

快捷代码库

武添或 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:

```
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

```
1 s = input()
2 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

```
1 a = 'abc'
2 a[3:]
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

```
1 s = str(list[0])
2 for x in list[1:]:
3     s+= ' '+str(x)
4 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）

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

保护圈初始化

```
1 #原始为(n,m)的矩阵
2 board = []
3 wall = -1
4 #最上面多加一层
5 board.append([wall for x in range(n+2)])
6 for y in range(n):
7     board.append([wall]+[ 0 for x in range(m)]+[wall])
8 #最后面多加一层
9 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

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

字符串型二维数组

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

二维数组的输出

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

带保护圈的二维数组输出

```
1  for i in range(1,n+1):
2      board[i] = board[i][1:-1]
3      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

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

- 深拷贝：新指针、新地址（不与原来值影响）

```
1 import copy
2 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！

```
1 students = [[3, 'Jack', 12], [2, 'Rose', 13], [1, 'Tom', 10],
2             [5, 'Sam', 12], [4, 'Joy', 8]]
3
4 >> sorted(students, key=(lambda x: x[0]))
5 >> [[1, 'Tom', 10], [2, 'Rose', 13], [3, 'Jack', 12], [4, 'Joy',
6             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

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

将列表输出为间隔为空格的形式

```
1 somelist = [1,2,3,4]
2 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

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

整数的长度

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

列表中每个元素乘以相同值10

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

dp代码

质数的计算方法

```

1 limit = 1000000
2 def calculate_prime_flag_for_each_number_upto_limit():
3     prime_flag = [True] * limit
4     prime_flag[0] = prime_flag[1] = False
5     for i in range(2, limit):
6         if prime_flag[i] == True:
7             for j in range(i*i, limit, i):
8                 prime_flag[j] = False
9     return prime_flag

```

牛顿迭代法

```

1 # Newton's method
2 def solve(function = "x**3-5*x**2+10*x-80", a=4.0):
3     def f(x):
4         return eval(function)
5
6     def df(x, dx):
7         return (f(x + dx) - f(x)) / dx
8
9     while abs(f(a)) > 1e-10:
10         if df(a, 1e-10) == 0:
11             a -= 1e-10
12         else:
13             a = a - f(a) / df(a, 1e-10)
14     return a
15
16
17 ans = solve()
18 print("{:.9f}".format(ans))

```

自定义迭代器

```

1 class Squares:
2     def __init__(self, start, stop): # 迭代起始、终止位
3         self.value = start
4         self.stop = stop
5
6     def __iter__(self): # 返回自身的迭代器

```

```

7         return self
8
9     def __next__(self):      # 返回下一个元素
10        if self.value > self.stop:  # 结尾时抛出异常
11            raise (StopIteration)
12        item = self.value**2
13        self.value += 1
14        return item
15
16 if __name__ == "__main__":
17     for i in Squares(1, 5):
18         print(i, end=" ")
19
20     s = Squares(1,5)
21     print()
22     print(9 in s)

```

运行结果

```

1  1 4 9 16 25
2  True

```

图的遍历

遍历框(邻接矩阵类型)

上下左右

```

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

```

八格的

```

1  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

```
1 graph = {
2     "A": ["B", "C"],    # 与A相连的节点是B,C
3     "B": ["A", "C", "D"], # 与B相连的节点是A,C,D
4     "C": ["A", "B", "D", "E"],
5     "D": ["B", "C", "E", "F"],
6     "E": ["C", "D"],
7     "F": ["D"]
8 }
9 # graph: dict type, s: start node code
10 def BFS(graph, s):
11     queue = []    # 初始化一个空队列
12     queue.append(s) # 将所有节点入队列
13     seen = set()
14     seen.add(s)
15     parent = {s : None}
16
17     while(len(queue) > 0):
18         vertex = queue.pop(0)
19         nodes = graph[vertex]
20         for w in nodes:
21             if w not in seen:
22                 queue.append(w)
23                 seen.add(w)
24                 parent[w] = vertex
25
26     return parent
27
28
29 parent = BFS(graph, "E")
30 for key in parent:
31     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

```
1 6
2 r##bb#
3 ###b##
4 #r##b#
5 #r##b#
6 #r####
7 #####
```

代码

```
1 black = 'b'
2 red = 'r'
3 empty = '#'
4 neighbour = [[0, -1], [0, 1], [1, 0], [-1, 0]]
5
6
7 def bfs(start, area, visited, color):
8     r, c = start
9     visited[r][c] = 1
10    queue = []
11    queue.append([r, c])
12    while len(queue) != 0:
13        x, y = queue.pop()
14        for dx, dy in neighbour:
15            temp_x = x + dx
16            temp_y = y + dy
17
18            if area[temp_x][temp_y] == color and visited[temp_x]
19            [temp_y] == 0:
20                visited[temp_x][temp_y] = 1
21                queue.append([temp_x, temp_y])
22    return
23 # count areas
24 c_black = 0
```

```

24 c_red = 0
25 # 边长
26 n = int(input())
27 #初始化
28 area = []
29 visited =[]
30 area.append(empty*(n+2))
31 visited.append([0] * (n + 2))
32
33 for __ in range(n):
34     area.append(empty+input()+empty)
35     visited.append([0] * (n + 2))
36 area.append(empty*(n+2))
37 visited.append([0] * (n + 2))
38
39 # 遍历所有可能的起点
40 for row in range(1,n+1):
41     for column in range(1,n+1):
42         if area[row][column] != empty and visited[row][column]
== 0:
43             color = black
44             if area[row][column] == black:
45                 c_black += 1
46             else:
47                 color = red
48                 c_red += 1
49             # bfs
50
51         bfs([row,column],area=area,visited=visited,color=color)
52
53 print(c_red,c_black,sep=' ')
54

```

无向图dfs

字典列链表形式

默认输出的是遍历顺序

```
1 graph = {
2     "A": ["B", "C"],
3     "B": ["A", "C", "D"],
4     "C": ["A", "B", "D", "E"],
5     "D": ["B", "C", "E", "F"],
6     "E": ["C", "D"],
7     "F": ["D"]
8 }
9 # graph: dict type, s: start node code
10 def DFS(graph, s):
11     stack = []
12     stack.append(s)
13     seen = set()
14     seen.add(s)
15     while(len(stack) > 0):
16         vertex = stack.pop()
17         nodes = graph[vertex]
18         for w in nodes:
19             if w not in seen:
20                 stack.append(w)
21                 seen.add(w)
22         print(vertex)
23
24 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

```
1 6
2 r##bb#
3 ###b##
4 #r##b#
5 #r##b#
6 #r####
7 #####
```

代码

```
1
2 black = 'b'
3 red = 'r'
4 empty = '#'
5 neighbour = [[0, -1], [0, 1], [1, 0], [-1, 0]]
6
7
8 def dfs(start, area, visited, color):
9     r, c = start
10    visited[r][c] = 1
11    stack = []
12    stack.append([r, c, -1])
13    while len(stack) != 0:
14        x, y, previous_n = stack[-1]
15        if visited[x][y] != 1:
16            visited[x][y] = 1
17            # judge if remove in the end
18            remove_from_stack = True
19
20        for dire in range(4):
21            if dire != previous_n:
22                dx, dy = neighbour[dire]
23                temp_x = x + dx
24                temp_y = y + dy
25                # have children node, not remove now
26                if area[temp_x][temp_y] == color and
visited[temp_x][temp_y] == 0:
27                    remove_from_stack = False
28                    parent = neighbour.index([-dx, -dy])
29                    stack.append([temp_x, temp_y, parent])
30
31        # decide if remove
```

```

32         if remove_from_stack is True:
33             stack.pop()
34
35     return
36
37
38 # count areas
39 c_black = 0
40 c_red = 0
41 # 边长
42 n = int(input())
43 # 初始化
44 area = []
45 visited = []
46 area.append(empty * (n + 2))
47 visited.append([0] * (n + 2))
48
49 for __ in range(n):
50     area.append(empty + input() + empty)
51     visited.append([0] * (n + 2))
52 area.append(empty * (n + 2))
53 visited.append([0] * (n + 2))
54
55 # 遍历所有可能的起点
56 for row in range(1, n + 1):
57     for column in range(1, n + 1):
58         if area[row][column] != empty and visited[row][column]
== 0:
59             color = black
60             if area[row][column] == black:
61                 c_black += 1
62             else:
63                 color = red
64                 c_red += 1
65             # dfs
66             dfs([row, column], area=area, visited=visited,
color=color)
67
68 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

```
1  ## create graph 字典类型，有向图
2  graph = {}
3  ids = set()
4
5  for _ in range(int(input())):
6      ln = input().split(':')
7      u = int(ln[0].rstrip())
8      ids.add(u)
9      if u not in graph:
10         neighbours = [int(_) for _ in ln[1].split()]
11         graph[u] = neighbours
12  # bfs: graph(dict type), initial node code
13  def bfs(graph, initial):
14      visited = []
15      queue = [initial]
16
17      while queue:
18         node = queue.pop(0)
19         if node not in visited:
20             visited.append(node)
21
22             if node not in graph:
23                 continue
24
25             for nei in graph[node]:      # neighbour
26                 queue.append(nei)
27      return visited
28
29  ## formal code: bfs travel
30  maxp = 0
31  for i in ids:
32      bfs_path = bfs(graph, i)
33      if -1 in bfs_path:
34          bfs_path.remove(-1)
35      maxp = max(maxp, len(bfs_path))
```

```

36     #print(len(bfs_path), bfs_path)
37     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

```

1  sample1 in:
2  5                #结点数量
3  53 : -1          #结点标号, 结点连接对象的标号 (-1表示无连接)
4  118 : 119 136 137
5  92 : 107 93 102 91
6  102 : -1
7  130 : 66 132 135 103
8
9  sample1 out:
10 5

```

非递归实现

```

1  # 读取数据: 字典类型, 有向图
2  graph = {}
3  ids = set()
4
5  for _ in range(int(input())):
6      ln = input().split(':')
7      u = int(ln[0].rstrip())
8      ids.add(u)
9      if u not in graph:
10         neighbours = [int(_) for _ in ln[1].split()]
11         graph[u] = neighbours
12
13  def dfs_Non_recursion(graph, initial):
14      visited = [initial]
15      stack = [initial]

```

```

16     while stack:
17         node = stack[-1]
18         if node not in visited:
19             visited.append(node)
20
21         if node not in graph:
22             stack.pop()
23             continue
24
25         remove_from_stack = True
26         for nei in graph[node]:           # neighbour
27             if nei not in visited:
28                 stack.append(nei)
29                 remove_from_stack = False
30                 break
31         if remove_from_stack:
32             stack.pop()
33     return visited
34
35
36 ## dfs_Non_recursion travel
37 maxp = 0
38 for i in ids:
39     dfs_path = dfs_Non_recursion(graph, i)
40     if -1 in dfs_path:
41         dfs_path.remove(-1)
42     maxp = max(maxp, len(dfs_path))
43     #print(len(bfs_path), bfs_path) #返回每次bfs路径长度, 和
具体过程
44 print(maxp) #返回最大路径长度

```

递归实现

```

1  # 读取数据: 字典类型, 有向图
2  graph = {}
3  ids = set()
4
5  for _ in range(int(input())):
6      ln = input().split(':')
7      u = int(ln[0].rstrip())
8      ids.add(u)
9      if u not in graph:
10         neighbours = [int(_) for _ in ln[1].split()]

```



```

11         graph[u] = neighbours
12 # dfs graph(dict type),node(start node), visted list
13 def dfs(graph, node, visited):
14     if node not in visited:
15         visited.append(node)
16         if node not in graph:
17             return visited
18
19         for nei in graph[node]:           # neighbour
20             dfs(graph, nei, visited)
21     return visited
22
23 ## dfs travel
24 maxp = 0
25 for i in ids:
26     dfs_path = dfs(graph, i, [])
27     if -1 in dfs_path:
28         dfs_path.remove(-1)
29     maxp = max(maxp, len(dfs_path))
30     #print(len(dfs_path), dfs_path) #返回每次dfs路径长度, 和
    具体过程
31 print(maxp) #返回最大路径长度

```