# 模块 Module

## 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
2. 使用：模块名.成员

*"""*

*模块调用*

*调用module01.py的成员*

*练习:下载double\_list\_helper.py模块*

*新建exercise01.py模块,以3种方式,测试工具类中的方法.*

*"""*

*# 导入方式1:*

*# 导入 模块名称*

*# 本质:将该模块作用域 赋值给 变量* *module01*

**import** module01

module01.fun01()

c01 = module01.MyClass01()

c01.fun02()

module01.\_fun03()

### from import

1. 语法：

from 模块名 import 成员名[ as 别名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。

*# 导入方式2:*

*# 本质:将该模块指定成员 赋值给 变量* *fun01,MyClass01*

**from** module01 **import** fun01,MyClass01,\_fun03

fun01()

c01 = MyClass01()

c01.fun02()

\_fun03()

### 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。
3. 模块中以下划线(\_)开头的属性，不会被导入，通常称这些成员为隐藏成员。

*# 导入方式3:*

**from** module01 **import** \*

fun01()

c01 = MyClass01()

c01.fun02()

\_fun03()

**模块变量**

\_\_all\_\_变量：定义可导出成员，仅对from xx import \*语句有效。

\_\_doc\_\_变量：文档字符串。

\_\_file\_\_变量：模块对应的文件路径名。

\_\_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。

module01.py

*"""*

*文档字符串*

*"""*

*# 可以导出*

\_\_all\_\_ = [**"fun01"**,**"fun03"**,**"MyClass01"**]

*# 只在第一次被导入时执行*

print(**"我是module01"**)

**def** fun01():

print(**"fun01"**)

**class** MyClass01:

**def** fun02(self):

print(**"fun02"**)

**def** fun03():

print(**"fun03"**)

**def** fun04():

print(**"fun04"**)

**from** module01 **import** \*

fun01()

fun03()

*# 获取模块文档注释*

**import** my\_exercise01 **as** me

*# import text*

print(me.\_\_doc\_\_)

*# 获取模块文件路径*

print(me.\_\_file\_\_)

*# 获取模块名称*

print(me.\_\_name\_\_) *#模块名* *\_\_main\_\_*

## 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。

## 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。
3. 第三方模块（通常为开源），需要自己安装。
4. 用户自己编写的模块（可以作为其他人的第三方模块）

*"""*

*标准库模块* *-- 时间*

*练习:exercise02*

*"""*

**import** time

*# 返回时间戳(1970年后经过的浮点秒数)*

*# 1555579061.7284493*

print(time.time())

*# 时间戳* *--> 时间元组(年 月 日 时 分 秒 星期 一年的第几天 夏令时)*

print(time.localtime(1555579061.7284493))

*# 时间元组* *--> 时间戳*

print(time.mktime(time.localtime()))

*# 时间元组* *--> 字符串* *(时间的格式化)*

*# time.localtime() --> str*

print(time.strftime(**"%Y %m %d %H:%M:%S"**, time.localtime()))

print(time.strftime(**"%y %m %d %H:%M:%S"**, time.localtime()))

*# 字符串* *--> 时间元组*

print(time.strptime(**"2019 04 14"**, **"%Y %m %d"**)

## 搜索顺序

搜索内建模块(builtins)

sys.path 提供的路径，通常第一个是程序运行时的路径。

# 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。

## 导入

from 包名 import 模块名 [as 模块新名]

from 包名.子包名 import 模块名 [as 模块新名]

from 包名.子包名.模块名 import 成员名 [as 属性新名]

# 导入包内的所有子包和模块

from 包名 import \*

from 包名.模块名 import \*

*"""*

*包(文件夹)*

*# 练习1: 目标* *-- 在实际项目结构中,随心所欲的调用不同模块成员.*

*# 1. 根据目录结构,创建相应的包与模块.*

*# 2. 在相应模块中,定义函数/类.*

*# 3. 在main中,调用skill\_manager模块中成员*

*# 4. 在skill\_manager模块中调用skill\_deployer模块成员*

*# 5. 在模块skill\_data中,调用list\_helper模块成员.*

*练习2:创建如下结构:*

*project*

*package01*

*-- m01.py*

*-- exercise01.py*

*main.py*

*在exercise01.py 中调用m01模块的成员.*

*测试:在main中开始运行*

*在exercise01中开始运行*

*要求:使用终端*

*"""*

*# python程序结构*

*# 包*

*# 模块*

*# 类*

*# 函数()/方法()*

*# 语句*

*# # 方式1:import 包.模块*

*# import package01.module01*

*# # 使用:包.模块.成员*

*# package01.module01.fun01()*

*# # 方式2:from 包* *import 模块*

*# from package01 import module01*

*# # 使用:模块.成员*

*# module01.fun01()*

*# # 方式3:*

*# from package01 import \**

*# # 在包的\_\_init\_\_文件中,定义\_\_all\_\_ 属性*

*# module01.fun01()*

*# [推荐]*

*# from 包.模块* *import 成员*

**from** package01.module01 **import** \*

fun01()

## 搜索顺序

sys.path 提供的路径

## \_\_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

### \_\_all\_\_

记录from 包 import \* 语句需要导入的模块

案例：

my\_ project /

\_\_init\_\_.py

main.py

common/

\_\_init\_\_.py

double\_list\_helper.py

list\_helper.py

sill\_system/

\_\_init\_\_.py

skill\_data.py

skill\_deployer.py

skill\_manager.py

# 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。

-- 类型异常(TypeError)：不同类型数据进行运算。

-- 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。

-- 为实现异常(NotImplementedError)：尚未实现的方法。

-- 异常基类Exception。

## 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:

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。

## raise 语句

1. 作用：抛出一个错误，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。。

*"""*

*异常处理*

*练习1:exercise02*

*练习2: 将学生管理系统中可能出错的代码(转换为整数的代码),进行异常处理*

*"""*

**def** div\_apple(apple\_count):

*# 分苹果*

person\_count = int(input(**"请输入人数:"**)) *# ValueError*

result = apple\_count / person\_count *# ZeroDivisionError*

print(**"每个人分类%d个苹果"** % result)

*# 缺点:不能根据具体错误,做出相应处理逻辑.*

*# try:*

*# div\_apple(10)*

*# except Exception as e: # 捕获所有类型的异常*

*# print("分苹果失败啦")*

*# try:*

*# div\_apple(10)*

*# except ValueError: # 捕获ValueError类型的异常*

*# print("输入的人数有误")*

*# except ZeroDivisionError: # 捕获ZeroDivisionError类型的异常*

*# print("人数不能是零")*

*# except Exception as e: # 捕获所有类型的异常*

*# print("未知类型的错误")*

**try**:

div\_apple(10)

**except** ValueError **as** e: *# 捕获ValueError类型的异常*

print(**"输入的人数有误"**)

**except** ZeroDivisionError: *# 捕获ZeroDivisionError类型的异常*

print(**"人数不能是零"**)

**except** Exception **as** e: *# 捕获所有类型的异常*

print(**"未知类型的错误"**)

**else**: *# 没有发生异常,执行的逻辑*

print(**"分苹果成功喽"**)

**finally**:

print(**"无论是否发生异常,都执行的逻辑"**)

print(**"后续逻辑......."**)

## 自定义异常

1. 定义：

class 类名Error(Exception):

def \_\_init\_\_(self,参数):

super().\_\_init\_\_(参数)

self.数据 = 参数

1. 调用：

try:

….

raise 自定义异常类名(参数)

….

except 定义异常类 as 变量名:

变量名.数据

1. 作用：封装错误信息

*"""*

*自定义异常*

*练习:成绩异常(1-100)*

*"""*

**class** AgeError(Exception):

*"""*

*封装错误信息*

*"""*

**def** \_\_init\_\_(self,msg,code,age\_value):

super().\_\_init\_\_(msg)

self.msg = msg

self.code = code

self.age\_value = age\_value

**class** Wife:

**def** \_\_init\_\_(self,age):

self.age = age

@property

**def** age(self):

**return** self.\_\_age

@age.setter

**def** age(self,value):

**if** 20 <= value <= 30:

self.\_\_age = value

**else**:

*# print("我不要")*

*# raise ValueError("我不要") # 人为抛出异常*

**raise** AgeError(**"我不要"**,27,value)

**try**:

w01 = Wife(80)

print(w01.age)

**except** AgeError **as** e:

print(**"错误信息:"**,e.msg)

print(**"错误代码行号:"**,e.code)

print(**"输入的年龄是:"**,e.age\_value)

*# 练习:定义方法,在控制台中获取成绩int(1 -- 100)*

*# 要求:如果输入有误,重新输入.*

**def** get\_score():

**while True**:

**try**:

number = int(input(**"请输入成绩:"**))

*# except ValueError:*

**except**:

print(**"输入有误"**)

**continue** *# 跳过本次循环*

**if** 1<= number <= 100:

**return** number

print(**"成绩不在范围内"**)

get\_score()

# 迭代

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。

## 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

## 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。

*"""*

*可迭代对象：具有\_\_iter\_\_()方法，可以返回迭代器的对象．*

*"""*

list01 = [2,434,5,6,8]

*# for item in list01:*

*# print(item)*

*# 面试题：*

*# 能够被for循环的条件是：可迭代对象（具有\_\_iter\_\_()方法的对象）*

*# for循环原理*

*# 1. 获取迭代器对象*

*# 2. 循环迭代(调用迭代器的\_\_next\_\_方法)*

*# 3. 捕获StopIteration异常*

*#1.*

*# iterator = list01.\_\_iter\_\_()*

*#2.*

*# item = iterator.\_\_next\_\_()*

*# print(item)*

*#*

*# item = iterator.\_\_next\_\_()*

*# print(item)*

*#*

*# item = iterator.\_\_next\_\_()*

*# print(item)*

*#*

*# item = iterator.\_\_next\_\_()*

*# print(item)*

*#*

*# item = iterator.\_\_next\_\_() # 最后一次*

*# print(item)*

*# 3.*

*# item = iterator.\_\_next\_\_()# StopIteration*

*# print(item)*

*# 1.　获取迭代器对象*

iterator = list01.\_\_iter\_\_()

**while True**:

**try**:*# 如果获取了全部元素,则执行except*

*# 2.　获取下一个元素(迭代过程)*

item = iterator.\_\_next\_\_()

print(item)

*# 3.停止迭代（StopIteration　错误）*

**except** StopIteration:

**break** *# 跳出循环体*

*"""*

*迭代器*

*练习：exercise02/exercise03*

*"""*

**class** Skill:

**pass**

**class** SkillIterator:

*"""*

*技能迭代器*

*"""*

**def** \_\_init\_\_(self, target):

self.target = target

self.index = 0

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

*# 如果索引越界　则抛出异常*

**if** self.index > len(self.target) - 1:

**raise** StopIteration()

*# 返回下一个元素*

item = self.target[self.index]

self.index += 1

**return** item

**class** SkillManager:

*"""*

*可迭代对象*

*"""*

**def** \_\_init\_\_(self, skills):

self.skills = skills

**def** \_\_iter\_\_(self):

*# 创建迭代器对象 传递　需要迭代的数据*

**return** SkillIterator(self.skills)

*#---------------以下是客户端代码---------------------*

manager = SkillManager([Skill(), Skill(), Skill()])

*# for item in manager.skills:*

**for** item **in** manager:*# 获取manager对象中列表元素(获取manager对象的聚合类型对象元素)*

print(item)

*# iterator = manager.\_\_iter\_\_()*

*# while True:*

*# try:*

*# item = iterator.\_\_next\_\_()*

*# print(item)*

*# except:*

*# break*

*"""*

*迭代器* *--> yield*

*练习：exercise04/exercise05*

*"""*

**class** Skill:

**pass**

*# class SkillIterator:*

*# """*

*# 技能迭代器*

*# """*

*# def \_\_init\_\_(self, target):*

*# self.target = target*

*# self.index = 0*

*#*

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

*# if self.index > len(self.target) - 1:*

*# raise StopIteration()*

*# item = self.target[self.index]*

*# self.index += 1*

*# return item*

**class** SkillManager:

**def** \_\_init\_\_(self, skills):

self.skills = skills

*# def \_\_iter\_\_(self):*

*# return SkillIterator(self.skills)*

**def** \_\_iter\_\_(self):

*"""*

*执行过程：*

*１．调用\_\_iter\_\_()方法不执行*

*２．调用\_\_next\_\_()方法时执行，到yield语句暂时离开．*

*３．再次调用\_\_next\_\_()方法时，从上次离开的代码开始执行，到yield语句暂时离开*

*4. 待执行完方法体，抛出StopIteration异常．*

*原理：如果方法体中包含yield关键字，那么会自动生成迭代器对象．*

*生成迭代器代码的大致规则：*

*1. 将yield关键字以前的代码，放到\_\_next\_\_方法中．*

*2. 将yield关键字以后的数据，作为\_\_next\_\_方法的返回值*

*"""*

*# print("准备返回第一个元素")*

*# yield self.skills[0] # 暂时离开点　　　再次执行点*

*#*

*# print("准备返回第二个元素")*

*# yield self.skills[1]*

*#*

*# print("准备返回第三个元素")*

*# yield self.skills[2]*

**for** item **in** self.skills:

**yield** item

*#---------------以下是客户端代码---------------------*

manager = SkillManager([Skill(), Skill(), Skill()])

*# for item in manager:*

*# print(item)*

iterator = manager.\_\_iter\_\_()

**while True**:

**try**:

item = iterator.\_\_next\_\_()

print(item)

**except** Exception **as** e:

print(e)

**break**

***#*** *练习１：****("****悟空****","****八戒****","****沙僧****","****唐僧****")***

*# 使用while + 迭代器　获取元组所有元素*

t01 = (**"悟空"**, **"八戒"**, **"沙僧"**, **"唐僧"**)

iterator = t01.\_\_iter\_\_()

**while True**:

**try**:

item = iterator.\_\_next\_\_()

print(item)

**except**:

**break**

*# 练习２：{"张三丰"：101,"张无忌":102,"赵敏":102}*

*# 不使用for循环，获取字典所有元素．*

d01 = {**"张三丰"**: 101, **"张无忌"**: 102, **"赵敏"**: 102}

iterator = d01.\_\_iter\_\_()

**while True**:

**try**:

key = iterator.\_\_next\_\_()

print(key, d01[key])

**except**:

**break**

# 生成器generator

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，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

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

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. 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。

*"""*

*yield　－－＞　　生成器函数*

*"""*

*# class MyRange:*

*# def \_\_init\_\_(self, stop):*

*# self.stop = stop*

*#*

*# def \_\_iter\_\_(self):*

*# start = 0*

*# while start < self.stop:*

*# yield start*

*# start += 1*

*#*

*# for item in MyRange(5):*

*# print(item)*

*# 生成器函数*

**def** my\_range(stop):

start = 0

**while** start < stop:

**yield** start

start += 1

*# for item in my\_range(5):*

*# print(item)*

iter01 = my\_range(5)

**for** item **in** iter01:

print(item)

*"""*

*练习：迭代员工管理器对象*

*"""*

**class** Employee:

**pass**

**class** EmployeeIterator:

*"""*

*迭代器*

*"""*

**def** \_\_init\_\_(self, target):

self.target = target

self.index = 0

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

**if** self.index > len(self.target) - 1:

**raise** StopIteration()

item = self.target[self.index]

self.index += 1

**return** item

*# 可迭代对象*

**class** EmployeeManager:

**def** \_\_init\_\_(self, employees):

self.all\_employee = employees

**def** \_\_iter\_\_(self):

*# 返回迭代器*

**return** EmployeeIterator(self.all\_employee)

manager = EmployeeManager([Employee(), Employee()])

*# 需求：ｆｏｒ循环自定义类的对象*

*# for item in manager:*

*# print(item) #*

iterator = manager.\_\_iter\_\_()

**while True**:

**try**:

item = iterator.\_\_next\_\_()

print(item)

**except**:

**break**

*"""*

*参照下列代码，定义ＭｙＲａｎｇｅ类，实现相同效果．*

*"""*

**class** MyRangeIterator:

**def** \_\_init\_\_(self, stop):

self.stop = stop

self.start = 0

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

**if** self.start + 1 > self.stop:

**raise** StopIteration()

temp = self.start

self.start += 1

**return** temp

**class** MyRange:

**def** \_\_init\_\_(self, stop):

self.stop = stop

**def** \_\_iter\_\_(self):

*# 创建迭代器对象*

**return** MyRangeIterator(self.stop)

*# for item in range(5):*

*# print(item)#*

*# for item in MyRange(5):*

*# print(item)#0 1 2 3 4*

iterator = MyRange(5)

**for** item **in** iterator:

print(item) *#*

*# iterator = iterator.\_\_iter\_\_()*

*# while True:*

*# try:*

*# item = iterator.\_\_next\_\_()*

*# print(item)*

*# except:*

*# break*

*"""*

*改造为yield实现*

*"""*

**class** Employee:

**pass**

*# class EmployeeIterator:*

*# """*

*# 迭代器*

*# """*

*# def \_\_init\_\_(self, target):*

*# self.target = target*

*# self.index = 0*

*#*

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

*# if self.index > len(self.target) - 1:*

*# raise StopIteration()*

*# item = self.target[self.index]*

*# self.index += 1*

*# return item*

*# 可迭代对象*

**class** EmployeeManager:

**def** \_\_init\_\_(self, employees):

self.all\_employee = employees

*# def \_\_iter\_\_(self):*

*# # 返回迭代器*

*# return EmployeeIterator(self.all\_employee)*

**def** \_\_iter\_\_(self):

**for** item **in** self.all\_employee:

**yield** item

manager = EmployeeManager([Employee(), Employee()])

*# 需求：ｆｏｒ循环自定义类的对象*

*# for item in manager:*

*# print(item) #*

iterator = manager.\_\_iter\_\_()

**while True**:

**try**:

item = iterator.\_\_next\_\_()

print(item)

**except**:

**break**

*"""*

*改造为yield实现*

*"""*

*# class MyRangeIterator:*

*# def \_\_init\_\_(self, stop):*

*# self.stop = stop*

*# self.start = 0*

*#*

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

*# if self.start + 1 > self.stop:*

*# raise StopIteration()*

*# temp = self.start*

*# self.start += 1*

*# return temp*

**class** MyRange:

**def** \_\_init\_\_(self, stop):

self.stop = stop

*# def \_\_iter\_\_(self):*

*# # 创建迭代器对象*

*# return MyRangeIterator(self.stop)*

**def** \_\_iter\_\_(self):

start = 0

**while** start < self.stop:

**yield** start

start += 1

iterator = MyRange(5)

*# for item in iterator:*

*# print(item) #*

iterator = iterator.\_\_iter\_\_()

**while True**:

**try**:

item = iterator.\_\_next\_\_()

print(item)

**except**:

**break**

*"""*

*生成器函数练习*

*体会：方法／函数，需要向外返回多个结果时，使用生成器函数．*

*惰性操作/延迟操作　(生成器函数的"循环(next)一次,计算一次,返回一次")*

*"""*

list01 = [23,3,4,556,677,68,8,98,98]

*# 练习１：在list01中，挑出所有偶数．*

*# 要求：1)使用生成器函数实现*

**def** get\_even01(target):

**for** item **in** target:

**if** item % 2 == 0:

**yield** item

iter01 = get\_even01(list01)*# 没有执行方法体*

**for** item **in** iter01:*# 循环(next)一次,计算一次,返回一次*

print(item)

*# def get\_even02(target):*

*# result = []*

*# for item in target:*

*# if item % 2 == 0:*

*# result.append(item)*

*# return result*

*#*

*# iter01 = get\_even02(list01)# 执行方法体,将所有结果存在内存中．*

*# for item in iter01:*

*# print(item)*

*# 练习２：定义函数，选出所有女同学.*

**class** Student:

**def** \_\_init\_\_(self,name,sex,age,score):

self.name = name

self.sex = sex

self.age = age

self.score = score

**def** \_\_str\_\_(self):

**return "%s--%s--%d--%d"**%(self.name,self.sex,self.age,self.score)

list\_stu = [

Student(**"张无忌"**,**"男"**,28,82),

Student(**"赵敏"**,**"女"**,25,95),

Student(**"周芷若"**,**"女"**,26,88),

]

**def** find\_woman(target):

**for** item **in** target:

**if** item.sex == **"女"**:

**yield** item

**for** item **in** find\_woman(list\_stu):

print(item)

*# 练习３：选出所有成绩大于９０的学生*

**def** find\_by\_score(target):

**for** item **in** target:

**if** item.score > 90:

**yield** item

**for** item **in** find\_by\_score(list\_stu):

print(item)

## 内置生成器

### 枚举函数enumerate

1. 语法：

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。

*"""*

*内置生成器：enumerate*

*"""*

list01 = [**"a"**, **"b"**, **"c"**]

**for** item **in** enumerate(list01):

*# (索引,元素)*

print(item)

**for** index, element **in** enumerate(list01):

print(index, element)

*# 练习：参照上述代码，自定义函数，my\_enumerate.*

*# 10:46*

**def** my\_enumerate(target):

index = 0

**for** item **in** target:

**yield** (index,item)

index += 1

**for** item **in** my\_enumerate(list01):

print(item)

**for** index, element **in** my\_enumerate(list01):

print(index, element)

### zip

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。

*"""*

*内置生成器：zip*

*"""*

list02 = [101,102,103]

list03 = [**"张三丰"**,**"张无忌"**,**"赵敏"**]

**for** item **in** zip(list02,list03):

*# (101, '张三丰')*

print(item)

*#练习：参照上述代码，自定义函数，my\_zip 11:05*

**def** my\_zip(list01,list02):

**for** index **in** range(len(list01)):

*# yield (list01[index],list02[index])*

**yield** list01[index], list02[index]

**for** item **in** my\_zip(list02,list03):

*# (101, '张三丰')*

print(item)

## 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] )

*"""*

*生成器表达式*

*练习：exercise01 exercise02*

*"""*

list01 = [2,3,4,6]

*# result = []*

*# for item in list01:*

*# result.append(item \*\* 2)*

*# 列表推导式[] 字典推导式* *{键:值　for ...} 集合* *{ for ... }*

result = [item \*\* 2 **for** item **in** list01]

print(result)

*# def fun01(target):*

*# for item in target:*

*# yield item \*\* 2*

*# 生成表达式*

*# for item in fun01(list01):*

*# print(item)*

result = (item \*\* 2 **for** item **in** list01)

**for** item **in** result:

print(item)

list02 = [2,3,4,6]

*# 练习：使用列表推导式，与生成器表达式，获取list02中，大于３的数据．*

result01 = [item **for** item **in** list02 **if** item > 3 ] *# 本行代码：执行所有操作，保存所有结果*

result02 = (item **for** item **in** list02 **if** item > 3 ) *# 本行代码：返回生成器对象*

**for** item **in** result01:*# 从结果中获取数据*

print(item)

**for** item **in** result02:*# 循环一次，计算一次，返回一次*

print(item)

*# １．创建技能类(编号，技能名称，冷却时间，攻击力，消耗法力)*

*# 　　创建技能列表．*

*# 　　－－　定义函数：查找冷却时间为０的所有技能对象*

*# 　　－－　定义函数：查找攻击力大于５的所有技能对象*

*# 　　－－　定义函数：查找攻击力大于１０，并且不需要消耗法力的所有技能．*

*# 使用列表推导式，与生成器表达式完成*

*# 通过断点调试，审查程序执行过程，体会两项技术的差异．*

*# 11:47*

**class** SkillData:

**def** \_\_init\_\_(self, id, name, cd, atk, costSP):

self.id = id

self.name = name

self.cd = cd

self.atk = atk

self.costSP = costSP

**def** \_\_str\_\_(self):

**return** self.name

list\_skills = [

SkillData(101, **"降龙十八掌"**, 60, 10, 5),

SkillData(102, **"如来神掌"**, 50, 15, 0),

SkillData(103, **"六脉神剑"**, 0, 20, 8),

SkillData(104, **"一阳指"**, 0, 50, 15),

SkillData(105, **"冷酷追击"**, 15, 30, 9),

]

*# def find\_demo02(target):*

*# for item in target:*

*# if item.cd == 0:*

*# yield item*

result01 = (item **for** item **in** list\_skills **if** item.cd == 0)

result02 = [item **for** item **in** list\_skills **if** item.cd == 0]

**for** item **in** result01:

print(item)

**for** item **in** result02:

print(item)

*# def find\_demo03(target):*

*# for item in target:*

*# if item.atk > 5:*

*# yield item*

result01 = (item **for** item **in** list\_skills **if** item.atk > 5)

result02 = [item **for** item **in** list\_skills **if** item.atk > 5]

**for** item **in** result01:

print(item)

**for** item **in** result02:

print(item)

*# def find\_demo04(target):*

*# for item in target:*

*# if item.atk > 10 and item.costSP == 0:*

*# yield item*

result01 = (item **for** item **in** list\_skills **if** item.atk > 10 **and** item.costSP == 0)

result02 = [item **for** item **in** list\_skills **if** item.atk > 10 **and** item.costSP == 0]

**for** item **in** result01:

print(item)

**for** item **in** result02:

print(item)

# 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。

2. 高阶函数：将函数作为参数或返回值的函数。

## 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。

*"""*

*函数式编程　－－　方法作为参数*

*练习：ｅｘｅｒｃｉｓｅ０３　０４*

*"""*

**def** fun01():

print(**"fun01执行喽"**)

*# 调用fun01，将返回值赋值给变量ａ*

*# a = fun01()*

*# 将函数值赋值给变量ａ（没有执行fun01）*

a = fun01

*# 调用变量ａ，间接执行函数fun01*

a()

*# 将方法fun01作为方法的参数func进行传递*

**def** fun02(func):

print(**"fun02执行喽"**)

*# 对于fun02的定义者而言，不知道也不需要知道func的具体逻辑．*

func()

fun02(fun01)

list01 = [1,2,33,4,45,6]

*# def find\_demo01(target):*

*# for item in target:*

*# if item> 5:*

*# yield item*

*#*

*# def find\_demo02(target):*

*# for item in target:*

*# if item % 2 != 0:*

*# yield item*

*#*

*# def find\_demo03(target):*

*# for item in target:*

*# if item < 3:*

*# yield item*

*# --------------------------------------------*

*# 相同点：*

**def** find\_demo(target,func):

**for** item **in** target:

*# 本行代码，又将不变的与变化的紧密相连*

*# if condition01(item):*

*# 本行代码，使用形参func将不变的与变化的隔离开．*

**if** func(item):

**yield** item

*# --------------------------------------------*

*# 提取不同点：*

**def** condition01(item):

**return** item> 5

**def** condition02(item):

**return** item % 2 != 0

**def** condition03(item):

**return** item < 3

**for** item **in** find\_demo(list01,condition03):

print(item)

### lambda 表达式

1. 定义：是一种匿名方法。
2. 作用：作为参数传递时语法简洁，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

-- 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，且不支持赋值语句。

*"""*

*lambda 表达式(匿名方法)*

*语法：*

*lambda　参数:方法体*

*练习：*

*exercise05.py*

*"""*

**def** fun01():

print(**"我是普通方法"**)

fun01()

**def** fun02(a):

print(**"我是普通方法,参数是,"**, a)

fun02(500)

**def** fun03():

**return True**

print(fun03())

*# -------------------------*

a01 = **lambda**: print(**"我是lambda方法"**)

a01()

a02 = **lambda** a: print(**"我是lambda方法,参数是,"**, a)

a02(500)

a03 = **lambda**: **True**

print(a03())

*#------------------------------------*

**from** common.custom\_list\_tools **import** ListHelper

list01 = [1,2,33,4,45,6]

**for** item **in** ListHelper.find\_all(list01,**lambda** item:item > 5):

print(item)

**for** item **in** ListHelper.find\_all(list01,**lambda** item:item % 2 != 0):

print(item)

*# 提取不同点：*

*# def condition01(item):*

*# return item> 5*

*#*

*# def condition02(item):*

*# return item % 2 != 0*

*#*

*# def condition03(item):*

*# return item < 3*

*# for item in ListHelper.find\_all(list01,condition01):*

*# print(item)*

*# 练习：将以下代码，不变的部分提取到一个方法find\_all中，*

*# 将变化点，提取到不同方法中．*

*# 再实现原有功能*

*# def find\_demo02(target):*

*# for item in target:*

*# if item.cd == 0:*

*# yield item*

*#*

*# def find\_demo03(target):*

*# for item in target:*

*# if item.atk > 5:*

*# yield item*

*#*

*# def find\_demo04(target):*

*# for item in target:*

*# if item.atk > 10 and item.costSP == 0:*

*# yield item*

**class** SkillData:

**def** \_\_init\_\_(self, id, name, cd, atk, costSP):

self.id = id

self.name = name

self.cd = cd

self.atk = atk

self.costSP = costSP

list\_skills = [

SkillData(101, **"降龙十八掌"**, 60, 10, 5),

SkillData(102, **"如来神掌"**, 50, 15, 0),

SkillData(103, **"六脉神剑"**, 0, 20, 8),

SkillData(104, **"一阳指"**, 0, 50, 15),

SkillData(105, **"冷酷追击"**, 15, 30, 9),

]

*# 不变的*

**def** find\_demo04(target,func\_condition):

**for** item **in** target:

*# if condition01(item):*

*# func\_condition 由具体变化点抽象而来*

**if** func\_condition(item):

**yield** item

*# 变化的*

**def** condition01(item):

**return** item.cd == 0

**def** condition02(item):

**return** item.atk > 5

**def** condition03(item):

**return** item.atk > 10 **and** item.costSP == 0

**for** item **in** find\_demo04(list\_skills,condition01):

print(item)

*# 练习：将不变的代码，提取到ListHelper模块中．*

*# 15:40 上课*

**class** SkillData:

**def** \_\_init\_\_(self, id, name, cd, atk, costSP):

self.id = id

self.name = name

self.cd = cd

self.atk = atk

self.costSP = costSP

list\_skills = [

SkillData(101, **"降龙十八掌"**, 60, 10, 5),

SkillData(102, **"如来神掌"**, 50, 15, 0),

SkillData(103, **"六脉神剑"**, 0, 20, 8),

SkillData(104, **"一阳指"**, 0, 50, 15),

SkillData(105, **"冷酷追击"**, 15, 30, 9),

]

*# 不变的*

*# def find\_demo04(target,func\_condition):*

*# for item in target:*

*# if func\_condition(item):*

*# yield item*

*# 变化的*

**def** condition01(item):

**return** item.cd == 0

**def** condition02(item):

**return** item.atk > 5

**def** condition03(item):

**return** item.atk > 10 **and** item.costSP == 0

*# for item in find\_demo04(list\_skills,condition01):*

*# print(item)*

*# 通过模块调用*

**from** common.custom\_list\_tools **import** ListHelper

**for** item **in** ListHelper.find\_all(list\_skills,condition01):

print(item)

*# 练习：lambda实现对技能列表的查找　16:17*

**class** SkillData:

**def** \_\_init\_\_(self, id, name, cd, atk, costSP):

self.id = id

self.name = name

self.cd = cd

self.atk = atk

self.costSP = costSP

list\_skills = [

SkillData(101, **"降龙十八掌"**, 60, 10, 5),

SkillData(102, **"如来神掌"**, 50, 15, 0),

SkillData(103, **"六脉神剑"**, 0, 20, 8),

SkillData(104, **"一阳指"**, 0, 50, 15),

SkillData(105, **"冷酷追击"**, 15, 30, 9),

]

*# 变化的*

*# def condition01(item):*

*# return item.cd == 0*

*#*

*# def condition02(item):*

*# return item.atk > 5*

*#*

*# def condition03(item):*

*# return item.atk > 10 and item.costSP == 0*

*# 通过模块调用*

**from** common.custom\_list\_tools **import** ListHelper

**for** item **in** ListHelper.find\_all(list\_skills, **lambda** item: item.cd == 0):

print(item)

**for** item **in** ListHelper.find\_all(list\_skills, **lambda** item: item.atk > 5):

print(item)

*# 练习：16：30*

*# 解决的问题：*

*# 查找编号是101的(单个)技能对象*

*# 查找名称是"降龙十八掌"的(单个)技能对象*

*# 查找cd大于10的(单个)技能对象*

*# 解决的步骤：*

*# 1. 逐个解决问题*

*# 2. 将共性提取到ListHelper中*

*# 3. 将变化用lambda表示.*

*# def find\_demo01(target):*

*# for item in target:*

*# if item.id == 101:*

*# return item*

*#*

*#*

*# def find\_demo02(target):*

*# for item in target:*

*# if item.name == "降龙十八掌":*

*# return item*

*#*

*#*

*# def find\_demo03(target):*

*# for item in target:*

*# if item.cd > 10:*

*# return item*

item = ListHelper.first(list\_skills, **lambda** item: item.id == 101)

print(item.name)

*# 练习：*

*# 解决的问题：筛选对象* *select*

*# 技能列表　--> 编号列表*

*# 技能列表　--> 名称列表*

*# list\_skills = [*

*# SkillData(101, "降龙十八掌", 60, 10, 5),*

*# SkillData(102, "如来神掌", 50, 15, 0),*

*# SkillData(103, "六脉神剑", 0, 20, 8),*

*# SkillData(104, "一阳指", 0, 50, 15),*

*# SkillData(105, "冷酷追击", 15, 30, 9),*

*# ]*

*# list01 = [101,102,103,104,105]*

*# 17:15*

*# def find\_demo01(target):*

*# for item in target:*

*# yield item.id*

*# yield xxx(item)*

*#*

*# def find\_demo02(target):*

*# for item in target:*

*# yield item.name*

*# def xxx(item):*

*# return item.id*

**for** item **in** ListHelper.select(list\_skills, **lambda** item: item.name):

print(item)

*# 练习：17:29*

*# 解决的问题：求和* *sum*

*# 技能列表　--> 所有技能编号的和*

*# 技能列表　--> 所有技能cd的和*

*# list\_skills = [*

*# SkillData(101, "降龙十八掌", 60, 10, 5),*

*# SkillData(102, "如来神掌", 50, 15, 0),*

*# SkillData(103, "六脉神剑", 0, 20, 8),*

*# SkillData(104, "一阳指", 0, 50, 15),*

*# SkillData(105, "冷酷追击", 15, 30, 9),*

*# ]*

**def** demo01(target):

sum\_value = 0

**for** item **in** target:

*# sum\_value += item.id*

sum\_value += xxx(item)

**return** sum\_value

**def** demo02(target):

sum\_value = 0

**for** item **in** target:

sum\_value += item.cd

**return** sum\_value

**def** xxx(item):

**return** item.id

item = ListHelper.sum(list\_skills, **lambda** item: item.cd)

print(item)

### 内置高阶函数

1. map（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。
2. filter(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。
3. sorted(可迭代对象，key = 函数,reverse = bool值)：排序，返回值为排序结果。
4. max(可迭代对象，key = 函数)：根据函数获取可迭代对象的最大值。
5. min(可迭代对象，key = 函数)：根据函数获取可迭代对象的最小值。

*"""*

*高阶函数*

*"""*

**from** day16.common.custom\_list\_tools **import** ListHelper

*# 敌人类（编号／姓名／攻击力／血量／攻击速度．．．）*

**class** Enemy:

**def** \_\_init\_\_(self, id, name, hp, atk, atk\_speed):

self.id = id

self.name = name

self.hp = hp

self.atk = atk

self.atk\_speed = atk\_speed

list01 = [

Enemy(101, **"玄冥大老"**, 200, 800, 5),

Enemy(102, **"玄冥小老"**, 150, 700, 3),

Enemy(103, **"qtx"**, 800, 1000, 50),

Enemy(104, **"吕泽玛利亚"**, 0, 300, 2),

Enemy(105, **"赵金多"**, 500, 900, 10),

]

*# 1. filter 过滤，类似于ListHelper.find\_all.*

*# 过滤出编号大于102的敌人*

*# for item in filter(lambda e: e.id > 102, list01):*

*# print(item.id)*

*#*

*# for item in ListHelper.find\_all(list01, lambda e: e.id > 102):*

*# print(item.id)*

*# 2. map 映射，类似于ListHelper.select*

*# 映射出所有敌人的姓名*

**for** item **in** map(**lambda** e:e.name,list01):

print(item)

**for** item **in** ListHelper.select(list01,**lambda** e:e.name):

print(item)

*# # 按照血量升序排列,类似于ListHelper.order\_by*

*# for item in sorted(list01,key = lambda e:e.hp ):*

*# print(item.hp)*

*#*

*# # 按照血量降叙排列*

*# for item in sorted(list01,key = lambda e:e.hp,reverse=True):*

*# print(item.hp)*

*# ListHelper.order\_by(list01,lambda e:e.hp)*

*# for item in list01:*

*# print(item.hp)*

*# 获取攻击力最大的敌人*

result = max(list01,key = **lambda** e:e.atk)

print(result.name)

result = ListHelper.get\_max(list01,**lambda** e:e.atk)

print(result.name)

## 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：在一个函数内部的函数,同时内部函数又引用了外部函数的变量。
2. 本质：闭包是将内部函数和外部函数的执行环境绑定在一起的对象。
3. 优点：内部函数可以使用外部变量。
4. 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。
5. 作用：实现python装饰器。

*"""*

*Encolsing 外部嵌套作用域*

*"""*

*# 全局变量G*

g01 = 100

**def** fun01():

*# fun01局部变量Ｌ*

*# E外部嵌套作用域*

a = 1

**def** fun02():

b = 2 *# fun02局部变量Ｌ*

*# print("fun02:", a) # 可以访问外部嵌套变量a*

*# a = 2222 # 没有修改外部嵌套变量a，而是创建了新的局部变量a*

*# print("fun02:",a)*

**nonlocal** a *# 声明外部嵌套变量a*

a = 2222

print(**"fun02:"**,a)

fun02()

print(**"fun01:"**,a)

fun01()

*"""*

*闭包*

*"""*

**def** fun01():

print(**"fun01执行喽"**)

a = 1

**def** fun02():

print(**"fun02执行喽"**)

print(**"外部变量是:"**,a)

**return** fun02

*# 得到的是内部函数*

result = fun01()

*# 调用内部函数，因为内部函数使用了外部变量，所以称之为闭包．*

result()*# 可以使用外部变量，说明外部函数在调用后没有释放．*

*# 案例：*

**def** give\_gift\_money(money):

*"""*

*获取压岁钱*

*"""*

print(**"得到了%d压岁钱"**%money)

**def** child\_buy(target,price):

*"""*

*孩子需要买东西*

*"""*

**nonlocal** money

**if** money >= price:

money -= price

print(**"孩子花了%d钱，买了%s,还剩下%d钱．"**%(price,target,money))

**else**:

print(**"压岁钱不够了"**)

**return** child\_buy

action = give\_gift\_money(10000)

action(**"98k"**,3500)

action(**"小猪佩奇"**,300)

action(**"大黄蜂"**,8000)

*# 体会：闭包使得逻辑连续（因为内部函数可以使用外部变量）．*

### 函数装饰器decorators

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. 语法

def 函数装饰器名称(func):

def 内嵌函数(\*args, \*\*kwargs):

需要添加的新功能

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

return wrapper

@ 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。

*"""*

*装饰器*

*-- 闭包的应用*

*练习:exercise01*

*"""*

*# def say\_hello():*

*# print("hello")*

*#*

*#*

*# def say\_goodbye():*

*# print("goodbye")*

*#*

*#*

*# say\_hello()*

*# say\_goodbye()*

*# 需求：在两个方法实现的功能基础上，增加新功能(打印方法名称)*

*# def say\_hello():*

*# print(say\_hello.\_\_name\_\_)*

*# print("hello")*

*#*

*#*

*# def say\_goodbye():*

*# print(say\_goodbye.\_\_name\_\_)*

*# print("goodbye")*

*#*

*#*

*# say\_hello()*

*# say\_goodbye()*

*# 缺点：代码重复.*

*# 解决：提取打印方法名称的功能*

*# def print\_func\_name(func):*

*# print(func.\_\_name\_\_)*

*#*

*# def say\_hello():*

*# # print(say\_hello.\_\_name\_\_)*

*# print\_func\_name(say\_hello)*

*# print("hello")*

*#*

*#*

*# def say\_goodbye():*

*# # print(say\_goodbye.\_\_name\_\_)*

*# print\_func\_name(say\_goodbye)*

*# print("goodbye")*

*# say\_hello()*

*# say\_goodbye()*

*# 缺点：在两个已有功能的内部，增加新功能，代码可读性差．*

*# def say\_hello():*

*# # print\_func\_name(say\_hello)*

*# print("hello")*

*# def say\_goodbye():*

*# # print\_func\_name(say\_goodbye)*

*# print("goodbye")*

*#*

*# def print\_func\_name(func):*

*# # 包装新旧功能*

*# def wrapper():*

*# # 增加的新功能*

*# print(func.\_\_name\_\_)*

*# # 旧功能*

*# func()*

*#*

*# return wrapper # 返回包装器*

*#*

*# say\_hello = print\_func\_name(say\_hello)*

*# say\_goodbye = print\_func\_name(say\_goodbye)*

*#*

*# say\_hello()*

*# say\_goodbye()*

*# 缺点：调用者完成包装新旧方法的任务．*

*# 解决：应该有定义者完成．*

*# def print\_func\_name(func):*

*# # 包装新旧功能*

*# def wrapper():*

*# # 增加的新功能*

*# print(func.\_\_name\_\_)*

*# # 旧功能*

*# func()*

*#*

*# return wrapper # 返回包装器*

*#*

*# @print\_func\_name # say\_hello = print\_func\_name(say\_hello)*

*# def say\_hello():*

*# print("hello")*

*# return "哈哈"*

*#*

*# @print\_func\_name*

*# def say\_goodbye():*

*# print("goodbye")*

*#*

*# #---------以上是定义者--以下是调用者-----------------*

*# say\_hello()*

*# say\_goodbye()*

*# 缺点：旧功能的返回值不能被客户端代码接受到．*

*# 旧功能的参数，客户端代码也无法传入.*

*# def print\_func\_name(func):*

*# # 包装新旧功能*

*# def wrapper(name):*

*# # 增加的新功能*

*# print(func.\_\_name\_\_)*

*# # 旧功能*

*# return func(name)*

*#*

*# return wrapper # 返回包装器*

*# 缺点：包装器不能适应所有的旧功能参数*

**def** print\_func\_name(func):

*# 包装新旧功能*

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

*# 增加的新功能*

print(func.\_\_name\_\_)

*# 旧功能*

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

**return** wrapper *# 返回包装器*

@print\_func\_name *# say\_hello = print\_func\_name(say\_hello)*

**def** say\_hello(name):

print(name,**"hello"**)

**return "哈哈"**

@print\_func\_name

**def** say\_goodbye(name,age):

print(age,name,**"goodbye"**)

*#---------以上是定义者--以下是调用者-----------------*

print(say\_hello(**"张无忌"**))

say\_goodbye(**"赵敏"**,25)

*"""*

*练习：使用装饰器实现：*

*为两个已有功能(进入后台，删除订单)，增加新功能（验证权限）.*

*"""*

*# 1. 定义装饰器(新功能　＋　旧功能)*

**def** verify\_permissions(func):

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

print(**"验证权限"**)

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

**return** wrapper

*# 2. 拦截调用*

@verify\_permissions

**def** enter\_background(loginId,pwd):

print(loginId,pwd)

print(**"进入后台系统....."**)

@verify\_permissions

**def** delete\_order(order\_id):

print(**"删除%d订单..."**%order\_id)

enter\_background(**"zs"**,123)

delete\_order(101)

*"""*

*练习２：为两个已有功能(存款取款),添加新功能(验证账户)*

*"""*

**def** verify\_accont(func):

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

print(**"验证账户"**)

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

**return** wrapper

*#deposit = verify\_accont(deposit)*

@verify\_accont

**def** deposit(money):

print(**"存款:"**,money)

@verify\_accont

**def** withdraw():

print(**"取钱"**)

**return** 10000

deposit(5000)

print(withdraw())

*"""*

*练习：*

*为学生的学习方法，添加新功能(打印执行时间)*

*"""*

**import** time

**def** print\_execute\_time(func):

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

*# 记录执行前的时间*

start\_time = time.time()

result = func(\*args, \*\*kwargs)

*# 统计执行时间*

execute\_time = time.time() - start\_time

print(**"执行时间是："**, execute\_time)

**return** result

**return** wrapper

**class** Student:

**def** \_\_init\_\_(self, name):

self.name = name

@print\_execute\_time

**def** study(self):

print(**"开始学习喽"**)

time.sleep(2) *# 睡眠两秒　模拟学习了两秒*

s01 = Student(**"无忌"**)

s01.study()