# 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。

### 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

#### 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

#### 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。

### 模块变量

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

\_\_doc\_\_变量：文档字符串。(连注释都可以打印出来) print(\_\_doc\_\_)

\_\_file\_\_变量：模块对应的文件路径名。

\_\_name\_\_变量：模块自身名字，可以判断是否为主模块。

当此模块作为主模块(第一个运行的模块)运行时，\_\_name\_\_绑定'\_\_main\_\_'，不是主模块，而是被其它模块导入时,存储模块名。（当在当前文件print(\_\_name\_\_)打印出来的就是\_\_main\_\_，但是在别的文件到我本文件，再用本文件去点\_\_name\_\_的话得到的是文件名，而不是\_\_main\_\_）

"""

    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

"""

print(.\_\_doc\_\_) # 打印本文件的顶端的三缸注释里面的内容（只打印最顶端的三缸注释）

# 模块变量

import demo01

# -- 获取文档字符串

print(demo01.\_\_doc\_\_) # 打印demo01.py文件里面顶端的三缸注释里面的内容

# -- 获取文件完整路径

# /home/tarena/2005/day15/demo03.py

print(demo01.\_\_file\_\_)

# -- 获取模块名称

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

print(\_\_name\_\_)  # \_\_main\_\_  # 只有当在本文件内打印才是"\_\_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。

### 分类

1. 内置模块(builtins)，在解析器的内部可以直接使用。例如：print()就在内置模块里
2. 标准库模块，安装Python时已安装且可直接使用。(需要import导入就可以直接使用)
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）

## 包package

### 定义

将模块以文件夹的形式进行分组管理。(包其实就是文件夹，只不过文件夹里有一个\_\_init\_\_.py文件，当模块太多的时候就会放到这个文件夹里)

创建包：pycharm项目文件夹下右键选择new-->python package就创建了一个包，里面自动生成\_\_init\_\_.py文件

### 作用

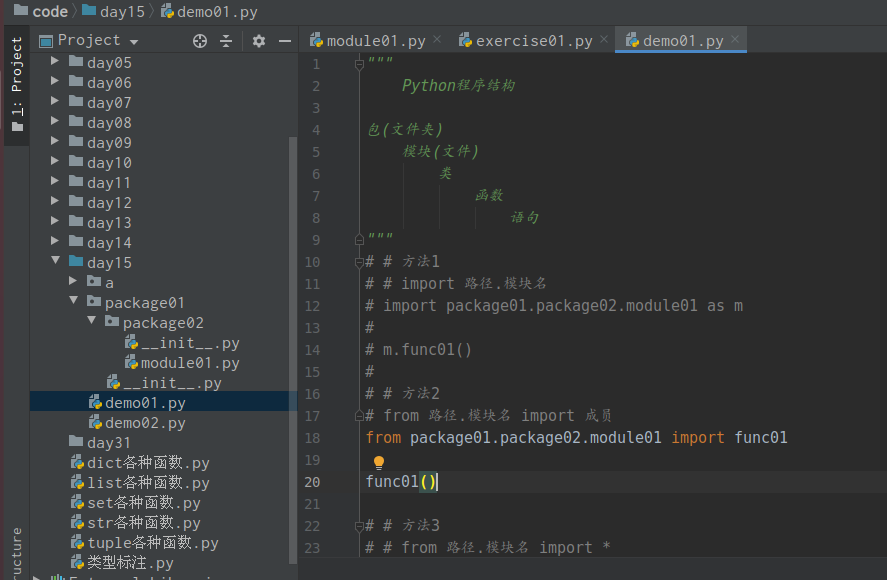
让一些相关的模块组织在一起，使逻辑结构更加清晰。

### 导入

快捷导入键：选中使用的方法按alt+回车，选中import this name

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]



"""

    Python程序结构

包(文件夹)

    模块(文件)

        类

            函数

                语句

"""

# # 方法1

# # import 路径.模块名

# import package01.package02.module01 as m

#

# 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()

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 \*

"""

    特殊导入方式

       设置包的\_\_init\_\_.py文件

"""

# 方式1：import 包

# 设置：import package01.package02.module01

# import package01.package02 as p

#

# p.module01.func01()

# 方式2：from 包 import 包

# 设置：from package01.package02 import module01

# from package01 import package02

#

# package02.module01.func01()

# 方式3：from 包 import \*

# 设置：\_\_all\_\_ = ["module01"]

from package01.package02 import \*

module01.func01() # 上面导入的是import \* 本来是不可以直接使用module01的，但是在module01.py旁边的文件\_\_init\_\_.py文件里面写了\_\_all\_\_ = ["module01"]所以才可以直接使用module01

### \_\_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\_\_ ：上面代码有介绍

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。

### 搜索顺序

1、内置模块

2、sys.path 提供的路径

# 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。（在except 后面用Exception接(或者直接不写Exception)的话，所有异常都能接住）

## 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: # 如果上面的except语句都没有执行，则执行else的语句

未发生异常的语句

finally:

无论是否发生异常都执行 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（抛异常，可以在前面调用的地方用try接异常）

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)

            raise Exception("年龄超过范围",1,2,3)

try:

    w01 = Wife(300)  # 1

    print(w01.age)

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

    print(e.args) # ('年龄超过范围', 1, 2, 3)

    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。

"""

    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.

        迭代器iterator:实施迭代过程的对象

    练习:exercise01

"""

message = "我是齐天大圣孙悟空"

# for item in message:

#     print(item)

# for 循环原理

# 1. 获取迭代器对象

iterator = message.\_\_iter\_\_()

while True:

    try:

        # 2. 获取下一个元素

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

        print(item)

        # 3. 如果没有元素,则停止循环.

    except StopIteration:

        break

# 面试题:可以参与for循环的条件是?

# 能够获取迭代器对象(可迭代对象)

# 具有\_\_iter\_\_函数

"""

练习1:使用迭代思想，打印元组中所有元素。

练习2:不使用for循环，打印字典中所有记录(键和值)

"""

tuple01 = (43, 4, 5, 67, 87, 89)

iterator = tuple01.\_\_iter\_\_()

while True:

    try:

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

        print(item)

    except StopIteration:

        break

dict01 = {"a": "A", "b": "B"}

iterator = dict01.\_\_iter\_\_()

while True:

    try:

        key = iterator.\_\_next\_\_()

        value = dict01[key]

        print(key, value)

    except StopIteration:

        break

"""

    函数返回结果:

        return 数据  -- 单个

        yield 数据   -- 多个

"""

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, 9003, "王荆轲", 16000),

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

    EmployeeModel(1004, 9003, "冯舜禹", 17000),

    EmployeeModel(1005, 9003, "曹海欧", 15000),

    EmployeeModel(1006, 9003, "魏鑫珑", 12000),

]

# 1. 定义函数,在list\_employee中查找薪资大于等于15000的所有员工

def find\_employees\_gt\_15k():

    for emp in list\_employee:

        if emp.money >= 15000:

            yield emp

for item in find\_employees\_gt\_15k():

    print(item.\_\_dict\_\_)

# 2. 定义函数,在list\_employee中查找员工编号为1005的员工

def find\_employee\_at\_eid():

    for emp in list\_employee:

        if emp.eid == 1005:

            return emp

employee = find\_employee\_at\_eid()

print(employee.\_\_dict\_\_)

# 3. 定义函数,在list\_employee中查找所有员工的姓名

def select\_names\_by\_employee():

    for emp in list\_employee:

        yield emp.name

for name in select\_names\_by\_employee():

    print(name)

# 4. 定义函数,在list\_employee中查找薪资最高的员工

def get\_max\_employee\_by\_money():

    max\_emp = list\_employee[0]

    for i in range(1,len(list\_employee)):

        if max\_emp.money  < list\_employee[i].money:

            max\_emp = list\_employee[i]

    return max\_emp

emp = get\_max\_employee\_by\_money()

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

## 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。

1. 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

    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):

        self.data = data

        self.index = -1

    def \_\_next\_\_(self):

        self.index += 1

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

            return self.data[self.index]

        else:

            raise StopIteration()

class SkillManager:

    def \_\_init\_\_(self):

        self.\_\_skills = []

    def add\_skill(self, skill):

        self.\_\_skills.append(skill)

    def \_\_iter\_\_(self):

        return SkillIterator(self.\_\_skills)

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:

#     print(skill)

iterator = manager.\_\_iter\_\_()

while True:

    try:

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

        print(item)  # 降龙十八掌

    except StopIteration:

        break

"""

    迭代员工管理器

"""

class EmployeeIterator:

    def \_\_init\_\_(self, data):

        self.data = data

        self.index = -1

    def \_\_next\_\_(self):

        self.index += 1

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

            return self.data[self.index]

        else:

            raise StopIteration()

class EmployeeManager:

    def \_\_init\_\_(self):

        self.all\_employee = []

    def add\_employee(self, emp):

        self.all\_employee.append(emp)

    def \_\_iter\_\_(self):

        return EmployeeIterator(self.all\_employee)

manager = EmployeeManager()

manager.add\_employee("老王")

manager.add\_employee("老李")

manager.add\_employee("老孙")

for item in manager:

    print(item)

"""

    自定MyRange类

    实现range(5)效果

"""

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)

for item in range(9999999999999999999999999999999):

    print(item)

"""

    迭代器 --> yield

    练习:exercise05,06

"""

class SkillManager:

    def \_\_init\_\_(self):

        self.\_\_skills = []

    def add\_skill(self, skill):

        self.\_\_skills.append(skill)

    # def \_\_iter\_\_(self):

    #     print("准备")

    #     yield self.\_\_skills[0]

    #

    #     print("准备")

    #     yield self.\_\_skills[1]

    #

    #     print("准备")

    #     yield self.\_\_skills[2]

    def \_\_iter\_\_(self):

        for skill in self.\_\_skills:

            print("准备")

            yield skill

        # 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:

    print(skill)

# iterator = manager.\_\_iter\_\_()

# while True:

#     try:

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

#         print(item)  # 降龙十八掌

#     except StopIteration:

#         break

# 现象:调用\_\_iter\_\_方法,但是不执行

#     调用\_\_next\_\_方法,执行\_\_iter\_\_方法

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

# 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

以后函数返回两个以上的结果的时候都用yield，只返回一个的时候才用return

返回的生成器只能被for一次，for第二次的时候拿不出来东西

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

    MyRange3.0

    类 --> 函数

    练习:exercise08/09

"""

"""

class Generator:# 生成器 = 可迭代对象 + 迭代器

    def \_\_iter\_\_(self): # 可迭代对象

        return self

    def \_\_next\_\_(self): # 迭代器

        ....

"""

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

# 循环一次 计算一次 返回一次

# for item in my\_range(5):的原理就是下面几行代码

range = my\_range(5)

iterator = range.\_\_iter\_\_()

while True:

    try:

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

        print(item)

    except StopIteration:

        break

"""

    返回列表中所有数字的个位

    方式1:传统思想

        定义函数,创建列表,循环计算每个元素的个位,存入列表

    方式2:生成器思想

        定义函数,循环计算每个元素的个位,通过yield返回

    体会:惰性操作/延迟操作/推算数据

"""

def find\_number\_unit(list\_target):

    list\_result = []

    for number in list\_target:

        unit = number % 10

        list\_result.append(unit)

    return list\_result

# 测试

list01 = [54,55,36,67,28,69,90]

result = find\_number\_unit(list01)

for item in result:

    print(item)

def find\_number\_unit(list\_target):

    for number in list\_target:

        unit = number % 10

        yield unit

# 测试

list01 = [54,55,36,67,28,69,90]

# 调用不执行,延迟/惰性

result = find\_number\_unit(list01)

for item in result:

    print(item)

## 内置生成器

### 枚举函数enumerate（遍历容器的时候可以获得每个元素的索引）

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。如果放进去的参数是一个字典，则返回的是字典的“索引”和键

"""

    内置生成器enumerate

"""

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

# (索引, 元素)

# for item in enumerate(list01):

for i, element in enumerate(list01):

    if element % 2:

        list01[i] = 0

print(list01) # [54, 0, 6, 76, 8, 0]

### 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

"""

list01 = ["a", "b", "c"]

list02 = ["A", "B", "C"]

for item in zip(list01, list02):

    print(item) # ('a', 'A') ('a', 'A') ('c', 'C')

## 生成器表达式

生成器表达式和列表推导式的外观几乎一模一样，列表推导式用的是中括号，生成器表达式用的是小括号，但是运行方法差别很大，生成器表达式是惰性操作，而且结果还得用for来获取。

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] )

"""

    生成器函数

        给其他人用

    生成器表达式

        给自己使用

"""

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

# 查找所有字符串

# list\_result = []

# for item in list01:

#     if type(item) == str:

#         list\_result.append(item)

# print(list\_result)

# 列表推导式

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

for item in list\_result: # 可以直接打印list\_result

    print(item)

# 生成器表达式

list\_result = (item for item in list01 if type(item) == str)

for item in list\_result: # 必须用for来获取 ，不能直接打印list\_result

    print(item)

# 函数式编程

最终目标是把lambda函数作为参数传进另一个函数

1. 定义：用一系列函数解决问题。

-- 函数可以赋值给变量，赋值后变量绑定函数。

-- 允许将函数作为参数传入另一个函数。

-- 允许函数返回一个函数。

1. 高阶函数：将函数作为参数或返回值的函数。

"""

    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")

# 将函数赋值给变量

a = func01

# 通过变量调用函数

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?

func03(func01)

func03(func02)

## 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

用def定义的叫有名称函数，用lambda定义的叫匿名函数

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。

"""

    函数式编程 - 思想

        面向对象

            封装：分

            继承: 隔

        函数式编程

            分：将变化点分为多个函数

            隔: 使用参数隔离具体变化的函数

    练习:exercise05

"""

list01 = [42,45,5,66,7,89]

# 需求：定义函数,获取所有大于10的数字

def find01():

    for item in list01:

        if item > 10:

            yield item

# 需求：定义函数,获取所有偶数

def find02():

    for item in list01:

        if item % 2 == 0:

            yield item

for number in find01():

    print(number)

for number in find02():

    print(number)

# －－－－－－－－－－－－－－－－－－

# 提取变化函数

def condition01(item):

    return item > 10

def condition02(item):

    return item % 2 == 0

# 提取通用函数 -- 万能查找

def find(func):#　创建了钩子

    for item in list01:

        # if item % 2 == 0:

        # if condition02(item):

        if func(item):# 拉起钩子(执行条件)

            yield item

for number in find(condition02):# 向钩子上挂条件

    print(number)

from common.iterable\_tools import IterableHelper

for number in IterableHelper.find\_all(list01, condition01):

    print(number)

"""

    lambda 表达式

        匿名函数:

            lambda 参数: 函数体

            def不能转换为lambda写法

                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

"""

# 写法1:有参数,有返回值

# def func01(p1, p2):

#     return p1 > p2

#

# print(func01(10, 5))

func01 = lambda p1, p2: p1 > p2

print(func01(10, 5))

# 写法2:有参数,无返回值

# def func02(p1):

#     print("参数是:",p1)

#

# func02(10)

func02 = lambda p1: print("参数是:", p1)

func02(10)

# 写法3:无参数,有返回值

# def func03():

#     return "结果"

#

# result = func03()

# print(result)

func03 = lambda: "结果"

result = func03()

print(result)

# 写法4:无参数,无返回值

# def func04():

#     print("func04执行喽")

#

# func04()

func04 = lambda: print("func04执行喽")

func04()

# def不能转换为lambda写法

def func05(p1):

    p1[0] = 100

list01 = [10]

func05(list01)

print(list01[0]) # 100

# 1. lambda表达式不能赋值

# func05 = lambda p1:p1[0] = 100

def func06():

    for i in range(5):

        print(i)

func06()

# 2. lambda 函数体只能有一条语句

# func06 = lambda :for i in range(5):

#                     print(i)

终极版本（以后开发可以一步步推过来）

"""

    函数式编程 - 思想

        面向对象

            封装：分

            继承：隔

            多态：做

        函数式编程

            分：将通用和变化点分为多个函数

            隔：使用参数隔离具体变化的函数

            做：通常使用lambda定义变化函数

    练习:

"""

list01 = [42, 45, 5, 66, 7, 89]

# 提取变化函数

# def condition01(item):

#     return item > 10

#

# def condition02(item):

#     return item % 2 == 0

# 提取通用函数 -- 万能查找

def find(func):  # 创建了钩子

    for item in list01:

        if func(item):  # 拉起钩子(执行条件)

            yield item

# for number in find(condition02):# 向钩子上挂条件

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

    print(number)

from common.iterable\_tools import IterableHelper

# for number in IterableHelper.find(list01, condition01):

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

    print(number)

*"""  
 练习1需求:  
 在员工列表中,查找删除薪资15000以内的所有员工  
 在员工列表中,查找删除部门编号是9005的所有员工  
  
 练习2需求:  
 在员工列表中,查找薪资最高的员工  
 在员工列表中,查找员工编号最大的员工  
  
 练习3需求:  
 在员工列表中,根据薪资升序排列  
 在员工列表中,根据员工编号升序排列  
  
 步骤  
 $1. 定义函数,完成需求.  
 x2. 将变化点定义为函数  
 将通用代码定义为函数  
 x3. 通用函数使用参数隔离变化点  
 $4. 在IterableHelper类中创建通用函数  
 $5. 在当前模块中调用通用函数(lambda)  
"""***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),  
]  
  
"""  
def delete01():  
 count = 0  
 for i in range(len(list\_employee)-1,-1,-1):  
 if list\_employee[i].money < 15000:  
 del list\_employee[i]  
 count += 1  
 return count  
  
def delete02():  
 count = 0  
 for i in range(len(list\_employee)-1,-1,-1):  
 if list\_employee[i].did == 9005:  
 del list\_employee[i]  
 count += 1  
 return count  
"""  
  
# 参数是列表中的每个元素  
# count = IterableHelper.delete\_all(list\_employee,lambda emp:emp.money < 15000)  
# print(count)  
  
count = IterableHelper.delete\_all(list\_employee, **lambda** item: item.did == 9005)  
print(count)  
  
# def get\_max01():  
# max\_value = list\_employee[0]  
# for i in range(1, len(list\_employee)):  
# if max\_value.money < list\_employee[i].money:  
# max\_value = list\_employee[i]  
# return max\_value  
#  
# def get\_max02():  
# max\_value = list\_employee[0]  
# for i in range(1, len(list\_employee)):  
# if max\_value.eid < list\_employee[i].eid:  
# max\_value = list\_employee[i]  
# return max\_value  
  
max = IterableHelper.get\_max(list\_employee, **lambda** item: item.money)  
print(max.\_\_dict\_\_)  
max = IterableHelper.get\_max(list\_employee, **lambda** item: item.eid)  
print(max.\_\_dict\_\_)  
# 在列表中根据薪资升序排列  
IterableHelper.ascending\_order(list\_employee, **lambda** emp: emp.money)  
IterableHelper.ascending\_order(list\_employee, **lambda** emp: emp.eid)  
print(list\_employee)

### 内置高阶函数（和自己做的通用方法（common文件夹）差不多）

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。

**Sorted函数会返回排序后新的容器，我们自己做的方法操作的是原来容器，不会生成新的容器，默认是升序，把reverse=True后就变成倒序了**

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。

*"""  
 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  
# map(func,iterable) -> 生成器  
**for** item **in** map(**lambda** item: item.name, list\_employee):  
 print(item)  
# 此为自己的方法select，和上面方法类似，在IterableHelper文件中，需要导入  
# for item in IterableHelper.select(list\_employee,lambda item:item.name):  
# print(item)  
  
# 2.filter:过滤  
# filter(func,iterable) -> 生成器  
**for** item **in** filter(**lambda** emp: emp.money > 15000, list\_employee):  
 print(item.\_\_dict\_\_)  
# 此为自己的方法find\_all，和上面方法类似，在IterableHelper文件中，需要导入  
# for item in IterableHelper.find\_all(list\_employee,lambda emp:emp.money > 15000):  
# print(item.\_\_dict\_\_)  
  
# 3.max 4.min : 最大最小  
# 极值 = max(容器, key=函数)  
max\_emp = max(list\_employee, key=**lambda** item: item.money)  
print(max\_emp.\_\_dict\_\_)  
# 此为自己的方法get\_max，和上面方法类似，在IterableHelper文件中，需要导入  
# max\_emp =IterableHelper.get\_max(list\_employee,lambda item:item.money)  
# print(max\_emp.\_\_dict\_\_)  
  
# 5. 排序  
# -- 升序  
# 　排序结果 = sorted(容器, key=函数)  
result = sorted(list\_employee, key=**lambda** e: e.money)  
print(result)  
# 此为自己的方法ascending\_order，和上面方法类似，在IterableHelper文件中，需要导入，而且不会生成新的容器占用内存，比sorted方法好  
# IterableHelper.ascending\_order(list\_employee,lambda e:e.money)  
# print(list\_employee)  
  
# -- 降序  
# 　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)

*"""  
使用内置高阶函数实现：  
 1. ([1,1],[2,2,2],[3,3,3],[4,4,4,4,4])  
 获取元组中长度最大的列表  
 2. 在员工列表列表，获取所有员工编号与姓名  
 3. 在员工列表中，获取所有工资大于13000的员工  
 4. 对员工列表，根据员工编号进行降序排列  
 5. 获取所有工资大于15000的员工姓名.  
"""***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.  
tuple01 = ([1, 1], [2, 2, 2], [3, 3, 3], [4, 4, 4, 4, 4])  
# lambda 参数：函数体  
# 参数得到的是容器中元素  
# 函数体是对元素的处理  
max\_value = max(tuple01, key=**lambda** item: len(item))  
print(max\_value)  
  
# 2.  
tuple02 = list(map(**lambda** emp: (emp.eid, emp.name), list\_employee))  
print(tuple02)  
  
# 3.  
**for** item **in** filter(**lambda** element: element.money > 13000, list\_employee):  
 print(item.\_\_dict\_\_)  
  
# 4.  
result = sorted(list\_employee, key=**lambda** item: item.eid, reverse=**True**)  
print(result)  
  
# 5.   
result = list(map(**lambda** e: e.name, filter(**lambda** element: element.money > 15000, list\_employee)))  
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 # 注意返回函数不写括号

-- 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。 (连续几次调用内部函数来调用外部变量，使用的都是同一个外部变量)
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。

*"""  
 闭包 - 语法  
 字面意思：封闭(保存外部函数)内存空间(栈帧)  
 目的：内部函数,可以在外部函数执行后,访问其变量  
 价值：  
"""***def func01**():  
 a = 10  
  
 **def func02**():  
 print(a) # 可以访问外部变量(因为外部函数栈帧没释放)  
  
 **return** func02  
  
# 调用外部函数(不执行内部函数)  
result = func01()  
# 调用内部函数  
result()# 10

*"""  
 闭包 - 应用  
 逻辑连续  
 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  
  
 从一次得钱,到多次花钱的过程,可以连续不中断  
"""***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("购买了%s,花了%d元,还剩下%d元" % (commodity, price, money))  
 **return** child\_buy  
  
action = give\_gife\_money(1000) # 得到了1000元压岁钱  
action("变形金刚",200) # 购买了变形金刚,花了200元,还剩下800元  
action("遥控飞机",500) # 购买了遥控飞机,花了500元,还剩下300元  
action("糖",100) # 购买了糖,花了100元,还剩下200元

### 函数装饰器decorator

1. 定义：在不改变原函数的调用以及内部代码情况下，为其添加新功能的函数。
2. 语法（记住这个固定语法套路就行）

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 # 会把下一行的原函数当成参数传进函数装饰器里

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。(拦截功能：调用旧函数会自动拦截，而去运行装饰器函数里的内嵌函数)

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 - 应用  
"""*# 需求：不改变func01调用,以及内部的情况下  
# 为其增加新功能(打印函数名称)  
  
**def print\_func\_name**(func): # 得到旧功能  
 **def wrapper**(\*args, \*\*kwargs): # 新功能 + 旧功能  
 # 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  
  
@print\_func\_name # 调用外部函数 绑定下面函数作为参数传进去   
**def func01**():  
 print("func01执行了")  
  
# 调用外部函数(得到了1000元)  
# func01 = print\_func\_name(func01) #上面的 @print\_func\_name那行，由这个版本升级  
  
# 调用内部函数(花钱)  
func01()  
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  
 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  
 **return** wrapper  
  
@print\_func\_name  
**def func01**(p1, p2): # 4  
 print("func01执行,参数是:", p1, p2)  
 **return** 100  
  
# 调用内部函数wrapper  
re = func01(1, p2 = 2) # 1 6  
print(re) # 100

*"""  
 练习:使用装饰器，为旧功能增加新功能(验证权限).  
"""*# 验证权限  
**def verif\_permissions**(func):  
 **def wrapper**(\*args, \*\*kwargs):  
 print("验证权限")  
 **return** func(\*args, \*\*kwargs)  
 **return** wrapper  
  
@verif\_permissions  
**def insert**():  
 print("插入成功")  
  
@verif\_permissions  
**def delete**():  
 print("删除成功")  
  
insert()  
delete()

*"""  
 练习:使用装饰器，为旧功能增加新功能(验证权限).  
"""*# 验证权限  
**def verif\_permissions**(func):  
 **def wrapper**(\*args, \*\*kwargs):  
 print("验证权限")  
 **return** func(\*args, \*\*kwargs)  
 **return** wrapper  
  
@verif\_permissions  
**def insert**(data):  
 print("插入成功", data)  
 **return** "ok"  
  
@verif\_permissions  
**def delete**(id):  
 print("删除成功", id)  
 **return** "yes"  
  
result = insert("10条数据")  
print(result)  
result = delete(id=1001)  
print(result)