# 程序结构

## 模块 Module

### 定义

包含一系列数据、函数、类的文件，通常以.py结尾。（就是每天做的py文件）

### 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\*\*\*

#### 方式1：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

#### 方式2：from import

1. 语法：

from 模块名 import 成员名[ as 别名1]

1. 作用：将模块内的一个或多个成员导入到当前模块的作用域中。
2. 使用：直接使用成员名

#### 方式3：from import \* \*代表全部

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。

模块

练习：

将student\_info\_manager.py分解为4个模块

-- bll.py --> 业务business 逻辑logic 层layer

存储Controller类

-- usl.py --> 用户user 显示show 层layer

存储View类

-- model.py --> 存储数据模型Model

-- main.py --> 存储入口代码

"""

# 建议设置项目根目录

# -- 在day14文件夹上,右键选择"Mark Directory as"

# -- 在选择"Sources Root"（标蓝后调用时有提示）

“””

module01.py 一个单独的文件

def func01():

print("func01")

class MyClass:

def func02(self):

print("func02")

demo03.py 导入module.py的文件

# **导入方式1**： 本质1：创建变量module01关联该模块

# 语法：

# import 模块

# 模块.成员

# 适合：面向过程的技术(全局变量、函数)

import module01 适合面向过程

module01.func01()

# m01 = module01.MyClass() 不适合面向过程

# m01.func02()

# **导入方式2**： 本质2：将该模块成员导入到当前作用域中

# 语法：from 文件名 import 内容

# 适合：面向对象的技术(类)

#**alt +ENTER 快捷键操作 推荐做法\*\*\*\*\*\***

# from module01 import func01不适合面向对象，不符合创建对象方式

# func01()

from module01 import MyClass 适合面向对象

# m01 = MyClass() 符合创建对象方式

# m01.func02()

# 导入方式3： 本质2 将该模块所有成员导入到当前作用域中

# 语法：from 文件名 import \*

# 适合：面向对象的技术(类)

from module01 import \*

func01()

m01 = MyClass()

m01.func02()

### 模块变量

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

\_\_doc\_\_变量：文档字符串。 “”” 最顶端””” import.demo01 print(demo01.\_\_doc\_\_)

\_\_file\_\_变量：模块对应的文件(完整)路径名。 print(demo01.\_\_file\_\_) / print(01.\_\_file\_\_)

\_\_name\_\_变量：模块自身名字，可以判断是否为主模块。

print(demo01.\_\_name\_\_) #demo01

print(\_\_name\_\_) #\_\_main\_\_ 应用if\_\_name\_\_ = “\_\_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。

# 模块变量

# -- 获取文档字符串

print(\_\_doc\_\_) 应用一

import demo01

print(demo01.\_\_doc\_\_)应用二

# -- 获取文件完整路径

print(\_\_doc\_\_)应用一

# /home/tarena/2005/day15/demo03.py

print(demo01.\_\_file\_\_)应用二

# -- 获取模块名称（可变的）

print(\_\_name\_\_) # 只有当前模块是主模块时,才是"\_\_main\_\_" 应用一

print(demo01.\_\_name\_\_) # 被导入的模块是模块名demo01 应用二

快捷键main

if \_\_name\_\_ == '\_\_main\_\_':

print("我是主模块")

### 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。

import demo01

import demo01

import demo01

导入三次，但只执行一次

### 分类

1. 内置模块(builtins)，在解析器的内部可以直接使用。（不需要导入 print）
2. 标准库模块，安装Python时已安装且可直接使用。（必须导入才可使用 random）

时间模块

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

"""

对时间的处理

https://www.runoob.com/python3/python3-date-time.html

"""

import time

# 1. 人类时间

# 时间元组: 年/月/日/时/分/秒/星期/一年的第几天/夏令时

print(time.localtime())

# 2. 计算机时间(一个数省内存)

# 时间戳: 从1970年1月1日 0时0分0秒到现在经过的秒数

print(time.time()) # 1592377581.6000147

# 3. 时间戳 --> 时间元组

tuple\_time = time.localtime(1592377581.6000147)

# 4. 时间元组 --> 时间戳

print(time.mktime(tuple\_time))

# 5. 时间元组 --> 字符串

# 语法:字符串 = time.strftime(格式,时间元组)

# 20/06/17 15:06:21

print(time.strftime("%y/%m/%d %H:%M:%S",tuple\_time))

# 2020/06/17 15:06:21

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

# 6. 字符串 --> 时间元组

# 语法:时间元组 = time.strptime(时间字符串,格式)

print(time.strptime("2020/06/17 15:06:21","%Y/%m/%d %H:%M:%S"))

## 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.模块名 [as 模块新名] #模块别名可以起也可以不起

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]

#**alt +ENTER 快捷键操作**

# 导入包内的所有子包和模块

from 包名 import \*

from 包名.模块名 import \*

"""

Python程序结构

分而治之，结构清晰

包(文件夹) 文件夹内多了一个init文件（不是构造函数，是标志），导入包时执行一次

模块(文件)

类

函数

语句

导入模块是否成功的唯一标准：

导入路径 + sys.path = 实际路径

导入路径 common （from common import \*）

sys.path home/tarena/month01/code/day15/my\_projcet

实际根目录/home/tarena/month01/code/day15/my\_projcet/skill\_system

import sys

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

print(sys.path)

"""

#module01.py 被导入的模块

def func01():

print("func01执行喽")

import sys

# 真正的项目根目录：主模块所在文件夹

# 主模块：第一次运行的py文件

print(sys.path)

# # 方法1

# # import 路径.模块名

# import package01.package02.module01 as m

# package01.package02.module01.func01() -- > m.func01()

# # 方法2

# from 路径.模块名 import 成员

from package01.package02.module01 import func01

func01()

# # 方法3

# # from 路径.模块名 import \*

# from package01.package02.module01 import \*

# func01()

### \_\_init\_\_.py 文件 (不是构造函数)

是包内必须存在的文件

会在包加载时被自动调用

"""

特殊导入方式 导入包不导入成员名 非主流极个别情况使用

设置包的\_\_init\_\_.py文件

"""

# 方式1：import 包

# 必须设置\_\_init\_\_.py文件：import package01.package02.module01

# import package01.package02 as p

#

# p.module01.func01()

# 方式2：from 包 import 包

# 必须设置\_\_init\_\_.py文件：from package01.package02 import module01

# from package01 import package02

# package02.module01.func01()

# 方式3：from 包 import \*

# 必须设置\_\_init\_\_.py文件（定义可导出成员module01）：\_\_all\_\_ = ["module01"]

from package01.package02 import \*

module01.func01()

#### \_\_all\_\_

记录from 包 import \* 语句需要导入的模块

案例：

1. 根据下列结构，创建包与模块。

my\_ project /

main.py

common/

\_\_init\_\_.py

list\_helper.py

skill\_system/

\_\_init\_\_.py

skill\_deployer.py

skill\_manager.py

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

# from common.list\_helper import ListHelper #方法一

#from common import \* #方法二 在使用第三方模块时使用本方法

# 在\_\_init\_\_.py中配置

# \_\_all\_\_ = ["list\_helper"]

# 导入模块是否成功的唯一标准：

# 导入路径 + sys.path = 实际路径

#程序执行的根目录skill\_system与认为的根目录my\_projcet不一致时,会出错

# 1.导入路径common

# 2.sys.path

import sys

# sys.path.append("/home/tarena/month01/code/day15/my\_projcet")

print(sys.path)

# 3.实际路径

#能够导入模块的实际目录/home/tarena/month01/code/day15/my\_projcet/common

#程序执行的根目录 /home/tarena/month01/code/day15/my\_projcet/skill\_system

# 导入路径是在根目录内开始写起来的即my\_projcet/skill\_system,

# 但是common.list.py不在skill\_system里,而是在my\_projcet/common里

# mark as SourceRoot,相当于强制设置了根目录

class SkillDeployer:

def func02(self):

print("func02执行了")

# ListHelper.func03() #方法一

list\_helper.ListHelper.func03() #方法二

### 搜索顺序

内置模块 builtins

sys.path 提供的路径

"""

总结 -- 包和模块

1. 为什么要有模块和包?

为了结构更加清晰

2. 导入方法

(1) import 路径.模块 as 别名

别名.成员

(2) from 路径.模块 import 成员

直接使用成员

3. 是否成功的唯一标准

导入路径 + sys.path = 真实路径

主模块:第一次执行的模块

根目录:主模块所在文件夹

4. 模块变量

"""

# 异常处理Error

## 异常

1. 定义：运行时检测到的错误。 （运行时数据超过了有效范围引发的逻辑错误）
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。

链接a 调b 调 c，最后一个链接为异常处。

1. 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。（爹是异常，错误是一种异常）

## 处理

## 原则：就近处理

1. 语法：

方式一：对症下药

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

方式二：起死回生（开发项目中常用）

try:

except Exception [as 变量3]： 只写except：

不是以上错误类型的处理语句

方式三：无论是否发生异常都执行的语句，一定执行的逻辑

try:（打开、处理文件）

finally:（关闭文件）

无论是否发生异常的语句

方式四：

try: 是在检测错误，else是没有出错才执行的逻辑

except：

else:

未发生异常的语句

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。

"""

异常处理

目的:错误状态(向上翻) --> 正常状态(向下走)

核心价值：保障程序按照既定流程执行，不紊乱 \*\*\*\*\*\*

"""

def div\_apple(apple\_count):

person\_count = int(input("请输入人数:"))

result = apple\_count / person\_count

print("每人%d个苹果" % (result))

# 写法1:对症下药(官方建议)

"""

# 检测可能出错的代码

try:

div\_apple(10)

# 定位错误代码

except ValueError:

print("不能输入非整数类型")

except ZeroDivisionError:

print("不能输入零")

except StopIteration: (与raise StopIteration配合使用)

print("不能输入零")

print("后续逻辑")

"""

# 写法2:包治百病(常用)

"""

# 检测可能出错的代码

try:

div\_apple(10)

# 定位全部错误代码 Exception：所有异常的基类，抽象的异常

# except Exception:

except: #简写与except Exception:功能一致

print("出错喽")

print("后续逻辑")

"""

# 写法3:一定执行的逻辑

"""

# 检测可能出错的代码

try:

div\_apple(10)

# 打开文件

# 处理文件(发生错误)

# 最终 无论是否发生错误,一定执行的代码.

finally:

# 关闭文件

print("分苹果结束")

print("后续逻辑")

"""

# 写法4:没有出错才执行的逻辑

# 检测可能出错的代码

try:

div\_apple(10)

except:

print("出错啦")

# 没有出错才执行的逻辑

else:

print("分苹果成功啦")

print("后续逻辑")手机屏幕的截图

描述已自动生成

## 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。

"""

raise

"""

class Wife:

def \_\_init\_\_(self, age=0):

self.age = age # 2

@property

def age(self):

return self.\_\_age

@age.setter

def age(self, value): # 3

if 22 <= value <= 30:

self.\_\_age = value

else:

# 主动抛出异常(快速传递错误消息#3—>#1)

raise Exception("年龄超过范围",1001)

备注：Exception(\*args) —>传递信息可以随意设置。

try:

w01 = Wife(300) # 1

print(w01.age)

except Exception as e: # 创建变量e,通过e变量操作抛出的异常对象

print(e.args)

print("出错啦")

# 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。

#注意：迭代过程重复，但是迭代内容不同，不是重复拿某一个元素

""" 注意：迭代过程重复，但是迭代内容不同，不是重复拿某一个元素

迭代 iter: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。

可迭代iterable:能够完成迭代过程的对象.（如message、列表等）

迭代器iterator:实施迭代过程的对象（可以获取下一个元素）

练习:exercise01

"""

message = "我是齐天大圣孙悟空"

# for item in message:

# print(item)

前期用法总结：语法上为for，原理上为迭代，开发时使用本方法

# for 循环原理（包括下述1.2.3步）学习原理应付面试，开发中不使用

message = "我是齐天大圣孙悟空"

iterator = message.\_\_iter\_\_() # 1. 获取迭代器对象 [容器].\_\_iter\_\_()

while True:

try:

item = iterator.\_\_next\_\_() # 2. 获取下一个元素（没有元素后发生异常）

print(item)

except StopIteration: # 3. 如果没有元素,则停止循环（捕获异常）

break

# **面试题**:可以参与for循环的条件是什么? 答案一或答案二

# 答案一：能够获取迭代器对象(才是可迭代对象—能够被重复如message)

# 答案二语法具象解释：具有\_\_iter\_\_函数 的对象

## 可迭代对象iterable

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

-- 创建：

class 可迭代对象名称:

  def \_\_iter\_\_(self):

      return 迭代器

-- 使用：

for 变量名 in 可迭代对象:

语句

1. 原理：

迭代器 = 可迭代对象.\_\_iter\_\_()

while True:

try:

print(迭代器.\_\_next\_\_())

except StopIteration:

break

## 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（开发时不用此代码而用yield，但是只有通过迭代器才能理解生成器惰性原理）

让自定义对象参与for循环

迭代自定义对象

练习:exercise02,03

"""

class SkillIterator:

def \_\_init\_\_(self, data): 创建对象

self.data = data

self.index = -1

def \_\_next\_\_(self): 步骤4：iterator.\_\_next\_\_()通过while True循环调用此方法

self.index += 1 注：因为是被循环调用（0、1、2），不用自身循环，自增实现即可

if self.index < len(self.data): 索引不要越界，设定循环结束条件

return self.data[self.index]

else:

raise StopIteration() 步骤5：让死循环停止

class SkillManager:

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

self.\_\_skills = []

def add\_skill(self, skill):

self.\_\_skills.append(skill)

def \_\_iter\_\_(self): #步骤2：创建iter方法

#iterator = SkillIterator( )

#return iterator

上两行代码简化为下面一行

return SkillIterator(self.\_\_skills) return返回给iterator的对象必须能够.\_\_next\_\_()

skillmanager中创建skilliterator对象，通过构造函数将self.skills传递给skilliterator

manager = SkillManager()

manager.add\_skill("降龙十八掌")

manager.add\_skill("六脉神剑")

manager.add\_skill("乾坤大挪移")

# 迭代自定义对象SkillManager

#SkillManager内没有\_\_iter\_\_()，不是可迭代对象，不能参与for循环

# for skill in manager:

# print(skill)

为了for循环能执行需执行如下步骤 自己创建类和可迭代对象后可参与for循环：

iterator = manager.\_\_iter\_\_() 步骤1：要实现此步，必须在SkillManager类里创建iter方法

while True:

try:

item = iterator.\_\_next\_\_() 步骤3：iterator返回的对象执行.\_\_next\_\_()

print(item) # 降龙十八掌

except StopIteration: 接受raise StopIteration()

break

"""

迭代图形管理器

画出架构设计图

"""

class GraphicIterator:

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

self.data = data

self.index = -1

def \_\_next\_\_(self): #重写

self.index += 1

if self.index < len(self.data):

return self.data[self.index]

else:

raise StopIteration()

class GraphicManager:

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

self.all\_graphic = []

def add\_graphic(self, graphic):

self.all\_graphic.append(graphic)

def \_\_iter\_\_(self): #重写

return GraphicIterator(self.all\_graphic)

manager = GraphicManager()

manager.add\_graphic("圆形")

manager.add\_graphic("矩形")

manager.add\_graphic("三角形")

for item in manager:

print(item)

手机屏幕的截图

描述已自动生成迭代架构图（Python语法不需要做父类Ierable和Iterator，直接重写子类就可以）

# 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。

"""

MyRange1.0

迭代器实现MyRange类

"""

class MyRangeIterator:# 迭代器

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

self.number = -1

self.stop = stop

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

self.number += 1

if self.number < self.stop:

return self.number

else:

raise StopIteration()

class MyRange:

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

self.end = end

def \_\_iter\_\_(self):# 可迭代对象

return MyRangeIterator(self.end)

# for item in MyRange(5): # 0~4

# print(item)

# 可以生成撑爆内存的数字

# 循环一次 计算一次 返回一次

for item in MyRange(9999999999999999999999999999999):

print(item)

#整数生成器，通过以上MyRange的生成思想看懂range的原理

for item in range(9999999999999999999999999999999):

print(item)

"""

迭代器 --> yield 迭代器被yield取代，删除class SkillIterator \*\*\*\*\*

核心价值：生成海量大数据 练习:exercise05,06

"""

class SkillManager:

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

self.\_\_skills = []

def add\_skill(self, skill):

self.\_\_skills.append(skill)

# def \_\_iter\_\_(self): 写死了 无循环

# print("准备")

# yield self.\_\_skills[0]

# print("准备")

# yield self.\_\_skills[1]

# print("准备")

# yield self.\_\_skills[2]

# 写了3个yield，但是执行4个yield，最后一个隐藏的yield执行抛出异常

def \_\_iter\_\_(self): 写活了 有循环

for skill in self.\_\_skills:

print("准备")

yield skill

# yield在做一个标记，让现在看见的代码，生成删除迭代器之前的代码

# yield 生成迭代器代码的大致规则:

# 1. 将yield之前的代码定义到\_\_next\_\_方法中

# 2. 将yield之后的数据作为\_\_next\_\_方法返回值

manager = SkillManager()

manager.add\_skill("降龙十八掌")

manager.add\_skill("六脉神剑")

manager.add\_skill("乾坤大挪移")

# 迭代自定义对象

for skill in manager: 开发使用

print(skill)

# iterator = manager.\_\_iter\_\_() 查看原理 步骤1

# while True:

# try:

# item = iterator.\_\_next\_\_() 步骤2

# print(item) # 降龙十八掌

# except StopIteration:

# break

# 以上代码执行的过程现象:

# 调用\_\_iter\_\_方法,但是不执行 步骤1（因为iter的代码被yield切割到next里）

# 调用\_\_next\_\_方法,执行\_\_iter\_\_方法 步骤2 (类内\_\_next\_\_被删除了)

# 执行到yield返回,当再次调用\_\_next\_\_方法继续执行\_\_iter\_\_方法

"""

MyRange2.0

使用 yield 代替迭代器

"""

class MyRange:

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

self.end = end

# 使用yield 返回0~4

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

start = 0

while start < self.end:

yield start

start += 1

for item in MyRange(5):

print(item)

"""

yield --> 生成器

MyRange3.0

类 --> 函数 类太臃肿了，进一步简化为函数

练习:exercise08/09

"""

"""

生成器源码.pyc（结合for原理代码看）

class Generator:# 生成器 = 可迭代对象 + 迭代器(生成器的核心即迭代器)

def \_\_iter\_\_(self): # 可迭代对象

return self #range与iterator是一个对象，所以返回self

def \_\_next\_\_(self): # 迭代器 核心是next ，只有next才可以被for

print(“第n部分”)

return “结果n”

"""

def my\_range(end):

start = 0

while start < end:

yield start

start += 1

for item in my\_range(5):

print(item)

# 延迟操作/惰性操作

# 生成器理念：循环一次 计算一次 返回一次 （不存储数据）

#for原理代码（结合生成器源码与debug看）

range = my\_range(5) 调用my\_range函数返回生成器对象（推算数据）

iterator = range.\_\_iter\_\_() #range与iterator是一个对象，所以返回self

while True:

try:

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

print(item)

except StopIteration:

break

"""

函数返回结果:

yield 数据 -- 多个

return 数据 -- 单个

无返回值

"""

Day17 homework

if \_\_name\_\_ == '\_\_main\_\_':

# 调用yield（两种方式，方式一常用）

# -- 方式一：通过for执行生成器 优点：节省内存，惰性延迟操作

result = find\_skills\_by\_atk\_rate()

for item in result: 执行第一次

print(item.\_\_dict\_\_)

# 生成器缺点1:只能用一次

# for item in result: 本次调用不再执行第二次

# print(item.\_\_dict\_\_)

# 生成器缺点2:不能使用索引/切片

# result[-1]

# -- 方式二：生成器 -->转换为容器 优点：可以反复调用，能够进行索引/切片

# 容器缺点:将所有数据加载到内存中,占用可能过多.（容器种类根据需要自由选择）

result = tuple(find\_skills\_by\_cost\_sp\_and\_duration())

for item in result:

print(item.\_\_dict\_\_)

# 调用return

skill = find\_skills\_by\_name()

print(skill.\_\_dict\_\_)

emp = get\_max\_skill\_by\_cost\_sp()

print(emp.\_\_dict\_\_)

# 调用无返回值

ascending\_order\_by\_duration()

for item in list\_skills:

print(item.\_\_dict\_\_)

## 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

"""

list01 = [54, 5, 6, 76, 8, 9]

# 遍历元素 -- 读取

for item in list01:

print(item)

# 遍历索引 -- 修改

# for i in range(len(list01)):

# list01[i] = 0

# 需求:修改奇数为0

# for i in range(len(list01)):

# if list01[i] % 2:

# list01[i] = 0

print(list01)

通过上述前期代码引出枚举函数语法，同时获取索引和元素

# (索引, 元素)

# for item in enumerate(list01): 内置生成器，用一次给你一个

for i, element in enumerate(list01):

if element % 2:

list01[i] = 0

print(list01)

exercise：顽强的意志力，无所畏惧的勇气

dict01 = {"a": "A", "b": "B"}

for i, (k, v) in enumerate(dict01.items()):

print(i, k, v)

### 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 竖向取

"""

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

list02 = ["A", "B", "C"]

for item in zip(list01, list02):

print(item)

# 练习:通过zip实现矩阵转置

list01 = [

[1, 2, 3, 4],

[5, 6, 7, 8],

[9, 10, 11, 12],

[13, 14, 15, 16],

]

list02 = []

for item in zip(\*list01): \*号将实参列表拆开

list02.append(list(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] )

Vs 列表推导式[ ]

"""

生成器函数（必须有def函数才能使用yield）

适用场景：给其他人用

生成器表达式

适用场景：给自己使用

exercise 03/04

"""

list01 = [54, 65.5, True, "a", False, "b", 3, "c"]

# 需求：查找所有字符串（类型是str）

# 传统列表

# list\_result = []

# for item in list01:

# if type(item) == str:

# list\_result.append(item) 没有def函数不能使用yield

# print(list\_result)

# 列表推导式（由传统方式思考后简化的方式）

list\_result = [item for item in list01 if type(item) == str] list\_result为列表（存储）

for item in list\_result:

print(item)

# 生成器表达式（核心思维方式惰性操作）

list\_result = (item for item in list01 list\_result为生成器（推算）

if type(item) == str)

for item in list\_result:

print(item)

# 生成器函数

def is\_str\_type(list\_target):

for item in list\_target:

if type(item) == str:

yield start

for item in is\_str\_type (list01):

print(item)

#转化为生成器表达式写法

str\_type = (item for item in list\_target if type(item) == str)

for item in str\_type :

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。

""" day 17 demo04

函数式编程 - 语法

理论支柱：

将函数赋值给变量,通过变量调用函数

应用：

将函数赋值给参数,通过参数调用函数

（价值：通过参数调用函数,可以将(核心)逻辑注入到函数中.）

"""

def func01():

print("func01执行了")

# 将函数func01赋值给变量 注意不是func01(),因为加了括号就是函数返回值

a = func01 不是func01()，函数不执行

# 通过变量调用函数，只是一种语法，无意义

a()

def func02():

print("func02执行了")

# 价值:通过参数调用函数,可以将(核心)逻辑注入到函数中.

def func03(a):

print("func03执行了")

a()# 执行? a抽象、灵活，执行结果，取决于传递的实参

func03(func01)

func03(func02)

""" day 17 demo05

函数式编程 - 思想 （第三阶段js也会用,叫法为回调，思想即函数式编程）

面向对象

封装：分

继承: 隔

多态：做

函数式编程 与面向对象异曲同工

分：将变化点分为多个函数（通用与变化的）

隔: 使用参数隔离具体变化的函数（参数func）

做：通常使用lambda定义变化函数（代替def condition）

目的：创造更有弹性，更灵活的软件

练习:exercise05、exercise06

"""

函数式编程思想适用场景：

多个需求的函数代码主体结构极相似，但是部分代码（核心算法逻辑）不一致，每个需求单独写的代码臃肿，因此采用分隔做的思想，把变化点和通用函数分别提出。

函数式编程执行步骤

1. 定义函数,完成需求.

2. 将变化点定义为函数

将通用代码定义为函数

3. 通用函数使用参数隔离变化点

4. 将通用函数移动到IterableHelper类中

5. 在当前模块中调用通用函数(lambda)

list01 = [42,45,5,66,7,89]

# 需求1：定义函数,获取所有大于10的数字

def find01():

for item in list01:

if item > 10: 变化点

yield item

# 需求2：定义函数,获取所有偶数

def find02():

for item in list01:

if item % 2 == 0: 变化点

yield item

for number in find01():

print(number)

# －－－－－－－－－－－－－－－－对原需求函数分隔做拆分，适应性更广

# 提取变化函数 (因为条件无穷无尽，开发时用lambda匿名函数替换)

def condition01(item):

return item > 10

def condition02(item):

return item % 2 == 0

# －－－－－－－－－－－－－－－－

# 提取通用函数 -- 万能查找（将变化条件注入通用函数，开发减少循环使用）

def find(func):#　创建了钩子 参数func隔离condition1与condition2

for item in list01:

# if item % 2 == 0: 原需求

# if condition02(item): 变化语法形式与原需求一致

if func(item):# 拉起钩子(执行条件) 每次for循环时拉起

yield item

for number in find(condition02):# 向钩子上挂条件（将变化条件注入通用函数）

print(number)

for number in find(lambda item: item % 2 == 0): # 向钩子上挂条件

print(number)

#common.iterable\_tools将通用函数放入IterableHelpe类，形成可迭代对象工具

from common.iterable\_tools import IterableHelper

# for number in IterableHelper.find(list01, condition01): 静态方法，类.调用

注：condition01函数传给find类方法，类方法内部调用执行condition01()函数

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

print(number)

"""

common/

iterable\_tools.py

可迭代对象工具

"""

class IterableHelper:

"""

可迭代对象助手/类（以后开发时只用不做）

自定义高阶函数

"""

函数式编程意义（对比内置函数）

# 1. 教学意义：深刻掌握函数式编程思想

# 　　分、隔、做

# 2. 实用价值：在开发过程中,不断壮大自定义高阶函数

# 功能更强大，节约30%代码量

# 3. 利于面试：精通函数式编程

常常遇到xx功能,发现实现方式大部分逻辑相同,只是核心xx不同,

我就将共性提取到xx类中,放到单独的模块中.

在各种项目中,直接导入使用.

思想来源于微软Linq技术（不支持跨平台，不是Python）

详细说法如下：

开发过程中，经常在一个容器中查找单个元素，发现主体部分一样，核心逻辑不同，将这个功能写成一个通用的万能的查找方法，放到了一个类中，放到了一个模块中，甚至包中，每次开发时都把这个包，模块，功能放到新项目里，在某一个地方导入这个包，调用功能，不再写循环，一句话可以完成查找任务。

@staticmethod （静态方法，不操作变量，最为独立，不需要隐式传参）

def find\_all(iterable, func):

for item in iterable:

if func(item):

yield item

@staticmethod

def find\_single(iterable, func):

for emp in iterable:

if func(emp):

return emp

@staticmethod

def select(iterable, func):

for emp in iterable:

yield func(emp)

@staticmethod

def delete\_all(iterable, func):

count = 0

for i in range(len(iterable) - 1, -1, -1):

# if iterable[i].money < 15000:

# $调用传入的函数lambda

if func(iterable[i]):

del iterable[i]

count += 1

return count

@staticmethod

def get\_max(iterable, func):

max\_value = iterable[0]

for i in range(1, len(iterable)):

# if max\_value.money < iterable[i].money:

# if xx(max\_value) < xx(iterable[i]):

if func(max\_value) < func(iterable[i]): #一个参数，两次调用func

max\_value = iterable[i]

return max\_value

@staticmethod

def ascending\_order(iterable, func):

for r in range(len(iterable) - 1):

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

# if iterable[r].money > iterable[c].money:

if func(iterable[r]) > func(iterable[c]):

iterable[r], iterable[c] = iterable[c], iterable[r]

# def xx(item):

# return item.money

条件可以简化为lambda

# lambda item:item.money

"""review.py

编程范式（开发中灵活运用，突破变化）

面向过程

考虑问题从步骤(细节/实现)出发（一个函数实现功能）

先做 -- 后用(def函数 直接调用 其他def函数，不考虑参数)

函数式编程： - 功能

"封装"： 分 -- 多个函数 [先用]

通用函数 (思想层面：姓名是xx的员工 编号是xx的员工...)

"继承"： 隔 -- 参数 ------

func 对小功能抽象（排序）

"多态"： 做 -- lambda [后做]

姓名是xx的员工 编号是xx的员工

面向对象 - 架构（大的问题，软件架构）

封装： 分 -- 多个类 [先用]

老张 (思想层面：汽车 火车...)

继承： 隔 -- 父类 ------

交通工具 对大职责抽象，一个类好多函数

多态：做 -- 子类重写 [后做]

汽车 火车

"""

## 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 表达式

匿名函数语法:

lambda 参数: 函数体

def可转换为lambda写法有4种

def不能转换为lambda写法有2种

1. lambda表达式不能赋值

2. lambda 函数体只能有一条语句

应用:作为实参

作为方法的实参，代替变化的条件函数def condition（）

# def xx(item):

# return item.money

# lambda item:item.money 转换为lambda

"""

# def能转换为lambda写法

# 写法1:有参数,有返回值

# def func01(p1, p2): 普通写法1

# return p1 > p2

# print(func01(10, 5))

func01 = lambda p1, p2: p1 > p2 lambda写法1 （不要def、括号、return）

print(func01(10, 5))

# 写法2:有参数,无返回值

# def func02(p1): 普通写法2

# print("参数是:",p1)

# func02(10)

func02 = lambda p1: print("参数是:", p1) lambda写法2 （不要def、括号）

func02(10)

# 写法3:无参数,有返回值

# def func03(): 普通写法3

# return "结果"

# result = func03()

# print(result)

func03 = lambda: "结果" lambda写法3 （不要def、括号、return）

result = func03()

print(result)

# 写法4:无参数,无返回值 普通写法4

# def func04():

# print("func04执行喽")

#

# func04()

func04 = lambda: print("func04执行喽") lambda写法4 （不要def、括号）

func04()

# def不能转换为lambda写法

普通写法1：

def func05(p1):

p1[0] = 100

list01 = [10]

func05(list01)

print(list01[0]) # 100

# lambda不支持写法1. lambda表达式不能赋值

# func05 = lambda p1:p1[0] = 100 报错

普通写法2：

def func06():

for i in range(5):

print(i)

func06()

# lambda不支持写法2. lambda 函数体只能有一条语句

# func06 = lambda :for i in range(5): 两条语句报错

# print(i)

### 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

"""

from common.iterable\_tools import IterableHelper

class EmployeeModel:

def \_\_init\_\_(self, eid=0, did=0, name="", money=0.0):

self.eid = eid

self.did = did

self.name = name

self.money = money

list\_employee = [

EmployeeModel(1001, 9003, "林玉玲", 13000),

EmployeeModel(1002, 9005, "王荆轲", 16000),

EmployeeModel(1003, 9003, "刘岳浩", 11000),

EmployeeModel(1004, 9007, "冯舜禹", 17000),

EmployeeModel(1005, 9005, "曹海欧", 15000),

EmployeeModel(1006, 9005, "魏鑫珑", 12000),

]

# 1. map : 映射

#内置高阶函数写法：语法map(func,iterable) -> 返回值为生成器

for item in map(lambda item: item.name, list\_employee):

print(item)

#函数式编程写法：返回值为生成器

for item in IterableHelper.select(list\_employee,lambda item:item.name):

print(item)

# 2.filter:过滤

#内置高阶函数写法：filter(func,iterable) -> 返回值为生成器

for item in filter(lambda emp: emp.money > 15000, list\_employee):

print(item.\_\_dict\_\_)

#函数式编程写法：返回值为生成器

for item in IterableHelper.find\_all(list\_employee,lambda emp:emp.money > 15000):

print(item.\_\_dict\_\_)

# 3.max 4.min : 最大最小

#内置高阶函数写法： 极值 = max(容器, key=函数) -> 返回值为新值

max\_emp = max(list\_employee, key=lambda item: item.money)

print(max\_emp.\_\_dict\_\_)

#函数式编程写法：返回值为单个数据

max\_emp =IterableHelper.get\_max(list\_employee,lambda item:item.money)

print(max\_emp.\_\_dict\_\_)

# 5. 排序

# -- 升序

#内置高阶函数写法：排序结果 = sorted(容器, key=函数) -> 返回值为新列表

result = sorted(list\_employee, key=lambda e: e.money)

print(result)

#函数式编程写法：无返回值，对原容器修改

# IterableHelper.ascending\_order(list\_employee,lambda e:e.money)

# print(list\_employee)

# -- 降序

# 内置高阶函数写法：排序结果 = sorted(容器, key=函数, reverse=True)

result = sorted(list\_employee, key=lambda e: e.money, reverse=True)

print(result)

## 函数作为返回值

逻辑连续，当内部函数被调用时，不脱离当前的逻辑。

### 闭包

1. 三要素：

-- 必须有一个内嵌函数。

-- 内嵌函数必须引用外部函数中变量。

-- 外部函数返回值必须是内嵌函数。

1. 语法

-- 定义：

def 外部函数名(参数):

外部变量

def 内部函数名(参数):

使用外部变量

return 内部函数名

-- 调用：

变量 = 外部函数名(参数)

变量(参数)

1. 定义：是由函数及其相关的引用环境组合而成的实体。
2. 优点：内部函数可以使用外部变量。
3. 缺点：外部变量一直存在于内存中，不会在调用结束后释放，占用内存。
4. 作用：实现python装饰器。

"""

函数作用域

- 外部嵌套作用域 语法

# 内部函数可以访问外部函数变量

# 如果修改外部函数变量,必须通过nonlocal声明

作用：有内部、外部函数才能形成闭包

"""

语法一# 外部函数

def func01():

# 局部变量：对文件而言

# 外部嵌套变量：对func02而言

a = 10 #a是相对的，即可是局部，也可是外部的

# 内部函数

def func02():

# 内部函数可以访问外部函数变量

print(a)

func02() #外部函数执行时，func02才执行调用内部函数

# 调用外部函数(没有func02()的情况下，内部函数不执行)

func01()

语法二# 外部函数

def func03():

a = 10

# 内部函数

def func04():

# 如果修改外部函数变量,必须通过nonlocal声明

nonlocal a

a = 20

func04()

print(a)# ?

func03()

"""

闭包 - 语法（基于函数作用域语法）

字面意思：闭：封闭 包：内存空间

封闭(保存外部函数) 内存空间(栈帧)

总结：保存外部函数栈帧供内部函数访问

闭包目的：内部函数,可以在外部函数执行后,访问其变量a

价值（适用范围）：逻辑连续并作为装饰器基础

"""

def func01():

a = 10

def func02(): 正常情况下，外部函数执行后栈帧释放，不可以访问外部变量a

print(a) # 闭包技术下：可以访问外部变量(因为外部函数栈帧没释放)

return func02注意：return的是函数名称，不执行，不能加括号()

# 调用外部函数(不执行内部函数)

result = func01()

# 调用内部函数(相当于return func02变成func02()的效果)

手机屏幕截图

描述已自动生成result()# 10

闭包内存图，红框相当于封闭内存空间，不释放

"""

闭包 - 应用

逻辑连续

抽象：外部函数调用一次,内部函数调用多次,

内部函数都可以访问外部函数变量（反复访问）

具体：从一次得钱,到多次花钱的过程,可以连续不中断

注意：Python中通过面向对象解决逻辑连续的问题，有语言不支持面向对象，则需要闭包

"""

def give\_gife\_money(money): # 得钱

print("得到了%d元压岁钱" % money)

def child\_buy(commodity, price): # 花钱

nonlocal money

money -= price

print("购买%s,花%d元,还剩%d元" % (commodity, price, money))

return child\_buy

action = give\_gife\_money(1000) 调用外部函数，返回内部函数

action("变形金刚",200) 调用内部函数1次

action("遥控飞机",500) 调用内部函数2次

action("糖",100) 调用内部函数3次

### 函数装饰器decorator

1. 定义：在不改变原函数的调用以及内部代码情况下，为其添加新功能的函数。
2. 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内嵌函数

原函数 = 内嵌函数

@ 函数装饰器名称

def 原函数名称(参数):

函数体

原函数(参数)

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

原函数名称 = 函数装饰器名称（原函数名称）

1. 装饰器链：

一个函数可以被多个装饰器修饰，执行顺序为从近到远。

@print\_func\_name

@print\_func\_name

@print\_func\_name

"""

装饰器 - 应用

装饰器增加新功能为临时性的，不是每天都需要用

"""

# 需求：不改变func01调用,以及内部的情况下

# 为其（func01）增加新功能(打印函数名称)

初始代码：

初始1：def func01():

初始2： print("func01执行了")

初始3 func01() 调用初始2

函数装饰器代码：

新增1：def print\_func\_name(func): # 得到旧功能 外部函数

def wrapper(\*args, \*\*kwargs): # 新功能 + 旧功能 内部函数

# 新功能：打印传入的函数名称

print("-----", func.\_\_name\_\_, "-----")

# 旧功能：执行传入的函数

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

return wrapper

新增2：@print\_func\_name # 调用外部函数 绑定下面函数（创建属性调用下面方法）

初始1：def func01()

初始2： print("func01执行了")

# 调用外部函数(得到旧功能) 原理代码，实际开发使用@print\_func\_name

被新增2替代（不用）：# func01 = print\_func\_name(func01) 返回内部函数

# 调用内部函数(花钱)

初始3：func01() 不调初始2,调的print\_func\_name(func01)的返回值，即wrapper内部函数

func01()

"""

装饰器 - 细节语法

1. 内部函数返回值是旧功能返回值

"""

def print\_func\_name(func):

# \*合:将多个位置实参合并为一个元组

# \*\*合:将多个关键字实参合并为一个字典

def wrapper(\*args, \*\*kwargs): # 2 形参（def）

print("-----", func.\_\_name\_\_, "-----")

# 调用的是func01

# \*拆：将一个序列拆分为多个元素

# \*\*拆：将一个字典拆分为多个键值对

return func(\*args, \*\*kwargs) # 3 5 实参（调用函数）

return wrapper

@print\_func\_name

def func01(p1, p2): # 4 形参（def）

print("func01执行,参数是:", p1, p2)

return 100

# 调用内部函数wrapper，不是func01

re = func01(1, p2 = 2) # 1 6 实参（调用函数）

print(re) # 100