模块与包

模块

模块介绍

在计算机程序的开发过程中，随着程序代码越写越多，在一个文件里代码就会越来越长，越来越不容易维护。

为了编写可维护的代码，我们把很多函数分组，分别放到不同的文件里，这样，每个文件包含的代码就相对较少，很多编程语言都采用这种组织代码的方式。而这样的一个`py`文件在Python中称为模块（Module）。

模块是组织代码的更高级形式，大大提高了代码的阅读性和可维护性。

模块一共3种：

- 解释器内建模块
- 第三方模块
- 应用程序自定义模块

另外，使用模块还可以避免函数名和变量名冲突。相同名字的函数和变量完全可以分别存在不同的模块中，因此，我们自己在编写模块时，不必考虑名字会与其他模块冲突。

模块导入

```
'''
# 方式1：导入一个模块
import 模块名
import 模块名 as 别名

# 方式2：导入多个模块
import (
    模块1
    模块2
    )

import 模块1,模块2

# 方式3：导入成员变量
from 模块名 import 成员变量
from 模块名 import *
'''
```

> 导入模块时会执行模块，多次导入只执行一次。

案例：

```
cal.py
logger.py
main.py
```

```python
#cal.py
def add(x,y):
    return x + y

def mul(x,y):
    return x * y

print('这是cal模块文件')
```

```python
#logger.py
def get_logger():
    print('打印日志! ')

print('这是logger模块文件')
```

```python
#main.py
import cal #导入了cal模块
import logger #导入和logger模块
#注意：import导入模块，就好比是将模块中的代码执行了
from cal import mul #将cal模块中的mul成员进行导入


#调用用了cal模块中的add函数
result = cal.add(1,2)
print(result)
c = mul(3,4)
print(c)
```

```
#调用了logger模块中的get_logger函数
logger.get_logger()
```

- 注意：执行源文件的名字一定不要和模块的名字同名

__name__

`__name__`是python内置变量，存储的是当前模块名称。

对于很多编程语言来说，程序都必须要有一个入口。像C，C++都有一个main函数作为程序的入口，而Python作为解释性脚本语言，没有一个统一的入口，因为Python程序运行时是从模块顶行开始，逐行进行翻译执行，所以，最顶层（没有被缩进）的代码都会被执行，所以Python中并不需要一个统一的main()作为程序的入口。

在刚才的案例中三个模块都打印一次`__name__`

```
#cal.py
def add(x,y):
    return x + y

def mul(x,y):
    return x * y

print('这是cal模块文件,__name__内置变量的值为: ',__name__)
```

```
#logger.py
def get_logger():
    print('打印日志! ')

print('这是logger模块文件,__name__内置变量的值
为: ',__name__)

#main.py: 作为执行文件
import cal #导入了cal模块
import logger #导入和logger模块

print('main文件的__name__这个内置变量
为: ',__name__)
```

结果为:

```
这是cal模块文件,__name__内置变量的值为:  cal
这是logger模块文件,__name__内置变量的值为:  logger
main文件的__name__这个内置变量为:  __main__
```

通过结果发现`__name__`只有在执行模块中打印`__main__`,在其他导入模块中打印各自模块的名称。

所以, `__name__`可以有以下作用:

- 利用`__name__=="__main__"`声明程序入口。
- 可以对导入的模块进行功能测试

包

什么是包

当一个项目中模块越来越多，维护和开发不是那么高效的时候，我们可以引入一种比模块更高级语法：包。

包是对相关功能的模块`py`文件的组织方式。

包可以理解为文件夹，更确切的说，是一个包含`__init__`文件的文件夹。

导入包的语法

1. `import 包名[.模块名 [as 别名]]`
2. `from 包名 import 模块名 [as 别名]`
3. `from 包名.模块名 import 成员名 [as 别名]`

案例：将上面案例中的`cal .py`文件放到`utils`包中管理，`logger.py`放到`logger`包中管理。

```
-- demo
   main.py  #执行文件
   -- m_log #包
      __init__.py
      logger.py #存储在logger包中的一个模块
```

```
from m_log import logger
logger.get_logger() #调用了logger模块中的
get_logger函数
```

常见模块

time模块

```
# <1> 时间戳

>>> import time
>>> time.time()
1493136727.099066

# <2> 时间字符串
>>> time.strftime("%Y-%m-%d %X") #%Y:年 %m: 月
%d: 天 %X:时分秒
'2017-04-26 00:32:18'

#<3> 程序暂定固定的时间
import time
print('正在下载数据.....')
time.sleep(2) #程序暂定n秒
print('下载成功! ')
```

小结：时间戳是计算机能够识别的时间；时间字符串是人能够看懂的时间；元组则是用来操作时间的

```
import time
#计算一组程序执行的耗时
start = time.time()
#测试代码
num = 0
for i in range(10000000):
    num += 1
print(num)
################################
print('总耗时：',time.time()-start)
```

random模块（了解）

```
>>> import random
>>> random.random()       # 大于0且小于1之间的小数
0.7664338663654585
>>> random.randint(1,5)  # 大于等于1且小于等于5之间的整数
2
>>> random.randrange(1,3) # 大于等于1且小于3之间的整数
1
```

```
>>> random.choice([1,'23',[4,5]])  # 返回列表中的随机一个元素
1
>>> random.sample([1,'23',[4,5]],2) # 列表元素任意2个随机组合
[[4, 5], '23']
>>> random.uniform(1,3) #大于1小于3的小数
1.6270147180533838
>>> item=[1,3,5,7,9]
>>> random.shuffle(item) # 直接将原来的列表元素打乱次序，不会返回一个新列表
>>> item
[5, 1, 3, 7, 9]
```

os模块(了解)

os模块是与操作系统交互的一个接口

```
import os

os.getcwd() # 获取当前工作目录，即当前python脚本工作的目录路径
os.chdir("dirname")  # 改变当前脚本工作目录；相当于shell下cd
os.curdir   # 返回当前目录: ('.')
os.pardir  # 获取当前目录的父目录字符串名: ('..')
```

```
os.makedirs('dirname1/dirname2')    # ***可生成多层递归目录
os.removedirs('dirname1')     # ***若目录为空，则删除，并递归到上一级目录，如若也为空，则删除，依此类推
os.mkdir('dirname')     #*** 生成单级目录；相当于shell中mkdir dirname
os.rmdir('dirname')     # *** 删除单级空目录，若目录不为空则无法删除，报错；相当于shell中rmdir dirname
os.listdir('dirname')     # ***列出指定目录下的所有文件和子目录，包括隐藏文件，并以列表方式打印
os.remove()  #*** 删除一个文件
os.rename("oldname","newname")  #*** 重命名文件/目录
os.stat('path/filename') #  获取文件/目录信息
os.sep     # 输出操作系统特定的路径分隔符，win下为"\\",Linux下为"/"
os.linesep     # 输出当前平台使用的行终止符，win下为"\t\n",Linux下为"\n"
os.pathsep     # 输出用于分割文件路径的字符串 win下为;,Linux下为:
os.name     # 输出字符串指示当前使用平台。win->'nt'; Linux->'posix'
os.system("bash command")  # 运行shell命令，直接显示
os.environ  # 获取系统环境变量
os.path.abspath(path)  # ***返回path规范化的绝对路径
```

```
os.path.split(path)  # 将path分割成目录和文件名二元组返回
os.path.dirname(path)  # 返回path的目录。其实就是os.path.split(path)的第一个元素
os.path.basename(path) #  返回path最后的文件名。如何path以 / 或\结尾，那么就会返回空值。即os.path.split(path)的第二个元素
os.path.exists(path) #  ***如果path存在，返回True；如果path不存在，返回False
os.path.isabs(path)  # 如果path是绝对路径，返回True
os.path.isfile(path)  # ***如果path是一个存在的文件，返回True。否则返回False
os.path.isdir(path)  # ***如果path是一个存在的目录，则返回True。否则返回False
os.path.join(path1[, path2[, ...]])  # 将多个路径组合后返回，第一个绝对路径之前的参数将被忽略
os.path.getatime(path)  # 返回path所指向的文件或者目录的最后访问时间
os.path.getmtime(path)  # 返回path所指向的文件或者目录的最后修改时间
os.path.getsize(path) # ***返回path的大小
```

序列化模块：json（重点）

> 序列化： 将python中的字典，列表对象转换成指定形式字符串

> 反序列化：将指定格式的字符串转换成字典，列表对象

- 基本使用

```
import json
dic = {
    'hobby':['football','pingpang','smoke'],
    'age':20,
    'score':97.6,
    'name':'zhangsan'
}
#序列化：将字典对象转换成了json格式的字符串
r = json.dumps(dic)
print(r)
```

```
import json

str = '{"hobby": ["football", "pingpang",
"smoke"], "age": 20, "score": 97.6, "name":
"zhangsan"}'
#反序列化：将字符串转换成了字典对象
dic = json.loads(str)
print(dic)
```

```
#持久化存储字典
import json
dic = {
    'hobby':['football','pingpang','smoke'],
    'age':20,
    'score':97.6,
    'name':'zhangsan'
}
fp = open('./dic.json','a')
#dump首先将dic字典进行序列化，然后将序列化后的结果写入到了fp表示的文件中
json.dump(dic,fp)
fp.close()
```

```
import json
fp = open('./dic.json','r')
#load将文件中的字符串数据进行读取，且将其转换成字典类型
dic = json.load(fp)
print(dic)
fp.close()
```