快捷代码库

武添彧 1700011412

版本号：v20210102

读取数据

读入一行以空格分割的3个数字，并赋值给abc

方法1：

```
a, b, c = map(int, input().split(' '))
```

方法2：先读入字符串再进行数组化，会保留原字符串

```
s = input()
a = [int(i) for i in s.split()]
```

如果读取string片段下表超出其范围，返回空字符串

```
a = 'abc'
a[3:]
# return ''
```

特殊形式的输出

迭代器输出空格分割，且最后没有空格

```
s = str(list[0])
for x in list[1:]:
    s+= ' '+str(x)
print(s)
```

先对list进行.sort排序后再变为集合，变回到list后仍然可能是乱序的

二维数组

二维数组的初始化（避免浅拷贝）

```
#原始为(n,m)的矩阵
matrix = []
for i in range(n):
    matrix.append([0] * m)
```

保护圈初始化

```
#原始为(n,m)的矩阵
board = []
wall = -1
#最上面多加一层
board.append([wall for x in range(n+2)])
for y in range(n):
    board.append([wall]+[ 0 for x in range(m)]+[wall])
#最后面多加一层
board.append([wall for x in range(n+2)])
```

二维数组读取： +保护圈（四边再加一层）

数组型二维数组

```
#原始为(n,m)的矩阵
board = []
#最上面多加一层
board.append([0 for x in range(m+2)])
for y in range(n):
    board.append([0]+[int(x) for x in input.split()]+[0])
#最后面多加一层
board.append([0 for x in range(m+2)])
```

字符串型二维数组

```
#原始为(n,m)的矩阵
board = []
wall = '#'
#最上面多加一层
board.append(wall*(m+2))
for y in range(n):
    board.append(wall+input()+wall)
#最后面多加一层
board.append(wall*(m+2))
```

二维数组的输出

```
for i in range(3):
    print(*matrix[i], sep=' ')
```

带保护圈的二维数组输出

```
for i in range(1,n+1):
    board[i] = board[i][1:-1]
    print(*board[i], sep=' ')
```

二维数组的复制（深拷贝）

- 浅拷贝：复制出来一个新的指针，指向同一个地址

```
# 以下都是浅拷贝的方式
b = a
b = a[:]
import copy
b = copy.copy(a)
```

- 深拷贝：新指针、新地址（不与原来值影响）

```
import copy
b = copy.deepcopy(a)
```

列表相关

按列表元素中指定的某个元素值进行排序

按照二维数组中第一个元素排序，从大到小排序！

```
students = [[3,'Jack',12],[2,'Rose',13],[1,'Tom',10],
[5,'Sam',12],[4,'Joy',8]]
students =sorted(students,key=(lambda x:x[0]))

>>[[1, 'Tom', 10], [2, 'Rose', 13], [3, 'Jack', 12], [4, 'Joy',
8], [5, 'Sam', 12]]
```

按照列表中第一个从小到大排序后，再按照第3个从大到小排序

```
students = [[3,'Jack',12],[2,'Rose',13],[1,'Tom',10],
[5,'Sam',12],[4,'Joy',8]]
students.sort(key=(lambda x: (x[0],-x[2])))

```

将列表输出为间隔为空格的形式

```
somelist = [1,2,3,4]
print(*somelist, sep=' ')
```

循环体部分重复任务超时的解决方法

1. 很简单，想办法提前运行一次将所有可能值存入表中，之后判断即可
2. 多次输出合并为一次

```
ans = []
for i in range(10):
    ans.append(i)
print('\n'.join(map(str,ans)))
```

整数的长度

```
len(str(integer))
```

列表中每个元素乘以相同值10

```
my_list = [i * 10 for i in my_list]
```

dp代码

质数的计算方法

```python
limit = 1000000
def calculate_prime_flag_for_each_number_upto_limit():
    prime_flag = [True] * limit
    prime_flag[0] = prime_flag[1] = False
    for i in range(2,limit):
        if prime_flag[i] == True:
            for j in range(i*i, limit, i):
                prime_flag[j] = False
    return prime_flag
```

牛顿迭代法

```python
# Newton's method
def solve(function = "x**3-5*x**2+10*x-80", a=4.0):
    def f(x):
        return eval(function)

    def df(x, dx):
        return (f(x + dx) - f(x)) / dx

    while abs(f(a)) > 1e-10:
        if df(a, 1e-10) == 0:
            a -= 1e-10
        else:
            a = a - f(a) / df(a, 1e-10)
    return a


ans = solve()
print("{:.9f}".format(ans))
```

自定义迭代器

```python
class Squares:
    def __init__(self, start, stop):  # 迭代起始、终止位
        self.value = start
        self.stop = stop

    def __iter__(self):     # 返回自身的迭代器
```

```
        return self

    def __next__(self):      # 返回下一个元素
        if self.value > self.stop:    # 结尾时抛出异常
            raise (StopIteration)
        item = self.value**2
        self.value += 1
        return item

if __name__ == "__main__":
    for i in Squares(1, 5):
        print(i, end=" ")

    s = Squares(1,5)
    print()
    print(9 in s)
```

运行结果

```
1 4 9 16 25
True
```

图的遍历

遍历框(邻接矩阵类型)

上下左右

```
neighbour = [[0, -1], [0, 1], [1, 0], [-1, 0]]
```

八格的

```
neighbour = [[0, -1], [0, 1], [1, 0], [-1, 0],[1,-1],[1,1],
[-1,-1],[-1,1]]
```

无向图

无向图bfs

字典列链表形式

默认输出的是节点的父节点

```
graph = {
    "A":["B", "C"],   # 与A相连的节点是B,C
    "B":["A", "C", "D"], # 与B相连的节点是A,C,D
    "C":["A", "B", "D", "E"],
    "D":["B", "C", "E", "F"],
    "E":["C", "D"],
    "F":["D"]
 }
# graph: dict type, s: start node code
def BFS(graph, s):
    queue = []  # 初始化一个空队列
    queue.append(s) # 将所有节点入队列
    seen = set()
    seen.add(s)
    parent = {s : None}

    while(len(queue) > 0):
        vertex = queue.pop(0)
        nodes = graph[vertex]
        for w in nodes:
            if w not in seen:
                queue.append(w)
                seen.add(w)
                parent[w] = vertex

    return parent


parent = BFS(graph, "E")
for key in parent:
    print(key, parent[key])
```

二维矩阵遍历

(注意邻接是4点邻接还是8点邻接)

以四格遍历为例，输出连成片的地区：r地区、b地区、#空

input

```
6
r##bb#
###b##
#r##b#
#r##b#
#r####
######
```

代码

```
black = 'b'
red = 'r'
empty = '#'
neighbour = [[0, -1], [0, 1], [1, 0], [-1, 0]]


def bfs(start, area, visited, color):
    r, c = start
    visited[r][c] = 1
    queue = []
    queue.append([r, c])
    while len(queue) != 0:
        x, y = queue.pop()
        for dx, dy in neighbour:
            temp_x = x + dx
            temp_y = y + dy

            if area[temp_x][temp_y] == color and visited[temp_x][temp_y] == 0:
                visited[temp_x][temp_y] = 1
                queue.append([temp_x, temp_y])
    return
# count areas
c_black = 0
```

```
c_red = 0
# 边长
n = int(input())
#初始化
area = []
visited =[]
area.append(empty*(n+2))
visited.append([0] * (n + 2))

for __ in range(n):
    area.append(empty+input()+empty)
    visited.append([0] * (n + 2))
area.append(empty*(n+2))
visited.append([0] * (n + 2))

# 遍历所有可能的起点
for row in range(1,n+1):
    for column in range(1,n+1):
        if area[row][column] != empty and visited[row][column] == 0:
            color = black
            if area[row][column] == black:
                c_black += 1
            else:
                color = red
                c_red += 1
            # bfs

 bfs([row,column],area=area,visited=visited,color=color)


print(c_red,c_black,sep=' ')

```

无向图**dfs**

字典列链表形式

默认输出的是遍历顺序

```
graph = {
    "A":["B", "C"],
    "B":["A", "C", "D"],
    "C":["A", "B", "D", "E"],
    "D":["B", "C", "E", "F"],
    "E":["C", "D"],
    "F":["D"]
 }
# graph: dict type, s: start node code
def DFS(graph, s):
    stack = []
    stack.append(s)
    seen = set()
    seen.add(s)
    while(len(stack) > 0):
        vertex = stack.pop()
        nodes = graph[vertex]
        for w in nodes:
            if w not in seen:
                stack.append(w)
                seen.add(w)
        print(vertex)

DFS(graph, "A")
```

二维矩阵遍历

(注意邻接是4点邻接还是8点邻接)

以四格遍历为例，输出连成片的地区：r地区、b地区、#空

input

```
6
r##bb#
###b##
#r##b#
#r##b#
#r####
######
```

代码

```

black = 'b'
red = 'r'
empty = '#'
neighbour = [[0, -1], [0, 1], [1, 0], [-1, 0]]


def dfs(start, area, visited, color):
    r, c = start
    visited[r][c] = 1
    stack = []
    stack.append([r, c, -1])
    while len(stack) != 0:
        x, y, previous_n = stack[-1]
        if visited[x][y] != 1:
            visited[x][y] = 1
        # judge if remove in the end
        remove_from_stack = True

        for dire in range(4):
            if dire != previous_n:
                dx, dy = neighbour[dire]
                temp_x = x + dx
                temp_y = y + dy
                # have children node, not remove now
                if area[temp_x][temp_y] == color and
visited[temp_x][temp_y] == 0:
                    remove_from_stack = False
                    parent = neighbour.index([-dx, -dy])
                    stack.append([temp_x, temp_y, parent])

        # decide if remove

```

```
        if remove_from_stack is True:
            stack.pop()

    return


# count areas
c_black = 0
c_red = 0
# 边长
n = int(input())
# 初始化
area = []
visited = []
area.append(empty * (n + 2))
visited.append([0] * (n + 2))

for __ in range(n):
    area.append(empty + input() + empty)
    visited.append([0] * (n + 2))
area.append(empty * (n + 2))
visited.append([0] * (n + 2))

# 遍历所有可能的起点
for row in range(1, n + 1):
    for column in range(1, n + 1):
        if area[row][column] != empty and visited[row][column]
== 0:
            color = black
            if area[row][column] == black:
                c_black += 1
            else:
                color = red
                c_red += 1
            # dfs
            dfs([row, column], area=area, visited=visited,
color=color)

print(c_red, c_black, sep=' ')
```

有向图

有向图bfs

- 字典列链表形式

  求最大连通子图的节点数为例

```python
## create graph 字典类型，有向图
graph = {}
ids = set()

for _ in range(int(input())):
    ln = input().split(':')
    u = int(ln[0].rstrip())
    ids.add(u)
    if u not in graph:
        neighbours = [int(_) for _ in ln[1].split()]
        graph[u] = neighbours
# bfs: graph(dict type),initial node code
def bfs(graph, initial):
    visited = []
    queue = [initial]

    while queue:
        node = queue.pop(0)
        if node not in visited:
            visited.append(node)

            if node not in graph:
                continue

            for nei in graph[node]:       # neighbour
                queue.append(nei)
    return visited

## formal code: bfs travel
maxp = 0
for i in ids:
    bfs_path = bfs(graph, i)
    if -1 in bfs_path:
        bfs_path.remove(-1)
    maxp = max(maxp, len(bfs_path))
```

```
    #print(len(bfs_path), bfs_path)
print(maxp)
```

有向图dfs

- 字典列链表形式

  求最大连通子图的节点数为例

  graph = {id: [connects_list]}

  input sample

```
sample1 in:
5                #结点数量
53 : -1          #结点标号，结点连接对象的标号（-1表示无连接）
118 : 119 136 137
92 : 107 93 102 91
102 : -1
130 : 66 132 135 103

sample1 out:
5
```

  非递归实现

```
# 读取数据：  字典类型，有向图
graph = {}
ids = set()

for _ in range(int(input())):
    ln = input().split(':')
    u = int(ln[0].rstrip())
    ids.add(u)
    if u not in graph:
        neighbours = [int(_) for _ in ln[1].split()]
        graph[u] = neighbours

def dfs_Non_recursion(graph, initial):
    visited = [initial]
    stack = [initial]
```

```
    while stack:
        node = stack[-1]
        if node not in visited:
            visited.append(node)

        if node not in graph:
            stack.pop()
            continue

        remove_from_stack = True
        for nei in graph[node]:          # neighbour
            if nei not in visited:
                stack.append(nei)
                remove_from_stack = False
                break
        if remove_from_stack:
            stack.pop()
    return visited


## dfs_Non_recursion travel
maxp = 0
for i in ids:
    dfs_path = dfs_Non_recursion(graph, i)
    if -1 in dfs_path:
        dfs_path.remove(-1)
    maxp = max(maxp, len(dfs_path))
    #print(len(bfs_path), bfs_path) #返回每次bfs路径长度，和具体过程
print(maxp) #返回最大路径长度
```

递归实现

```
# 读取数据：  字典类型，有向图
graph = {}
ids = set()

for _ in range(int(input())):
    ln = input().split(':')
    u = int(ln[0].rstrip())
    ids.add(u)
    if u not in graph:
        neighbours = [int(_) for _ in ln[1].split()]
```

```
        graph[u] = neighbours
# dfs graph(dict type),node(start node), visted list
def dfs(graph, node, visited):
    if node not in visited:
        visited.append(node)
        if node not in graph:
                return visited

        for nei in graph[node]:         # neighbour
            dfs(graph, nei, visited)
    return visited

## dfs travel
maxp = 0
for i in ids:
    dfs_path = dfs(graph, i, [])
    if -1 in dfs_path:
        dfs_path.remove(-1)
    maxp = max(maxp, len(dfs_path))
    #print(len(dfs_path), dfs_path) #返回每次bfs路径长度，和具体过程
print(maxp) #返回最大路径长度
```