Part 1：葵花宝典

查看提示信息

```
>>help(func)
```

易错点及提示

1.`split()`使用时注意可能存在连续多个空格的情况

2.`RuntimeError`考虑使用`sys.setrecrusionlimit(layor)`避免爆栈

3.不要默认最大值初始是0，有可能所有数据都小于零，最后输出0导致WA

4.注意浮点数精度问题

5.候选人追踪$k=314159$等特殊情况（边界情况）

6.矩阵行数列数区分好

7.多组数据不能使用exit()

8.不能对空列表进行某些操作，如`min`，`max`等

9.考虑统一输出节省时间

10.多组测试数据中途`break`考虑忽略后续输入是否会引起下一组测试数据输入起点不对齐

11.注意输入数据可能的重复性

12.二分查找上下界设定要小心

13.第一行加`# pylint: skip-fle`可以忽略检查

14.标准答案一般不多于50行

快速读写

```
import sys
input = lambda: sys.stdin.readline().strip()
write = lambda x: sys.stdout.write(str(x))
```

位运算

符号	描述	运算规则
&	与	两个位都为1时，结果才为1
|	或	两个位都为0时，结果才为0
^	异或	两个位相同为0，相异为1
~	取反	0变1，1变0
<<	左移	各二进位全部左移若干位，高位丢弃，低位补0
>>	右移	各二进位全部右移若干位，对无符号数，高位补0，有符号数，各编译器处理方法不一样，有的补符号位（算术右移），有的补0（逻辑右移）

与运算的用途

清零

如果想将一个单元清零，即使其全部二进制位为0，只要与一个各位都为零的数值相与，结果为零。

取一个数的指定位

比如取数 `X=1010 1110` 的低4位，只需要另找一个数Y，令Y的低4位为1，其余位为0，即 `Y=0000 1111`，然后将X与Y进行按位与运算 `X&Y=0000 1110` 即可得到X的指定位。

判断奇偶

只要根据最未位是0还是1来决定，为0就是偶数，为1就是奇数。因此可以用 `if ((a & 1) == 0)` 来判断a是不是偶数。

小技巧

1.进制转换：2进制 `bin()` 输出0b...，八进制 `oct()` 输出0o...，十六进制 `hex()` 输出0x...，`int(str,base)` 可以把字符串按照指定进制转为十进制默认base=10

2.format：

```
print('{:.2f}'.format(num))
```

3.字符串匹配等

```
iterable.count(value)
str.find(sub)       #未找到抛出-1
list.index(x)       #未找到抛出ValueError
```

4.使用 `try` + `except` 判断错误类型，辅助处理RE问题

5.math库

```
math.pow(x, y) == x**y
math.factorial(n) == n!
```

6. `ord()` 把字符变为ASCII，`chr()` 把ASCII变为字符

7. `calendar.isleap(year)` 返回T/F判断闰年

8. `for a1, a2, ..., am in zip(b1, b2, ..., bn)` 旋转矩阵

9.排列组合

```
from itertools import permutations,combinations
permutations(list)          #生成list的全排列（每个以元组形式存在）
combinations(list,k)        #生成list的k元组合（无序）（每个以元组形式存在）
```

10. `replace()` 替换字符串中的指定字符，`eval()` 函数计算表达式的值

11. `print(*r)` 快速输出空格连接的列表/元组等

12.取整函数：`ceil` 向上取整（math库），`floor` 向下取整（math库），`round(num, n)` 四舍五入, 小数点后最终有n位

13. `enumerate` 快速获取索引和值：`for index, value in enumerate(list, start)`

14.补齐位数：`str.ljust(width, string)` 右侧补充 `string` 至 `str` 长度为 `width`，`str.rjust(width, string)` 左侧补充 `string` 至 `str` 长度为 `width`

优先队列

```
from queue import PriorityQueue

q = PriorityQueue()                 #创建PriorityQueue对象
q.put((priority number, data))      #存入数据，其中priority number越小代表优先级越大
q.empty()                           #判断优先队列是否为空
q.get()                             #弹出优先级最高的优先级和元素（以元组的形式）
q.qsize()                           #返回优先队列的大小
```

装饰器

```
from functools import lru_cache
```

二分查找

```
import bisect

index_left = bisect.bisect_left(list, vary)
#在list中查找vary的插入位置，使得插入后序列仍然保持有序,返回插入位置的索引;如果元素已存在，则返回最左边的位置
index3_right = bisect.bisect_right(list, num)
#在list中查找vary的插入位置，使得插入后序列仍然保持有序,返回插入位置的索引;如果元素已存在，则返回最右的位置
bisect.insort_left(list, vary)
#将vary的插入list，使得插入后序列仍然保持有序;如果元素已存在，则插入到最左边的位置
bisect.insort_right(list, num)
#将vary的插入list，使得插入后序列仍然保持有序;如果元素已存在，则插入到最右边的位置
```

defaultdict（默认值字典）

```
from collections import defaultdict

dic = defaultdict(key_type)              #初始化时须指定值的类型
dic = defaultdict(lambda: default_value)         #不使用默认初始值
```

deque（双端队列）

```
from collections import deque

d = deque()
d.append(1)          #在队尾添加元素
d.appendleft(0)      #在队头添加元素
d.pop()              #弹出队尾元素
d.popleft()          #弹出队头元素
len(d)               #打印队列长度
d.index(2)           #查找元素的索引

#使用迭代方式访问队列元素
for item in d:
    print(item)
```

Counter（计数字典）

```
from collections import Counter

c = Counter(list)           #返回计数字典
```

欧拉筛

```
lim = LIMIT
nums = {i: 1 for i in range(2, lim + 1)}
primes = []

for i in range(2, max_sqrt + 1):
    if nums[i]:
        primes.append(i)
    for j in primes:
        if i*j > lim:        #大于边界值时停止
            break
        nums[i*j] = 0
        if i % j == 0:       #保证每个数被最小的因数筛掉
            break
```

Part 2：排序

归并排序求逆序数（分治）

```
def merge_sort(i, j):
    if j <= i:
        return 0
    mid = (i + j) >> 1
    t = merge_sort(i, mid) + merge_sort(mid + 1, j)

    temp = ls[i: j + 1]
    mid -= i
    l, r = 0, mid + 1
    for idx in range(i, j + 1):
        if l > mid:
            ls[idx] = temp[r]
            r += 1
        elif r > j - i:
            ls[idx] = temp[l]
            l += 1
        elif temp[l] <= temp[r]:
            ls[idx] = temp[l]
            l += 1
        else:
            ls[idx] = temp[r]
            r += 1
            t += mid - l + 1

    return t
```

Part 3：队列和栈

中序转后序（调度场算法）

```
for _ in range(int(input())):
    s = input().strip()
    ans = []
    op = []
    operators = {'+': 1, '-': 1, '*': 2, '/': 2}
    dic = {i: True for i in '0123456789.'}

    idx = 0
    n = len(s)
    while idx < n:
        if s[idx] in dic:
            i = idx
            while i < n and s[i] in dic:
                i += 1
            ans.append(s[idx: i])
            idx = i - 1
        elif s[idx] == '(':
            op.append('(')
        elif s[idx] == ')':
            while op[-1] != '(':
                ans.append(op.pop())
            op.pop()
        else:
            while op and op[-1] != '(' and operators[op[-1]] >=
operators[s[idx]]:
                ans.append(op.pop())
            op.append(s[idx])
        idx += 1
    if op:
        op.reverse()
        print(f'{" ".join(ans)} {" ".join(op)}')
    else:
        print(" ".join(ans))
```

奶牛排队（单调栈）

```
stack = []
n = int(input())
h = [int(input()) for _ in range(n)]
left, right = [0]*n, [0]*n

for i in range(n):
    while stack and h[stack[-1]] >= h[i]:
        right[stack.pop()] = i
    stack.append(i)
while stack:
    right[stack.pop()] = n

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

```
    while stack and h[stack[-1]] <= h[i]:
        left[stack.pop()] = i
    stack.append(i)
while stack:
    left[stack.pop()] = -1

ans = 0
for i in range(n):
     if i - left[i] <= ans:
        continue
    for j in range(left[i] + 1, i):
        if right[j] > i:
            ans = max(ans, i - j + 1)
print(ans)
```

最大矩形面积问题（单调栈）

略。

出栈序列统计（卡特兰数）

```
from math import factorial
n = int(input())
print(factorial(2 * n) // (factorial(n))**2 // (n + 1))
```

卡特兰数$(Catalan\ number)$:

$$f(n) = \frac{C_{2n}^{n}}{n+1} \tag{1}$$

Part 4：堆

动态中位数

略。左堆扔右堆，右堆扔左堆。

滑动窗口最大值（懒删堆）

略。利用字典记录堆中真实存在的元素个数。

Part 5：树

树状数组

```
def lowbit(x):
    return x & -x


def query(x, y):            #查询[x, y]，索引从1开始
    x -= 1
    ans = 0
    while y > x:
        ans += tr[y]
```

```
        y -= lowbit(y)
    while x > y:
        ans -= tr[x]
        x -= lowbit(x)
    return ans


def add(i, k):                  #原数组第i个数加上k，更新树状数组，索引从1开始
    while i <= n:
        tr[i] += k
        i += lowbit(i)


tr = [0] * (n + 1)
for i in range(1, n+1):         #O(nlogn)建树
    add(i, ls[i - 1])
```

树的重量（类树状数组）

```
from math import log2


def father(x):
    return (x + 1) // 2 - 1


k, n = map(int, input().split())
ls = [[0, 0] for _ in range((1 << k) - 1)]
h = [int(log2(x + 1)) for x in range((1 << k) - 1)]

for _ in range(n):
    s = list(map(int, input().split()))
    x = s[1] - 1
    if len(s) == 3:
        y = s[2]
        ls[x][1] += y
        p = father(x)
        cnt = (1 << (k - h[x])) - 1
        while p >= 0:
            ls[p][0] += y * cnt
            p = father(p)
        continue

    p = x
    cnt = (1 << (k - h[x])) - 1
    ans = 0
    while p >= 0:
        ans += ls[p][1]*cnt
        p = father(p)
    print(ans + ls[x][0])
```

字典树（Trie）

```
class TrieNode:
    def __init__(self):
        self.child={}


class Trie:
    def __init__(self):
        self.root = TrieNode()

    def insert(self, nums):
        curnode = self.root
        for x in nums:
            if x not in curnode.child:
                curnode.child[x] = TrieNode()
            curnode=curnode.child[x]

    def search(self, num):
        curnode = self.root
        for x in num:
            if x not in curnode.child:
                return 0
            curnode = curnode.child[x]
        return 1
```

森林的带度数层次序列存储（根据层次遍历建树）

队列建树（即BFS）

表达式树与树的可视化

```
# 仅保留重点部分
length = 2**tree.h - 1
visualize = []
new_leaves = []
idx = 0
while leaves:
    visualize.append([' '] * length)
    visualize.append([' '] * length)
    for leaf in leaves:
        if leaf.l:
            leaf.x = (leaf.l.x + leaf.r.x) // 2
        else:
            leaf.x = idx
            idx += 2

        if leaf.v:
            visualize[-1][leaf.x] = leaf.v
        if leaf.l and leaf.l.v:
            visualize[-2][leaf.x - 1] = '/'
        if leaf.r and leaf.r.v:
            visualize[-2][leaf.x + 1] = '\\'
```

```
        if leaf.father and leaf.father.l == leaf:
            new_leaves.append(leaf.father)

    leaves = new_leaves[:]
    new_leaves = []

visualize.reverse()
visualize.pop()
```

分类并查集(种类有限已知，未知请考虑最小生成树/拓扑排序)

```
class Circle:
    def __init__(self, root):
        self.sexes = {root: True}
        self.root = root

    def join(self, circle, rotate):
        if rotate:
            for idx, sex in circle.sexes.items():
                self.sexes[idx] = not sex
                roots[idx] = self.root
            return

        for idx, sex in circle.sexes.items():
            self.sexes[idx] = sex
            roots[idx] = self.root
```

Part 5：图

深度优先搜索

A Knight's Journey（优化版DFS)

```
def dfs(x, y, ans):
    if len(ans) == 2*p*q:
        print(ans)
        return True

    for dx, dy in [(-1, -2), (1, -2), (-2, -1), (2, -1), (-2, 1), (2, 1), (-1,
2), (1, 2)]:
        nx, ny = x + dx, y + dy
        if 0 <= nx < p and 0 <= ny < q:
            key = chr(s + ny) + str(nx + 1)
            if key not in vis or not vis[key]:
                vis[key] = True             # 采用这种写法，省去复制vis数组带来的时间和
空间复杂度
                if dfs(nx, ny, ans + key):
                    return True
                vis[key] = False

    return False
```

棋盘分割（记忆化DFS）

```
def f(n, x1, y1, x2, y2):
    if (n, x1, y1, x2, y2) in dp:
        return dp[(n, x1, y1, x2, y2)]
    if n == 1:
        su = 0
        for i in range(x1, x2 + 1):
            for j in range(y1, y2 + 1):
                su += l[i][j]
        dp[(n, x1, y1, x2, y2)] = su*su
        return su*su

    mi = float('inf')
    for i in range(x1, x2):
        mi = min(mi, f(n - 1, x1, y1, i, y2) + f(1, i + 1, y1, x2, y2))
        mi = min(mi, f(1, x1, y1, i, y2) + f(n - 1, i + 1, y1, x2, y2))
    for i in range(y1, y2):
        mi = min(mi, f(n - 1, x1, y1, x2, i) + f(1, x1, i + 1, x2, y2))
        mi = min(mi, f(1, x1, y1, x2, i) + f(n - 1, x1, i + 1, x2, y2))
    dp[(n, x1, y1, x2, y2)] = mi
    return mi
```

骑士周游（$Warnsdorff$算法）

略。每一次选择的时候都选择可行子节点最少的子节点。

Sticks（DFS）

```
N = L = 0
used = []
length = []


def Dfs(R, M):
    if R==0 and M==0:
        return True
    if M==0:
        M = L
    for i in range(N):
        if used[i]==False and length[i] <= M:
            if i > 0:
                if used[i-1]==False and length[i]==length[i-1]:
                    continue # 不要在同一个位置多次尝试相同长度的木棒，剪枝1
            used[i] = True
            if (Dfs(R - 1, M - length[i])):
                return True
            else:
                used[i] = False
                # 不能仅仅通过替换最后一根木棒来达到目的，剪枝3
                # 替换第一个根棍子是没有用的，因为就算现在不用，也总会用到这根木棍，剪枝2
                if length[i]==M or M==L:
                    return False
    return False
```

```
while True:
    N = int(input())
    if N==0:
    break
    length = [int(x) for x in input().split()]
    length.sort(reverse = True) # 排序是为了从长到短拿木棒进行尝试
    totalLen = sum(length)
    for L in range(length[0], totalLen//2 + 1):
        if totalLen % L:
            continue # 不是木棒长度和的因子的长度，直接否定
        used = [False]*65
        if Dfs(N, L):
            print(L)
            break
    else:
        print(totalLen)
```

注：还有一种没有使用的剪枝是单调拼接剪枝

最小生成树

堆Prim

```
vertex = {i: False for i in range(n)}
vertex[0] = True
distance = [float('inf')]*n
distance[0] = [0]

def d(a, b):
    return sum([trucks[a][i] != trucks[b][i] for i in range(7)])

q = 0
heap = []
for i in range(1, n):
    t = d(0, i)
    distance[i] = t
    heap.append((t, i))
heapify(heap)

while heap:
    t, i = heappop(heap)
    if vertex[i]:
        continue

    vertex[i] = True
    q += t
    for j in range(n):
        if i == j or vertex[j]:
            continue

        if d(i, j) < distance[j]:
            distance[j] = d(i, j)
```

```
            heappush(heap, (distance[j], j))
```

Kruskal

将边由小到大加入图中，若不成环，则保留；反之，舍弃

拓扑排序

判断有向图是否成环

略

强连通图（Kosaraju/2 DFS）

```
def dfs1(graph, node, visited, stack):
    visited[node] = True
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs1(graph, neighbor, visited, stack)
    stack.append(node)

def dfs2(graph, node, visited, component):
    visited[node] = True
    component.append(node)
    for neighbor in graph[node]:
        if not visited[neighbor]:
            dfs2(graph, neighbor, visited, component)

def kosaraju(graph):
   # Step 1: Perform first DFS to get finishing times
   stack = []
   visited = [False] * len(graph)
   for node in range(len(graph)):
       if not visited[node]:
           dfs1(graph, node, visited, stack)

    # Step 2: Transpose the graph
    transposed_graph = [[] for _ in range(len(graph))]
    for node in range(len(graph)):
        for neighbor in graph[node]:
            transposed_graph[neighbor].append(node)

    # Step 3: Perform second DFS on the transposed graph to find SCCs
    visited = [False] * len(graph)
    sccs = []
    while stack:
        node = stack.pop()
        if not visited[node]:
            scc = []
            dfs2(transposed_graph, node, visited, scc)
            sccs.append(scc)
    return sccs
```

Part 6：算法类

KMP

前缀中的周期

```
n = int(input())
s = input()
ne = [0]*n
for i in range(1, n):            # 建next数组
    if ne[i - 1] == 0:
        ne[i] = (s[i] == s[0])
    else:
        idx = ne[i - 1]
        while True:
            if idx == 0 and s[0] != s[i]:
                ne[i] = 0
                break

            if s[i] == s[idx]:
                ne[i] = idx + 1
                break
            else:
                idx = ne[idx - 1]

    if (i + 1) % (i + 1 - ne[i]) == 0 and ne[i]:
        print(i + 1, (i + 1) // (i + 1 - ne[i]))
```

动态规划（DP）

最大子矩阵

```
n = int(input())
ls = []
count = 0
while len(ls) < n*n:
    ls += list(map(int, input().split()))
matrix = [[0]*n] + [[ls[j] for j in range(n)]] + [[] for _ in range(n - 1)]
for i in range(2, n + 1):
    for j in range(n):
        matrix[i].append(matrix[i - 1][j] + ls[(i - 1)*n + j])
ans = set()
for i in range(1, n + 1):
    for j in range(i, n + 1):
        dp = [matrix[j][0] - matrix[i - 1][0]]
        for k in range(1, n):
            dp.append(max(0, dp[-1]) + matrix[j][k] - matrix[i - 1][k])
        ans.add(max(dp))
print(max(ans))
```

接雨水

```
n = int(input())
h = list(map(int, input().split()))
right = v = index = 0
while index < n - 1:
    top = 0
    for i in range(index + 1, n):
        if h[i] > top: top, right = h[i], i
        if h[i] >= h[index]:
            top = h[index]
            break
    for i in range(index + 1, right):
        v += top - h[i]
    index = right
print(v)
```

定容背包问题

```
if dp[i - 1][j - w[i]] != -1:           #若容量j-w[i]存在定容分配，才可以更新dp[i][j]
    dp[i][j] = max(dp[i][j], v[i] + dp[i - 1][j - v[i]])
```

分组背包问题

```
for k in range(1, ts + 1):                  #循环每一组
    for i in range(m, -1, -1):              #循环背包容量
        for j in range(1, cnt[k] + 1):      #循环该组的每一个物品
            if i >= w[t[k][j]]:             #背包容量充足
                dp[i] = max(dp[i], dp[i - w[t[k][j]]] + c[t[k][j]])
```

背包问题最优解方案数

$f_{i,j}$为在只能放前i个物品的情况下，刚好装满容量为j的背包所能达到的最大总价值，$g_{i,j}$表示对应的方案数。如果$f_{i,j}=f_{i-1,j}$且$f_{i,j}\neq f_{i-1,j-w}+v$说明我们此时不选择把物品放入背包更优，方案数由$g_{i-1,j}$转移过来；如果$f_{i,j}\neq f_{i-1,j}$且$f_{i,j}=f_{i-1,j-w}+v$说明我们此时选择把物品放入背包更优，方案数由$g_{i-1,j-w}$转移过来；如果$f_{i,j}=f_{i-1,j}$且$f_{i,j}=f_{i-1,j-w}+v$说明放入或不放入都能取得最优解，方案数由$g_{i-1,j}$和$g_{i-1,j-w}$转移过来。

二进制分组

略。可以考虑采用二进制分组优化程序。

充实的寒假生活——区间背包问题

```
#注：区间不可重合，若可重合应使用递推DP（记忆化DFS）
s, e, v = [], [], []
ls = []
for _ in range(int(input())):
    si, ei, vi = map(str, input().split())
    ls.append((si, ei, vi))
ls.sort(key=lambda t: t[1])          #关键步骤：对区间右端点排序
for si, ei, vi in ls:
```

```
    s.append(si)
    e.append(ei)
    v.append(vi)
dp = [[0]*46 for _ in range(count)]
for i in range(46):
    if i >= e[0]:
        dp[0][i] = v[0]
for i in range(1, count):
    for j in range(1, 46):
        dp[i][j] = max(dp[i][j - 1], dp[i - 1][j])
        if j >= e[i]:
            dp[i][j] = max(dp[i][j], dp[i - 1][s[i] - 1] + v[i])
print(dp[-1][-1])
```

最长非增子序列

略

最大连续子序列和

```
dp = [0]*n
dp[0] = a[0]
for i in range(1, n):
    dp[i] = max(dp[i-1]+ls[i], ls[i])
print(max(dp))
```

最大上升子序列和

```
if ls[i] > ls[j]:
    dp[i] = max(dp[i], dp[j] + ls[i])
```

最长公共子序列

```
while True:
    try:
        x, y = input().split()
    except EOFError:
        break
    lx, ly = len(x), len(y)
    dp = [[0]*(ly + 1) for _ in range(lx + 1)]
    for i in range(1, lx + 1):
        for j in range(1, ly + 1):
            if x[i - 1] == y[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
            dp[i][j] = max([dp[i][j], dp[i - 1][j], dp[i][j - 1]])
    print(dp[-1][-1])
```

最长回文子序列问题：A的最长回文子序列长度等于A与reversed(A)的最长公共子序列长度

核电站

```
n, m = map(int, input().split())
a = [0] * (n + 1)
a[0] = 1
for i in range(1, n + 1):
    if i < m:
        a[i] = 2 * a[i - 1]
    elif i == m:
        a[i] = 2 * a[i - 1] - 1
    else:
        a[i] = 2 * a[i - 1] - a[i - 1 - m]
print(a[n])
```

贪心

Radar Installation

```
from math import sqrt

num = 1
while True:
    n, d = map(int, input().split())
    if n + d == 0:
        break
    x_y = {}
    for _ in range(n):
        x, y = map(int, input().split())
        if x in x_y:
            x_y[x] = max(x_y[x], y)
        else:
            x_y[x] = y
    n = len(x_y)
    count = 1
    x = list(sorted(x_y.keys()))
    try:
        ls = [x[i] + sqrt(d**2 - x_y[x[i]]**2) for i in range(n)]
        for i in range(2, n + 1):
            ls[-i] = min(ls[-i], ls[-i + 1])
        l = ls[0]
        for i in range(1, n):
            if (x[i] - l)**2 + x_y[x[i]]**2 - d**2 > 0.001:     # 覆盖不到时，安装雷
达
                l = ls[i]
                count += 1
    except ValueError:
        count = -1
    if d < 0:
        count = -1
    print(f'Case {num}: {count}')
    num += 1
    input()
```

田忌赛马

```
while True:
    n = int(input())
    if not n:
        break
    t = list(map(int, input().split()))
    k = list(map(int, input().split()))
    t.sort()
    k.sort()
    k_j = t_j = n - 1
    ans = k_k = t_k = 0
    for i in range(n):
        if k[k_j] < t[t_j]:
            k_j -= 1
            t_j -= 1
            ans += 1
        elif k[k_j] > t[t_j]:
            ans -= 1
            t_k += 1
            k_j -= 1
        else:
            if k[k_k] < t[t_k]:
                k_k += 1
                t_k += 1
                ans += 1
            else:
                ans -= (k[k_j] > t[t_k])
                k_j -= 1
                t_k += 1
    print(ans*200)
```

Jumping Cows

```
n = int(input())
a = [int(input()) for _ in range(n)]
ans = 0
b = True
for i in range(n - 1):
    if b:
        if a[i] > a[i + 1]:
            ans += a[i]
            b = False
    else:
        if a[i] < a[i + 1]:
            ans -= a[i]
            b = True
if b:
    ans += a[-1]
print(ans)
```

桶

4 Values whose Sum is 0

略。枚举c和d时不存储，直接计算，节省内存。

Part 7：正则表达式

限定符：

`a?` a出现0/1次	`a+` a出现一次以上	`a{2, }` a出现2次以上
`a*` a可以出现0/多次****	`a{6}` a出现6次	`a{2, 6}` a出现2-6次

匹配多个字符：

`(ab)+` ab出现一次以上

或运算：

`a (cat|dog)` 匹配 a cat or a dog　　`a cat|dog` 匹配 a cat or dog

字符类：

`[abc]+` a/b/c出现一次以上 abc aabbcc	`[a-zA-Z0-9]` ABCabc123	`\[^0-9]` 匹配0-9之外的数据(包括换行符)

元字符：

`\d` 数字字符 `\d+` 匹配一个以上的数字	`\s` 空白符（包含空格和换行符）	`.` 任意字符（不包含换行符）
`\D` 非数字字符	`\S` 非空白字符	`\.` 表示. 通过\进行了转义
`\w` 单词字符（单词，数字，下划线，即英文字符）	`\b` 单词的边界	`^` 匹配行首
`\W` 非单词字符	`\B` 非单词的边界	`$` 匹配行尾

贪婪匹配/懒惰匹配：

`<.+>` 贪婪匹配　　`<.+?>` ？设置为懒惰匹配

模块引入：

```
from re import match
print('YES' if match(pattern, string) else 'NO')
```