# 9.原理与拓展

## 9.1.数据类型的底层实现

### 9.1.1.列表的实现

**1、错综复杂的复制**

list\_1 = [1, [22, 33, 44], (5, 6, 7), {"name": "Sarah"}]

浅拷贝

*# list\_3 = list\_1 # 错误！！！*

list\_2 = list\_1.copy() *# 或者list\_1[:] \ list(list\_1) 均可实习浅拷贝*

对浅拷贝前后两列表分别进行操作

list\_2[1].append(55)

print("list\_1: ", list\_1)

print("list\_2: ", list\_2)

list\_1: [1, [22, 33, 44, 55], (5, 6, 7), {'name': 'Sarah'}]

list\_2: [1, [22, 33, 44, 55], (5, 6, 7), {'name': 'Sarah'}]

**2、列表的底层实现**

引用数组的概念

列表内的元素可以分散的存储在内存中

列表存储的，实际上是这些元素的地址！！！——地址的存储在内存中是连续的

list\_1 = [1, [22, 33, 44], (5, 6, 7), {"name": "Sarah"}]

list\_2 = list(list\_1) *# 浅拷贝 与list\_1.copy()功能一样*

（1）新增元素

list\_1.append(100)

list\_2.append("n")

print("list\_1: ", list\_1)

print("list\_2: ", list\_2)

list\_1: [1, [22, 33, 44], (5, 6, 7), {'name': 'Sarah'}, 100]

list\_2: [1, [22, 33, 44], (5, 6, 7), {'name': 'Sarah'}, 'n']

（2）修改元素

list\_1[0] = 10

list\_2[0] = 20

print("list\_1: ", list\_1)

print("list\_2: ", list\_2)

list\_1: [10, [22, 33, 44], (5, 6, 7), {'name': 'Sarah'}, 100]

list\_2: [20, [22, 33, 44], (5, 6, 7), {'name': 'Sarah'}, 'n']

（3）对列表型元素进行操作

list\_1[1].remove(44)

list\_2[1] += [55, 66]

print("list\_1: ", list\_1)

print("list\_2: ", list\_2)

list\_1: [10, [22, 33, 55, 66], (5, 6, 7), {'name': 'Sarah'}, 100]

list\_2: [20, [22, 33, 55, 66], (5, 6, 7), {'name': 'Sarah'}, 'n']

（4）对元组型元素进行操作

list\_2[2] += (8,9)

print("list\_1: ", list\_1)

print("list\_2: ", list\_2)

list\_1: [10, [22, 33, 55, 66], (5, 6, 7), {'name': 'Sarah'}, 100]

list\_2: [20, [22, 33, 55, 66], (5, 6, 7, 8, 9), {'name': 'Sarah'}, 'n']

（5）对字典型元素进行操作

list\_1[-2]["age"] = 18

print("list\_1: ", list\_1)

print("list\_2: ", list\_2)

list\_1: [10, [22, 33, 55, 66], (5, 6, 7), {'name': 'Sarah', 'age': 18}, 100]

list\_2: [20, [22, 33, 55, 66], (5, 6, 7, 8, 9), {'name': 'Sarah', 'age': 18}, 'n']

**3、深拷贝**

浅拷贝之后

针对不可变元素（数字、字符串、元组）的操作，都各自生效了

针对可变元素（列表、集合）的操作，发生了一些混淆

引入深拷贝

深拷贝将所有层级的相关元素全部复制，完全分开，泾渭分明，避免了上述问题

**import** **copy**

list\_1 = [1, [22, 33, 44], (5, 6, 7), {"name": "Sarah"}]

list\_2 = copy.deepcopy(list\_1)

list\_1[-1]["age"] = 18

list\_2[1].append(55)

print("list\_1: ", list\_1)

print("list\_2: ", list\_2)

list\_1: [1, [22, 33, 44], (5, 6, 7), {'name': 'Sarah', 'age': 18}]

list\_2: [1, [22, 33, 44, 55], (5, 6, 7), {'name': 'Sarah'}]

### 9.1.2.字典的实现

**1、查找非常快**

**import** **time**

ls\_1 = list(range(1000000))

ls\_2 = list(range(500))+[-10]\*500

start = time.time()

count = 0

**for** n **in** ls\_2:

**if** n **in** ls\_1:

count += 1

end = time.time()

print("查找**{}**个元素，在ls\_1列表中的有**{}**个，共用时**{}**秒".format(len(ls\_2), count,round((end-start),2)))

查找1000个元素，在ls\_1列表中的有500个，共用时6.19秒

**import** **time**

d = {i:i **for** i **in** range(1000000)}

ls\_2 = list(range(500))+[-10]\*500

start = time.time()

count = 0

**for** n **in** ls\_2:

**try**:

d[n]

**except**:

**pass**

**else**:

count += 1

end = time.time()

print("查找**{}**个元素，在ls\_1列表中的有**{}**个，共用时**{}**秒".format(len(ls\_2), count,round(end-start)))

查找1000个元素，在ls\_1列表中的有500个，共用时0秒

2、字典的底层实现

通过稀疏数组来实现值的存储与访问

字典的创建过程

第一步：创建一个散列表（稀疏数组 N >> n）

通过hash()计算键的散列值

d = {}

print(hash("python"))

print(hash(1024))

print(hash((1,2)))

-4771046564460599764

1024

3713081631934410656

893833775213427022

d["age"] = 18 *# 增加键值对的操作，首先会计算键的散列值hash("age")*print(hash("age"))

893833775213427022

第二步：根据计算的散列值确定其在散列表中的位置

极个别时候，散列值会发生冲突，则内部有相应的解决冲突的办法

第三步：在该位置上存入值

**for** i **in** range(2, 2):

print(i)

键值对的访问过程

d["age"]

第一步：计算要访问的键的散列值

第二步：根据计算的散列值，通过一定的规则，确定其在散列表中的位置

第三步：读取该位置上存储的值

如果存在，则返回该值

如果不存在，则报错KeyError

3、小结

（1）字典数据类型，通过空间换时间，实现了快速的数据查找

也就注定了字典的空间利用效率低下

（2）因为散列值对应位置的顺序与键在字典中显示的顺序可能不同，因此表现出来字典是无序的

回顾一下 N >> n  
 如果N = n，会产生很多位置冲突

思考一下开头的小例子，为什么字典实现了比列表更快速的查找

### 9.1.3.字符串的实现

通过紧凑数组实现字符串的存储，即数组的每个元素就是字符串的每个字符。

数据在内存中是连续存放的，效率更高，节省空间

思考一下，同为序列类型，为什么列表采用引用数组，而字符串采用紧凑数组

### 9.1.4.可变与不可变类型

**1、不可变类型：数字、字符串、元组**

在生命周期中保持内容不变

换句话说，改变了就不是它自己了（id变了）

不可变对象的 += 操作 实际上创建了一个新的对象

x = 1

y = "Python"

print("x id:", id(x))

print("y id:", id(y))

x id: 140718440616768

y id: 2040939892664

x += 2

y += "3.7"

print("x id:", id(x))

print("y id:", id(y))

x id: 140718440616832

y id: 2040992707056

元组并不是总是不可变的

t = (1,[2])

t[1].append(3)

print(t)

(1, [2, 3])

**2、可变类型：列表、字典、集合**

id 保持不变，但是里面的内容可以变

可变对象的 += 操作 实际在原对象的基础上就地修改

ls = [1, 2, 3]

d = {"Name": "Sarah", "Age": 18}

print("ls id:", id(ls))

print("d id:", id(d))

ls id: 2040991750856

d id: 2040992761608

ls += [4, 5]

d\_2 = {"Sex": "female"}

d.update(d\_2) *# 把d\_2 中的元素更新到d中*

print("ls id:", id(ls))

print("d id:", id(d))

ls id: 2040991750856

d id: 2040992761608

**3.列表操作的几个小例子**

【例1】 删除列表内的特定元素

方法1 存在运算删除法

缺点：每次存在运算，都要从头对列表进行遍历、查找、效率低

alist = ["d", "d", "d", "2", "2", "d" ,"d", "4"]

s = "d"

**while** **True**:

**if** s **in** alist:

alist.remove(s)

**else**:

**break**

print(alist)

['2', '2', '4']

方法2 一次性遍历元素执行删除

alist = ["d", "d", "d", "2", "2", "d" ,"d", "4"]

**for** s **in** alist:

**if** s == "d":

alist.remove(s) *# remove（s） 删除列表中第一次出现的该元素*

print(alist)

['2', '2', 'd', 'd', '4']

解决方法：使用负向索引

alist = ["d", "d", "d", "2", "2", "d" ,"d", "4"]

**for** i **in** range(-len(alist), 0):

**if** alist[i] == "d":

alist.remove(alist[i]) *# remove（s） 删除列表中第一次出现的该元素*

print(alist)

['2', '2', '4']

【例2】 多维列表的创建

ls = [[0]\*10]\*5

ls

[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

ls[0][0] = 1

ls

[[1, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[1, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[1, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[1, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[1, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

## 9.2.简洁的语法

### 9.2.1.解析语法

前面案例中创建多维列表，由于直接\*5，列表每个元素的位置是一样的。

如何解决这个问题？

ls = [[0]\*10 **for** i **in** range(5)]

ls

[[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

ls[0][0] = 1

ls

[[1, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

**1、解析语法的基本结构——以列表解析为例（也称为列表推导）**

[expression for value in iterable if conditihon]

三要素：表达式、可迭代对象、if条件（可选）

执行过程

（1）从可迭代对象中拿出一个元素

（2）通过if条件（如果有的话），对元素进行筛选

若通过筛选：则把元素传递给表达式

若未通过： 则进入（1）步骤，进入下一次迭代

（3）将传递给表达式的元素，代入表达式进行处理，产生一个结果

（4）将（3）步产生的结果作为列表的一个元素进行存储

（5）重复（1）~（4）步，直至迭代对象迭代结束，返回新创建的列表

*# 等价于如下代码*

result = []

**for** value **in** iterale:

**if** condition:

result.append(expression)

【例】求20以内奇数的平方

squares = []

**for** i **in** range(1,21):

**if** i%**2** == 1:

squares.append(i\*\*2)

print(squares)

[1, 9, 25, 49, 81, 121, 169, 225, 289, 361]

squares = [i\*\*2 **for** i **in** range(1,21) **if** i%**2** == 1]

print(squares)

[1, 9, 25, 49, 81, 121, 169, 225, 289, 361]

支持多变量

x = [1, 2, 3]

y = [1, 2, 3]

results = [i\*j **for** i,j **in** zip(x, y)]

results

[1, 4, 9]

支持循环嵌套

colors = ["black", "white"]

sizes = ["S", "M", "L"]

tshirts = ["**{}** **{}**".format(color, size) **for** color **in** colors **for** size **in** sizes]

tshirts

['black S', 'black M', 'black L', 'white S', 'white M', 'white L']

**2、其他解析语法的例子**

解析语法构造字典（字典推导）

squares = {i: i\*\*2 **for** i **in** range(5)}

**for** k, v **in** squares.items():

print(k, ": ", v)

0 : 0

1 : 1

2 : 4

3 : 9

4 : 16

解析语法构造集合（集合推导）

squares = {i\*\*2 **for** i **in** range(10)}

squares

{0, 1, 4, 9, 16, 25, 36, 49, 64, 81}

生成器表达式

squares = (i\*\*2 **for** i **in** range(10))

squares

<generator object <genexpr> at 0x000001DB37A58390>

colors = ["black", "white"]

sizes = ["S", "M", "L"]

tshirts = ("**{}** **{}**".format(color, size) **for** color **in** colors **for** size **in** sizes)

**for** tshirt **in** tshirts:

print(tshirt)

black S

black M

black L

white S

white M

white L

### 9.2.2.条件表达式

基本语法：

expr1 if condition else expr2

【例】将变量n的绝对值赋值给变量x

n = -10

**if** n >= 0:

x = n

**else**:

x = -nx

10

n = -10

x = n **if** n>= 0 **else** -n

x

10

条件表达式和解析语法简单实用、运行速度相对更快一些，相信大家会慢慢的爱上它们

## 9.3.生成器、迭代器和装饰器

### 9.3.1.生成器

正常生成1000000个元素的列表

ls = [i\*\*2 **for** i **in** range(1, 1000001)]

**for** i **in** ls:

**pass**

缺点：占用大量内存

**生成器**

（1）采用惰性计算的方式

（2）无需一次性存储海量数据

（3）一边执行一边计算，只计算每次需要的值

（4）实际上一直在执行next()操作，直到无值可取

**1、生成器表达式**

海量数据，不需存储

squares = (i\*\*2 **for** i **in** range(1000000))

**for** i **in** squares:

**pass**

求0~100的和

无需显示存储全部数据，节省内存

sum((i **for** i **in** range(101)))

5050

2、生成器函数——yield

生成斐波那契数列

数列前两个元素为1,1 之后的元素为其前两个元素之和

**def** fib(max):

ls = []

n, a, b = 0, 1, 1

**while** n < max:

ls.append(a)

a, b = b, a + b

n = n + 1

**return** ls

fib(10)

[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]

构造生成器函数

在每次调用next()的时候执行，遇到yield语句返回，再次执行时从上次返回的yield语句处继续执行，只有在调用的时候才生成，不调用的时候，并不会计算和存储。

**def** fib(max):

n, a, b = 0, 1, 1

**while** n < max:

**yield** a

a, b = b, a + b

n = n + 1

fib(10)

<generator object fib at 0x000001BE11B19048>

**for** i **in** fib(10):

print(i)

1

1

2

3

5

8

13

21

34

55

### 9.3.2.迭代器

**1、可迭代对象**

可直接作用于for循环的对象统称为可迭代对象：Iterable

（1）列表、元组、字符串、字典、集合、文件

可以使用isinstance()判断一个对象是否是Iterable对象

**from** **collections** **import** Iterable

isinstance([1, 2, 3], Iterable)

True

isinstance({"name": "Sarah"}, Iterable)

True

isinstance('Python', Iterable)

True

（2）生成器

squares = (i\*\*2 **for** i **in** range(5))

isinstance(squares, Iterable)

True

生成器不但可以用于for循环，还可以被next()函数调用

print(next(squares))

print(next(squares))

print(next(squares))

print(next(squares))

print(next(squares))

0

1

4

9

16

直到没有数据可取，抛出StopIteration

print(next(squares))

**StopIteration**:

**2.迭代器**

可以被next()函数调用并不断返回下一个值，直至没有数据可取的对象称为迭代器：Iterator

可以使用isinstance()判断一个对象是否是Iterator对象

**（1） 生成器都是迭代器**

**from** **collections** **import** Iterator

squares = (i\*\*2 **for** i **in** range(5))

isinstance(squares, Iterator)

True

**（2） 列表、元组、字符串、字典、集合不是迭代器**

isinstance([1, 2, 3], Iterator)

False

可以通过iter(Iterable)创建迭代器

isinstance(iter([1, 2, 3]), Iterator)

True

for item in Iterable 等价于：

先通过iter()函数获取可迭代对象Iterable的迭代器

然后对获取到的迭代器不断调用next()方法来获取下一个值并将其赋值给item

当遇到StopIteration的异常后循环结束

**（3）zip enumerate 等itertools里的函数是迭代器**

x = [1, 2]

y = ["a", "b"]

zip(x, y)

<zip at 0x1be11b13c48>

**for** i **in** zip(x, y):

print(i)

isinstance(zip(x, y), Iterator)

(1, 'a')

(2, 'b')

True

numbers = [1, 2, 3, 4, 5]

enumerate(numbers)

<enumerate at 0x1be11b39990>

**for** i **in** enumerate(numbers):

print(i)

isinstance(enumerate(numbers), Iterator)

(0, 1)

(1, 2)

(2, 3)

(3, 4)

(4, 5)

True

**（4）文件是迭代器**

**with** open("测试文件.txt", "r", encoding = "utf-8") **as** f:

print(isinstance(f, Iterator))

True

**（5）迭代器是可耗尽的**

squares = (i\*\*2 **for** i **in** range(5))

**for** square **in** squares:

print(square)

0

1

4

9

16

**（6）range()不是迭代器**

numbers = range(10)

isinstance(numbers, Iterator)

False

print(len(numbers)) *# 有长度*

print(numbers[0]) *# 可索引*

print(9 **in** numbers) *# 可存在计算*

next(numbers) *# 不可被next()调用*

10

0

True

**TypeError**: 'range' object is not an iterator

**for** number **in** numbers:

print(number)

### 9.3.3.装饰器

**1、需求的提出**

（1）需要对已开发上线的程序添加某些功能

（2）不能对程序中函数的源代码进行修改

（3）不能改变程序中函数的调用方式

比如说，要统计每个函数的运行时间

**2、函数对象**

函数是Python中的第一类对象

（1）可以把函数赋值给变量

（2）对该变量进行调用，可实现原函数的功能

**def** square(x):

**return** x\*\*2

print(type(square)) *# square 是function类的一个实例*

<class 'function'>

pow\_2 = square *# 可以理解成给这个函数起了个别名pow\_2*

print(pow\_2(5))

print(square(5))

25

25

可以将函数作为参数进行传递

**3、高阶函数**

（1）接收函数作为参数

（2）或者返回一个函数

满足上述条件之一的函数称之为高阶函数

**def** square(x):

**return** x\*\*2

**def** pow\_2(fun):

**return** fun

f = pow\_2(square)

f(8)

64

print(f == square)

True

**4、 嵌套函数**

在函数内部定义一个函数

**def** outer():

print("outer is running")

**def** inner():

print("inner is running")

inner()

outer()

outer is running

inner is running

**5、闭包**

闭包：

延伸了作用域的函数

如果一个函数定义在另一个函数的作用域内，并且引用了外层函数的变量，则该函数称为闭包

闭包是由函数及其相关的引用环境组合而成的实体(即：闭包=函数+引用环境)

**def** outer():

x = 1

z = 10

**def** inner():

y = x+100

**return** y, z

**return** inner

f = outer() *# 实际上f包含了inner函数本身+outer函数的环境*

print(f)

<function outer.<locals>.inner at 0x000001BE11B1D730>

print(f.\_\_closure\_\_) *# \_\_closure\_\_属性中包含了来自外部函数的信息*

**for** i **in** f.\_\_closure\_\_:

print(i.cell\_contents)

(<cell at 0x000001BE0FDE06D8: int object at 0x00007FF910D59340>, <cell at 0x000001BE0FDE0A98: int object at 0x00007FF910D59460>)

1

10

res = f()

print(res)

(101, 10)

一旦在内层函数重新定义了相同名字的变量，则变量成为局部变量

**def** outer():

x = 1

**def** inner():

x = x+100 *# x就变为局部变量，这里会报错*

**return** x

**return** inner

f = outer()

f()

**UnboundLocalError**: local variable 'x' referenced before assignment

nonlocal允许内嵌的函数来修改闭包变量

**def** outer():

x = 1

**def** inner():

**nonlocal** x

x = x+100

**return** x

**return** inner

f = outer()

f()

1

101

**6、一个简单的装饰器**

嵌套函数实现

**import** **time**

**def** timer(func):

**def** inner():

print("inner run")

start = time.time()

func()

end = time.time()

print("**{}** 函数运行用时**{:.2f}**秒".format(func.\_\_name\_\_, (end-start)))

**return** inner

**def** f1():

print("f1 run")

time.sleep(1)

f1 = timer(f1) *# 包含inner()和timer的环境，如传递过来的参数func*

f1()

inner run

f1 run

f1 函数运行用时1.00秒

语法糖

**import** **time**

**def** timer(func):

**def** inner():

print("inner run")

start = time.time()

func()

end = time.time()

print("**{}** 函数运行用时**{:.2f}**秒".format(func.\_\_name\_\_, (end-start)))

**return** inner

@timer *# 相当于实现了f1 = timer(f1)*

**def** f1():

print("f1 run")

time.sleep(1)

f1()

inner run

f1 run

f1 函数运行用时1.00秒

**7、装饰有参和有返回值函数**

有参

**import** **time**

**def** timer(func):

**def** inner(\*args, \*\*kwargs):

print("inner run")

start = time.time()

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

end = time.time()

print("**{}** 函数运行用时**{:.2f}**秒".format(func.\_\_name\_\_, (end-start)))

**return** inner

@timer *# 相当于实现了f1 = timer(f1)*

**def** f1(n):

print("f1 run")

time.sleep(n)

f1(2)

inner run

f1 run

f1 函数运行用时2.00秒

被装饰函数有返回值的情况

**import** **time**

**def** timer(func):

**def** inner(\*args, \*\*kwargs):

print("inner run")

start = time.time()

res = func(\*args, \*\*kwargs)

end = time.time()

print("**{}** 函数运行用时**{:.2f}**秒".format(func.\_\_name\_\_, (end-start)))

**return** res

**return** inner

@timer *# 相当于实现了f1 = timer(f1)*

**def** f1(n):

print("f1 run")

time.sleep(n)

**return** "wake up"

res = f1(2)

print(res)

inner run

f1 run

f1 函数运行用时2.00秒

wake up

**8、带参数的装饰器**

装饰器本身要传递一些额外参数

需求：有时需要统计绝对时间，有时需要统计绝对时间的2倍

**def** timer(method):

**def** outer(func):

**def** inner(\*args, \*\*kwargs):

print("inner run")

**if** method == "origin":

print("origin\_inner run")

start = time.time()

res = func(\*args, \*\*kwargs)

end = time.time()

print("**{}** 函数运行用时**{:.2f}**秒".format(func.\_\_name\_\_, (end-start)))

**elif** method == "double":

print("double\_inner run")

start = time.time()

res = func(\*args, \*\*kwargs)

end = time.time()

print("**{}** 函数运行双倍用时**{:.2f}**秒".format(func.\_\_name\_\_, 2\*(end-start)))

**return** res

**return** inner

**return** outer

@timer(method="origin") *# 相当于timer = timer(method = "origin") f1 = timer(f1)*

**def** f1():

print("f1 run")

time.sleep(1)

@timer(method="double")

**def** f2():

print("f2 run")

time.sleep(1)

f1()

print()

f2()

inner run

origin\_inner run

f1 run

f1 函数运行用时1.00秒

inner run

double\_inner run

f2 run

f2 函数运行双倍用时2.00秒

9、何时执行装饰器

一装饰就执行，不必等调用

func\_names=[]

**def** find\_function(func):

print("run")

func\_names.append(func)

**return** func

@find\_function

**def** f1():

print("f1 run")

@find\_function

**def** f2():

print("f2 run")

@find\_function

**def** f3():

print("f3 run")

run

run

run

**for** func **in** func\_names:

print(func.\_\_name\_\_)

func()

print()

f1

f1 run

f2

f2 run

f3

f3 run

## 9.4.作业练习

数据类型的底层实现：

1、以下两段程序分别会输出怎样的结果？请运行验证，并给出理由

代码1：

**def** f1(ls=[]):

ls.append(1)

**return** ls

print(f1())

print(f1())

print(f1())

代码2：

person = {"name": "", "id": 0}

team = []

**for** i **in** range(3):

x = person

x["id"] = i

team.append(x)

team[0]["name"] = "Peter"

team[1]["name"] = "Paul"

team[2]["name"] = "Mary"

print(team[1])

print(team)

简洁的语法：

2、用列表推导实现一个过滤器

\*实现一个判断数字n是否为素数的函数isprime(n)；

\*利用列表推导，获得100以内的素数列表；

3、使用列表推导，实现两列表对应元素的操作

获得新列表z，列表内元素

z[i] = pow(x[i], y[i])

x = [1，2, 3, 4]

y = [0，2, 3, 1]

4、使用条件表达式，将x，y 中的最大值赋值给z

编程题：

5、构造一个生产n 以内素数的生成器，与题2进行相互验证。

6、某大型网上购物网站进行中秋节优惠促销活动，请在不改变原计费函数

实现原计费函数 charge(商品名称, 商品数量 )。

假设只有三种商品，商品名称和单价存储在字典{“water”：1.5，“egg”: 1，“meat”: 15}中；

假设每次只购买一种商品，购买数量1 ~ 5 件, 不打折；购买数量6 ~ 10 件，打95 折；购买数量大于10 打9 折；

返回商品应付总货款。

在不改变原计费函数源代码和调用方式的前提下，增加如下功能：

 每次 charge 函数被调用时，输出“中秋节快乐!”；

 在原计费函数 charge 返回总价的基础上，对总价打8 折。

答案：

1. 代码1：

[1]

[1, 1]

[1, 1, 1]

代码2：

{'name': 'Mary', 'id': 2}

[{'name': 'Mary', 'id': 2}, {'name': 'Mary', 'id': 2}, {'name': 'Mary', 'id': 2}]

2.

\*是否为素数的函数

**from** **math** **import** sqrt

**def** is\_prime(num):

*'''判断一个数是否为素数'''*

**if** num == 1:

**return** **False**

**for** i **in** range(2, int(sqrt(num)) + 1):

**if** num % i == 0:

**return** **False**

**return** **True**

\*100以内的素数列表

[i **for** i **in** range(1, 100) **if** is\_prime(i)]

3.

**from** **math** **import** pow

x = [1, 2, 3, 4]

y = [0, 2, 3, 1]

z = [pow(x[i], y[i]) **for** i **in** range(len(x))]

z

[1.0, 4.0, 27.0, 4.0]

4.

x = 10

y = 12

z = x **if** x > y **else** y

z

12

5.

*#coding=utf-8*

*# 参考Blog：https://blog.csdn.net/ZSGG\_ACM/article/details/51584624*

*# '埃式素数生成法'*

**def** odd\_list():

*'''奇数生成器(不包括1)生成的是一个无限序列'''*

n = 1

**while** **True**:

n += 2

**yield** n

**def** filter\_not\_prime(n):

*'''过滤因子'''*

**return** **lambda** x : x % n *#返回闭包*

**def** primes():

*'''生成素数序列'''*

**yield** 2

it = odd\_list()

**while** **True**:

n = next(it)

**yield** n

it = filter(filter\_not\_prime(n), it)

**def** is\_prime(num):

*'''遍历素数生成器'''*

**for** i, x **in** enumerate(primes()): *# enumerate迭代遍历键值对(索引:素数)*

*# if x > num:*

**if** i > num:

**break**

**else**:

print(x)

**def** run():

is\_prime(100);

**if** \_\_name\_\_ == "\_\_main\_\_":

run()

6.

goods = {"water": 1.5, "egg": 1, "meat": 15}

**def** charge(goods\_name, goods\_count):

*'''计算购物所需花费'''*

cost = goods[goods\_name] \* goods\_count

**if** goods\_count **in** range(6):

**return** cost

**elif** goods\_count **in** range(11):

**return** cost \* 0.95

**elif** goods\_count > 10:

**return** cost \* 0.9

**def** shop(goods\_name, goods\_count):

print("中秋节快乐！")

**return** charge(goods\_name, goods\_count) \* 0.8

cost = shop("water", 3)cost

中秋节快乐！

3.6

cost = shop("meat", 20)cost

中秋节快乐！

216.0