

模块和包

什么是模块

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，我们把很多代码按功能分组，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。在Python中，一个.py文件就可以称之为一个模块（Module）。

使用模块有什么好处？

1. 最大的好处是大大提高了代码的可维护性。其次，编写代码不必从零开始。当一个模块编写完毕，就可以被其他地方引用。我们在编写程序的时候，也经常引用其他模块，包括Python内置的模块和来自第三方的模块。
2. 使用模块还可以避免函数名和变量名冲突。每个模块有独立的命名空间，因此相同名字的函数和变量完全可以分别存在不同的模块中，所以，我们自己在编写模块时，不必考虑名字会与其他模块冲突
3. 拿来主义，提升开发效率
同样的原理，我们也可以下载别人写好的模块然后导入到自己的项目中使用，这种拿来主义，可以极大地提升我们的开发效率，避免重复造轮子。

模块分类

- 内置标准模块（又称标准库）执行help('modules')查看所有python自带模块列表
- 第三方开源模块，可通过pip install 模块名 联网安装
- 自定义模块

模块导入&调用

结合使用 from 和 import

```
1 import module_a #导入
2 from module import xx # 导入某个模块下的某个方法 or 子模块
3 from module.xx.xx import xx as rename #导入后一个方法后重命名
4 from module.xx.xx import * #导入一个模块下的所有方法，不建议使用
5 module_a.xxx #调用
```

注意：模块一旦被调用，即相当于执行了另外一个py文件里的代码

import

- 1 模块可以包含可执行的语句和函数的定义，这些语句的目的是初始化模块，它们只在模块名第一次遇到导入import语句时才执行（import语句是可以在程序中的任意位置使用的，且针对同一个模块很import多次，为了防止你重复导入，python的优化手段是：第一次导入后就将模块名加载到内存了，后续的import语句仅是对已经加载到内存中的模块对象增加了一次引用，不会重新执行模块内的语句），如下

```
1 import spam #只在第一次导入时才执行spam.py内代码,此处的显式效果是只打印一次'from the
spam.py',当然其他的顶级代码也都被执行了,只不过没有显示效果
```

代码示例:

```
1 import tbjx
2 import tbjx
3 import tbjx
4 import tbjx
5 import tbjx
6 执行结果: 只是打印一次:
7 # from the tbjx.py
```

第一次导入模块执行事件

1. 执行tbjx.py
2. 为源文件(tbjx模块)创建新的名称空间, 将tbjx.py运行过程中产生的名字都丢到tbjx的名称空间中。
3. 在当前文件中产生的有一个名字tbjx, 该名字指向2中产生的名称空间

ps: 重复导入会直接引用内存中已经加载好的结果

被导入模块有独立的名称空间

每个模块都是一个独立的名称空间, 定义在这个模块中的函数, 把这个模块的名称空间当做全局名称空间, 这样我们在编写自己的模块时, 就不用担心我们定义在自己模块中全局变量会在被导入时, 与使用者的全局变量冲突

引用:

foo.py

```
1 print('模块foo==>')
2
3 x=1
4
5 def get():
6     print(x)
7
8 def change():
9     global x
10    x=0
```

1. 模块名.名字, 是指名道姓地问某一个模块要名字对应的值

```
1 import foo
2 x=11111111111111
3 print(x)
4 print(foo.x)
```

2. 无论是查看还是修改操作的都是模块本身, 与调用位置无关

```

1 import foo
2
3 x=3333333333
4 foo.get()
5
6 foo.change()
7 print(x)
8
9 print(foo.x)
10 foo.get()

```

3. 可以以逗号为分隔符在一行导入多个模块

```

1 import os,sys,json    这样写可以但是不推荐
2 推荐写法
3 import os
4 import sys
5 import json

```

示例:

tbjx.py

```

1 name = "太白金星"
2 print("from the tbjx.py")
3
4
5 def read1():
6     print("tbjx模块: %s" % name)
7
8
9 def change():
10    global name
11    name = "barry"
12

```

当前是meet.py

```

1 import tbjx
2 name = '小王'
3 print(name)
4 print(tbjx.name)
5 '''
6 from the tbjx.py
7 小王
8 太白金星
9 '''
10 def read1():
11     print(666)
12 tbjx.read1()
13 '''
14 from the tbjx.py
15 tbjx模块: 太白金星
16 '''

```

```

17 name = '日天'
18 tbjx.change()
19 print(name)
20 print(tbjx.name)
21 '''
22 from the tbjx.py
23 日天
24 barry
25 '''

```

为模块起别名

别名其实就是一个绰号,好处可以将很长的模块名改成很短,方便使用.

```

1 import tbjx as t
2 t.read1()

```

有利于代码的扩展和优化

```

1 #mysql.py
2 def sqlparse():
3     print('from mysql sqlparse')
4 #oracle.py
5 def sqlparse():
6     print('from oracle sqlparse')
7 #test.py
8 db_type=input('>>: ')
9 if db_type == 'mysql':
10     import mysql as db
11 elif db_type == 'oracle':
12     import oracle as db
13 db.sqlparse()

```

1.6 from ... import ...

from...import...使用

```

1 from tbjx import name, read1
2 print(name)
3 read1()
4 '''
5 执行结果:
6 from the tbjx.py
7 太白金星
8 tbjx模块: 太白金星
9 '''

```

from...import... 与import对比

唯一的区别就是：使用from...import...则是将模块中的名字直接导入到当前的名称空间中，所以在当前名称空间中，直接使用名字就可以了、无需加前缀：tbjx.

from...import...的方式有好处也有坏处

好处：使用起来方便了

坏处：容易与当前执行文件中的名字冲突

示例演示：

执行文件有与模块同名的变量或者函数名，会有覆盖效果。

```
1 name = '程序猿'
2 from tbjx import name, read1, read2
3 print(name)
4 '''
5 执行结果:
6 太白金星
7 '''
8 -----
9 from tbjx import name, read1, read2
10 name = '程序猿'
11 print(name)
12 '''
13 执行结果:
14 程序猿
15 '''
16 -----
17 def read1():
18     print(666)
19 from tbjx import name, read1, read2
20 read1()
21 '''
22 执行结果:
23 tbjx模块: 太白金星
24 '''
25 -----
26 from tbjx import name, read1, read2
27 def read1():
28     print(666)
29 read1()
30 '''
31 执行结果:
32 tbjx模块: 666
33 '''
```

当前位置直接使用read1和read2就好了 执行时，仍然以tbjx.py文件全局名称空间

tbjx.py

```
1 name = "太白金星"
2 print("from the tbjx.py")
3
4
5 def read1():
6     print("tbjx->read1->name = '%s'" % name)
```

```

7
8
9 def change():
10     global name
11     name = "barry"
12
13 def read2():
14     print("tbjx->read2 calling read")
15     read1()

```

```

1 #测试一：导入的函数read1，执行时仍然回到tbjx.py中寻找全局变量name
2 #test.py
3 from tbjx import read1
4 name = '小王'
5 read1()
6 '''
7     执行结果：
8     from the spam.py
9     tbjx->read1->name = '太白金星'
10 '''
11 #测试二：导入的函数read2，执行时需要调用read1()，仍然回到tbjx.py中找read1()
12 #test.py
13 from tbjx import read2
14 def read1():
15     print('=====')
16     read2()
17 '''
18     执行结果：
19     from the tbjx.py
20     tbjx->read2 calling read
21     tbjx->read1->tbjx 'barry'
22 '''

```

也支持as

```

1 from tbjx import read1 as read
2 read()

```

一行导入多个

```

1 from tbjx import read1,read2,name

```

from ... import *

```

1 from tbjx import * #把tbjx中所有的不是以下划线(_)开头的名字都导入到当前位置
2 #大部分情况下我们的python程序不应该使用这种导入方式，因为*你不知道你导入什么名字，很有可能会覆盖掉你之前已经定义的名字。而且可读性极其的差，在交互式环境中导入时没有问题

```

“单下划线”开始的成员变量叫做保护变量，可看作为“私有的”，在模块和类外不可以使用

可以使用all来控制*（用来发布新版本），在tbjx.py中新增一行

```
1 | __all__=['money','read1'] #这样在另外一个文件中用from tbjx import *就能导入列
   | 表中规定的两个名字
```

python文件的两种用途与区别

```
1 | 一个py文件有几种用途？
2 |     执行py文件与导入py文件的区别是什么？
3 |     1、被当成程序运行（执行文件）
4 |
5 |     2、被当做模块导入（导入文件）
6 |         执行文件在运行的时候会产生执行文件的名称空间，并将程序运行过程中产生的名字存放到执
   | 行文件的名称空间。
7 |         如果执行文件中有导入模块的操作，import foo（模块名）会将模块名存放到执行文件的名称空
   | 间，执行文件的模块名指向的就是被导入模块的名称空间。
8 |         导入模块会执行被导入的模块文件，产生被导入模块的名称空间，执行过程中产生的名字存放到被
   | 导入模块的名称空间，并存放到内存的内置模块中提供给执行文件使用，
9 |         当执行文件运行完毕后，导入模块的文件才会跟着结束。
```

区分py文件的两种用途

一个Python文件有两种用途，一种被当主程序/脚本执行，另一种被当模块导入，为了区别同一个文件的不同用途，

每个py文件都内置了**name**变量，该变量在py文件被当做脚本执行时赋值为“**main**”，在py文件被当做模块导入时赋值为模块名

可以在导入模块写内容测试一下：

```
1 |
2 | if __name__ == '__main__':
3 |     print('我被执行了')  导入模块自己执行会打印
4 | else:
5 |     print('我被导入了')  执行文件导入该模块会被打印
```

模块的搜索路径

模块的查找顺序是：内存中已经加载的模块->内置模块->sys.path路径中包含的模块

```
1 | #模块的查找顺序
2 | 1、在第一次导入某个模块时（比如tbjx），会先检查该模块是否已经被加载到内存中（当前执行文件
   | 的名称空间对应的内存），如果有则直接引用
3 | ps: python解释器在启动时会自动加载一些模块到内存中，可以使用sys.modules查看
4 | 2、如果没有，解释器则会查找同名的内置模块
5 | 3、如果还没有找到就从sys.path给出的目录列表中依次寻找tbjx.py文件。
6 | #需要特别注意的是：我们自定义的模块名不应该与系统内置模块重名。虽然每次都说，但是仍然会有人
   | 不停的犯错。
7 | #在初始化后，python程序可以修改sys.path,路径放到前面的优先于标准库被加载。
8 | >>> import sys
9 | >>> sys.path
10 | >>> sys.path.append('/a/b/c/d')
11 | >>> sys.path.insert(0, '/x/y/z') #排在前的目录，优先被搜索
12 | 注意：搜索时按照sys.path中从左到右的顺序查找，位于前的优先被查找，sys.path中还可能包
   | 含.zip归档文件和.egg文件，python会把.zip归档文件当成一个目录去处理，
```

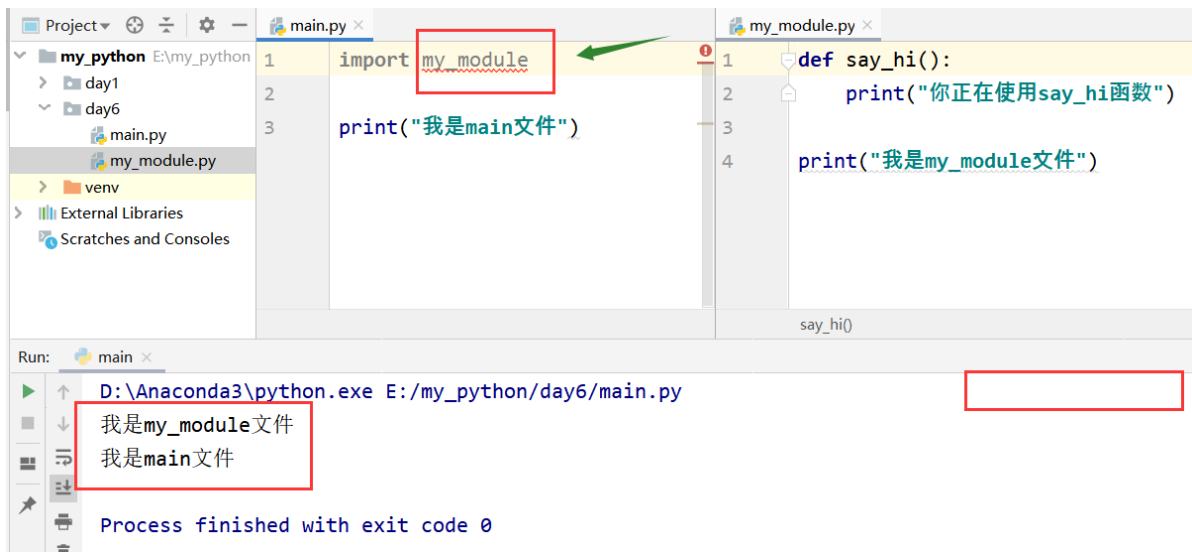
```

13 #首先制作归档文件: zip module.zip foo.py bar.py
14 import sys
15 sys.path.append('module.zip')
16 # sys.path获取到的是一个列表,所以我们可以执行列表的操作
17 import foo
18 import bar
19 #也可以使用zip中目录结构的具体位置
20 sys.path.append('module.zip/lib/python')
21 #windows下的路径不加r开头,会语法错误,r表示字符的原始含义
22 sys.path.insert(0,r'C:\Users\Administrator\PycharmProjects\a')

```

自定义模块

这个最简单, 创建一个.py文件, 就可以称之为模块, 就可以在另外一个程序里导入



模块的查找路径

有没有发现, 自己写的模块只能在当前路径下的程序里才能导入, 换一个目录再导入自己的模块就报错说找不到了, 这是为什么?

这与导入模块的查找路径有关

```

1 import sys
2 print(sys.path)

```

输出 (注意不同的电脑可能输出的不太一样)

```

1 ['E:\\my_python\\day6', 'E:\\my_python', 'D:\\Anaconda3\\python37.zip',
  'D:\\Anaconda3\\DLLs', 'D:\\Anaconda3\\lib', 'D:\\Anaconda3',
  'D:\\Anaconda3\\lib\\site-packages', 'D:\\Anaconda3\\lib\\site-
  packages\\win32', 'D:\\Anaconda3\\lib\\site-packages\\win32\\lib',
  'D:\\Anaconda3\\lib\\site-packages\\Pythonwin']

```

你导入一个模块时, Python解释器会按照上面列表顺序去依次到每个目录下去匹配你要导入的模块名, 只要在一个目录下匹配到了该模块名, 就立刻导入, 不再继续往后找。

注意列表第一个元素为空, 即代表当前目录, 所以你自己定义的模块在当前目录会被优先导入。

我们自己创建的模块若想在任何地方都能调用，那就得确保你的模块文件至少在模块路径的查找列表中。

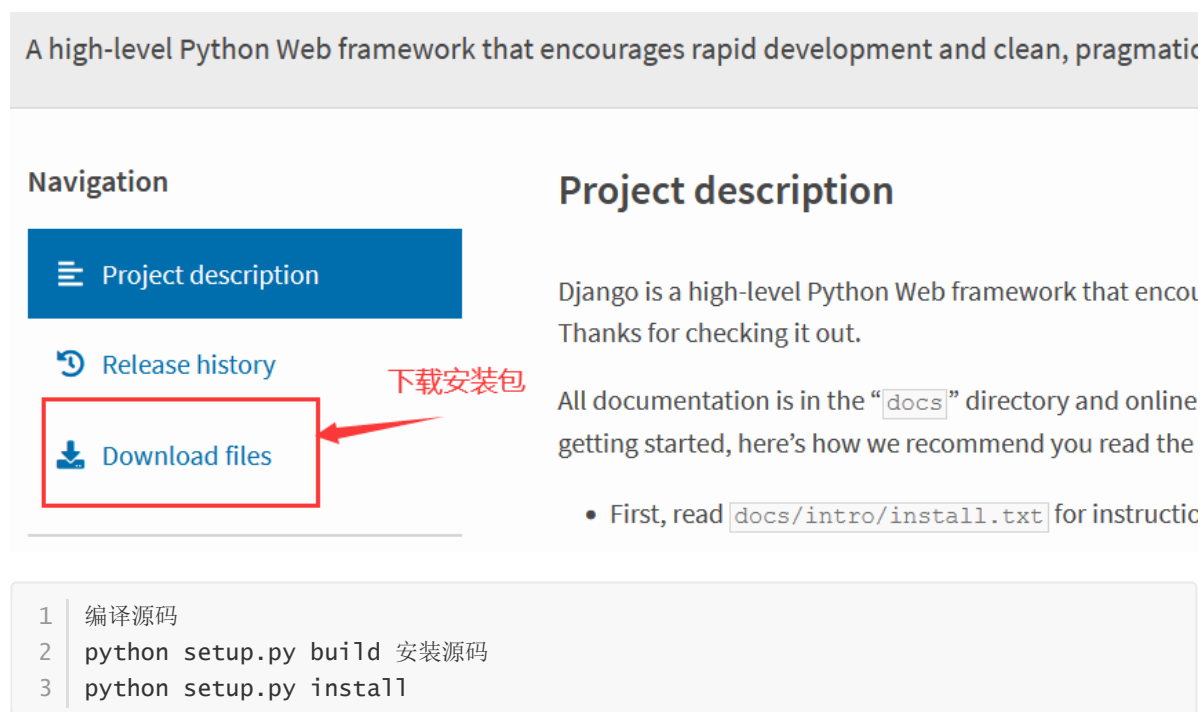
我们一般把自己写的模块放在一个带有“site-packages”字样的目录里，我们从网上下载安装的各种第三方的模块一般都放在这个目录。

第3方开源模块安装

<https://pypi.python.org/pypi> 是python的开源模块库，截止2020年5.26日，已经收录了326,082个来自全世界python开发者贡献的模块，几乎涵盖了你想用python做的任何事情。事实上每个python开发者，只要注册一个账号就可以往这个平台上传你自己的模块，这样全世界的开发者都可以容易的下载并使用你的模块。

那如何从这个平台上下载代码呢？

1.直接在上面这个页面上点download,下载后，解压并进入目录，执行以下命令完成安装



A high-level Python Web framework that encourages rapid development and clean, pragmatic design.

Navigation

- Project description
- Release history
- Download files** (highlighted with a red box and a red arrow pointing to it, with the text '下载安装包' next to the arrow)

Project description

Django is a high-level Python Web framework that encourages rapid development and clean, pragmatic design. Thanks for checking it out.

All documentation is in the “docs” directory and online getting started, here’s how we recommend you read the

- First, read `docs/intro/install.txt` for instructions.

```
1 | 编译源码
2 | python setup.py build 安装源码
3 | python setup.py install
```

2.直接通过pip安装

```
1 | pip3 install requests #requests 是模块名
```

pip命令会自动下载模块包并完成安装。

软件一般会被自动安装到你python安装目录的这个子目录里

```
1 | /your_python_install_path/3.6/lib/python3.6/site-packages
```

pip命令默认会连接在国外的python官方服务器下载，速度比较慢，你还可以使用国内的豆瓣源，数据会定期同步国外官网，速度快好多

```
1 | pip install pandas -i https://mirrors.aliyun.com/pypi/simple
```

国内镜像

```
1 豆瓣
2     https://pypi.douban.com/simple
3 阿里云
4     https://mirrors.aliyun.com/pypi/simple
5 清华大学
6     https://pypi.tuna.tsinghua.edu.cn/simple
7 中国科技大学
8     https://pypi.mirrors.ustc.edu.cn/simple
9
```

pip 说明:

pip 最常用命令

显示版本和路径

```
1 | pip --version
```

获取帮助

```
1 | pip --help
```

升级 pip

```
1 | pip install -U pip
```

如果这个升级命令出现问题，可以使用以下命令：

```
1 | sudo easy_install --upgrade pip
```

安装包

```
1 | pip install SomePackage           # 最新版本
2 | pip install SomePackage==1.0.4    # 指定版本
3 | pip install 'SomePackage>=1.0.4'  # 最小版本
```

比如我要安装 Django。用以下的一条命令就可以，方便快捷。

```
1 | pip install Django==1.11.*
```

卸载包

```
1 | pip uninstall SomePackage
```

搜索包

```
1 | pip search SomePackage
```

显示安装包信息

```
1 | pip show
```

列出已安装的包

```
1 | pip list
```

注意事项

如果 Python2 和 Python3 同时有 pip，则使用方法如下：

Python2:

```
1 | python2 -m pip install xxx
```

Python3:

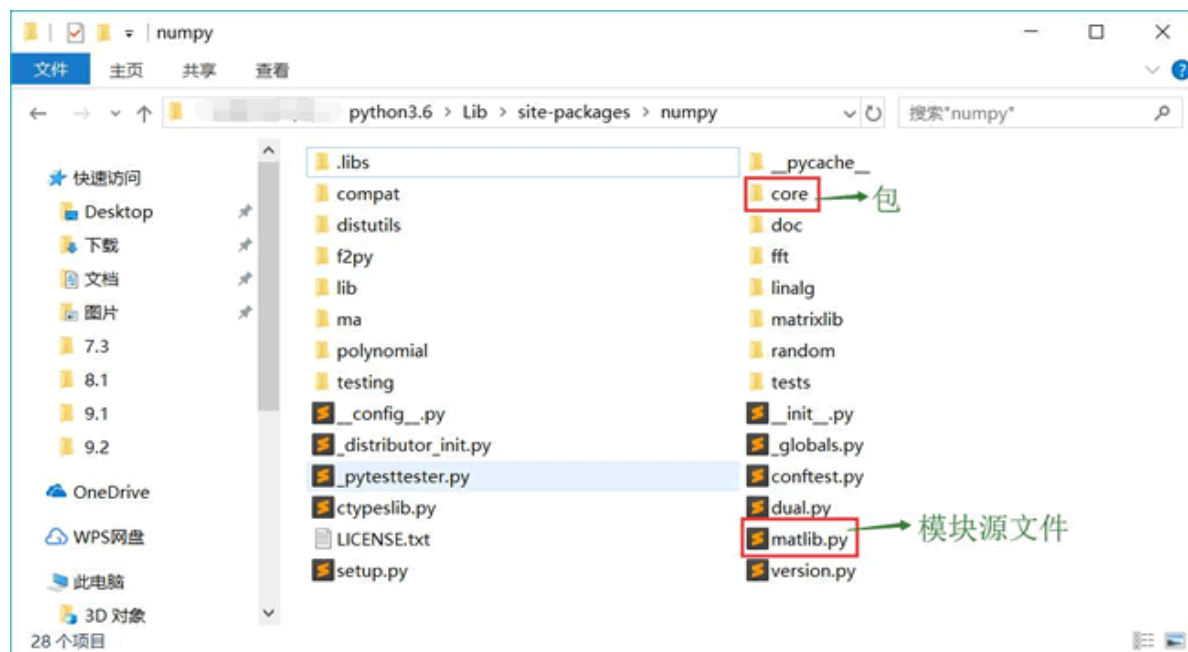
```
1 | python3 -m pip install xxx
```

什么是包 (package)

为了更好地管理多个模块源文件，Python 提供了包的概念。那么问题来了，什么是包呢？

从物理上看，包就是一个文件夹，在该文件夹下包含了一个 `__init__.py` 文件，该文件夹可用于包含多个模块源文件；从逻辑上看，包的本质依然是模块。

根据上面介绍可以得到一个推论，包的作用是包含多个模块，但包的本质依然是模块，因此包也可用于包含包。典型地，当我们为 Python 安装了 numpy 模块之后，可以在 Python 安装目录的 `Lib\site-packages` 目录下找到一个 numpy 文件夹，它就是前面安装的 numpy 模块（其实是一个包）。该文件夹的内容如图 1 所示：



从图 1 可以看出，在 numpy 包（也是模块）下既包含了 `matlib.py` 等模块源文件，也包含了 `core` 等子包（也是模块）。这正对应了我们刚刚介绍的：包的本质依然是模块，因此包又可以包含包。

定义包

掌握了包是什么之后，接下来学习如何定义包。定义包更简单，主要有两步：

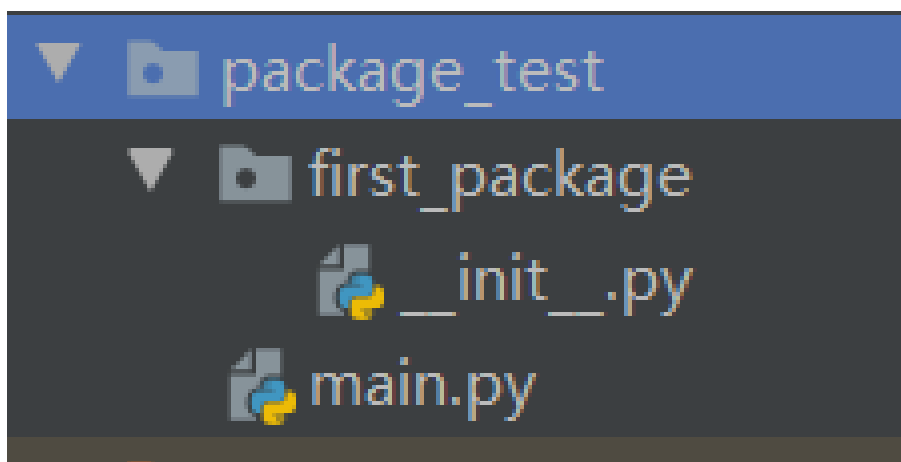
1. 创建一个文件夹，该文件夹的名字就是该包的包名。
2. 在该文件夹内添加一个 `__init__.py` 文件即可。

下面定义一个非常简单的包。先新建一个 `first_package` 文件夹，然后在该文件夹中添加一个 `__init__.py` 文件，该文件内容如下：

```
1  '''
2  这是学习包的第一个示例
3  '''
4  print('this is first_package')
```

上面的 Python 源文件非常简单，该文件开始部分的字符串是该包的说明文档，接下来是一条简单的输出语句。

目录结构：



下面通过如下程序来使用该包：

main.py

```
1  # 导入first_package包（模块）
2  import first_package
3  print('=====')
4  print(first_package.__doc__)
5  print(type(first_package))
6  print(first_package)
```

再次强调，包的本质就是模块，因此导入包和导入模块的语法完全相同。因此，上面程序中第 2 行代码导入了 `first_package` 包。程序最后三行代码输出了包的说明文档、包的类型和包本身。

运行该程序，可以看到如下输出结果：

```
1  this is first_package
2  =====
3
4  这是学习包的第一个示例
5
6  <class 'module'>
7  <module 'first_package' from
   'E:\\my_python\\myself\\first_package\\__init__.py'>
```

从上面的输出结果可以看出，在导入 `first_package` 包时，程序执行了该包所对应的文件夹下的 `init.py`；从倒数第二行输出可以看到，包的本质就是模块；从最后一行输出可以看到，使用 `import first_package` 导入包的本质就是加载并执行该包下的 `init.py` 文件，然后将整个文件内容赋值给与包同名的变量，该变量的类型是 `module`。

与模块类似的是，包被导入之后，会在包目录下生成一个 `pycache` 文件夹

由于导入包就相当于导入该包下的 `init.py` 文件，因此我们完全可以在 `init.py` 文件中定义变量、函数、类等程序单元，但实际上往往并不会这么做。想一想原因是什么？包的主要作用是包含多个模块，因此 `init.py` 文件的主要作用就是导入该包内的其他模块。

下面再定义一个更加复杂的包，在该包下将会包含多个模块，并使用 `init.py` 文件来加载这些模块。

新建一个 `fk_package` 包，并在该包下包含二个模块文件：

- `print_shape.py`
- `arithmetic_chart.py`

`fk_package` 的文件结构如下：

```
fk_package
├──arithmetic_chart.py
├──print_shape.py
└──init.py
```

其中，`arithmetic_chart.py` 模块文件的内容如下：

```
1 def print_multiple_chart(n):
2     '打印乘法口诀表的函数'
3     for i in range(n):
4         for j in range(i + 1):
5             print('%d * %d = %2d' % ((j + 1), (i + 1), (j + 1) * (i + 1)),
6               end=' ')
7         print('')
```

上面模块文件中定义了一个打印乘法口诀表的函数。

`print_shape.py` 模块文件的内容如下：

```
1 def print_blank_triangle(n):
2     '使用星号打印一个空心的三角形'
3     if n <= 0:
4         raise ValueError('n必须大于0')
5     for i in range(n):
6         print(' ' * (n - i - 1), end='')
7         print('*', end='')
8         if i != n - 1:
9             print(' ' * (2 * i - 1), end='')
10        else:
11            print('*' * (2 * i - 1), end='')
12        if i != 0:
13            print('*')
14        else:
15            print('')
```

`tk_package` 包下的 `__init__.py` 文件暂时为空，不用编写任何内容。

上面三个模块文件都位于 `fk_package` 包下，总共提供了两个函数和一个类。这意味着 `fk_package` 包（也是模块）总共包含 `arithmetic_chart`和 `print_shape` 两个模块。在这种情况下，这两个模块就相当于 `fk_package` 包的成员。

导入包内成员

如果需要使用 `arithmetic_chart` 和 `print_shape` 这两个模块，则可以在程序中执行如下导入代码：

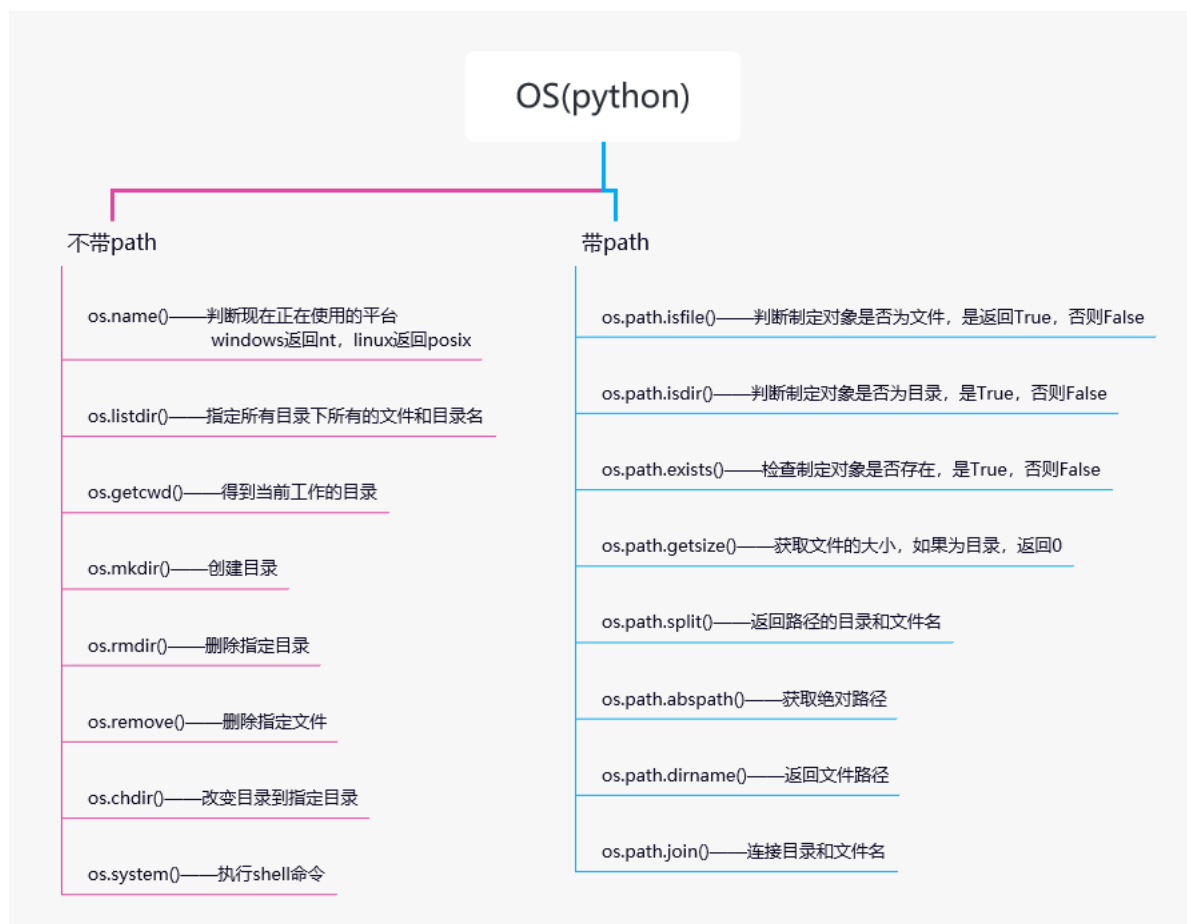
`main.py`

```
1 # 实际上就是导入fk_package包（模块）中的arithmetic_chart.py文件
2 from fk_package import arithmetic_chart
3 # 实际上就是导入fk_package包（模块）中的print_shape.py文件
4 from fk_package import print_shape
5 # 通过arithmetic_chart模块名调用内部的函数
6 arithmetic_chart.print_multiple_chart(5)
7 # 通过print_shape模块名调用内部的函数
8 print_shape.print_blank_triangle(6)
```

几个常用Python模块

OS模块

`os` 模块提供了很多允许你的程序与操作系统直接交互的功能



举例中的目录形式如下所示：

```
1 In [36]: pwd
2 Out[36]: '/home/python/Desktop/code'
3 In [37]: ls
4 hello.py hello.txt test.py 文件夹01/ 文件夹02/ 文件夹03/
```

1.当前路径及路径下的文件

`os.getcwd()`: 查看当前所在路径

`os.listdir(path)`:列举目录下的所有文件。返回的是列表类型。

```
1 >>> import os
2 >>> os.getcwd()
3 >>> '/home/python/Desktop/code'
4 >>> os.listdir(os.getcwd())
5 >>> ['文件夹01', '文件夹03', '文件夹02', 'test.py', '.idea', 'hello.txt', >>>
'hello.py']
```

2.绝对路径

`os.path.abspath(path)`:返回path的绝对路径

```
1 In [4]: os.path.abspath('.')
2 Out[4]: '/home/python/Desktop/code'
3 In [5]: os.path.abspath '..'
4 Out[5]: '/home/python/Desktop'
```

3.查看路径的文件夹部分和文件名部分

`os.path.split(path)`:将路径分解为(文件夹,文件名)，返回的是元组类型。可以看出，若路径字符串最后一个字符是,则只有文件夹部分有值；若路径字符串中均无,则只有文件名部分有值。若路径字符串有\, 且不在最后，则文件夹和文件名均有值。且返回的文件夹的结果不包含。

```
1 >>> os.path.split('.')
2 >>> ('', '.')
3 >>> os.path.split('/home')
4 >>> ('/', 'home')
5 >>> os.path.split('/home/Desktop')
6 >>> ('/home', 'Desktop')
7 >>> os.path.split('/home/Desktop/code')
8 >>> ('/home/Desktop', 'code')
9 >>> os.path.split('/home/Desktop/code/')
10 >>> ('/home/Desktop/code', '')
```

`os.path.join(path1,path2,...)`:将path进行组合，若其中有绝对路径，则之前的path将被删除。

```
1 >>> os.path.join('/home', 'Desktop')
2 >>> '/home/Desktop'
3 >>> os.path.join('/home/Desktop', 'code')
4 >>> '/home/Desktop/code'
```

`os.path.dirname(path)`:返回path中的文件夹部分，结果不包含"

```
1 >>> os.path.dirname(os.getcwd())
2 >>> '/home/python/Desktop'
```

`os.path.basename(path)`:返回path中的文件名。

```
1 >>> os.path.basename(os.getcwd())
2 >>> 'code'
3 >>> os.path.basename('.')
4 >>> '.'
5 >>> os.path.basename('/home/Desktop/code')
6 >>> 'code'
7 >>> os.path.basename('/home/Desktop/code/')
8 >>> ''
9 >>> os.path.basename('/home/Desktop/code/hello.txt')
10 >>> 'hello.txt'
```

4.查看文件大小

`os.path.getsize(path)`:文件或文件夹的大小

```
1 >>> os.getcwd()
2 >>> '/home/python/Desktop/code'
3 >>> os.path.getsize('/home/python/Desktop/code')
4 >>> 4096
5 >>> os.path.getsize('/home/python/Desktop/code/hello.txt')
6 >>> 61
```

5.查看文件是否存在

`os.path.exists(path)`:文件或文件夹是否存在，返回True 或 False。

```
1 >>> os.path.exists('/home/python/Desktop/code/hello.txt')
2 >>> True
3 >>> os.path.exists('/home/python/Desktop/code/hehe.txt')
4 >>> False
```

实例：

输出D盘下的所有文件：

time 模块

在平常的代码中，我们常常需要与时间打交道。在Python中，与时间处理有关的模块就包括：time，datetime

我们写程序时对时间的处理可以归为以下3种：

时间的显示，在屏幕显示、记录日志等 "2022-03-04"

时间的转换，比如把字符串格式的日期转成Python中的日期类型

时间的运算，计算两个日期间的差值等

在Python中，通常有这几种方式来表示时间：

1. 时间戳（timestamp），表示的是从1970年1月1日00:00:00开始按秒计算的偏移量。例子：
1554864776.161901

2. 格式化的时间字符串，比如“2020-10-03 17:54”

元组 (struct_time) 共九个元素。由于Python的time模块实现主要调用C库，所以各个平台可能有所不同， mac上：time.struct_time(tm_year=2020, tm_mon=4, tm_mday=10, tm_hour=2, tm_min=53, tm_sec=15, tm_wday=2, tm_yday=100, tm_isdst=0)

序号	属性	值
0	tm_year	2008
1	tm_mon	1 到 12
2	tm_mday	1 到 31
3	tm_hour	0 到 23
4	tm_min	0 到 59
5	tm_sec	0 到 61 (60或61 是闰秒)
6	tm_wday	0到6 (0是周一)
7	tm_yday	1 到 366(儒略历)
8	tm_isdst	-1, 0, 1, -1是决定是否为夏令时的旗帜

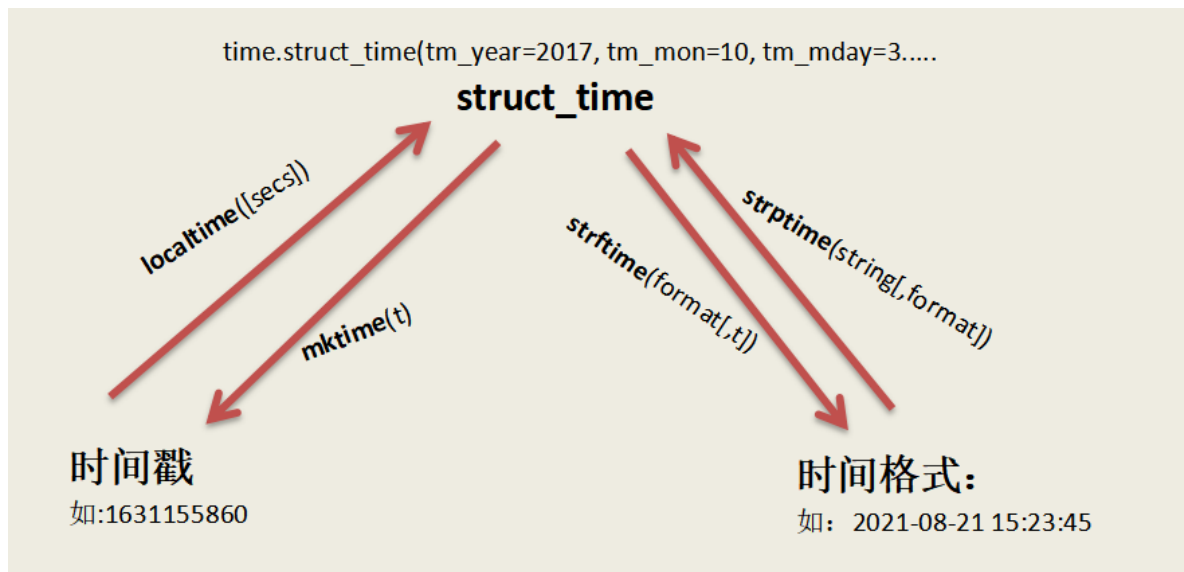
UTC**时间

UTC (Coordinated Universal Time，世界协调时) 亦即格林威治天文时间，世界标准时间。在中国为UTC+8， 又称东8区。 DST (Daylight Saving Time) 即夏令时。



time模块的常用方法

```
1 #常用方法
2 1.time.sleep(secs)
3 (线程)推迟指定的时间运行。单位为秒。
4 2.time.time()
5 获取当前时间戳
```



- **time.localtime([secs])**: 将一个时间戳转换为当前时区的`struct_time`。若`secs`参数未提供, 则以当前时间为准。
- **time.gmtime([secs])**: 和`localtime()`方法类似, `gmtime()`方法是 将一个时间戳转换为UTC时区 (0时区) 的`struct_time`。
- **time.time()**: 返回当前时间的时间戳。
- **time.mktime(t)**: 将一个`struct_time`转化为时间戳。
- **time.sleep(secs)**: 线程推迟指定的时间运行,单位为秒。
- **time.strftime(format[,t])***
作用: 将`struct_time`类型的时间转换为`format`参数指定格式的字符串。
参数:
 - `format`: 指定转换时间的字符串格式。
 - `t`: `struct_time`类型的时间, 如果不填默认为当前时间 (即`time.localtime()`返回的时间)
- **time.strptime(string[,format])**
作用: 根据格式解析表示时间的字符串。
参数:
 - `string`: 字符串类型的时间。
 - `format`: 提供字符串类型的时间的格式。

字符串转时间格式对应表

%y 两位数的年份表示 (00-99)

%Y 四位数的年份表示 (000-9999)

%m 月份 (01-12)

%d 月内中的一天 (0-31)

%H 24小时制小时数 (0-23)

%I 12小时制小时数 (01-12)

%M 分钟数 (00-59)

%S 秒 (00-59)

```
1 #结构化时间-->时间戳
2 #time.mktime(结构化时间)
3 >>>time_tuple = time.localtime(1500000000)
4 >>>time.mktime(time_tuple)
5 1500000000.0
6 #结构化时间-->字符串时间
7 #time.strftime("格式定义","结构化时间") 结构化时间参数若不传，则显示当前时间
8 >>>time.strftime("%Y-%m-%d %X")
9 '2017-07-24 14:55:36'
10 >>>time.strftime("%Y-%m-%d",time.localtime(1500000000))
11 '2017-07-14'
12 #字符串时间-->结构化时间
13 #time.strptime(时间字符串,字符串对应格式)
14 >>>time.strptime("2017-03-16", "%Y-%m-%d")
15 time.struct_time(tm_year=2017, tm_mon=3, tm_mday=16, tm_hour=0,
16 tm_min=0, tm_sec=0, tm_wday=3, tm_yday=75, tm_isdst=-1)
17 >>>time.strptime("07/24/2017", "%m/%d/%Y")
18 time.struct_time(tm_year=2017, tm_mon=7, tm_mday=24, tm_hour=0,
19 tm_min=0, tm_sec=0, tm_wday=0, tm_yday=205, tm_isdst=-1)
```

计算时间差:

```
1 import time
2 true_time=time.mktime(time.strptime('2017-09-11 08:30:00', '%Y-%m-%d
%H:%M:%S'))
3 time_now=time.mktime(time.strptime('2017-09-12 11:00:00', '%Y-%m-%d
%H:%M:%S'))
4 dif_time=time_now-true_time
5 struct_time=time.gmtime(dif_time)
6 print('过去了%d年%d月%d天%d小时%d分钟%d秒'%(struct_time.tm_year-
1970,struct_time.tm_mon-1,
7 struct_time.tm_mday-
1,struct_time.tm_hour,
8 struct_time.tm_min,struct_time.tm_sec))
```

datetime模块

获取当前日期和时间

```
1 from datetime import datetime
2 print(datetime.now())
3 '''
4 结果:2018-12-04 21:07:48.734886
5 '''
```

注意: `datetime` 是模块, `datetime` 模块还包含一个 `datetime` 的类, 通过 `from datetime import datetime` 导入的才是 `datetime` 这个类。

如果仅导入 `import datetime`, 则必须引用全名 `datetime.datetime`。

`datetime.now()` 返回当前日期和时间, 其类型是 `datetime`。

获取指定日期和时间

要指定某个日期和时间，我们直接用参数构造一个 `datetime`

```
1 from datetime import datetime
2 dt = datetime(2018,5,20,13,14)
3 print(dt)
4 '''
5 结果:2018-05-20 13:14:00
6 '''
```

datetime转换为timestamp

```
1 from datetime import datetime
2 dt = datetime.now()
3 new_timestamp = dt.timestamp()
4 print(new_timestamp)
5 '''
6 结果:1543931750.415896
7 '''
```

timestamp转换为datetime

```
1 import time
2 from datetime import datetime
3 new_timestamp = time.time()
4 print(datetime.fromtimestamp(new_timestamp))
```

str转换为datetime

很多时候，用户输入的日期和时间是字符串，要处理日期和时间，首先必须把str转换为datetime。转换方法是通过 `datetime.strptime()` 实现，需要一个日期和时间的格式化字符串：

```
1 from datetime import datetime
2 t = datetime.strptime('2018-4-1 00:00','%Y-%m-%d %H:%M')
3 print(t)
4 '''
5 结果: 2018-04-01 00:00:00
6 '''
```

datetime转换为str

如果已经有了datetime对象，要把它格式化为字符串显示给用户，就需要转换为str，转换方法是通过 `strftime()` 实现的，同样需要一个日期和时间的格式化字符串

```
1 from datetime import datetime
2 now = datetime.now()
3 print(now.strftime('%a, %b %d %H:%M'))
4 Mon, May 05 16:28
```

datetime加减

对日期和时间进行加减实际上就是把datetime往后或往前计算，得到新的datetime。加减可以直接用 `+` 和 `-` 运算符，不过需要导入 `timedelta` 这个类：

```
1 from datetime import datetime, timedelta
2 now = datetime.now()
3 now
4 datetime.datetime(2015, 5, 18, 16, 57, 3, 540997)
5 now + timedelta(hours=10)
6 datetime.datetime(2015, 5, 19, 2, 57, 3, 540997)
7 now - timedelta(days=1)
8 datetime.datetime(2015, 5, 17, 16, 57, 3, 540997)
9 now + timedelta(days=2, hours=12)
10 datetime.datetime(2015, 5, 21, 4, 57, 3, 540997)
```

可见，使用 `timedelta` 你可以很容易地算出前几天和后几天的时刻。

小结

`datetime` 表示的时间需要时区信息才能确定一个特定的时间，否则只能视为本地时间。

如果要存储 `datetime`，最佳方法是将其转换为timestamp再存储，因为timestamp的值与时区完全无关。

序列化

什么是序列化,回想一想我们可以将python的数据结构存储到文件中,这个很简单.但是要从文件中把列表在转换出来

并且能够实现列表的操作就是比较麻烦的,要是把字典从文件中在转换出来就是难上加难了.

序列化就可以将咱们说的这个搞定,并且还有一个很大的好处就是可以持久保存了,以前咱们定义第一个列表和字典当程序

运行完,在次运行的时候就会重新创建一份,使用了序列化后就不需要在重新创建了.我们来看看说的这么nb怎么用啊

json

首先我们需要使用import将json导入

```
1 import json
```

我们就用它二个功能 dumps loads

dumps

```
6 @File : oldboy.py
7 '''
8
9 import json
10
11 dic = {'key': 'value'}
12 f_dic = json.dumps(dic)
13 print(f_dic)
14 print(type(f_dic))
15
16
17
18
19
```

dumps是将python中的数据类型转成字符串

dumps(参数是要转换的数据类型)

loads

loads是将python的数据类型转换成字符串的内容在转换成之前的数据类型

loads(要转换成原数据类型的字符串)

```
6 @File : oldboy.py
7 '''
8
9 import json
10
11 dic = '{"key": "value"}'
12 s_dic = json.loads(dic)
13 print(s_dic)
14 print(type(s_dic))
15
16
17
18
```

sys模块

sys模块是与python解释器交互的一个接口

1	sys.argv	命令行参数List，第一个元素是程序本身路径
2	sys.exit(n)	退出程序，正常退出时exit(0),错误退出sys.exit(1)
3	sys.version	获取Python解释程序的版本信息
4	sys.path	返回模块的搜索路径，初始化时使用PYTHONPATH环境变量的值
5	sys.platform	返回操作系统平台名称

random模块

```
1 >>> random.randrange(1,10) #返回1-10之间的一个随机数，不包括10
```

```
2 >>> random.randint(1,10) #返回1-10之间的一个随机数，包括10
3 >>> random.randrange(0, 100, 2) #随机选取0到100间的偶数
4 >>> random.random() #返回一个随机浮点数
5 >>> random.choice('abce3#$@1') #返回一个给定数据集中的随机字符
6 '#'
7 >>> random.sample('abcdefghij',3) #从多个字符中选取特定数量的字符
8 ['a', 'd', 'b']
9 #生成随机字符串
10 >>> import string
11 >>> ''.join(random.sample(string.ascii_lowercase + string.digits, 6))
12 '4fvda1'
13 #洗牌
14 >>> a
15 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
16 >>> random.shuffle(a)
17 >>> a
18 [3, 0, 7, 2, 1, 6, 5, 8, 9, 4]
```