

对小数点精度保留：

1. 四舍五入式：

round(x,m):对x进行到m位小数点的四舍五入保存（但是print不会自动补充0到第m位）

f”{x:.2f}”保留两位（会自动补充到n位）

1. 完全截断式（无现成函数）

high=int(high\*100)

return (f"{high / 100:.2f}")n,k=map(int,input().split())

一个典型的截断式数据处理和print设计

！！print中给它int形式的2.00会给你print出来2.0（保留一位）

def count(i):#判断函数

low=0

high=max(we)

def s(low,high):#内外的list名称可以不变公用，但是单一变量还是要给出全局描述或者给函数嵌入的

while high-low>0.001:

if high<0.01:

return "0.00"#要用str形式

操作mid

high=int(high\*100)

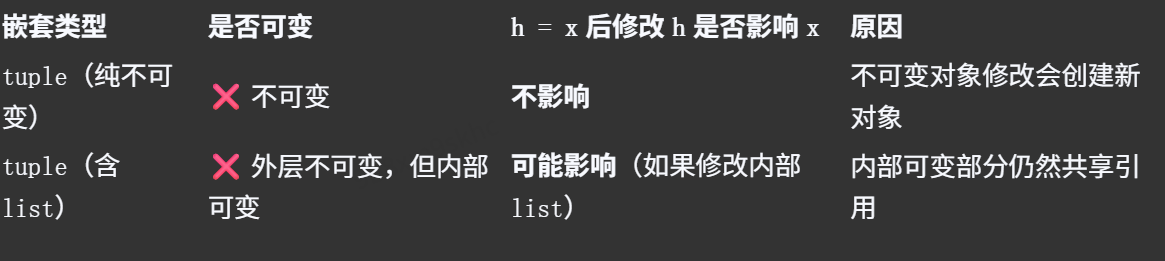
return (f"{high / 100:.2f}")

print(s(low,high))

list中的子对象为列表时候引用和改变带来的问题：

| **操作** | **效果** | **适用场景** |
| --- | --- | --- |
| h = x | h 和 x 指向同一对象（联动修改） | 需要引用原列表时 |
| h = x.copy() | h 是 x 的独立副本（浅拷贝） | 单层列表的复制 |
| h = x[:] | 同上（切片法浅拷贝） | 同上 |
| h = deepcopy(x) | h 是完全独立的副本（深拷贝） | 嵌套列表或多层结构[[a],[b]] |

如果list中的子对象限制在是**不可变对象**即tuple，str，int，float等不可变对象的时候就不用考虑，可变对象如dict,list,set



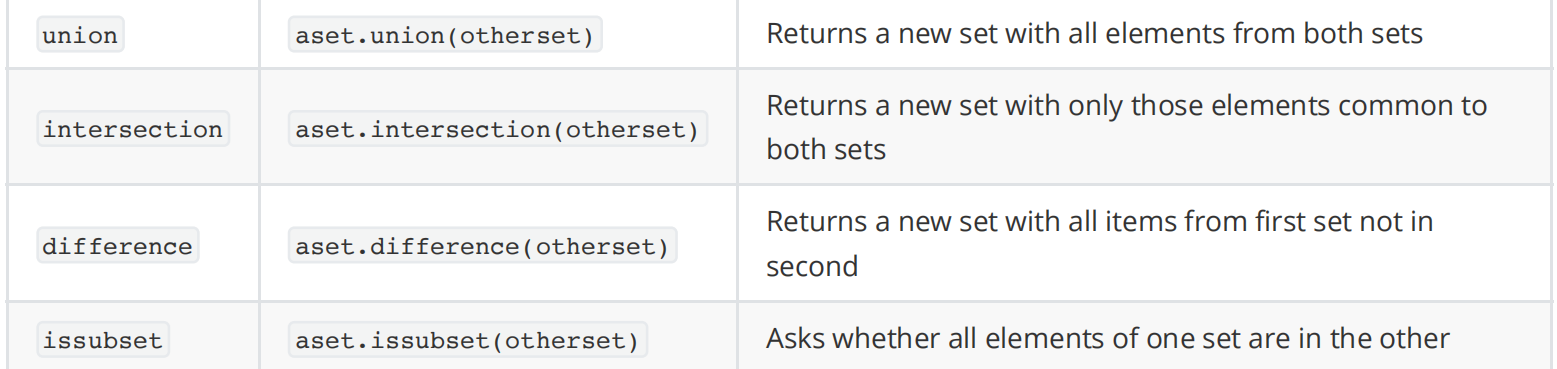
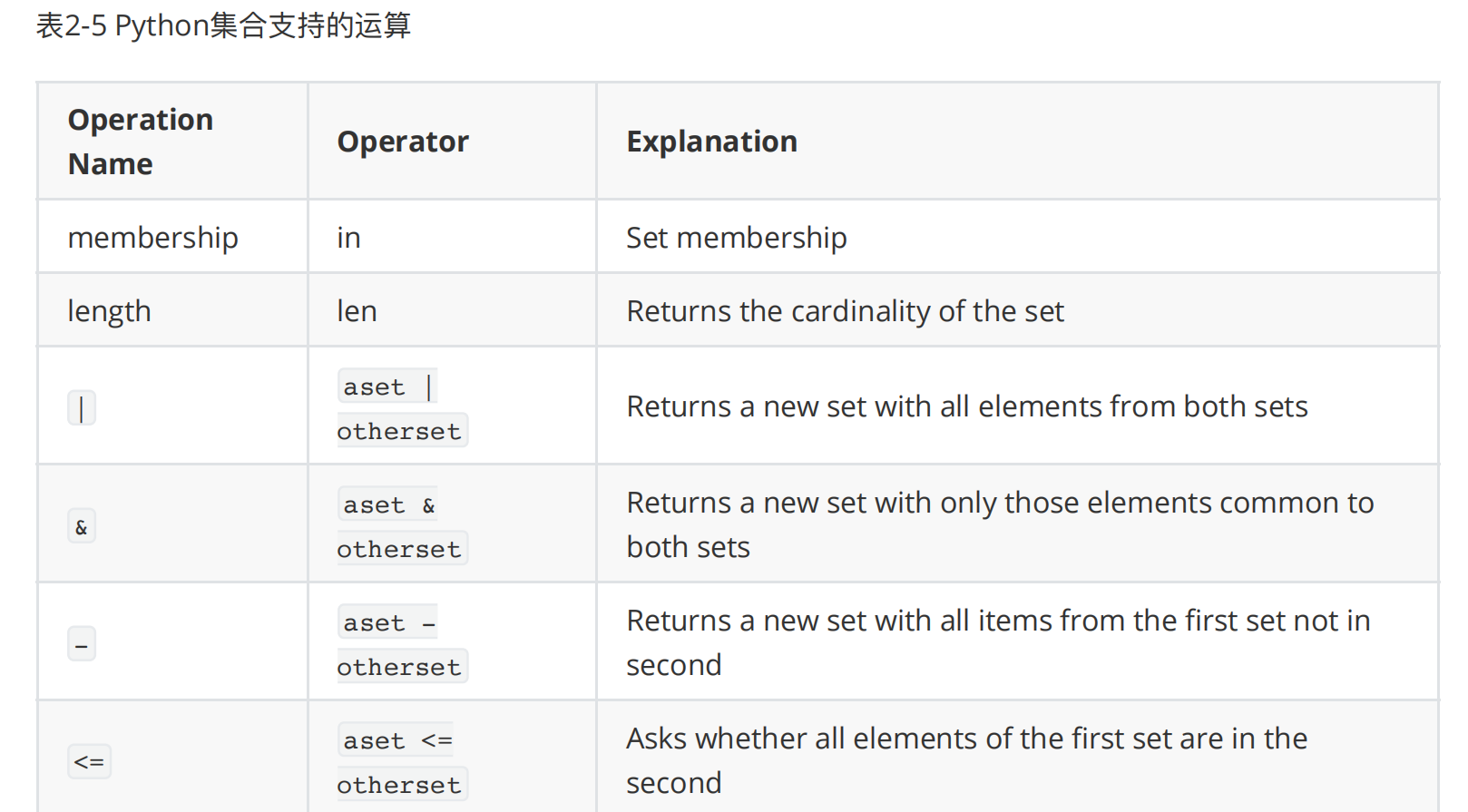
复习：欧几里得算法最大公约数；def gcd(a, b):

while b != 0:

a, b = b, a % b

return a

集合运算



## 卡特兰数的应用

通用代码：

n = int(input())

dp = [0] \* (n + 1)

dp[0] = 1

for i in range(1, n + 1):

for j in range(i):

dp[i] += dp[j] \* dp[i - 1 - j]

print(dp[n])

def catalan(n):

c = 1

for i in range(n):

c = c \* 2 \* (2 \* i + 1) // (i + 2)

return c

## 链表翻转代码：

curr\_head = prev\_group\_end.next # 当前组的原始头（反转后变为尾）

prev = None

curr = curr\_head

for \_ in range(k):

next\_node = curr.next

curr.next = prev

prev = curr

curr = next\_node

prev\_group\_end.next = prev # prev是当前组新头



from itertools import permutations，product

data = ['A', 'B', 'C']

perms = permutations(data, r=2) # 生成长度为 2 的**排列，不是重复**

binary\_perms = product([0, 1], repeat=8)#生成对01的8重可能

## tree：

Segment Tree线段树：

在测算区间数值的时候好用，

运行逻辑：  
对长度为n的数据建立新列表x

使得从索引n到索引2n-1是元数据组对应的数据

每个索引x对应的子树是2x (x<<1) and 2x+1(x<<1|1)

每个叶对应的兄弟叶是x^1（偶数+1，奇数-1）

对应的父节点是i>>1

根节点索引是1（ps，一个完全的二叉树的父节点总个数也只是n-1）

建立逻辑：

def build(arr) :

# insert leaf nodes in tree

for i in range(n) :

tree[n + i] = arr[i];

# build the tree by calculating parents

for i in range(n - 1, 0, -1) :

tree[i] = tree[i << 1] + tree[i << 1 | 1]#从后往前建树

更新逻辑

# function to update a tree node

def updateTreeNode(p, value) :

# set value at position p

tree[p + n] = value

p = p + n

# move upward and update parents

i = p

while *i > 1 :*

tree[*i >> 1*] = tree[i] + tree[i ^ 1]#看更新极限点在哪

i >>= 1

**建树：**

扩充二叉树：

def build\_tree(s, index):

# 如果当前字符为'.'，表示空结点，返回None，并将索引后移一位

if s[index] == '.':

**return None, index + 1#（表示终结一段left/right建树）**

# 否则创建一个结点

node = Node(s[index])

index += 1

# 递归构造左子树

**node.left, index = build\_tree(s, index)#这个index的留存很妙，在left建树n层到底之后，返回最后一次用的index，然后从高层开始建起right**

# 递归构造右子树

node.right, index = build\_tree(s, index)#在left结束之后，高层right优先开始建树

return node, index

**建树的node暂时留存和入stack长久保存结合的建树思维**

def parse\_tree(s):

stack = []**#stack作为长久储存的栈，最后一位用来储存还没等到）右括号结束的node，等到后弹出**

node = None**#node作为暂时保存的量，用来操作建立新节点和稍暂存储等待（左括号出现后入其children的作用**

for char in s:

if char.isalpha(): # 如果是字母，创建新节点

node = TreeNode(char)

if stack: # 如果栈不为空，把节点作为子节点加入到栈顶节点的子节点列表中stack[-1].children.append(node)

elif char == '(': # 遇到左括号，当前节点可能会有子节点

if node:

stack.append(node) # 把当前节点推入栈中

node = None

elif char == ')': # 遇到右括号，子节点列表结束

if stack:

node = stack.pop() # 弹出当前节点

return node # **根节点留root**

后序建树：

def build\_tree(postfix):

stack = []

for char in postfix:

node = TreeNode(char)

if char.isupper():

node.right = stack.pop()

node.left = stack.pop()

stack.append(node)

return stack[0]

源码页

24591: 中序表达式转后序表达式

stack, http://cs101.openjudge.cn/practice/24591/

思路：

用oi表示括号层数，用compare排列优先级别，更优先的更早输出

for \_ in range(int(input())):

stack=[]

output=[]

i=input().strip()

oi=0

ind=0

compare={"+":1,"-":1,"\*":2,"/":2}

while ind < len(i):

r=i[ind]

if r =="(":

oi+=1

elif r.isdigit():

for t in range(ind,len(i)):

if i[t].isdigit() or i[t]==".":

if t==len(i)-1:

t=len(i)

break

continue

break

output.append(i[ind:t])

ind=t

continue

elif r==")":

while stack and oi==stack[-1][-1]:

output.append(stack.pop()[0])

oi-=1

else:

while stack and compare[stack[-1][0]]>=compare[r] and oi == stack[-1][-1]:

output.append(stack.pop()[0])

stack.append((i[ind],oi))

ind+=1

while stack:

sm=stack.pop()

output.append(sm[0])

print(" ".join(map(str,output)))

## 记忆化+颜色标记在同一个列表判环

class Solution:

def largestPathValue(self, colors: str, edges: List[List[int]]) -> int:

n = len(colors)

g = [[] for \_ in range(n)]

for x, y in edges:

if x == y: # 自环

return -1

g[x].append(y)

memo = [None] \* n#memo同时承担**判环和记录**两个责任

def dfs(x: int) -> Dict[str, int]:

if memo[x] is not None: # x 计算中或者计算过

return memo[x] # 如果是 0，表示有环

memo[x] = 0 # 用 0 表示计算中#dfs的函数表示更容易用标记颜色法，计算完之后记忆化，在最后的标注处写上dp值（二维dp，即该处为止各颜色的最大长度）

res = defaultdict(int)

for y in g[x]:

cy = dfs(y)

if not cy: # 有环#如果是没处理，在dfs它的时候for这一步就不会进行，直接返回一个列表其中自己颜色为1，这也是res[colors[x]] += 1写在循环外的好处所在，此时not cy只有可能是0

return cy

for ch, c in cy.items():

res[ch] = max(res[ch], c)

res[colors[x]] += 1

memo[x] = res # 记忆化，同时也表示 x 计算完毕

return res

ans = 0

for x, c in enumerate(colors):

res = dfs(x)

if not res: # 有环

return -1

ans = max(ans, res[c])

return ans

## kmp

compute\_lps 函数用于计算模式字符串的LPS表。LPS表是一个数组，

其中的每个元素表示模式字符串中当前位置之前的子串的最长前缀后缀的长度。

该函数使用了两个指针 length 和 i，从模式字符串的第二个字符开始遍历。

"""

def compute\_lps(pattern):

"""

计算pattern字符串的最长前缀后缀（Longest Proper Prefix which is also Suffix）表

:param pattern: 模式字符串

:return: lps表

"""

m = len(pattern)

lps = [0] \* m # 初始化lps数组

length = 0 # 当前最长前后缀长度

for i in range(1, m): # 注意i从1开始，lps[0]永远是0

while length > 0 and pattern[i] != pattern[length]:

length = lps[length - 1] # 回退到上一个有效前后缀长度

if pattern[i] == pattern[length]:

length += 1

lps[i] = length

return lps

#lps的作用是在**ab**sdhjk**ab**中找到和后面的“ab”（后缀）相同的最长的前缀，相当于一个回溯

#pattern是目标串

def kmp\_search(text, pattern):

n = len(text)

m = len(pattern)

if m == 0:

return 0

lps = compute\_lps(pattern)

matches = []

# 在 text 中查找 pattern

j = 0 # 模式串指针

for i in range(n): # 主串指针

while j > 0 and text[i] != pattern[j]:

j = lps[j - 1] # 模式串回退

if text[i] == pattern[j]:

j += 1

if j == m:

matches.append(i - j + 1) # 匹配成功

j = lps[j - 1] # 查找下一个匹配

return matches