_\_', '\_\_package\_\_', 'sum2']

#print(a.sum2(100))

print(a.\_\_builtins\_\_)

#print(a.\_\_doc\_\_)

#print(a.\_\_file\_\_)

#print(a.\_\_name\_\_)

#print dir(a)

一个模块被另一个程序第一次引入时，其主程序将运行。如果我们想在模块被引入时，模块中的某一程序块不执行，我们可以用\_\_name\_\_属性来使该程序块仅在该模块自身运行时执行。每个模块都有一个\_\_name\_\_属性，当其值是'\_\_main\_\_'时，表明该模块自身在运行，否则是被引入。

#a.py

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

print('True')

else:

print('False')

2、help()

可以获取相关帮助信息



3.3 math模块为浮点运算提供了对底层C函数库的访问:

import math

print math.ceil(23.45)

print math.sqrt(16)

print math.pi

print math.sin(2/math.pi)

3.4 随机random

import random

print(random.choice(['apple', 'pear', 'banana']))

print(random.sample(range(100), 10)) # sampling without replacement

#[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]

print(random.random()) # random float

print(random.randrange(6)) # random integer chosen from range(6)

执行结果：

apple

[7, 10, 80, 17, 83, 26, 30, 9, 52, 55]

0.0893055452858

1

**第三部分**

1. **包**

在创建许许多多模块后，我们可能希望将某些功能相近的文件组织在同一文件夹下，这里就

需要运用包的概念了  
包是一个有层次的文件目录结构，它定义了由n个模块或n个子包组成的python应用程序执

行环境。通俗一点：包是一个包含\_\_init\_\_.py 文件的目录，该目录下一定得有这个\_\_init\_\_.py

文件和其它模块或子包。

通常包总是一个目录，可以使用import导入包，或者from + import来导入包中的部分模块。包目录下为首的一个文件便是 \_\_init\_\_.py。然后是一些模块文件和子目录，假如子目录中也有 \_\_init\_\_.py 那么它就是这个包的子包了。比如C:\Users\afeng\AppData\Local\Programs\Python\Python36\Lib\email是包，mime就是它的子包。

包对应于文件夹，使用包的方式跟模块也类似，唯一需要注意的是，当文件夹当作包使用时，文件夹需要包含\_\_init\_\_.py文件，主要是为了避免将文件夹名当作普通的字符串。\_\_init\_\_.py的内容可以为空，一般用来进行包的某些初始化工作或者设置\_\_all\_\_值，\_\_all\_\_是在from package-name import \*这语句使用的，全部导出定义过的模块。

 常见的包结构如下：

package\_a

├── \_\_init\_\_.py

├── module\_a1.py

└── module\_a2.py

package\_b

├── \_\_init\_\_.py

├── module\_b1.py

└── module\_b2.py

可以从包中导入单独的模块。

1、import PackageA.SubPackageA.ModuleA,使用时必须用全路径名

2、from PackageA.SubPackageA import ModuleA, 可以直接使用模块名而不用加上包前缀。

3、也可以直接导入模块中的函数或变量：from PackageA.SubPackageA.ModuleA import functionA

 4.2 自定义包

Python中的package定义很简单，其层次结构与程序所在目录的层次结构相同，这一点与Java类似，唯一不同的地方在于，python中的package必须包含一个\_\_init\_\_.py的文件。

\_\_init\_\_.py可以为空，只要它存在，就表明此目录应被作为一个package处理。当然，\_\_init\_\_.py中也可以设置相应的内容。有时在import语句中会出现通配符\*，导入某个module中的所有元素，这是怎么实现的呢？答案就在\_\_init\_\_.py，我们在subPack1的\_\_init\_\_.py文件中写  
\_\_all\_\_ = ['module\_13', 'module\_12']，from package1.subPack1 import \*就会导入'module\_13', 'module\_12'这两个模块，也就是说，以\*导入时，package内的module是受\_\_init\_\_.py限制的。

定义好的包放在C:\Users\afeng\AppData\Local\Programs\Python\Python36\Lib或者C:\Users\afeng\AppData\Local\Programs\Python\Python36下

Package

\_\_init\_\_.py

Sum.py

Subpackage

\_\_init\_\_.py

Sum1.py

Sum2.py

Package的\_\_init\_\_.py:

\_\_all\_\_ = ['Sum','subpackage']

Sum.py

#encoding=utf-8

def sum():

a = 0

for i in range(101):

a=a+i

return a

Subpackage的\_\_init\_\_.py:

\_\_all\_\_ = ['Sum1','Sum2']

Sum1.py

#encoding=utf-8

def sum1(begin):

a = 0

for i in range(begin,101):

a=a+i

return a

Sum2.py:

#encoding=utf-8

def sum2(begin,end):

a = 0

for i in range(begin,end+1):

a=a+i

return a

调用子包下的方法：

**import** package  
**from** package.subpackage **import** \*  
print(package.subpackage.sum1.sum1(20))