

0. 简述python的发展过程、应用方向及相关的第三方包或者库

发展过程：1989年，吉多·范罗苏姆（Guido van Rossum）决定开发一个新的脚本解释程序，作为ABC语言的一种继承，并将其命名python。1992年，他发布了Python的web框架Zope1。Python1.0版本于1994年1月发布。2000年10月份，Python2.0发布了。这个版本的主要新功能是内存管理和循环检测垃圾收集器以及对Unicode的支持。2008年的12月份，Python3.0发布了。

应用方向：网络爬虫（request, scrapy），数据分析（numpy, pandas），机器学习（theano），web开发（Django, flask），数据可视化（echart）及其他第三方库

1. python的安装(安装时记得勾选pip及添加path至环境变量中)。 如何启动python，如何查看python版本？不要删除Mac或者linux中内置的python 2.x 版本，不然系统一些功能会失效或者报错

Windows打开命令行界面，输入Python即可打开 或 使用pycharm, vscode等编译器

使用 `python -v` 查看版本

2.理解环境变量

- Mac(不理解直接跳过即可，不影响学习下面的内容):
 1. 终端shell是bash `~/.bash_profile`
 2. 终端shell是zsh `~/.zshrc`

解释：Mac系统上装有bash、zsh等shell。用户可以根据自己的偏好修改喜欢的shell，
--切换bash: `chsh -s /bin/bash`
--切换zsh: `chsh -s /bin/zsh`
- Windows: 我的电脑 右击 » 属性 » 关于 » 高级系统设置 » 环境变量 » 用户变量 (**尽量修改用户变量，不用动系统变量**)

3.python虚拟环境

- `venv` --python 3.4 之后默认
- `virtualenv` ---推荐使用该软件 Windows与Mac 操作系统略有不同 `virtualenv`从版本20开始，默认就是`--no-site-packages` 简述使用`virtualenv`进行python虚拟空间创建的主要步骤。
- `pipenv`

生产环境: docker 中配置python

4.pip是什么，有什么用？python3对应的pip命令一般是什么？

pip是一个以Python计算机程序语言写成的软件包管理系统，他可以安装和管理软件包。

查看pip：直接在cmd窗口中输入pip命令，会显示pip所有的参数使用方法。

pip版本：用`pip -V`可以查看版本。

查看已经安装的第三方库：`pip list`。

安装第三方库：`pip install 库名`。

查看安装：`pip show 库名`

卸载第三方库：pip uninstall 库名

卸载pip：python -m pip uninstall pip

5.python解释器有哪些，常见的是哪个？你现在安装的是什么版本？如果项目中部分内容使用非python语言开发，要与python开发的内容进行交互，你建议采用什么方式并阐述相关理由？

pycharm, vscode, CPython, IPython, PyPy, IronPython等，常见的有pycharm, vscode

安装的为pycharm

建议使用python的扩展机制，可以方便的使用其他语言开发

6.python编辑器VS Code或者IDE的安装(pycharm)

7.变量的命名规则

- 命名规则三条是：1. 不能以数字开头，2.不能是python内置关键字，3.只能包含大小写字母，下划线，数字0-9
- 区分大小写
- 理解变量引用的概念
- 变量语法糖

```
a = b = 2    # 链式赋值
c = 3
b, c = c, b # 解包赋值
print(a, b, c)
结果: 2 3 2
```

8.输入输出

- input 传入的类型：字符串
- print 参数 sep="" end=""

9.理解编译器与解释器的概念，并简要阐述解释器执行python代码的过程。

编译器：把源代码转换成（翻译）低级语言的程序。

解释器：直接把高级编程语言一行一行转译运行。

python执行时不会一次把整个程序转译出来，因此运行速度比较缓慢，它每转译一行程序就立刻运行，然后再转译下一行，再运行，如此不停地进行下去。

10.理解高级语言、低级语言、机器语言、汇编语言、动态语言、静态语言、强类型、弱类型等概念

高级语言：我们现在大多数人使用的语言，如C、C++、Python、Java、Matlab、LabVIEW等等，都属于高级语言，相对于低级语言，它更接近于我们平时正常的人思维，其最大的特点是编写容易，代码可读性好。实现同样的功能，使用高级语言耗时更少，程序代码量更短，更容易阅读。其次，高级语言是可移植的，也就是说，仅需稍作修改甚至不用修改，就可将一段代码运行在不同类型的计算机上。

低级语言：泛指机器语言和汇编语言，其中，机器语言是计算机最原始的语言，由0和1的代码构成，计算机在工作的时候只认识机器语言，即0和1的代码；汇编语言，它用人类容易记忆的语言和符号来表示

一组0和1的代码，如AND表示加法助记符。相对于高级语言，其优点是执行速度快，但代码编写难度较大，可读性较差。另外，低级语言编写的程序只能在一种计算机上运行，想要运行在不同的机器上，必须重写。低级语言是早期的一种计算机编程语言，现在只在很少的特殊场景中使用了。

机器语言：器语言是计算机能直接运行的语言，是二进制语言，属于低级语言；

汇编语言：汇编语言是面向机器的低级语言，不能被机器直接识别，需要编译；

动态语言：动态语言（弱类型语言）是运行时才确定数据类型的语言，变量在使用之前无需申明类型，通常变量的值是被赋值的那个值的类型。比如PHP、ASP、JavaScript、Python、Perl等等。

静态语言：静态语言（强类型语言）是编译时变量的数据类型就可以确定的语言，大多数静态语言要求在使用变量之前必须申明数据类型。比如Java、C、C++、C#等。

强类型与弱类型：强类型语言是一种强制类型定义的语言，即一旦某一个变量被定义类型，如果不经强制转换，那么它永远就是该数据类型。而弱类型语言是一种弱类型定义的语言，某一个变量被定义类型，该变量可以根据环境变化自动进行转换，不需要经过强行强制转换。

11.理解对象的概念，对象三要素分别是什么？

封装 继承 多态

12.python 8类内置数据类型分别是什么？

Number, String, Boolean, None, List, Tuple, Dict, Set

13. Number类型包含哪三种类型？进行运算的类型主要有哪些？

三种类型分别为整型，浮点型，复数型，进行运算的类型主要有：加减乘除，整除，幂运算，取余运算。

14.字符串的索引:从0开始 且最终方向只能从左到右，不然将报错。

从左向右：开始为0

自右向左：开始为-1

```
str = "/usr/bin/python3"
#print(str[:])
#print(str[:])
#print(str[:5])
#print(str[0:])
#print(str[1:5:-1])
#print(str[-3:-1])
#print(str[::-1])

/usr/bin/python3
/usr/bin/python3
/usr/
/usr/bin/python3
报错
on
3nohtyp/nib/rsu/
```

15.简要叙述下Zen of Python这首诗的主要思想。

Python以编写优美的代码为目标，优美的代码应当是明了的，简洁的，优美的代码应当是扁平的，不能有太多的嵌套，优美的代码是可读的，这些原则要遵守

二.语法

1.输入输出

print()函数也可以接受多个字符串，用逗号","隔开，就可以连成一串输出。

print()会依次打印每个字符串，遇到逗号","会输出一个空格

Python提供了input()，可让用户输入字符串，并存放到变量里

2.对象与变量

Python中一切皆对象。

每个对象由:标识(identity, 对应对象在计算机中地址)、类型(type)、value (值组成)

对象的本质：一个内存块,拥有特定的值,支持特定类型的相关操作。

在Python中，变量也称为：对象的引用

变量存储的就是对象的地址

变量位于：栈内存（压栈出栈）

对象位于：堆内存

3.字符集

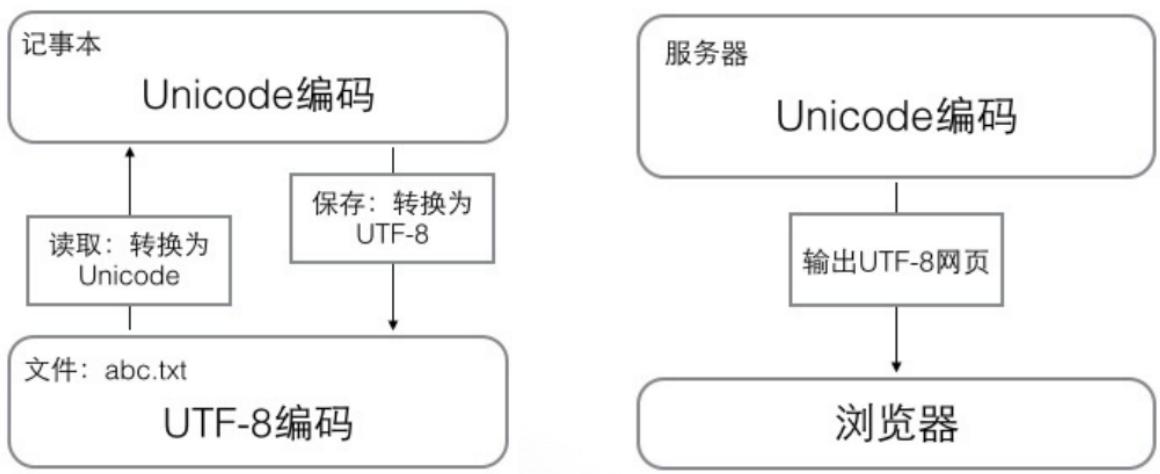
ASCII 编码：标准 ASCII 编码共收录了 128 个字符，其中包含了 33 个控制字符（具有某些特殊功能但是无法显示的字符）和 95 个可显示字符

GB2312编码；要处理中文显然一个字节是不够的，至少需要**两个字节**

多种语言时会出现乱码

Unicode字符集：Unicode标准也在不断发展，但最常用的是UCS-16编码，用**两个字节**表示一个字符（如果要用到非常偏僻的字符，就需要**4个字节**）

UTF-8编码：UTF-8编码把一个Unicode字符根据不同的数字大小编码成1-6个字节，常用的英文字母被编码成1个字节，汉字通常是3个字节，只有很生僻的字符才会被编码成4-6个字节。当要传输的文本包含大量英文字符，用UTF-8编码就能节省空间



4, 运算符

//:向下取整

```
9//2 = 4  
9.0//2.0 = 4.0,  
-11//3 = -4,  
-11.0//3 = -4.0
```

a = 0011 1100

b = 0000 1101

a&b = 0000 1100

a|b = 0011 1101

a^b = 0011 0001

~a = 1100 0011

5.is

```
a=10  
b=10  
if a is b:  
    print('a is b')  
else:  
    print('a is not b')
```

```
b=20  
if a is b:  
    print('a is b')  
else:  
    print('a is not b')
```

输出:

```
a is b  
a is not b
```

is:地址相等 值也相等

6.数据类型

- Number (整数、浮点数、复数)
- String
- Boolean 布尔 (True、False)
- None 空值
- list (列表)
- tuple (元组)
- dict (字典)
- set (集合)

P
x
x
0,
lo
ty
is

可变数据类型和不可变数据类型：

可变数据类型：list (列表) 、 dict (字典) 、 set (集合，不常用)

不可变数据类型：数值类型 (int、float、bool) 、 string (字符串) 、 tuple (元组)

可变数据类型：当该数据类型对应的变量的值发生了变化时，如果它对应的内存地址不发生改变，那么这个数据类型就是 可变数据类型。

不可变数据类型：当该数据类型对应的变量的值发生了变化时，如果它对应的内存地址发生了改变，那么这个数据类型就是 不可变数据类型。

####

和 list 比较， dict 有以下几个特点：

- 查找和插入的速度极快，不会随着 key 的增加而变慢；
- 需要占用大量的内存，内存浪费多。

而 list 相反：

- 查找和插入的时间随着元素的增加而增加；
- 占用空间小，浪费内存很少。

所以， dict 是用空间来换取时间的一种方法。

```
num=[1,2,2,2,3,4,2]
for i in num:
    print(i)
    if i == 2:
        num.remove(i)

print(num)
```

结果：

1
2

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

每次移除第一个二，迭代器后移

`lst[::-1]` 返回一个新列表，而 `list.reverse` 只是执行列表的逆转

tuple: 元组

当只有一个变量时，需要加逗号

```
a=(1,)
print(type(a))
```

bytes:

bytes 只负责以字节序列的形式（二进制形式）来存储数据，至于这些数据到底表示什么内容（字符串、数字、图片、音频等），完全由程序的解析方式决定。如果采用合适的字符编码方式（字符集），字节串可以恢复成字符串；反之亦然，字符串也可以转换成字节串。

encode: 编码

decode: 解码

str字符串默认编码为： **Unicode**

```
1 #通过构造函数创建空 bytes
2 b1 = bytes()
3 #通过空字符串创建空 bytes
4 b2 = b''
5
6 #通过b前缀将字符串转换成 bytes
7 b3 = b'http://c.biancheng.net/python/'  

8 print("b3: ", b3)
9 print(b3[3])
10 print(b3[7:22])
11
12 #为 bytes() 方法指定字符集
13 b4 = bytes('C语言中文网8岁了', encoding='UTF-8')
14 print("b4: ", b4)
15
16 #通过 encode() 方法将字符串转换成 bytes
17 b5 = "C语言中文网8岁了".encode('UTF-8')
18 print("b5: ", b5)
```

运行结果：

```
b3: b'http://c.biancheng.net/python/'
112
b'c.biancheng.net'
b4:
b'C\xe8\xaf\xad\xe8\xa8\x80\xe4\xb8\xad\xe6\x96\x87\xe7\xbd\x918\xe5\xb2\x81\xe4\xba\x86'
b5:
b'C\xe8\xaf\xad\xe8\xa8\x80\xe4\xb8\xad\xe6\x96\x87\xe7\xbd\x918\xe5\xb2\x81\xe4\xba\x86'
```



```
print(len('abc'))
print(len('中文'))
print(len(b'abc'))
print(len('中文'.encode('utf-8')))

3
2
3
6
```

while后else:

for / while 后面的else的执行条件:

1. 循环正常结束，会执行else后面的语句；
2. continue跳出当次循环，等循环结束，会执行else后面的语句；
3. break跳出循环，不会执行else后面的语句。

```
a=0
while a<5:
    a=a+1
    print(a)
else:
    print('a >=5')
```

结果:

```
1
2
3
4
5
a >=5
```

函数:

1. 定义函数时，需要确定函数名和参数个数；
2. 如果有必要，可以先对参数的数据类型做检查；
3. 函数体内部可以用return随时返回函数结果；
4. 函数执行完毕也没有return语句时，自动return None；
5. 函数可以同时返回多个值，但其实就是一个 tuple 。

易错题:

```
def a(l=[]):
    l.append('end')
    return l

print(a())
print(a())

['end']
['end', 'end']
```

```
def a(l=None):
    if l is None:
        l=[]
    l.append('end')
    return l
else:
    l.append('start')

print(a())
print(a())

['end']
['end']
```

函数的参数传递本质上就是：从实参到形参的赋值操作。Python中“一切皆对象”，所有的赋值操作都是“引用的赋值”。所以，Python中参数的传递都是“引用传递”，不是“值传递”。具体操作时分为两类：

1. 对“可变对象”进行“写操作”，直接作用于原对象本身。
2. 对“不可变对象”进行“写操作”，会产生一个新的“对象空间”，并用新的值填充这块空间。（起到其他语言的“值传递”效果，但不是“值传递”）

可变参数

```
def calc(*numbers):  
    sum = 0  
    for n in numbers:  
        sum = sum + n * n  
    return sum
```

```
nums = [1, 2, 3]  
calc(nums[0], nums[1], nums[2])  
calc(*nums)
```

关键字参数:

```
def person(name, age, **kw):  
    print('name:', name, 'age:', age, 'other:', kw)  
person('Michael', 30)  
name: Michael age: 30 other: {}  
person('Bob', 35, city='Beijing')  
name: Bob age: 35 other: {'city': 'Beijing'}  
person('Adam', 45, gender='M', job='Engineer')  
name: Adam age: 45 other: {'gender': 'M', 'job': 'Engineer'}  
extra = {'city': 'Beijing', 'job': 'Engineer'}  
person('Jack', 24, **extra)
```

命名关键字参数:

传入时候需要传入参数名，否则相当于位置参数

```
def f1(a, b, c=0, *args, **kw):  
    print('a =', a, 'b =', b, 'c =', c, 'args =', args, 'kw =', kw)  
  
def f2(a, b, c=0, *, d, **kw):  
    print('a =', a, 'b =', b, 'c =', c, 'd =', d, 'kw =', kw)
```

```
>>> f1(1, 2)
a = 1 b = 2 c = 0 args = () kw = {}
>>> f1(1, 2, c=3)
a = 1 b = 2 c = 3 args = () kw = {}
>>> f1(1, 2, 3, 'a', 'b')
a = 1 b = 2 c = 3 args = ('a', 'b') kw = {}
>>> f1(1, 2, 3, 'a', 'b', x=99)
a = 1 b = 2 c = 3 args = ('a', 'b') kw = {'x': 99}
>>> f2(1, 2, d=99, ext=None)
a = 1 b = 2 c = 0 d = 99 kw = {'ext': None}
```

```
>>> args = (1, 2, 3, 4)
>>> kw = {'d': 99, 'x': '#'}
>>> f1(*args, **kw)
a = 1 b = 2 c = 3 args = (4,) kw = {'d': 99, 'x': '#'}
>>> args = (1, 2, 3)
>>> kw = {'d': 88, 'x': '#'}
>>> f2(*args, **kw)
a = 1 b = 2 c = 3 d = 88 kw = {'x': '#'}
```

§ Python的函数具有非常灵活的参数形态，既可以实现简单的调用，又可以传入非常复杂的参数。

§ 默认参数一定要用不可变对象，如果是可变对象，程序运行时会有逻辑错误！

§ 要注意定义可变参数和关键字参数的语法：

§ *args是可变参数，args接收的是一个tuple；

§ **kw是关键字参数，kw接收的是一个dict。

调用函数时如何传入可变参数和关键字参数的语法：

§ 可变参数既可以直接传入：func(1, 2, 3)，又可以先组装list或tuple，再通过*args传入：func(*(1, 2, 3))；

§ 关键字参数既可以直接传入：func(a=1, b=2)，又可以先组装dict，再通过*kw传入：func(**{'a': 1, 'b': 2})。

§ 使用*args和**kw是Python的习惯写法，当然也可以用其他参数名，但最好使用习惯用法。

§ 命名的关键字参数是为了限制调用者可以传入的参数名，同时可以提供默认值。

§ 定义命名的关键字参数在没有可变参数的情况下不要忘了写分隔符*，否则定义的将是位置参数。

```
def a(*b):
    print(b)

s=[1,2,3]
a(s)

([1, 2, 3],)
```

作用域：

- L (Local) 局部作用域
- E (Enclosing) 闭包函数外的函数中
- G (Global) 全局作用域
- B (Built-in) 内建作用域

变量/函数的查找顺序：

L -> E -> G ->B

Python中关键字global与nonlocal的区别

一、功能不同。

- global关键字标识该变量是全局变量，对该变量进行修改就是修改全局变量。
- nonlocal关键字标识该变量是上一级函数中的局部变量，如果上一级函数中不存在该局部变量，nonlocal位置会发生错误（最上层的函数使用nonlocal修饰变量必定会报错）。

二、范围不同。

- global关键字可以用在任何地方，包括最上层函数中和嵌套函数中，即使之前未定义该变量，global修饰后也可以直接使用。
- nonlocal关键字只能用于嵌套函数中，并且外层函数中定义了相应的局部变量，否则会发生错误（见第一）。

输出下列代码的结果

```
def func():
    print('func', x, id(x))

x = '"hello main"'
print('main1', x, id(x))
func()
print('main2', x, id(x))
```

```
main1 "hello main" 1994848157232 # x为全局变量
func "hello main" 1994848157232
main2 "hello main" 1994848157232
```

1、在func函数中修改x的值

```
def func():
    x = 'hello func'
    print('func', x, id(x))

x = '"hello main"'
print('main1', x, id(x))
func()
print('main2', x, id(x))
```

```
main1 "hello main" 2434486386288 # 全局变量x
func hello func 2434486386352 # func函数中的x为局部变量
main2 "hello main" 2434486386288 # 没有改变全局变量x的值
```

函数内使用了与全局变量同名的变量，如果不对该变量赋值，那么该变量就是全局变量，如果对该变量进行赋值，那么该变量就是局部变量。

2、使用global关键字 试图在func函数内修改全局变量x

```
def func():
    global x = 'hello func'
    print('func', x, id(x))

x = '"hello main"'
print('main1', x, id(x))
func()
print('main2', x, id(x))
```

```
SyntaxError: invalid syntax # 语法错误，global修饰变量时不能直接赋值
```

```
def func():
    global x
    print('func1', x, id(x))
    x = 'hello func'
    print('func2', x, id(x))

x = '"hello main"'
print('main1', x, id(x))
func()
print('main2', x, id(x))
```

```
main1 "hello main" 2733145996976
func1 "hello main" 2733145996976 # 标记为全局变量
func2 hello func 2733145996784
main2 hello func 2733145996784 # 函数外x的值也被改变
```

global关键字修饰函数内部变量后标志其是全局变量，必须在修改该变量前进行修饰（否则会发生变量未定义的错误）。

3、nonlocal关键字

```
def func():
    nonlocal x
    print('func1', x, id(x))
    x = 'hello func'
    print('func2', x, id(x))

x = '"hello main"'
print('main1', x, id(x))
func()
print('main2', x, id(x))
```

SyntaxError: no binding for nonlocal 'x' found # 报错

```
def func(): # 定义一个嵌套函数
    print('func', x, id(x))
    def ifunc():
        print('ifunc', x, id(x))

    ifunc()

x = '"hello main"'
print('main1', x, id(x))
func()
print('main2', x, id(x))
```

```
main1 "hello main" 2506267245936
func "hello main" 2506267245936
ifunc "hello main" 2506267245936 # 嵌套函数内也默认使用全局变量
main2 "hello main" 2506267245936
```

```
def func():
    x = 'hello func' # 在func函数中修改x的值
    print('func1', x, id(x))
    def ifunc():
        print('ifunc', x, id(x))

    ifunc()
    print('func2', x, id(x))

x = '"hello main"'
print('main1', x, id(x))
func()
print('main2', x, id(x))
```

```
main1 "hello main" 2470774718384
func1 hello func 2470774718128 # 修改后，x被标识为局部变量
ifunc hello func 2470774718128 # ifunc中的x是func函数中的局部变量
func2 hello func 2470774718128
main2 "hello main" 2470774718384 # 没有改变全局变量x的值
```

```

def func():
    x = 'hello func'
    print('func1', x, id(x))
    def ifunc():
        x = 'hello ifunc' # 在ifunc中也修改x的值
        print('ifunc', x, id(x))

    ifunc()
    print('func2', x, id(x))

x = '"hello main"'
print('main1', x, id(x))
func()
print('main2', x, id(x))

```

```

main1 "hello main" 2027568054192
func1 hello func 2027568053872
ifunc hello ifunc 2027568053808 # ifunc修改x之后即没有影响func中的局部变量x，也没有影响全局变量x，ifunc中的x是函数ifunc自己的局部变量。
func2 hello func 2027568053872
main2 "hello main" 2027568054192

```

嵌套函数和函数中存在和全局变量同名的变量，如果直接使用，而不修改变量的值，那么这三个位置的变量使用的是同一个全局变量，如果在函数中修改了变量值，那么该变量会被标识为该函数的局部变量，嵌套函数直接使用时使用的是该函数的局部变量。如果在嵌套函数中修改同名变量的值，那么嵌套函数中的该变量会被标识为该嵌套函数的局部变量，它的修改不影响函数中同名变量和全局变量。

4、在嵌套函数中使用global关键字

```

def func():
    x = 'hello func'
    print('func1', x, id(x))
    def ifunc():
        global x
        print('ifunc', x, id(x))

    ifunc()
    print('func2', x, id(x))

x = '"hello main"'
print('main1', x, id(x))
func()
print('main2', x, id(x))

```

```

main1 "hello main" 1628658941744
func1 hello func 1628658941488
ifunc "hello main" 1628658941744 # ifunc中的x为全局变量
func2 hello func 1628658941488
main2 "hello main" 1628658941744

```

```

def func():
    x = 'hello func'

```

```

print('func1', x, id(x))
def ifunc():
    global x
    x = 'hello ifunc'
    print('ifunc', x, id(x))

ifunc()
print('func2', x,id(x))

x = '"hello main"'
print('main1', x, id(x))
func()
print('main2', x, id(x))

```

```

main1 "hello main" 1658151321456
func1 hello func 1658151321136
ifunc hello ifunc 1658151321072 # x为全局变量，修改全局变量的值
func2 hello func 1658151321136 # func函数中的x为局部变量，未修改
main2 hello ifunc 1658151321072 # 全局变量x的值被修改

```

5、在嵌套函数中使用nonlocal关键字

```

def func():
    x = 'hello func'
    print('func1', x, id(x))
    def ifunc():
        nonlocal x
        x = 'hello ifunc'
        print('ifunc', x, id(x))

    ifunc()
    print('func2', x,id(x))

x = '"hello main"'
print('main1', x, id(x))
func()
print('main2', x, id(x))

```

```

main1 "hello main" 2940482502512
func1 hello func 2940482502192
ifunc hello ifunc 2940482502128 # ifunc中的x和func中的x是同一个变量，修改x的值
func2 hello ifunc 2940482502128 # ifunc中修改x的值影响了func中的x
main2 "hello main" 2940482502512 # 没有改变全局变量x。

```

6、在嵌套函数中使用global与nonlocal关键字

```

def func():
    global x
    x = 'hello func'
    print('func1', x, id(x))
    def ifunc():
        x = 'hello ifunc'
        print('ifunc', x, id(x))

```

```
ifunc()
print('func2', x, id(x))

x = '"hello main"'
print('main1', x, id(x))
func()
print('main2', x, id(x))
```

```
main1 "hello main" 2861722230896
func1 hello func 2861722189552
ifunc hello ifunc 2861722189680 # ifunc中的x为局部变量
func2 hello func 2861722189552
main2 hello func 2861722189552
```

```
def func():
    global x
    x = 'hello func'
    print('func1', x, id(x))
    def ifunc():
        nonlocal x
        x = 'hello ifunc'
        print('ifunc', x, id(x))

    ifunc()
    print('func2', x, id(x))

x = '"hello main"'
print('main1', x, id(x))
func()
print('main2', x, id(x))
```

```
SyntaxError: no binding for nonlocal 'x' found # 运行时报错，没有为ifunc中的x找到绑定。
```

global可以在任何地方修饰变量，被global修饰的变量直接被标识为全局变量，对该变量修改会影响全局变量的值，但不影响函数中未被global修饰的同名变量（依然是局部变量）。
nonlocal只能在嵌套函数中使用，用于标识嵌套函数中的变量是包含该嵌套函数的函数中的同名变量，在嵌套函数中修改变量会影响函数中的变量。
如果在函数中使用global修饰了变量，那么在嵌套函数中用nonlocal修饰同名变量会发生报错，因为nonlocal表示该变量在函数中已经定义，但检查时因为同名变量被global修饰为全局变量，所以不存在同名的局部变量，从而导致错误。

```
# globals() : 以dict的方式存储所有全局变量
def foo():
    print("I am a func")

def bar():
    foo = "I am a string"
```

```
foo_dup = globals().get("foo")
foo_dup()

bar()

# locals(): 以dict的方式存储所有局部变量
other = "test"

def foobar():
    name = "MING"
    gender = "male"
    for key, value in locals().items():
        print(key, "=", value)

foobar()

I am a func
name = MING
gender = male
```

改变作用域：

▫ 关键字：global

将局部变量 变为全局变量

▫ 关键字：nonlocal

可以在闭包函数中，引用并使用闭包外部函数的变量（非全局）

■ 关键字global与nonlocal的区别

- 任何一层子函数，若直接使用全局变量且不对其改变的话，则共享全局变量的值；一旦子函数中改变该同名变量，则其降为该子函数所属的局部变量；
- global可以用于任何地方，声明变量为全局变量（声明时，不能同时赋值）；声明后再修改，则修改了全局变量的值；
- 而nonlocal的作用范围仅对于所在子函数的上一层函数中拥有的局部变量，必须在上层函数中已经定义过，且非全局变量，否则报错。

闭包（重点）：

闭包的定义：包含两层含义（嵌套函数， 内部函数引用外部函数的变量），严谨的概念除了前面讲到的内涵外，还包括外部函数返回内部函数名

- # 当某个函数被当成对象返回时，夹带了外部变量，就形成了一个闭包
- # 在一些语言中，在函数中可以（嵌套）定义另一个函数时，如果内部的函数引用了外部的函数的变量，则可能产生闭包。闭包可以用来在一个函数与一组“私有”变量之间创建关联关系。在给定函数被多次调用的过程中，这些私有变量能够保持其持久性。—— 维基百科
-

```
def a():
    name='abc'
    def b():
        print(name)
    return b

a()()
```

abc

```
def a():
    num=1
    def b():
        num=2
        print(num)
    print(num)
    return b

a()()
```

1
2
不能在函数b中使用num=num+1

```
def a():
    num=1
    def b():
        nonlocal num
        num=num+1
        print('b',num)
    print('a',num)
    return b

a()()
```

a 1
b 2

```
def func_3():
    list_funcs = []
    for i in range(1, 5):
        def wrapper():
            print(i)

    list_funcs.append(wrapper)
```

```
return list_funcs
```

```
func_3()[0]()
func_3()[1]()
```

```
4 4
i一直等于4
```

```
def func_4():
    list_funcs = []
    for i in range(1, 5):
        def wrapper(i):
            def inner():
                print(i)

        return inner

    list_funcs.append(wrapper(i))
    return list_funcs
```

```
func_4()[0]()
func_4()[1]()
func_4()[2]()
func_4()[3]()
1 2 3 4
```

```
def count_steps(original_steps=0):

    def wrapper(new_steps):
        nonlocal original_steps
        total_steps = original_steps + new_steps
        original_steps = total_steps
        return original_steps

    return wrapper

count_steps = count_steps(8)
print(count_steps(2))
print(count_steps(3))
print(count_steps(5))
print(count_steps(5))
```

```
# 闭包 = 函数 + 环境变量
```

```

# 现场
def curve_pre():
    a = 25
    b=12
    def curve(x):
        print(b)
        return a * x * x

    return curve

a = 10
f = curve_pre()
print(f(2))
print(f.__closure__)
print(f.__closure__[1].cell_contents)

12
100
(<cell at 0x000001849077B4F0: int object at 0x00007FF903EB09A0>, <cell at
0x000001849086C370: int object at 0x00007FF903EB0800>)
12

```

列表推导式：

列表推导式生成列表对象，语法如下：

[表达式 for item in 可迭代对象]
或者 : {表达式 for item in 可迭代对象 if 条件判断}

- **lambda表达式返回一个函数对象**

例子：

func = lambda x,y:x+y

func 相当于下面这个函数

def func(x,y):

return x+y

- **注意def是语句，而lambda是表达式**

- **[(lambda x:x*x)(x) for x in range(1,11)]**

高阶函数：

sorted () :

- **sorted ()**

```
list_tuples = [(1, 'byd' ),  
                (3, 'xiaopeng' ),  
                (2, 'tesla' ),  
                (4, 'weilai')]
```

```
listed_tuples = sorted(list_tuples, key=lambda x: x[1])
```

map () :

- **map(function_to_apply, list_of_inputs)**
- **function_to_apply:** 代表函数
- **list_of_inputs:** 代表输入序列
- **注意:** python3中 map函数返回的是迭代器

```
items = [1, 2, 3, 4, 5]
```

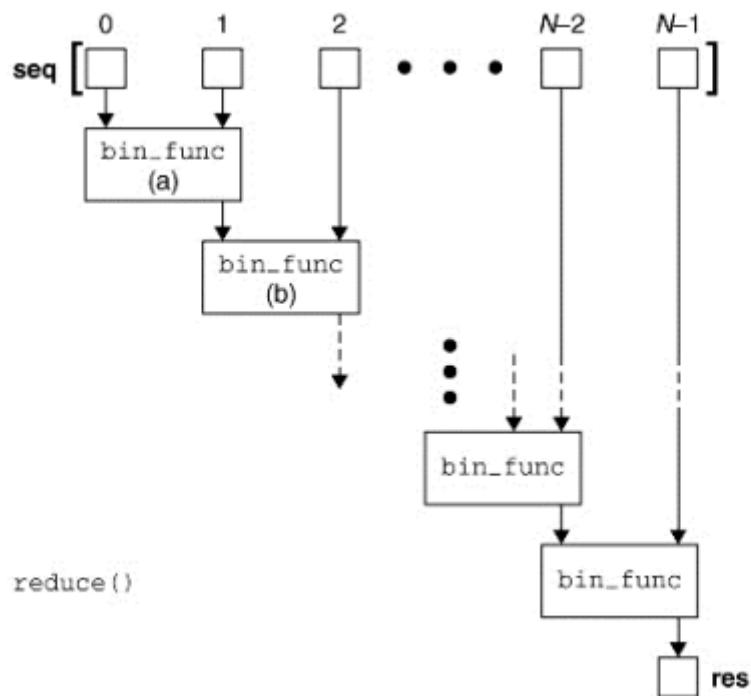
```
def f(x):  
    return x ** 2
```

```
squared = list(map(f, items))
```

```
items = [1, 2, 3, 4, 5]
```

```
squared = list(map(lambda x: x**2, items))
```

```
reduce () :
```



■ `reduce(function, iterable[, initializer])`

- `function`: 代表函数
- `iterable`: 序列
- `initializer`: 初始值 (可选)
- 与`map`不同, `reduce`不可以直接使用, 需要用`from functools import reduce`导入

```
def f(x, y):  
    return x * y
```

```
# 定义序列, 含1~10的元素  
items = range(1, 11)  
# 使用reduce方法  
result = reduce(f, items)  
print(result)
```

返回 10! 的结果

`filter () :`

- 用于过滤序列, 过滤掉不符合条件的元素, 返回一个迭代器对象
- `filter(function, iterable)`
 - `function` -- 判断函数。
 - `iterable` -- 可迭代对象。
- 接收两个参数, 第一个为函数, 第二个为序列, 序列的每个元素作为参数传递给函数进行判, 然后返回 True 或 False, 最后将返回 True 的元素放到新容器中。

```
def is_odd(n):
    return n % 2 == 1

f_list = filter(is_odd, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
print(list(f_list))
```

偏函数partial () :

```
int2 = partial(int, base=2)
print(int2('111000111'))
```

```
import functools

# 无法体会偏函数参数的位置问题，容易给人造成partial的第二个参数也是原函数的第二个参数的假象
def remainder(m, n):
    return m % n

print(remainder(100, 7)) # 2

# 使用偏函数的
new_rmd = functools.partial(remainder, 100)
print(new_rmd(7)) # 2
```

生成器：

保存的是算法，每次调用next(g)，就计算出g的下一个元素的值，直到计算到最后一个元素，没有更多的元素时，抛出StopIteration的错误

```
>>> L = [x * x for x in range(10)]
>>> L
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> g = (x * x for x in range(10))
>>> g
<generator object <genexpr> at 0x1022ef630>

>>> next(g)
0
>>> next(g)
1
>>> next(g)
4
>>> next(g)
9
>>> next(g)
16
```

```
>>> next(g)
25
>>> next(g)
36
>>> next(g)
49
>>> next(g)
64
>>> next(g)
81
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

或者

```
>>> g = (x * x for x in range(10))
>>> for n in g:
...     print(n)
...
0
1
4
9
16
25
36
49
64
81
```

函数生成器：

```
def fib(max):
    n, a, b = 0, 0, 1
    while n < max:
        print(b)
        a, b = b, a + b
        n = n + 1
    return 'done'

def fib(max):
    n, a, b = 0, 0, 1
    while n < max:
        yield b
        a, b = b, a + b
        n = n + 1
    return 'done'
```

如果一个函数定义中包含 `yield` 关键字，那么这个函数就不再是一个普通函数，而是一个 `generator` 函数，调用一个 `generator` 函数将返回一个 `generator: generator` 的函数，在每次调用 `next()` 的时候执行，遇到 `yield` 语句返回，再次执行时从上次返回的 `yield` 语句处继续执行。

```
def odd():
    print('step 1')
    yield 1
```

```
print('step 2')
yield(3)
print('step 3')
yield(5)
```

调用该generator函数时，首先要生成一个generator对象，然后用next()函数不断获得下一个返回值：

```
>>> o = odd()
>>> next(o)
step 1
1
>>> next(o)
step 2
3
>>> next(o)
step 3
5
>>> next(o)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

用for循环调用generator时，发现拿不到generator的return语句的返回值。如果想要拿到返回值，必须捕获StopIteration错误，返回值包含在StopIteration的value中：

```
>>> g = fib(6)
>>> while True:
...     try:
...         x = next(g)
...         print('g:', x)
...     except StopIteration as e:
...         print('Generator return value:', e.value)
...         break
...
g: 1
g: 1
g: 2
g: 3
g: 5
g: 8
Generator return value: done
```

装饰器：

在代码运行期间动态增加功能的方式，称之为“装饰器”（Decorator）。

- 写出下列代码运行结果，并改写为不使用@语法糖的版本

```
def get_time(func):
    def wrapper():
        func()
        print("used time: xxx")
    return wrapper

@get_time
def foo():
    print("foo")

foo()
```

运行结果: foo

used time:xxx

不使用语法糖:

```
def get_time(func):
    def wrapper():
        func()
        print("used time: xxx")

    return wrapper

def foo():
    print("foo")

foo=get_time(foo)

foo()
```

1. 写出下列代码运行结果

```
def f_a(func):
    print("f_a")

    def w_a():
        print(1)
        func()
    return w_a

def f_b(func):
    print("f_b")

    def w_b():
        print(2)
        func()
    return w_b

@f_a
@f_b
def f_c():
    print("f_c")
    print(3)

f_c()
```

结果: f_b

f_a
1
2
f_c
3

1. (被装饰函数接收参数) 写出下列代码运行结果

```
def decorator(func):
    def wrapper(*args):
        print("call function")
        return func(*args)
    return wrapper

@decorator
def foo(*args):
    print("this is foo")
    print(args)

foo(1, 2, 3)
```

结果:

```
call function
this is foo
(1, 2, 3)
```

1. (装饰器接收参数) 写出下列代码的运行结果

```
def func_out(info):
    def decorator(func):
        def wrapper():
            print(info)
            print("call function")
            return func()
        return wrapper
    return decorator

@func_out(info="hello!")
def foo():
    print("this is foo")

foo()
```

结果:

```
hello!
call function
this is foo
```

1. 编写代码实现一个装饰器，该装饰器具有为被装饰的函数实现统计函数运行时间的功能。

```
import time

def func(foo):
    def fun():
        start_time = time.time()
```

```
foo()
end_time = time.time()
ms = (end_time - start_time) * 1000
print('time is "{}" ms'.format(ms))
return fun

@func
def foo():
    print("hello world")
    time.sleep(1)

foo()
```

```
def log(func):
    def wrapper(*args, **kw):
        print('call %s():' % func.__name__)
        return func(*args, **kw)
    return wrapper
```

观察上面的`log`, 因为它是一个decorator, 所以接受一个函数作为参数, 并返回一个函数。我们要借助Python的@语法, 把decorator置于函数的定义处

```
@log (相当于 now = log(now))
def now():
    print('2015-3-25')

>>> now()
call now():
2015-3-25
```

多个装饰器执行的顺序就是从最后一个装饰器开始, 执行到第一个装饰器, 再执行函数本身。

```
def dec1(func):
    print("1111")
    def one():
        print("2222")
        func()
        print("3333")
    return one

def dec2(func):
    print("aaaa")
    def two():
        print("bbbb")
        func()
        print("cccc")
    return two

@dec1
@dec2
def test():
    print("test test")
```

```
test()
```

```
aaaa  
1111  
2222  
bbbb  
test test  
cccc  
3333
```

```
def a(fn):  
    def w():  
        return "a"+fn()+'/a'  
    return w  
  
def b(fn):  
    def w():  
        return 'b'+fn()+'/b'  
    return w  
  
@a  
@b  
def hello():  
    return 'hello'  
  
abhello/b/a
```

@wraps(view_func)的作用:

不改变使用装饰器原有函数的结构(如name, doc)

```
def decorator(func):  
    """this is decorator __doc__"""  
    def wrapper(*args, **kwargs):  
        """this is wrapper __doc__"""  
        print("this is wrapper method")  
        return func(*args, **kwargs)  
    return wrapper  
  
@decorator  
def test():  
    """this is test __doc__"""  
    print("this is test method")  
  
print("__name__: ", test.__name__)  
print("__doc__: ", test.__doc__)  
=====  
结果:  
__name__:  wrapper  
  
__doc__:  this is wrapper __doc__  
=====
```

```

from functools import wraps
def decorator(func):
    """this is decorator __doc__"""
    @wraps(func)
    def wrapper(*args, **kwargs):
        """this is wrapper __doc__"""
        print("this is wrapper method")
        return func(*args, **kwargs)
    return wrapper

@decorator
def test():
    """this is test __doc__"""
    print("this is test method")

print("__name__: ", test.__name__)
print("__doc__: ", test.__doc__)
"""

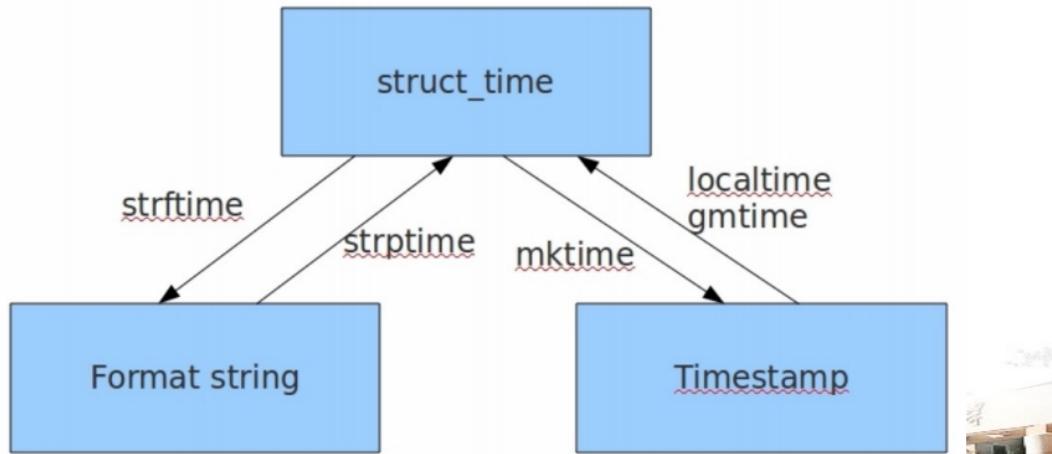
结果:
__name__:  test

__doc__:  this is test __doc__
"""

```

时间:

在Python中，通常有这三种方式来表示时间：时间戳、元组(struct_time)、格式化的时间字符串



格式	含义
%a	本地 (locale) 简化星期名称
%A	本地完整星期名称
%b	本地简化月份名称
%B	本地完整月份名称
%c	本地相应的日期和时间表示
%d	一个月中的第几天 (01 - 31)
%H	一天中的第几个小时 (24小时制, 00 - 23)
%I	第几个小时 (12小时制, 01 - 12)
%j	一年中的第几天 (001 - 366)
%m	月份 (01 - 12)
%M	分钟数 (00 - 59)
%p	本地am或者pm的相应符
%S	秒 (01 - 61)
%U	一年中的星期数。 (00 - 53星期天是一个星期的开始。) 第一个星期天之前的所有天数都放在第0周。
%w	一个星期中的第几天 (0 - 6, 0是星期天)
%W	和%U基本相同, 不同的是%W以星期一为一个星期的开始。
%x	本地相应日期
%X	本地相应时间
%y	去掉世纪的年份 (00 - 99)
%Y	完整的年份
%Z	时区的名字 (如果不存在为空字符串)
%%	'%'字符

time

编写代码实现效果：

1. 获取 UNIX 时间戳，输出（带上单位，秒？微秒？毫秒？），使用 ctime 获取本地日期时间字符串，输出

```
import time

time01=time.time()
print(time01)
print(time.ctime(time01))
```

输出：
1636715634.7844768 s
Fri Nov 12 19:13:54 2021

2. 由 UNIX 时间戳，使用 gmtime 获取 UTC/GMT struct_time 对象，输出

```
import time

time01=time.time()
print(time.gmtime(time01))

输出：
time.struct_time(tm_year=2021, tm_mon=11, tm_mday=12, tm_hour=11, tm_min=15,
tm_sec=16, tm_wday=4, tm_yday=316, tm_isdst=0)
```

1. 由 2 获得的 struct_time 对象，使用 asctime 获取简单格式化的日期时间字符串，输出

```
import time

time01=time.time()
time02=time.gmtime(time01)
time03=time.asctime(time02)
print(time03)

输出：
Fri Nov 12 11:16:55 2021
```

2. 由 2 获得的 struct_time 对象，使用 strftime 获取格式为 'Sat Mar 28 22:24:24 2016' (星期, 月, 日, 时, 分, 秒, 年) 的日期时间字符串，输出

```
import time

time01=time.time()
time02=time.gmtime(time01)
time03=time.strftime('%a %m %d %H:%M:%S %Y',time02)
print(time03)

输出：
Fri 11 12 11:30:44 2021
```

3. 由 4 获得的日期时间字符串，使用 strptime 获取 struct_time 对象，输出

```
import time

time01=time.time()
time02=time.gmtime(time01)
time03=time.strftime('%a %m %d %H:%M:%S %Y',time02)
time04=time.strptime(time03, '%a %m %d %H:%M:%S %Y')
print(time04)
```

输出：

```
time.struct_time(tm_year=2021, tm_mon=11, tm_mday=12, tm_hour=11, tm_min=33,
tm_sec=27, tm_wday=4, tm_yday=316, tm_isdst=-1)
```

4. 由 5 获得的 struct_time 对象，使用 calendar.timegm() 获取 UNIX 时间戳，输出（带上单位，秒？微秒？毫秒？），输出

```
import calendar

import time

time01=time.time()
time02=time.gmtime(time01)
time03=time.strftime('%a %m %d %H:%M:%S %Y',time02)
time04=time.strptime(time03, '%a %m %d %H:%M:%S %Y')
time05=calendar.timegm(time04)
print(time05)
```

输出：

```
1636717021 秒
```

5. 由 UNIX 时间戳，使用 localtime 获取本地时间 struct_time 对象，输出

```
import time

time01=time.time()
time02=time.localtime(time01)
print(time02)
```

输出：

```
time.struct_time(tm_year=2021, tm_mon=11, tm_mday=12, tm_hour=19, tm_min=38,
tm_sec=2, tm_wday=4, tm_yday=316, tm_isdst=0)
```

6. 由 7 获得的 struct_time 对象，使用 asctime 获取简单格式化的日期时间字符串，输出

```
import time

time01=time.time()
time02=time.localtime(time01)
time03=time.asctime(time02)
print(time03)
```

输出：
Fri Nov 12 20:05:54 2021

7. 由 7 获得的 struct_time 对象，使用 strftime 获取格式为 'Sat Mar 28 22:24:24 2016'（星期，月，日，时，分，秒，年）的日期时间字符串，输出

```
import time

time01=time.time()
time02=time.localtime(time01)

time03=time.strftime('%a %m %d %H:%M:%S %Y',time02)
print(time03)
```

输出：
Fri 11 12 20:07:44 2021

8. 由 9 获得的日期时间字符串，使用 strptime 获取 struct_time 对象，输出

```
import time

time01=time.time()
time02=time.localtime(time01)

time03=time.strftime('%a %m %d %H:%M:%S %Y',time02)
time04=time.strptime(time03, '%a %m %d %H:%M:%S %Y')
print(time04)

输出：
time.struct_time(tm_year=2021, tm_mon=11, tm_mday=12, tm_hour=20, tm_min=8,
tm_sec=36, tm_wday=4, tm_yday=316, tm_isdst=-1)
```

9. 由 10 获得的 struct_time 对象，使用 mktime 获取 UNIX 时间戳，输出（带上单位，秒？微秒？毫秒？），输出

```
import time

time01=time.time()
time02=time.localtime(time01)

time03=time.strftime('%a %m %d %H:%M:%S %Y',time02)
time04=time.strptime(time03, '%a %m %d %H:%M:%S %Y')
time05=time.mktime(time04)
print(time05)
```

输出：
1636718960.0 秒

10. 计算本地时间和 UTC 时间的时差，输出（带上单位，x 小时 x 分 x 秒），输出

```
import time

time01=time.time()
time02=time.localtime(time01)
time03=time.gmtime(time01)
h=time02.tm_hour-time03.tm_hour
m=time02.tm_min-time03.tm_min
s=time02.tm_sec-time03.tm_sec
print(h,'时：',m,'分：',s,'秒')
```

输出：
8 时： 0 分： 0 秒

datetime

1. 使用 now 函数，获取当前的 datetime，输出

```
import datetime

print(datetime.datetime.now())
```

输出：
2021-11-12 20:21:37.318099

2. 由 1 获得的 datetime 对象，获取当前的时间戳，输出（带单位）

```
import datetime

d=datetime.datetime.now()
print(d.timestamp())
输出：
1636719881.89436
```

3. 由 2 获得的时间戳，获得本地（不是UTC） datetime，输出

```
import datetime

d=datetime.datetime.now()
d1=d.timestamp()
d2=datetime.datetime.fromtimestamp(d1)
print(d2)
```

输出: 2021-11-12 20:26:05.036269

4. 生成 1 小时 40 分的 timedelta 对象, 输出

```
import datetime

d2=datetime.timedelta(0,0,0,0,40,1)
print(d2)
输出:
1:40:00
```

5. 由 3 获得的 datetime 对象和 4 获得的 timedelta 对象, 计算 1 小时 40 分后的 datetime 对象, 输出

```
import datetime

d1=datetime.datetime.now()
d2=d1+datetime.timedelta(0,0,0,0,40,1)
print(d2)
输出:
2021-11-12 22:11:02.393903
```

6. 将 5 获得的新的 datetime 对象, 设置时区为北京时间, 使用函数 astimezone 转换为京东时区, 输出

```
from datetime import datetime, timezone, timedelta

utc_time = datetime.utcnow() # 获取当前 UTC 时间
print(f'UTC时间为: {utc_time}')
local_time = datetime.now() # 获取当前本地时间
print(f'本地时间为: {local_time}')

utc = timezone.utc # 获取 UTC 的时区对象
utc_time = datetime.utcnow().replace(tzinfo=utc) # 强制转换加上 UTC 时区。此处敲黑板, 需要特别注意。
# replace的tzinfo参数为时区对象, 所以
# 也可以这样 replace(tzinfo=timezone(timedelta(hours=0)))
print(f'1、强制更改后的UTC时间为: {utc_time}')
Tokyo = timezone(timedelta(hours=9))

beijing = timezone(timedelta(hours=8))
```

```

time_beijing = utc_time.astimezone(beijing)

time_tokyo = utc_time.astimezone(Tokyo)
print('2、更改时区为北京后的时间: ', time_beijing)
print('3、获取时区信息: ', time_beijing.tzinfo)
print('4、更改时区为东京后的时间: ', time_tokyo)
print('5、获取时区信息: ', time_tokyo.tzinfo)

输出:
UTC时间为: 2021-11-12 12:42:29.572502
本地时间为: 2021-11-12 20:42:29.572502
1、强制更改后的UTC时间为: 2021-11-12 12:42:29.572502+00:00
2、更改时区为北京后的时间: 2021-11-12 20:42:29.572502+08:00
3、获取时区信息: UTC+08:00
4、更改时区为东京后的时间: 2021-11-12 21:42:29.572502+09:00
5、获取时区信息: UTC+09:00

```

进程已结束，退出代码为 0

面向对象编程：

类的定义：

```

class per:
    pass

p=per()

```

p是类per的实例
per()称为实例化

属性方法：

```

.....
实例能访问其类属性
•类无法访问实例属性
类方法：有 classmethod 装饰的函数
实例方法：没有任何装饰器的普通函数
静态方法：有 staticmethod 装饰的函数

```

普通函数（未定义在类里）和静态方法，都是函数（`function`） • 实例方法和类方法，都是方法（`method`）

```

class person:
    #类属性
    unit='NKU'

    #类方法
    @classmethod
    def c1(cls):

```

```

    pass
    #静态方法
    @staticmethod
    def s():
        pass

    #实例方法
    def a(self):
        pass

    def __init__(self, name):
        #实例属性
        self.name=name

```

私有属性和方法：

Python的私有属性和方法的外部访问

外部可以通过：_类名属性名 或者 _类名方法名 来直接访问

同类中的私有属性和私有方法

继承：

python继承可以继承父类私有属性，但不能使用

mro算法：

```

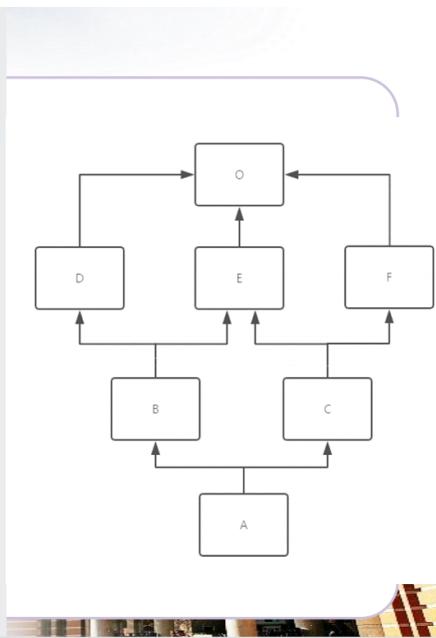
原式= [A] + merge( mro(B),mro(C),[B,C] )

mro(B) = mro( B(D,E) )
= [B] + merge( mro(D), mro(E), [D,E] ) # 多继承
= [B] + merge( [D,O] , [E,O] , [D,E] ) # 单继承mro(D(O))=[D,O]
= [B,D] + merge( [O] , [E,O] , [E] ) # 拿出并删除D
= [B,D,E] + merge([O] , [O])
= [B,D,E,O]

mro(C) = mro( C(E,F) )
= [C] + merge( mro(E), mro(F), [E,F] )
= [C] + merge( [E,O] , [F,O] , [E,F] )
= [C,E] + merge( [O] , [F,O] , [F] ) # 跳过O, 拿出并删除
= [C,E,F] + merge([O] , [O])
= [C,E,F,O]

原式= [A] + merge( [B,D,E,O], [C,E,F,O], [B,C] )
= [A,B] + merge( [D,E,O], [C,E,F,O], [C] )
= [A,B,D] + merge( [E,O], [C,E,F,O], [C] ) # 跳过E
= [A,B,D,C] + merge([E,O], [E,F,O])
= [A,B,D,C,E] + merge([O], [F,O]) # 跳过O
= [A,B,D,C,E,F] + merge([O], [O])
= [A,B,D,C,E,F,O]

```



super () :

super() 是用来解决多重继承问题的，直接用类名调用父类方法在使用单继承的时候没问题，但如果使用多继承，会涉及到查找顺序（MRO）、重复调用（钻石继承）等种种问题

- super() 语法: super(type[, object-or-type])

- type -- 类

- object-or-type - 一般是 self

```
class A:  
    def test(self):  
        print('A', self)
```

```
class B:  
    def test(self):  
        print('B', self)
```

A
A
B

```
class C(A, B):  
    def __init__(self):  
        super().test() # 调用A类中的test方法  
        super(C, self).test() # 调用A类中的test方法  
        super(A, self).test() # 调用B类中的test方法
```

C()

super调用时找到第一个参数后面的mro调用

```
class Base(object):
    def __init__(self):
        print("enter Base")
        print("leave Base")

class A(Base):
    def __init__(self):
        print("enter A")
        super(A, self).__init__()
        print("leave A")

class B(Base):
    def __init__(self):
        print("enter B")
        super(B, self).__init__()
        print("leave B")

class C(A, B):
    def __init__(self):
        print("enter C")
        super(C, self).__init__()
        print("leave C")

c = C()
```

D e s i g n

```
enter C
enter A
enter B
enter Base
leave Base
leave B
leave A
leave C
```

```
class goat(object):
    def __init__(self):
        self.name='乔丹'
        print(self.name)

    def god_goat(self):
        print(self.name,'篮球之神')

class mamba(goat):
    def __init__(self):
        self.name='科比'
        print(self.name)
        super().__init__()
        super().god_goat()

    def god_mamba(self):
        print(self.name,'曼巴精神')

class king(mamba):
    def __init__(self):
        self.name='詹姆斯'
        print(self.name)
```

```
def god(self):
    print(self.name)
    super().__init__()
    super().god_mamba()
```

```
k=king()
k.god()
```

詹姆斯
科比
乔丹
乔丹 篮球之神
乔丹 曼巴精神

运行时修改self.name的值

多态：

- 多态发生的条件：
 - 继承：发生在父类和子类之间
 - 重写：子类重写父类的方法
- 多态的作用：
 - 增加程序的灵活性
 - 增加程序的扩展性

抽象类：

抽象类是特殊的类，只能被继承，不能被实例化

- 抽象类的作用
 - 在不同的模块中通过抽象基类来调用，可以用最精简的方式展示出代码之间的逻辑关系，让模块之间的依赖清晰、简单
 - 提供了逻辑和实现解耦的能力
- abc模块：@abstractmethod、ABCMeta、ABC

@property

Python内置的@property装饰器就是负责把一个方法变成属性调用的

```
class Student(object):
    @property
```

```

def score(self):
    return self._score

@score.setter
def score(self, value):
    if not isinstance(value, int):
        raise ValueError('score must be an integer!')
    if value < 0 or value > 100:
        raise ValueError('score must between 0 ~ 100!')
    self._score = value

```

`@property`的实现比较复杂，我们先考察如何使用。把一个`getter`方法变成属性，只需要加上`@property`就可以了，此时，`@property`本身又创建了另一个装饰器`@score.setter`，负责把一个`setter`方法变成属性赋值，于是，我们就拥有一个可控的属性操作：

```

>>> s = Student()
>>> s.score = 60 # OK, 实际转化为s.set_score(60)
>>> s.score # OK, 实际转化为s.get_score()
60
>>> s.score = 9999
Traceback (most recent call last):
...
ValueError: score must between 0 ~ 100!

```

还可以定义只读属性，只定义`getter`方法，不定义`setter`方法就是一个只读属性：

判断类实例的函数：

`type()` 和 `isinstance()` 函数的区别

- **`isinstance(obj, class_)`**: 判断obj 是否是 class_类或者其子类的实例
 - **`type(obj)`**: 获取实例obj的类型
 - 与**`class`**作用相同
 - 知识拓展： **`type()`函数既可以返回一个对象的类型，也可以创建出新的类型**
- `type()`函数创建类参考元类相关知识点**

元类：

元类：

- 通过继承type创建元类

- 通过 metaclass=

声明类的创建方式

- 通过继承type之后,

可以修改创建类的方式 (override)

- type继承object类

```
class ModelMeta(type):
    def __init__(cls, *args, **kwargs):
        super().__init__(*args, **kwargs)
        print("init a", cls)
        print("init a", cls.__class__.__name__)

class Model(object, metaclass=ModelMeta):
    pass

init a <class '__main__.Model'>
init a ModelMeta

print(Model.__mro__)
print(ModelMeta.__mro__)

(<class '__main__.Model'>, <class 'object'>)
(<class '__main__.ModelMeta'>, <class 'type'>, <class 'object'>)
```

元类：

- 通过继承type创建元类

```
class ModelMeta(type):
    def __init__(cls, *args, **kwargs):
        super().__init__(*args, **kwargs)
        print("init a", cls)
        print("init a", cls.__class__.__name__)

class Model(object, metaclass=ModelMeta):
    pass

def func(self): pass

Foo = type("Foo", (object,), {"count": 0, "func": func})
Foo = ModelMeta("Foo", (), {"count": 0, "func": func})
```

协议编程：

- Dunders(Double UNDERscore): 被双下划线包围的属性名和方法名

- **__new__(cls)**: 构造方法，创建并返回一个实例对象。

- __new__方法调用一次，就会得到一个对象。
 - 继承自object的新式类才有__new__这一方法
 - __new__方法至少必须要有一个参数cls，代表要实例化的类，此参数在实例化时由Python解释器自动提供
 - __new__必须要有返回值，返回实例化出来的实例（很重要），这点在自己实现__new__时要特别注意，可以return父类__new__出来的实例，或者直接是object的__new__出来的实例，若__new__没有正确返回当前类cls的实例，__init__方法不会被调用，即使是父类的实例也不行。

```

class A:
    pass

class B(A):
    def __init__(self):
        print("__init__被调用")

    def __new__(cls, *args, **kwargs):
        print("__new__被调用")
        print("cls: ", cls, id(cls))
        return object.__new__(A)

```

`__new__` 没有正确返回当前类 `cls` 的实例，
`__init__` 方法不会被调用，即使是父类也不行

```

b = B()
print("b: ", b)
print("A: ", A, id(A))
print("B: ", B, id(B))

```

`__new__` 被调用
`cls: <class '__main__.B'> 1483397664288`
`b: <__main__.B object at 0x000001596176B610>`
`A: <class '__main__.A'> 1483397663328`
`B: <class '__main__.B'> 1483397664288`

```

class A:
    pass

class B(A):
    def __init__(self):
        print("__init__被调用")

    def __new__(cls, *args, **kwargs):
        print("__new__被调用")
        print("cls: ", cls, id(cls))
        return object.__new__(B)

```

```

b = B()
print("b: ", b)
print("A: ", A, id(A))
print("B: ", B, id(B))

```

`__new__` 被调用
`cls: <class '__main__.B'> 2160074773280`
`__init__` 被调用
`b: <__main__.B object at 0x000001F6EF08B610>`
`A: <class '__main__.A'> 2160074769440`
`B: <class '__main__.B'> 2160074773280`

补充知识：

- 单例模式 (Singleton)
 - 涉及到一个单一的类，该类负责创建自己的对象，同时确保只有单个对象被创建
 - 主要解决：一个全局使用的类，频繁地创建与销毁实例（如页面缓存），节省系统资源（如写文件操作）
 - 涉及到的Python高级语法知识：**装饰器、元类、`__new__`、`__call__`、`super`** 等
- 单例模式的写法
 - 总体思路：判断是否已有这个单例，如果有则返回，如果没有则创建。
 - 使用装饰器、使用关键字**两大类**

协议编程：

- Dunders(Double UNDERscore): 被双下划线包围的属性名和方法名
 - **`_init_(self)`**: 初始化方法, 在创建实例对象(`_new_`)之后为其赋值时使用。`_init_`必须至少有一个参数`self`, 即`_new_`返回的实例, `_init_`不需要返回值。
 - **`_del_(self)`**: 析构方法, 对应命令`del`, 即对象被垃圾回收的时候, 自动调用该方法, 以释放资源。除非有特殊要求, 一般不要重写。例如在关闭数据库连接对象、未保存数据的时候, 可以在释放资源。
 - 实例被创建以后, 该实例会被自动加上`_class_`、`_dict_`两个属性:

协议编程：

- Dunders(Double UNDERscore): 被双下划线包围的属性名和方法名
 - `_class_`: 获取对象所属的类(对象.`_class_`)
 - `_base_`: 获取类的父类(类.`_base_`)
 - `_bases_`: 获取类的父类元组(多继承, 类.`_bases_`)
 - `_mro_`: 获取类继承关系的元组(类.`_mro_`, 也可以使用这种形式类.`mro()`)
 - `_subclasses_`: 获取子类的列表(类.`_subclasses_`)

协议编程：

- Dunders(Double UNDERscore): 被双下划线包围的属性名和方法名
 - `_dict_`: 获取类或者实例属性字典
 - 类名.`_dict_`会输出包含类中所有的类属性的字典
 - 实例.`_dict_`会输出包含实例属性的字典
 - 实例的`_dict_`可通过字典的形式修改属性值: 实例.`_dict_[属性名] = "新属性值"`
 - 类的`_dict_`只读, 不可修改属性值

```
a=1
b='abc'
print(type(1))
print(type(int))
print(type(b))
print(type(str))

class st:
```

```

pass

class my(st):
    pass

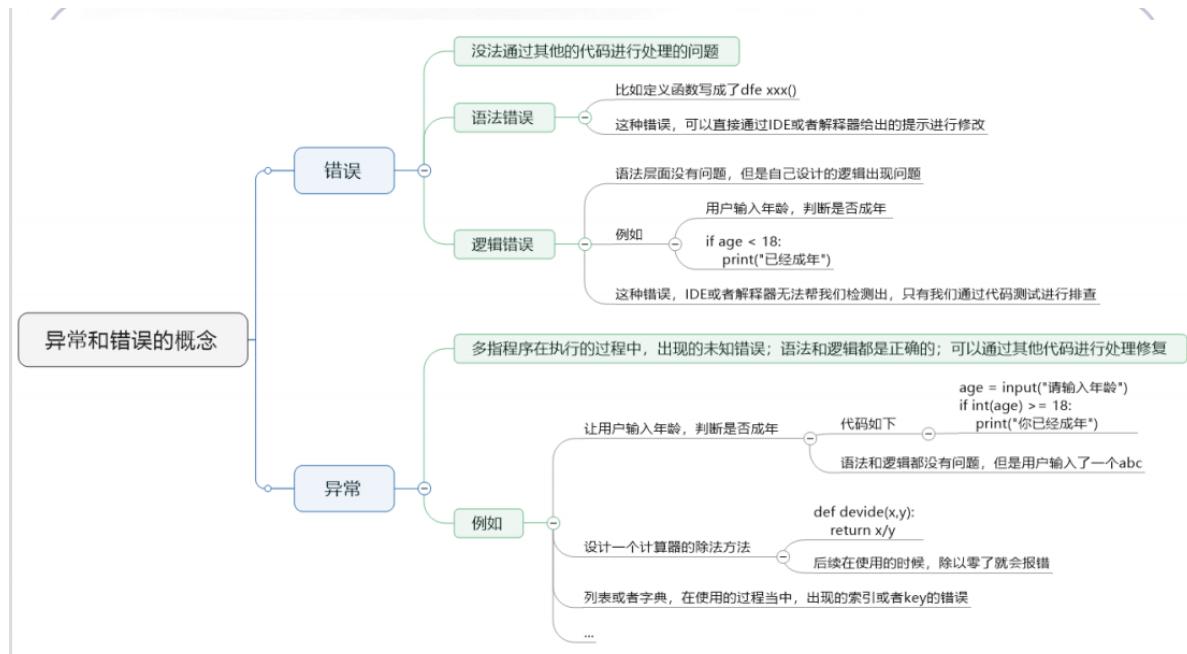
s=st()
print(type(s))
print(type(st))
print(int.__bases__)
print(str.__bases__)
print(st.__bases__)
print(my.__bases__)
print(type.__bases__)
print(object.__bases__)
print(type(object))

<class 'int'>
<class 'type'>
<class 'str'>
<class 'type'>
<class '__main__.st'>
<class 'type'>
(<class 'object'>,)
(<class 'object'>,)
(<class 'object'>,)
(<class '__main__.st'>,)
(<class 'object'>,)
()
<class 'type'>

```

进程已结束，退出代码为 0

异常：



```
def func():
```

```
try:  
    return 123  
except:  
    return 111  
else:  
    print(222)  
finally:  
    print(321)  
print(func())
```

321

123

首先执行try内语句，返回123，此时已经return，不再执行except和else中语句，然后执行finally语句打印321，最后print func () 的值123

拷贝：

可变对象：创建之后，可以改变的对象。set（集合）、list（列表）、dict（字典）

• 不可变对象：创建之后无法改变的对象。像str、bytes、int、float、bool、complex、tuple、frozenset

```
test_list_a = [12, 34, 56]  
test_list_c = test_list_a  
print(id(test_list_a))  
print(id(test_list_c))
```

2033168746184

2033168746184

```
test_list_a = [12, 34, 56]  
test_list_c = test_list_a  
print(id(test_list_a))  
print(id(test_list_c))  
test_list_a[0] = 0  
print(test_list_a)  
print(id(test_list_a))  
print(id(test_list_c))
```

2628103309000
2628103309000
[0, 34, 56]
2628103309000
2628103309000

```
import copy

L1 = [1, 2, [3]]
L2 = L1
L3 = L1[:]
L4 = list(L1)
L5 = copy.copy(L1)
L6 = copy.deepcopy(L1)
```

```
L1 id is :4360013824
L1[2] id is :4358958656
L3 id is :4360669376
L3[2] id is :4358958656
L4 id is :4360710784
L4[2] id is :4358958656
L5 id is :4360668864
L5[2] id is :4358958656
L6 id is :4360712128
L6[2] id is :4360709504
```



浅拷贝：

浅拷贝 (shallow copy) :

- 浅拷贝会创建新对象，其内容非原对象本身的引用，而是原对象内第一层对象的引用。
(拷贝组合对象，不拷贝子对象)
- 常见的浅拷贝有：切片操作、工厂函数、对象的copy()方法、copy模块中的copy函数
- **浅拷贝和深拷贝的不同仅是对组合对象来说，所谓的组合对象就是包含了其它对象的对象，如列表，类实例。而对于数字、字符串以及其它“原子”类型，没有拷贝一说，产生**的都是原对象的引用****

深拷贝：

深拷贝 (deep copy) :

- 是指创建一个新的对象，然后递归的拷贝原对象所包含的子对象
- 它的时间和空间开销要高
- 深拷贝出来的对象与原对象没有任何关联

数据结构操作方法：

Python 字符串方法

- isalpha()

如果字符串中的所有字符都是字母，并且至少有一个字符，返回 True，否则返回 False。

```
>>> ''.isalpha()
False
>>> 'ABC'.isalpha()
True
>>> 'ABC123'.isalpha()
False
```

数字: Unicode 数字, 全角数字(双字节), 罗马数字, 汉字数字

汉字数

字: "〇", "零", "一", "壹", "二", "弌", "三", "參", "四", "五", "六", "七", "八", "九", "十", "廿", "卅", "卅", "百", "千", "万", "万", "亿"

- `isalnum()`

如果字符串中的所有字符都是字母或数字且至少有一个字符, 则返回 `True`, 否则返回 `False`。如果 `c.isalpha()`, `c.isdecimal()`, `c.isdigit()`, 或 `c.isnumeric()` 之中有一个返回 `True`, 则字符c是字母或数字。

```
>>> ''.isalnum()
False
>>> 'ABC'.isalnum()
True
>>> '123'.isalnum()
True
>>> 'ABC123...'.isalnum()
False
```

- `isnumeric()`

如果字符串中至少有一个字符且所有字符均为数值字符则返回 `True`, 否则返回 `False`。

```
>>> ''.isnumeric()
False
>>> 'ABC'.isnumeric()
False
>>> '1'.isnumeric()
True
>>> '1'.isnumeric()
True
>>> 'I'.isnumeric()
False
>>> '一'.isnumeric()
True
```

- `isdigit()`

如果字符串中的所有字符都是数字, 并且至少有一个字符, 返回 `True`, 否则返回 `False`。数字包括十进制字符和需要特殊处理的数字, 如兼容性上标数字。

```
>>> '1'.isdigit()
True
>>> '一'.isdigit() # 全角字符
True
>>> 'I'.isdigit()
False
>>> '一'.isdigit()
False
```

- `isdecimal()`

如果字符串中的所有字符都是十进制字符且该字符串至少有一个字符, 则返回 `True`, 否则返回 `False`。

```
>>> '1'.isdecimal()
True
>>> '1'.isdecimal()
True
>>> 'I'.isdecimal()
False
>>> '—'.isdecimal()
False
```

- `isupper()`

如果字符串中至少有一个区分大小写的字符且此类字符均为大写则返回 `True`，否则返回 `False`。

```
>>> 'Upper'.isupper()
False
>>> 'UPPER'.isupper()
True
>>> 'UPPER123'.isupper()
True
```

- `islower()`

如果字符串中至少有一个区分大小写的字符且此类字符均为小写则返回 `True`，否则返回 `False`。

```
>>> 'Lower'.islower()
False
>>> 'lower'.islower()
True
>>> 'lower123'.islower()
True
>>> 'lower123...'.islower()
True
```

- `istitle()`

如果字符串中至少有一个字符且为标题字符串则返回 `True`，例如大写字符之后只能带非大写字符而小写字符必须有大写字符打头。否则返回 `False`。

```
>>> 'This Is Title'.istitle()
True
>>> 'This Is not Title'.istitle()
False
```

- `isspace()`

如果字符串中只有空白字符且至少有一个字符则返回 `True`，否则返回 `False`。

```
>>> ' '.isspace()
True
>>> '  '.isspace() # Tab
True
>>> '1'.isspace()
False
```

- `isidentifier()`

如果字符串是有效的标识符，返回 `True`。

什么是有效标识符？[标识符和关键字](#)

```
>>> '_123'.isidentifier()True>>> '123abc'.isidentifier()False>>>
'abc123'.isidentifier()True
```

- `startswith()`

`startswith(prefix[, start[, end]])`

如果字符串以指定的 `prefix` 开始则返回 `True`，否则返回 `False`。`prefix` 也可以为由多个供查找的前缀构成的元组。如果有可选项 `start`，将从所指定位置开始检查。如果有可选项 `end`，将在所指定位置停止比较。

```
>>> 'abc123'.startswith('abc', 0, 6)True>>> 'abc123'.startswith('abc')True>>>
'abc123'.startswith(('abc', 'edf'))True
```

- `endswith()`

`endswith(suffix[, start[, end]])`

如果字符串以指定的 `suffix` 结束返回 `True`，否则返回 `False`。`suffix` 也可以为由多个供查找的后缀构成的元组。如果有可选项 `start`，将从所指定位置开始检查。如果有可选项 `end`，将在所指定位置停止比较。

```
>>> 'abc123'.endswith('123', 0, 6)True>>> 'abc123'.endswith('123')True>>>
'abc123'.endswith(('123', '456'))True
```

- `casefold()`

返回原字符串消除大小写的副本。消除大小写的字符串可用于忽略大小写的匹配。

消除大小写类似于转为小写，但是更加彻底一些，因为它会移除字符串中的所有大小写变化形式。例如，德语小写字母 `'ß'` 相当于 `"ss"`。由于它已经是小写了，`lower()` 不会对 `'ß'` 做任何改变；而 `casefold()` 则会将其转换为 `"ss"`。

```
>>> 'ABCabc'.casefold()'abcabc'>>> 'ABCabcß'.casefold()'abcabcss'
```

- `lower()`

返回原字符串的副本，其所有区分大小写的字符均转换为小写。

```
>>> 'ABCabc'.lower()'abcabc'>>> 'ABCabcß'.lower()'abcabcß'
```

- `upper()`

返回原字符串的副本，其中所有区分大小写的字符均转换为大写。

```
>>> 'ABCabc'.upper()'ABCABC'>>> 'ABCabcß'.upper()'ABCABCSS'
```

- `capitalize()`

返回原字符串的副本，其首个字符大写，其余为小写。

```
>>> 'capitalize'.capitalize()'Capitalize'>>>
'caPiTaLize'.capitalize()'Capitalize'
```

- `title()`

返回原字符串的标题版本，其中每个单词第一个字母为大写，其余字母为小写。

```
>>> 'this is title'.title()'This Is Title'>>> 'tHis is tiTle'.title()'This Is
Title'
```

- `translate()`

`translate(table)`

返回原字符串的副本，其中每个字符按给定的转换表进行映射。转换表必须是一个使用 `getitem()` 来实现索引操作的对象，通常为 mapping 或 sequence。

你可以使用 `str.maketrans()` 基于不同格式的字符到字符映射来创建一个转换映射表（见下文）。

```
>>> 'aaabbb'.translate({97: 'b', 98: 'a'}) # 97, 98 是 ascii'bbbbaaa'>>>
'aaabbb'.translate(str.maketrans({'a': 'b', 'b': 'a'}))'bbbbaaa'
```

- `maketrans()`

`maketrans(x[, y[, z]])`

此静态方法返回一个可供 `str.translate()` 使用的转换对照表。

如果只有一个参数，则它必须是一个将 Unicode 码位序号（整数）或字符（长度为 1 的字符串）映射到 Unicode 码位序号、（任意长度的）字符串或 `None` 的字典。字符键将会被转换为码位序号。

如果有两个参数，则它们必须是两个长度相等的字符串，并且在结果字典中，`x` 中每个字符将被映射到 `y` 中相同位置的字符。

如果有第三个参数，它必须是一个字符串，其中的字符将在结果中被映射到 `None`。

```
>>> str.maketrans({'a': 'b'}){97: 'b'}>>> str.maketrans('abc', 'def'){97: 100,
98: 101, 99: 102}>>> str.maketrans('abc', 'def', 'efg'){97: 100, 98: 101, 99:
102, 101: None, 102: None, 103: None}>>> 'abcefg'.translate(str.maketrans({'a':
'b'}))'bbcefg'>>> 'abcefg'.translate(str.maketrans('abc', 'def'))'defefg'>>>
'abcefg'.translate(str.maketrans('abc', 'def', 'efg'))'def'
```

- `replace()`

`replace(old, new[, count])`

返回字符串的副本，其中出现的所有子字符串 `old` 都将被替换为 `new`。如果给出了可选参数 `count`，则只替换前 `count` 次出现。

```
>>> 'abcabcdef'.replace('abc', 'def')'defdefdef'>>> 'abcabcdef'.replace('abc',
'def', 1)'defababcdef'
```

- `center()`

`center(width[, fillchar])`

返回长度为 `width` 的字符串，原字符串在其正中。使用指定的 `fillchar` 填充两边的空位（默认使用 ASCII 空格符）。如果 `width` 小于等于 `len(s)` 则返回原字符串的副本。

```
>>> 'title'.center(4)'title'>>> 'title'.center(20)'title'>>>  
'title'.center(20, '*')'*****title*****'
```

- ljust()

ljust(width[, fillchar])

返回长度为 width 的字符串，原字符串在其中靠左对齐。使用指定的 fillchar 填充空位 (默认使用 ASCII 空格符)。如果 width 小于等于 len(s) 则返回原字符串的副本。

```
>>> 'title'.ljust(4)'title'>>> 'title'.ljust(20)'title'>>>  
'title'.ljust(20, '*')'title*****'
```

- rjust()

rjust(width[, fillchar])

返回长度为 width 的字符串，原字符串在其中靠右对齐。使用指定的 fillchar 填充空位 (默认使用 ASCII 空格符)。如果 width 小于等于 len(s) 则返回原字符串的副本。

```
>>> 'title'.rjust(4)'title'>>> 'title'.rjust(20)'title'>>>  
'title'.rjust(20, '*')'*****title*****'
```

- join()

str.join(iterable)

返回一个由 iterable (如 list) 中的字符串拼接而成的字符串。如果 iterable 中存在任何非字符串值包括 bytes 对象则会引发 TypeError。调用该方法的字符串将作为元素之间的分隔。

```
>>> '_'.join(['a', 'b', 'c'])'a_b_c'>>> '_'.join(['a', 'b', 'c', 1])Traceback  
(most recent call last):  File "<stdin>", line 1, in <module>TypeError: sequence  
item 3: expected str instance, int found
```

- split()

split(sep=None, maxsplit=-1)

1 返回一个由字符串内单词组成的列表，使用 sep 作为分隔字符串。

2 如果给出了 maxsplit，则最多进行 maxsplit 次拆分（因此，列表最多会有 maxsplit+1 个元素）。如果 maxsplit 未指定或为 -1，则不限制拆分次数（进行所有可能的拆分）。

3 如果给出了 sep，则连续的分隔符不会被组合在一起而是被视为分隔空字符串（例如 '1,,2'.split(',')）将返回 ['1', '', '2']。

4 sep 参数可能由多个字符组成（例如 '1<>2<>3'.split('<>') 将返回 ['1', '2', '3']）。

5 使用指定的分隔符拆分空字符串将返回 []。

6 如果 sep 未指定或为 None，则会应用另一种拆分算法：连续的空格会被视为单个分隔符。如果字符串包含前缀或后缀空格的话，其结果将不包含开头或末尾的空字符串。

7 使用 None 拆分空字符串或仅包含空格的字符串将返回 []。

```
>>> 'AzBzCzD'.split(sep='z')['A', 'B', 'C', 'D']>>> 'AzBzCzD'.split(sep='z', maxsplit=2)['A', 'B', 'CzD']>>> 'AzzBzzCzzD'.split(sep='z')['A', '', 'B', '', 'C', '', 'D']>>> 'AzzBzzCzzD'.split(sep='zz')['A', 'B', 'C', 'D']>>> ''.split(sep='zz')['']>>> ' A B C '.split()['A', 'B', 'C']>>> ' '.split()[]
```

- `splitlines()`

`splitlines([keepends])`

返回由原字符串中各行组成的列表，在行边界的位置拆分。结果列表中不包含行边界，除非给出了 `keepends` 且为真值。

对于空字符串此方法将返回一个空列表，而末尾的换行不会令结果中增加额外的行。

```
>>> 'line1\nline2\nline3'.splitlines()['line1', 'line2', 'line3']>>> 'line1\nline2\nline3'.splitlines(keepends=True)['line1\n', 'line2\n', 'line3']>>> ''.splitlines()[]>>> 'line1\n'.splitlines()['line1']
```

- `rsplit()`

`rsplit(sep=None, maxsplit=-1)`

返回一个由字符串内单词组成的列表，使用 `sep` 作为分隔字符串。如果给出了 `maxsplit`，则最多进行 `maxsplit` 次拆分，从最右边开始。如果 `sep` 未指定或为 `None`，任何空白字符串都会被作为分隔符。除了从右边开始拆分，`rsplit()` 的其他行为都类似于 `split()`。

```
>>> 'A B C D'.rsplit()['A', 'B', 'C', 'D']>>> 'AzBzCzD'.rsplit(sep='z')['A', 'B', 'C', 'D']>>> 'AzBzCzD'.rsplit(sep='z', maxsplit=2)['AzB', 'C', 'D']
```

- `partition()`

`partition(sep)`

在 `sep` 首次出现的位置拆分字符串，返回一个 3 元组，其中包含分隔符之前的部分、分隔符本身，以及分隔符之后的部分。如果分隔符未找到，则返回的 3 元组中包含字符本身以及两个空字符串。

```
>>> 'ABCD'.partition('z')('ABCD', '', '')>>> 'ABzCD'.partition('z')('AB', 'z', 'CD')
```

- `rpartition()`

`rpartition(sep)`

在 `sep` 最后一次出现的位置拆分字符串，返回一个 3 元组，其中包含分隔符之前的部分、分隔符本身，以及分隔符之后的部分。如果分隔符未找到，则返回的 3 元组中包含两个空字符串以及字符串本身。

```
>>> 'ABCD'.rpartition('z')( '', ' ', 'ABCD')>>> 'AzBzCzD'.rpartition('z')('AzBzC', 'z', 'D')
```

- `strip()`

返回原字符串的副本，移除其中的前导和末尾字符。`chars` 参数为指定要移除字符的字符串。如果省略或为 `None`，则 `chars` 参数默认移除空白符。实际上 `chars` 参数并非指定单个前缀或后缀；而是会移除参数值的所有组合。

最外侧的前导和末尾 `chars` 参数值将从字符串中移除。开头端的字符的移除将在遇到一个未包含于 `chars` 所指定字符集的字符时停止。类似的操作也将在结尾端发生。

```
>>> 'line'.strip()'line'>>> '\nline'.strip()'line'>>> 'line'.strip('le')'in'
```

- `rstrip()`

`rstrip([chars])`

返回原字符串的副本，移除其中的末尾字符。`chars` 参数为指定要移除字符的字符串。如果省略或为 `None`，则 `chars` 参数默认移除空白符。实际上 `chars` 参数并非指定单个后缀；而是会移除参数值的所有组合。

```
>>> 'line'.rstrip()'line'>>> 'line\n'.rstrip()'line'>>> 'line'.rstrip('ne')'li'
```

- `removeprefix()`

`removeprefix(prefix)`

如果字符串以 前缀 `prefix` 字符串开头，返回 `string[len(prefix):]`。否则，返回原始字符串的副本。

```
>>> 'headheadababcabc'.removeprefix('head')'headababcabc'>>>
'eadheadababcabc'.removeprefix('head')'eadheadababcabc'>>>
'headheadababcabc'[4:]'headababcabc'
```

- `removesuffix()`

`removesuffix(suffix)`

如果字符串以 后缀 `suffix` 字符串结尾，并且 后缀 非空，返回 `string[:-len(suffix)]`。否则，返回原始字符串的副本。

```
>>> 'abcabctailtail'.removesuffix('tail')'abcabctail'>>>
'abcabctailtai'.removesuffix('tail')'abcabctailtai'>>>
'abcabctailtail'[:-4]'abcabctail'
```

- `index()`

`index(sub[, start[, end]])`

类似于 `find()`，但在找不到子类时会引发 `ValueError`。

```
>>> 'abcdefg'.index('cde')2>>> 'abcdefg'.index('cde', 1)2>>>
'abcdefg'.index('cde', 3)Traceback (most recent call last): File "<stdin>",
line 1, in <module>ValueError: substring not found>>> 'abcdefg'.index('cde', 1,
5)2>>> 'abcdefg'.index('cde', 1, 4)Traceback (most recent call last): File "
<stdin>", line 1, in <module>ValueError: substring not found
```

- `rindex()`

`rindex(sub[, start[, end]])`

类似于 `rfind()`，但在子字符串 `sub` 未找到时会引发 `ValueError`。

```
>>> 'abcdefg'.rindex('cde')2>>> 'abcdefg'.rindex('cde', 1)2>>>
'abcdefg'.rindex('cde', 3)Traceback (most recent call last): File "<stdin>",
Line 1, in <module>ValueError: substring not found>>> 'abcdefg'.rindex('cde', 1,
5)2>>> 'abcdefg'.rindex('cde', 1, 4)Traceback (most recent call last): File "<stdin>",
Line 1, in <module>ValueError: substring not found
```

- `find()`

```
find(sub[, start[, end]])
```

返回子字符串 `sub` 在 `s[start:end]` 切片内被找到的最小索引。可选参数 `start` 与 `end` 会被解读为切片表示法。如果 `sub` 未被找到则返回 -1。

```
>>> 'abcdefg'.find('cde')2>>> 'abcdefg'.find('cde', 1)2>>> 'abcdefg'.find('cde',
3)-1>>> 'abcdefg'.find('cde', 1, 5)2>>> 'abcdefg'.find('cde', 1, 4)-1
```

- `rfind()`

```
rfind(sub[, start[, end]])
```

返回子字符串 `sub` 在字符串内被找到的最大（最右）索引，这样 `sub` 将包含在 `s[start:end]` 当中。可选参数 `start` 与 `end` 会被解读为切片表示法。如果未找到则返回 -1。

```
>>> 'abcdecdefg'.rfind('cde')5>>> 'abcdecdefg'.rfind('cde', 5)5>>>
'abcdecdefg'.rfind('cde', 6)-1>>> 'abcdecdefg'.rfind('cde', 5, 8)5>>>
'abcdecdefg'.rfind('cde', 5, 7)-1
```

- `count()`

```
count(sub[, start[, end]])
```

返回子字符串 `sub` 在 `[start, end]` 范围内非重叠出现的次数。可选参数 `start` 与 `end` 会被解读为切片表示法。

```
>>> 'zzaaaazz'.count('aaa')1>>> 'zzaaaazz'.count('aaa', 2)1>>>
'zzaaaazz'.count('aaa', 2, 6)1
```

- `sort()`

字符串本身没有 `sort` 方法，但是可以将字符串转换为 `list`，进行 `sort` 之后再 `join` 变回字符串。

```
>>> lst = list('acdb')>>> lst.sort()>>> lst['a', 'b', 'c', 'd']>>>
''.join(lst)'abcd'
```

- `reverse()`

字符串本身没有 `reverse` 方法，但是可以将字符串转换成 `list`，进行 `reverse` 之后再 `join` 变回字符串。

```
>>> lst = list('abcd')>>> lst.reverse()>>> lst['d', 'c', 'b', 'a']>>>
''.join(lst)'dcba'
```

- `encode()`

```
encode(encoding='utf-8', errors='strict')
```

返回原字符串编码为字节串对象的版本。默认编码为 'utf-8'。可以给出 errors 来设置不同的错误处理方案。errors 的默认值为 'strict'，表示编码错误会引发 UnicodeError。

将 UNICODE 字符串转换为其它编码格式的字节串，与之相反的是 decode。

字典的操作方法

字典的键几乎可以是任何值。非 hashable 的值，即包含列表、字典或其他可变类型的值（此类对象基于值而非对象标识进行比较）不可用作键。数字类型用作键时遵循数字比较的一般规则：如果两个数值相等（例如 1 和 1.0）则两者可以被用来索引同一字典条目。（但是请注意，由于计算机对于浮点数存储的只是近似值，因此将其用作字典键是不明智的。）

字典可以通过将以逗号分隔的键: 值对列表包含于花括号之内来创建，例如: {'jack': 4098, 'sjoerd': 4127} 或 {4098: 'jack', 4127: 'sjoerd'}，也可以通过 dict 构造器来创建。

- update()

`update([other])`

使用来自 other 的键/值对更新字典，覆盖原有的键。返回 None。

`update()` 接受另一个字典对象，或者一个包含键/值对（以长度为二的元组或其他可迭代对象表示）的可迭代对象。如果给出了关键字参数，则会以其所指定的键/值对更新字典: `d.update(red=1, blue=2)`。

```
>>> dct = {'a': 1}>>> dct.update(a=2)>>> dct{'a': 2}>>> dct.update({'a': 3})>>>  
dct{'a': 3}>>> dct.update({'b': 3})>>> dct{'a': 3, 'b': 3}
```

- del

del d[key]

将 `d[key]` 从 `d` 中移除。如果映射中不存在 `key` 则会引发 `KeyError`。

```
>>> dct = {'a': 1}>>> del dct['b']Traceback (most recent call last):  File "<stdin>", line 1, in <module>KeyError: 'b'>>> del dct['a']>>> dct{}
```

- clear()

移除字典中的所有元素。

```
>>> dct = {'a': 1, 'b': 2}>>> dct.clear()>>> dct{}
```

- pop()

`pop(key[, default])`

如果 key 存在于字典中则将其移除并返回其值，否则返回 default。如果 default 未给出且 key 不存在于字典中，则会引发 KeyError。

```
>>> dct = {'a': 1, 'b': 2}>>> dct.pop('c')Traceback (most recent call last):File "<stdin>", line 1, in <module>KeyError: 'c'>>> dct.pop('c', 3)3>>>dct.pop('a', 3)1
```

- `popitem()`

从字典中移除并返回一个(键, 值)对。键值对会按LIFO(后进先出)的顺序被返回。

`popitem()`适用于对字典进行消耗性的迭代，这在集合算法中经常被使用。如果字典为空，调用`popitem()`将引发`KeyError`。

在3.7之前的版本中，`popitem()`会返回一个任意的键/值对。

```
>>> dct = {}>>> dct['a'] = 1>>> dct['b'] = 1>>> dct.popitem()('b', 1)>>>dct.popitem()('a', 1)>>> dct{}
```

- `fromkeys()`

`fromkeys(iterable[, value])`

使用来自`iterable`的键创建一个新字典，并将键值设为`value`。

`fromkeys()`是一个返回新字典的类方法。`value`默认为`None`。所有值都只引用一个单独的实例，因此让`value`成为一个可变对象例如空列表通常是没有意义的。要获取不同的值，请改用字典推导式。

```
>>> dict.fromkeys(['a', 'b', 'c']){'a': None, 'b': None, 'c': None}>>>dict.fromkeys(['a', 'b', 'c'], 1){'a': 1, 'b': 1, 'c': 1}
```

- `setdefault()`

`setdefault(key[, default])`

如果字典存在键`key`，返回它的值。如果不存在，插入值为`default`的键`key`，并返回`default`。`default`默认为`None`。

```
>>> dct = {'a': 1}>>> dct.setdefault('a')1>>> dct.setdefault('b')>>> dct{'a': 1, 'b': None}>>> dct.setdefault('c', 1)>>> dct{'a': 1, 'b': None, 'c': 1}
```

- `get()`

`get(key[, default])`

如果`key`存在于字典中则返回`key`的值，否则返回`default`。如果`default`未给出则默认为`None`，因而此方法绝不会引发`KeyError`。

```
>>> dct = {'a': 1}>>> print(dct.get('a'))1>>> print(dct.get('b'))None>>>dct{'a': 1}
```

- `keys()`

`keys()`

返回由字典键组成的一个新视图。

```
>>> dct = {'a': 1, 'b': 2}>>> dct.keys()dict_keys(['a', 'b'])
```

- `values()`

返回由字典值组成的一个新视图。

两个 `dict.values()` 视图之间的相等性比较将总是返回 `False`。这在 `dict.values()` 与其自身比较时也同样适用。

```
>>> dct1 = {'a': 1, 'b': 2}>>> dct2 = {'a': 1, 'b': 2}>>>
dct1.values()dict_values([1, 2])>>> dct2.values()dict_values([1, 2])>>>
dct1.values() == dct2.values()False>>> dct1.values() == dct1.values()False
```

- `items()`

返回由字典项 (键, 值) 对组成的一个新视图。

```
>>> dct.items()dict_items([('a', 1), ('b', 2)])
```

列表操作方法

列表是可变序列，通常用于存放同类项目的集合。

- `append()`

在列表末尾添加一个元素，相当于 `a[len(a):] = [x]`。

```
>>> a = []>>> a.append(1)>>> a[1]>>> a[len(a):] = [2]>>> a[1, 2]
```

- `extend()`

`extend(iterable)`

用可迭代对象的元素扩展列表。相当于 `a[len(a):] = iterable`。

```
>>> a = []>>> a = [1]>>> b = [2, 3, 4, 5]>>> a.extend(b)>>> a[1, 2, 3, 4, 5]
```

- `pop()`

`pop(i)` 或 `pop()`

删除列表中指定位置的元素，并返回被删除的元素。未指定位置时，`a.pop()` 删除并返回列表的最后一个元素。

```
>>> a = [1, 2, 3, 4, 5]>>> a.pop(1)>>> a[1, 3, 4, 5]>>> a.pop()>>> a[1, 3, 4]
```

- `del`

按索引，而不是值从列表中移除元素。与返回值的 `pop()` 方法不同，`del` 语句也可以从列表中移除切片，或清空整个列表（之前是将空列表赋值给切片）。

```
>>> a = [1, 2, 3, 4, 5]>>> del a[0]>>> a[2, 3, 4, 5]>>> del a[1:3]>>> a[2, 5]>>>
del a[:]>>> a[]>>> del a # 删除对象>>> aTraceback (most recent call last):  File "<stdin>", line 1, in <module>NameError: name 'a' is not defined
```

- `remove()`

`remove(x)`

从列表中删除第一个值为 x 的元素。未找到指定元素时，触发 `ValueError` 异常。

```
>>> a = [1, 2, 3, 4, 5]>>> a.remove(1)>>> a[2, 3, 4, 5]>>> a.remove(1)Traceback  
(most recent call last): File "<stdin>", line 1, in <module>ValueError:  
list.remove(x): x not in list
```

- `clear()`

删除列表里的所有元素，相当于 `del a[:]`。

```
>>> a = [1, 2, 3, 4, 5]>>> a.clear()>>> a[]>>> a = [1, 2, 3, 4, 5]>>> del  
a[:]>>> a[]
```

- `index()`

`index(x[, start[, end]])`

返回列表中第一个值为 x 的元素的以0起始的索引。未找到指定元素时，触发 `ValueError` 异常。

可选参数 `start` 和 `end` 是切片符号，用于将搜索限制为列表的特定子序列。返回的索引是相对于整个序列的开始计算的，而不是 `start` 参数。

```
>>> a = [0, 1, 2, 3, 4, 5]>>> a.index(3)3>>> a.index(3, 2)3>>> a.index(3, 2,  
4)3>>> a.index(6)Traceback (most recent call last): File "<stdin>", line 1, in  
<module>ValueError: 6 is not in list
```

- `insert()`

`insert(i, x)`

在指定位置插入元素。第一个参数是插入元素的索引，因此，`a.insert(0, x)` 在列表开头插入元素，`a.insert(len(a), x)` 等同于 `a.append(x)`。

```
>>> a = [1, 2, 3, 4, 5]>>> a.insert(0, 0)>>> a[0, 1, 2, 3, 4, 5]>>>  
a.insert(len(a), 6)>>> a[0, 1, 2, 3, 4, 5, 6]
```

- `count()`

`count(x)`

返回列表中元素 x 出现的次数。

```
>>> a = [1, 2, 2, 3, 3, 3]>>> a.count(0)0>>> a.count(3)3
```

- `sort()`

`sort(*, key=None, reverse=False)`

就地排序列表中的元素。

`key` 指定带有单个参数的函数，用于从 `iterable` 的每个元素中提取用于比较的键（例如 `key=str.lower`）。默认值为 `None`（直接比较元素）。

`reverse` 为一个布尔值。如果设为 `True`，则每个列表元素将按反向顺序比较进行排序。

```
>>> a = [5, 4, 3, 2, 1]>>> a.sort()>>> a[1, 2, 3, 4, 5]>>> a.sort(key=lambda x:  
-x) # 取相反数进行比较>>> a[5, 4, 3, 2, 1]>>> a.sort(key=lambda x: -x,  
reverse=True) # 取相反数且反向排序>>> a[1, 2, 3, 4, 5]
```

- `reverse()`

`reverse()`

翻转列表中的元素。

```
>>> a = [5, 4, 3, 2, 1]>>> a.reverse()>>> a[1, 2, 3, 4, 5]
```

元组操作方法

元组由多个用逗号隔开的值组成。

```
>>> (1) # 不是元组1>>> (1,) # 是元组(1,)>>> (1, 2)(1, 2)
```

- `del`

元组中的元素值是不允许删除的，但我们可以使用`del`语句来删除整个元组。

```
>>> a = (0, 1, 2, 3)>>> del a[0]Traceback (most recent call last): File "  
<stdin>", line 1, in <module>TypeError: 'tuple' object doesn't support item  
deletion>>> del a>>> aTraceback (most recent call last): File "<stdin>", line 1,  
in <module>NameError: name 'a' is not defined
```

- `count(x)`

统计 `x` 在元组中出现的总次数。

```
>>> a = (1, 2, 2, 3, 3, 3)>>> a.count(0)0>>> a.count(3)3
```

- `index()`

`index(x[, start[, end]])`

返回元组中第一个值为 `x` 的元素的以 0 起始的索引。未找到指定元素时，触发 `ValueError` 异常。

可选参数 `start` 和 `end` 是切片符号，用于将搜索限制为元组的特定子序列。返回的索引是相对于整个元组的开始计算的，而不是 `start` 参数。

```
>>> a = (0, 1, 2, 3, 4, 5)>>> a.index(3)3>>> a.index(3, 2)3>>> a.index(3, 2,  
4)3>>> a.index(6)Traceback (most recent call last): File "<stdin>", line 1, in  
<module>ValueError: tuple.index(x): x not in tuple
```

集合 set/frozenset 操作方法

集合是由不重复元素组成的无序容器。基本用法包括成员检测、消除重复元素。集合对象支持合集、交集、差集、对称差分等数学运算。

`set` 类型是可变的，其内容可以使用 `add()` 和 `remove()` 这样的方法来改变。由于是可变类型，它没有哈希值，且不能被用作字典的键或其他集合的元素。

frozenset 类型是不可变并且为 hashable，其内容在被创建后不能再改变；因此它可以被用作字典的键或其他集合的元素。

```
>>> st = {} # 不能用空 {} 创建集合, 空{} 创建的是字典>>> type(st)<class 'dict'>>> st = {0}>>> type(st)<class 'set'>>> st = set([0, 1, 2, 3])>>> st.add(4)>>> st{0, 1, 2, 3, 4}>>> fst = frozenset([0, 1, 2, 3])>>> fst.add(4)Traceback (most recent call last):  File "<stdin>", line 1, in <module>AttributeError: 'frozenset' object has no attribute 'add'
```

- add()

add(elem)

将元素 elem 添加到集合中。

```
>>> st = set()>>> st.add(0)>>> st{0}
```

- update()

update(*others)

等同于 `set |= other | ...`

更新集合，添加来自 others 中的所有元素。

```
>>> st = set()>>> st1 = set({1})>>> st2 = set({2})>>> st3 = set({3})>>> st4 = set({4})>>> st.update(st1)>>> st{1}>>> st |= st2>>> st{1, 2}>>> st |= st3 | st4>>> st{1, 2, 3, 4}
```

- copy()

返回原集合的浅拷贝。

```
>>> st = {0, 1, 2}>>> st1 = st.copy()>>> st1{0, 1, 2}
```

- remove()

remove(elem)

从集合中移除元素 elem。如果 elem 不存在于集合中则会引发 KeyError。

```
>>> st = {0, 1, 2}>>> st.remove(1)>>> st{0, 2}>>> st.remove(1)Traceback (most recent call last):  File "<stdin>", line 1, in <module>KeyError: 1
```

- pop()

从集合中移除并返回任意一个元素。如果集合为空则会引发 KeyError。

```
>>> st = {0, 1, 2}>>> st.pop()>>> st = set()>>> st.pop()Traceback (most recent call last):  File "<stdin>", line 1, in <module>KeyError: 'pop from an empty set'
```

- discard()

discard(elem)

如果元素 elem 存在于集合中则将其移除。如果 elem 不存在于集合中，不会引发 KeyError。

```
>>> st = {0, 1, 2}>>> st.discard(1)>>> st{0, 2}>>> st.discard(1)>>> st{0, 2}
```

- clear()

从集合中移除所有元素。

```
>>> st = {0, 1, 2}>>> st.clear()>>> stset()
```

- & intersection()

intersection(*others)

set & other & ...

返回一个新集合，其中包含原集合以及 others 指定的所有集合中共有的元素。

```
>>> st1 = {1, 2}>>> st2 = {1, 3}>>> st3 = {1, 4}>>> st1.intersection(st2){1}>>>
st1.intersection(st2, st3){1}>>> st1 & st2{1}>>> st1 & st2 & st3{1}
```

- &= intersection_update() 改变左侧的set内容

intersection_update(*others)

set &= other & ...

更新集合，只保留其中在所有 others 中也存在的元素。

```
>>> st1 = {1, 2}>>> st2 = {1, 2}>>> st3 = {1, 4}>>>
st1.intersection_update(st2)>>> st1{1, 2}>>> st1.intersection_update(st2,
st3)>>> st1{1}>>> st1 = {1, 2}>>> st2 = {1, 2}>>> st3 = {1, 3}>>> st1 &= st2>>>
st1{1, 2}>>> st1 &= st2 & st3>>> st1{1}
```

- | union()

union(*others)

set | other | ...

返回一个新集合，其中包含来自原集合以及 others 指定的所有集合中的元素。

```
>>> st1 = {1}>>> st2 = {2}>>> st3 = {3}>>> st1.union(st2){1, 2}>>>
st1.union(st2, st3){1, 2, 3}>>> st1 | st2{1, 2}>>> st1 | st2 | st3{1, 2, 3}
```

- difference()

difference(*others)

set - other - ...

返回一个新集合，其中包含原集合中在 others 指定的其他集合中不存在的元素。

```
>>> st1 = {1, 2, 3, 4, 5}>>> st2 = {2, 3}>>> st3 = {4, 5}>>> st1.difference(st2)
{1, 4, 5}>>> st1.difference(st2, st3){1}>>> st1 - st2{1, 4, 5}>>> st1 - st2 -
st3{1}
```

- -= difference_update()

difference_update(*others)

```
set -= other | ...
```

更新集合，移除其中也存在于 others 中的元素。

```
>>> st1 = {1, 2, 3, 4, 5}>>> st2 = {2}>>> st3 = {3}>>> st4 = {4}>>> st5 = {5}>>>
st1.difference_update(st2)>>> st1{1, 3, 4, 5}>>> st1.difference_update(st3, st4,
st5)>>> st1{1}>>> st1 = {1, 2, 3, 4, 5}>>> st2 = {2}>>> st3 = {3}>>> st4 =
{4}>>> st5 = {5}>>> st1 -= st2>>> st1{1, 3, 4, 5}>>> st1 -= st3 -st4 -st5>>>
st1{1, 4, 5}
```

- ^ symmetric_difference()

```
symmetric_difference(other)¶
```

```
set ^ other
```

返回一个新集合，其中的元素或属于原集合或属于 other 指定的其他集合，但不能同时属于两者。

```
>>> st1 = {0, 1}>>> st2 = {0, 2}>>> st1.symmetric_difference(st2){1, 2}>>> st1 ^
st2{1, 2}
```

- ^= symmetric_difference_update()

```
symmetric_difference_update(other)
```

```
set ^= other
```

更新集合，只保留存在于集合的任一方而非共同存在的元素。

```
>>> st1 = {0, 1}>>> st2 = {0, 2}>>> st1.symmetric_difference_update(st2)>>>
st1{1, 2}>>> st1 = {0, 1}>>> st1 ^= st2>>> st1{1, 2}
```

- isdisjoint() !=
 - isdisjoint(other)

如果集合中没有与 other 共有的元素则返回 True。当且仅当两个集合的交集为空集合时，两者为不相交集合。

```
>>> st1 = {0, 1}>>> st2 = {0, 2}>>> st3 = {3}>>> st1.isdisjoint(st2)False>>>
st1.isdisjoint(st3)True
```

```
* !=
```

集合 (set 或 frozenset 的实例) 可进行类型内部和跨类型的比较。

它们将比较运算符定义为子集和超集检测。这类关系没有定义完全排序 (例如 {1,2} 和 {2,3} 两个集合不相等，即不为彼此的子集，也不为彼此的超集)。

```
>>> st1 = {0, 1}>>> st2 = {1, 0}>>> st3 = {0, 2}>>> id(st1), id(st2), st1 !=
st2, st1 == st2(2414743974400, 2414743974176, False, True)>>> st1 != st3True
```

- issuperset() >=
- issuperset() >=

```
issuperset(other)
```

set >= other

检测是否 other 中的每个元素都在集合之中。

```
>>> st1 = {1, 2, 3}>>> st2 = {1, 2, 3}>>> st3 = {2, 3}>>> st4 = {4}>>>
st1.issuperset(st2)True>>> st1.issuperset(st3)True>>>
st1.issuperset(st4)False>>> st1 >= st2True>>> st1 >= st3True>>> st1 >= st4False
```

* >

set > other

检测集合是否为 other 的真超集，即 set >= other and set != other。

```
>>> st1 = {1, 2, 3}>>> st2 = {1, 2, 3}>>> st3 = {2, 3}>>> st4 = {4}>>> st1 >
st2False>>> st1 > st3True>>> st1 > st4False
```

- issubset() < <=
- issubset() <=

issubset(other)

set <= other

检测是否集合中的每个元素都在 other 之中。

```
>>> st1 = {1, 2, 3}>>> st2 = {1, 2, 3}>>> st3 = {2, 3}>>> st4 = {4}>>>
st4.issubset(st1)False>>> st3.issubset(st1)True>>> st2.issubset(st1)True>>> st4
<= st1False>>> st3 <= st1True>>> st2 <= st1True
```

- <

set < other

检测集合是否为 other 的真子集，即 set <= other and set != other。

```
>>> st1 = {1, 2, 3}>>> st2 = {1, 2, 3}>>> st3 = {2, 3}>>> st4 = {4}>>> st4 <
st1False>>> st3 < st1True>>> st2 < st1False
```

新

链式传值

```
li=[1,2]
print(li*3)
[1,2,1,2,1,2]
```

```
p=()
print(p)
()

p=(1)
```

```
print(p)
1

a,b=1,2
print(a,b)
1 2

a,b="12"
print(a,b)
"1" "2"

a,b=b,a=1,2
print(a,b)
2 1
```

打包解包

打包：

函数定义时，形参前加*号（如：*args）：收集实参中所有的位置参数，打包成新元组并将该元组赋值给args变量

实参位置参数：实参中所有不带形参名的参数均是位置参数（如实参传递：a, b, 1）

二、函数定义时，形参前带*号（如：*kwargs）：收集实参中的所有关键字参数，打包成新字典并将该字典赋值给kwargs变量

实参关键字参数：实参中所有带形参名的参数均是关键字参数（如实参传递：a = 1, b = 2）

解包

在函数调用时，实参前加 *号（如：*list）：将列表、元组、字符串解包成独立的元素作为参数

注意：*号后必须是可迭代序列（list / tuple / str）

```
pratice.py > ...
1 my_list = [1, 2, 3]
2
3 def sum_s(*args):
4     # 求和
5     print(f"kwargs的类型是: {type(args)}, 值是: {args}")
6
7
8 sum_s(*my_list) ← 使用*号解包列表my_list后得到三个位置参数: 1, 2, 3
9
10
11
12
```

问题 输出 调试控制台 终端 2: Python +

```
Windows PowerShell
版权所有 (C) Microsoft Corporation。保留所有权利。
尝试新的跨平台 PowerShell https://aka.ms/pscore6
PS C:\Users\huihe12\Desktop\python> & "C:/Program Files/Python38/python.exe" c:/Users/huihe12/Desktop/python/pratice.py
kwargs的类型是: <class 'tuple'>, 值是: (1, 2, 3)
PS C:\Users\huihe12\Desktop\python>
```

二、在函数调用时实参前加 **号 (如: **dict) : 将字典解包成独立的关键字参数 (键值对) 作为参数

注意: **号后必须是dict类型数据

示例:

```
pratice.py > ...
1 pratice.py > ...
2 my_dict = {"name": "Rachel", "age": 18}
3
4 def sum_s(**kwargs):
5     # 求和
6     print(f"kwargs的类型是: {type(kwargs)}, 值是: {kwargs}")
7
8 sum_s(**my_dict) ← 使用**解包my_dict字典后得到两个关键字参数: name = "Rachel", age = 18
9
10
11
12
```

问题 输出 调试控制台 终端 2: Python +

```
Windows PowerShell
版权所有 (C) Microsoft Corporation。保留所有权利。
尝试新的跨平台 PowerShell https://aka.ms/pscore6
PS C:\Users\huihe12\Desktop\python> & "C:/Program Files/Python38/python.exe" c:/Users/huihe12/Desktop/python/pratice.py
kwargs的类型是: <class 'dict'>, 值是: {'name': 'Rachel', 'age': 18}
PS C:\Users\huihe12\Desktop\python>
```

列表与字典 (可变对象) 解包传递与不解包传递的差异:

解包与直接传递的差异:

解包传递: 解包传递是值传递, 修改被传递的值不会影响原始列表的数据;

直接传递: 直接传递是引用传递, 传递的是原列表的地址给新的变量, 对该变量作任意修改将影响原始列表的数据;

总结:

1、函数定义时, 形参前加*或*号是打包, *号是以新元组形式打包实参中所有的位置参数并将新元组赋值给*号后的变量, *是以新字典形式打包实参中的所有关键字参数并将新字典赋值给**号后的变量;

2、调用函数时，实参前加*或**是对序列解包，*号是解包列表/元组/字符串为独立的元素作为参数，*号是解包字典键值对（每一对键值对都将解包成一个关键字参数）为关键字参数作为实参；

3、当形参中使用动态参数（参数带*或**）时，实参可解包序列传递，或需要解包序列传递参数且不确定序列中元素个数时，在形参中可使用动态参数接收；

4、解包列表或字典传递即使在函数中有修改操作，也不会修改原列表与字典的数据；

若直接传递列表或字典时函数内一旦有修改操作则会原列表或字典数据将被修改，因为列表和字典是可变类型，直接传递时实际传递的是列表与字典的内存地址，函数内可通过地址修改原列表与字典；

```
def a(*b):
    print(b)

s=(1,2,3)
a(s)
a(*s)

((1, 2, 3),)
(1, 2, 3)

def a(s,*b):
    print(s)
    print(b)

s=(1,2,3)
a(s)
a(*s)
(1, 2, 3)
()
1
(2, 3)
```

装饰器

第一种：普通装饰器

第二种：带参数的函数装饰器

第三种：不带参数的类装饰器

第四种：带参数的类装饰器

第五种：使用偏函数与类实现装饰器

第六种：能装饰类的装饰器

reverse和sorted

reversed reverse 方法

reverse():

是[python](#)中列表的一个内置方法 (也就是说，在字典，字符串或者元组中，是没有这个内置方法的) ，
用于列表中数据的反转；

```
lista = [1, 2, 3, 4]
lista.reverse()
print(lista)
[4,3,2,1]
```

reversed():

而reversed()是[python](#)自带的一个方法，准确说，应该是一个类；

reverse (sequence) ->反转迭代器的序列值
返回反向迭代器

也就是说，在经过reversed()的作用之后，返回的是一个把序列值经过反转之后的迭代器，所以，需要通过遍历，或者List,或者next()等方法，获取作用后的值；

```
bb = [1,3,5,7]
print(list(reversed(bb)))
[7, 5, 3, 1]
```

sort sorted 函数

list.sort() 不会返回对象，会改变原有的list

sorted()函数会返回一个列表，而sort () 函数是直接在原来的基础上修改

```
list=["cc", "d", "a", "bbbb"]
list.sort()
print(list)

list=["cc", "d", "a", "bbbb"]
list1=sorted(list)
print(list1)
```

re

用 \d 可以匹配一个数字， \w 可以匹配一个字母或数字，

. 可以匹配任意字符

用 * 表示任意个字符 (包括0个) ， 用 + 表示至少一个字符，用 ? 表示0个或1个字符，用 {n} 表示n个字符，用 {n,m} 表示n-m个字符：

如果要匹配 '010-12345' 这样的号码呢？由于 '-' 是特殊字符，在正则表达式中，要用 '\-' 转义，所以，上面的正则是 \d{3}\-\d{3,8} 。

要做更精确地匹配，可以用 [] 表示范围，比如：

- [0-9a-zA-Z_] 可以匹配一个数字、字母或者下划线；
- [0-9a-zA-Z_]+ 可以匹配至少由一个数字、字母或者下划线组成的字符串，比如 'a100'，'0_Z'，'Py3000' 等等；
- [a-zA-Z_] [0-9a-zA-Z_] * 可以匹配由字母或下划线开头，后接任意个由一个数字、字母或者下划线组成的字符串，也就是Python合法的变量；
- [a-zA-Z_] [0-9a-zA-Z_] {0, 19} 更精确地限制了变量的长度是1-20个字符（前面1个字符+后面最多19个字符）。

A|B 可以匹配A或B，所以 (P|p)ython 可以匹配 'Python' 或者 'python'。

^ 表示行的开头，^\d 表示必须以数字开头。

\$ 表示行的结束，\d\$ 表示必须以数字结束。

你可能注意到了，py 也可以匹配 'python'，但是加上 ^py\$ 就变成了整行匹配，就只能匹配 'py' 了。

建议使用Python的 r 前缀，就不用考虑转义的问题了：

```
s = r'ABC\~-001' # Python的字符串  
# 对应的正则表达式字符串不变:  
# 'ABC\~-001'
```

match() 方法判断是否匹配，如果匹配成功，返回一个 Match 对象，否则返回 None。常见的判断方法就是：

```
test = '用户输入的字符串'  
if re.match(r'正则表达式', test):  
    print('ok')  
else:  
    print('failed')
```

字符	描述
\d	代表任意数字，就是阿拉伯数字 0-9 这些玩意。
\D	大写的就是和小写的唱反调，\d 你代表的是任意数字是吧？那么我 \D 就代表不是数字的。
\w	代表字母，数字，下划线。也就是 a-z、A-Z、0-9、_。
\W	跟 \w 唱反调，代表不是字母，不是数字，不是下划线的。
\n	代表一个换行。
\r	代表一个回车。
\f	代表换页。
\t	代表一个 Tab。
\s	代表所有的空白字符，也就是上面这个：\n、\r、\t、\f。
\S	跟 \s 唱反调，代表所有不是空白的字符。
\A	代表字符串的开始。
\Z	代表字符串的结束。
	匹配字符串开始的位置。

compile()

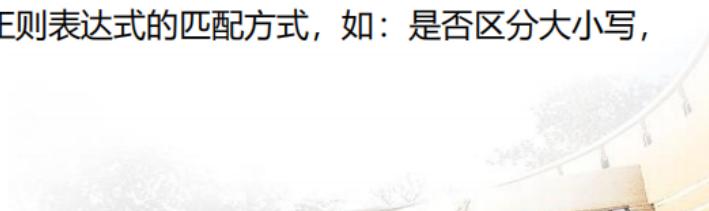
编译正则表达式模式，返回一个对象的模式。（可以把那些常用的正则表达式编译成正则表达式对象，这样可以提高一点效率。）

格式：

re.compile(pattern,flags=0)

pattern: 编译时用的表达式字符串。

flags 编译标志位，用于修改正则表达式的匹配方式，如：是否区分大小写，多行匹配等。



compile()

常用的flags有：

标志	含义
re.S(DOTALL)	使.匹配包括换行在内的所有字符
re.I (IGNORECASE)	使匹配对大小写不敏感
re.L (LOCALE)	做本地化识别 (locale-aware) 匹配, 法语等
re.M(MULTILINE)	多行匹配, 影响^和\$
re.X(VERBOSE)	该标志通过给予更灵活的格式以便将正则表达式写得更容易理解
re.U	根据Unicode字符集解析字符, 这个标志影响 \w,\W,\b,\B

match()

决定RE是否在字符串刚开始的位置匹配。//注：这个方法并不是完全匹配。
当pattern结束时若string还有剩余字符，仍然视为成功。想要完全匹配，可以在表达式末尾加上边界匹配符'\$'

格式：

```
re.match(pattern, string, flags=0)
```

```
print(re.match('com','comwww.runcomoob').group())
print(re.match('com','Comwww.runcomoob',re.I).group())
```

search()

格式：

```
re.search(pattern, string, flags=0)
```

re.search函数会在字符串内查找模式匹配,只要找到第一个匹配然后返回，如果字符串没有匹配，则返回None。

```
print(re.search('\dcom','www.4comrunoob.5com').group())
```

执行结果如下：

4com

*注：match和search一旦匹配成功，就是一个match object对象，而match object对象有以下方法：

search()

group() 返回被 RE 匹配的字符串

start() 返回匹配开始的位置

end() 返回匹配结束的位置

span() 返回一个元组包含匹配(开始,结束)的位置

group() 返回re整体匹配的字符串，可以一次输入多个组号，对应组号匹配的字符串。

a. group () 返回re整体匹配的字符串，

b. group (n,m) 返回组号为n, m所匹配的字符串，如果组号不存在，则返回indexError异常

c.groups () groups() 方法返回一个包含正则表达式中所有小组字符串的元组，从 1 到所含的小组号，通常groups()不需要参数，返回一个元组，元组中的元就是正则表达式中定义的组。

findall()

re.findall遍历匹配，可以获取字符串中所有匹配的字符串，返回一个列表。

格式：

re.findall(pattern, string, flags=0)

findall()

```
p = re.compile(r'\d+')
print(p.findall('01n2m3k4'))
```

执行结果如下：

```
['1', '2', '3', '4']
```

```
import re
tt = "Tina is a good girl, she is cool, clever, and so on..."
rr = re.compile(r'\w*oo\w*')
print(rr.findall(tt))
print(re.findall(r'(\w)*oo(\w)', tt)) # ()表示子表达式
```

执行结果如下：

```
['good', 'cool']
[('g', 'd'), ('c', 'l')]
```

