第三章 高级特性

3.1 对象池

pythobn中，所有数据类型都是对象，即使是值类型，变量中存放的也不是数据值，而是值类型对象的内存地址。这种通过内存地址间接访问数据对象的方式称为引用。

通过变量之间的赋值可以使两个变量引用相同的对象，而使用身份运算符is则可以判定两个变量是否引用同一个对象

```
In [1]: a = 3

In [2]: b = a  # 将a中的内存地址赋值给b

In [3]: print(a is b) # 判断是否引用了同一个对象
True
```

实际上，Python为了优化速度，使用了**小整数对象池**， 避免为整数频繁申请和销毁内存空间。[-5, 256] 这些整数对象是提前建立好的，不会被垃圾回收。所有位于这个范围内的整数使用的都是同一个对象

```
In [1]: a = 3

In [2]: b = 3

In [3]: print(a is b)
True
```

整数类型的池中的值是不可变的，也就是说当修改数据的值时，会将修改后的值所在的内存地址赋值给变量，而不会影响原来的值：

```
In [16]: a = 3

In [17]: print(id(a)) # id函数可以获取变量引用的内存地址
4492330400

In [18]: a += 1

In [19]: print(id(a))
4492330432
```

而对于其他的值类型，即使值相同，也不是引用的同一个对象：

```
In [6]: d = 3.1

In [7]: e = 3.1

In [8]: print(d is e)
False
```

对于其他的基本数据类型，只要值相同，它们引用的也是同一个对象：

```
In [1]: str_a = "abc"

In [2]: str_b = "abc"

In [3]: print(str_a is str_b)
True

In [4]: a = True

In [5]: b = True

In [6]: print(a is b)
True

In [7]: c = None

In [8]: d = None

In [9]: print(c is d)
True
```

3.2 高阶函数

在python中，调用函数时也可以将其它函数名称作为实参来使用，这种能够接受函数名称作为参数的函数称为高阶函数。

3.2.1 函数式编程

在python中，可以将函数名称赋值给变量，赋值后变量指向函数对象；也允许将函数名称作为参数传入另一个函数，还允许从函数中返回另一个函数。

```
def add(x, y):  # 定义加法函数
    return x + y


def substract(x, y):  # 定义减法函数
    return x - y
```

```
def multiply(x, y):  # 定义乘法函数
    return x * y


def divide(x, y):  # 定义乘法函数
    return x / y


def artithmetic(x, y, operate):  # 定义算术运算函数。形参operate接受函数名称
    return operate(x, y)


a, b = eval(input("请输入两个数（以，号分割：）"))
c = input("请输入运算符")
if c == "+":
    op = add  # 函数名称赋值给变量
elif c == "-":
    op = substract
elif c == "*":
    op = multiply
elif c == "/":
    op = divide
else:
    op = -1
if op != -1:
    print(artithmetic(a, b, op))  # 将实参op指向的函数作为参数赋值给函数
```

3.2.2 map函数

内置的map()函数是一个可用于序列对象的高阶函数，其调用格式如下：

```
    map(func,iterable)
```

- func 用于指定映射函数
- iterable：用于指定可迭代对象
- return: map()函数将映射函数func()作用到可迭代对象iterable的每一个元素并组成新的map对象返回

【例题】将输入的所有单词转换为大写

```
upper = str.upper  # 为了防止str变量覆盖内置函数


def get_strs():
    strs = []
    print("请输入英文单词（q=退出）")
```

```
    while 1:
        str = input("请输入：")
        if str.lower() == "q": break
        strs.append(str)
    return strs


strs = get_strs()

print("=" * 30)

print("输入的单词：")
for str in strs:
    end_str = "," if strs.index(str) < len(strs) - 1 else "\n"
    print(str, end=end_str)

print("=" * 30)
print("格式化后的单词")
# 调用map函数，以 upper作为参数，也就是将其执行的upper函数传入map函数。第二个参数为可迭代的对象
strs = map(upper, strs)
list = list(strs)  # map对象转换为列表。目的是使用列表中的index方法
i = 0
for str in list:
    i += 1
    end_str = "," if list.index(str) < len(list) - 1 else "\r"
    print(str, end=end_str)
```

3.2.3 filter函数

内置函数filter()也是一个高阶函数，可用于对序列进行筛选。格式如下：

```
filter(func,iterable)
```

- func：指定筛选函数
- iterable：指定可迭代对象
- func()函数将作用于迭代对象中的每一个元素，并根据func的返回值是True还是False类判断是保留还是丢弃该元素
- return： 返回值是一个filter对象。

【例题】 奇数筛选

```
# 奇数筛选函数
def is_odd(num):
    return num % 2 == 1
```

```
nums = [x for x in range(1, 51)]
# 筛选奇数
nums_fliter = filter(is_odd, nums)

filter_list = list(nums_fliter)

i = 0
for lst in filter_list:
    i += 1
    print(lst, end="\t")
    if i % 5 == 0: print("\r")
```

3.3 匿名函数

匿名函数是指没有名称的函数，它只能包含一个表达式，而不能包含其他语句，该表达式的值就是函数的返回值。

使用def语句创建用户自定义函数时必须指定一个函数名称，以后需要时便可以通过该名称来调用函数，这样有助于提高代码的复用性。如果某项计算功能只需要临时使用一次而不需要在其他地方重复使用，则可以考虑通过定义匿名函数来实现这项功能。

3.3.1 匿名函数的定义

语法格式：

```
lambda 参数列表:表达式
```

- 关键字lamba 表示匿名函数，因此匿名函数也称为lambda函数
- 参数列表中的各个参数以逗号分隔
- 表达式的结果就是函数的返回值

3.3.2 匿名函数的调用

函数式编程是匿名函数的主要应用场景。虽然没有名称，但匿名函数仍然是函数对象，在程序中可以将匿名函数赋值给一个变量，然后通过该变量来调用匿名函数。

```
f = lambda x, y: x * y

res = f(3, 4)
print(res)  # 12
```

3.3.3 作为高阶函数的参数

【例题】 匿名函数作为过滤函数的参数，用来筛选出1-10之间的所有奇数

```
odd = filter(lambda x: x % 2 == 1, range(1, 11))
print(list(odd))  # [1, 3, 5, 7, 9]
```

3.3.4 作为序列或字典的元素

```
op = {"add": lambda x, y: x + y, "sub": lambda x, y: x - y, "multi": lambda
x, y: x * y, "divide": lambda x, y: x / y}

res = op["add"](3, 4)
print(res) # 7
```

3.3.5 作为函数的返回值

【例题】 算术四则运算

```
def arithmetric(c):
    if c == "+":
        return lambda x, y: x + y
    elif c == "-":
        return lambda x, y: x - y
    elif c == "*":
        return lambda x, y: x * y
    elif c == "/":
        return lambda x, y: x / y


if __name__ == '__main__':
    x, y = eval(input("请输入两个数字"))
    choice = input("请选择一种运算方式:")
    if choice in ["+", "-", "*", "/"]:
        print("{0}{1}{2}={3}".format(x, choice, y, arithmetric(choice)(x,
y)))
    else:
        print("无效的运算方式")
```

3.4 递归函数

递归函数是指自我调用的函数，即在一个函数内部直接或间接调用该函数自身。 递归函数具有以下的特性：

- 必须有一个明确的递归结束条件
- 每当进入更深一层递归时，问题规模相比上次递归都应该有所减少
- 相邻两次递归之间有紧密的联系。通常前一次的输出会作为后一次的输入。

【例题】 从键盘输入一个正整数，然后计算其阶乘 阶乘运算可以表示为：

$$\begin{cases} 1 & n \leq 1 \\ n(n-1)! & n > 1 \end{cases}$$

```
def fact(n):
    if n>1:
        return n*fact(n-1)
    else:
        return 1
print(fact(4))
```

程序中能够对变量进行存取操作的范围称为变量的作用域。变量按照作用域的不同一般分为局部变量和全局变量。

3.5 变量作用域

3.5.1 局部变量

在一个函数体或语句块内部定义的变量称为局部变量。局部变量的作用域就是定义它的函数体或语句块。

定义一个函数时，可以在函数体内部定义另一个函数，此时两个函数形成嵌套关系，内层函数只能在外层函数中调用，而不能在模块级别被调用

在具有嵌套关系的函数中，在外层函数中定义的局部变量可以直接在内层函数中使用。默认情况下，不属于当前局部作用域的变量具有只读性质，可以直接对其进行读取，但如果对其赋值，则python会在当前作用域中定义个新的同名局部变量。

如果外层函数和内层函数中定义了同名变量，则在内层函数中将优先使用自身所定义的局部变量；在存在同名变量的情况下，如果要在内层函数中使用外层函数中定义的局部变量，则应使用关键字nonlocal对变量进行声明。

```
def outer():
    x = 3

    def inner():
        nonlocal x  # 声明使用外层变量
        x = 5  # 对外层变量进行赋值
        print("内层函数中，x={}".format(x))

    inner()
    print("外层函数中，x={}".format(x))


if __name__ == "__main__":
    outer()
```

运行结果：

```
内层函数中，x=5
外层函数中，x=5
```

如果不用nonlocal进行声明：

```
def outer():
    x = 3

    def inner():

        x = 5  # 对x赋值，会创建一个与同名变量
        print("内层函数中，x={}".format(x))

    inner()
    print("外层函数中，x={}".format(x))


if __name__ == "__main__":
    outer()
```

运行结果：

```
内层函数中，x=5
外层函数中，x=3
```

3.5.2 全局变量

在python中，一个python程序文件就是一个模块。模块级别上，在所有函数外部定义的变量称为全局变量。它可以在当前模块范围内被引用。

默认情况下，在python程序中引用变量的优先顺序如下：

1. 当前作用域局部变量
2. 外层作用域变量
3. 当前模块中的局部变量
4. python内置变量

如果在某个函数内部定义的局部变量与全局变量同名，则优先使用局部变量。在这种情况下，如果要在函数内部使用全局变量，则应使用global关键字对变量进行声明，否则会创建同名的局部变量。

```
x = "global"
y = "other global"

def func():
    x = "local"
    global y  # 声明使用全局变量
    y = "local"
    print("函数内部x={},y={}".format(x,y))

func()
print("模块中，x={},y={}".format(x,y))
```

运行结果：

```
函数内部x=local,y=local
模块中，x=global,y=local
```

通过定义全局变量可以在函数之间提供直接传递数据的通道。

- 将一些参数的值存放在全局变量中，可以减少调用函数时传递的数据量
- 将函数的执行结果存放在全局变量中，则可以使函数返回多个值

3.6 闭包

如果在函数内部定义了另一个函数，在内层函数中对外层函数中定义的局部变量进行了存取操作，并且外层函数返回对内层函数的引用，则这个内层函数称为闭包（closure）。 闭包是函数式编程的重要语法结构，是将函数的语句和执行环境打包在一起的对象。当执行嵌套函数时，闭包将获取内层嵌入函数所需要的整个环境。闭包是由函数和与其相关的环境组合成的实体。在程序运行时可以生成闭包的多个实例，不同的引用环境和相同的函数组合可以产生不同的实例。

```
def outer(name):
    msg = "学生信息"
    # inner就是一个闭包
    def inner(age, gender):
        print("{}:{},{},{}".format(msg, name, age, gender))  # 使用外层函数中
的信息

    return inner  # 返回内层函数


if __name__ == "__main__":
    stu1 = outer("张三")  # 返回的是内层函数
    stu1(18, "男")  # 调用内层函数
    stu2 = outer("李四")
    stu2(19, "女")
```

3.7 装饰器

装饰器(decorator)是在闭包的基础上发展起来的。装饰器在本质上也是一个嵌套函数，其外层函数的返回值是一个新的函数对象的引用，所不同的是，其外层函数可以接受一个现有函数对象引用作为参数。

装饰器可以用于包装现有函数，即在不修改任何代码的前提下为现有函数增加额外功能。

装饰器通常应用于有切面（面向切面编程是指运行时动态实现程序维护的一种技术）需求的场景，例如插入日志、性能测试、事务处理 缓存以及权限校验等。装饰器是解决这类问题的绝佳设计，有了装饰器，就可以抽离出大量与函数功能本身无关的雷同代码并继续重用它们。

3.7.1 无参数装饰器

引入装饰器是为了在不修改原函数定义和函数调用代码的情况下拓展程序的功能。装饰器是在闭包的基础上传递了一个函数，然后覆盖原来函数的执行入口，以后调用这个函数时即可额外添加一些功能。

(1) 定义装饰器

装饰器的实质是一个高阶函数，其参数是要装饰的函数名，其返回值是完成装饰的函数名，其作用是为已经存在的函数对象添加额外的功能，其特点是不需要对象有函数做任何代码上的变动。

定义装饰器通常会涉及以下3个函数：

- 装饰器函数：它在嵌套关系中作为外层函数出现，其函数体内容包括定义一个内层函数以完成装饰功能的函数，通过return语句向调用者返回内层函数对象引用。
- 目标函数：即需要进行装饰的函数，它作为装饰器函数的形参出现，该函数的定义则出现在调用装饰器的地方
- 完成装饰的函数：它在函数嵌套中作为内层函数出现，用于为待装饰的目标函数添加额外的功能。在这个内层函数中调用目标函数，并未目标函数添加一些新的功能

装饰器的语法模板：

```
def 装饰器名称(待装饰函数名称):
    def 装饰函数名称():
        #目标函数前添加功能
        #目标函数调用
        #目标函数执行后添加功能
    return 装饰函数名称
```

例如：

```
def decorator(func):     # 定义装饰器函数
    def wrapper(x, y):  # 定义完成装饰的函数
        print("参数x={},参数y={}".format(x, y)) # 添加打印参数的功能
        return func(x, y) # 调用待装饰函数

    return wrapper # 返回完成装饰的函数的对象引用
```

(2) 调用装饰器

调用装饰器时，需要在装饰器函数前面加上符号“@”，后面跟要装饰的目标函数定义。 语法如下：

```
@装饰器名称
def 目标函数名称():
    函数体

目标函数调用
```

例如：

```
@decorator  # 调用装饰器
def add(x, y):  # 定义目标函数
    return x + y

print("{}+{}={}".format(2, 4, add(2, 4)))  # 调用模板函数
```

调用装饰器后，调用目标函数时实际调用的是装饰函数：

```
@decorator  # 调用装饰器
def add(x, y):  # 定义目标函数
    return x + y


print(add)
```

结果：

```
<function decorator.<locals>.wrapper at 0x115403f28>
```

目标函数如果不添加装饰器，调用的则是函数自身：

```
def add(x, y):  # 定义目标函数
    return x + y


print(add) # <function add at 0x11a427268>
```

【例题】 通过装饰器为模拟游戏函数添加计时功能

```
import time


def decorator(func):
    def wrapper():
        print("游戏现在开始：")
        start = time.time()
        func()
        end = time.time() - start
        print("游戏历时{:2f}".format(end))
    return wrapper

@decorator
def play_game():
    for i in range(100000000):
        pass

if __name__=="__main__":
    play_game()
```

运行结果：

```
游戏现在开始：
游戏历时2.08
```

3.7.2 有参装饰器

有参装饰器的调用格式是”@z装饰器名称(参数)“。通过提供参数，可以为装饰器的定义和调用带来更大的灵活性。

无参数装饰器本质就是一个双层结构的高阶函数；有参数装饰器则是一个三层结构的高阶函数。有参数装饰器可以看做是在无参数装饰器外面又封装了一层函数。

所谓有参数是指最外层装饰器函数可以有一个或多个参数，这些参数可以在内部各层函数中使用，而且最外层装饰器的返回值就是内层的无参数装饰器。

【例题】 通过有参数装饰器为模拟游戏添加计时功能，并允许指定游戏名称。

```
import time


def decorator(name):  # 最外层函数接收参数
    def _decorator(func):  # 中层函数用于接受要装饰的目标函数
        def wrapper():  # 内层函数用于完成装饰功能
            print("《{0}》游戏现在开始:".format(name))
            start = time.time()
            func()
            end = time.time() - start
            print("《{}》游戏历时{:.2f}:".format(name, end))

        return wrapper

    return _decorator


@decorator("王者荣耀")
def play_game1():
    for i in range(10000000):
        pass


@decorator("英雄联盟")
def play_game2():
    for i in range(10000000):
        pass


if __name__ == "__main__":
    play_game1()
    play_game2()
```

调用模板函数时，实际调用的还是最内层的无参的完成装饰的函数：

```
 print(play_game2)  # <function decorator.<locals>._decorator.
<locals>.wrapper at 0x1123b0598>
```

3.7.3 多重装饰器

多重装饰器时指使用多个装饰器来修改同一个函数，此时要注意多重装饰器的执行顺序是后面的装饰器先执行，前面的装饰器后执行，即后来者居上

```
# 定义第一个装饰器
def first(func):
    print("函数{}传递到了第一个装饰器".format(func.__name__))
```

```
    def _first(*args, **kw):
        print("在_first()中调用函数{}".format(func.__name__))
        return func(*args, **kw)

    return _first


# 定义第二个装饰器
def second(func):
    print("函数{}传递到了第二个装饰器".format(func.__name__))

    def _second(*args, **kw):
        print("在_second()中调用函数{}".format(func.__name__))
        return func(*args, **kw)

    return _second


# 调用两个装饰器
@first
@second
def test():
    print("现在开始执行test()函数")


if __name__ == "__main__":
    test()
```

运行结果：

```
函数test传递到了第二个装饰器
函数_second传递到了第一个装饰器
在_first()中调用函数_second
在_second()中调用函数test
现在开始执行test()函数
```

3.8 模块

在python中，一个扩展名为".py"的程序文件就是一个模块(module)。为了方便组织和维护代码，通常可以将相关的代码存放到一个python模块中。

3.8.1 模块的定义与使用

(1) 导入整个模块

语法：

```
import 模块名1 [as 别名1] [,模块名n [as 别名n]]
```

- 模块名是去掉扩展名".py"后的文件名
- 导入多个模块时，各个模块名之间用逗号分隔

导入一个模块后，就可以调用该模块中的所有函数，调用格式如下：

```
 模块名或别名.函数名(参数)
```

也可以将"模块名.函数名"赋值给一个变量，然后通过该变量来调用模块中的函数。

(2) 从模块中导入指定项目

语法格式：

```
from 模块名 import 项目1 [,项目n]
```

通过这种方式导入模块中指定的项目，在这种情况下可以直接调用函数，而不需要添加模块名作为前缀。

如果要导入所有项目，可以使用下面的语句：

```
from 模块名 import *
```

【案例】 首先创建一个名为moudule.py文件，添加如下代码：

```
def test():
    print("我是moudule模块中的代码")
```

然后创建一个demo.py文件，添加如下代码：

```
import module  # 导入模块

module.test()  # 调用模块中的函数
```

3.8.2 设置模块的搜索路径

python解释器遇到import语句时，如果所指定的模块包含在搜索路径列表中，系统会导入该模块

python导入模块时的的搜索路径可以通过sys.path对象来查看：

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

该对象是一个列表：

- 第一个元素为当前程序文件所在的目录
- 还包括标准模块所在目录
- 通过PYTHONPATH环境变量配置的目录
- 在扩展名为".pth"的文件中设置的目录

(1) 动态添加模块搜索路径

通过调用sys.path.insert(0,path)或sys.path.append(path)函数可以动态地添加模块搜索路径，将指定的目录添加到搜索路径中。例如：

```
import sys
sys.path.insert(0,"/Users/fray.hao/")
print(sys.path)
```

使用这种方式添加的模块搜索目录是临时性的，只在程序运行期间有效。当退出python环境后，搜索路径将失效

(2) 通过环境变量配置搜索路径

要设置永久的python模块搜索路径，可以使用PYTHONPATH环境变量来配置搜素路径。所设置的路径会自动添加到sys.path列表中，而且可以在不同的pyahton版本中共享。

【例题】mac中添加环境变量

首先打开文件

```
vim ~/.bash_profile
```

添加环境变量：

```
# 配置python搜索路径
export  PYTHONPATH="/Users/free.hao":"/Users/fray.hao"
```

最后使文件生效：

```
source ~/.bash_profile
```

(3) 使用扩展名为".pth"文件设置搜索路径

要永久设置pytho的搜索路径，也可以在python安装路径的site-packages目录（sys.path列表中的第四个元素就是该目录的路径）中创建一个扩展名为".pth"的文件，并将模块的搜索路径写进去，其中每一个目录占一行。例如，创建search.pth，它的内容：

```
/Users/free.hao
/Users/
```

3.8.3 模块探微

(1) dir()

在python中，模块也是函数，加载一个模块之后，可以使用内置函数dir()列出该模块对象所包含的函数名称和全局变量名称的列表 例如有一个文件名为demo.py的文件，其源代码如下：

```
'''
    模块名：demo
    内容：全局变量x,y；函数func()
    功能：用于演示
'''
x=123
y="demo"
def func():
    '''
    功能：打印hello world
    功能：打印hello world
    :return：无
    '''
    print("hello world")
```

当其它程序文件中导入demo模块时，可以使用dir()函数来查看demo中包含的变量和函数。

```
import demo

print(dir(demo))
```

运行结果：

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__', 'func', 'x', 'y']
```

dir()函数返回的是一个列表对象，其中包括在该模块中定义的所有变量名称和函数名称，还有一些内置全局变量的名称。 内置全局变量功能如下：

全局变量	功能
__builtins__	对内置模块的引用。该模块在python启动后首先被加载。该模块中的函数即内置函数，可直接被使用
__cached__	表示当前模块经过编译后生成的字节码文件的路径
__doc__	表示当前模块的文档字符串
__file__	表示当前模块的完整路径
__loader__	表示用于加载模块的加载器
__name__	表示当前模块执行过程中的名称。当直接运行模块时，则模块名为”__main__“。当模块被其它模块导入时，值为模块的名称
__package__	表示当前模块所在的包，也就是获取导入文件的路径
__spec__	表示当前模块的规范（名称，加载器和源文件）

(2) help()

将模块名传入help()，可以查看关于模块的名称、函数、变量、文件路径等信息

```
help(demo)
```

运行结果：

```
Help on module demo:
NAME
    demo
DESCRIPTION
    模块名：demo
    内容：全局变量x,y；函数func()
    功能：用于演示
FUNCTIONS
    func()
        功能：打印hello world
        功能：打印hello world
        :return: 无
DATA
    x = 123
    y = 'demo'
FILE
    /Users/free.hao/PycharmProjects/Demo/demo.py
```

也可以将模块的函数名传入，以查看关于该函数的帮助信息

```
help(demo.func)
```

运行结果：

```
Help on function func in module demo:
func()
    功能：打印hello world
    :return: 无
```

(3) __name__

Python模块中可以定义一些变量、函数和类，以便其他模块导入和调用；模块中也可以包含能够直接运行的代码，如函数的调用。当导入该模块时，不仅导入了一些变量和函数，还会直接调用函数。 在很多情况下只希望在直接运行模块时才去执行函数的调用，在被其他模块导入时，则不执行。这种情况下就可以使用__name__来加以区分。当直接运行模块时内置全局变量__name__的名称为__main__，因此可以通过下面的语句对函数的调用进行限制：

```
# 只有在直接运行模块时才会调用函数。在其他模块中导入时，不会调用
if __name__ == "__main__":
    func()
```

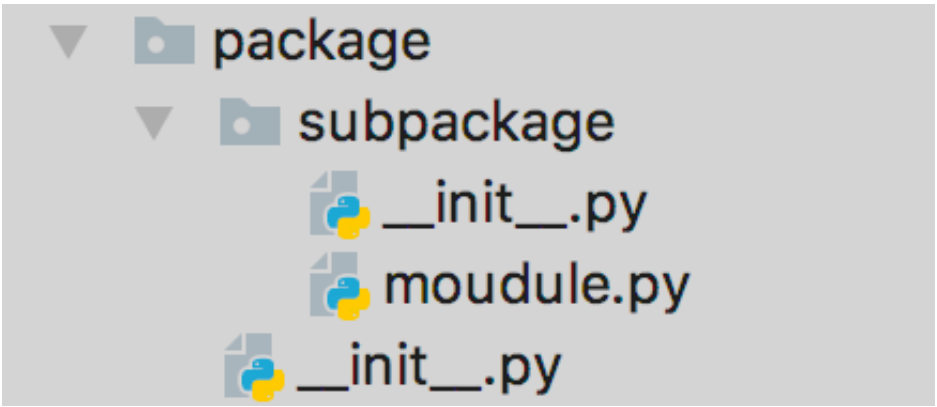
3.9 包

创建许多模块后，如果希望某些功能相近的模块组织在同一个文件夹下，就需要用到包。

在python中，包是一个有层次的文件目录结构，它定义了由若干模块或子包组成的应用程序执行环境。

包是python模块文件所在的目录。在包目录中必须有一个文件名为__init__.py的包定义文件，用于对包进行初始化操作。

如下面的包结构：



包的使用方法与模块类似。根据需要，可以从包中导入单独的模块，例如：

```
import package.subpackage.moudule # 最后一项必须是包或模块，不能是类、函数或变量
```

使用这种语法格式导入带包的模块时，必须通过完整路径形式来使用模块中的函数或变量。例如：

```
package.subpackage.moudule.test()
```

可以使用from ... import语句来导入包中的模块：

```
from  package.subpackage import moudule

moudule.test() # 此时，可以直接使用模块名而不用加上包前缀
```

还可以直接导入模块中的函数或变量：

```
from  package.subpackage.moudule import test

test() # 直接使用函数，不需要模块前缀
```

还可以在包文件__init__.py通过列表变量__all__设置要导入的模块的列表，则可以使用模糊导入。例如，在subpacakge包的包文件中添加：

```
__all__=["moudule"]
```

则可以使用模糊导入：

```
from package.subpackage import *
moudule.test()
```