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
3. 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”
4. 执行过程：
   (1) 调用生成器函数会自动创建迭代器对象。
   (2) 调用迭代器对象的__next__()方法时才执行生成器函数。
   (3) 每次执行到 yield 语句时返回数据，暂时离开。
   (4) 待下次调用__next__()方法时继续从离开处继续执行。
5. 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。

生成器
特点:惰性操作
本质:可迭代对象 + 迭代器
面试题: 请简述生成器与迭代器的关系?
生成器函数会通过 yield 将代码编译为迭代器的 next 方法.
就可以一次循环,一次计算,一次返回,从而实现惰性操作.
便可节省内存空间.

生成器函数:
class 生成器:

```
        def __iter__():
            return self

        def __next__():
            return 数据

    def 方法名称():
        ...
        yield 数据
        ...

    for item in 方法名():
        item

    延迟/惰性 --> 立即
        容器(方法名())

  生成器表达式:
    (对象 item 操作 for item in 可迭代对象)
    (对象 item 操作 for item in 可迭代对象 if 条件)

函数式编程
  将函数作为参数
    参数是数据,传递数值,字符串,容器,自定义类
    参数是逻辑(行为/算法/功能),传递函数.

    待重构代码:
        def 功能 1():
            不变的代码
            变化的代码 1

        def 功能 2():
            不变的代码
            变化的代码 2

        def 功能 3():
            不变的代码
            变化的代码 3

    "封装":分
        def 条件 1():
            变化的代码 1
        def 条件 2():
            变化的代码 2
        def 条件 3():
            变化的代码 3

        def 稳定的/通用的():
```

不变的代码

"继承":隔
def 稳定的/通用的(参数):
不变的代码
参数()
"多态":执
稳定的/通用的(条件 3)

list_helper.py
def 稳定的/通用的():
....

将函数作为返回值
....

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
2. 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。

练习:自定义函数 my_enumerate,实现以下功能

```
list01 = [4, 5, 5, 56, 6, 77]
# for item in enumerate(list01):
#     print(item)# (索引, 元素)
def my_enumerate(list_target):
    index = 0
    for item in list_target:
        yield (index,item)
        index +=1
for item in my_enumerate(list01):
    print(item,end=" ")# (索引, 元素)(0, 4) (1, 5) (2, 5) (3, 56) (4, 6) (5, 77)
print()
for index,item in my_enumerate(list01):
    print(index,item,end="|")#  索引 元素  0 4 | 1 5 | 2 5 | 3 56 | 4 6 | 5 77
```

3.

zip

1. 语法：
   for item in zip(可迭代对象 1, 可迭代对象 2….):
   语句
2. 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。
3. 语法
   -- 定义：
   变量 = lambda 形参: 方法体
   -- 调用：
   变量(实参)
4. 说明：
   -- 形参没有可以不填
   -- 方法体只能有一条语句，且不支持赋值语句。

```
"""
    lambda ： 匿名函数
```

```
   语法规则:lambda 参数:函数体
"""
# 定义有参数lambda
func = lambda item:item % 2 == 0
re = func(5)
print(re)
# 定义无参数lambda
func = lambda :100
re = func()
print(re)
# 定义多个参数lambda
func =lambda a,b,c:a+b+c
re = func(1,2,3)
print(re)
# 定义无返回值lambda
func = lambda a:print("变量是:",a)
func(10)
class A:
   def __init__(self,a):
      self.a = a
def fun01(obj):
   obj.a = 100
o = A(10)
fun01(o)
print(o.a)
# SyntaxError: can't assign to lambda
# lambda 不支持赋值语句
# func = lambda obj:obj.a = 100
# lambda 只支持一条语句
def fun01(a,b):
   if a % 2 == 0:
      print(a+b)
# lambda a,b:if a % 2 == 0: print(a+b)
list01 = [4,5,5,6,767,8,10]
# def condition01(item):
#     return item % 2 == 0
#
# def condition02(item):
#     return item % 2
#
# def condition03(item):
```

```
#     return item  > 10
def find(target,func):
    for item in target:
        if func(item):
            yield item
# for item in find(list01,condition03):
#     print(item)
for item in find(list01,lambda item:item  > 10):
    print(item)
```

内置高阶函数

1. map（函数，可迭代对象）：使用可迭代对象中的每个元素调用函数，将返回值作为新可迭代对象元素；返回值为新可迭代对象。
2. filter(函数，可迭代对象)：根据条件筛选可迭代对象中的元素，返回值为新可迭代对象。
3. sorted(可迭代对象，key = 函数,reverse = bool 值)：排序，返回值为排序结果。
4. max(可迭代对象，key = 函数)：根据函数获取可迭代对象的最大值。
5. min(可迭代对象，key = 函数)：根据函数获取可迭代对象的最小值。

"""

```
    内置高阶函数
    17:05
"""
from common.list_helper import ListHelper
class Wife:
    def __init__(self, name, age, weight, height):
        self.name = name
        self.age = age
        self.weight = weight
        self.height = height
list01 = [
    Wife("翠花", 36, 60, 1.5),
    Wife("如花", 39, 75, 1.3),
    Wife("赵敏", 25, 46, 1.7),
    Wife("灭绝", 42, 50, 1.8)
]
# for item in ListHelper.find_all(list01,lambda
item:item.age < 40):
#     print(item.name)
# 1. 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。
```

```
for item in filter(lambda item:item.age < 40,list01):
    print(item.name)
# for item in ListHelper.select(list01,lambda item:
(item.name,item.age)):
#     print(item)
# 2. 映射: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；
for item in map(lambda item:
(item.name,item.age),list01):
    print(item)
# 3. 排序(返回排序结果,支持升序与降序)
# ListHelper.order_by(list01,lambda item:item.height)
# for item in list01:
#     print(item.height)
# 升序排列
for item in sorted(list01,key = lambda
item:item.height):
    print(item.height)
# 降序排列
for item in sorted(list01,key = lambda
item:item.height,reverse=True):
    print(item.height)
# 4. 获取最大值
re = max(list01, key=lambda item: item.weight)
print(re.name)
# 5. 获取最小值
re = min(list01, key=lambda item: item.weight)
print(re.name)
```

函数作为返回值

逻辑连续，当内部函数被调用时，不脱离当前的逻辑。

闭包

1. 三要素:
   -- 必须有一个内嵌函数。
   -- 内嵌函数必须引用外部函数中变量。
   -- 外部函数返回值必须是内嵌函数。
2. 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)

3. 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。
4. 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。
5. 优点：内部函数可以使用外部变量。
6. 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。
7. 作用：实现 python 装饰器。

"""

```
    Enclosing  外部嵌套作用域
"""
# 外部
def fun01():
    # 对于fun01而言,a是局部变量
    # 对于fun02而言,a是外部嵌套变量
    a = 10
    # 内部
    def fun02():
        b = 20
        # 可以访问外部变量
        # print(a)
        # 并没有修改外部变量,而是创建了局部变量.
        # a = 100
        # 声明外部变量
        nonlocal a
        a = 100
    fun02()
    print(a)
fun01()
```

"""

```
    闭包:外部函数执行过后,不立即释放内存,
        而是等着内部函数执行完毕后,再统一释放.
"""
def fun01():
    a = 10
    def fun02():
```

```
        print(a)
    return fun02
# re 变量 指向的是 内部函数
re = fun01()# 调用外部函数          ---> 执行完毕后,不释放栈帧.
re()# 调用内部函数
```

"""

```
    闭包应用
"""
def give_gife_money(money):
    """
        获取压岁钱
    """
    def child_buy(target, price):
        """
            购买商品
        """
        nonlocal money
        if money > price:
            money -= price
            print("购买%s,花了%d钱" % (target, price))
        else:
            print("钱不够了")
    return child_buy
# 价值:逻辑连续(多次购买商品,不脱离获得的压岁钱)
aciton = give_gife_money(10000)
aciton("无人机", 8000)
aciton("外星人", 20000)
```

8.