python cheatsheet

位运算

1. **按位与（AND）`&`**：
   - 对两个数值的每一位执行逻辑与操作。如果两个相应的二进制位都为1，则结果对应的位也为1。

```
a = 60 # 二进制: 0011 1100
b = 13 # 二进制: 0000 1101
print(a & b) # 输出: 12 (即二进制: 0000 1100)
```

2. **按位或（OR）`|`**：
   - 对两个数值的每一位执行逻辑或操作。只要两个相应的二进制位有一个为1，结果对应的位就为1。

```
print(a | b) # 输出: 61 (即二进制: 0011 1101)
```

3. **按位异或（XOR）`^`**： $a^{$1$对应反转0/1，a}0$
   - 对两个数值的每一位执行逻辑异或操作。如果两个相应的二进制位不同，则结果对应的位为1。

```
print(a ^ b) # 输出: 49 (即二进制: 0011 0001)
#a^1对应反转0/1
#a^0对应不变
```

4. **按位取反（NOT）`~`**：
   - 对数值的每一位执行逻辑非操作。它会反转所有位，包括符号位。由于Python中的整数是补码形式表示，这实际上会导致值变为负数并减1。

```
print(~a) # 输出: -61 (二进制: ...1100 0011)
```

5. **左移（Left shift）`<<`**：
   - 将一个数的二进制位向左移动指定的位数，右边用0填充。相当于乘以2的幂次。

```
print(a << 2) # 输出: 240 (即二进制: 1111 0000)
```

6. **右移（Right shift）`>>`**：
   - 将一个数的二进制位向右移动指定的位数，左边用符号位填充（对于正数用0填充，负数用1填充）。相当于除以2的幂次。

```
print(a >> 2) # 输出: 15 (即二进制: 0000 1111)
```

应用场景

- **掩码操作**：用于控制某些位的开启或关闭。
- **加密算法**：很多加密算法依赖于位运算来实现数据的安全转换。
- **低层编程**：如操作系统开发、硬件驱动程序编写等。

列表

修改列表

1. **添加元素**：
   - `append()`

     ：在列表末尾添加一个元素。

     ```
     my_list.append(6)
     print(my_list)  # 输出: [1, 2, 3, 4, 5, 6]
     ```

   - `insert()`

     ：在指定位置插入一个元素。

     ```
     my_list.insert(2, 10)
     print(my_list)  # 输出: [1, 2, 10, 3, 4, 5, 6]
     ```

   - `extend()`

     ：扩展列表，将另一个列表的元素添加到当前列表。

     ```
     my_list.extend([7, 8, 9])
     print(my_list)  # 输出: [1, 2, 10, 3, 4, 5, 6, 7, 8, 9]
     ```

2. **删除元素**：
   - `remove()`

     ：删除第一个匹配的元素。

     ```
     my_list.remove(10)
     print(my_list)  # 输出: [1, 2, 3, 4, 5, 6, 7, 8, 9]
     ```

   - `pop()`

     ：删除并返回指定位置的元素（默认是最后一个元素）。

     ```
     last_element = my_list.pop()
     print(last_element)  # 输出: 9
     print(my_list)       # 输出: [1, 2, 3, 4, 5, 6, 7, 8]
     ```

   - `del`

：删除指定位置的元素或切片。

```
解释del my_list[2]
print(my_list)  # 输出: [1, 2, 4, 5, 6, 7, 8]
del my_list[3:5]
print(my_list)  # 输出: [1, 2, 4, 7, 8]
```

3. **修改元素**：

```
my_list[0] = 0
print(my_list)  # 输出: [0, 2, 4, 7, 8]
```

列表方法

1. `index()`：返回第一个匹配元素的索引。

```
index = my_list.index(4)
print(index)  # 输出: 2
```

2. `count()`：返回元素在列表中出现的次数。

```
count = my_list.count(4)
print(count)  # 输出: 1
```

3. `sort()`：对列表进行排序。

```
my_list.sort()
print(my_list)  # 输出: [0, 2, 4, 7, 8]
```

4. `reverse()`：反转列表。

```
my_list.reverse()
print(my_list)  # 输出: [8, 7, 4, 2, 0]
```

5. `copy()`：创建列表的一个浅拷贝。

```
copied_list = my_list.copy()
print(copied_list)  # 输出: [8, 7, 4, 2, 0]
```

列表推导式

```
squares = [x**2 for x in range(10)]
print(squares)  # 输出: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

常见操作

1. **检查元素是否存在**：

```
if 4 in my_list:
    print("4 在列表中")
```

2. **遍历列表**：

```
for item in my_list:
    print(item)
```

3. **列表长度**：

```
length = len(my_list)
print(length)  # 输出: 5
```

4. **列表连接**：

```
解释list1 = [1, 2, 3]
list2 = [4, 5, 6]
combined_list = list1 + list2
print(combined_list)  # 输出: [1, 2, 3, 4, 5, 6]
```

列表切片

```
new_list = original_list[start:stop:step]
```

这里的参数说明如下：

- `start`：切片开始的位置索引（包含该位置），默认为0。
- `stop`：切片结束的位置索引（不包含该位置），默认为列表长度。
- `step`：步长，默认为1。如果设置为负数，则可以倒序获取列表中的元素。

列表排序

使用 `sorted()`（返回新列表）：

```
sorted_data = sorted(data, key=lambda x: x[1])
print(sorted_data)
# 输出: [(4, 1), (3, 2), (1, 5), (2, 8)]
```

使用 `list.sort()`（原地修改原列表）：

```
data.sort(key=lambda x: x[1])
print(data)
# 输出: [(4, 1), (3, 2), (1, 5), (2, 8)]
```

字典

创建字典:

- 使用花括号 `{}` 和 `:` 分隔键和值。
- 使用 `dict()` 构造函数。

```
person = {'name': 'Alice', 'age': 25, 'city': 'Beijing'}
```

访问字典中的值:

- 使用方括号 `[]` 和键来获取对应的值。
- 使用 `.get()` 方法，可以设置默认值。

```
name = person['name']  # 获取值
# 如果键不存在，返回 None 或者指定的默认值
address = person.get('address', 'Not provided')
```

删除元素:

- 使用 `del` 语句。
- 使用 `.pop()` 方法，可以指定一个默认值以防键不存在时引发错误。
- 使用 `.popitem()` 移除并返回一个任意的 (key, value) 对。

```
del person['city']  # 删除键为 'city' 的项
age = person.pop('age', 'No age specified')  # 删除并返回值
key, value = person.popitem()  # 删除并返回任意一项
```

遍历字典:

- 使用 `for` 循环遍历所有键。
- 使用 `.values()` 遍历所有值。
- 使用 `.items()` 遍历所有键值对。

```
for key in person:
    print(key)
for value in person.values():
    print(value)
for key, value in person.items():
    print(f'{key}: {value}')
```

检查键是否存在:

- 使用 `in` 关键字。

```
if 'name' in person:
    print("Name is set")
```

合并字典:

- Python 3.9+ 可以使用 `|` 和 `|=`, 在更早版本中可以使用 `.update()` 方法。

```
new_info = {'hobby': 'reading'}
combined_dict = person | new_info  # 创建新字典
person |= new_info  # 更新原字典
```

字典推导式:

- 类似列表推导式，可以从已有字典或其他可迭代对象创建新的字典。

```
squares = {x: x*x for x in range(6)}
```

其他方法:

- `.clear()` 清空字典。
- `.copy()` 返回字典的浅复制。
- `.fromkeys(seq[, value])` 创建一个新字典，其中键来自序列 `seq`，所有键都指向同一个值 `value`，如果提供了的话，默认为 `None`。
- `.setdefault(key[, default])` 如果键存在则返回其值，否则插入该键并将值设为默认值并返回。

集合

创建集合

1. **使用花括号 `{}`**:

   ```
   s = {1, 2, 3, 4}
   ```

2. **使用 `set()` 构造函数**:

   ```
   s = set([1, 2, 3, 4])
   ```

添加元素

- `add(element)`：添加一个元素到集合中。

```
s = {1, 2, 3}
s.add(4)
print(s)  # 输出: {1, 2, 3, 4}
```

- `update(iterable)`：添加多个元素到集合中。

```
s = {1, 2, 3}
s.update([4, 5, 6])
print(s)  # 输出: {1, 2, 3, 4, 5, 6}
```

删除元素

- `remove(element)`：删除指定元素，如果元素不存在会引发 `KeyError`。

```
s = {1, 2, 3}
s.remove(2)
print(s)  # 输出: {1, 3}
```

- `discard(element)`：删除指定元素，如果元素不存在不会引发错误。

```
解释s = {1, 2, 3}
s.discard(2)
print(s)  # 输出: {1, 3}
s.discard(4)  # 不会引发错误
```

- `pop()`：随机删除一个元素并返回它，如果集合为空会引发 `KeyError`。

```
解释s = {1, 2, 3}
element = s.pop()
print(element)  # 输出可能是 1, 2 或 3
print(s)  # 输出可能是 {2, 3}, {1, 3} 或 {1, 2}
```

- `clear()`：清空集合。

```
s = {1, 2, 3}
s.clear()
print(s)  # 输出: set()
```

集合运算

- **并集 (`union`)**：

```
解释s1 = {1, 2, 3}
s2 = {3, 4, 5}
s3 = s1.union(s2)
print(s3)  # 输出: {1, 2, 3, 4, 5}
```

- **交集 (`intersection`)**：

```
解释s1 = {1, 2, 3}
s2 = {3, 4, 5}
s3 = s1.intersection(s2)
print(s3)  # 输出: {3}
```

- **差集 (`difference`)**:

```
解释s1 = {1, 2, 3}
s2 = {3, 4, 5}
s3 = s1.difference(s2)
print(s3)  # 输出: {1, 2}
```

- **对称差集 (`symmetric_difference`)**:

```
解释s1 = {1, 2, 3}
s2 = {3, 4, 5}
s3 = s1.symmetric_difference(s2)
print(s3)  # 输出: {1, 2, 4, 5}
```

检查元素是否存在

- `in` 关键字:

```
s = {1, 2, 3}
print(2 in s)  # 输出: True
print(4 in s)  # 输出: False
```

遍历集合

- 使用 `for` 循环:

```
s = {1, 2, 3}
for element in s:
    print(element)
```

示例：去除列表中的重复元素

```
lst = [1, 2, 2, 3, 4, 4, 5]
unique_lst = list(set(lst))
print(unique_lst)  # 输出: [1, 2, 3, 4, 5]
```

十进制转其他进制

Python 提供了几个内置的函数可以直接将十进制数转换为二进制、八进制或十六进制。

- **十进制转二进制**: `bin()`
- **十进制转八进制**: `oct()`
- **十进制转十六进制**: `hex()`

这些函数返回的字符串前缀分别是 `'0b'`（二进制）、`'0o'`（八进制）和 `'0x'`（十六进制），表示数字的进制类型。

```
decimal_number = 255
print(bin(decimal_number))  # 输出: 0b11111111
```

树

普通树

```
class Tree:
    def __init__(self,val):
        self.val = val
        self.children = []

    def insert(self,val):
        self.children.append(val)

    def height1(self): #递归求树的高度
        if not self.children: 注意class内的函数不能以None为自变量
            return 0
        h = 0
        for child in self.children:
            h = max(h,child.height1())
        return h+1
```

二叉树

```
class BinaryTree:
    def __init__(self,val):
        self.val = val
        self.left = None
        self.right = None


    def height2(self):
        if not self.left and not self.right:
            return 0
        hleft = self.left.height2() if self.left else 0
        hright = self.right.height2() if self.right else 0
        h = max(hleft,hright)
        return h + 1
```

二叉树的遍历

前序遍历（Preorder Traversal）

根节点 -> 左子树 -> 右子树

```
def preorder(root):
    if not root:
        return ''
    res = root.val
    res += preorder(root.left)
    res += preorder(root.right)
    return res
```

中序遍历（Inorder Traversal）

左子树 -> 根节点 -> 右子树

```
def inorder(root):
    if not root:
        return ''
    res = inorder(root.left)
    res += root.val
    res += inorder(root.right)
    return res
```

后序遍历（Postorder Traversal）

左子树 -> 右子树 -> 根节点

```
def postorder(root):
    if not root:
        return ''
    res = postorder(root.left)
    res += postorder(root.right)
    res += root.val
    return res
```

图

生成图

```
P = int(input())
locations = []
name_to_index = {}
for i in range(P):
    name = input().strip()
    locations.append(name)
```

```
    name_to_index[name] = i

Q = int(input())
adj = [[] for _ in range(P)]
for _ in range(Q):
    parts = input().split()
    u_name, v_name, distance = parts[0], parts[1], int(parts[2])
    u = name_to_index[u_name]
    v = name_to_index[v_name]
    adj[u].append((v, distance))
    adj[v].append((u, distance))
```

图的dijkstra算法

```
dist = [float('inf')] * P
    dist[start] = 0
    prev = [-1] * P #用来储存访问的路径
    heap = []
    heapq.heappush(heap, (0, start))

    while heap:
        current_dist, u = heapq.heappop(heap)
        if u == end:
            break
        if current_dist > dist[u]:
            continue
        for (v, distance) in adj[u]:
            if dist[v] > dist[u] + distance:
                dist[v] = dist[u] + distance
                prev[v] = u #记录最优路径对应的路径
                heapq.heappush(heap, (dist[v], v))

    #生成最优路径
    path = []
    current = end
    while current != -1:
        path.append(current)
        current = prev[current]
    path.reverse()
```

拓扑排序

```
from collections import deque

def topo_sort(n,graph,in_degree):
    new_degree = in_degree[:]
    q = deque()
    result = ''
```

```
    for i in range(n):
        if new_degree[i] == 0:
            q.append(i)
    multiple_choice = False

    while q:
        if len(q) > 1:
            multiple_choice = True
        u = q.popleft()
        result += chr(ord('A') + u)
        for v in graph[u]:
            new_degree[v] -= 1
            if new_degree[v] == 0:
                q.append(v)
    if len(result) < n:
        return "INCONSISTENT", ''
    elif multiple_choice:
        return "UNCERTAIN", ''
    else:
        return "DETERMINED", result

while True:
    n,m = map(int,input().split())
    if n ==m==0:
        break
    graph = [[] for _ in range(n)]
    in_degree = [0 for _ in range(n)]
    for i in range(m):
        inp = input()
        a = ord(inp[0]) - ord('A')
        b = ord(inp[2]) - ord('A')
        graph[a].append(b)
        in_degree[b] += 1

        status,ans = topo_sort(n,graph,in_degree)
        if status == "INCONSISTENT":
            print(f"Inconsistency found after {i + 1} relations.")
            # 丢弃剩余输入
            for _ in range(i + 1, m):
                input()
            break
        elif status == "DETERMINED":
            print(f"Sorted sequence determined after {i + 1} relations: {ans}.")
            # 丢弃剩余输入
            for _ in range(i + 1, m):
                input()
            break
    else:
        print("Sorted sequence cannot be determined.")
```

其他进制转十进制

对于从其他进制转回十进制，你可以使用 `int()` 函数，并指定第二个参数为进制基数。

```
binary_number = "11111111"
octal_number = "377"
hexadecimal_number = "ff"
print(int(binary_number, 2))    # 输出: 255
print(int(octal_number, 8))     # 输出: 255
print(int(hexadecimal_number, 16))  # 输出: 255
```

如何检验一个数据的类型

使用 `isinstance` 函数

`isinstance` 函数检查一个对象是否是指定的类型或其子类。它可以接受多个类型参数。

```
x = 10
print(isinstance(x, int))  # True
y = 10.5
print(isinstance(y, float))  # True
z = "hello"
print(isinstance(z, str))  # True
a = [1, 2, 3]
print(isinstance(a, list))  # True
b = (1, 2, 3)
print(isinstance(b, tuple))  # True
c = {1, 2, 3}
print(isinstance(c, set))  # True
d = {"a": 1, "b": 2}
print(isinstance(d, dict))  # True
# 检查多个类型
print(isinstance(10, (int, float)))  # True
print(isinstance(10.5, (int, float)))  # True
print(isinstance("hello", (int, float)))  # False
```

处理浮点数

保留小数点后x位

使用 `str.format()`

```
number = 3.14159
formatted_number = "{:.2f}".format(number)
print(formatted_number)  # 输出: 3.14
```

使用 f-string

```
number = 3.14159
formatted_number = f"{number:.2f}"
print(formatted_number)  # 输出: 3.14
```

保留x位有效数字

在 Python 中，你可以使用字符串格式化来控制浮点数的显示：

```
num = 123.456
formatted_num = f"{num:.1g}"  # 使用g格式，自动选择固定小数点或科学计数法
print(formatted_num)  # 输出: 1e2
```

或者使用 `format()` 函数：

```
num = 123.456
formatted_num = format(num, ".1g")
print(formatted_num)  # 输出: 1e2
```

以上示例为保留一位有效数字。

Python 中的 Unicode

在 Python 中，字符串默认使用 Unicode 编码。你可以使用 `ord()` 获取字符的编码和 `chr()` 获取编码对应字符 。

正则表达式

基本概念

1. **字符类**：
   - `.`：匹配任意单个字符（除了换行符）。
   - `[abc]`：匹配方括号内的任意一个字符。
   - `[^abc]`：匹配不在方括号内的任意一个字符。
   - `[a-z]`：匹配小写字母 a 到 z 之间的任意一个字符。
   - `[A-Z]`：匹配大写字母 A 到 Z 之间的任意一个字符。
   - `[0-9]`：匹配数字 0 到 9 之间的任意一个字符。
   - `\d`：匹配任意一个数字，等同于 `[0-9]`。
   - `\D`：匹配任意一个非数字，等同于 `[^0-9]`。
   - `\w`：匹配任意一个字母、数字或下划线，等同于 `[a-zA-Z0-9_]`。
   - `\W`：匹配任意一个非字母、数字或下划线，等同于 `[^a-zA-Z0-9_]`。
   - `\s`：匹配任意一个空白字符（包括空格、制表符、换行符等）。
   - `\S`：匹配任意一个非空白字符。
2. **量词**：
   - `*`：匹配前面的字符零次或多次。
   - `+`：匹配前面的字符一次或多次。
   - `?`：匹配前面的字符零次或一次。
   - `{n}`：匹配前面的字符恰好 n 次。

- `{n,}`：匹配前面的字符至少 n 次。
- `{n,m}`：匹配前面的字符至少 n 次，至多 m 次。

3. **锚点**：
   - `^`：匹配字符串的开头。
   - `$`：匹配字符串的结尾。
   - `\b`：匹配单词边界。
   - `\B`：匹配非单词边界。
4. **分组和引用**：
   - `()`：分组，可以用于提取匹配的子串或应用量词。
   - `|`：表示“或”，用于匹配多个选项之一。
   - `\1`, `\2`, ...：引用前面的分组。

正则表达式模块 `re` 的常用函数

1. `re.match(pattern, string)`：
   - 从字符串的开头开始匹配，如果匹配成功返回一个匹配对象，否则返回 `None`。
2. `re.search(pattern, string)`：
   - 在字符串中搜索第一个匹配的子串，如果匹配成功返回一个匹配对象，否则返回 `None`。
3. `re.findall(pattern, string)`：
   - 返回字符串中所有与模式匹配的子串，作为一个列表。
4. `re.sub(pattern, repl, string)`：
   - 替换字符串中所有与模式匹配的子串，返回替换后的字符串。
5. `re.split(pattern, string)`：
   - 根据模式分割字符串，返回一个列表。
6. `re.fullmatch(pattern, string)`：
   - 检查整个字符串是否完全匹配模式，如果匹配成功返回一个匹配对象，否则返回 `None`。

不断输入

当你在 `while True:` 循环中使用 `try:` 块，并且特别关注 `EOFError` 异常时，你可以捕获并处理这种特定的异常。`EOFError` 通常在程序尝试从标准输入读取数据但到达文件末尾（EOF）时引发。

```
while True:
    try:

    except EOFError:
        break
```

一次性读取

`sys.stdin.read()` 是 Python 中 `sys.stdin` 对象的一个方法，它用于读取来自标准输入流的所有数据，直到输入结束。这个方法会返回一个字符串，其中包含了从标准输入读取的所有内容。

```
import sys
print("请输入数据，按 Ctrl+D 结束输入：")
input_data = sys.stdin.read()
```

可以直接用`sys.stdin.read().split()`将输入的数据按行分割

用`ctrl+D`实现输入完毕的效果

深拷贝

在`copy`模块中，实现深拷贝的方法：

```
original_list = [[1, 2, 3], [4, 5, 6]]
copied_list = copy.deepcopy(original_list)
```

collection模块

2. `deque`

- **用途**：双端队列，支持从两端高效地添加或删除元素。

```
from collections import deque
d = deque([1, 2, 3])
d.append(4)
d.appendleft(0)
d.popleft()
print(d)  # 输出: deque([0, 1, 2, 3, 4])
```

3. `Counter`

- **用途**：计数器，用于统计可哈希对象的频率。

```
from collections import Counter
c = Counter(['a', 'b', 'c', 'a', 'b', 'b'])
print(c)  # 输出: Counter({'b': 3, 'a': 2, 'c': 1})
```

4. `defaultdict`

- **用途**：字典的子类，提供了一个默认值工厂函数，当访问不存在的键时自动调用该函数。

```
from collections import defaultdict
d = defaultdict(int)
d['a'] += 1
d['b'] += 1
print(d)  # 输出: defaultdict(<class 'int'>, {'a': 1, 'b': 1})
```

5. `OrderedDict`

- **用途**：有序字典，保持插入顺序。

```
from collections import OrderedDict
od = OrderedDict()
od['a'] = 1
od['b'] = 2
od['c'] = 3
print(od)  # 输出: OrderedDict([('a', 1), ('b', 2), ('c', 3)])
```

无穷大

```
float('inf')
```

math模块

常见的数学常量

`math` 模块提供了一些常用的数学常量：

- `math.pi`：圆周率 π，约等于 3.141592653589793
- `math.e`：自然对数的底 e，约等于 2.718281828459045
- `math.tau`：2π，约等于 6.283185307179586
- `math.inf`：正无穷大
- `math.nan`：非数字（Not a Number）

常见的数学函数

1. 基本数学函数

- `math.ceil(x)`：返回大于或等于 x 的最小整数。
- `math.floor(x)`：返回小于或等于 x 的最大整数。
- `math.trunc(x)`：返回 x 的整数部分，去掉小数部分。
- `math.fabs(x)`：返回 x 的绝对值。
- `math.pow(x, y)`：返回 x 的 y 次幂。
- `math.sqrt(x)`：返回 x 的平方根。
- `math.isclose(a, b, rel_tol=1e-09, abs_tol=0.0)`：判断两个数是否接近。

2. 三角函数

- `math.sin(x)`：返回 x 的正弦值（x 以弧度为单位）。
- `math.cos(x)`：返回 x 的余弦值（x 以弧度为单位）。
- `math.tan(x)`：返回 x 的正切值（x 以弧度为单位）。
- `math.asin(x)`：返回 x 的反正弦值（结果以弧度为单位）。

- `math.acos(x)`：返回 x 的反余弦值（结果以弧度为单位）。
- `math.atan(x)`：返回 x 的反正切值（结果以弧度为单位）。
- `math.atan2(y, x)`：返回 y/x 的反正切值（结果以弧度为单位）。

3. 对数函数

- `math.log(x[, base])`：返回 x 的自然对数，如果指定了 base，则返回以 base 为底的对数。
- `math.log10(x)`：返回 x 的以 10 为底的对数。
- `math.log2(x)`：返回 x 的以 2 为底的对数。

4. 角度转换

- `math.degrees(x)`：将弧度 x 转换为角度。
- `math.radians(x)`：将角度 x 转换为弧度。

5. 其他函数

- `math.gcd(a, b)`：返回 a 和 b 的最大公约数。
- `math.lcm(a, b)`：返回 a 和 b 的最小公倍数。
- `math.factorial(x)`：返回 x 的阶乘。
- `math.isfinite(x)`：如果 x 是有限的（既不是无穷大也不是 NaN），则返回 True。
- `math.isnan(x)`：如果 x 是 NaN，则返回 True。
- `math.modf(x)`：返回 x 的小数部分和整数部分。

最小堆方法

```
import heapq
```

创建堆

你可以使用列表来表示堆，并通过 `heapq.heapify()` 函数将其转换为堆：

```
data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
heapq.heapify(data)
print(data)  # 输出: [0, 1, 2, 3, 9, 5, 4, 7, 8, 6] (最小堆)
```

```
import heapq
heapq.heappush(data, item)#推入元素
heapq.heappop(data)#弹出堆顶
heapq.heappushpop(data,item)#先推再弹,更高效
heapq.heapreplace(data,item)#先弹再推(SN1)(bushi)
```

如果你只是想查看堆中的最小元素而不将其移除，可以直接访问列表的第一个元素：

```
min_value = data[0]
print(min_value)  # 输出: 0
```

使用 `heapq.merge(*iterables, key=None, reverse=False)` 可以合并多个已排序的输入迭代器，返回一个新的排序迭代器：

```
list1 = [1, 3, 5]
list2 = [2, 4, 6]
merged = list(heapq.merge(list1, list2))
print(merged)  # 输出: [1, 2, 3, 4, 5, 6]
```

如果你想从一个列表中构建一个新的堆，可以使用 `heapq.nsmallest(n, iterable, key=None)` 或 `heapq.nlargest(n, iterable, key=None)` 函数来获取前 n 个最小或最大的元素：

```
data = [5, 7, 9, 1, 3]
smallest_three = heapq.nsmallest(3, data)
print(smallest_three)  # 输出: [1, 3, 5]

largest_three = heapq.nlargest(3, data)
print(largest_three)   # 输出: [9, 7, 5]
```

全排列

`permutations` 是 Python 的 `itertools` 模块中提供的一个函数，用于生成从给定的元素集合中选取指定数量元素的所有排列方式。

要使用 `permutations`，首先需要导入 `itertools` 模块：

```
from itertools import permutations
```

`permutations` 函数接受两个参数：

- 第一个参数是一个可迭代对象（例如列表、字符串等），表示你想要从中获取排列的元素。
- 第二个参数是可选的，它指定了每个排列所含有的元素数目。如果不提供这个参数，则默认生成所有元素的全排列。

下面是一些使用 `permutations` 的例子：

1. 生成一个列表中所有元素的全排列：

```
for p in permutations(['a', 'b', 'c']):
    print(p)
```

这将输出：

```
('a', 'b', 'c')
('a', 'c', 'b')
('b', 'a', 'c')
('b', 'c', 'a')
('c', 'a', 'b')
('c', 'b', 'a')
```

1. 生成长度为 2 的所有排列：

```
for p in permutations(['x', 'y', 'z'], 2):
    print(p)
```

这将输出：

```
('x', 'y')
('x', 'z')
('y', 'x')
('y', 'z')
('z', 'x')
('z', 'y')
```

1. 将排列结果转换为列表：

```
perm_list = list(permutations('ABC', 2))
print(perm_list)
```

这将输出：

```
[('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('C', 'A'), ('C', 'B')]
```

1. 使用排列解决实际问题，比如找到一组数字可以组成的全部可能的不同整数：

```
digits = ['1', '2', '3']
all_possible_integers = set()
for perm in permutations(digits):
    all_possible_integers.add(int(''.join(perm)))
print(all_possible_integers)
```

这将输出：

```
{123, 132, 213, 231, 312, 321}
```

记住，`permutations` 返回的是一个迭代器，所以如果你想要查看所有的排列，你需要遍历它或将它转换为列表或元组等形式。由于排列的数量随着输入大小的增长而迅速增加，因此在处理较大的输入时应谨慎，以免消耗过多内存。

列表计数

`enumerate()` 是 Python 内置函数之一，它允许你在遍历一个可迭代对象（如列表、元组或字符串等）的同时获取元素的索引。

`enumerate()` 函数的基本语法如下：

```
enumerate(iterable, start=0)
```

- `iterable` 参数是必须的，表示你想要遍历的可迭代对象。
- `start` 参数是可选的，默认值为 0，表示索引的起始位置。

`bisect` 模块

1. `bisect.bisect_left(a, x, lo=0, hi=len(a))`:
   - 查找在已排序的列表 `a` 中，元素 `x` 应该插入的位置（保持列表有序），如果 `x` 已经存在，则返回其左侧（较低索引）的位置。
   - 如果 `x` 存在于列表中，它将返回第一个等于 `x` 的元素的索引。
2. `bisect.bisect_right(a, x, lo=0, hi=len(a))` **或者直接使用** `bisect.bisect(a, x, lo=0, hi=len(a))`:
   - 与 `bisect_left` 类似，但是它会返回 `x` 应该插入的位置，如果 `x` 已经存在，则返回其右侧（较高索引）的位置。
   - 如果 `x` 存在于列表中，它将返回最后一个等于 `x` 的元素之后的索引。

这两个函数都可以接受可选参数 `lo` 和 `hi` 来限制搜索的范围，默认情况下它们会搜索整个列表。

算法

欧拉筛(ES)

- 筛出1<=i<=n所有的素数
- 输入范围n,输出1~n的素数列表
- 如果你想查询一个很大的数是不是素数,把里面prime一开始做成集合再返回用来查询

```
def ES(n):
  isprime=[True for _ in range(n+1)]
  prime=[]
  for i in range(2,n+1):
      if isprime[i]:
          prime.append(i)
      for j in range(len(prime)):
          if  i*prime[j]>n:break
          isprime[i*prime[j]]=False
          if i%prime[j]==0 :break

  return prime
```

dfs

```
def dfs(x,y):
    d=[(-1,-1),(-1,0),(-1,1),(0,-1),(0,1),(1,-1),(1,0),(1,1)]
    for dx,dy in d:
        nx,ny=x+dx,y+dy
        if 0<=nx<len(mat) and 0<=ny<len(mat[0]):
            if mat[nx][ny]==1:
                mat[x][y]=0
                dfs(nx,ny)
                mat[x][y]=1
```

bfs

```
from collections import deque
def bfs(x0,y0,mat):
    q=deque([(x0,y0)])
    d=[(-1,0),(1,0),(0,1),(0,-1)]
    v=set()
    while q:#确定当前位置
        x,y=q.popleft()
        v.add((x,y))
        for dx,dy in d:#从当前位置开始探路
            nx,ny=x+dx,y+dy
            if 0<=nx<len(mat) and 0<=ny<len(mat[0]) and mat[nx][ny]==1:
                q.append((nx,ny))
```

Dijkstra算法

```
import heapq

directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    visited = [[float('inf')] * M for _ in range(N)]
    heap = []
    heapq.heappush(heap, (0, start[0], start[1]))
    visited[start[0]][start[1]] = 0
    found = False
    result = 0

    while heap:
        time, x, y = heapq.heappop(heap)
        if (x, y) == end:
            found = True
            result = time
            break
        if time > visited[x][y]:
            continue
```

```
        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            if 0 <= nx < N and 0 <= ny < M:
                if grid[nx][ny] == '#':
                    continue
                new_time = time + 1
                if grid[nx][ny] == 'x':
                    new_time += 1
                if new_time < visited[nx][ny]:
                    visited[nx][ny] = new_time
                    heapq.heappush(heap, (new_time, nx, ny))
```

Prim算法（最小生成树）

```
keys = [float('inf') for _ in range(n)]
in_mist = [False for _ in range(n)]
ans = 0
keys[0] = 0

for i in range(n):
    min_key = float('inf')
    for v in range(n):
        if not in_mist[v] and keys[v] < min_key:
            min_key = keys[v]
            min_index = v

    in_mist[min_index] = True
    ans+=min_key
    for v in range(n): #graph[i][j]是i与j之间的路径长度
        if graph[min_index][v] != float('inf') and not in_mist[v] :
            keys[v] = min(keys[v], graph[min_index][v])
print(ans)
```

二分查找

```
Min = min(numbers)
Max = sum(numbers)
result =Max
while Min <= Max:
    Mid = (Min + Max) // 2
    if search(numbers,Mid,m):
        result = Mid
        Max = Mid-1
    else:
        Min = Mid +1
print(result)
```

冒泡排序

```
for i in range(n):
    for j in range(n-1-i):
        if l[j] + l[j+1] > l[j+1] + l[j]:
            l[j],l[j+1] = l[j+1],l[j]
```

判断平方数

```
import math
def isPerfectSquare(num):
    if num < 0:
        return False
    sqrt_num = math.isqrt(num)
    return sqrt_num * sqrt_num == num
print(isPerfectSquare(97)) # False
```

中缀表达式转后缀表达式

算法步骤：

- 初始化一个空的操作符栈和一个空的输出队列。
- 从左到右读取中缀表达式的每个元素：
  - 如果是操作数，则直接加入输出队列。
  - 如果是操作符（如`+`, `-`, `*`, `/`），则将其压入操作符栈，但在压栈之前需要弹出所有优先级高于或等于当前操作符的操作符并加入输出队列。
  - 如果是左括号`(`，则直接压入操作符栈。
  - 如果是右括号`)`，则不断弹出栈顶的操作符并加入输出队列直到遇到左括号`(`，然后将这对括号丢弃。
- 当表达式全部处理完毕后，如果操作符栈中还有操作符，依次弹出并加入输出队列。

```
def infix_to_postfix(expression):
    precedence = {'+':1, '-':1, '*':2, '/':2, '^':3}
    stack = [] # 操作符栈
    output = [] # 输出队列
    tokens = expression.split()

    for token in tokens:
        if token.isalnum(): # 如果是操作数
            output.append(token)
        elif token == '(':
            stack.append(token)
        elif token == ')':
```

```
            top_token = stack.pop()
            while top_token != '(':
                output.append(top_token)
                top_token = stack.pop()
        else: # 是操作符
            while (stack and stack[-1] != '(' and
                   precedence[token] <= precedence.get(stack[-1],0)):
                output.append(stack.pop())
            stack.append(token)

    while stack:
        output.append(stack.pop())

    return ' '.join(output)

# 示例
print(infix_to_postfix("A * B + C * D"))  # "A B * C D * +"
```

中缀表达式转前缀表达式

主要步骤与后缀类似，但是：

- 在处理时，从右向左读取输入。
- 对于括号，处理方式变为先处理`)`，然后是`(`。
- 最终输出需要反转。

算法步骤：

- 初始化一个空栈。
- 从左至右扫描表达式的每个元素：
  - 如果是操作数，压入栈。
  - 如果是操作符，从栈中弹出所需数量的操作数，执行相应的运算，然后将结果压回栈。
- 扫描结束后，栈底元素即为最终结果

```
def evaluate_postfix(postfix):
    stack = []
    tokens = postfix.split()

    for token in tokens:
        if token.isdigit():
            stack.append(int(token))
        else:
            b = stack.pop()
            a = stack.pop()
            if token == '+': result = a + b
            elif token == '-': result = a - b
            elif token == '*': result = a * b
            elif token == '/': result = a / b
            stack.append(result)
```

```
    return stack[0]

# 示例
print(evaluate_postfix("7 8 + 3 2 + /"))  # 结果为3
```

最后 逃生指南

1. 除法是否使用地板除得到整数？（否则 4/2=2.0）

2. 是否有缩进错误？

3.每次接受不同组的输入时有没有重置数据？

4. 用于调试的print是否删去？

5. 非一般情况的边界情况是否考虑？

6. 递归中return的位置是否准确？（缩进问题,逻辑问题）

7. 贪心是否最优？有无更优解？

8. 正难则反（参考 #蒋子轩 23工院# 乌鸦坐飞机）

9. 审题是否准确？ 是否漏掉了输出？是否区分‘·’和‘.’？（参考）

10.深浅拷贝问题有没有注意？二维列表的深拷贝必须用`copy`模块中的`a=copy.deepcopy(b)`实现