



PYTHON 核心开发

1. package & modules

为了更加友好的对 python 代码进行组织管理，python 中出现了包和模块的概念

类似生活中整理我们的物品一样，将代码按照不同的功能进行整理整合，可以很大程度的提升代码可读性和代码质量，方便在项目中进行协同开发！



这一部分，主要从以下几个部分进行说明：

- (1) 包和模块的定义
- (2) 包和模块的复用
 - a) `import` 引入和 `from...import` 引入
 - b) 绝对引入和相对引入
- (3) 自定义模块和包
- (4) 包的发布管理
- (5) python 中的“main 函数”







1.1.1. 包和模块的定义

python 中的包和模块，首先是按照代码的功能进行整理整合，想相似功能的代码/大量代码整理到一起方便统一管理

模块(module): python 中每个 **python 文件**就是一个**模块**，每个 python 文件中，封装类似功能的变量、函数、类型等等，可以被其他的 python 模块通过 **import** 关键字引入重复使用!

包(package): 包含多个 python 文件/模块的文件夹，并且文件夹中有一个名称为 `__init__.py` 的特殊声明文件，那么这个文件夹就是一个包(模块包)，可以将大量功能相关的 python 模块包含起来统一管理，同样也可以被其他模块通过 `import` 关键字引入重复使用封装的模块和代码!

	
具备一定功能的工具	包含很多工具、功能强大的工具箱
python 中的模块	python 中的包(模块包/程序包)

1.1.1.1 模块的定义

在工作目录 workspace/文件夹中创建自定义测试文件夹 demo01/

在 demo01/中创建文件 `utils.py`，这就是一个工具模块[见名知意]

```
demo01/utils.py
```

```
-----
```

```
'''工具模块，主要定义项目中可能使用到的各种工具变量、常量、函数、类型等等'''
```





1.1.2. 包的定义

在 demo01/文件夹中，创建一个文件夹 modules/

在 modules/文件夹中，创建一个空文件__init__.py

此时，modules 就是程序包，可以在该文件夹中定义各种模块，如 User.py..

```
demo01/  
    modules/  
        __init__.py  
        User.py    # 属于modules 包的一个模块
```

1.2. 包和模块的复用

python 中，将代码封装成包和模块，最主要的目的是通过有效的整理代码，提高代码的复用性能，这里整理好封装起来的包和模块的代码，就可以被其他代码引入使用了，类似生活中的工具被其他人借用一样

 <p># 飞机大战游戏开发 import pygame</p>	
<p>自行开发的游戏代码中，本身没有对键盘鼠标、显示器的控制代码，但是通过引入了pygame 模块，让自己的代码中具备了这样的功能，pygame 代码被复用了</p>	<p>生活中的三口之家，家里买车~但是可以通过租车实现自驾游！具备了有车的功能物品车被复用了</p>

1.2.1. import 和 from .. import

包和模块的引入，通常有两个关键语法

- import 包/模块





- from 包/模块 import 具体对象

(1) import 方式

基本语法

```
# 引入方式
import 模块
# 使用模块中的数据
模块.变量
模块.函数
模块.类型
```

REMARK: import 方式可以引入包、模块;

import 引入的包和模块会自动从当前文件夹中、系统环境变量 **PYTHONPATH** 中、以及系统的 **sys.path** 路径中查询是否存在该名称的包/模块

如果不存在, 就会出现错误: no module named 'xxxxxx'

案例操作代码:

(2) from xx import 方式

from xx import 方式基本语法如下

```
from . import xxx # 从当前模块路径下, 引入 xxx 模块
from .. import xxx # 从当前模块的父级路径下, 引入 xxx 模块
from pkg import module # 从 pkg 包中引入一个模块 module
from pkg.module import vars, func, clazz# 从指定的模块中直接引入
```

REMARK: from xx import 语法方式, 主要是针对出现了包结构的 python 代码而特定的代码引入方式, 首先要非常明确代码的组织结构才能正确使用 from import 语法进行代码的引入和复用

REMARK2: 通过上述代码, 可以看到 from import 语法区分为两种操作
使用了 ./.. 的相对路径的引入操作方式
直接使用包/模块名称的绝对引入操作方式





1.2.2. from import 相对引入

首先，相对引入本身是相对当前正在操作的文件的路径

同一级路径使用符号：.

上一级路径使用符号：..

创建测试项目文件夹 demo02/

项目中创建一个工具模块：utils.py

```
# coding:utf-8
# 测试变量
test_msg = "hello"
# 测试函数
def test_func():
    print("test 函数操作")
# 测试类型
class Test:
    pass
```

项目中创建主测试模块：main.py

```
# 模块的相对引入
from . import utils
# 测试变量
print(utils.test_msg)
# 测试函数
utils.test_func()
# 测试类型
t = utils.Test()
```

包的引入可以通过相对路径直接操作，这里的 demo02/ 可以是 python 的包，也可以是一个普通包含 python 代码的文件夹。

但是需要注意的是，一旦使用相对路径，就要明确所谓相对路径~是依赖他们所属的父级文





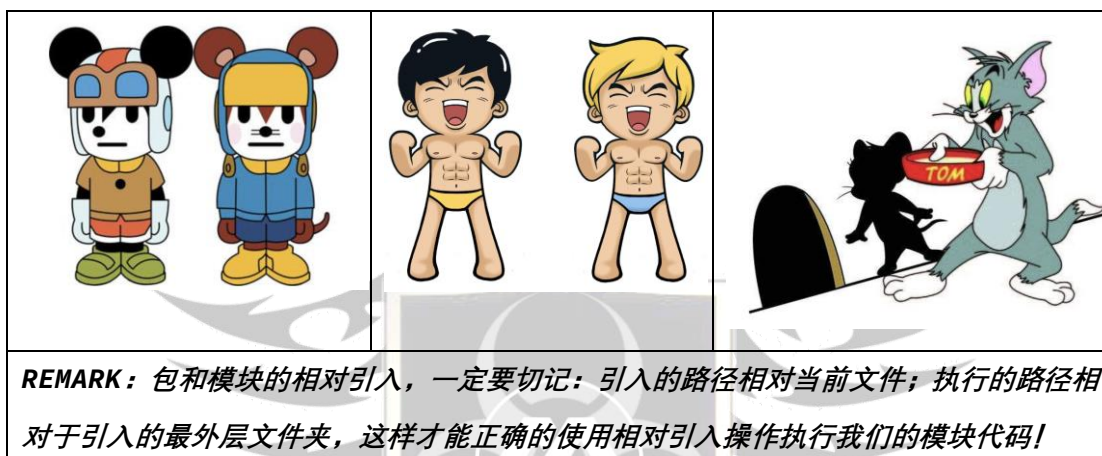
件夹确定的, 就类似生活中的兄弟姐妹的称谓一样, 是相对他们的父亲来说的, 所以运行代码需要在 demo02/ 文件夹所在路径, 执行如下命令运行

```
# 命令行执行运行代码的命令, 告诉python解释器在执行指定路径中的python代码  
python -m demo02.main
```

REMARK: 如果直接在 demo02/ 文件夹中, 执行命令

`python main.py` 就会出现如下错误

`ImportError: cannot import name 'utils'`



REMARK: 包和模块的相对引入, 一定要切记: 引入的路径相对当前文件; 执行的路径相对于引入的最外层文件夹, 这样才能正确的使用相对引入操作执行我们的模块代码!

1.2.3. from import 绝对引入

绝对引入操作方式比较直接, 从最外层的包的源头直接开始操作;

如: `from pygame import K_A, K_S, K_D, K_W`

这种操作方式在第三方模块的操作中是司空见惯的, 但是在独立的项目开发中使用较少!

还是上面那个案例操作, 修改 main.py 代码如下:

```
# 相对引入操作  
# from . import utils  
# 绝对引入操作  
from demo02 import utils  
  
# 使用utils中的变量、函数、类型等等..
```

运行操作方式当然和前面讲过的类似, 既然你确定了项目是从 demo02/ 为最外层文件夹的





话，那么运行也是参考 demo02 来执行命令

```
python -m demo02.main
```

1.3. 再说模块和包

前面的所有操作，都是在已有代码的基础上，对存在的文件进行处理，为了更友好的管理代码，我们对模块和包的操作方式进行详细的描述和说明

1.3.1. 模块(module)

python 中的模块，指代的就是一个 Python 文件

- (1) 在一个 python 模块中可以包含的数据有：变量、函数、类型等等，是一个完整的独立的代码块！
- (2) 独立的一个模块中的变量：全局变量、局部变量；能被其他模块引入使用的只有当前模块中的全局变量，其他模块对于当前模块中全局变量的操作和普通变量一致！
- (3) 模块一旦被其他模块引入，就会自动执行模块中的所有代码
- (4) 模块中的测试代码可以包含在 `if __name__ == "__main__":` 语句块中，这样不会再其他模块引入时执行这些测试代码

一个标准模块的定义方式【模块名称：见名知意】

```
# coding:utf-8

# 引入系统标准模块
# 引入第三方模块
# 引入自定义开发模块

# 声明定义变量
# 声明定义函数
# 声明定义类型
```





```
# 当前模块测试代码
```

```
if __name__ == "__main__":
```

```
    #测试代码位置，其他模块引入不会执行这里的代码
```

1.3.2. 包(package)

python 解释器在执行处理代码时，会默认将包含 Python 文件的文件夹处理成默认包
默认包只具备文件路径关联的功能，无其他功能！

python 中的标准包，是在文件夹中包含包声明文件 `__init__.py` 的文件夹，主要包含了一个名称为 `__init__.py` 的模块文件，该文件夹就是一个 Python 模块包

在一个 python 的包中，可以创建多个模块，统一由 python 包进行路径管理和导入方式的管理，如定义了一个 python 的数据模型包如下：

```
demo03/  
    users/  
        __init__.py    # 包声明模块  
        moduels.py    # 数据类型定义模块-> User<class>  
        manager.py    # 数据模型管理模块-> UserManager<class>  
        menus.py      # 界面处理模块-> show_index<function>  
        main.py       # 程序主模块
```

demo03/users/menus.py 中代码如下

```
# coding:utf-8  
  
def show_index():  
    print("系统主菜单")
```

demo03/main.py 中代码如下：

```
# coding:utf-8
```





```
# 引入包中指定的模块
```

```
from users import menus
```

```
# 使用模块中的代码
```

```
menus.show_index()
```

REMARK: 定义了包之后, 包中的变量、函数、类型的引入方式相似

1.3.3. 包: __all__

默认情况下, 包中所有的模块都是可以直接导入的, 同样为了使用方便, 可以通过通配符的方式来一次引入包中所有指定的模块

`__all__`属性就是用于模糊导入的特殊魔法属性, 值是一个包含模块名称的列表, 主要声明在`__init__.py`文件中, 用于定义可以使用通配符的方式引入的模块

```
demo03/users/__init__.py
```

```
-----
```

```
__all__ = ['module', 'manager', 'menus']
```

此时可以在`main.py`中通过通配符的方式将`__all__`中包含的模块一次性导入

```
from users import *
```

大牧莫邪

相当于

```
from users import module, manager, menus
```

1.4. 再说导入

从前面的内容可以看出, 代码的导入方式主要有两种语法结构, 这一小节对于两种导入结构的操作方式做一个简短的小结

1.4.1. import 导入

基本语法结构如下:





```
# 导入一个包/模块
import 包名称/模块名称

# 导入一个包中的某个模块
import 包名称.模块名称

# 导入一个包中的某个模块中的某个函数
import 包名称.模块名称.函数名称

# 导入一个包中的某个模块中的某个类型[名称较长, 使用别名]
import 包名称.模块名称.类型名称 as 别名
```

1.4.2. from import 导入

基本语法结构如下

```
# 导入一个包中的所有模块[__all__定义的模块]
from 包名称 import *

# 导入一个包中指定的模块
from 包名称 import 模块名称

# 导入一个包中指定模块中的某个函数
from 包名称.模块名称 import 函数名称

# 导入一个包中指定模块中的某个类型[名称较长, 使用别名]
from 包名称.模块名称 import 类型名称 as 别名
```

1.5. 自定义包的发布

如果已经开发好了具备某些通用功能的模块包, 恰好作者也是一个特别具有分享精神的开发人员, 要将自己开发的 python 模块发布出来, 提供给其他人进行操作使用, 类似于我们使用 pygame 这样第三方模块一样, 别人也可以通过命令直接安装使用, 应该怎么操作?

这一部分就是关于包的发布操作进行的详细说明





1.5.1. 本地发布

在我们已经开发好的一个包文件夹下，创建一个 python 模块：setup.py

模块中定义如下内容

```
# 引入构建包信息的模块
from distutils.core import setup

# 定义发布的包文件的信息
setup(
    name="damu_pkg01", # 发布的包文件名称
    version="1.0", # 发布的包的版本序号
    description="我的测试包", # 发布包的描述信息
    author="大牧莫邪", # 发布包的作者信息
    author_email="damu@163.com", # 作者联系邮箱信息
    py_modules=['__init__.py', '..'], # 发布的包中的模块文件列表
)
```

执行当前程序包文件的构建操作命令:按照标准格式组织包中的所有数据文件

```
python setup.py build
```

REMARK: 构建完毕的文件

可以通过 `python setup.py install` 命令直接当成第三方模块进行安装

大牧莫邪

执行命令进行包的打包发布

```
python setup.py sdist
```

REMARK: 打包的文件，主要是方便进行网络传输，打包之后会在 `dist` 中创建包含包中所有信息的 `tar.gz` 压缩包文件；该文件就可以通过 `git` 等方式提交给对应的开源组织发布你的自定义模块了！

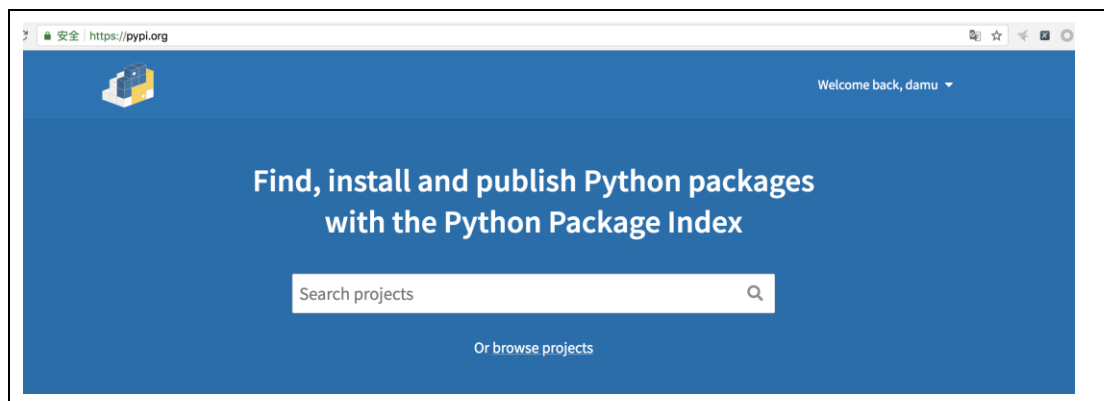
1.5.2. 网络发布

经历了前面一小节的操作之后，我们能不能让我们自己的包也类似于 `pip install` 这样的方式通过网络进行安装呢？当然可以！当然前提条件是我们将自己的包发布到网络上！



首先, 进入 <http://pypi.python.org/pypi> 网站上, 注册一个自己的账号吧!

这个网站是目前大部分 python 第三方模块集中的一个管理社区平台



其次: 你已经准备好你自己的 python 程序包, 并在包中准备好了 `setup.py` 文件

在 `setup.py` 中, 已经定义好了程序包的所有描述信息

```
damy_users/  
  __init__.py  
  moduels.py  
  manager.py  
  menus.py  
  setup.py
```

执行打包命令

```
python setup.py sdist
```

大牧莫邪

接下来: 将是激动人心的时刻!

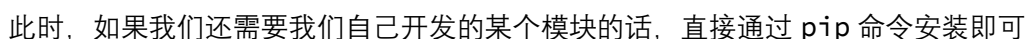
首先安装第三方模块: `twine`, 用于上传我们打包的项目文件

```
pip install twine
```

上传项目

```
twine upload dist/*
```





```

C:\Users\user> pip install damu_users
Collecting damu_users
  Downloading https://files.pythonhosted.org/packages/ad/83/174d888f9
839c8434f1f39c6bac9fdf80de477b074d9a0113/damu_users-1.0.tar.gz
Building wheels for collected packages: damu-users
  Running setup.py bdist_wheel for damu-users ... done
  Stored in directory: C:\Users\user\AppData\Local\pip\Cache\wheels\33\7f\
a115128e38adcbe59a515723547bd8360160cf66370
Successfully built damu-users
Installing collected packages: damu-users
Successfully installed damu-users-1.0

```

大牧莫邪

1.6. 关于 Main 主方法

类比其他编程语言，为了区别同一个应用程序中运行程序的入口，都会提供一个程序运行的主方法（类似 Java 中的 `main` 方法）或者主函数（类似 C/C++ 中的主函数）

PYTHON 最小运行单元是代码块，代码块中甚至只是包含一行代码也能正常运行，所以没有主方法或者函数的概念，但是区别程序运行的入口是非常有必要的，比如我们在自定义开发一个模块时，为了保证模块代码的正确性，很多人都会在模块中编写一些测试代码(规范不是很推荐这样做，但是不否认这是大多数人的懒癌操作方式)，这些代码只有当前模块作为程序入口时才允许运行，被其他模块引入时要求不让运行，怎么做到这样一点呢？PYTHON 中的模块属性 `__name__` 可以做到模拟主函数的操作

下面通过两种常规操作说明模拟主函数的意义





1.6.1. 普通模块的定义和问题

在开发过程中，我们常规模块的定义开发，经常定义如下格式的代码(demo.py)：

```
'''某个模块的注释'''  
# import 各种依赖模块  
# 定义模块中的变量  
count = 0  
msg = None  
# 定义模块中的函数  
def show_info(info):  
    print("info for send: {}".format(info))  
  
# 测试代码  
show_info("my name is damu")
```

定义好模块之后，通过执行命令 `python demo.py`，就可以测试模块中的代码的正确性

使用场景：如果我们自己开发了另一个模块，需要依赖当前开发的这个模块就需要通过 `import demo` 的方式在另一个模块中引入，那么问题在哪里呢？

大牧莫邪

问题显现：模块一旦被其他模块引入，模块中的代码都会直接执行，那么这里 `demo` 模块被其他模块引入时，正常功能代码都会加载，但是用于测试的代码也被运行了，这可能会引起不必要的错误和系统资源的浪费，并不是我们想要的

1.6.2. PYTHON 中的 main 方法

PYTHON 中的模块有一个魔法属性 `__name__`，表示当前模块的运行名称，具有如下特点：

- 当前模块如果是程序运行入口，`__name__` 的值： `__main__`
- 当前模块如果被其他模块 `import` 引入，`__name__` 的值：模块名称

模块的魔法属性 `__name__`

定义模块 `demo.py`，`main.py`，用于测试魔法属性的操作




```
demo.py
print(__name__)
print("my name is demo")
```

```
main.py
import demo
print("my name is main")
```

执行测试运行如下:

首先直接运行 demo: python demo.py, 得到结果如下:

```
__main__
my name is demo
```

直接运行 main: python main.py

```
demo
my name is demo
my name is main
```

很明显可以看到运行的过程中, 作为入口和被 import 的区别, 我们可以利用__name__的特性完成 main 方法的设计

```
重新定义 demo.py
-----
# coding:utf-8
# import 各种依赖
# 定义模块变量
count = 0
msg = None
# 定义模块函数
def show_info(info):
    print("info for send: {}".format(info))

# 测试代码: 包含在模拟的 main 方法中, 解决被 import 带来的执行问题
if __name__ == "__main__":
```





执行测试的代码: 该代码只有当前模块在运行时才会执行, 其他模块 import 操作并不会执行, 避免了由于测试代码导致错误发生的可能

```
show_info("hello my name is main")
```

