第九章合并表达

> 本文是Python通用编程系列教程，已全部更新完成，实现的目标是从零基础开始到精通Python编程语言。本教程不是对Python的内容进行泛泛而谈，而是精细化，深入化的讲解，共5个阶段，25章内容。所以，需要有耐心的学习，才能真正有所收获。虽不涉及任何框架的使用，但是会对操作系统和网络通信进行全局的讲解，甚至会对一些开源模块和服务器进行重写。学完之后，你所收获的不仅仅是精通一门Python编程语言，而且具备快速学习其他编程语言的能力，无障碍阅读所有Python源码的能力和对

> 计算机与网络的全面认识。对于零基础的小白来说，是入门计算机领域并精通一门编程语言的绝佳教材。对于有一定Python基础的童鞋，相信这套教程会让你的水平更上一层楼。

一 三元表达式

在学习三元表达式之前，我们如需比较两个值的最大值。

```
def max2(x, y):
    if x > y:
        return x
    else:
        return y


res = max2(10, 11)
print(res)
```

三元表达式的使用

```
x = 12
y = 11

# 三元分别指的是if左边，else右边和if条件语句
res = x if x > y else y  # 注意不要有逗号或者冒号
print(res)
```

三元表达式仅应用于：条件成立返回 一个值，条件不成立返回一个值。

```
def max2(x, y):
    return x if x > y else y


print(max2(10, 11))
```

二 函数递归调用

1. 函数递归调用说明

函数的递归调用指的是在函数调用的过程中，又直接或间接地调用了函数本身，是函数嵌套调用的一种特殊形式。

```
# 直接调用
def foo():
    print('from foo')
    foo()


foo()



# 间接调用
def bar():
    print('from bar')
    foo1()


def foo1():
    print('from foo')
    bar()


foo1()
```

以上这两种递归会让程序陷入死循环中，为了避免程序占用大量的内存而是机器卡死，Python中有一个最大递归深度，所以你看到的报错其实是为了保护你的机器。

```
import sys

print(sys.getrecursionlimit())  # 默认最大递归深度1000

sys.setrecursionlimit(10000)  # 可以修改成10000甚至1亿，但是受限于机器硬件性能限制

print(sys.getrecursionlimit())
```

2. 递归阶段性说明

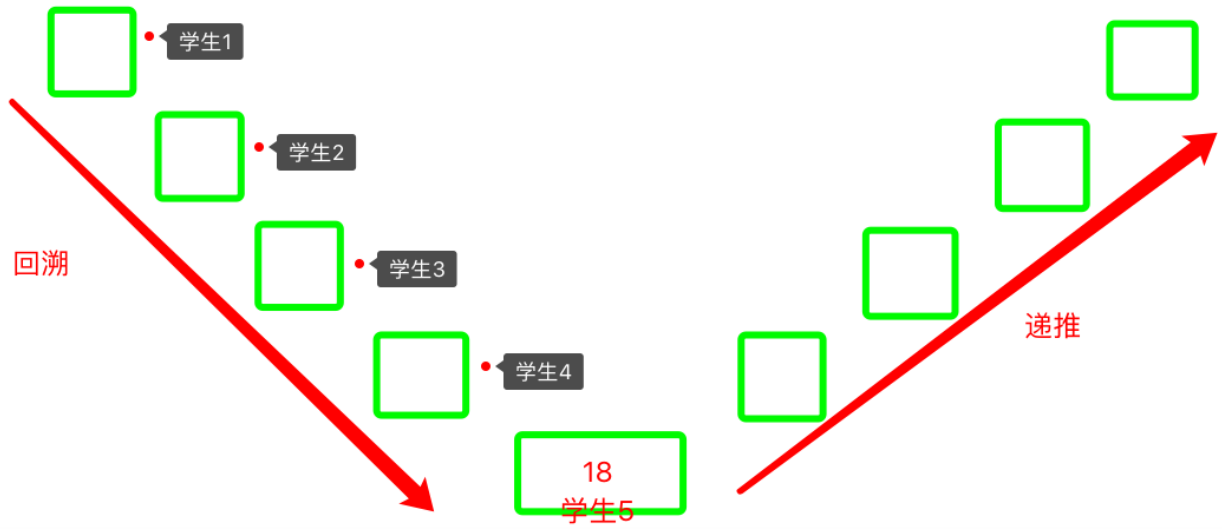
递归分为两个阶段：

1. 回溯
2. 递推

注意：一定要在满足某种条件结束回溯，否则的无限递归。

> 假如有这样一个小学数学题目，老师询问五位学生的年龄，第一位学生说，他比第二位学生大两岁，这时老师就要问第二个位学生，第二位学生说，他比第三位学生大两岁，问三，三说，比四大两岁，问四，四说，比五大两岁，问五，五说：“18岁”。求这五位学生分别多少岁？

老师依次询问的过程就是回溯，老师根据第五位学生的年龄推导出其他四位学生的年龄的过程就是递推，可用如下图表示。



把以上问题整理成数学表达式，并使用递归来完成。

```
def get_age(n):
    if n == 1:
        return 18
    return get_age(n - 1) + 2  # age(4)+2

    # 可以写成以下三元表达式形式，貌似高端，但是会让程序看起来不够简洁
    # return 18 if n == 1 else age(n - 1) + 2


print(get_age(5))
```

正确使用递归：

1. 递归一定要有一个明确地结束条件
2. 进入下一次递归之前，问题的规模都应该减小
3. 在Python中没有尾递归优化

> Python语言不像一些其他编程语言有尾递归优化，Python会全部保存每一层递归的状态，而一些其他的编程语言会去掉一些无结果的状态，这样能够从一定程度上更省内存。关于尾递归优化大家不必深究，需要提醒的是：以后再看到一些讲Python尾递归优化的文章，那就是瞎扯蛋，肯定扯不动，就直接跳过吧，因为Python根本就没有。

3. 递归的应用

递归也有循环的效果，与while循环不同的是：while循环一定要有一个明确的循环的条件，而递归不需要，在有些场景下使用递归能够让程序设计更简洁。

```
# 要求：取出列表中的1, 2, 3, 4, 5, 6.....
items = [1, [2, [3, [4, [5, [6, [7, [8, [9, [10, ]]]]]]]]]]


def tell(l):
    for item in l:
        if type(item) is not list:
            print(item)
        else:
            tell(item)


tell(items)
```

三 匿名函数

1. 匿名函数介绍

匿名函数就是没有名字的函数，我们以前讲过的函数都是要有一个函数名字的。

```python
# 有名函数
def func():  # 函数名为func可以多次调用
    print('from func')


func()
func()
func()


# 如果需要输出一个数的n次幂

# 使用有名函数
def foo(x, n):  # foo对应函数的内存地址
    return x ** n


# 使用匿名函数
"""
匿名函数使用规则：
    1 冒号左边为参数
    2 冒号右边为返回值
    3 不需要写return
"""
f = lambda x, n: x ** n  # 我们强行给匿名函数赋值一个变量是为了帮助你理解
print(f)
print(f(2, 3))
print(f(2, 4))
```

2. 匿名函数使用规则

1. 匿名的目的就是要没有名字，给匿名函数赋给一个名字是没有意义的
2. 匿名函数的参数规则、作用域关系与有名函数是一样的
3. 匿名函数的函数体通常应该是一个表达式,该表达式必须要有一个返回值

如果一个匿名函数的函数体没有返回值，那么这是没有意义的。

```python
# 没有意义
f = lambda x, n: print('=======>')  # print肯定是没有返回值的
print(f(2, 3))
```

```
# 有意义
f1 = lambda x, y, z: x + y + z
print(f1(1, 2, 3))
```

3. 匿名函数的应用

(1) max，min，sorted

> 我们一直在强调匿名函数本来就是没有名字的，给他赋值没有意义，那么我们应该怎么使用?

匿名函数一般不会单独使用，常与我们接下来讲的内置函数联用。

```
print(max([3, 2, 5, 44, 66, 11, 13, ]))
print(min([3, 2, 5, 44, 66, 11, 13, ]))
print(sorted([3, 2, 5, 44, 66, 11, 13, ]))  # 升序
print(sorted([3, 2, 5, 44, 66, 11, 13, ], reverse=True))  # 降序
```

(2) 匿名函数与max，min，sorted联用

```
"""
求工资最高的那个人是谁
求工资最低的那个人是谁
把这些人按薪资待遇排序
"""
salaries = {
    'james': 300000,
    'kd': 100000,
    'zimuge': 10000,
    'harden': 90000
}

print(max(salaries))
print(min(salaries))
print(sorted(salaries))
```

显然这个结果是不对的，max，min和sorted它们的原理其实都是for循环，我们比较的依据是薪资，而for循环字典默认比较的字典的`key`，所以，你看到的结果是按照自己字符串的大小(字符编码表的顺序)比较的。


```
"""
求工资最高的那个人是谁
求工资最低的那个人是谁
把这些人按薪资待遇排序
"""
salaries = {
    'james': 30000000,
    'kd': 10000000,
    'zimuge': 1000000,
    'harden': 9000000
}


def get(k):
    return salaries[k]


# max可以指定key，代表比较依据，与字典的键重名纯属巧合
print(max(salaries, key=get))  # 这里需要传一个for循环迭代而来的参数，并输入一个结果

# max比较原理
"""
for k in salaries:
    print(k)  # 告诉max，比较的依据是salaries[k]
"""

# 显然我们定义一个get函数是不需要的，这时可以使用匿名函数
print(max(salaries, key=lambda x: salaries[x]))

# 求工资最低的那个人是谁
print(min(salaries, key=lambda x: salaries[x]))

# 把这些人按薪资待遇排序
salaries1 = sorted(salaries, key=lambda x: salaries[x])  # 升序
print(salaries1)

salaries2 = sorted(salaries, key=lambda x: salaries[x], reverse=True)  # 降序
print(salaries2)
```

(3) 匿名函数与map，reduce，filter联用

> 在计算机中map，reduce和filter分别指的是映射，合并和过滤，这个三个词是与大数据的概念息息相关的，通过海量数据的采集，经过映射之后难道需要的数据，然后在把多方数据合并，最后过滤出来需要的数据，这就是大数据的概念，最终到达的目的是分析预测。假如你在某猫公司买了10次商品，其中有9次是假货，你都容忍了，那么基本上就能预测你这个人了，以后也没必要给你发真货了；再假如你在某狗公司买了2次商品，都给你发了假货，你每次都投诉，闹的天翻地覆，程序也能预测到你这个人，以后就只给你推荐贵的商品，保证真货。

map，reduce和filter最好与匿名函数联用才能发挥作用，所以我们不再单独介绍。

```
# 1 map

# 数字映射
nums = [1, 2, 3, 4, 5]

# map第一个参数是函数，即映射方式，第二个参数是可以迭代对象
res = map(lambda x: x ** 2, nums)

# 在Python2中res是一个列表，Python3中是一个迭代器
print(res)

# list用for循环的原理把迭代器转化成列表
print(list(res))

# 字符串映射
names = ['James', 'Harden', 'Curry']
res = map(lambda x: x + ' is super star', names)
print(list(res))

names = ['James', 'Harden', 'Curry', 'Albert']
res = map(lambda x: x + ' is referee' if x == 'Albert' else x + ' is super
star', names)

print(list(res))

# 2 reduce
"""
reduce在Python2中还是一个内置函数
在Python3中需要导入一下，下一章节就会讲解模块与包的导入
"""
```

```python
from functools import reduce  # 这样代码是导入reduce

# 数字合并
"""
reduce可以传三个参数：
    第一个是必传项，指的合并规则，即函数
    第二个是必传项，指的可迭代对象
    第三个是可选项，指的初始值
"""
# 计算1+2+3+++++100
res1 = reduce(lambda x, y: x + y, range(1, 101), 0)
# 初始值给x，可迭代对象内的值给y，相加之后再作为初始值给x，可迭代对象内新的值给y
print(res1)

res = reduce(lambda x, y: x + y, range(1, 101))
# 不指定初始值，则由可迭代对象的第一个值作为初始值
print(res)

# 字符串合并
list1 = ['Today', 'is', 'the', 'first', 'day', 'of', 'the', 'rest', 'of',
'your', 'life']
res = reduce(lambda x, y: x + ' ' + y + ' ', list1)
print(res)

# 3 filter

# 过滤出年龄不小于30的
ages = [18, 19, 10, 23, 99, 30]
# res = filter(lambda n: True if n >= 30 else False, ages)
# filter只能过滤真假，上面写写法有点啰嗦
res = filter(lambda n: n >= 30, ages)
print(list(res))

# 过滤出裁判
names = ['James is super star', 'Harden is super star', 'Albert is referee']
res = filter(lambda x: x.endswith('referee'), names)
print(list(res))
```

四 内置函数

1. format用法

(1) format三种基本用法

```
res = '{} {} {}'.format('Albert', 18, 'male')  # 相当于%s，其占位符的作用
print(res)
res = '{1} {0} {1}'.format('Albert', 18, 'male')  # 按照索引对应值，与索引序列相关
print(res)
res = '{name} {age} {sex}'.format(sex='male', name='Albert', age=18)  # 按照关键字对应值，只与key对应
print(res)

# Albert18 male
# 18 Albert 18
# Albert 18 male
```

(2) format与%s用法异同

<1> 相同点

formate与% 都可以用作对象属性格式化，两者都是用作字符串中的占位符，一般基础阶段的程序员只会用%，而且不管多麻烦，你都会想到用%的办法，我首先为你的执着点个赞，但是我要说的是，简单的可以用%，复杂一些，我还是推荐使用formate或者说是 { }。

```
class Person(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return "This guy is %s, %s years old." % (self.name, self.age)


p1 = Person('albert', 18)
print(p1)


class Person(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

```
    def __str__(self):
        return "This guy is {self.name}, {self.age} years
old.".format(self=self)


p1 = Person('albert', 18)
print(p1)
```


<2> 不同点

[1] 占位形态

```
# 定义一个坐标值
c = (250, 250)
# 使用%来格式化
s1 = "敌人坐标：%s" % c
print(s1)

"""
报错说明：
not all arguments converted during string formatting
翻译：在执行字符串格式化期间，不是所有的字符串都能被转化
其实道理很简单，因为执行字符串转化的时候，程序以为变量c是一个字符串才去进行转化的，然而一转化，
才发现：
老兄，你坑我啊，你这个字符串我怎么看不明白啊（用 % 去执行这个程序的时候，默认以为 c 是一个字符
串），
从始至终程序都是为c是一个字符串，其实它不是，所以才会有这样的报错
"""

# 定义一个坐标值
c = (250, 250)
# 使用format来格式化
s1 = "敌人坐标：{}".format(c)
print(s1)

"""
区别：
    format本意就是变形的意思，72变，需要什么我就变什么
    % 是一个基本的占位符，只能给字符串，哦，怕程序不理解，是真正的字符串占位
"""
```

[2] f-strings

这里的新特性f-strings其实和format没有直接关联，只是使用的format的特性来进行格式化，这个新特性只在Python3.6 才有，以后有没有只有龟叔知道了。

```
name = 'Albert'
print(f'he said his name is {name}')  # 注意字符串前面的f
```

> PEP 498 introduces a new kind of string literals: f-strings, or formatted string literals.Formatted string literals are prefixed with 'f' and are similar to the format strings accepted by str.format(). They contain replacement fields surrounded by curly braces. The replacement fields are expressions, which are evaluated at run time, and then formatted using the format() protocol.
> 以下是我的翻译：
> PEP 498引入了一种新的字符串文本:f字符串，或格式化的字符串文字。格式化字符串文字以'f'开头，类似于以string .format()形式传入的格式化字符串。它们包含由花括号包围的替换字段。替换字段是表达式，在运行时进行计算，然后使用format()协议进行格式化。

[3] 填充与对齐

填充常跟对齐一起使用，^、<、>分别是居中、左对齐、右对齐，后面带宽度，冒号后面带填充的字符，只能是一个字符，不指定的话默认是用空格填充。

```
print("{:>10}".format('18'))
print("{:3<10}".format('18'))
# print("{:aa<10}".format('18')) #写两个字符就会报错 ，错误就是让写一个，你偏要写俩，老子看球不懂
print("{:*^10}".format('18'))
```

补充一个字符串自带的zfill()方法，Python zfill()方法返回指定长度的字符串，原字符串右对齐，前面填充0。zfill()方法语法：str.zfill(width)，参数width指定字符串的长度。原字符串右对齐，前面填充0，返回指定长度的字符串。

```
print('a'.zfill(10))
```

[4] float精度调整

```
print('{:.5f}'.format(3.14159265753))

# 进度通常和float类型一起使用
# 其中.5表示长度为5的精度
# 3.14159
```

[5] 千位分隔符

```
print("{:,}".format(1234567890))

# '1,234,567,890'
```

[6] 进制转换

b、d、o、x分别是二进制、十进制、八进制、十六进制

```
print("{:b}".format(18))
print("{:d}".format(18))
print("{:o}".format(18))
print("{:x}".format(18))
```

2. 其他内置函数

```
# 1 取绝对值
print(abs(-1))  # 绝对值

# 2 逻辑判断
print(all([1, 'a', True]))  # 列表中所有元素的布尔值为真，最终结果才为真
print(any([0, 'a', None, False]))  # 列表中所有元素的布尔值只要有一个为真，最终结果就为真
print(any([]))  # 传给any的可迭代对象如果为空，最终结果为假
```

```python
# 3 进制转换
print(bin(11))  # 十进制转二进制
print(oct(11))  # 十进制转八进制
print(hex(11))  # 十进制转十六进制

# 4 布尔值判断
print(bool(0))  # 0,None,空的布尔值为假

# 5 bytes类型构造
# res='你好Albert'.encode('utf-8') # unicode按照utf-8进行编码，得到的结果为bytes类型
res = bytes('你好Albert', encoding='utf-8')  # 同上
print(res)


# 6 对象是否可调用
def func():
    pass


print(callable(func))  # 判断某个对象是否是可以调用的，可调用指的是可以加括号执行某个功能
print(callable('abc'.strip))  # 判断某个对象是否是可以调用的，可调用指的是可以加括号执行某个功能
print(callable(max))  # 判断某个对象是否是可以调用的，可调用指的是可以加括号执行某个功能

# 7 字符与十进制转换
print(chr(90))  # 按照ascii码表将十进制数字转成字符
print(ord('Z'))  # 按照ascii码表将字符转成十进制数字

# 8 查看对象下可调用的方法
print(dir('abc'))  # 查看某个对象下可以调用到哪些方法

# 9 去商和余数，返回元组
print(divmod(1311, 25))

# 10 将字符内的表达式拿出运行一下，并拿到该表达式的执行结果
res1 = eval('2*3')
print(res1, type(res1))
res2 = eval('[1,2,3,4]')
print(res2, type(res2))
res3 = eval('{"name":"Albert","age":18}')
print(res3, type(res3))

# 11 集合添加
s = {1, 2, 3}
```

```python
s.add(4)  # 集合添加一个元素
print(s)

# 12 不可变集合
f_set = frozenset({1, 2, 3})  # 不可变集合

# 13 名字与值的绑定关系
x = 2
print(globals())  # 查看全局作用域中的名字与值的绑定关系
print(dir(globals()['__builtins__']))


def func():
    x = 1
    print(locals())  # 查看局部作用域中的名字与值的绑定关系


func()

# 14 哈希
print(hash('a'))
print(hash((1, 2, 3, 4)))


# dict1 = {[1, 2, 3]: 'a'}  # 报错，字典的key必须是不可变类型

# 不可hash的类型list,dict,set==  可变的类型
# 可hash的类型int,float,str,tuple ==  不可变的类型

# 哈希下一章节会有更加详细的讲解，这里先了解即可

# 15 查看文档注释
def func():
    """
    帮助信息
    :return:
    """
    pass


print(help(max))  # 查看文档注释
print(help(func))

# 16 len，next，iter等自动执行的方法
# __开头__结尾的方法都是在某种情况下自动触发执行，通常我们不需要自己写
```

```python
print(len({'x': 1, 'y': 2}))  # 触发 {'x':1,'y':2}.__len__()
print({'x': 1, 'y': 2}.__len__())  # 与上面等同
obj = iter('Albert')  # 'Albert'.__iter__()
print(next(obj))  # obj.__next__()

# 17 2**3 % 3
print(pow(2, 3, 3))  # 2 ** 3 % 3

# 18 反转顺序
l = [1, 4, 3, 5]
res = reversed(l)
print(list(res))
print(l)

# 19 四舍五入
print(round(3.5))
print(round(3.4))

# 20 切片对象
sc = slice(1, 5, 2)  # 1:5:2
l = ['a', 'b', 'c', 'd', 'e', 'f']
# print(l[1:5:2])
print(l[sc])

t = (1, 2, 3, 4, 5, 6, 7, 8)
# print(t[1:5:2])
print(t[sc])

# 21 求和
print(sum([1, 2, 3, 4]))

# 22 拉链函数
left = 'hello'
right1 = {'x': 1, 'y': 2, 'z': 3}
right2 = [1, 2, 3, 4, 5]

res1 = zip(left, right1)
res2 = zip(left, right2)
print(list(res1))
print(list(res2))
```

五 列表生成式

1. 列表生成式基本使用

列表生成式就是生成列表的表达式，使用通常的理解是为了使我们的程序设计变得更加简洁。

```
# 要求：造出来1000个蛋
l = []
for i in range(1000):
    l.append('egg%s' % i)

# 使用列表生成式
l1 = ['egg%s' % i for i in range(1000)]
l2 = ['egg%s' % i for i in range(1000) if i > 10]  # 可以有if条件，注意和三元表达式的区别
print(l1)
print(l2)

# 列表生成式语法格式
"""
[
expression for item1 in iterable1 if condition1
                for item2 in iterable2 if condition2
                    ...
                        for itemN in iterableN if conditionN
]
"""
```

2. 使用注意事项

使用列表生成式可以有多个for循环和多个if条件，但是一般情况下我们只写一组，就像上面代码中写的，如果你在列表生成式中写了多个for循环和if条件就丧失了程序设计的简洁性，这并不符合Python的语法风格。

3. 列表生成式提升效率

从底层上讲，列表生成式是在构建列表的过程中，在列表中放入多个元素，而我们前面普通的方式是先创建一个空列表，再逐一添加元素，很明显这种普通的方式会徒增插入的时间，这个时间虽然并不是很多，但是当数据量足够大的时候会有明显的差距。

> 举例说明一下，你需要一个一个的清洗10个苹果，然后再把这10个苹果放进冰箱，第一种方案：把10个苹果分别清洗好了，逐一放进冰箱，这中间肯定有多次重复开关冰箱门的操作；第二种方案：先清洗好10个苹果，一次开关冰箱门，统一放进冰箱。

```
"""
因为我们要造的数据内容是一样的，为了避免你多余的顾虑（后面的代码覆盖前面的数据，这其实不存在）
因而后面的可能会快一些这种错误的想法，我们把列表生成式放到前面
"""

import time

# 使用列表生成式
time1 = time.time()

l1 = ['egg%s' % i for i in range(1000000)]

time2 = time.time()
print(time2 - time1)

# 使用普通方式
time3 = time.time()

l = []
for i in range(1000000):
    l.append('egg%s' % i)
time4 = time.time()

print(time4 - time1)
```

你可以清楚的看到使用列表生成式的执行效率明显高于普通的方式，所以，在合适的场景下使用列表生成式是一个不错的选择，既能简化代码，又能提高效率，但是比如有些情况，你需要优化程序，而所有的优化手段都用上了还不够，再优化的话就是使用比较复杂的列表生成式了，这也未尝不可，具体还要自己权衡。我需要告诉你的是：

1. 如果真的要用复杂的列表生成式，前提一定是你已经使用了所有的其他优化方式还不能到达预期，因为列表生成式提升效率并不是按照数量级来提升的。
2. 提升硬件性能的优化远比从软件上深层优化容易得多，提升硬件性能的花费一定会比工程师时间上的花费所兑换的金钱要少。

六 生成器表达式

列表生成式会构造出一个列表，如果数据量非常大，那么会构造出一个非常大的列表，占用大量的内存，机器就卡了，建议你自己试一下，以后就会深有体会(你正常使用的卡顿与机器真的卡了差别很大)，卡机代码如下：

```
l1 = ['egg%s' % i for i in range(10000000000000000000000000000000000000)]
```

所以，我们有了生成器表达式，返回的值是一个生成器，可以迭代取值。

```
# 只需要把中括号改成小括号就可以了，其他用法一样
l = ('egg%s' % i for i in range(10000000000000000000000000) if i > 100000)
print(next(l))
print(next(l))
print(next(l))
```