内容目录

[一、模块 Module 3](#__RefHeading___Toc856_3022085775)

[定义 3](#__RefHeading___Toc858_3022085775)

[作用 3](#__RefHeading___Toc860_3022085775)

[导入 3](#__RefHeading___Toc862_3022085775)

[import 3](#__RefHeading___Toc864_3022085775)

[from import 4](#__RefHeading___Toc866_3022085775)

[from import \* 4](#__RefHeading___Toc868_3022085775)

[模块变量 5](#__RefHeading___Toc870_3022085775)

[加载过程 6](#__RefHeading___Toc872_3022085775)

[分类 6](#__RefHeading___Toc874_3022085775)

[搜索顺序 6](#__RefHeading___Toc876_3022085775)

[二、包package 7](#__RefHeading___Toc878_3022085775)

[定义 7](#__RefHeading___Toc880_3022085775)

[作用 8](#__RefHeading___Toc882_3022085775)

[导入 8](#__RefHeading___Toc884_3022085775)

[搜索顺序 8](#__RefHeading___Toc886_3022085775)

[\_\_init\_\_.py 文件 8](#__RefHeading___Toc888_3022085775)

[\_\_all\_\_ 8](#__RefHeading___Toc890_3022085775)

[三、异常处理Error 9](#__RefHeading___Toc892_3022085775)

[异常 9](#__RefHeading___Toc894_3022085775)

[处理 9](#__RefHeading___Toc896_3022085775)

[raise 语句 10](#__RefHeading___Toc898_3022085775)

[自定义异常 10](#__RefHeading___Toc900_3022085775)

[四、迭代 10](#__RefHeading___Toc902_3022085775)

[可迭代对象iterable  10](#__RefHeading___Toc904_3022085775)

[迭代器对象iterator 11](#__RefHeading___Toc906_3022085775)

[五、生成器generator 11](#__RefHeading___Toc908_3022085775)

[生成器函数 12](#__RefHeading___Toc910_3022085775)

[内置生成器 12](#__RefHeading___Toc912_3022085775)

[枚举函数enumerate 12](#__RefHeading___Toc914_3022085775)

[zip 13](#__RefHeading___Toc916_3022085775)

[生成器表达式 13](#__RefHeading___Toc918_3022085775)

[六、函数式编程 13](#__RefHeading___Toc1040_3022085775)

[函数作为参数 13](#__RefHeading___Toc924_3022085775)

[lambda 表达式 13](#__RefHeading___Toc926_3022085775)

[内置高阶函数 14](#__RefHeading___Toc928_3022085775)

[函数作为返回值 14](#__RefHeading___Toc930_3022085775)

[闭包 14](#__RefHeading___Toc932_3022085775)

[函数装饰器decorators 15](#__RefHeading___Toc934_3022085775)

# 一、模块 Module

"""

模块

17:00

练习:day14 \_exercise03.py

"""

## 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

# 导入方式1

# 本质：使用变量名module01（ｄａｙ14）关联模块地址

# import module01

# module01.fun01()

# my02 = module01.MyClass02()

# my02.fun02()

# as 为导入的成员其另外一个名称

import module01 as m01

m01.fun01()

my02 = m01.MyClass02()

my02.fun02()

### 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：将指定的成员导入到当前模块作用域中

# 小心：导入进来的成员不要和当前模块成员名称相同

# from module01 import fun01

# from module01 import MyClass02

#

# def fun01():

# print("当前模块fun01")

#

# fun01()

# my02 = MyClass02()

# my02.fun02()

### 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

# 本质：将指定模块的所有成员导入到当前模块作用域中

# 小心：导入进来的成员和其他模块成员冲突

from module01 import \*

fun01()

my02 = MyClass02()

my02.fun02()

# from module01 import \*

#

# fun01()

"""

模块相关概念

"""

# # 1. 隐藏成员，不能通过from 模块 import × 形式导入

# \_fun02()

# from module01 import \_fun02

#

# # 隐藏成员，可以通过其他形式调用

# \_fun02()

**模块变量**

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

for example: (day15\_module01.py)

# 2. 通过\_\_all\_\_指定可导出成员

from module01 import \*

MyClass.fun03()

\_fun02()

\_\_doc\_\_变量：文档字符串。(可以通过该属性查看代码注释)

# 3.可以通过该属性，查看文档注释

print(\_\_doc\_\_)

\_\_file\_\_变量：模块对应的文件路径名。

# 4.返回当前模块的绝对路径（从系统根目录开始计算的）

print(\_\_file\_\_)

\_\_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。

补充：(主模块叫做：'\_\_main\_\_'，非主模块叫做：模块真名）

for example: day15\_double\_list\_helper，project中的student\_manger\_system下的main.py

# 5.

# 现象：

# 主模块叫做：\_\_main\_\_

# 非主模块叫做：真名

print(\_\_name\_\_)

# 作用1： 不是主模块不执行。(测试代码)

# 作用2： 只有是主模块才执行。(主模块代码)

# 使用：

if \_\_name\_\_ == "\_\_main\_\_":

pass

## 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）

## 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。

补充：

"""

时间处理

练习：day15－exercise01.py

day15－exercise02.py

"""

import time

# 1. 获取当前时间戳(从1970年1月1日到现在经过的秒数)

# 1560998261.108855

print(time.time())

# 2. 时间元组(年，月，日，时，分，秒，一周的第几天，一年的第几天，夏令时)

# 时间戳 --> 时间元组

print(time.localtime(1560998261))

# 通过元组的操作获取时间

tuple\_time = time.localtime()

for item in tuple\_time:

print(item)

print(tuple\_time[1]) # 获取月

# 通过类的操作获取时间

print(type(tuple\_time))

# print(time.struct\_time)

print(tuple\_time.tm\_year) # 获取年

# 时间元组 --> 时间戳

print(time.mktime(tuple\_time))

# 3. 时间元组 --> str

str\_time01 = time.strftime("%Y / %m / %d %H:%M:%S", tuple\_time)

print(str\_time01)

# str --> 时间元组

print(time.strptime(str\_time01, "%Y / %m / %d %H:%M:%S"))

注意区别：

week\_time=time.localtime()

*# 获取星期,其返回值为０，１，２，３，４，５，６*

*#* 　　　　*星期一，二，三，四，五，六，日*

print(week\_time[6])

*## %w 星期（0-6），星期天为星期的开始*

*# 获取星期,其返回值为０，１，２，３，４，５，６*

*#* 　　　　*星期日，一，二，三，四，五，六*

print (time.strftime(**"%w "**, week\_time))

# 二、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

import 包名 [as 包别名] 需要设置\_\_all\_\_

import 包名.模块名 [as 模块新名]

import 包名.子包名.模块名 [as 模块新名]

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

python程序结构

文件夹 ---- 项目根目录

包

模块

类

函数

语句

"""

# # form 包.模块 import 成员

# from package01.module\_a import fun01

# fun01()

#

# form 包.包.模块 import 成员

from package01.package02.module\_b import fun02

fun02()

# import package01.module\_a as pm

# pm.fun01()

## 搜索顺序

sys.path 提供的路径

# 15:50

print("common -- list\_helper")

import sys

# 如果不再pycharm中运行当前模块，则导包失败.

# 将项目根目录加入path中，导包才会成功.

sys.path.append("/home/tarena/1905/month01/code/day15/my\_project")

print(sys.path)

from main import \*

main\_fun01()

## \_\_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

# 语法：from 包 import \*

# 依赖于在包的\_\_init\_\_.py文件中指定\_\_all\_\_ = [可导出的模块]

# from skill\_system import \*

# manager = skill\_manager.SkillManager()

# manager.generate()

案例：

my\_ project /

\_\_init\_\_.py

main.py

common/

\_\_init\_\_.py

double\_list\_helper.py

list\_helper.py

skill\_system/

\_\_init\_\_.py

skill\_deployer.py

skill\_manager.py

# 三、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

"""

异常处理

"""

def div\_apple(apple\_count):

# ValueError

person\_count = int(input("请输入人数："))

# ZeroDivisionError

result = apple\_count / person\_count

print("每人%d个苹果"%result)

"""

try:

# 可能出错的代码

div\_apple(10)

except Exception:

print("出错喽")

"""

"""

# "建议"分门别类的处理

try:

# 可能出错的代码

div\_apple(10)

except ValueError:

print("输入的人数必须是整数")

except ZeroDivisionError:

print("输入的人数不能是零")

except Exception:

print("未知错误")

"""

"""

try:

# 可能出错的代码

div\_apple(10)

except Exception:

print("出错喽")

else:

# 如果异常，不执行else语句.

print("没有出错")

"""

try:

# 可能出错的代码

div\_apple(10)

finally:

# 无论是否异常，一定会执行的代码.

print("finally")

# 作用：不能处理的错误，但是一定要执行的代码，就定义到finally语句中。

print("后续逻辑.....")

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。。

## 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

"""

自定义异常类

"""

class AgeError(Exception):

"""

年龄错误

"""

def \_\_init\_\_(self,message,age\_value,code\_line,error\_number):

super().\_\_init\_\_("出错啦啦啦")

self.message = message

self.age\_value = age\_value

self.code\_line = code\_line

self.error\_number = error\_number

class Wife:

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

self.age = age

@property

def age(self):

return self.\_\_age

@age.setter

def age(self,value):

if 21 <= value <= 31:

self.\_\_age = value

else:

# 5

# raise ValueError("我不要")

raise AgeError("超过我想要的范围啦",value,26,1001)

w01 = Wife(81)

# try:

# w01 = Wife(81)

# except AgeError as e:

# print("请重新输入")

# print(e.message)

# print(e.age\_value)

# print(e.code\_line)

# 四、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

"""

可迭代对象

练习：day16\_exercise01.py

"""

# 可迭代对象 -- 容器

list01 = [43,3,4,5,567]

# 迭代过程

# for item in list01:

# print(item)

# 迭代原理

# 面试题：for循环的原理是什么？

# 答：1. 获取迭代器

# 2. 循环获取下一个元素

# 3. 遇到异常停止迭代

# 可以被for的条件是什么？

# 答：能被for的对象必须具备\_\_iter\_\_方法

# 答：能被for的对象是可迭代对象

#1. 获取迭代器

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

#2. 循环获取下一个元素

while True:

try:

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

print(item)

#3. 遇到异常停止迭代

except StopIteration:

break# 退出循环

## 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。

"""

迭代器

练习:day16\_exercise03.py

"""

class Skill:

pass

class SkillManager:

"""

技能管理器 可迭代对象

"""

def \_\_init\_\_(self):

self.\_\_skills = []

def add\_skill(self, skill):

self.\_\_skills.append(skill)

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

# 创建一个迭代器对象,并传递需要迭代的数据。

return SkillIterator(self.\_\_skills)

class SkillIterator:

"""

技能迭代器

"""

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

self.\_\_target = target

self.\_\_index = 0

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

# 如果没有数据了，则抛出异常

if self.\_\_index > len(self.\_\_target) - 1:

raise StopIteration

# 返回下一个数据

temp = self.\_\_target[self.\_\_index]

self.\_\_index += 1

return temp

manager = SkillManager()

manager.add\_skill(Skill())

manager.add\_skill(Skill())

manager.add\_skill(Skill())

for item in manager:

print(item)

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

while True:

try:

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

print(item)

except StopIteration:

break

# 五、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。

"""

迭代器 --> yield

练习: 将迭代器版本的图形管理器改为yield实现.

day16\_exercise03.py -->exercise06.py

"""

class MyRange:

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

self.stop\_value = stop\_value

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

# return MyRangeIterator(self.stop\_value)

# 0 --> self.stop\_value

# yield 作用: 将下列代码改为迭代器模式的代码.

# 生成迭代器代码的大致规则:

# 1. 将yield以前的语句定义在next方法中

# 2. 将yield后面的数据作为next方法返回值

number = 0

while number < self.stop\_value:

yield number

number += 1

# print("准备数据")

# yield 0

# print("准备数据")

# yield 1

# print("准备数据")

# yield 2

# # ...

"""

class MyRangeIterator:

def \_\_init\_\_(self, end\_value):

self.\_\_end\_value = end\_value

self.\_\_number = 0

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

if self.\_\_number == self.\_\_end\_value:

raise StopIteration

temp = self.\_\_number

self.\_\_number += 1

return temp

"""

# next一次，计算一次，返回一次。

# for item in MyRange(10):

# print(item)

my01 = MyRange(10)

iterator = my01.\_\_iter\_\_()

while True:

try:

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

print(item)

except StopIteration:

break

## 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

练习:day 16\_exercise07.py

"""

"""

class MyRange:

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

self.stop\_value = stop\_value

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

number = 0

while number < self.stop\_value:

yield number

number += 1

my01 = MyRange(10)

iterator = my01.\_\_iter\_\_()

while True:

try:

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

print(item)

except StopIteration:

break

"""

"""

# 生成器原理

class MyGenerator:

# 生成器 = 可迭代对象 + 迭代器

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

self.begin = 0

self.stop\_value = stop\_value

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

return self

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

if self.begin >= self.stop\_value:

raise StopIteration

temp = self.begin

self.begin+=1

return temp

"""

def my\_range(stop\_value):

number = 0

while number < stop\_value:

yield number

number += 1

my01 = my\_range(10)

print(type(my01), dir(my01))# dir 获取对象所有成员

print(id(my01.\_\_iter\_\_()), id(my01))

for item in my01:

print(item)

## 内置生成器

### 枚举函数enumerate（day17ｅｘｅｒｃｉｓｅ０１）

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。

### zip（day17ｅｘｅｒｃｉｓｅ０２）

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。

## 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:day17\_exercise03.py

"""

list01 = [3, "54", True, 6, "76", 1.6, False, 3.5]

# 生成器函数

def find01():

for item in list01:

if type(item) == int:

yield item + 1

re = find01()

for item in re:

print(item)

# 生成器表达式

# 此时没有计算,更没有结果

re = (item + 1 for item in list01 if type(item) == int)

# 一次循环,一次计算,一个结果

for item in re:

print(item)

# 列表推导式

# 此时已经完成所有计算,得到所有结果

re = [item + 1 for item in list01 if type(item) == int]

# 只是获取所有结果

for item in re:

print(item)

（元组没有推导式）

# 变量 = [itme for item in 可迭代对象 if 条件] 列表推导

# 变量 = {k,v for k,v in 可迭代对象 if 条件} 字典推导

# 变量 = {item for item in 可迭代对象 if条件} 集合推导

# 变量 = (item for item in 可迭代对象 if条件) 生成器表达式

# 六、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。

"""

函数式编程 思想

练习:day17\_exercise05.py

"""

class SkillData:

def \_\_init\_\_(self,id,name,atk\_ratio,duration):

self.id = id

self.name = name

self.atk\_ratio = atk\_ratio

self.duration = duration

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

return "技能数据是:%d,%s,%d,%d"%(self.id,self.name,self.atk\_ratio,self.duration)

list\_skill = [

SkillData(101,"乾坤大挪移",5,10),

SkillData(102,"降龙十八掌",8,5),

SkillData(103,"葵花宝典",10,2),

]

# 需求1:获取攻击比例大于6的所有技能

def find01():

for item in list\_skill:

if item.atk\_ratio > 6:

yield item

# 需求2:获取持续时间在4--11之间的所有技能

def find02():

for item in list\_skill:

if 4<item.duration<11:

yield item

# 需求3:获取技能名称大于4个字并且持续时间小于6的所有技能

def find04():

for item in list\_skill:

if len(item.name) > 4 and item.duration < 6:

yield item

# "封装"(分而治之 变则疏之)

# 将每个变化的条件,单独定义在函数中.

def condition01(item):

return item.atk\_ratio > 6

def condition02(item):

return 4<item.duration<11

def condition03(item):

return len(item.name) > 4 and item.duration < 6

# "继承"(隔离变化)

def find(func\_condition):

"""

通用的查找方法

:param func\_condition: 查找条件,函数类型.

函数名(变量) --> 返回值bool类型

:return:

"""

for item in list\_skill:

# "多态":调用父(变量),执行子(具体函数).

# 不同子类重写父类方法,执行逻辑不同.

# if item.atk\_ratio > 6:

# if condition01(item):

if func\_condition(item):

yield item

for item in find(condition01):

print(item)

for item in find(condition02):

print(item)

## 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。

### 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 匿名函数

语法:lambda 参数列表:函数体

注意:函数体自带return

"""

from common.list\_helper import \*

list01 = [43, 4, 5, 5, 6, 7, 87]

# def condition01(item):

# return item % 2 == 0

#

# def condition02(item):

# return item > 10

#

# def condition03(item):

# return 10 < item < 50

# for item in ListHelper.find\_all(list01, condition02):

# print(item)

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

print(item)

# ------------------------------

# 无参数函数 --> lambda

def fun01():

return 100

a = lambda: 100

re = a()

print(re)

# 多参数函数 --> lambda

def fun02(p1, p2):

return p1 > p2

b = lambda p1, p2: p1 > p2

re = b(1, 2)

print(re)

# 无返回值函数 --> lambda

def fun03(p1):

print("参数是:", p1)

c = lambda p1: print("参数是:", p1)

c(100)

# 方法体只能有一条语句，且不支持赋值语句

def fun04(p1):

p1 = 2

# d = lambda p1:p1 = 2

### 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。

"""

内置高阶函数

练习:

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

获取元组中，长度最大的列表.

2. 根据敌人列表，获取所有敌人的姓名与血量与攻击力.

3.　在敌人列表中，获取攻击力大于100的所有活人.

4. 根据防御力对敌人列表进行降序排列.

"""

from common.list\_helper import \*

class Enemy:

def \_\_init\_\_(self, name, hp, atk, defense):

self.name = name

self.hp = hp

self.atk = atk

self.defense = defense

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

return "%s--%d--%d--%d" % (self.name, self.hp, self.atk, self.defense)

list01 = [

Enemy("玄冥二老", 86, 120, 58),

Enemy("成昆", 0, 100, 5),

Enemy("谢逊", 120, 130, 60),

Enemy("灭霸", 0, 1309, 690),

]

# 1.filter: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

# 需求：获取所有死人

for item in ListHelper.find\_all(list01, lambda item: item.hp == 0):

print(item)

re = filter(lambda item: item.hp == 0, list01)

for item in re:

print(item)

# 2.map:通用的筛选方法

# 需求：获取所有敌人的姓名

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

print(item)

#

re = map(lambda item: item.name, list01)

for item in re:

print(item)

# 3. max:获取最大值

# 需求：获取血量最大的敌人

print(ListHelper.get\_max(list01, lambda item: item.hp))

print(max(list01, key=lambda item: item.hp))

# 4. min:获取最小值

# 略

# 5.

# 内部直接修改列表，使用时无需通过返回值获取数据

# ListHelper.order\_by(list01,lambda item:item.atk)

# for item in list01:

# print(item)

# 内部返回新列表，使用时必须获取返回值．

# re = sorted(list01, key=lambda item: item.atk)

# for item in re:

# print(item)

# 支持降序排列

re = sorted(list01, key=lambda item: item.atk, reverse=True)

for item in re:

print(item)

## 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。（内嵌函数而不是内嵌函数的值，故内嵌函数不加括号）

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。

"""

闭包

"""

def fun01():

a = 1

def fun02():

print(a)

return fun02

# 调用外部函数，返回值是内嵌函数

result = fun01()

# 调用内嵌函数

result() # 可以访问外部变量a

# 闭包应用:逻辑连续，当内部函数被调用时，不脱离当前的逻辑

# 压岁钱

def give\_gife\_money(money):

"""

得到压岁钱

:return:

"""

print("得到了%d压岁钱" % money)

def child\_buy(target, price):

"""

孩子购买商品

:param target: 需要购买的商品

:param price: 商品单价

"""

nonlocal money

if money >= price:

money -= price

print("孩子花了%.1f钱，购买了%s" % (price, target))

else:

print("钱不够啦")

return child\_buy

# 下列代码是一个连续的逻辑

action = give\_gife\_money(10000)

action("唐僧肉", 0.5)

action("小汽车", 2000)

action("手机", 8000)

### 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

"""

# 需求：对以下两个功能增加权限验证.

"""

# 需要增加的功能

def verify\_permissions():

print("权限验证")

# 已有功能

def enter\_background():

verify\_permissions()

print("进入后台")

def delete\_order():

verify\_permissions()

print("删除订单")

enter\_background()

delete\_order()

# 缺点：增加新功能，需要修改已有功能． [违反开闭原则]

"""

"""

# 需要增加的功能

def verify\_permissions(func):

def wrapper():

print("权限验证")

func()

return wrapper

# 已有功能

def enter\_background():

print("进入后台")

def delete\_order():

print("删除订单")

# enter\_background = 新功能 + 旧功能

enter\_background = verify\_permissions(enter\_background)

delete\_order = verify\_permissions(delete\_order)

enter\_background()

delete\_order()

缺点：每次拦截对已有功能(enter\_background)的调用,不科学.

"""

"""

# 需要增加的功能

def verify\_permissions(func):

def wrapper():

print("权限验证")

func()

return wrapper

# 已有功能

# enter\_background = verify\_permissions(enter\_background)

@verify\_permissions

def enter\_background():

print("进入后台")

@verify\_permissions

def delete\_order():

print("删除订单")

enter\_background()

delete\_order()

缺点：如果已有功能参数不统一，则无法包装.

"""

def verify\_permissions(func):

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

print("权限验证")

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

return wrapper

# 已有功能

@verify\_permissions

def enter\_background(login\_id, pwd):

print(login\_id, pwd, "进入后台")

@verify\_permissions

def delete\_order(id):

print("删除订单", id)

# enter\_background = verify\_permissions(enter\_background)

# delete\_order = verify\_permissions(delete\_order)

enter\_background("abc", 1234)

delete\_order(101)

#####day14\_\_module01.py

print("模块1")

def fun01():

print("模块1的fun01")

class MyClass02:

def fun02(self):

print("MyClass02—fun02")