# 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

### 方法一：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. 本质：将该模块作用域 赋值给 变量 模块名
3. 使用：模块名.成员

### 方法二：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。

本质：将模块指定成员赋值给变量 模块名

### 方法三：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。
4. 弊端：很可能会造成多个模块中间成员的命名冲突．

**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。

*# 获取模块文档注释*

**import** 模块名

print(模块名.\_\_doc\_\_)

\_\_file\_\_变量：模块对应的文件路径名。

*# 获取模块文件路径*

print(模块名.\_\_file\_\_)

\_\_name\_\_变量：模块自身名字，可以判断是否为主模块。

*# 获取模块名称*

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

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。

## 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(如time模块)。
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。

# 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 \*

推荐导入方式：**from 项目根目录.子包名.模块名 import \***

## 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 /

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。

-- 未实现异常(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。。

## 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

# 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：

**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

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

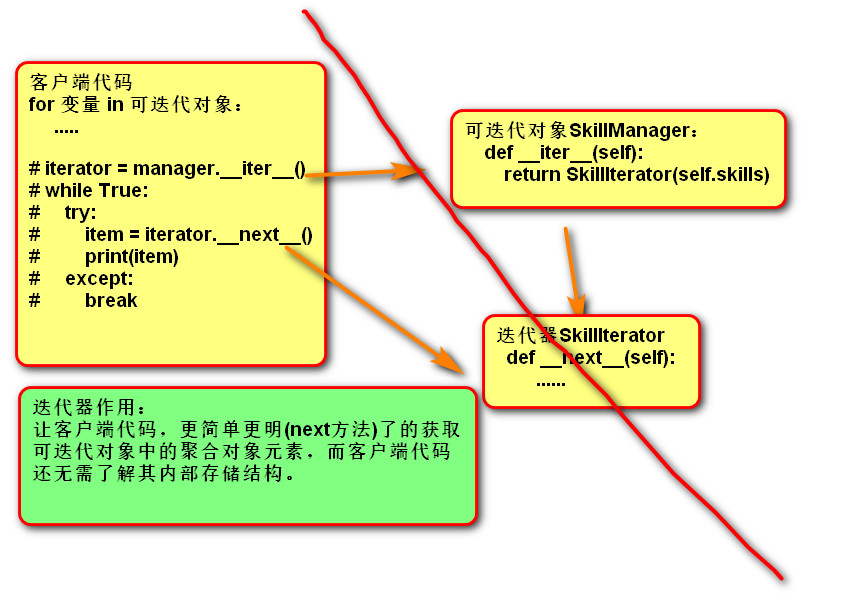
return 聚合对象元素

1. 说明：

-- 聚合对象通常是容器对象。

4. 作用：使用者(客户端代码)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。

**5.技能管理器迭代内存图：**



# 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

**方法／函数，需要向外返回多个结果时，使用生成器函数**．

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。

## 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。

### 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。

## 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] )

# 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。

### 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。

### 内置高阶函数(性能高于手写的函数)（day17）

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)：**过滤，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(默认的是False)):**返回值为排序结果**．**

**sorted**结果返回的是新的列表，不会改变旧的列表。

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

## 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。
6. 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（闭包的应用）-------(day18)

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

调试：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。

# 练习

## 每日课堂练习：

### 四月十八(day13)

#### 1.标准库模块 -- 时间

**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"**))

#### 2.定义函数, 输入年月日,返回星期.

**import** time

**def** get\_week(year, month, day):

time\_tuple= time.strptime(**"%d/%d/%d"** % (year,month,day),**"%Y/%m/%d"**)

weeks = {

0: **"星期一"**,

1: **"星期二"**,

2: **"星期三"**,

3: **"星期四"**,

4: **"星期五"**,

5: **"星期六"**,

6: **"星期日"**,

}

*# time\_tuple[6] 从时间元组中获取星期数*

**return** weeks[time\_tuple[6]]

print(get\_week(2019, 4, 18)) *#星期四*

#### 3.定义函数:根据生日(年月日),返回活了多少天.

*# -- 根据年月日构建时间元组*

*# -- 根据构建的时间元组获取时间戳*

*# -- 使用当前时间戳* *- 生日的时间戳*

*# -- 将活的秒数换算为天*

**def** life\_days(year, month, day):

time\_tuple = time.strptime(**"%d/%d/%d"** % (year, month, day), **"%Y/%m/%d"**)

*# time.mktime( time\_tuple)*

life\_seconds = time.time() - time.mktime(time\_tuple)

**return** life\_seconds / 60 / 60 // 24

print(life\_days(1984, 6, 11))

### 四月二十二（day15）

#### 01．可迭代对象

*"""*

*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]

*# 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*

#### 02．迭代器

*"""*

*迭代器*

*"""*

**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*

#### 03．技能管理器－－＞yield

**class** Skill:

**pass**

**class** SkillManager:

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

self.skills = 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**

#### 04．Range( )生成器函数

*yield　－－＞　　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) *# 0 1 2 3 4*

#### 05．获取元组/字典的所有元素

*# 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**

#### 06．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**

#### 07．定义ＭｙＲａｎｇｅ类，实现range()的相同效果

**class** MyRangeIterator: *# 迭代器*

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

self.stop = stop *# range的最后一个值*

self.start = 0 *# range的第一个值*

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

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

**raise** StopIteration()

temp = self.start

self.start += 1

**return** temp

**class** MyRange: *#可迭代对象*

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

self.stop = stop

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

*# 创建迭代器对象*

**return** MyRangeIterator(self.stop)

iterator = MyRange(5)

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

print(item)

#### 08．改造员工管理器为yield实现

*"""*

*改造为yield实现*

*"""*

**class** Employee:

**pass**

*# 可迭代对象*

**class** EmployeeManager:

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

self.all\_employee = employees

**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**

#### 09．改造MyRange为yield实现

**class** MyRange:

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

self.stop = 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**

#### 10．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 = [2,3,4,5,6,7,8,9]

*#* ***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)

*#另一种方法*

*# 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)

### **四月二十三（day16）**

#### 1．内置生成器：enumerate

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

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

*# (索引,元素)*

print(item) *# (0, 'a')*

*# (1, 'b')*

*# (2, 'c')*

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

print(index, element) *# 0 a*

*# 1 b*

*# 2 c*

*# 练习：参照上述代码，自定义函数，my\_enumerate.*

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

index = 0

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

**yield** (index,item)

index += 1

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

print(item) *# (0, 'a')*

*# (1, 'b')*

*# (2, 'c')*

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

print(index, element) *# 0 a*

*# 1 b*

*# 2 c*

#### 2．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*

**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)

#### 3．生成器表达式：

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 ...} 集合推导式* *{ 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)

#### 4．函数式编程－－－方法作为参数

**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)

*#练习*

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

*# －－－－－－－－－－－测试－－－－－－－－－－－－－－－*

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

print(item)

##### 4-1．函数式编程练习二：

*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

*# -------------------------测试-------------------*

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

print(item.name)

##### 4-2．函数式编程练习三：

*#　将不变的代码，提取到ListHelper*（针对列表的自定义工具）*模块中．*

**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custom\_list\_tools(针对列表的自定义工具)*

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

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

print(item.name)

#### 5．lambda表达式

**解决的步骤：**

**1. 逐个解决问题**

**2. 将共性提取到ListHelper中(工具模块中)**

**3. 将变化用lambda表示.**

*# lambda 表达式(匿名方法)*

*# 语法：*

*# lambda　参数:方法体*

*# 练习：*

**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** custom\_list\_tools **import** ListHelper *# 调用自定义工具模块中的custom\_list\_tools*

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)*

##### 5-1．lambda实现对技能列表的查找

**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** custom\_list\_tools **import** ListHelper

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

print(item.name)

print(**"-----------------------------"**)

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

print(item.name)

print(**"-----------------------------"**)

*# 练习：*

*# 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*

*# 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.id)

*# 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]*

*# 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)

print(**"------------"**)

*# 练习：*

*# 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)

### 四月二十四（day17）

#### １．内置高阶函数

**from** custom\_list\_tools **import** ListHelper　*# 导入模块*

*# 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(**"filter 过滤:"**,item.id)

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

print(**"find\_all:"**,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(**"map 映射:"**,item)

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

print(**"select:"**,item)

*# 3.sorted ,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(**"sorted升序:"**,item.hp)

*# # 按照血量降叙排列*

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

print(**"sorted降序："**,item.hp)

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

**for** item **in** list01:

print(**"order\_by:"**,item.hp)

*# 4.max, 获取攻击力最大的敌人*

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

print(**"max攻击力最大的敌人:"**,result.name)

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

print(**"get\_max攻击力最大的敌人:"**,result.name)

### 四月二十五（day18）

#### １．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()

#### ２．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）．*

#### ３．装饰器　-- 闭包的应用

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

##### 推理步骤１．定义两个函数

**def** say\_hello():

print(**"hello"**)

**def** say\_goodbye():

print(**"goodbye"**)

say\_hello() *# hello*

say\_goodbye() *# goodbye*

##### 推理步骤２．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\_hello*

*#hello*

say\_goodbye()*#say\_goodbye*

*#goodbye*

*# 缺点：代码重复.*

*# 解决：提取打印方法名称的功能*

##### 推理步骤３．

**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\_hello*

*# hello*

say\_goodbye()*# say\_goodbye*

*# goodbye*

*# 缺点：在两个已有功能的内部，增加新功能，代码可读性差．*

##### 推理步骤４．

**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\_hello*

*#hello*

say\_goodbye()*#say\_goodbye*

*#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．*

*# 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.*

##### 推理步骤５．

**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*

##### 推理步骤６．

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

*# 包装新旧功能*

**def** wrapper():

*# 增加的新功能*

print(func.\_\_name\_\_)*# 打印调用方法名称*

*# 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()

##### 推理步骤７．

*# 包装新增功能*

**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*

*# 原函数*

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

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

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

**return "哈哈"**

*# 原函数*

@print\_func\_name *# 函数装饰器名称*

**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)

## 每日早上复习：

### 四月十八(day13)

*"""*

*面向对象:考虑问题,要从对象的角度出发.*

*谁?干嘛?*

*主要思想:*

*识别对象/分配职责/建立交互*

*封装变化/隔离变化/执行变化*

*封装 继承 多态*

*特征:*

*封装:*

*1) 数据:将多个基本类型,合成一个自定义类型.*

*优势:复合人类的思考方式*

*比如:学生,向量(x,y)*

*2) 行为:对外提供简单的必要的功能,隐藏实现的细节.*

*优势:模块化开发,简化编程*

*比如:学生管理系统* *-- 逻辑控制器,供界面调用.*

*3) 设计:*

*分而治之:分解需求,让多个类协同完成.*

*封装变化:每个变化点单独做成一个类.*

*-------------------------------*

*高内聚:类的内部处理一个变化点*

*低耦合:类与类的关系,尽量做到互不影响.*

*继承:重用现有类的概念,并在此基础上进行扩展.*

*(子类的共性) (子类相比父类更加具体)*

*作用:隔离客户端代码与实现方式(隔离用与做)*

*比如:交通工具隔离了人与飞机/火车...的变化*

*图形隔离了图形管理器与圆形/矩形...的变化*

*多态:调用父一个方法,执行子类方法,不同实现方式不一样,所以表现形态就不一样.*

*作用:通过重写执行不同变化点*

*比如:人调用交通工具的运输方法,执行的是飞机/火车的运输方法*

*图形管理器调用图形的计算面积方法,执行的是圆形/矩形的计算面积方法.*

*原则:*

*开闭原则:增加新功能,不修改客户端代码.*

*比如:某个技能,增加新影响效果,只要创建新的效果类,不需要修改其他代码.*

*单一职责:每个类有且只有一个改变的原因.*

*比如:技能系统中,每个类的职责明确.*

*依赖倒置:使用抽象(父),而不是用具体(子)*

*比如:技能释放器,调用的是影响,而不是伤害生命/降低防御力..*

*组合复用:使用关联关系,代替继承关系.*

*比如:技能释放器与影响效果,使用了关联关系.*

*里氏替换:父类出现的地方,可以被子类替换*

*替换后,保持原有功能.*

*比如:技能释放器使用影响效果,但可以被所有具体效果替换.*

*迪米特法则:低耦合*

*比如:影响效果的变化,不影响释放器.*

*每种效果之间,互不影响.*

*"""*

### 四月二十二（day15）

*day14 复习*

*包:*

*导入方式*

*from 包名.模块名* *import 成员* *---> 使用:成员*

*from 包* *import 模块* *---> 使用:模块.成员*

*import 包名.模块名* *--> 使用:包名.模块名*

*备注:所有路径,从项目根目录开始算起.*

*异常处理:*

*异常:运行时遇到的错误.程序返回调用端.*

*处理:将异常状态变为正常状态.*

*自行抛出异常***:raise** *异常对象* *xxxError(参数)*

*自定义异常:封装错误信息*

*定义:class XXXError(Exception):*

*def \_\_init\_\_(self,参数):*

*self.成员变量* *= ?*

*抛出异常:*

*raise XXXError(参数)*

*处理:*

*try:*

*.....*

*except XXXError as 变量:*

*变量.成员*

### 四月二十三（day16）

*day15 复习*

*可迭代对象：*

*1. 定义：具有\_\_iter\_\_方法*

*2. 语法：for 变量　in 可迭代对象:*

*3. 原理：迭代器　＝　可迭代对象.\_\_iter\_\_()*

*while True:*

*try:*

*变量　＝　迭代器.\_\_next\_\_()*

*except:*

*break*

*迭代器：*

*1. 定义:具有\_\_next\_\_方法．*

*2. 语法：*

*class 迭代器名称:*

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

*返回下一个元素*

*如果没有元素了，则raise StopInteration()*

*3. 作用：隔离客户端代码，与可迭代对象(包含了聚合对象)的变化．*

*生成器：*

*1. 定义／特点：动态(循环一次计算一次返回一次)获取数据．*

*2.　生成器函数:函数中具有yield语句*

### 四月二十四（day17）

*day16 复习*

*生成器*

*-- 特点：惰性操作／延迟操作（循环一次，计算一次，返回一次．）*

*-- 本质: 迭代器　＋　可迭代对象*

*-- 生成器函数：使用　yield　返回数据(结果有多个)*

*使用　return　返回数据(结果有一个)*

*-- 生成器表达式:(对变量的操作　for 变量　in 可迭代对象* *if ..)*

*函数式编程*

*函数作为参数：将核心逻辑传入方法体，使方法适用性更广．*

*-- lambda 语法：　lambda 参数:方法体*

*函数作为返回值：*

list01 = [33, 4, 55, 6, 7, 8]

**def** fun01():

result = []

**for** item **in** list01:

**if** item > 5:

result.append(item)

**return** result

**def** fun02():

**for** item **in** list01:

**if** item > 5:

*# 生成器　告诉　客户端代码　的结果是右边item*

*# 客户端代码　告诉　生成器　的信息是左边value*

value = **yield** item

print(**"生成器收到:"**, value)

result01 = fun01()

result02 = fun02()

*# print(result02)# 返回值是生成器对象　generator*

*# print(dir(result02))*

*# for item in result02:*

*# print(item)*

item = result02.\_\_next\_\_()

print(item)

**while True**:

**try**:

*# item　是　yield 右侧返回的结果*

*# item = result02.\_\_next\_\_()*

*# item　是　yield 右侧返回的结果*

*# send　参数是　yield 左边*

item = result02.send(**"qtx"**)

print(**"客户端代码收到的是："**, item)

**except**:

**break**

*# 可迭代对象 迭代器*

**class** MyGenerator:

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

self.target = target

self.index = 0

*# 生成器具有当前方法，就是为了可以通过for获取结果．*

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

**return** self

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

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

**raise** StopIteration()

item = self.target[self.index]

self.index += 1

**return** item

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

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

**raise** StopIteration()

print(**"生成器收到："**, value)

item = self.target[self.index]

self.index += 1

**return** item

*# for item in MyGenerator([3,4,4,5,7]):*

*# print(item)*

iterator = MyGenerator([3, 4, 4, 5, 7])

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

print(item)

**while True**:

**try**:

*# item　是　yield 右侧返回的结果*

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

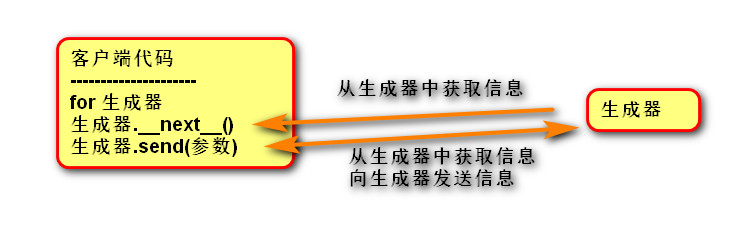
*# item　是　yield 右侧返回的结果*

*# send　参数是　yield 左边*

item = iterator.send(**"qtx"**)

print(**"客户端代码收到的是："**, item)

**except**:

 **break**

### 四月二十五（day18）

*"""*

*day17 复习*

*函数式编程：用一系列函数解决问题，函数是一等公平.*

*-- 函数作为参数：将核心逻辑传入方法体，使该方法适用性更为广泛．*

*传入数据:整数／小数／容器／自定义类的对象*

*传入逻辑：方法*

*-- 内置高阶函数:*

*filter －－＞* *find\_all*

*map －－＞* *select*

*sorted －－＞* *order\_by*

*max －－＞* *get\_max*

*min －－＞* *get\_min*

*-- 函数作为返回值:....*

*"""*

## 每日作业：

### 四月二十二（day15）

*# 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：查找编号是１０１的技能对象*

*# 　　－－　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．*

**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),

]

*# 因为需要查找的结果只有一个，所以使用return返回数据．*

*# 比使用yield返回数据，在调用者看来，更加方便吧．*

**def** find\_demo01(target):

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

**if** item.id == 101:

**return** item

s01 = find\_demo01(list\_skills)

print(s01.id)

**def** find\_demo02(target):

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

**if** item.cd == 0:

**yield** item

result = find\_demo02(list\_skills)

*# 不能获取指定结果*

*# 因为：此时生成器函数并没有计算处结果．*

*# print(result[1].name)*

*# for item in result:*

*# print(item.name)*

result= find\_demo02(list\_skills)

*# 通过生成器创建列表*

*# 由惰性查找(优势：节省内存)　转换为　立即查找(优势：灵活获取结果)*

result= list(result)

*# print(result[1].name)*

**def** find\_demo03(target):

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

**if** item.atk > 5:

**yield** item

*# 调用生成器函数，创建迭代器对象*

result = find\_demo03(list\_skills)

**for** item **in** result:*# \_\_next\_\_()*

print(item.name)

print(**"---------------"**)

*# 如果没有下一行代码，再次for使用过的生成器对象，不会再有结果．*

*# result = find\_demo03(list\_skills)*

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

print(item.name)

*# for item in find\_demo03(list\_skills):*

*# ....*

**def** find\_demo04(target):

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

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

**yield** item

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

print(item.name)

### 四月二十三（day16）

#### 熟练运用ListHelper.

*准备：*

*－－　创建敌人类（编号/姓名/攻击力/血量/攻击速度...）*

*－－　创建敌人列表*

*练习1：*

*1. 查找所有死人.*

*2. 查找编号是101的敌人*

*3. 查找所有活人.*

*4. 计算所有敌人攻击力总和*

*5. 查找所有攻击速度在５－－１０之间的敌人*

*6. 查找所有敌人的姓名*

**from** 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. 查找所有死人.*

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

print(item.name)

*# 2. 查找编号是101的敌人*

result = ListHelper.**first**(list01,**lambda** e:e.id == 101)

print(result.name)

*# 3. 查找所有活人.*

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

print(item.name)

*# 4. 计算所有敌人攻击力总和*

result = ListHelper.**sum**(list01,**lambda** e:e.atk)

print(result)

*# 5. 查找所有攻击速度在５－－１０之间的敌人*

**for** item **in** ListHelper.**find\_all**(list01,**lambda** e: 5 <= e.atk\_speed <= 10):

print(item.name)

*# 6. 查找所有敌人的姓名*

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

print(item)

*# 结果是生成器对象*

result = ListHelper.**select**(list01,**lambda** e:e.name)

*# 生成器　--> 列表*

result = list(result)

**for** item **in** result[1:3]:

print(item)

##### 1.解决的问题：获取满足条件的最后一个对象

*# 获取最后一个活人*

*# 获取攻击速度大于５的最后一个敌人*

*# def find\_demo01():*

*# for i in range(len(list01) - 1,-1,-1):*

*# if list01[i].hp >0:*

*# if xxx(list01[i]):*

*# return list01[i]*

*# def find\_demo02():*

*# for i in range(len(list01) - 1,-1,-1):*

*# if list01[i].atk\_speed >5:*

*# return list01[i]*

*# def xxx(item):*

*# return item.hp >0*

result = ListHelper.**last**(list01,**lambda** e:e.atk\_speed >5)

print(result.name)

##### 2.解决的问题：获取满足条件的对象总数

*# 获取具有生命值的对象总数*

*# 获取攻击速度小于２０的敌人总数*

*# def find\_demo01(target):*

*# count\_value = 0*

*# for item in target:*

*# # if item.hp >0:*

*# if xxx(item):*

*# count\_value += 1*

*# return count\_value*

*#*

*# def xxx(item):*

*# return item.hp >0*

count = ListHelper.**get\_count**(list01,**lambda** e:e.atk\_speed < 20)

print(count)

##### 3.解决的问题：判断列表中是否包含某个元素

*# 获取列表中是否具有死人*

*# 获取列表中是否具有攻击速度大于１０的敌人*

*# def demo01(target):*

*# for item in target:*

*# # if item.hp == 0:*

*# if xxx(item):*

*# return True*

*# return False*

*#*

*# def xxx(item):*

*# return item.hp == 0*

result = ListHelper.**exists**(list01,**lambda** e:e.hp == 0)

print(result)

##### 4.解决的问题：删除满足条件的所有对象

*# 删除所有死人*

*# 删除编号是101的敌人*

*# 删除攻击力小于５的敌人*

count = ListHelper.**delete\_all**(list01,**lambda** e:e.hp == 0)

print(count)

##### 5.解决的问题：获取满足条件的最大值

*# 获取血量最大的敌人*

*# 获取攻击力最强的*

*# def demo01(target):*

*# max\_value = target[0]*

*# for i in range(1, len(target)):*

*# # if max\_value.hp < target[i].hp:*

*# if xxx(max\_value) < xxx(target[i]):*

*# max\_value = target[i]*

*# return max\_value*

*#*

*# def xxx(item):*

*# return item.hp*

max = ListHelper.**get\_max**(list01,**lambda** e:e.hp)

print(max.name)

##### 解决的问题6.：根据指定条件升序排列列表

*# 按照血量升序排列*

*# 按照攻击力升序排列*

*# def demo01(target):*

*# for r in range(len(target)-1):*

*# for c in range(r+1,len(target)):*

*# # if target[r].hp > target[c].hp:*

*# if xxx(target[r]) > xxx(target[c]):*

*# target[r],target[c] = target[c],target[r]*

*#*

*# def demo02(target):*

*# for r in range(len(target)-1):*

*# for c in range(r+1,len(target)):*

*# if target[r].atk > target[c].atk:*

*# target[r],target[c] = target[c],target[r]*

*#*

*# def xxx(item):*

*# return item.hp*

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

**for** item **in** list01:

print(item.hp)

### 四月二十四（day17）

#### 练习１：获取技能列表中，攻击力最小的技能．

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

**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),

]

result = ListHelper.get\_min(list\_skills,**lambda** e:e.atk)

print(**"攻击力最小的敌人:"**,result.name)

result = min(list\_skills,key = **lambda** e:e.atk)

print(**"攻击力最小的敌人:"**,result.name)

#### 练习２：根据ｃｄ对技能列表进行降叙排列:

--------分别使用内置高阶函数和ListHelper实现．

ListHelper.**order\_by\_descending**(list\_skills,**lambda** e:e.cd)

**for** item **in** list\_skills:

print(**"order\_by\_descending降序："**,item.cd)

**for** item **in sorted**(list\_skills,key = **lambda** e:e.cd,reverse=**True**):

print(**"sorted降序："**,item.cd)

*# 如果使用**sorted排序*(*sorted不会改变原有列表*)*，要想修改原有列表，则使用下列代码．*

list\_skills**[:]** = **sorted**(list\_skills,key = **lambda** e:e.cd,reverse=**True**)

**for** item **in** list\_skills:

print(**"sorted降序（修改原有列表）："**,item.cd)

# 针对列表的自定义工具

**class** ListHelper:

@staticmethod

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

*"""*

*查找列表中满足条件的所有元素*

**:param** *target: 列表*

**:param** *func\_condition: 条件*

*函数／方法类型*

*－－　参数：列表中的元素*

*－－　返回值：是否满足条件bool值*

**:return***:*

*"""*

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

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

**yield** item

@staticmethod

**def first**(target, func\_condition):

*"""*

*查找列表中满足条件的第一个元素*

**:param** *target:*

**:param** *func\_condition:*

**:return***:*

*"""*

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

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

**return** item

@staticmethod

**def select**(target, func\_condition):

*"""*

*筛选列表中指定条件的数据*

**:param** *target:*

**:param** *func\_condition:*

**:return***:*

*"""*

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

*# yield xxx(item)*

**yield** func\_condition(item)

@staticmethod

**def sum**(target, func\_condition):

*"""*

*计算列表中指定条件的所有元素和*

**:param** *target:*

**:param** *func\_condition:*

**:return***:*

*"""*

sum\_value = 0

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

*# sum\_value += xxx(item)*

sum\_value += func\_condition(item)

**return** sum\_value

@staticmethod

**def last**(target,func\_condition):

*"""*

*查找满足条件的最后一个对象*

**:param** *target:*

**:param** *func\_condition:*

**:return***:*

*"""*

**for** i **in** range(len(target) - 1,-1,-1):

*# if xxx(list01[i]):*

**if** func\_condition(target[i]):

**return** target[i]

@staticmethod

**def get\_count**(target,func\_condition):

*"""*

*获取所有满足条件的对象总数*

**:param** *target:*

**:param** *func\_condition:*

**:return***:*

*"""*

count\_value = 0

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

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

count\_value += 1

**return** count\_value

@staticmethod

**def exists**(target,func\_condition):

*"""*

*判断是否包含满足条件的对象*

**:param** *target:*

**:param** *func\_condition:*

**:return***:*

*"""*

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

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

**return True**

**return False**

@staticmethod

**def delete\_all**(target,func\_condition):

*"""*

*删除满足条件的所有对象*

**:param** *target:*

**:param** *func\_condition:*

**:return***:*

*"""*

del\_count = 0

**for** i **in** range(len(target) - 1,-1,-1):

**if** func\_condition(target[i]):

**del** target[i]

del\_count += 1

**return** del\_count

@staticmethod

**def get\_max**(target,func\_condition):

*"""*

*获取指定条件的最大对象(第一个)*

**:param** *target:*

**:param** *func\_condition:*

**:return***:*

*"""*

max\_value = target[0]

**for** i **in** range(1, len(target)):

*# if max\_value.hp < target[i].hp:*

**if** func\_condition(max\_value) < func\_condition(target[i]):

max\_value = target[i]

**return** max\_value

@staticmethod

**def order\_by**(target,func\_condition):

*"""*

*根据指定条件升序排列*

**:param** *target:*

**:param** *func\_condition:*

**:return***:*

*"""*

**for** r **in** range(len(target)-1):

**for** c **in** range(r+1,len(target)):

*# if target[r].hp > target[c].hp:*

**if** func\_condition(target[r]) > func\_condition(target[c]):

target[r],target[c] = target[c],target[r]

@staticmethod

**def get\_min**(target, func\_condition):

*"""*

*获取指定条件的最小对象(第一个)*

**:param** *target:*

**:param** *func\_condition:*

**:return***:*

*"""*

min\_value = target[0]

**for** i **in** range(1, len(target)):

*# if min\_value.hp > target[i].hp:*

**if** func\_condition(min\_value) > func\_condition(target[i]):

min\_value = target[i]

**return** min\_value

@staticmethod

**def order\_by\_descending**(target,func\_condition):

*"""*

*根据指定条件降序排列*

**:param** *target:*

**:param** *func\_condition:*

**:return***:*

*"""*

**for** r **in** range(len(target)-1):

**for** c **in** range(r+1,len(target)):

**if** func\_condition(target[r]) < func\_condition(target[c]):

target[r],target[c] = target[c],target[r]

# 面试

## 排序题

### 方案一：

推导：

*# def sort03(target,xxx):*

*# for r in range(len(target) - 1):*

*# for c in range(r + 1, len(target)):*

*# # if target[r].atk < target[c].atk:*

*# if xxx(target[r],target[c]):*

*# target[r], target[c] = target[c], target[r]*

*#*

*# # def xxx(item01,item02):*

*# # return item01.atk < item02.atk*

**def** sort(target,func\_condition):

*"""*

*万能排序*

**:param** *target: 需要排序的数据*

**:param** *func\_condition: 排序的逻辑*

*func\_condition 类型是函数*

*参数是列表中两个元素*

*返回值是比较的结果*

*方法提是比较的条件*

**:return***:*

*"""*

**for** r **in** range(len(target) - 1):

**for** c **in** range(r + 1, len(target)):

**if** func\_condition(target[r],target[c]):

target[r], target[c] = target[c], target[r]

list01 = [3,34,5,6,8]

sort(list01,**lambda** e1,e2:e1 < e2)

print(**"万能排序："**,list01)

### 方案二：

**def** sort01(target):

**for** r **in** range(len(target) - 1):

**for** c **in** range(r + 1, len(target)):

**if** target[r] > target[c]:

target[r], target[c] = target[c], target[r]

list01 = [2,6,4,4,5,7]

sort01(list01)

**for** item **in** list01:

print(**"sort01升序："**,item)

### 方案三：

**def** sort02(target):

**for** r **in** range(len(target) - 1):

**for** c **in** range(r + 1, len(target)):

**if** target[r] < target[c]:

target[r], target[c] = target[c], target[r]

list01 = [2,6,4,4,5,7]

sort02(list01)

**for** item **in** list01:

print(**"sort02降序："**,item)