数据类型及基本用法

数字

函数	返回值 (描述)
abs(x)	返回数字的绝对值，如abs(-10) 返回 10
ceil(x)	返回数字的上入整数，如math.ceil(4.1) 返回 5
exp(x)	返回e的x次幂(ex),如math.exp(1) 返回2.718281828459045
floor(x)	返回数字的下舍整数，如math.floor(4.9)返回 4
log(x)	如math.log(math.e)返回1.0,math.log(100,10)返回2.0
log10(x)	返回以10为基数的x的对数，如math.log10(100)返回 2.0
max(x1, x2,...)	返回给定参数的最大值，参数可以为序列。
min(x1, x2,...)	返回给定参数的最小值，参数可以为序列。
round(x[,n])]	返回浮点数 x 的四舍五入值，如给出 n 值，则代表舍入到小数点后的位数。**其实准确的说是保留值将保留到离上一位更近的一端。**
sqrt(x)	返回数字x的平方根。

字符串

不可以直接修改字符串中的字符

Python 字符串运算符

下表实例变量 a 值为字符串 "Hello"，b 变量值为 "Python":

操作符	描述	实例
+	字符串连接	a + b 输出结果: HelloPython
*	重复输出字符串	a*2 输出结果: HelloHello
[]	通过索引获取字符串中字符	a[1] 输出结果 **e**
[:]	截取字符串中的一部分，遵循**左闭右开**原则，str[0:2] 是不包含第 3 个字符的。	a[1:4] 输出结果 **ell**
in	成员运算符 - 如果字符串中包含给定的字符返回 True	**'H' in a** 输出结果 True
not in	成员运算符 - 如果字符串中不包含给定的字符返回 True	**'M' not in a** 输出结果 True

f-string格式化字符串

f-string 格式化字符串以 **f** 开头，后面跟着字符串，字符串中的表达式用大括号 **{}** 包起来，它会将变量或表达式计算后的值替换进去，实例如下：

```
>>> name = 'Runoob'
>>> f'Hello {name}' # 替换变量
'Hello Runoob'
>>> f'{1+2}'      # 使用表达式
'3'

>>> w = {'name': 'Runoob', 'url': 'www.runoob.com'}
>>> f'{w["name"]}: {w["url"]}'
'Runoob: www.runoob.com'
```

在 Python 3.8 的版本中可以使用 **=** 符号来拼接运算表达式与结果，并且可以**保留小数位数**：

```
>>> x = 1
>>> print(f'{x+1}')  # Python 3.6
2

>>> x = 1
>>> print(f'{x+1=}')  # Python 3.8
x+1=2

number = 123.456789
# 保留两位小数
formatted_number = f"{number:.2f}"
print(formatted_number)
# 输出: 123.46
```

Python 的字符串内建函数

序号	方法及描述
capitalize()	将字符串的第一个字符转换为大写
count(str, beg= 0,end=len(string))	返回 str 在 string 里面出现的次数，如果 beg 或者 end 指定则返回指定范围内 str 出现的次数。这个方法**会区分大小写，且不会重叠计数**（即一个子串的一部分不能作为另一个子串的一部分来计数）。如果 `start` 或 `end` **超过了字符串的实际长度**，Python **不会抛出错误**，而是将其视为字符串的最大或最小可能值。
find(str, beg=0, end=len(string))	检测 str 是否包含在字符串中，如果指定范围 beg 和 end ，则检查是否包含在指定范围内，如果**包含返回开始的索引值，否则返回-1**
index(str, beg=0, end=len(string))	跟find()方法一样，只不过如果**str不在字符串中会报一个异常。**

序号	方法及描述
isdigit()	如果字符串只包含数字则返回 True 否则返回 False.. 注意： **空字符串**：空字符串 `""` 被认为不是纯数字字符串，因此 `isdigit()` 会返回 `False`。 **浮点数**：包含小数点的字符串（如 `"123.45"`）不是纯数字字符串，因此 `isdigit()` 会返回 `False`。 **负号**：包含负号的字符串（如 `"-123"`）也不是纯数字字符串，因此 `isdigit()` 会返回 `False`。 **其他字符**：任何非数字字符（包括空格、字母等）都会导致 `isdigit()` 返回 `False`。
str.join(seq)	以指定字符串str作为分隔符，将 seq 中所有的元素(的字符串表示)合并为一个新的字符串
len(string)	返回字符串长度
lower()	转换字符串中所有大写字符为小写.
max(str)	返回字符串 str 中最大的字母。
min(str)	返回字符串 str 中最小的字母。
replace(old, new [, max])	replace(old, new [, max]) 把 将字符串中的 old 替换成 new,如果 max 指定，则替换不超过 max 次。返回一个新的字符串，并不修改原来的字符串
rfind(str, beg=0,end=len(string))	类似于 find()函数，不过是从右边开始查找.
rindex(str, beg=0, end=len(string))	类似于 index()，不过是从右边开始.
split(str="", num=string.count(str))	以 str 为分隔符截取字符串，如果 num 有指定值，则仅截取 num+1 个子字符串.不加参数使用按照所有的空格分隔，最好用
strip([chars])	在字符串上执行 lstrip()和 rstrip()
swapcase()	将字符串中大写转换为小写，小写转换为大写
title()	返回"标题化"的字符串,就是说所有单词都是以大写开始，其余字母均为小写(见 istitle())
upper()	转换字符串中的小写字母为大写

列表

Python列表脚本操作符

列表对 + 和 * 的操作符与字符串相似。+ 号用于组合列表，* 号用于重复列表。

如下所示:

Python 表达式	结果	描述
len([1, 2, 3])	3	长度
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	组合

Python 表达式	结果	描述
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	重复
3 in [1, 2, 3]	True	元素是否存在于列表中
for x in [1, 2, 3]: print(x, end=" ")	1 2 3	迭代

Python列表函数&方法

Python包含以下函数:

序号	函数
1	len(list) 列表元素个数
2	max(list) 返回列表元素最大值
3	min(list) 返回列表元素最小值
4	list(seq) 将元组转换为列表

Python包含以下方法:

序号	方法
1	list.append(obj) 在列表末尾添加新的对象
2	list.count(obj) 统计某个元素在列表中出现的次数
3	list.extend(seq) 在列表末尾一次性追加另一个序列中的多个值（用**新列表**扩展原来的列表）
4	list.index(obj) 从列表中找出某个值**第一个匹配项的索引**位置
5	list.insert(index, obj) 将对象插入列表
6	list.pop(index=-1]) 移除列表中的一个元素（**默认最后一个索引的元素**），并且**返回该元素的值**
7	list.remove(obj) 移除列表中某个值的**第一个匹配项**
8	list.reverse() **反向列表中元素**
9	list.sort(key=None, reverse=False) 对原列表进行**排序，reverse=True降序从大到小，False升序从小到大**
10	list.clear() 清空列表
11	list.copy() 复制列表

元组

元组的元素**不可以修改**，元组中只包含一个元素时，需要在元素后面添加逗号 `,` ，否则括号会被当作运算符使用

元组运算符

与字符串一样，元组之间可以使用 +、+=和 * 号进行运算，运算后会生成一个新的元组。

Python 表达式	结果	描述
`len((1, 2, 3))`	3	计算元素个数
`>>> a = (1, 2, 3) >>> b = (4, 5, 6) >>> c = a+b >>> c (1, 2, 3, 4, 5, 6)`	(1, 2, 3, 4, 5, 6)	连接，c 就是一个新的元组，它包含了 a 和 b 中的所有元素。
`>>> a = (1, 2, 3) >>> b = (4, 5, 6) >>> a += b >>> a (1, 2, 3, 4, 5, 6)`	(1, 2, 3, 4, 5, 6)	连接，a 就变成了一个新的元组，它包含了 a 和 b 中的所有元素。
`('Hi!',) * 4`	('Hi!', 'Hi!', 'Hi!', 'Hi!')	复制
`3 in (1, 2, 3)`	True	元素是否存在
`for x in (1, 2, 3): print (x, end=" ")`	1 2 3	迭代

元组内置函数

序号	方法及描述	实例
1	len(tuple) 计算元组元素个数。	`>>> tuple1 = ('Google', 'Runoob', 'Taobao') >>> len(tuple1) 3 >>>`
2	max(tuple) 返回元组中元素最大值。	`>>> tuple2 = ('5', '4', '8') >>> max(tuple2) '8' >>>`
3	min(tuple) 返回元组中元素最小值。	`>>> tuple2 = ('5', '4', '8') >>> min(tuple2) '4' >>>`
4	tuple(iterable) 将可迭代系列转换为元组。	`>>> list1= ['Google', 'Taobao', 'Runoob', 'Baidu'] >>> tuple1=tuple(list1) >>> tuple1 ('Google', 'Taobao', 'Runoob', 'Baidu')`

字典

字典是另一种**可变**容器模型，且可存储任意类型对象。

字典的每个键值 `key=>value` 对用冒号 **:** 分割，每个对之间用逗号(,)分割，整个字典包括在花括号 `{}` 中 ,格式如下所示：

```
d = {key1 : value1, key2 : value2, key3 : value3 }
```

注意： `dict` 作为 Python 的关键字和内置函数，变量名不建议命名为 **dict**。

键必须是唯一的，但值则不必。**值可以取任何数据类型，但键必须是不可变的，如字符串，数字，元组。**不允许同一个键出现两次。创建时如果同一个键被赋值两次，**后一个值会被记住**

舔删改

```
tinydict = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'}

tinydict['Age'] = 8               # 更新 Age
tinydict['School'] = "菜鸟教程"  # 添加信息
del tinydict['Name'] # 删除键 'Name'
tinydict.clear()     # 清空字典
del tinydict         # 删除字典
```

字典内置函数&方法

Python字典包含了以下内置函数：

序号	函数及描述	实例
1	**len(dict)** 计算字典元素个数，即键的总数。	`>>> tinydict = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'} >>> len(tinydict) #输出3`
2	str(dict) 输出字典，可以打印的字符串表示。	`>>> tinydict = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'} >>> str(tinydict) #输出" {'Name': 'Runoob', 'Class': 'First', 'Age': 7}"`
3	type(variable) 返回输入的变量类型，如果变量是字典就返回字典类型。	`>>> tinydict = {'Name': 'Runoob', 'Age': 7, 'Class': 'First'} >>> type(tinydict) #输出 <class 'dict'>`

Python字典包含了以下内置方法：

序号	函数及描述
1	dict.clear() 删除字典内所有元素
2	dict.copy() 返回一个字典的浅复制
3	**dict.fromkeys()** 创建一个新字典，**以序列seq中元素做字典的键，val为字典所有键对应的初始值（键值互换）**
4	dict.get(key, default=None) 返回指定键的值，如果键不在字典中返回 default 设置的默认值
5	key in dict 如果键在字典dict里返回true，否则返回false
6	dict.items() 以列表返回一个视图对象

序号	函数及描述
7	dict.keys() 返回一个视图对象
8	dict.setdefault(key, default=None) 和get()类似, 但如果键不存在于字典中，将会添加键并将值设为default
9	dict.update(dict2) 把字典dict2的键/值对更新到dict里
10	dict.values() 返回一个视图对象
11	pop(key[,default]) 删除字典 key（键）所对应的值，返回被删除的值。
12	popitem() 返回并删除字典中的最后一对键和值。

集合

创建格式：

```
parame = {value01,value02,...}
或者
set(value)
```

以下是一个简单实例：

```
set1 = {1, 2, 3, 4}              # 直接使用大括号创建集合
set2 = set([4, 5, 6, 7])       # 使用 set() 函数从列表创建集合
```

注意：创建一个空集合必须用 **set()** 而不是 **{ }**，因为 **{ }** 是用来创建一个空字典。

更多实例演示：

```
\>>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
\>>> print(basket)             # 这里演示的是去重功能
{'orange', 'banana', 'pear', 'apple'}
\>>> 'orange' in basket          # 快速判断元素是否在集合内
True
\>>> 'crabgrass' in basket
False

\>>> # 下面展示两个集合间的运算.
...
\>>> a = set('abracadabra')
\>>> b = set('alacazam')
\>>> a
{'a', 'r', 'b', 'c', 'd'}
\>>> a - b                     # 集合a中包含而集合b中不包含的元素
{'r', 'd', 'b'}
\>>> a | b                     # 集合a或b中包含的所有元素
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
\>>> a & b                     # 集合a和b中都包含了的元素
{'a', 'c'}
\>>> a ^ b                     # 不同时包含于a和b的元素
```

```
{'r', 'd', 'b', 'm', 'z', 'l'}
```

类似列表推导式，同样集合支持集合推导式(Set comprehension):

```
\>>> a = {x for x in 'abracadabra' if x not in 'abc'}
\>>> a
{'r', 'd'}
```

集合内置方法完整列表

方法	描述
add()	为集合添加元素
clear()	移除集合中的所有元素
copy()	拷贝一个集合
difference()	返回多个集合的差集
difference_update()	移除集合中的元素，该元素在指定的集合也存在。（移除交集）
discard()	删除集合中指定的元素，若该元素不存在不会发生错误
intersection()	返回集合的交集，返回一个新集合
intersection_update()	返回集合的交集。在原有集合的基础上移除不重叠的元素
isdisjoint()	判断两个集合是否包含相同的元素，如果没有返回 True，否则返回 False。
issubset()	判断指定集合是否为该方法参数集合的子集。
issuperset()	判断该方法的参数集合是否为指定集合的子集
pop()	随机移除元素
remove()	移除指定元素，若该元素不存在会发生错误
symmetric_difference()	返回两个集合中不重复的元素集合。
symmetric_difference_update()	移除当前集合中在另外一个指定集合相同的元素，并将另外一个指定集合中不同的元素插入到当前集合中。
union()	返回两个集合的并集
update()	给集合添加元素，且参数可以是列表，元组，字典等
len()	计算集合元素个数

defaultdict

- 这是一种"如果访问的key不存在与字典中,就给它新建一个默认值

```
dict[key]=default_value
```

的字典

```
from collections import defaultdict
a=defaultdict(list)#以空列表为默认值的defaultdict
#类似地,括号里是int,set也可以
```

- 如果要自定义默认值,用`defaultdict(lambda: xxxx)`

OrderedDict

- 这是一种"输入顺序即遍历顺序"的字典.这种字典可以像列表一样进行两端弹出

```
from collections import OrderedDict
od=OrderedDict()
od['a']=1
od['b']=2
od['c']=3
od['d']=4
```

- 删除一个元素:`del od[key]`
- 弹出一个元素:

```
od.pop(key(,default))#后面可加的default是当key不在字典中时返回的默认值
```

  - 注意,pop的时间复杂度是O(n),而普通字典是O(1)

- 弹出末尾:`od.popitem()`
- 弹出首:`od.popitem(last=False)`

语法

读取不定行输入

```
lines = []
while True:
    try:
        line = input()
        if line:  # 如果输入非空，则添加到列表中
            lines.append(line)
        else:  # 遇到空行则停止读取
            break
    except EOFError:  # 当用户输入 EOF (Ctrl+D on Unix, Ctrl+Z on Windows) 时退出循环
        break
```

缓存器

记忆化搜索（Memoization）以避免对相同位置进行重复计算代码：

```
from functools import lru_cache

# 定义移动方向
dx = [0, 0, -1, 1]
```

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

@lru_cache(maxsize=None)
def dfs(x, y):
    max_steps = 1  # 至少自己算一步
    for i in range(4):
        nx, ny = x + dx[i], y + dy[i]
        if 0 <= nx < r and 0 <= ny < c and h[nx][ny] < h[x][y]:
            max_steps = max(max_steps, 1 + dfs(nx, ny))
    return max_steps

# 读取输入
r, c = map(int, input().split())
h = []
for _ in range(r):
    h.append(tuple(map(int, input().split())))  # 使用tuple以便能被缓存

# 计算最大步数
max_step = 0
for i in range(r):
    for j in range(c):
        max_step = max(max_step, dfs(i, j))

print(max_step)
```

在这个版本中，我们做了以下改进：

1. **记忆化搜索**：使用`@lru_cache`装饰器来缓存已经计算过的结果，从而避免重复计算。
2. **输入处理**：将每一行的高度列表转换为元组（tuple），因为Python的`lru_cache`要求参数是可哈希的（hashable），而列表不是可哈希的，但元组是。

随机数debug

```
import random
x=random.randint(a,b)#>=a,<=b的随机整数
x=random.random()#0~1随机浮点数
x=random.uniform(a,b)#a,b之间随机浮点数
x=random.choice(list)#在列表list中随机选择
```

其他乱七八糟的语法

（一）保留x位小数

1.使用 `round()` 函数

`round()` 函数可以用来四舍五入到指定的小数位数。

```
number = 3.141592653589793
rounded_number = round(number, 2)
print(rounded_number)  # 输出 3.14
```

需要注意的是，`round()` 是根据四舍五入规则进行的，如果第三位小数是 5 或以上，则会进一位。

2.使字符串格式化

你可以使用字符串格式化方法来显示浮点数为保留两位小数的字符串形式。

使用 `format()` 方法

```
number = 3.141592653589793
formatted_number = "{:.2f}".format(number)
print(formatted_number)  # 输出 "3.14"
```

使用 f-string

```
number = 3.141592653589793
formatted_number = f"{number:.2f}"
print(formatted_number)  # 输出 "3.14"
```

3. 使用 `Decimal` 类型（精确计算）

如果你需要更精确地控制数字，并且避免浮点数运算带来的精度问题，可以使用 `decimal` 模块中的 `Decimal` 类型。

```
from decimal import Decimal, getcontext

# 设置全局精度
getcontext().prec = 4  # 总共四位，包括整数部分和小数部分

number = Decimal('3.141592653589793')
rounded_number = round(number, 2)
print(rounded_number)  # 输出 Decimal('3.14')
```

（二）字典推导式创建字典

```
keys = ['a', 'b', 'c']
values = [1, 2, 3]
my_dict = {k: v for k, v in zip(keys, values)}
```

（三）lambda函数

在Python中，`lambda` 关键字用于创建匿名函数，即没有名字的函数。`lambda` 函数可以**接受任意多个参数，但是只能有一个表达式，不能包含复杂的逻辑或多条语句。**`lambda` 函数通常用于需要一个简单函数的地方，比如排序或过滤操作。

基本语法

`lambda` 函数的基本语法如下：

```
lambda 参数1, 参数2, ... : 表达式
```

这里，`参数1, 参数2, ...` 是 `lambda` 函数的输入参数，而 `表达式` 是使用这些参数计算的结果，该结果会作为 `lambda` 函数的返回值。

也**可以没有形参，直接跟表达式**

示例

1.简单的加法:

```
add = lambda x, y: x + y
print(add(5, 3))  # 输出 8
```

2.排序列表中的字典: 如果你有一个由字典组成的列表，并且想要根据某个键对它们进行排序，你可以使用 `lambda` 函数来指定排序依据：

```
people = [
    {'name': 'Alice', 'age': 30},
    {'name': 'Bob', 'age': 25},
    {'name': 'Charlie', 'age': 35}
]
sorted_people = sorted(people, key=lambda person: person['age'])
print(sorted_people)
# 输出 [{'name': 'Bob', 'age': 25}, {'name': 'Alice', 'age': 30}, {'name':
'Charlie', 'age': 35}]
```

---

sorted()

`sorted()` 用于对序列进行**排序**，可以通过 `key` 参数自定义排序**规则**，通过 `reverse` 参数控制排序**方向**（**默认升序**）

基本语法

```
sorted(iterable, key=None, reverse=False)
```

- `iterable`：要排序的可迭代对象。
- `key`：一个函数，用来指定一个元素的排序关键字。
- `reverse`：一个布尔值，如果设置为 `True`，则排序结果为降序，默认为 `False`（升序）。

  示例：如按字符串长度排序

  ```
  words = ["apple", "banana", "cherry", "date"]
  sorted_words = sorted(words, key=len)
  print(sorted_words)  # 输出: ['date', 'apple', 'cherry', 'banana']
  ```

  在这个例子中，`key=len` 指定了排序的关键字为字符串的长度，因此列表中的字符串首先按照它们的长度进行了排序。

  如果我们想要降序排序，我们可以设置 `reverse=True`：

  ```
  sorted_words_desc = sorted(words, key=len, reverse=True)
  print(sorted_words_desc)  # 输出: ['banana', 'cherry', 'apple', 'date']
  ```

（四）正则表达式

正则表达式（Regular Expressions）是一种强大的文本处理工具，可以帮助你匹配、查找、替换字符串中的特定模式。

1.导入 `re` 模块

首先，你需要导入 Python 的 `re` 模块：

```
import re
```

2.基本概念

- **模式**：你要搜索的字符串模式。
- **目标字符串**：你想要在其中进行搜索的实际字符串。
- **匹配**：如果模式与目标字符串中的某部分完全一致，就称为“匹配”。

3. 常用函数

下面是一些常用的 `re` 函数及其简单示例：

3.1 `re.search()`

这个函数会在整个目标字符串中搜索第一个匹配的子串，并返回一个 `match` 对象。如果没有找到匹配，则返回 `None`。

```
import re

# 目标字符串
text = "Hello, my email is example@example.com"

# 模式
pattern = r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b'

# 搜索
match = re.search(pattern, text)

if match:
    print("找到了匹配:", match.group())  # 输出: 找到了匹配: example@example.com
else:
    print("没有找到匹配")
```

3.2 `re.findall()`

这个函数会找到所有匹配的子串，并返回一个列表。

```python
import re

# 目标字符串
text = "Numbers: 123, 456, 789"

# 模式
pattern = r'\d+'

# 查找所有数字
matches = re.findall(pattern, text)

print("找到的所有数字:", matches)  # 输出: 找到的所有数字: ['123', '456', '789']
```

3.3 `re.sub()`

这个函数可以用来替换字符串中的匹配项。

```python
import re

# 目标字符串
text = "The price is $100.00"

# 模式和替换内容
pattern = r'\$\d+\.\d{2}'
replacement = "€50.00"

# 替换
new_text = re.sub(pattern, replacement, text)

print("替换后的字符串:", new_text)  # 输出: 替换后的字符串: The price is €50.00
```

理解正则表达式确实需要一些时间，但我会尽量用更具体的例子和步骤来帮助你。我们从基础开始，一步步构建正则表达式。

4. 基本元字符

首先，了解一些基本的正则表达式元字符：

- `.`：匹配任何单个字符（除了换行符）。
- `*`：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，但不超过 m 次。
- `^`：匹配字符串的开始。
- `$`：匹配字符串的结束。
- `[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。
- `()`：用于分组，可以对一组字符应用量词。

- `\d`：匹配任何数字（等价于 `[0-9]`）。
- `\w`：匹配任何字母、数字或下划线（等价于 `[a-zA-Z0-9_]`）。
- `\s`：匹配任何空白字符（包括空格、制表符、换页符等）。
- `\b`：匹配单词边界。

5. 具体示例

示例 1: 匹配电话号码

假设我们要匹配格式为 `XXX-XXX-XXXX` 的电话号码。

```
import re

# 目标字符串
text = "Contact us at 123-456-7890 or 456-789-0123"

# 模式
pattern = r'\d{3}-\d{3}-\d{4}'

# 查找所有电话号码
phone_numbers = re.findall(pattern, text)

print("找到的电话号码:", phone_numbers)  # 输出: 找到的电话号码: ['123-456-7890',
'456-789-0123']
```

解释：

- `\d{3}`：匹配三个数字。
- `-`：匹配连字符 `-`。
- `\d{3}`：再匹配三个数字。
- `-`：再匹配连字符 `-`。
- `\d{4}`：最后匹配四个数字。

示例 2: 匹配电子邮件地址

假设我们要匹配电子邮件地址。

```
import re

# 目标字符串
text = "Hello, my email is example@example.com and another one is
test@domain.co.uk"

# 模式
pattern = r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b'

# 查找所有电子邮件地址
emails = re.findall(pattern, text)

print("找到的电子邮件地址:", emails)  # 输出: 找到的电子邮件地址:
['example@example.com', 'test@domain.co.uk']
```

解释：

- `\b`：匹配单词边界。
- `[A-Za-z0-9._%+-]+`：匹配一个或多个字母、数字或特殊字符（如 `.`、`_`、`%`、`+`、`-`）。
- `@`：匹配 `@` 符号。
- `[A-Za-z0-9.-]+`：匹配一个或多个字母、数字、`.` 或 `-`。
- `\.`：匹配点 `.`。
- `[A-Z|a-z]{2,}`：匹配两个或更多字母。
- `\b`：匹配单词边界。

示例 3: 提取日期

假设我们要从文本中提取日期（格式为 `YYYY-MM-DD`）。

```
import re

# 目标字符串
text = "Today's date is 2023-10-27 and tomorrow's date is 2023-10-28."

# 模式
pattern = r'\b\d{4}-\d{2}-\d{2}\b'

# 查找所有日期
dates = re.findall(pattern, text)

print("找到的日期:", dates)  # 输出: 找到的日期: ['2023-10-27', '2023-10-28']
```

解释：

- `\b`：匹配单词边界。
- `\d{4}`：匹配四位数字。
- `-`：匹配连字符 `-`。
- `\d{2}`：匹配两位数字。
- `-`：匹配连字符 `-`。
- `\d{2}`：匹配两位数字。
- `\b`：匹配单词边界。

6. 分步构建正则表达式

如果你需要构建一个复杂的正则表达式，可以一步一步来：

1. **确定你要匹配的内容**：明确你要从文本中提取什么信息。
2. **分解成小部分**：将整个模式分解成更小的部分。
3. **逐步构建**：从简单的部分开始，逐步添加更多的细节。
4. **测试和调整**：使用实际数据进行测试，并根据结果进行调整。

7. 小结

- 使用 `re` 模块来处理正则表达式。
- `re.search()` 用于查找第一个匹配。
- `re.findall()` 用于查找所有匹配。
- `re.sub()` 用于替换匹配的部分。
- 了解基本的正则表达式语法，如 `.`、`*`、`+`、`?`、`{n}` 等。

（五）rstrip（）

`rstrip()` 是 Python 中字符串的一个方法，用于**删除字符串末尾的指定字符。默认**情况下，它会删除字符串末尾的**所有空白**字符（包括空格、制表符 `\t`、换行符 `\n` 等）。你也可以**指定要删除的字符。**

注意，`rstrip()` 方法**并不支持删除末尾指定个数的字符**!

1. 删除末尾的所有空白字符

```
text = "Hello, World!   \n\t"
stripped_text = text.rstrip()
print(repr(stripped_text))  # 输出: 'Hello, World!'
```

在这个例子中，`rstrip()` 删除了字符串末尾的**所有空白**字符（包括空格、制表符和换行符）。

2. 删除指定的字符

你可以传递一个参数给 `rstrip()` 方法，指定要删除的字符。

```
text = "Hello, World!***"
stripped_text = text.rstrip('*')
print(repr(stripped_text))  # 输出: 'Hello, World!'
```

在这个例子中，`rstrip('*')` 删除了字符串末尾的**所有**星号 `*`。

3. 删除多个指定的字符

你可以传递一个包含多个字符的字符串，`rstrip()` 会删除这些字符中的任何一个在字符串末尾出现的情况。

```
text = "Hello, World!***---"
stripped_text = text.rstrip('*-')
print(repr(stripped_text))  # 输出: 'Hello, World!'
```

在这个例子中，`rstrip('*-')` 删除了字符串末尾的**所有**星号 `*` 和破折号 `-`。

4. 处理没有指定字符的情况

如果字符串末尾没有指定的字符，`rstrip()` **不会做任何改变。**

```
text = "Hello, World!"
stripped_text = text.rstrip('*-')
print(repr(stripped_text))  # 输出: 'Hello, World!'
```

在这个例子中，因为字符串末尾没有星号 `*` 或破折号 `-`，所以 `rstrip('*-')` 没有改变字符串。

总结

- `rstrip()` 用于删除字符串末尾的指定字符。
- 默认情况下，它删除**所有空白**字符。
- 你可以通过传递一个字符串参数来**指定**要删除的字符。
- 如果字符串末尾没有指定的字符，`rstrip()` 不会改变字符串。

（六）`global`声明全局变量（不常用）

在 Python 中，`global` 关键字用于**在函数内部声明一个全局变量**。这意味着你可以在函数内部修改全局作用域中的变量。**默认**情况下，当你在函数内部赋值给一个变量时，Python 会认为这是一个**局部变量**，除非你明确使用 `global` 关键字声明它是一个全局变量。

基本用法

1. 声明和使用全局变量

```
x = 10  # 全局变量

def my_function():
    global x  # 声明 x 是全局变量
    print(f"Inside function, before change: x = {x}")
    x = 20  # 修改全局变量 x
    print(f"Inside function, after change: x = {x}")

print(f"Outside function, before call: x = {x}")
my_function()
print(f"Outside function, after call: x = {x}")
```

输出：

```
Outside function, before call: x = 10
Inside function, before change: x = 10
Inside function, after change: x = 20
Outside function, after call: x = 20
```

在这个例子中，`global x` 声明了 `x` 是一个全局变量，因此在 `my_function` 内部对 `x` 的修改会影响到全局作用域中的 `x`。

2. 不使用 `global` 关键字

如果你不使用 `global` 关键字，函数内部的 `x` 会被视为一个新的局部变量：

```
x = 10  # 全局变量

def my_function():
    print(f"Inside function, before change: x = {x}")
    x = 20  # 这里会创建一个新的局部变量 x
    print(f"Inside function, after change: x = {x}")

print(f"Outside function, before call: x = {x}")
my_function()
print(f"Outside function, after call: x = {x}")
```

输出：

```
Outside function, before call: x = 10
Traceback (most recent call last):
  File "example.py", line 9, in <module>
    my_function()
  File "example.py", line 4, in my_function
    print(f"Inside function, before change: x = {x}")
UnboundLocalError: local variable 'x' referenced before assignment
```

在这个例子中，由于没有使用 `global` 关键字，Python 认为 `x` 是一个局部变量。当尝试在赋值前访问 `x` 时，会引发 `UnboundLocalError`，因为**局部变量 `x` 在赋值之前还没有被定义。**

使用 `global` 的注意事项

1. **避免滥用**：**尽量减少对全局变量的使用**，特别是在大型项目中。过多地使用全局变量可能会导致代码难以维护和调试。
2. **命名冲突**：使用全局变量时要注意避免命名冲突。如果多个函数都使用同一个全局变量，可能会导致意外的行为。
3. **可读性**：使用全局变量可能会影响代码的可读性和模块化。尽量将数据封装在函数或类中，通过参数传递数据。

示例：全局变量和局部变量的混合使用

```
# 定义全局变量
x = 10

def modify_global():
    global x  # 声明 x 是全局变量
    x = 20  # 修改全局变量 x
    print(f"Inside modify_global, x = {x}")

def use_local():
    x = 30  # 创建一个新的局部变量 x
    print(f"Inside use_local, x = {x}")

print(f"Outside functions, before calls: x = {x}")
modify_global()
use_local()
print(f"Outside functions, after calls: x = {x}")
```

输出：

```
Outside functions, before calls: x = 10
Inside modify_global, x = 20
Inside use_local, x = 30
Outside functions, after calls: x = 20
```

在这个例子中：

- `modify_global` 函数修改了全局变量 `x`。
- `use_local` 函数创建了一个新的局部变量 `x`，不影响全局变量 `x`。

（七）一行代码加保护圈

```
l = [[0]*(m+2)] +[[0] +list(map(int,input().split()))+[0] for _ in range(n)]+
[[0]*(m+2)]
```

（八）获得矩阵中最大值的写法：

```
maxk = max(max(l) for l in board)
num = sum(l.count(maxk) for l in board)
print(num, maxk)
```

或

```
a = list(c.values())
s = max(a)
print(a.count(s), s)
```

或

```
max_num=0
max_waste=0
for i in range(1025):
    for j in range(1025):
        if matrix[i][j]>max_waste:
            max_waste=matrix[i][j]
            max_num=1
        elif matrix[i][j]==max_waste:
            max_num+=1

print(max_num,max_waste)
```

（九）逻辑运算or：

- `or` 是一个逻辑运算符。在 Python 中，`or` 会**返回第一个真值（即非零或非空的值）**。如果**没有真值存在，它会返回最后一个值。**
- 在这个表达式中，`i % 2 or i < 3` 会首先计算 `i % 2`，如果结果为 1（真），则整个表达式的值为 1；如果结果为 0（假），则会评估 `i < 3` 的值（如果 `i` 小于 3，则返回 `True`，否则返回 `False`）。

(十）练习AXCII表的转化

```
a = input()
b = ord(a[0])
if b >= 97:
    b -= 32  #大写字母与小写字母的ASCII码差32
print (chr(b),end = '')
print(a[1:])
```

`ord()` 和 `chr()` 是 Python 中用于**处理字符和其对应的 ASCII 或 Unicode 编码值**的内置函数。

1. `ord()` 函数：

- 功能：`ord()` 函数接收一个长度为1的**字符串**，返回该字符对应的 **Unicode 编码值**。对于标准的 ASCII 字符集，这相当于返回字符的 ASCII 值。
- 示例：

```
print(ord('A'))  # 输出: 65
print(ord('a'))  # 输出: 97
print(ord('€'))  # 输出: 8364 (这是一个非 ASCII 的 Unicode 字符)
```

2. `chr()` 函数：

- 功能：`chr()` 函数接收一个**整数**参数，返回与之对应的**字符**。这个整数应该是有效的 Unicode 编码值。
- 示例：

```
print(chr(65))  # 输出: 'A'
print(chr(97))  # 输出: 'a'
print(chr(8364))  # 输出: '€'
```

这两个函数是互逆的，即 `ord(chr(n)) == n` 和 `chr(ord(c)) == c` 在有效范围内总是成立的。

(十一）enumerate函数

`enumerate` 函数是 Python 内置函数之一，它允许我们在遍历一个**可迭代对象**（如列表、元组、字符串等）的同时轻松获取元素的索引。这在你需要同时访问元素及其位置时非常有用。注意，它是一个迭代器，**不是用来直接访问某个特定元素和它的索引值的。**

`enumerate` 函数的基本语法如下：

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

- `iterable` 是你想要遍历的序列或可迭代对象。
- `start` 参数是可选的，默认从 0 开始计数，你可以指定一个整数来作为索引的起始值。

这里有一个使用 `enumerate` 的简单例子：

```python
# 定义一个列表
fruits = ['apple', 'banana', 'cherry']

# 使用 enumerate 函数遍历列表，并打印出每个元素及其索引
for index, fruit in enumerate(fruits):
    print(f"Index {index}: {fruit}")
```

输出将是：

```
Index 0: apple
Index 1: banana
Index 2: cherry
```

如果你想**改变索引的起始数字，比如从 1 开始**，你可以这样做：

```python
for index, fruit in enumerate(fruits, start=1):
    print(f"Item {index}: {fruit}")
```

这将输出：

```
Item 1: apple
Item 2: banana
Item 3: cherry
```

`enumerate` 返回的是一个枚举对象，它是一个迭代器，每次迭代会返回一个包含索引和对应值的 tuple。如果你需要将其转换为列表，可以直接使用 `list()` 函数，例如：

```python
enumerated_list = list(enumerate(fruits))
print(enumerated_list)
```

输出将是：

```
[(0, 'apple'), (1, 'banana'), (2, 'cherry')]
```

这样就可以方便地在遍历过程中利用到元素的索引了。

算法

一、欧拉筛(ES)找质数

- 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
```

二、贪心的两种解法

排序解法

用排序法常见的情况是输入一个包含几个（一般一到两个）权值的数组，通过排序然后遍历模拟计算的方法求出最优值。

后悔解法

思路是无论当前的选项是否最优都接受，然后进行比较，如果选择之后不是最优了，则反悔，舍弃掉这个选项；否则，正式接受。如此往复。

后悔解法例题：potions

题意：按顺序喝药，药效有正有负，每瓶药可以喝或者不喝，求最后的药效最大值

代码：

```
import heapq
n=int(input())
a=list(map(int,input().split()))
heap=[]
health=0

for i in a:
    heapq.heappush(heap,i)
    health+=i
    if health<0:
        if heap: #确保heap里有元素，防止index error（虽然heap里肯定有元素
            give_up=heapq.heappop(heap)
            health-=give_up #放弃一瓶肯定也就够了，因为它是最负的，而health在喝i之前>=0
print(len(heap))
```

三、dp

最大连续子序列和

```
n = int(input())
a, = map(int, input().split())

dp = [0]*n
dp[0] = a[0]
```

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

print(max(dp))
```

最长公共子序列

```
for i in range(len(A)):
    for j in range(len(B)):
        if A[i] == B[j]:
            dp[i][j] = dp[i-1][j-1]+1
        else:
            dp[i][j] = max(dp[i-1][j],dp[i][j-1])
```

最长单调子序列（和）

```
dp = [1]*n
for i in range(1,n):
    for j in range(i):
        if A[j]<A[i]:
            dp[i] = max(dp[i],dp[j]+1)
ans = sum(dp)
```

背包问题

0-1背包:只有一件,选择只有拿/不拿(重点:初始化,转移方程,提取结果)

```
def knapsack_2d(weights, values, W):
    n = len(weights)
    dp = [[0] * (W + 1) for _ in range(n + 1)]
    for i in range(1, n + 1):
        for j in range(W + 1):
            if j >= weights[i - 1]:  # 第i个物品能装进,判断不选/选这个物品
                dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weights[i - 1]] +
values[i - 1])
            else: dp[i][j] = dp[i - 1][j]
    return dp[n][W]
```

```
def knapsack_1d(weights, values, W):  # 一维视为二维的滚动数组实现
    n = len(weights)
    dp = [0] * (W + 1)  # 初始化 dp 数组，容量从 0 到 W
    for i in range(n):  # 遍历每件物品
        for j in range(W, weights[i] - 1, -1):  # 倒序遍历背包容量(保证每件物品只能选一
次)
            dp[j] = max(dp[j], dp[j - weights[i]] + values[i])
    return dp[W]
```

1. **状态转移关系的映射**

二维DP状态	一维DP状态
`dp[i][j]`	`dp[j]`
`dp[i-1][j]`	上一轮的 `dp[j]`
`dp[i-1][j-k]`	当前数组中未覆盖部分

2.**倒序遍历的原因**:在一维DP中,为了**保证当前状态** `dp[j]` **只使用上一轮的状态值**,我们需要从容量 `w` 倒序遍历.**如果正序遍历,** `dp[j]` **会被更新后的** `dp[j - weight[i]]` **影响**,从而导致错误的结果.

区间dp

`dp[i][j]` 表示区间 `[i, j]` 的最优解 状态转移:拆分区间`dp[i][j]=min/max{dp[i][k]+dp[k+1][j]+cost(i,j,k)}(i<=k<j)`,从小到大递推. 预处理:利用前缀和等快速计算区间`cost`;结合特殊性质优化(否则复杂度为$O(n^3)$)

```
n = int(input())  # 石子的堆数
stones = list(map(int, input().split()))
sum_ = [0] * (n + 1)  # 前缀和，用于快速计算区间和
for i in range(1, n + 1):
    sum_[i] = sum_[i - 1] + stones[i - 1]
dp = [[float('inf')] * n for _ in range(n)]  # dp 数组，初始化为正无穷
for i in range(n):
    dp[i][i] = 0  # 单堆的代价为 0
for L in range(2, n + 1):  # 枚举区间长度 L，从 2 到 n
    for i in range(n - L + 1):  # 起始位置从0到n-L
        j = i + L - 1  # 长度为L后的终点
        for k in range(i, j):  # 枚举分割点 k
            dp[i][j] = min(dp[i][j], dp[i][k] + dp[k + 1][j] + sum_[j + 1] -
sum_[i])
print(dp[0][n - 1])
```

完全背包问题： 允许在不超容量的前提下无限次选取

```
def knapsack_complete(weights, values, capacity):
    dp = [0] * (capacity + 1)  # dp[j]为当背包容量为j时,背包所能容纳的最大价值
    dp[0] = 0
    for i in range(len(weights)):  # 遍历所有物品
        for j in range(weights[i], capacity + 1):  # 从当前物品的重量开始，计算每个容量
的最大价值
            dp[j] = max(dp[j], dp[j - weights[i]] + values[i])
    return dp[capacity]
```

必须装满的类型:

```python
def knapsack_complete_fill(weights, values, capacity):
    dp = [-float('inf')] * (capacity + 1)  # 初始值为负无穷，表示不能达到该容量
    dp[0] = 0  # 容量为 0 时，价值为 0
    for i in range(len(weights)):  # 遍历所有物品
        for w in range(weights[i], capacity + 1):  # 遍历所有容量，从 weights[i] 开始
            dp[w] = max(dp[w], dp[w - weights[i]] + values[i])
    # 如果 dp[capacity] 仍为 -inf，说明无法填满背包
    return dp[capacity] if dp[capacity] != -float('inf') else 0
```

多重背包： 每个物品有上限

```python
def binary_optimized_multi_knapsack(weights, values, quantities, capacity):
    # 使用二进制优化解决多重背包问题
    n = len(weights)
    items = []
    # 将每种物品根据数量拆分成若干子物品（使用二进制优化）
    for i in range(n):
        w, v, q = weights[i], values[i], quantities[i]
        k = 1
        while k < q:
            items.append((k * w, k * v))  # 添加子物品(weight, value)
            q -= k
            k << 1  # 位运算,相当于k *= 2,按二进制拆分,物品时间复杂度由q变为log(q)
        if q > 0:
            items.append((q * w, q * v))  # 添加剩余部分,如果有的话
    # 动态规划求解0-1背包问题
    dp = [0] * (capacity + 1)
    for w, v in items:  # 遍历所有子物品
        for j in range(capacity, w - 1, -1):  # 01背包的倒序遍历
            dp[j] = max(dp[j], dp[j - w] + v)
    return dp[capacity]
```

dp中“正难则反”的常见类型

1. **区间问题**:从目标区间反向分解为子区间
2. **路径问题**:从终点反向推导到起点，利用已有结果
3. **子序列问题**:从末尾回溯，逐步构建解

双重dp

解决有不同方案的情况,e.g.土豪购物

```python
value = list(map(int, input().split(",")))
dp_keep = value[0]      # 不放回
dp_remove = value[0]    # 放回一件商品
ans = value[0]  # 答案记录
for i in range(1, len(value)):  # 对结尾为第i件商品的选法
    previous_dp_keep = dp_keep  # 保存结尾尾i-1时的选法
    dp_keep = max(dp_keep + value[i], value[i])  # 维护dp_keep
    dp_remove = max(previous_dp_keep, dp_remove + value[i])  # 判断是否放回第i个商品更划算
    ans = max(ans, dp_keep, dp_remove)
print(ans)
```

四、recursion

设置递归深度

```
import sys
sys.setrecursionlimit(1<<30)  # 设置递归深度为2^30
```

五、two pointers

接雨水

```
class Solution:
    def trap(self, height: List[int]) -> int:
        ans = left = pre_max = suf_max = 0  # 初始化结果、左指针和两个最大高度为0
        right = len(height) - 1  # 初始化右指针为数组末尾
        while left < right:  # 当左指针小于右指针时循环
            pre_max = max(pre_max, height[left])  # 更新左指针位置的最大高度
            suf_max = max(suf_max, height[right])  # 更新右指针位置的最大高度
            if pre_max < suf_max:  # 如果左指针位置的最大高度小于右指针位置的最大高度
                ans += pre_max - height[left]  # 计算并累加左指针位置能够接住的雨水量
                left += 1  # 移动左指针
            else:  # 否则
                ans += suf_max - height[right]  # 计算并累加右指针位置能够接住的雨水量
                right -= 1  # 移动右指针
        return ans  # 返回最终结果
```

六、DFS

dfs、bfs都可以遍历所有点，但是**dfs同时会遍历所有路径**，而bfs不一定，**bfs往往可以用于解决最短路径问题**

示例：sy314指定步数的迷宫问题 中等

辅助visited空间

```
MAXN = 5
n, m, k = map(int, input().split())
maze = []
for _ in range(n):
    row = list(map(int, input().split()))
    maze.append(row)

visited = [[False for _ in range(m)] for _ in range(n)]
canReach = False

MAXD = 4
dx = [0, 0, 1, -1]
dy = [1, -1, 0, 0]

def is_valid(x, y):
    return 0 <= x < n and 0 <= y < m and maze[x][y] == 0 and not visited[x][y]

def DFS(x, y, step):
    global canReach
```

```
    if canReach:
        return
    if x == n - 1 and y == m - 1:
        if step == k:
            canReach = True
        return
    visited[x][y] = True
    for i in range(MAXD):
        nextX = x + dx[i]
        nextY = y + dy[i]
        if step < k and is_valid(nextX, nextY):
            DFS(nextX, nextY, step + 1)
    visited[x][y] = False

DFS(0, 0, 0)
print("Yes" if canReach else "No")
```

七、BFS

bfs模板

广度优先搜索 (BFS)一般由**队列**实现,且总是**按层次的顺序进行遍历**，其基本写法如下(可作BFS模板用):

我们使用from collections import deque就满足要求，适用于需要频繁**从队列的两端进行操作**的场景，如广度优先搜索（BFS）、滑动窗口等问题。

> from queue import Queue适用于多线程编程中，需要在多个线程之间安全地共享和传递数据的场景。提供线程安全的特性，内置锁机制，可以在多线程环境中安全地使用。支持阻塞操作，如 `get` 和 `put` 方法可以设置超时时间，等待队列中有数据可用或空间可用。不支持从队列两端进行操作，只能从一端进行插入和删除。

```
from collections import deque

def bfs(start, end):
    q = deque([(0, start)])  # (step, start)
    in_queue = {start}


    while q:
        step, front = q.popleft() # 取出队首元素
        if front == end:
            return step # 返回需要的结果，如：步长、路径等信息

        # 将 front 的下一层结点中未曾入队的结点全部入队q，并加入集合in_queue设置为已入队
```

下面是对该模板中每一个步骤的说明,请结合代码一起看:

① 定义队列 q，并将起点(0, start)入队，0表示步长目前是0。
② 写一个 while 循环，循环条件是队列q非空。
③ 在 while 循环中，先取出队首元素 front。
④ 将front 的下一层结点中所有**未曾入队**的结点入队，并标记它们的层号为 step 的层号加1，并加入集合in_queue设置为已入队。
⑤ 返回 ② 继续循环。

为了防止走回头路，一般可以设置一个set类型集合**in_queue来记录每个位置是否在BFS中已入过队。**再强调一点，在BFS 中设置的 in_queue 集合的含义是判断结点是否已入过队，而不是**结点是否已被访问。**区别在于：如果设置成是否已被访问，有可能在某个结点正在队列中（但还未访问）时由于其他结点可以到达它而将这个结点再次入队，导致很多结点反复入队，计算量大大增加。因此BFS 中**让每个结点只入队一次**，故需要设置 in_queue 集合的含义为**结点是否已入过队**而非结点是否已被访问。

25572: 螃蟹采蘑菇

bfs, dfs, http://cs101.openjudge.cn/practice/25572/

思路：bfs模板题，占几个位置其实都是一样判断

```
from collections import deque
dx=[0,0,1,-1]
dy=[1,-1,0,0]

n=int(input())
maze=[]
for _ in range(n):
    line=list(map(int,input().split()))
    maze.append(line)

def bfs(a1,b1,a2,b2):
    q=deque()
    q.append(((a1,b1),(a2,b2)))
    inq=set()
    inq.add(((a1,b1),(a2,b2)))
    found=False
    while q:
        (x1,y1),(x2,y2)=q.popleft()
        if maze[x1][y1]==9 or maze[x2][y2]==9:
            found=True
            return found

        for i in range(4):
            nx1=x1+dx[i]
            nx2=x2+dx[i]
            ny1=y1+dy[i]
            ny2=y2+dy[i]
            if 0<=nx1<n and 0<=nx2<n and 0<=ny1<n and 0<=ny2<n and maze[nx1]
[ny1]!=1 and maze[nx2][ny2]!=1 and ((nx1,ny1),(nx2,ny2)) not in inq:
                q.append(((nx1,ny1),(nx2,ny2)))
                inq.add(((nx1,ny1),(nx2,ny2)))

    return found

start=[]
for i in range(n):
    for j in range(n):
        if maze[i][j]==5:
            start.append((i,j))
a1,b1,a2,b2=start[0][0],start[0][1],start[1][0],start[1][1]
result=bfs(a1,b1,a2,b2)
if result:
```

```
    print('yes')
else:
    print('no')
```

补充：队列

队列（Queue）是一种特殊的线性数据结构，其操作遵循**先进先出**（First In First Out, FIFO）的原则。这意味着最先被添加到队列中的元素也会是最先被移除的元素。队列的基本操作包括：

1. **入队（Enqueue）**：在队列的**尾部**添加一个新的元素。(**append**)
2. **出队（Dequeue）**：从队列的**头部**移除一个元素。(**popleft**)
3. **查看队头元素（Front）**：查看队列中最前面的元素，但不移除它。(**queue[0]**)
4. **查看队尾元素（Rear）**：查看队列中最后面的元素，但不移除它。(**queue[-1]**)
5. **判断队列是否为空（IsEmpty）**：检查队列中是否有任何元素。(**if not queue**)
6. **获取队列大小（Size）**：返回队列中元素的数量。(**len(queue)**)

Python 中的队列实现

Python 中可以使用多种方式实现队列，其中一种简单的方法是使用标准库 `collections` 中的 `deque` 类。`deque` 是双端队列的缩写，虽然它支持两端的操作，但也可以用作普通队列：

```
from collections import deque

# 创建一个空队列
queue = deque()

# 入队操作
queue.append('a')
queue.append('b')
queue.append('c')

# 查看队头元素
print(queue[0])  # 输出: 'a'

# 出队操作
item = queue.popleft()
print(item)  # 输出: 'a'

# 判断队列是否为空
if not queue:
    print("队列为空")
else:
    print("队列不为空")

# 获取队列大小
print(len(queue))  # 输出队列中元素的数量
```

在这个例子中，`deque` 被用来创建一个队列，并展示了如何向队列中添加元素（入队）、移除并获取队头元素（出队），以及如何检查队列是否为空和获取队列的大小。

Dijkstra

20106: 走山路

Dijkstra, http://cs101.openjudge.cn/practice/20106/

思路：和bfs最大的区别在于每一步的权重不都是1，而是不同的非负数！进队的条件需要改成到达该点的体力值更小，而不是没有进过对就行，同时允许重复进队以反复更新最小体力值

deque实现

```
from collections import deque
dx=[1,-1,0,0]
dy=[0,0,1,-1]

m,n,p=map(int,input().split())
landscape=[]
for _ in range(m):
    landscape.append(list(input().split()))

for _ in range(p):
    x1,y1,x2,y2=map(int,input().split())
    if landscape[x1][y1]=='#' or landscape[x2][y2]=='#':
        print('NO')
    else:
        queue=deque()
        force = [[float('inf')] * n for _ in range(m)]  # 记录到达每个点所需要的最小体力值
        force[x1][y1]=0
        queue.append((x1,y1))
        can_reach=False

        while queue:
            x,y=deque.popleft(queue)
            if x==x2 and y==y2:
                can_reach=True

            for i in range(4):
                nf=force[x][y]
                nx=x+dx[i]
                ny=y+dy[i]
                if 0<=nx<m and 0<=ny<n and landscape[nx][ny]!='#':
                    nf+=abs(int(landscape[nx][ny])-int(landscape[x][y]))
                    if nf<force[nx][ny]:
                        force[nx][ny]=nf
                        queue.append((nx,ny))

        if not can_reach:
            print('NO')
        else:
            print(force[x2][y2])
```

heapq实现

```
import heapq
directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
def dijkstra(xs, ys, xe, ye):  # 走山路
    if region[xs][ys] == "#" or region[xe][ye] == "#":
        return "NO"
    if (xs, ys) == (xe, ye):
        return 0
    pq = []  # 初始化堆
    heapq.heappush(pq, (0, xs, ys))  # (体力消耗, x坐标, y坐标)
    visited = set()  # 已访问坐标集
    efforts = [[float('inf')] * n for _ in range(m)]  # 到达图中位置需要的体力,初始化为inf,即不可达到
    efforts[xs][ys] = 0  # 初始化体力记录表
    while pq:
        current_effort, x, y = heapq.heappop(pq)  # 取出堆顶元素
        if (x, y) in visited: continue  # 若访问过,跳过这个节点
        visited.add((x, y))  # 未访问过,加入已访问
        if (x, y) == (xe, ye):  return current_effort  # 达到终点则返回
        for dx, dy in directions:
            nx, ny = x+dx, y+dy
            if 0 <= nx < m and 0 <= ny < n and (nx, ny) not in visited:
                if region[nx][ny] == "#":  continue  # 无法达到,跳过
                effort = current_effort + abs(int(region[x][y]) - int(region[nx][ny]))
                if effort < efforts[nx][ny]:
                    efforts[nx][ny] = effort
                    heapq.heappush(pq, (effort, nx, ny))  # 放入堆中
    return "NO"
```

04129: 变换的迷宫（三维数组）

bfs, http://cs101.openjudge.cn/practice/04129/

思路：原本写了一个和走山路差不多的代码，然后发现了关键问题——我们应该首先考虑能够到达的情况，然后再研究最短的时间。如果一个地方是石头，可以在其旁边来回踱步直到能够通过，如果按照更短时间进队的话，踱步的情况显然时间更长，无法被考虑到。

询问AI后得到的解决方案是在times数组里添加一个数据维度，将时间按照除以k的余数分类，以保证能通过的石头都通过，同时用时最短

```
from collections import deque

#方向数组
dx=[0,0,1,-1]
dy=[1,-1,0,0]

n=int(input())
for _ in range(n):
    r,c,k=map(int,input().split())
    maze=[]
    for _ in range(r):
        maze.append(list(input()))
```

```
    #初始化起点和终点的坐标
    sx,sy,ex,ey=-1,-1,-1,-1
    for i in range(r):
        for j in range(c):
            if maze[i][j]=='S':
                sx,sy=i,j
            if maze[i][j]=='E':
                ex,ey=i,j

    #bfs
    queue=deque()
    queue.append((sx,sy,0))  #初始位置和时间
    times=[[[float('inf')]*k for _ in range(c)] for _ in range(r)]  #记录到达每个点的最短用时,按t%k分类
    times[sx][sy][0]=0

    found=False  #是否找到终点
    while queue:
        x,y,t=deque.popleft(queue)

        #检查是否到达出口
        if (x,y)==(ex,ey):
            print(t)
            found=True
            break

        for i in range(4):
            nx=x+dx[i]
            ny=y+dy[i]
            nt=t+1
            nk=nt%k

            if 0<=nx<r and 0<=ny<c:
                # 如果下一步的时间是K的倍数，那么可以走任何位置；否则检查目标位置是否为石头
                if nk==0 or maze[nx][ny]!='#':
                    if nt<times[nx][ny][nk]:
                        times[nx][ny][nk]=nt
                        queue.append((nx,ny,nt))

    if not found:
        print('Oop!')
```

补充：heapq

`heapq` 是 Python 的一个内置模块，提供了堆队列算法的实现。堆是一种特殊的树结构，满足堆属性：对于任意给定的节点 C，如果 P 是 C 的父节点，则 P 的键值小于或等于 C 的键值（在最小堆中），或者大于或等于 C 的键值（在最大堆中）。这种结构非常适合用来实现**优先级队列**。

`heapq` 模块实现了最小堆，意味着堆中的最小元素总是被弹出。Python 中的 `heapq` 实际上是**在列表（list）的基础上实现的**，但是它提供了一系列函数来维护这个列表作为一个堆。

以下是 `heapq` 模块的一些常用函数：

- `heappush(heap, item)`：将 `item` **添加到堆** `heap` **中，并保持堆的性质。**
- `heappop(heap)`：从堆 `heap` 中弹出并返回最小的元素。**如果堆为空，则引发 IndexError。**

- `heapreplace(heap, item)`：用 `item` 替换堆中的最小元素并返回旧的最小元素。等价于 `heappop()` 后跟 `heappush()`，但更高效。
- `heappushpop(heap, item)`：将 `item` 添加到堆 `heap` 中，然后弹出并返回最小的元素。与 `heapreplace()` 相反，当有新的 `item` 要插入时先推后弹。
- `heapify(x)`：**将列表 `x` 转换成堆，原地修改，线性时间内完成。**
- `nlargest(n, iterable[, key])`：返回 `iterable` 中**最大的 `n` 个元素组成的列表，可以指定 `key` 函数用于排序。**
- `nsmallest(n, iterable[, key])`：返回 `iterable` 中**最小的 `n` 个元素组成的列表，可以指定 `key` 函数用于排序。**

使用 `heapq` 创建和操作堆的一个简单例子如下：

```
import heapq

# 初始化一个空堆
heap = []

# 使用 heappush 添加元素
heapq.heappush(heap, 10)
heapq.heappush(heap, 1)
heapq.heappush(heap, 5)

# 查看堆顶元素（最小元素）
print("The smallest element is:", heap[0])

# 弹出最小元素
min_element = heapq.heappop(heap)
print("Popped the smallest element:", min_element)

# 将列表转换为堆
lst = [3, 1, 4, 1, 5, 9]
heapq.heapify(lst)
print("Heapified list:", lst)

# 获取三个最大的元素
largest_three = heapq.nlargest(3, lst)
print("Three largest elements are:", largest_three)
```

八、辅助栈

辅助栈是一种数据结构设计模式，通常用于增强或扩展另一个栈的功能。通过使用一个或多个额外的栈（即辅助栈），可以更高效地实现某些操作或特性，而这些操作或特性在单个栈上可能难以实现或效率较低。

辅助栈的应用场景

1. **最小栈（Min Stack）**：
    - 问题描述：设计一个支持 `push`、`pop`、`top` 操作，并能在常数时间内检索到最小元素的栈。

- 解决方案：除了主栈外，我们还可以维护一个辅助栈，用来存储当前栈中的最小值。每当有新元素入栈时，如果该元素小于或等于辅助栈顶元素，则也将其压入辅助栈；**当从主栈弹出元素时，如果该元素等于辅助栈顶元素，则同时从辅助栈中弹出。这样，辅助栈顶始终保存着当前栈中的最小值。**

2. **平衡括号检查**：
   - 问题描述：给定一个包含多种类型括号（如圆括号 `()`、方括号 `[]` 和花括号 `{}`）的字符串，判断这些括号是否正确配对。
   - 解决方案：可以使用一个主栈来处理括号匹配的问题，而辅助栈则可以帮助记录每种括号最后出现的位置或类型，以便进行更复杂的匹配逻辑。
3. **回滚机制**：
   - 在某些应用中，你可能需要能够撤销最近的操作。可以使用辅助栈来保存每个操作的状态，从而允许用户回退到之前的状态。
4. **多态性操作**：
   - 有时你需要根据栈的不同状态执行不同的操作。例如，在一个表达式求值器中，你可以使用辅助栈来跟踪运算符的优先级或者存储中间计算结果。

最小栈示例代码

```python
class MinStack:

    def __init__(self):
        self.stack = []  # 主栈
        self.min_stack = []  # 辅助栈，用于保存每个状态下的最小值

    def push(self, x: int) -> None:
        self.stack.append(x)
        if not self.min_stack or x <= self.min_stack[-1]:
            self.min_stack.append(x)

    def pop(self) -> None:
        if self.stack:
            top = self.stack.pop()
            if top == self.min_stack[-1]:
                self.min_stack.pop()

    def top(self) -> int:
        if self.stack:
            return self.stack[-1]

    def getMin(self) -> int:
        if self.min_stack:
            return self.min_stack[-1]

# 示例用法
min_stack = MinStack()
min_stack.push(-2)
min_stack.push(0)
min_stack.push(-3)
print(min_stack.getMin())  # 返回 -3
min_stack.pop()
print(min_stack.top())     # 返回 0
```

```
print(min_stack.getMin())  # 返回 -2
```

在这个例子中，`MinStack` 类实现了基本的栈操作以及 `getMin` 方法，可以在 O(1) 时间复杂度内获取当前栈中的最小值。通过维护两个栈——一个用于所有元素，另一个仅用于最小值——我们可以确保每次 `push` 和 `pop` 操作都能正确更新最小值信息。这里用的辅助栈是普通的列表而非heapq，但是heapq更无脑XD

使用heapq的最小栈示例代码：快速堆猪

```
import heapq
pig=[]
heap=[]
def pop1(pig,heap):
    if pig and heap:
        remove=pig.pop()
        if remove==heap[0]:
            heapq.heappop(heap)
        while heap and (heap[0] not in pig):
            heapq.heappop(heap)
    return

def min1(pig,heap):
    if pig and heap:
        result=heap[0]
        return result
    else:
        return -1

def push1(n,pig,heap):
    pig.append(n)
    heapq.heappush(heap,n)
    return

while True:
    try:
        line=list(input().split())
        if line:  #如果输入非空，执行操作
            if line[0]=='pop':
                pop1(pig,heap)
            elif line[0]=='min':
                if min1(pig,heap)==-1:
                    continue
                else:
                    print(min1(pig,heap))
            elif line[0]=='push':
                n=int(line[1])
                push1(n,pig,heap)
        else:  # 遇到空行则停止读取
            break
    except EOFError:  # 当用户输入 EOF (Ctrl+D on Unix, Ctrl+Z on Windows) 时退出循环
        break
```