**Day10笔记**

day10(上午)：

1.栈和队列：

栈： 满足特点 ---> 先进后出，类似于我们生活中的子弹夹

队列： 满足特点 ---> 先进先出，类似于我们生活中的排队买票、安检

【注意点】：

对于栈结构而言：python语言中没有为其封装特点的函数来获取对象，

我们可以使用list(列表)来模拟栈的特点

对于队列结构而言：python语言中有为其封装特定的获取方式，

调用**collections模块中的deque()函数**就可以获取一个队列对象

操作步骤：

步骤一：导入collections模块

步骤二：collections.deque() --> 得到**一个列表对象**

步骤三：使用队列对象调用存(append)和取(popleft)的函数，完成需求

**使用list来模拟栈结构存取数据的特点：先进后出** *#定义一个list对象，stack(变量名、引用名)*stack = []  
*#向栈中添加数据(模拟压栈、进栈现象)*stack.append(**'A'**)  
print(stack)  
stack.append(**'B'**)  
print(stack)  
stack.append(**'C'**)  
print(stack)  
*#将栈中的数据弹出(模拟出栈、弹栈现象)*obj = stack.pop()  
print(**'弹出：'** + obj)  
obj = stack.pop()  
print(**'弹出：'** + obj)  
obj = stack.pop()  
print(**'弹出：'** + obj)

*#得到结果 ['A']  
['A', 'B']  
['A', 'B', 'C']  
弹出：C  
弹出：B  
弹出：A*

**演示队列存取数据的特点：先进先出**

**import** collections*#获取队列对象：queue*queue = collections.deque()  
print(queue,type(queue))  
*#向队列中添加数据(模拟进队、入队现象)*queue.append(**'A'**)  
print(queue)  
queue.append(**'B'**)  
print(queue)  
queue.append(**'C'**)  
print(queue)  
*#将队列中的数据弹出(出队现象)*obj = queue.popleft()  
print(**'弹出：'** + obj)  
obj = queue.popleft()  
print(**'弹出：'** + obj)  
obj = queue.popleft()  
print(**'弹出：'** + obj)

*#得到结果 ['A']  
['A', 'B']  
['A', 'B', 'C']  
弹出：A  
弹出：B  
弹出：C*

扩展：使用深度和广度遍历来实现目录(单、多层)的遍历操作

作用：为了更好的理解栈和队列的存取数据特点：

我们在书写深度遍历和广度遍历的代码过程中，

得知了不使用递归操作也是可以实现遍历多层级的目录，

这样做的好处是：更加的节省内存资源  
**#模拟栈结构特点实现目录遍历之深度遍历****import** os**def** getAllFileST(path):  
 *#定义一个list对象* lt = []  
 *#将path字符串路径(绝对路径)进行压栈* lt.append(path)  
 *#根据lt的长度来决定循环是否终止* **while** len(lt) != 0:  
 *#将lt中的数据弹出* file\_path = lt.pop()  
 *#将file\_path中的内容(文件、子目录)以列表的形式返回* files\_list = os.listdir(file\_path)  
 *#循环处理files\_list中的每一个元素：* **for** file **in** files\_list:  
 *# 将file中的内容还原成为其绝对路径的形式* fileAbsPath = os.path.join(file\_path,file) **判断是文件还是目录：  
 如果是文件，直接打印名字即可  
 如果是目录，在打印完名字后将其压入栈中  
 if** os.path.isfile(fileAbsPath):  
 print(**'文件：'** + file)  
 **else**:  
 print(**'目录：'** + file)  
 lt.append(fileAbsPath)  
*#执行函数*  
path = **r'C:\Users\Administrator\Desktop\a'**getAllFileST(path)

**#模拟使用队列结构实现遍历目录之广度遍历  
import** collections,os*#自定义函数：实现遍历多层级目录的操作(广度遍历)***def** getAllFileQU(path):  
 *#获取一个队列对象* queue = collections.deque()  
 *#将path数据进队* queue.append(path)  
 *#只要队列中还有内容，那么循环继续* **while** len(queue) != 0:  
 *#获取队列中的元素* file\_path = queue.popleft()  
 *#获取file\_path对应的子内容(文件、子目录)以列表的形式返回* files\_list = os.listdir(file\_path)  
 *#循环处理files\_list中的每一个元素* **for** file **in** files\_list:  
 *#将file的内容还原成为其绝对路径值* fileAbsPath = os.path.join(file\_path,file)  
 *#判断是文件还是目录，之后操作和深度遍历类似* **if** os.path.isfile(fileAbsPath):  
 print(**'文件：'** + file)  
 **else**:  
 print(**'目录：'** + file)  
 queue.append(fileAbsPath)  
*#执行函数*  
path = **r'C:\Users\Administrator\Desktop\123'**getAllFileQU(path)

2.**匿名函数**(没有名字的函数)

匿名函数的定义格式：

lambda [变量参数]:表达式

何时定义匿名函数？

1).功能/操作的逻辑比较简单

2).函数被执行的次数少(1次、2次...)

【注意事项】：

1).匿名函数同样需要被调用才能执行 --> 调用仍然使用()传参的方式实现

2).匿名函数同样可以定义有返回值没有返回值，有参数没有参数的情况

**演示匿名函数的定义和使用：** *#定义无参有返回值的有名函数：***def** func():  
 **return True***#定义无参有返回值的匿名函数：*f = **lambda** : **False** *#调用有名函数执行*print(func()) #True  
*#调用匿名函数执行*print(f()) #False

*#定义无参无返回值的有名函数：***def** func():  
 print(**True**)  
*#定义无参无返回值的匿名函数：*f = **lambda** : print(**False**)  
*#调用有名函数执行*func() #True  
*#调用匿名函数执行*f() #False

**需求：将字符串数据'this is \na \ttest' -->得到 'this is a test'  
步骤一：使用字符串中的split()函数 -->空参的split的作用：可以去除字符串中的空格、\n、\t等内容...  
 然后将子串数据以列表的形式返回  
步骤二：使用字符串的join()函数 --> 以空格(' ')作用连接方式，将列表中的每一个元素拼接得到一个字符串返回** str1 = **'this is \na \ttest'**lt = str1.split()  
print(lt)  *#得到['this', 'is', 'a', 'test']*  
str2 = **' '**.join(lt)  
print(str2,type(str2)) *#得到this is a test <class 'str'>*  
  
print((**lambda** x: **' '**.join(x.split()))(str1))

*上面一行代码和下面两行的效果一样（有参数的匿名函数），下面更好理解*

*#f = lambda x:' '.join(x.split())  
#print(f(str1))*

**分类匿名函数 --> 以参数进行  
1).1个参数  
2).多个参数  
3).默认参数  
4).可变参数***#先定义有名函数***def** m1(a):  
 **return** a \*\* 2  
**def** m2(a,b):  
 **return** a + b  
**def** m3(s1,s2=**'hello'**):  
 **return** s1 + s2  
print(m1(3))  
print(m2(10,20))  
print(m3(**'老郭'**))  
*#定义匿名函数的形式以及调用*print((**lambda** x: x \*\* 2)(3)) *######相当于f=lambda x:x\*\*2*

*Print(f(3))*  
print((**lambda** x,y: x + y)(10,20))

*######f=lambda x,y:x+y  
print(f(10,20))*

print((**lambda** s1,s2=**'hello，'**: s2 + s1)(**'老郭'**))

*#匿名函数的可变参格式*print((**lambda** \*x: x)(**'UI'**,**'H5'**,**'JS'**,**'PHP'**,**'软件测试'**,**'网络营销'**))

*#得到结果('UI', 'H5', 'JS', 'PHP', '软件测试', '网络营销')*print((**lambda** \*\*y: y)(name=**'zs'**,age=18))  
*#得到{'name': 'zs', 'age': 18}*

*# 需求：使用匿名函数实现得到两个数中的较大值*my\_max = **lambda** x,y: x **if** x > y **else** y  
*#调用匿名函数执行*print(my\_max(10,20)) *#得到20*  
  
*#演示有名函数和匿名函数嵌套使用情况***def** speak(content):  
 **return** (**lambda** x: x + **'你好'**)(content)  
print(speak(**'老郭'**))

*演示匿名函数配合容器类型使用(了解即可)  
# 1.匿名函数配合列表对象使用*lt = [**lambda** x:x \*\* 2,**lambda** x:x \*\* 3,**lambda** x:x \*\* 4]  
*#遍历lt对象***for** i **in** lt:  
 print(i(2))  *#得到4*

*8*

*16  
#单独获取某一个元素并且执行*print(lt[-1](3)) *#得到81*

*#匿名函数配合字典对象使用*key = **'C'**dic1 = {  
 **'A'**:**lambda** :2\*2,  
 **'B'**:**lambda** :3\*3,  
 **'C'**:**lambda** :4\*4  
}  
print(dic1[key]()) *#得到16*

*#列表嵌套字典*dic2 = [  
 {**'name'**:**'张学友'**,**'age'**:56,**'height'**:173.0},  
 {**'name'**:**'刘德华'**,**'age'**:55,**'height'**:176.0},  
 {**'name'**:**'郭富城'**,**'age'**:54,**'height'**:170.0},  
 {**'name'**:**'黎明'**,**'age'**:57,**'height'**:181.0},  
]  
*#需求：按照身高进行排序显示(降序)*dic2.sort(reverse=**True**,key=**lambda** x:x[**'height'**])  
print(dic2)

3.**生成器**：

什么是生成器？

它内部封装了一套公式/算法，只有等到需要调用/执行数据的时候 --》执行next()函数

才会将公司计算得到的数据结果返回给程序，这就是生成器的原理(核心思想)

格式：两种

执行完毕都返回generator类型对象

格式一：

(表达式 for 形参列表 in 容器对象 if ...)

格式二：

定义函数，在函数的内部需要使用yield关键字来记录返回的generator对象的数据公式，

【注意】：此时函数就算有return，已经无视了，返回的一定是一个generator类型的对象（即生成器对象）

【注意事项】：

1).生成器中是没有真实数据存在的，所以我们是不能直接使用len()函数来尝试得到其长度；

如果我们这么做了，错误提示：no len()

2).生成器关联的真实数据智能被使用一次，不可逆；如果超量调用，会报错：StopIteration

比较：列表生成式和生成器？（**面试题**）

对于生成器而言：

定义执行后会得到一个generator对象，

此对象中没有真实数据(第一手值)，它内部封装了一套公式/表达式

一旦需要使用数据了，就会去计算公式，得到内容；

就因为其这样的特点，所以我们得出一些结论：

1).生成器比较节省内存资源(好处)

2).生成器（刚生成的时候没有值）所以在数据运算效率上比列表生成式要低下一点(弊端)

对于列表生成式而言：

定义执行后会得到一个list对象，

列表中的数据已经确定(真实数据已经进内存)，

就因为其这样的特点，所以我们得出一些结论：

1).列表生成式比较占用内存资源(弊端)

2).列表生成式在运算时比生成器要效率更好(好处)

*演示生成器的定义以及使用，关注其一些注意事项；* **方式一：  
格式：  
 (表达式 for 参数列表 in 容器对象 if...)**

gen = (x **for** x **in** range(5))  
print(gen,type(gen))  
*#generator对象不能配合len()来获取其长度，因为它根本没有长度  
# print(len(gen))  
#使用next()函数来得到每一次generator中的内容*print(next(gen))  *#得到0*  
print(next(gen))  *#得到1*  
print(next(gen))  *#得到2*print(next(gen))  *#得到3*  
print(next(gen)) *#得到4*  
*#generator对象中的数据内容只能被解析使用一次，不可逆；一旦调用多了，直接报错了 --> StopIteration  
# print(next(gen))* **两种处理接续generator对象的方式，  
方式一：使用next()函数调用传入generator对象即可，此方式如果调用next()次数过多；报错了：StopIterator  
方式二：使用循环处理，这种方式不会过量的调用内部表达式执行，不会报错，比较友好**

gen = (x **for** x **in** range(5))  
print(gen,type(gen)）

print(next(gen)) *#得到0*  
print(next(gen)) *#得到1*  
print(next(gen)) *#得到2*

**for** g **in** gen:  
 print(g) *#得到3*

*4*

*#注意下面一行代码有误，因为next()只能出现迭代器对象，对于单值int、float...不能传入的，报错：TypeError  
#* **print(next(g))**

**需求：思考如果对于以上gen对象中的数据我在开发过程中需要多次调用，那该怎么办？  
可以将generator类型的对象转换为list等类型的对象即可...**gen = (x **for** x **in** [1,2,3,4,5])  
*# print(gen,type(gen)) 此时gen只是一个函数对象，并没有被调用，所以没有内容*lt = list(gen)  
print(lt,type(lt)) *#得到[1, 2, 3, 4, 5] <class 'list'>***for** i **in** lt:  
 print(i) *#得到 1*

*2*

*3*

*4*

*5*print(**'-'** \* 40) *#得到 -----------------------------------  
#这时候 list类型有数据有长度了，可以用while循环*

i = 0  
**while** i < len(lt):  
 print(lt[i])  
 i += 1

*#同样得到 1*

*2*

*3*

*4*

*5***方式二：  
定义函数，在函数内部需要使用关键字yield来记录返回的generator对象的数据公式；  
执行还是通过next()函数来实现  
  
【注意】：此时函数就算有return，已经无视了，返回的一定是一个generator类型的对象  
def** func(n):  
 lt = []  
 **for** i **in** range(1,n + 1):  
 **if** i % 2 == 0:  
 lt.append(i)  
 **yield** i  
 **return** lt  
*#调用function（n)*

f = func(10)  
print(f,type(f)) *###虽然有返回值但返回的一定是一个generator类型的对像<generator object func at 0x0000000001E06938> <class 'generator'>*print(next(f)) *#得到2*  
print(next(f)) *#得到4*  
**for** i **in** f:   
 print(i) *#得到6 该循环接上面的走*

*#得到8*

*#得到10*

**但是上面的题目可以用方式一更简单：**

gen = (x **for** x **in** range(1,11) **if** x % 2 == 0)  
**for** g **in** gen:  
 print(g)

**#需求：实现全排列**str1 = **'ABC'**str2 = **'XYZ'  
def** func2(str1,str2):  
 **for** s1 **in** str1:  
 **for** s2 **in** str2:  
 **yield** s1 + s2 *#yield后面加的是方式方法*  
gen = func2(str1,str2)  
*# print(gen,type(gen))***for** g **in** gen:  
 print(g) *#得到 AX*

*AY*

*AZ*

*BX*

*BY*

*BZ*

*CX*

*CY*

*CZ*

*用简单的方法*  
gen = (x + y **for** x **in** str1 **for** y **in** str2)  
**for** g **in** gen:  
 print(g)

4.**可迭代对象Iterable和迭代器对象Iterator**：

可迭代对象：

满足前提：

可以被循环操作的对象，就是**可迭代对象**

举例：str、list、tuple、set、dict、range、generator...

**如何高效的检测一个对象是否是可迭代对象？**

可以使用collections模块中的**Iterable**类型配合isinstance(obj,type)内置函数来判断即可

操作步骤如下：

步骤一：导入collections模块

步骤二：collections.**Iterable**(类型)配合内置函数isinstance()来进行判断，代码如下：

isinstance(obj,collections.**Iterable**)

如果返回值为True，那么它就是一个可迭代对象

如果返回值为False，那么它不是一个可迭代对象

**演示如何判断是否是可迭代对象**

**import** collectionsstr1 = **'abcdefg'**lt = [1,2,3,4,5]  
tp = (10,20,30)  
s = {100,200,300,400}  
dic1 = {**'aa'**:**'AA'**,**'bb'**:**'BB'**}  
r = range(10)  
gen = (x **for** x **in** range(5))  
  
print(isinstance(str1,collections.Iterable)) *#True*  
print(isinstance(lt,collections.Iterable)) *#True*  
print(isinstance(tp,collections.Iterable)) *#True*  
print(isinstance(s,collections.Iterable))  *#True*print(isinstance(dic1,collections.Iterable)) *#True*print(isinstance(r,collections.Iterable))  *#True*print(isinstance(gen,collections.Iterable)) *#True*  
print(isinstance(100,collections.Iterable)) *#False 对于这种整数型数据100没有循环的概念*

**迭代器对象：**

满足前提：

1).必须是一个可迭代对象

2).可以被next()函数所调用的

举例：generator...

**如何高效的检测一个对象是否是迭代器对象**？

可以使用collections模块中的**Iterator**类型配合isinstance(obj,type)内置函数来判断即可

操作步骤如下：

步骤一：导入collections模块

步骤二：collections.**Iterator**(类型)配合内置函数isinstance()来进行判断，代码如下：

isinstance(obj,collections.**Iterator**)

如果返回值为True，那么它就是一个可迭代对象

如果返回值为False，那么它不是一个可迭代对象

**演示是否是迭代器对象**

**import** collectionsstr1 = **'abcdefg'**lt = [1,2,3,4,5]  
tp = (10,20,30)  
s = {100,200,300,400}  
dic1 = {**'aa'**:**'AA'**,**'bb'**:**'BB'**}  
r = range(10)  
gen = (x **for** x **in** range(5))  
  
print(isinstance(str1,collections.Iterator)) *#False*print(isinstance(lt,collections.Iterator)) *#False*  
print(isinstance(tp,collections.Iterator)) *#False*print(isinstance(s,collections.Iterator)) *#False*print(isinstance(dic1,collections.Iterator)) *#False*print(isinstance(r,collections.Iterator))  *#False*

print(isinstance(gen,collections.Iterator))  *#Ture 只有gen能被next()函数调用* **以下代码有问题：问题在于iterator对象才能被next()函数所调用***# print(next(lt)) #TypeError*

**再谈类型转换：  
将可迭代对象转换为迭代器对象  
内置函数：iter(iterable)**

*#假设一个内置函数****iter（）****里面必须传入一个叫iterable对象，将可迭代对象转换成迭代器对象  
#内置函数iter 传入lt对象 返回给it1对象*

lt = [1,2,3,4,5]  
tp = (10,20,30)

it1 = iter(lt)  
print(it1,type(it1))  *#<tuple\_iterator object at 0x00000000021BD160> <class 'tuple\_iterator'>*it2 = iter(tp)  
print(it2,type(it2))  
print(next(it1)) *#1*  
print(next(it2))  *#10*

*#***print(next(lt))** *而直接next()是会报错的,因为next()不会执行lt对象类型 TypeError: 'list' object is not an iterator* **以下代码有问题：  
因为3.14属于float类型，它并不是一个Iterable(可迭代对象)，  
所以不能被iter()函数转换称为Iterator(迭代器对象)***# obj = iter(3.14)  
# print(obj,type(obj))*

**【注意事项】：**

**内置函数：**

**iter(iterable)必须只能调用可迭代对象，否则报错**

**5.模块(model)**：

概念：在python中.py结尾的文件，我们称为模块；

作用：将类、函数、属性...内容定义其中

分类：

1).标准库模块：安装完python环境之后就存在的模块，这些模块我们如果想要使用，导入即可；

例如：random、time、os、collections、math...

2).自定义模块：在项目开发过程中，整个团队中程序员自己定义，可以给自己、别人进行调用

3).第三方模块：世界各地"他"人写的比较有价值的代码(面向全世界使用)，如果我们需要使用它，

需要通过pip安装模块，才能使用

导入方式：

1).精确导入：

举例：

import time #导入time模块

from random import randint #从random模块导入random函数

from math import pi #从math模块导入属性pi

2).模糊导入：

举例：

from math import \*

from os import \*

**演示标准库模块的导入：  
import** randomprint(random.randint(1,3))

**from** random **import** shuffle

lt = [1,2,3,4,5,6,7]  
shuffle(lt)

**from** math **import** pi,e

print(pi,e) *#3.141592653589793 2.718281828459045*

**from** time **import** \*

print(**'我睡了'**)  
sleep(2) #time模块中的函数，所以会自动执行的  
print(**'我醒了'**)

**给导入的模块或者函数、属性起别名：使用as关键字来实现**

**from** random **import** randint **as** r *#相当于吧randint改名为r*

print(r(5,10)) *#相当于print(randint(5,10)),得到5-10之间的随机整数*

【注意】：如果一旦给模块、函数、属性起了别名，原本的名字就不可用了；\*\*\*

*# print(randint(3,8)) #此行代码是非法的 因为randint已经被改了名字了*

**import** os **as** d  *#给os模块起别名*

print(d.getcwd()) *#是将当前正在执行的文件的绝对路径返回*

**演示自定义模块的导入：**

*假设两个py文件在同一层级 1-模块.py 和 func.py*

*func.py文件里面封装两个函数：1)求和 2).得到两个数中的较大值***def** my\_sum(a,b):  
 **return** a + b  
  
**def** my\_max(a,b):  
 **return** a **if** a > b **else** b

*#开始导入*

*#假设此时在1-模块.py文件下*

**import** func *#在1-模块.py层级导入func模块*

print(func.my\_sum(10,20)) #30  
print(func.my\_max(100,200)) #200

**from** func **import** my\_sum *#直接精确导入func模块中的my\_sum函数*

print(my\_sum(111,222)) #使用的时候就不用加func了

*不同层级，在当前模块中导入haha目录下的demo模块，使用其中的函数show()*

*Haha是文件夹 demo是py文件，是模块*

**以下三种方式都可以:**

**Import** haha.demo

haha.demo.show()

**From** haha **Import** demo

demo.show()

**From** haha.demo **Import** show

show()

对于自定义模块：

需要先显示的定义导入自定义模块到当前模块中，然后就可以去使用其内容

代码：

if \_\_name\_\_ == '\_\_main\_\_': 此代码的作用是将不想被导入加载的代码定义其中

**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 print(**'我是func模块中的业务代码，不应该在被外部导入加载时触发执行...'**)

"包"的概念：package

创建一个python package，创建成功就是一个所谓的python的包；

它的作用：将多个有关联的模块定义在其中，方便之后的维护和管理

注意：创建一个python package就会主动生成一个\_\_init\_\_.py ：所以对于\_\_init\_\_.py和\_\_pycache\_\_目录我们都不需要去关注他们，但别删他们

**第三方模块**：

打开cmd————>>输入pip查看是否安装完成pip

涉及到的部分操作:

1. 查看当前安装的所有第三方模块：pip list
2. 查看某个第三方模块的详细信息：pip show 模块名
3. 安装某个第三方模块：pip install 模块名
4. 删除某个第三方模块：pip uninstall 模块名

...