# 程序结构

## 模块 Module ==文件

### 定义

包含一系列数据、函数、类的文件，通常以**.py**结尾。

### 作用

让一些相关的数据，函数，类有逻辑的组织在一起，使逻辑结构更加清晰。

有利于多人合作开发。

### 导入

#### import

1. 语法：

**import 模块名**

import 模块名 **as 别名**

1. 作用：将某模块整体**导入**到当前模块中,会将导入的东西执行一遍。
2. 使用：**模块名.成员**

****

#### from import

1. 语法：

from 模块名 import 成员名**[ as 别名1]**

1. 作用：将模块内的一个或多个成员导入到当前模块的**作用域**中。会将导入的东西执行一遍。
2. 使用：**直接使用成员名**

##### from import \*

1. 语法：from 模块名 import \*
2. 作用：将某模块的所有成员导入到当前模块。

练习1：

创建2个模块module\_exercise.py与exercise.py

将下列代码粘贴到module\_exercise模块中，并在exercise中调用。

data = 100

def func01():

print("func01执行喽")

class MyClass:

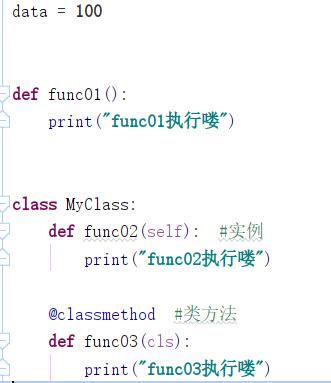
def func02(self):

print("func02执行喽")

@classmethod

def func03(cls):

print("func03执行喽")





练习2：将信息管理系统拆分为4个模块student\_info\_manager\_system.py

（1）创建目录student\_info\_manager\_system

（2）创建模块bll,存储XXController

业务逻辑层 business logic layer

（3）创建模块usl,存储XXView

用户显示层 user show layer

（4）创建模块model,存储XXModel

（5）创建模块main,存储调用XXView的代码

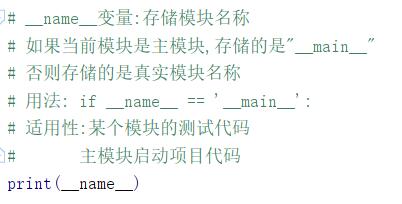
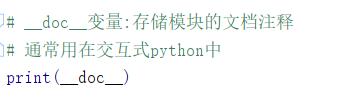
### 模块变量

\_\_**doc**\_\_变量：文档字符串。**主要用**在**交互式**当中，文件式中，ctrl + p 或者 ctrl +左键

\_**\_name**\_\_变量：模块自身名字，可以**判断是否为主模块**。

当此模块作为主模块(第一个运行的模块)运行时，\_\_name\_\_绑定'\_\_main\_\_'，不是主模块，而是被其它模块导入时,存储模块名。

根目录：主模块所在文件夹



### 加载过程

在模块导入时，模块的所有语句会执行。

如果一个模块已经导入，则再次导入时不会重新执行模块内的语句。（将模块加载到内存中）

**导入成功的唯一标准**：**导入路径** + **系统路径**(利用**mark标记根目录**) = **真实路径**

### 分类

1. 内置模块(builtins)，在解析器的内部可以直接使用。
2. 标准库模块，安装Python时已安装且可直接使用。（需要导入）如random、（时间元组）time、时间戳（最省存储）
3. 第三方模块（通常为开源），需要自己安装。（需要导入）
4. 用户自己编写的模块（可以作为其他人的第三方模块）。需要导入）

练习1：定义函数,根据年月日,计算星期。

输入：2020 9 15

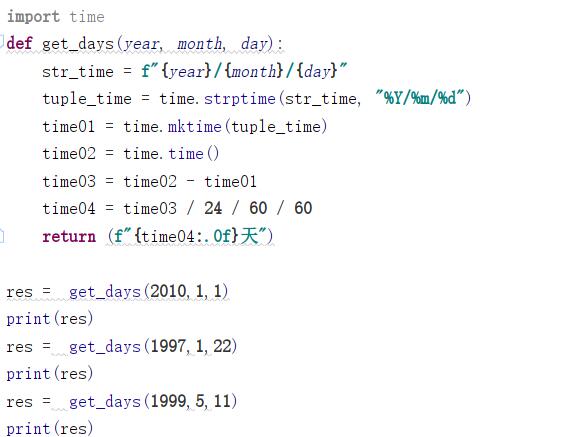
输出：星期二



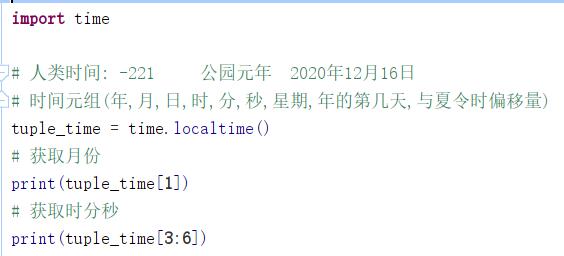
练习2：定义函数,根据生日(年月日),计算活了多天.

输入：2010 1 1

输出：从2010年1月1日到现在总共活了3910天



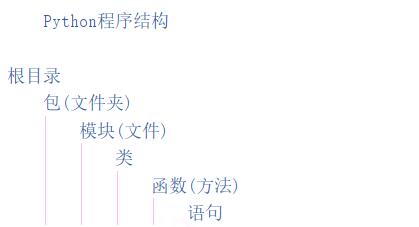
标准数据库，时间元组time、时间戳：



## 包package

### 定义

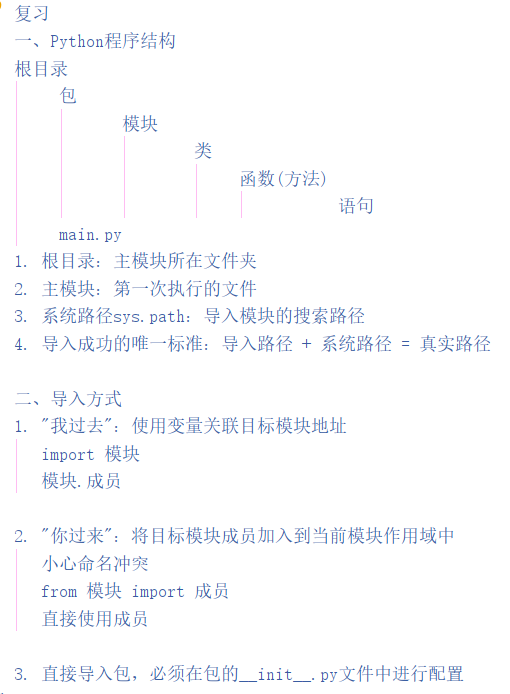
将模块以**文件夹的形式**进行分组管理。



### 作用

让一些相关的模块组织在一起，使逻辑结构更加清晰。

**复习:**

****

练习1：

1. 根据下列结构，创建包与模块。

my\_project01 /

main.py

common/

\_\_init\_\_.py

list\_helper.py

skill\_system/

\_\_init\_\_.py

skill\_deployer.py

skill\_manager.py

1. 在main.py中调用skill\_manager.py中实例方法。
2. 在skill\_manager.py中调用skill\_deployer.py中实例方法。
3. 在skill\_deployer.py中调用list\_helper.py中类方法。

练习2：

1. 根据下列结构，创建包与模块。

my\_project 02/

main.py

common/

\_\_init\_\_.py

list\_helper.py

skill\_system/

\_\_init\_\_.py

skill\_manager.py

1. 通过导入包的方式，在main.py中调用skill\_manager.py中实例方法。
2. 通过导入包的方式，在skill\_manager.py中调用list\_helper.py中类方法。

### \_\_init\_\_.py 文件

是包内必须存在的文件

会在包加载时被自动调用

**当导入包时,需要在包的\_\_init\_\_.py文件中配置**

#### 作用

记录import 包 语句需要导入的模块

### 搜索顺序

**导入搜索路径  
 按照列表顺序搜索,如果前面搜索到,立即停止  
 列表第一个元素永远是项目根目录(主模块所在文件夹)  
 将文件夹标记蓝色,实质就是添加到系统路径中**

**sys.path** 提供的路径

**import** sys  
  
print(sys.path)

# 异常处理Error

## 异常

1. 定义：运行时检测到的错误。
2. **现象**：当异常发生时，程序不会再向下执行，而不断转到函数的调用语句。
3. 常见异常类型：

-- 名称异常(NameError)：变量未定义。 print(data)

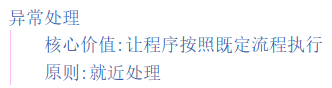
-- 类型异常(TypeError)：不同类型数据进行运算。 print(“1”+1)

-- 索引异常(IndexError)：超出索引范围。

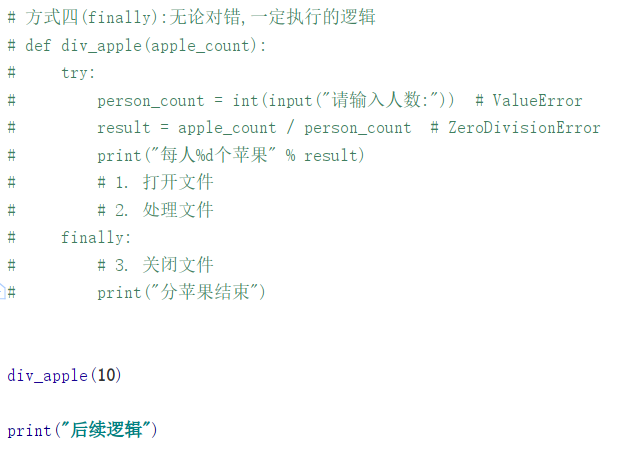
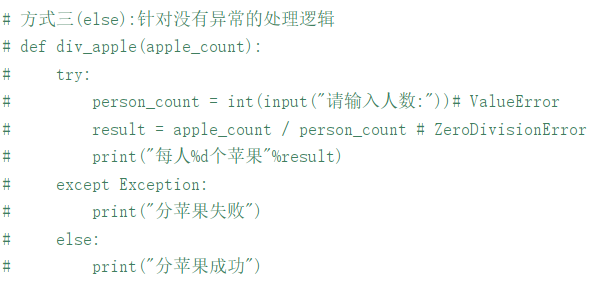
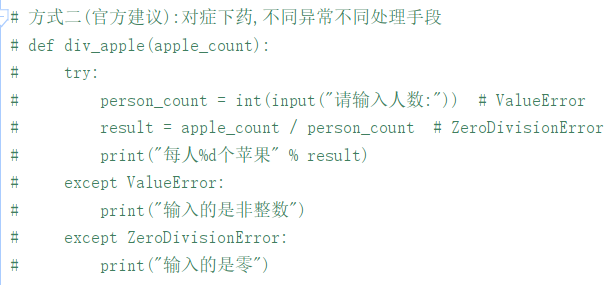
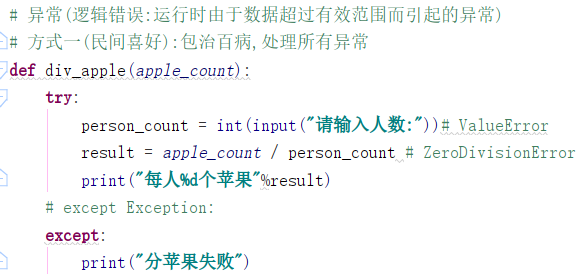
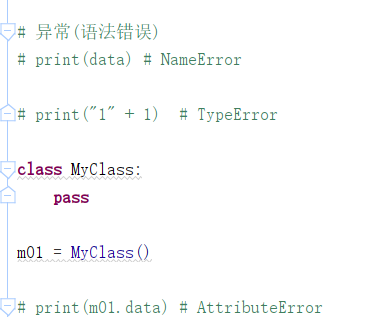
-- 属性异常(AttributeError)：对象没有对应名称的属性。

-- 键异常(KeyError)：没有对应名称的键。

-- 异常基类Exception。

4.

**常见的类型**：



## 处理

1. 语法：

**try:**

可能触发异常的语句

except 错误类型1 [as 变量1]：

处理语句1

except 错误类型2 [as 变量2]：

处理语句2

**except Exception** [as 变量3]：

不是以上错误类型的处理语句

**else: #(except Exception 与 else互斥)**

未发生异常的语句

finally:

**无论是否发生异常的语句**

1. 作用：将程序由异常状态转为正常流程。
2. 说明：

as 子句是用于绑定错误对象的变量，可以省略

except子句可以有一个或多个，用来捕获某种类型的错误。

else子句最多只能有一个。

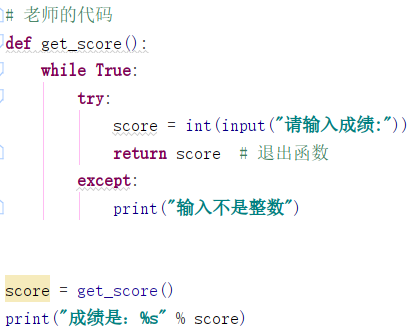
finally子句最多只能有一个，如果没有except子句，必须存在。

如果异常没有被捕获到，会向上层(调用处)继续传递，直到程序终止运行。

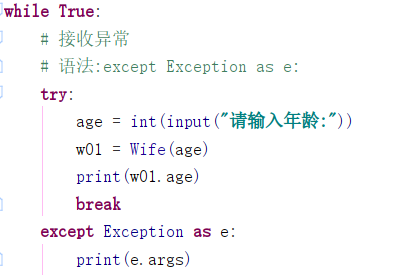
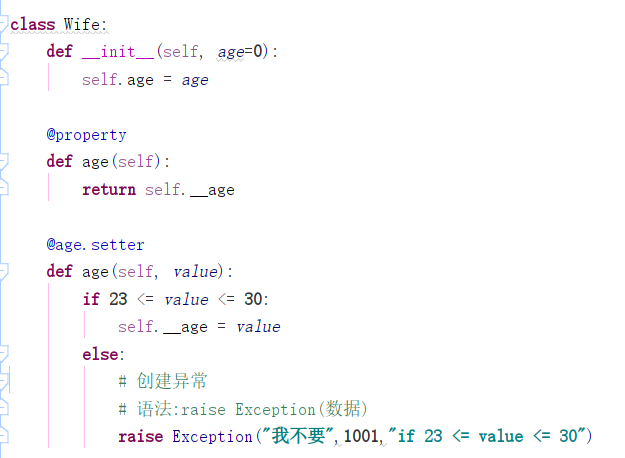
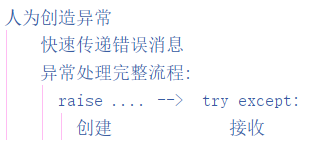
练习：创建函数，在终端中录入int类型成绩。如果格式不正确，重新输入。

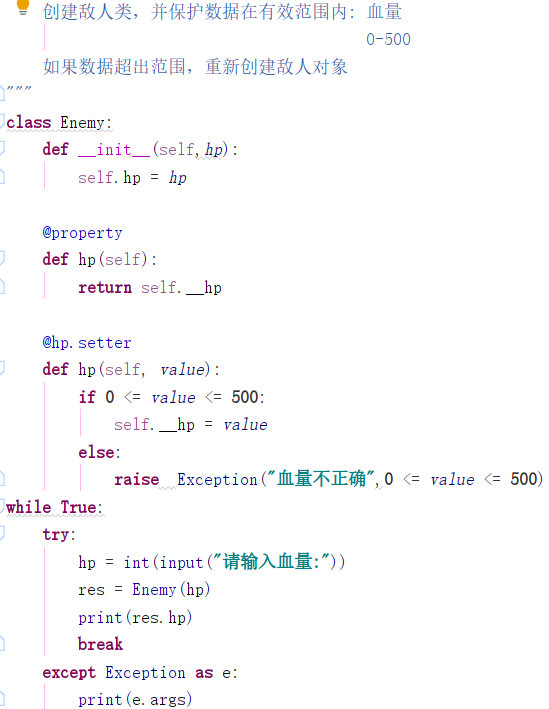
效果： score = get\_score()

print("成绩是：%d"%score)



## raise 语句

1. 作用：抛出（创造）一个错误，让程序进入异常状态。
2. 目的：在程序调用层数较深时，向主调函数传递错误信息要层层return 比较麻烦，所以人为抛出异常，可以直接传递错误信息。
3. 
4. **举例：**

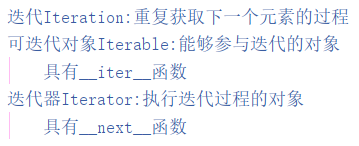


# 迭代iteration

每一次对过程的重复称为一次“迭代”，而每一次迭代得到的结果会作为下一次迭代的初始值。例如：循环获取容器中的元素。

**面试题:能够被for循环的对象,必须具备什么条件?**

**答:对象必须具备\_\_iter\_\_函数(必须是可迭代对象)**

****

## 可迭代对象iterable

1. 定义：具有**\_\_iter\_\_**函数的对象，可以返回迭代器对象。
2. 语法

-- 创建：

class 可迭代对象名称:

  def \_\_iter\_\_(self):

      return 迭代器

-- 使用：

for 变量名 in 可迭代对象:

语句

1. 原理：

迭代器 = 可迭代对象.\_\_iter\_\_()

while True:

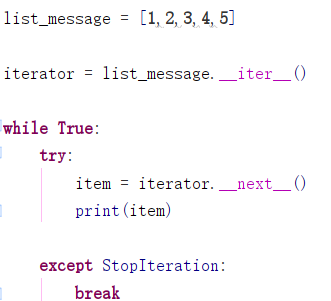
try:

print(迭代器.\_\_next\_\_())

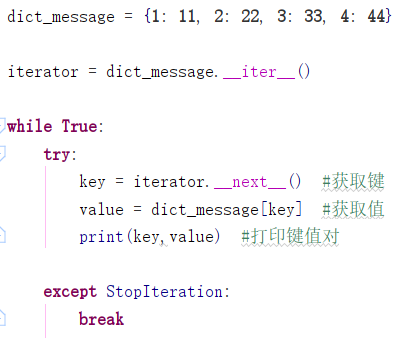
except StopIteration:

break

练习1：创建列表,使用迭代思想,打印每个元素.



练习2：创建字典,使用迭代思想,打印每个键值对.



## 迭代器对象iterator

1. 定义：可以被next()函数调用并返回下一个值的对象。
2. 语法

class 迭代器类名:

def \_\_init\_\_(self, 聚合对象):

self.聚合对象= 聚合对象

def \_\_next\_\_(self):

if 没有元素:

raise StopIteration

return 聚合对象元素

1. 说明：

-- 聚合对象通常是容器对象。

4. 作用：使用者只需**通过一种方式**，便可简洁明了的**获取聚合对象中各个元素**，而又**无需了解**其**内部结构**。

练习1：遍历商品控制器

class CommodityController:

pass

controller = CommodityController()

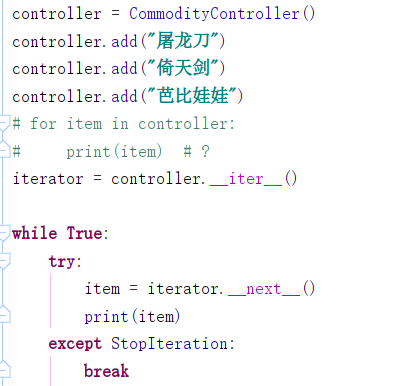
controller.add\_commodity("屠龙刀")

controller.add\_commodity("倚天剑")

controller.add\_commodity("芭比娃娃")

for item in controller:

print(item)



练习2：遍历图形控制器

class GraphicController:

pass

controller = CommodityController()

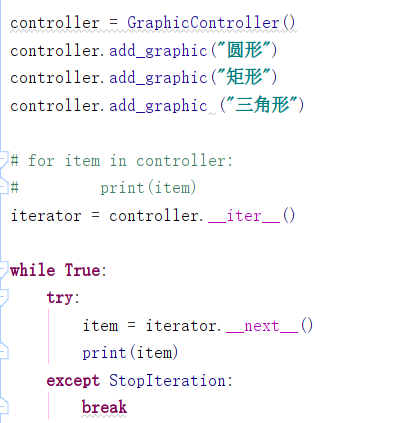
controller.add\_graphic("圆形")

controller.add\_graphic("矩形")

controller.add\_graphic ("三角形")

for item in controller:

print(item)



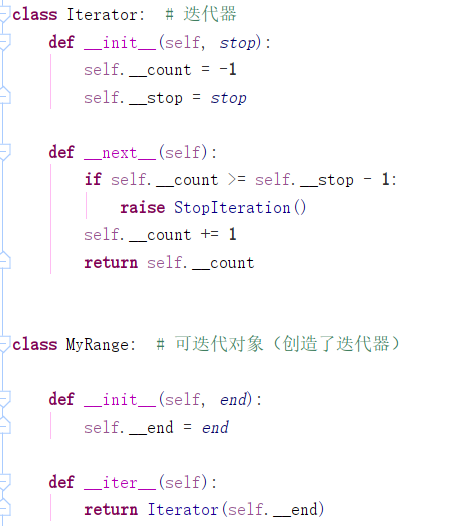
练习3：创建自定义range类，实现下列效果.

class MyRange:

pass

for number in MyRange(5):

print(number)# 0 1 2 3 4



# 生成器generator（推算满足体条件的元素）

1. 定义：能够动态(**循环一次计算一次返回一次**)提供数据的可迭代对象。
2. 作用：在循环过程中，按照某种算法**推算**数据，不必创建容器存储完整的结果，从而**节省内存空间**。数据量越大，优势越明显。
3. 以上作用也称之为**延迟操作**或**惰性操作**(**核心/特点**)，通俗的讲就是在需要的时候才计算结果，而不是一次构建出所有结果。

## 生成器函数(本质:迭代器)

1. 定义：含有yield语句的函数，返回值为**生成器对象**。
2. 语法

-- 创建：

def 函数名():

…

yield 数据

…

-- 调用：

**for** 变量名 **in** 函数名():

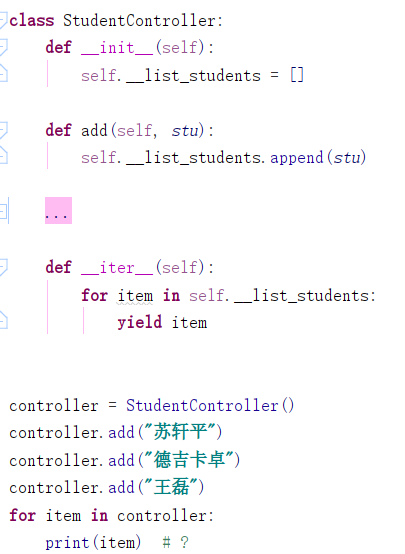
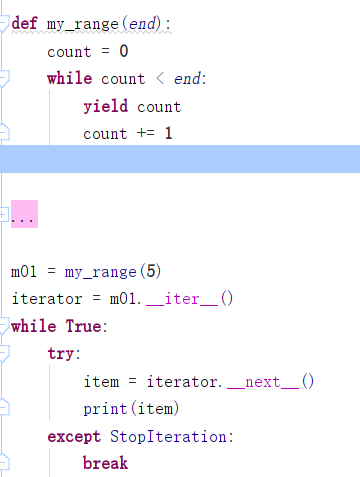
语句

**容器名**(函数名()) -->惰性操作转为立即操作

1. 说明：

-- 调用生成器函数将返回一个生成器对象，不执行函数体。

-- yield翻译为”产生”或”生成”

1. 执行过程：
2. 调用**生成器函数**会自动创建**迭代器对象**。
3. **调用**迭代器对象的**\_\_next\_\_()方法**时**才执行生成器函数**。
4. 每次执行到yield语句时返回数据，**暂时**离开。
5. 待下次调用**\_\_next\_\_()**方法时继续**从离开处**继续执行。
6. 
7. 
8. 原理：生成迭代器对象的大致规则如下

-- 将yield关键字以前的代码放在next方法中。

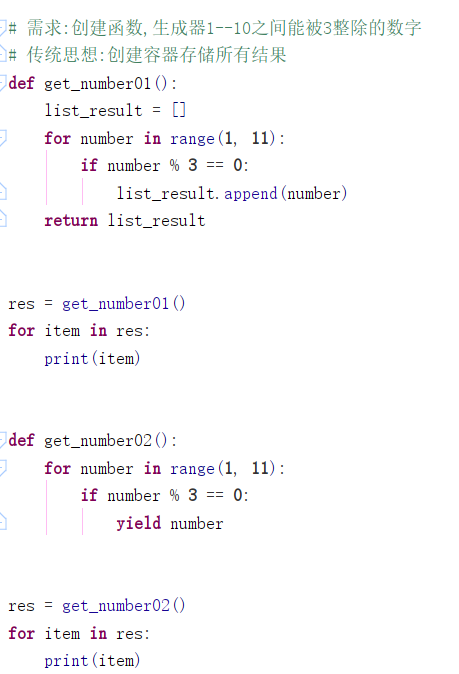
-- 将yield关键字后面的数据作为next方法的返回值。

1. **生成器应用:**

函数返回**多个结果**,不用容器存储.

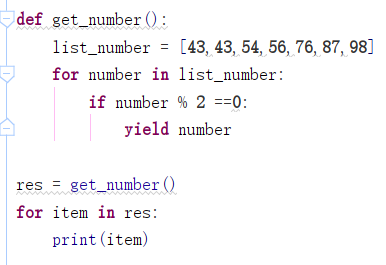
使用yield返回.

备注:函数有**一个结果**,使用**return**返回



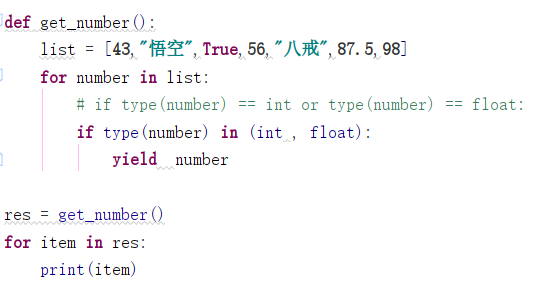
练习1：定义函数,在列表中找出所有偶数

[43,43,54,56,76,87,98]



练习2. 定义函数,在列表中找出所有数字

[43,"悟空",True,56,"八戒",87.5,98]



## 内置生成器

### 枚举函数enumerate --->内置生成器 ---->便捷,省内存

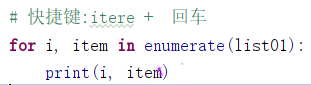
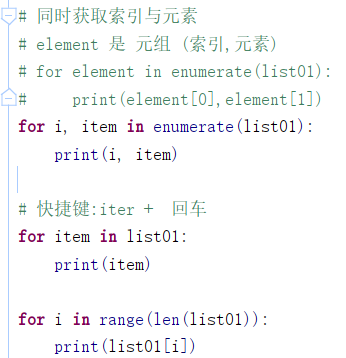
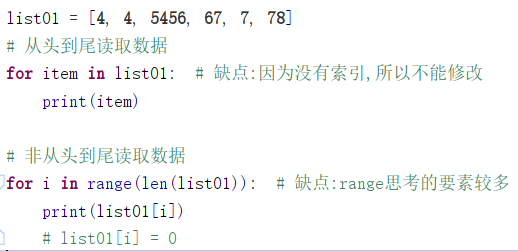
1. 语法：(**变量是元组，只不过拆包了**)

for 变量 in enumerate(可迭代对象):

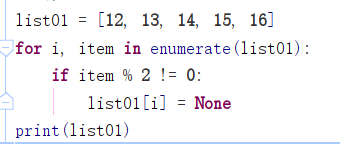
语句

for 索引, 元素in enumerate(可迭代对象):

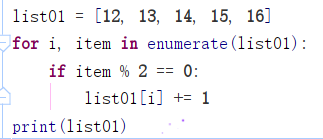
语句

1. 作用：遍历可迭代对象时，可以将**索引**与**元素**组**合为一个元组**。
2. 

练习1：将列表中所有奇数设置为None



练习2：将列表中所有偶数自增1

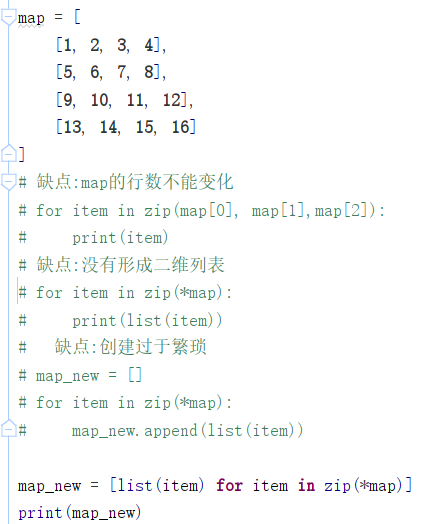
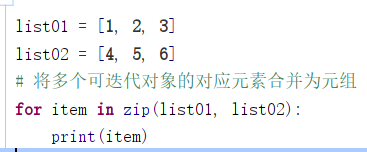


### Zip（压缩）

1. 语法：

for item in zip(可迭代对象1, 可迭代对象2….):

语句

1. 作用：将多个可迭代对象中对应的元素组**合成**一个个元组，生成的元组个数由最小的可迭代对象决定。
2. 

练习：使用学生列表封装以下三个列表中数据

list\_student\_name = ["悟空", "八戒", "白骨精"]

list\_student\_age = [28, 25, 36]

list\_student\_sex = ["男", "男", "女"]



## 生成器表达式

1. 定义：用推导式形式创建生成器对象。
2. 语法：变量 = ( 表达式 for 变量 in 可迭代对象 [if 真值表达式] )
3. **对比:列表**存储结果

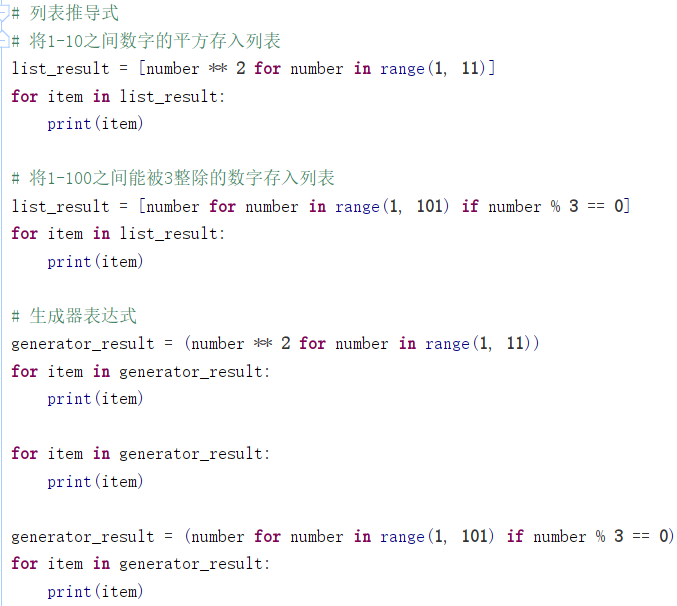
优点:获取数据灵活(索引切片,反复操作)

缺点:内存占用较大

**生成器**存储结果

优点:节省内存

缺点:获取数据不灵活(不能用索引切片,只能使用一次)



练习1：使用生成器表达式在列表中获取所有字符串.

list01 = [43, "a", 5, True, 6, 7, 89, 9, "b"]

练习2：在列表中获取所有整数,并计算它的平方.

# 函数式编程

1. 定义：用一系列函数解决问题。

-- 函数可以赋值给变量，赋值后变量绑定函数。

-- 允许将函数作为参数传入另一个函数。

-- 允许函数返回一个函数。

1. 高阶函数：将函数作为参数或返回值的函数。
2. 函数式编程 - **语法**

**函数**可以赋值给**变量**，赋值后**变量绑定函数**

1. **函数式编程思想:**

**解决的问题:**

多个函数,主体逻辑相同,核心算法不同

**思想:**

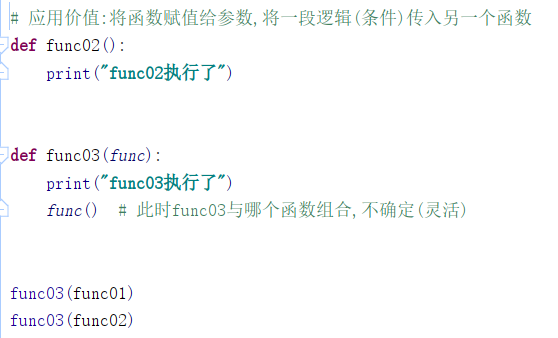
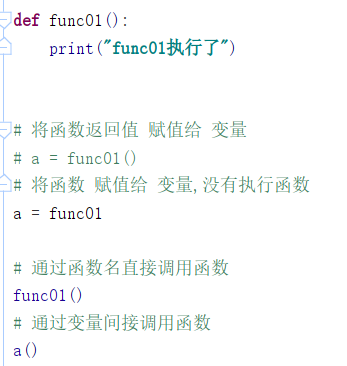
"封装"分:将不同的核心算法单独定义到函数中,

将主体逻辑也定义到单独函数中

"继承"隔:使用参数在主体逻辑函数中隔离不同的核心算法函数

"多态"做:将来再增加新的核心算法,

只需要按照在主体逻辑函数中使用参数的规则定义新函数

****

## 函数作为参数

将核心逻辑传入方法体，使该方法的适用性更广，体现了面向对象的开闭原则。

**能够传递判断条件，处理逻辑**

练习1：

需求：

定义函数，在列表中查找奇数

定义函数，在列表中查找能被3或5整除的数字

步骤：

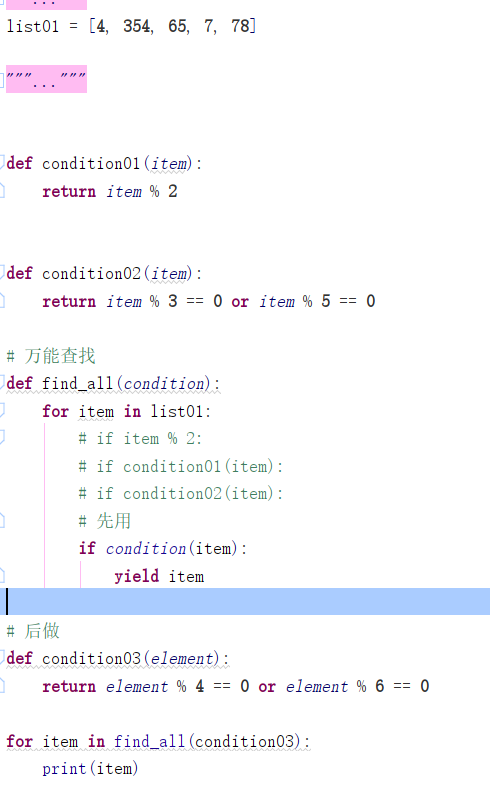
1. 根据需求，写出函数。

2. 因为主体逻辑相同,核心算法不同.

所以使用函数式编程思想(分、隔、做)

创建通用函数find\_all

1. 在当前模块中调用



练习2：

需求：

定义函数，在员工列表中查找编号是1003的员工

定义函数，在员工列表中查找姓名是孙悟空的员工

步骤：

1. 根据需求，写出函数。

2. 因为主体逻辑相同,核心算法不同.

所以使用函数式编程思想(分、隔、做)

创建通用函数find\_single

3. 在当前模块中调用

class Employee:

def \_\_init\_\_(self, eid, did, name, money):

self.eid = eid # 员工编号

self.did = did # 部门编号

self.name = name

self.money = money

# 员工列表

list\_employees = [

Employee(1001, 9002, "师父", 60000),

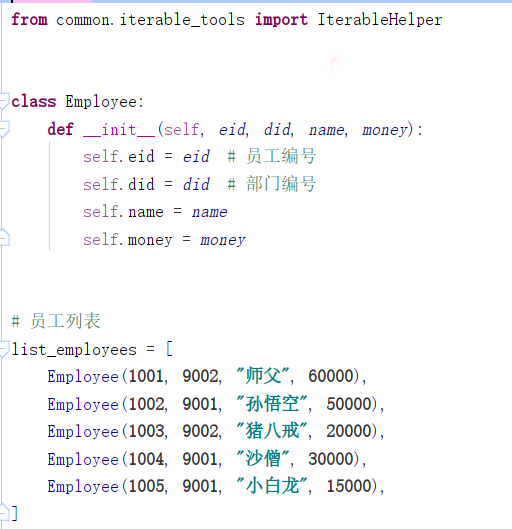
Employee(1002, 9001, "孙悟空", 50000),

Employee(1003, 9002, "猪八戒", 20000),

Employee(1004, 9001, "沙僧", 30000),

Employee(1005, 9001, "小白龙", 15000),

]



### lambda 表达式(函数名和之后的括号不要，return不要)

1. 定义：是一种**匿名**方法。**匿名**函数。
2. 作用：作为参数(**实参**)传递时**语法简洁**，**优雅**，**代码可读性强**。

**随时创建**和**销毁**，**减少程序耦合度**。

1. 语法

-- 定义：

变量 = lambda 形参: 方法体-->（方法体中选择的是return之后的语句或者包括print在内和其之后的语句。）

-- 调用：

变量(**实参**)

1. 说明：

-- 形参没有可以不填

-- 方法体**只能有一条语句**，-->lambda : for i in range(5):print(i) 会出错。 且**不支持赋值语句**。-->lambda p1:p1[0] = 1000 这个语句会出错。

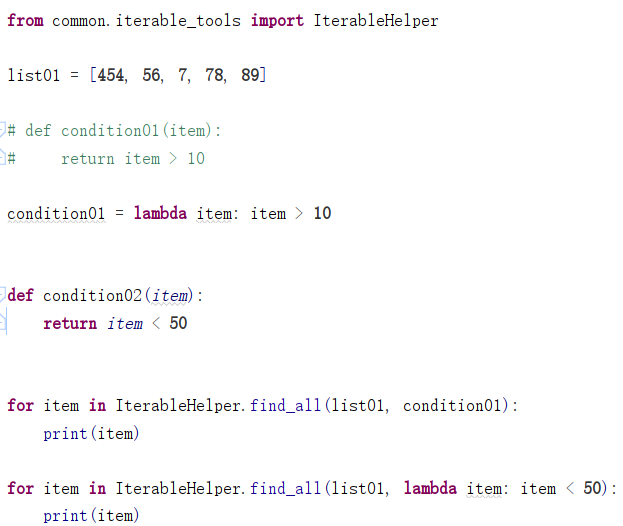
1. 函数中，没有return,不要用print。（因为没有return，返回的是true/false,这个时候不需要打印）
2. **lambda**与**def函数**对比

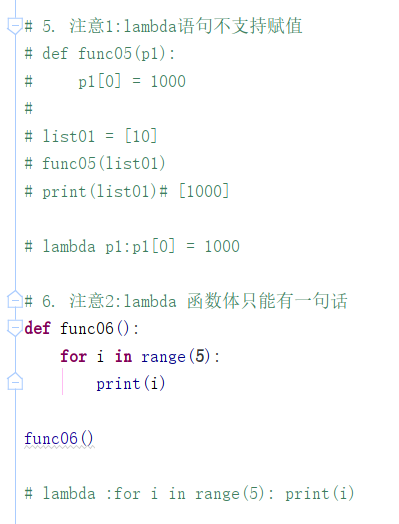
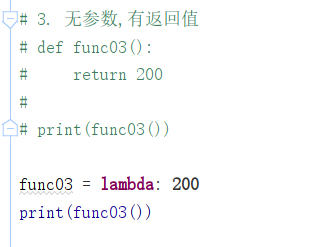
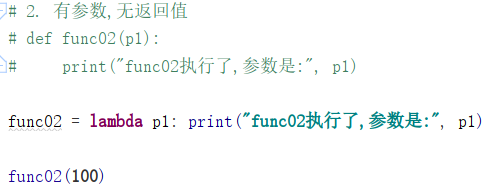
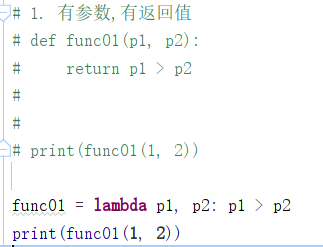
匿名函数可以完成的任务def函数都能完成

lambda语句**不支持赋值**

lambda 函数体**只能有一句话**

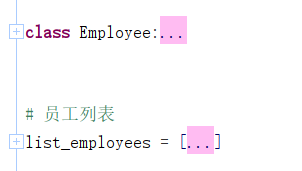
1. lambda **应用价值: 作为函数的实参**

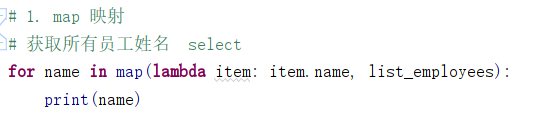
****

****

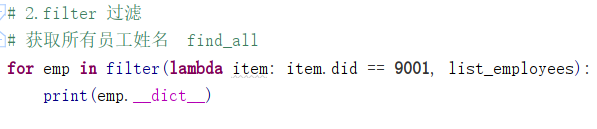
### 内置高阶函数（ctrl + p来查看自己内置高阶函数后（）里需要填的内容）

1. **Map(映射)**（**函数**，**可迭代对象**）：使用可迭代对象中的每个元素调用函数，将返回值作为新可迭代对象元素；返回值为新可迭代对象。**相当于我们自己的做的自定义函数中的select。返回的是生成器（直接用List也可以的，将惰性操作转为立即操作。）**





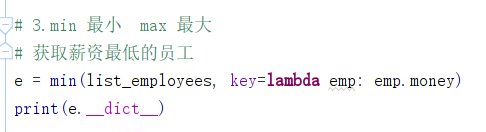
1. **Filter（过滤）**(**函数**，**可迭代对象**)：根据条件筛选可迭代对象中的元素，返回值为新可迭代对象。**相当于我们自己的做的自定义函数中的find\_all**



**Sorted（排序）**(**可迭代对象**，**key = 函数,reverse = bool值**)：**排序**，返回值为排序结果,返回值为新可迭代对象。Bool值选择：True （如用了True，就是降序操作）。**相当于我们自己的做的自定义函数中的descending\_order（降序）和aescending\_order（升序），内存占有率较大。**

****

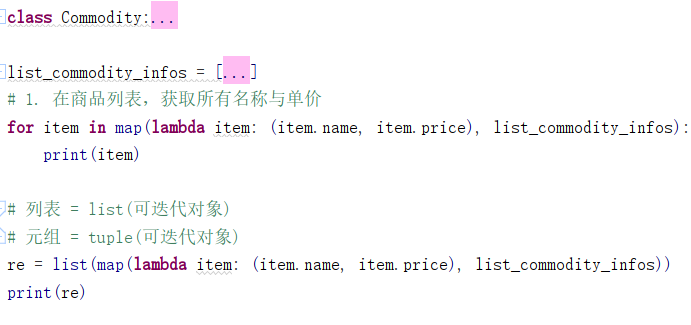
1. **max**(**可迭代对象**，**key = 函数**)：根据函数获取可迭代对象的最大值。

****

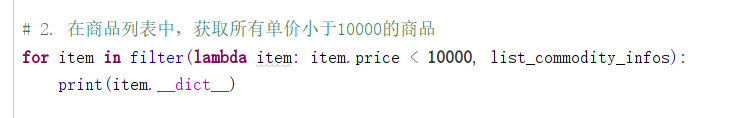
1. **min**(**可迭代对象**，**key = 函数**)：根据函数获取可迭代对象的最小值。

练习：

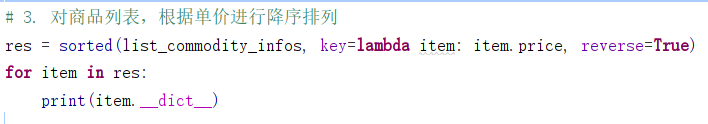
1. 在商品列表，获取所有名称与单价



1. 在商品列表中，获取所有单价小于10000的商品

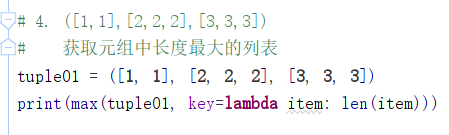


1. 对商品列表，根据单价进行降序排列

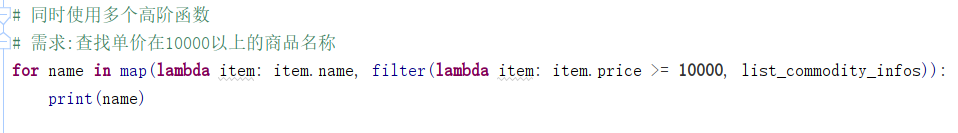


4. ([1,1],[2,2,2],[3,3,3])

获取元组中长度最大的列表



1. 查找单价在10000以上的商品名称



class Commodity:

def \_\_init\_\_(self, cid=0, name="", price=0):

self.cid = cid

self.name = name

self.price = price

list\_commodity\_infos = [

Commodity(1001, "屠龙刀", 10000),

Commodity(1002, "倚天剑", 10000),

Commodity(1003, "金箍棒", 52100),

Commodity(1004, "口罩", 20),

Commodity(1005, "酒精", 30),

]

## 函数作为返回值

**逻辑连续**，当内部函数被调用时，不脱离当前的逻辑。

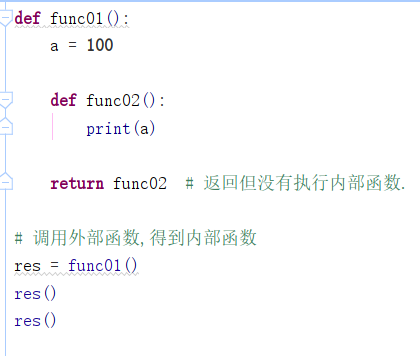
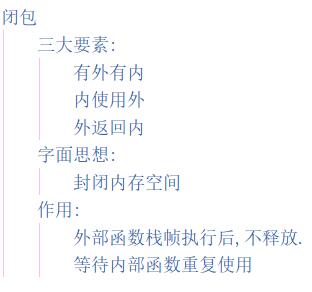
### 闭包

1. 三要素：

-- 必须有一个内嵌函数。

-- 内嵌函数必须引用外部函数中变量。

-- 外部函数返回值必须是内嵌函数。



1. 语法

-- 定义：

def 外部函数名(参数):

外部变量

def 内部函数名(参数):

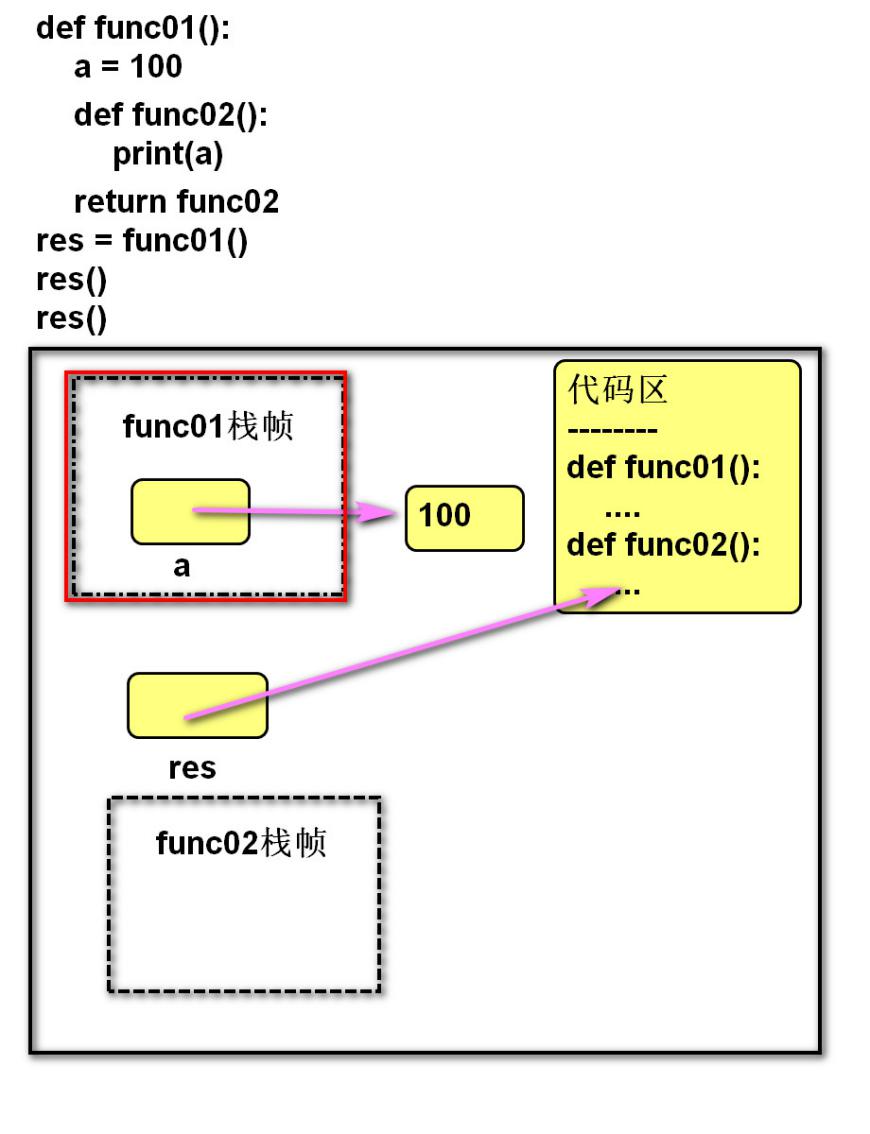
使用外部变量

return 内部函数名

-- 调用：

变量 = 外部函数名(参数)

变量(参数)

1. 定义：是由函数及其相关的引用环境组合而成的实体。
2. **优点**：内部函数可以**使用**外部变量。
3. **缺点**：外部变量一直存在于**内存**中，不会在调用结束后释放，**占用内存**。
4. 作用：实现python装饰器。
5. 逻辑连续(如：从得1000元,到不断购买商品的过程,不中断.)
6. 
7. **闭包内存图**
8. 

练习：使用闭包模拟以下情景：

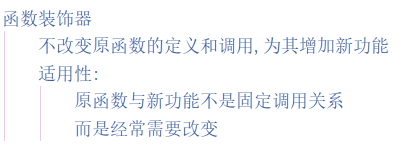
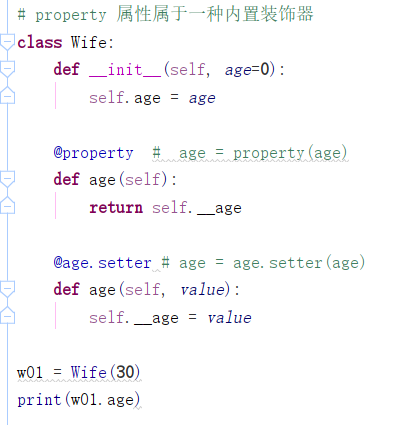
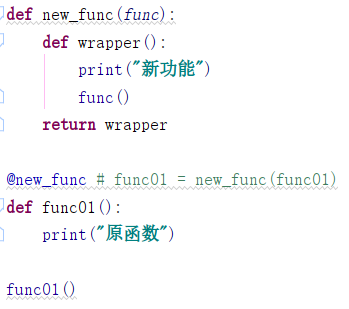
在银行开户存入10000

购买xx商品花了xx元

购买xx商品花了xx元



### 函数装饰器decorator

1. 定义：在**不改变**原函数的**调用**以及内部代码情况下，为其**添加新功能**的函数。
2. 
3. 
4. property 属性属于一种内置装饰器
5. 
6. 语法

def 函数装饰器名称(func):

def wrapper(\*args, \*\*kwargs):

需要添加的新功能

return func(\*args, \*\*kwargs)

return wrapper

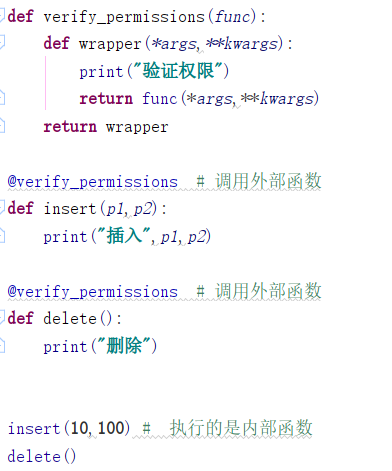
原函数 = 内嵌函数

@ 函数装饰器名称

def 原函数名称(参数):

函数体

原函数(参数)



1. **本质**：使用“@函数装饰器名称”修饰原函数，等同于创建与原函数名称相同的变量，关联内嵌函数；故调用原函数时执行内嵌函数。

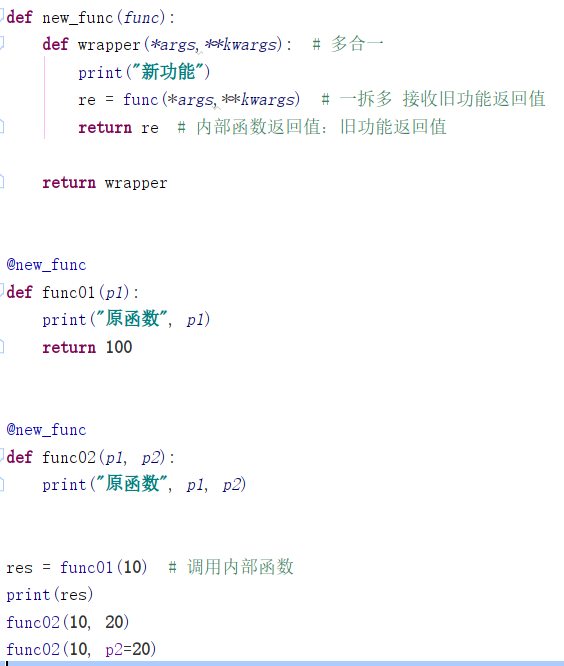
原函数名称 = 函数装饰器名称（原函数名称）

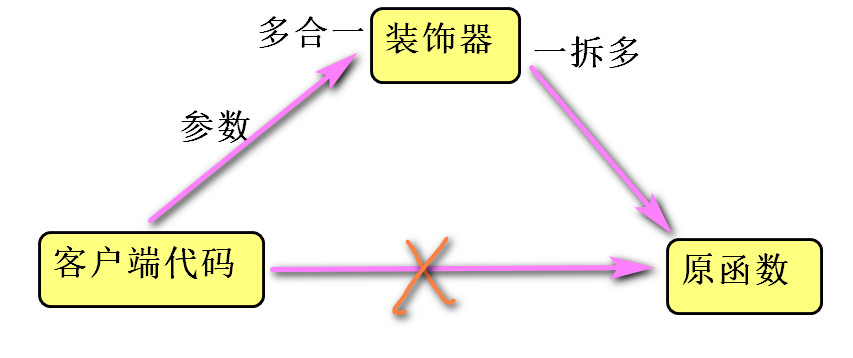
1. **装饰器链**：

一个函数可以被多个装饰器修饰，**执行顺序为从近到远**。

9.**函数装饰器-细节**

**内部函数返回值**：**旧功能返回值**

****

1. **装饰器参数设计思想**
2. 

练习1：不改变插入函数与删除函数代码，为其增加验证权限的功能

def verify\_permissions(func):

print("验证权限")

def insert():

print("插入")

def delete():

print("删除")

insert()

delete()



使用**装饰器**实现：



练习2：为sum\_data,增加打印函数执行时间的功能.

函数执行时间公式： 执行后时间 - 执行前时间

def sum\_data(n):

sum\_value = 0

for number in range(n):

sum\_value += number

return sum\_value

print(sum\_data(10))

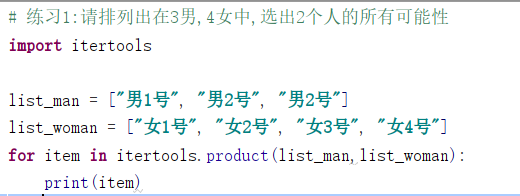
print(sum\_data(1000000))



**新的知识点**

1. 笛卡尔积(需要导入模块：import itertools)-->product





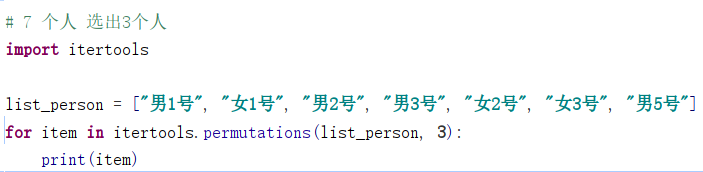
1. 排列-->permutations

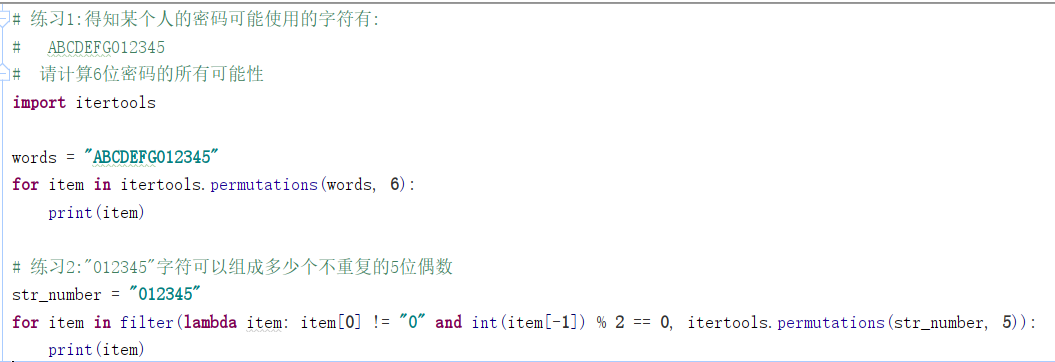
从n个元素中抽出m个元素,并按照顺序排列.

公式:n! / (n-m)!

7\*6\*5\*4\*3\*2\*1 / 4 \*3 \* 2 \* 1

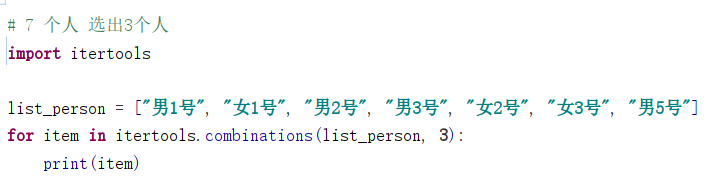
7\*6\*5

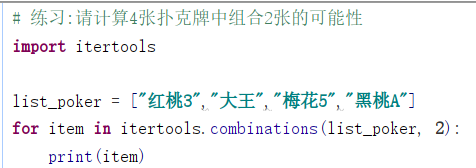




1. 组合-->combinations

从n个元素中抽出m个元素





1. Enclosing 外部嵌套作用域

内部函数能读取外部嵌套变量

必须通过nonlocal声明才能修改

