

15 python 字符编码问题解析

在计算机中，所有的数据在[存储](#)和运算时都要使用[二进制数](#)表示具体用哪些二进制数字表示哪个符号，当然每个人都可以约定自己的一套（这就叫[编码](#)）

最知名最早的字符集 ASCII

通用字符集 UNICODE 字符集有多个编码方式，分别是 UTF-8，UTF-16，UTF-32 和 UTF-7 编码。

python 字符串的类型有两种：unicode 和 str

str 本质上是一串二进制字节序列，而 unicode 是字符。

utf-8 是一种字符集。可以叫通用转换格式。（Unicode Transformation Format）：

是一种可变长度字符编码 UTF-8 用 1 到 6 个字节编码 Unicode 字符。

当我们要把 unicode 符号保存到文件或者传输到网络就需要经过编码处理转换成 str 类型。于是 python 提供了 encode 方法，从 unicode-----> str 反之亦然。

于是不难理解 UnicodeEncodeError 发生在 unicode 转换成 str 的时候，

```
name = u'python之禅'
name.encode('utf-8')
print(name,type(name))

python之禅 <class 'str'>
```

上代码表示将 unicode 转换成为 utf-8 形式的 str，因为 str 是二进制字节序列，所以可以直接存入。

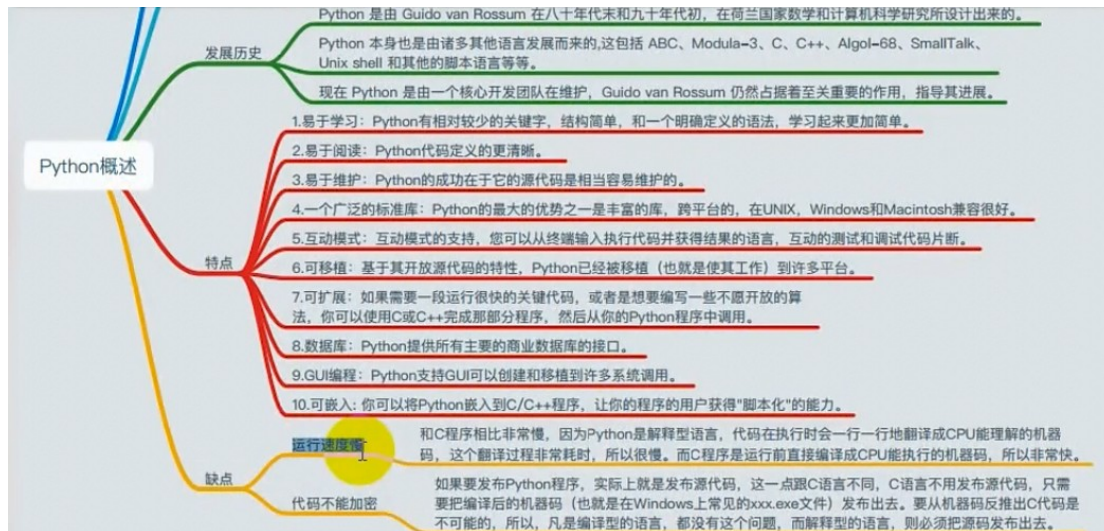
UnicodeDecodeError 一般是由于 str 转换成 unicode 时候出错。错误示例

```
a = u'禅'
print(a)
b = a.encode('utf-8')
print(b)
b.decode('gbk')
```

```
禅
b'\xe7\xa6\x85'
Traceback (most recent call last):
  File "C:/Users/Administrator/PycharmProjects/千锋教程/caogao.py", line 44, in <module>
    b.decode('gbk')
UnicodeDecodeError: 'gbk' codec can't decode byte 0x85 in position 2: incomplete multibyte sequence
```

错误分析：a 经过 utf-8 编码后生成的字节序列再用 gbk 解码成 unicode 字符串时候，因为对于中文字符而言，gbk 编码只占 2 个字节，而 utf-8 占 3 个字节，所以用 gbk 转换时多个字节，没法解析。因此避免 UnicodeDecodeError 的关键就是保持编码和解码时用的编码类型一致。

1：python 中的函数和方法是同一回事。



python 的缺点：

- 1 python 的代码开源，代码不能加密，发布文件就是发布了代码。（c 发布的是一种编译性的机器码，只有计算机可以看懂的。）凡是编译型语言都是加密的，解释型都是可以看到源代码的。
- 2 python 的运行速度慢，是一门解释性语言，代码在执行时会先翻译成 cpu 能理解的语言，这个过程很耗时间，所以说慢，（c 的运行是直接编译成 cpu 能执行的机器码，所以很快）
提到运行速度，需要考虑两个概念，数据密集和 IO 密集，计算数据比较多的用 C 比较好，但是磁盘的读写和网络的请求的 IO 请求 python 还是有优势的。

2：

数据是怎样存储的？

内存：与 cpu 沟通的桥梁计算机中所有的程序都是在内存中运行的。暂时存放 cpu 中的运算数据以及与硬盘等外部存储器的交换数据。

抽象来讲：一个开关，有两种状态，开启和关闭，一种对应状态对应 1，一种状态对应 0 所有的内存都是以二进制形式存入。将八个开关放在一个房间，这个房间称为一个‘字节’，一个开关表示‘一位’，每个房间都有门牌号，看做‘地址’，多个房间堆叠起来组成的摩天大厦就是‘内存’。

单位：

1bit：（可以读作一个比特位）对应一个位

8bit：== 一个字节

字符：可以理解为打印出来的符号，如一个单词，一个汉子，一个标点符号等。

1024bit == 1K

1024K == 1M

1024M == 1G

1024G == 1T

内存的地址用 16 进制数表示的。8 进制和 16 进制都可以很轻松和 2 进制抓换。

进制转换：10 转 2 进制

$10/2 = 5 \dots 0$ ---> $5/2 = 2 \dots 1$ ---> $2/2 = 1 \dots 0$ ---> $1/2 = 0 \dots 1$

倒取余数 1010 即 $10 = 1010 (2)$

2 进制转 10 进制：（从当前数字的右边起，第一位为 0 位）

二进制-》十进制
当前的数字，乘以2的位数次方，最后相加
 $0 \times 2^0 + 1 \times 2^1 + 0 \times 2^2 + 1 \times 2^3 = 0 + 2 + 0 + 8 = 10$
 $1010(2) \rightarrow 10(10)$

原码、反码、补码的概念

- 1 原码：规定了字节数，写明了符号位，就得到了数据原码。
- 2 反码：正数的反码就是其原码..

绝对路径：从根目录开始链接的路径。

相对路径：不从根目录开始链接的路径。

continue 语句与 break 语句

一：break 语句。作用：跳出 for 和 while 循环。注意：只能跳出距离它最近的一层循环。

```
for i in range(10):  
    print(i)  
    if i == 5:  
        break
```

0
1
2
3
4
5

同时需要注意的是：循环语句可以有 else 语句，但如果是 break 导致循环结束，else 语句就不会执行了。

```
num = 1  
while num <= 10:  
    print(num)  
    if num == 3:  
        break  
    num += 1  
else:  
    print("sunck is a good man")
```

#注意：循环语句可以有else语句，break导致循环截止，不会执行else下面的语句

二：continue 语句。与 break 类似，但是有区别。跳过当前循环中的剩余语句。然后继续下一次循环。

```
for i in range(5):  
    print(i)  
    if i == 3:  
        continue  
    print('*****')
```

0

1

2

3

4

字符串

字符串拼接：用+

字符串重复：用*

查找字符：用索引

```

str11 = "sunck is a good man!"
print(str11[1])
#str11[1] = "a" #字符串不可变
#截取字符串中的一部分
str13 = "sunck is a good man!"
#从给定下标出开始截取到给定下标之前
str15 = str13[6:15]
#从头截取到给定下标之前
str16 = str13[:5]
#从给定下标处开始截取到结尾
str17 = str13[16:]
print("str17 =", str17)

```

格式化字符串

```

a = 12.345

print('a is %f' % a)
print('a is %.3f' % a) # 默认保持6为小数如果要指定前面

```

test (1) ×

```

C:\Users\Administrator\Anaconda3\python.exe C:/User
a is 12.345000
a is 12.345

```

转义字符\'

```

print('frank is a \'good\' man!') #转义字符也可以转义单引号和双引号
多行打印可以用\n 换行也可以用"""打印，下面两个打印是等价的。
print('Good\nhandsome\nnice')
print("""
good
handsome
nice
""")

```

制表符\t

当需要打印很多\n时，或者很多制表符等特殊字符串时，可以在前面加'r'所有的字符都不会当做转义字符

```
print (r"\\t\\\\\\\\\\\\")
```

```
test (1) x
```

```
C:\Users\Administrator\Anaconda3\python.exe C
\\t\\\\\\\\\\\\
```

因此'r'加在路径前面可以很方便打印完整的路径不然所有的/前面都要再加一个/。

字符串操作的方法

1: eval(str) # eva 是 evaluate (评价, 求...的值) 的缩写

将字符串当成有效的表达式来求值, 并返回计算的结果。

```
print (type(eval('1213')))
```

```
test (1) x
```

```
C:\Users\Administrator\Anaconda3\python.exe C
<class 'int'>
```

将字符串变成了整数 1213

2 : len() 注意不是字节的长度, 是字符串的长度, (即字符串的个数)

```
#len(str)
```

```
#返回字符串的长度(字符个数)
```

```
print(len("sunck is a good man"))
```

3 : (str).lower() 将字符串中的大写字母转换为小写字母。

PS : 因为字符串是不可变的, 所以说原字符串并没有改变。字符串中的数字不改变。

```
a = 'SUNK is a good man!'
s = a.lower()
print(s)
print(a)
```

```
test (1) x
```

```
C:\Users\Administrator\Anaconda3\python.exe C:/
sunk is a good man!
SUNK is a good man!
```

4 : str.upper()与转换成大写字母, 用法与 lower 一样。

5 : str.swapcase() 将原字符串中小写—>大写, 大写--->小写

6 : str.capitalize() 将原字符串中的首字母转边成大写, 其它的都变成小写。

7 : str.title() 将元字符串的每个单词的首字母变成大写, 其它字母小写。

8 : str.center(width,fillchar):其中 width 意思是将字符串的长度变成 width 参数长, 不足的, 用 filchar 填充。当字符串本身长度大于 width 参数值时, 输出原字符串。


```
a = 'AbCd efg hij klm'
s = a.center(20, '*')
print(s)
```

test (1) ×

```
C:\Users\Administrator\Anaconda3\python.exe
**AbCd efg hij klm**
```

9 : str.ljust(width[fillchar]) 以左对齐的方式返回 width 长度的字符串，fillchar 参数可写可不写不写默认空格，写的啥符号就以啥符号填充。

```
a = 'AbCd'
s = a.ljust(10, '*')
print(s)
```

test (1) ×

```
C:\Users\Administrator\Anaconda3\python.exe
AbCd*****
```

10 : str.rjust(width[fillchar]) 与 9 相反，右对齐的方式。

11 : str.zfill(width) 右对齐，前面补 0

```
a = 'AbCd'
s = a.zfill(10)
print(s)
```

test (1) ×

```
C:\Users\Administrator\Anaconda3\python.exe
000000AbCd
```

12 : str0.count(str,[start],[end]) 从 str0 字符串中找 str 出现的次数。后面参数可以是指定从 str0 第 start 个字符串到第 end 个串中间找。

```
a = 'a bb cc bb aa bb'
s = a.count('bb')
t = a.count('bb', 1, 4)
print(s)
print(t)
```

test (1) ×

```
C:\Users\Administrator\Anaconda3\python.exe
3
1
```

13 : str.find(str,[start],[end]) 从左至右检测 str 字符串是否包含在字符串中。可以指定范围，默认从头到尾，得到的是第一次出现的开始下标。找不到的返回-1。

```

199 #find(str[, start[, end]])
200 #从左向右检测str字符串是否包含在字符串中，可以指定范围，默认从头到
    尾。得到的是第一次出现的开始下标，没有返回-1
201 str30 = "kaige is a very very nice man"
202 print(str30.find("very"))
203 print(str30.find("good"))
204

```

Run String(字符串)

11

-1

14 : str.rfind()与 13 一样，但是找的方法是从右至左找。

15 : str.index(str,[star],[end]) 与 find 方法一样，但是找不到的时候报错。

16 : str.rindex() 与 15 一样只是从右向左边找。

17 : str.lstrip() 截掉字符串左侧指定的字符。默认为空格。

```

226 str33 = "*****kaige is a nice man"
227
228 print(str33.lstrip("*"))
229
230
231

```

Run String(字符串)

11

16

kaige is a nice man

18 : str.rstrip() 截掉字符串右侧指定的字符。默认为空格。

19 : str.strip() 截掉字符串中开头和结尾的指定字符（可以一次指定多个，只要是被指定的字符在首或尾都会被去掉）。ps 中间的不能截。

```

a = '**a bb cc bb **aa bb*****'
s = a.strip('*')
print(s)

```

test (1) x

C:\Users\Administrator\Anaconda3\p

a bb cc bb **aa bb

一次去掉多个：

```

str = '* \n \t frank is a good man ! ** **'
print(str.strip('* \n \t f!'))
rank is a good man

```

20 : split(str='')以 str 为分隔符截取字符串。并以列表形式返回。指定 num 则指定截取 num 个字符串。

```

str38 = "sunck**is*****a***good*man"
print(str38.split("*"))
['sunck', '', 'is', '', '', '', '', '', 'a', '', '', 'good', 'man']

```

21 : splitlines()按照 ('/r'/'n'/'r,n') 分割，keepends==True 会保留换行符。


```
str40 = '''sunck is a good man!
sunck is a nice man!

sunck is handsome man!
'''

print(str40.splitlines())
```

```
['sunck is a good man!', 'sunck is a nice man!', 'sunck is handsome man!']
```

22 : max()、min() 按照 ascii 码值大小找出最大最小的字符，打印第二行为空格。

```
str43 = "sunck is a good man!z"
print(max(str43))
print("*"+min(str43)+"*")
```

```
z
```

```
* *
```

23 : str.replace(oldstr, newstr, count) 将指定元素替换成指定元素。count 为替换指定次数。

```
str44 = "sunck is a good good good man"
str45 = str44.replace("good", "nice")
print(str44)
print(str45)
```

```
sunck is a good good good man
sunck is a nice nice nice man
```

24 : 创建一个映射表替换。(不可控，很少用。)

```
#创建一个字符串映射表
```

```
print("*****")
```

```
#          要转换的字符串          目标字符串
```

```
t46 = str.maketrans("ac", "65")
```

```
# a—6    c—5
```

```
str47 = "sunck is a good man"
```

```
str48 = str47.translate(t46)
```

```
print(str48)
```

```
sun5k is 6 good m6n
```

25 : str.startswith(str, start=0, end=len(str))默认从下标 0 开始到最后，也可以指定，

下面中如果默认可以打印出 True

```
#startswith(str, start=0, end=len(str))
str49 = "sunck is a good man"
print(str49.startswith("sunck", 5, 16))
```

26 : str.endswith(str,start=0, end=len(str)) 用法和 25 一样，看末尾有没有指定的结尾的。

27 : 计算机里面，编码方法有很多种，英文的一般用 ascii,而中文有 unicode , utf-8,gbk,utf-16 等等。

unicode 是 utf-8,gbk,utf-16 这些的父编码，这些子编码都能转换成 unicode 编码，然后转化成子编码，例如 utf8 可以转成 unicode，再转 gbk，但不能直接从 utf8 转 gbk 所以，python 中就有两个方法用来解码（decode）与编码（encode），解码是子编码转 unicode，编码就是 unicode 转子编码

字符串在 Python 内部的表示是 unicode 编码，因此，在做编码转换时，通常需要以 unicode 作为中间编码，即：先将其他编码的字符串解码（decode）成 unicode，再从 unicode 编码（encode）成另一种编码。

1 : decode 的作用是将其他编码的字符串转换成 unicode 编码。

如 str1.decode('gb2312')，表示将 gb2312 编码的字符串 str1 转换成 unicode 编码。

2 : encode 的作用是将 unicode 编码转换成其他编码的字符串。

如 str2.encode('gb2312')，表示将 unicode 编码的字符串 str2 转换成 gb2312 编码。

28 : str.isalpha() 如果字符串中至少有一个字符并且所有的字符的全部都是字母的时候返回 True，否则 False。

```
str54 = "sunckisagoodman"
print(str54.isalpha())
```

True

29 : str.isalnum()如果字符串中至少有一个字符并且所有的字符的全部都是字母或数字的时候返回 True，否则 False。

```
str55 = "123"
print(str55.isalnum())
```

True

30 : str.isupper()如果字符串中至少有一个英文字符并且其全部英文字符都是大写的英文字母的时候返回 True，否则 False。

```
a = 'A12345#¥%.....6'
b = a.isupper()
print(b)
```

True

31 : str.lower() 与 30 用法一样，小写。

32 : str.title() 如果字符串是标题化（每个单词都是以大写字母开始）的话返回 true，否则 false。

33 : str.isdigit() 如果字符串只包含数字字符，返回 true，否则 false。

digit

英 ['dɪdʒɪt]   美 ['dɪdʒɪt]  

n. 数字; 手指, 足趾; 一指宽

34 : str.isnumeric() 与 isdigit() 的用法一样。

35 : str.isdecimal() (decimal 十进制的。)字符串只包含十进制字符。

35 : str.isspace() 字符串中只包含空格返回 true 否则返回 false。

今天在做老师布置的作业时候，找出三位数中水仙花数（各个位上数的 3 次方相加等于本身）

自己代码

```
for i in range(100,1000):
    g = i%10
    s = (i%100 - g)/10
    b = (i - 10*s - g)/100
    if i == g**3 + s**3 + b**3:
        print('shuixianhuashu',i)
```

test (1) ×

```
C:\Users\Administrator\Anaconda3\python.e
shuixianhuashu 153
shuixianhuashu 370
shuixianhuashu 371
shuixianhuashu 407
```

忘记了'/'就是取整，我求十位数字用的是(i%100 - g)/10。其实用'//'取整就是 i//10/10 就可以了。

字符串的大小比较

从左起第一位开始比较，谁的 ASCII 码值大谁就大。如果相等则比较下一个 ASCII 码值，谁大就谁大。

列表

列表的本质：是一种有序的集合。

列表的操作：

1 列表的组合

```
list5 = [1, 2, 3]
list6 = [4, 5, 6]
list7 = list5 + list6
print(list7)
```

2 列表的重复

```
list8 = [1, 2, 3]
print(list8 * 3)
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

3 判断元素是否在列表中

用成员将运算符 in

```
list9 = [1, 2, 3, 4, 5]
print(3 in list9)
True
```

注意列表的切片将产生一个新的列表，不会改变原列表。

列表的方法

1: list.append() 追加元素到列表的最后。（注意追加的多个也会当成一个元素。）

```
list = [1, 2, 3, 4, 5, 6]
list.append([7, 8, 9])
print(list)
[1, 2, 3, 4, 5, 6, [7, 8, 9]]
```

2 : list.extend() 在末尾一次性追加另一个列表中的多个值。（原理是用 for 循环迭代添加）

```
list = [1,2,3,4,5,6]
list.extend([7,8,9])
print (list)
```

[1, 2, 3, 4, 5, 6, 7, 8, 9]

3 : list.insert(index,object) 在列表的下标 index 处添加一个元素，不覆盖源数据，源数据顺延。要往前面添加则 index 为 0 即可。

```
1 list = [1,2,3,4,5,6]
2 list.insert(2,5)
3 print (list)
```

[1, 2, 5, 3, 4, 5, 6]

4 : list.pop(x = list[-1]) 删除列表中的指定下标处元素，（默认移除最后一个。）并且能返回删除的数据。并可以将移除的最后一个值赋值给变量：

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

[1, 2, 3, 5]
4

5 : list.remove()移除列表中的某个元素，第一个匹配的结果。

```
list = [1,2,3,4,5,2,6]
list.remove(2)
print (list)
```

[1, 3, 4, 5, 2, 6]

6 : list.clear() 清除列表中的全部元素。

```
list = [1,2,3,4,5,2,6]
list.clear()
print (list)
```

[]

7 : list.index() 从列表中找到第一个匹配的某个值的下标。还可以圈定一个范围

```
1 list = [1,2,3,4,5,2,6]
2 print (list.index(2))
3 list = [1,2,3,4,5,2,6]
4 print(list.index(2, 2, 6))
```

1
5

8 : len(list) 列表中元素的个数。


```
list = [1,2,3,4,5,2,6]
print(len(list))
```

7

9 : max(list)列表中的最大值(注意不是 list.max)

```
1 list = [1,2,3,4,5,2,6]
2 print(max(list))
```

6

10 : 最小值用 min 用法与 max 一样。

11 : list.count() 计算某个元素在列表中出现的次数。

```
list23 = [1, 2, 3, 4, 5, 3, 4, 5, 3, 3, 5, 6]
print(list23.count(3))
```

4

配合 remove 就可以灵活运用。删除某个列表中的所有指定元素。

```
num24 = 0
all = list23.count(3)
while num24 < all:
    list23.remove(3)
    num24 += 1
print(list23)
```

12 : list.reverse() 将列表倒序

```
list25 = [1, 2, 3, 4, 5]
list25.reverse()
print(list25)
```

[5, 4, 3, 2, 1]

13 list.sort() 将列表排序：升序

```
list26 = [2, 1, 3, 5, 4]
list26.sort()
print(list26)
```

[1, 2, 3, 4, 5]

14 : 列表拷贝直接赋值属于浅拷贝，.copy 属于深拷贝，在堆区重新生成了一个对象，而变量的赋值仅仅是在内存的栈区创建的。栈区地址指向堆区，将数据取出来，栈区程序结束之后所有的内存都会自动释放，但是堆区的内存理论上是需要程序员手动释放，但是现在的电脑也能自动释放堆区的内存。

15 : list (tuple) 将元组转换成为列表


```
list1 = list((1,2,3,4))
print (list1)
```

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

16 : `enumerate(list)` 枚举出来列表的所有下标及所对应的元素。可以用一个参数接收出一对下标和值做对应的元素，也可以用两个参数分别接收下标和对应的元素。

```
for i in enumerate([1,2,3,4,5,6]):
    print(i)
```

```
(0, 1)
(1, 2)
(2, 3)
(3, 4)
(4, 5)
(5, 6)
```

```
for i, j in enumerate([1, 2, 3, 4, 5, 6]):
    print(i,j)
```

```
0 1
1 2
2 3
3 4
4 5
5 6
```

17 : `'*'.join(list)` 将一个指定的字符串作为分隔符将列表中的元素组合成一个字符串。

`str.join(元组、列表、字典、字符串)` 之后生成的只能是字符串。所以很多地方很多时候生成了元组、列表、字典后，可以用 `join()` 来转化为字符串。

```
list=['1','2','3','4','5']
print("".join(list))
```

结果 : 12345

```
seq = {'hello':'nihao','good':2,'boy':3,'doiido':4}
print('.'.join(seq))    #字典只对键进行连接
```

结果 : hello-good-boy-doiido

元组

1 元组取值

`tuple[]` 注意是用 `[]` 不是用 `()`

2 元组的元素不可变，但是如果元素是一个列表，那么这个元素的内部元素是可以改变的。

```
a = (1,2,3,[4,5,6])
a[3][1] = 0
print(a)
```

```
(1, 2, 3, [4, 0, 6])
```

3 元组内部元素不可以删除，但是可以删除整个元组。

```
a = (1,2,3,4,5)
del a
print(a)
```

```
NameError: name 'a' is not defined
```

元组的操作：

1：元组可以相加

```
t7 = (1, 2, 3)
t8 = (4, 5, 6)
t9 = t7 + t8
print(t9)
(1, 2, 3, 4, 5, 6)
```

2：元组重复

```
t10 = (1, 2, 3)
print(t10 * 3)
(1, 2, 3, 1, 2, 3, 1, 2, 3)
```

3：判断元素是否在元组内：

```
t11 = (1, 2, 3)
print(1 in t11)
True
```

4：元组的截取。元组名[开始下标:结束下标]从开始下标截取到结束下标之前。

```
t12 = (1, 2, 3, 4, 5, 6, 7, 8, 9)
print(t12[3:7])
(4, 5, 6, 7)
```

5：元组的方法

1 len(tuple) 返回元组中元素的个数。

- 2 max(tuple) 返回元组中的最大值
- 3 min(tuple) 返回元组中的最小值
- 4 tuple(list) 将列表转元组

```
list = [1, 2, 3]
t15 = tuple(list)
print(t15)
(1, 2, 3)
```

总结：其实在 python 中元组和列表是差不多的，元组可以看做是不可变的列表，那为什么还要搞个元组搞个列表呢，因为安全，元组的数据不可修改，更加安全，在以后编程中间，能用元组的就尽量用元组。

(字符串（中的元素）、元组、列表都是可以遍历的)

字典

字典：使用键值对的形式存储数据，具有极快的查找速度。

key 的特性：

- 1：字典中的每一个 key 必须是唯一的。
- 2：key 必须是不可变的对象，如：字符串，整数等。
- 4：list 是可变的不能作为 key。大多数情况下是用字符串作为 key。
- 5：字典是无序的。

字典中获取对象

在字典中寻找数据可以用 get 方法代替 print 方法，好处是：如果在的话会将 key 所对应的值返回，如果字典中没有这个 key 的话，也不会报错，会返回 None，而 print 的话没有就会报错。因此 get 还可以用与循环条件的判断。

字典中存在 key：都可以返回 value。

```
dic = {'yangchao':160, 'liuliu':180}
print(dic['yangchao'])
print(dic.get('yangchao'))
```

160

160

如果字典中不存在 key :

```
dic = {'yangchao':160, 'liuliu':180}
print(dic.get('yang'))
```

None

```
dic = {'yangchao':160, 'liuliu':180}
print(dic['yang'])
```

Traceback (most recent call last):

File "C:/Users/Administrator/PycharmProjects/千锋教程/caogao.py", line 8, in <module>:
 print(dic['yang'])
KeyError: 'yang'

字典中添加对象

添加新的 key :

```
dic = {'yangchao':90, 'liuliu':95}
dic['yangzhi'] = 92
print(dic)
```

{'yangchao': 90, 'liuliu': 95, 'yangzhi': 92}

如果添加的是同样的 key 不同的 value 则会覆盖原先的旧 key 的 value 值。其实就是对 key 的修改。

```
dic = {'yangchao':90, 'liuliu':95}
dic['yangchao'] = 100
print(dic)
```

{'yangchao': 100, 'liuliu': 95}

删除对象

```
dic = {'yangchao':90, 'liuliu':95}
dic.pop('yangchao')
print(dic)
{'liuliu': 95}
```

遍历对象

- 1 遍历的话只遍历所有的 key。

```
dic = {'yangchao':90, 'liuliu':95, 'frank':98}
for i in dic:
    print(i)
yangchao
liuliu
frank
```

如果需要取出对应的值：print(dic[i]) 就是。

- 2 如果需要直接循环字典里面的 value，也可以做到：用 dic.values()。

```
dic = {'yangchao':90, 'liuliu':95, 'frank':98}
for i in dic.values():
    print(i)
90
95
98
```

- 3 如果想一次性遍历字典中的 key 和 value 也可以做到。用 dic.items() 该方法将字典返回可遍历的(键, 值) 元组为元素组成的列表。

```
dic = {'yangchao':90, 'liuliu':95, 'frank':98}
for i, j in dic.items():
    print(i,j)
# print(dic[i])

yangchao 90
liuliu 95
frank 98
```

- 4 用枚举的方法可以查看排序（字典本身无序，查看到的是往里面存的顺序）和 key

```
dic = {'yangchao':90, 'liuliu':95, 'frank':98}
for i, j in enumerate(dic):
    print(i,j)
0 yangchao
1 liuliu
2 frank
```

用字典和 list 比较：（可以看做 dictionary 和 list 的优缺点。）

1：查找和插入的速度极快。不回随 key-value 的增加而变慢。（list 是从头到尾遍历，所以速度多就会查找慢。）

2：需要占用的内存比 list 大，内存浪费多。

集合

set：类似字典，是一组 key 的集合，但是不存储 value。
本质：是一种无序，无重复的元素的集合。
重复元素在 set 中会自动过滤。

集合操作

--创建一个集合：

```
a = {1,2,3}
print(type(a), a)
<class 'set'> {1, 2, 3}
```

也可以用一个 list 或者 tuple 或者 dict 作为输入集合。

```
s1 = set([1,2,8,4,5])
print(s1)
s2 = set((1,2,3,4,5,6))
print(s2)
s3 = set({'1':'good',2:'nice',3:'handsome'})
print(s3)
```

```
{1, 2, 4, 5, 8}
{1, 2, 3, 4, 5, 6}
{1, 2, 3}
```

-----添加 set.add() (可以添加重复的，但是不会有效。)

```
s1 = set([1,2,8,4,5])
s1.add(6)
print(s1)
{1, 2, 4, 5, 6, 8}
```


需要注意:add()中的参数不能是可变的list。也不可以是字典。因为字典是可变的。

```
s1 = set([1,2,8,4,5])
s1.add([7,8,9])
print(s1)
```

Traceback (most recent call last):

File "C:/Users/Administrator/PycharmProjects/千锋教程/caogao.py", line 3, in <module>
s1.add([7,8,9])

TypeError: unhashable type: 'list'

但是可以是tuple，将整个元组当成一个元素加入进去。

```
s1 = set([1,2,8,4,5])
s1.add((7,8,9))
print(s1)
```

```
{1, 2, 4, 5, 8, (7, 8, 9)}
```

----添加 set.update 将 list、tuple、dict 打碎整个插入到 set 中间。

```
s1 = set([1,2,8,4,5])
s1.update([7,8,9])
print(s1)
```

```
C:\Users\Administrator\PycharmProjects\千锋教程\caogao.py
{1, 2, 4, 5, 7, 8, 9}
```

-----删除.remove() 要删除哪个元素就写哪个值。

```
s1 = set([1,2,8,4,5])
s1.remove(2)
print(s1)
```

```
{1, 4, 5, 8}
```

----可以遍历

```
s1 = set([1,2,8,4,5])
for i in s1:
    print(i)
```

```
1
2
4
5
8
```

-----枚举：可以按顺序打印下标出来，但是没有意义，因为不能根据下标取值。

```
s7 = set([1,2,3,4,5])
```

```
for index, data in enumerate(s7):
    print(index, data)
```

```
0 1
1 2
2 3
3 4
4 5
```

-----交集和并集结果还是以集合的形式展现。并用'{'表示。

```
s8 = set([1, 2, 3])
s9 = set([2, 3, 4])
#交集
a1 = s8 & s9
print(a1)
print(type(a1))

{2, 3}
<class 'set'>
```

----set 与 list、tuple 之间的转换。

set 转 list

```
s3 = {1, 2, 3, 4}
l3 = list(s3)
print(l3)

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

set 转 tuple

```
#set-->tuple
s4 = {2, 3, 4, 5}
t4 = tuple(s4)
print(t4)

(2, 3, 4, 5)
```

迭代器

迭代器：不但可以作用于 for 循环，还可以被 next () 函数不断地调用并返回下一个值。直到最后会抛出一个错误。StopIteration 错误。说明无法继续返回下一个值。

可以被 next () 函数调用并不断返回下一个值的对象，叫做迭代器 (Iterator)

可迭代对象：即可以直接作用于 for 循环的对象，称为可迭代对象 (iterable) 可以用 isinstance () 方法去判断一个数据是不是可迭代对象、(用 instance 方法需要先 from collections import Iterable)

一般分两类：

- 一：结合类数据类型。list、tuple、dict、set、string
- 二：generator。包括生成器和带 yield 的 generator function

```
print(isinstance([], Iterable))
print(isinstance(), Iterable)
print(isinstance({}, Iterable))
print(isinstance(" ", Iterable))
True
True
True
True
```

最基本的迭代器：(x for x in range(10))

```
l = (x for x in range(5))
print(next(l))
print(next(l))
print(next(l))
print(next(l))
print(next(l))
print(next(l))
0
1
2
3
4
StopIteration
```

```
l = (x for x in (1,2,5,6,7))
print(next(l))
print(next(l))
print(next(l))
print(next(l))
print(next(l))
1
2
5
6
7
```

转换成 Iterator 对象。用 `iter()`

```
a = iter([1, 2, 3, 4, 5])
print(next(a))
print(next(a))
```



函数

函数概述：在一个完整的项目中，某些功能会反复地使用，那么我们会将这个功能封装起来，成为函数。当我们要使用功能的时候，直接调用函数即可。

函数的本质：函数就是对功能的封装，函数的内容从 ':' 开始，函数内容即函数封装的功能。

为什么要使用函数：

1：简化代码结构，增加代码的复用度。

2：如果想修改某些功能和调试某个 BUG，只需要修改对应的函数即可。

return 用于结束函数（只要遇到 return 就结束了），并返回信息给函数的调用者。

最后 return 的表达式可以不写，如果不写的话相当于默认返回 None。

-----注意函数调用的本质：实参给形参赋值的过程。

-----函数调用的时候，就算定义的函数没有参数，()也不能省略。

-----形参：定义函数时候小括号中的变量，所以形参的本质是变量。

-----参数必须要按照顺序传递，目前来说，个数要对应。

注意：如果定义一个函数 fn，此时的 fn 为一个函数对象，指向内存中的某个地址，如果要调用 fn 的话必须要加上括号即 fn() 为调用该函数。

函数参数的传递分两种：

****值传递**：传递不可变的类型。（str,tuple,number）（函数内部可以改变参数，但是外部全局变量不会发生变化。）

```
def func(num):
    num = 10
    return num
num1 = 20
a = func(num1)
print(a)
print(num1)
```

10
20

****引用传递：**传递的是可变类型。(list dict set) (函数调用可变变量，可变变量的值随函数发生变化。) 如果想要外部不变可以传递对象的副本即.copy(),如果是列表可以传list[:]

```
def func(list1):
    list1[0] = 100
    return list1
li = [1, 2, 3, 4]
a = func(li)
print(a)
print(li)
```

[100, 2, 3, 4]
[100, 2, 3, 4]

函数参数

关键字参数：

允许函数参数调用时参数顺序与定义不一致。

```
def information(name, age):
    print(name, age)
information(age=18, name='frank')
```

frank 18

默认参数：(传参用传，不传用默)

调用函数时，如果没有传递参数，则使用默认参数。

```
def information(name, age=18):
    print(name, age)
information('yangcaho')
```

yangcaho 18

```
def information(name, age=18):
    print(name, age)
information('yangcaho', 20)
```

yangcaho 20

以后需要用到默认参数，最好能将默认参数放到最后。下代码参数传递错误原因：只传递了一个参数，默认会传给 name, 所以导致 age 没有参数，少了参数所以出错。

```
def information(name=frank, age):
    print(name, age)
information(18)
```

SyntaxError: non-default argument follows default argument

不定长参数（参数的装包）

概念：能处理比定义时更多的参数，加了*的变量，会存入所有未命名的变量参数。如果在函数调用时没哟指定参数，它就是一个空元组。一般用*args

注意：*arg 只能并且会接受所有的**位置实参**，（不能接受关键字参数）并且会将所有的实参统一保存到一个元组中。（装包）带*参数不一定要写到最后，但是带*参数后面只能放关键字参数。

不定长参数方式一：

```
def information(name, *arr):
    print(name)
    for i in arr:
        print(i)
information('frank', 'nice', 'good', 'handsome')
```

```
frank
nice
good
handsome
```

不定长参数方式二：**args 接收所有的关键字参数。以字典的方式保存起来。key 对应参数名，value 对应参数值。**args 只能有一个，并且必须写在最后。

```
def information(**kwargs):
    print(kwargs)
    print(type(kwargs))
information(a=1, b=2, c=3)
```

```
{'a': 1, 'b': 2, 'c': 3}
<class 'dict'>
```

综合以上两种不定长参数的传递类型，以下方式就是可以传递任意参数的函数写法。


```
def function(*args, **kwargs):
    pass

def function(*args, **kwargs):
    print(args)
    print(kwargs)
    print('ych is a good man!')
function('yangchao', 1, 2, 3, 4, a=6, b=6, c=12,)
('yangchao', 1, 2, 3, 4)
{'a': 6, 'b': 6, 'c': 12}
ych is a good man!
```

参数的解包

传递参数时，也可以在序列类型的参数前面添加*号，这样会自动将序列中的元素依次作为参数传递。(要求序列中元素的个数和形参的个数一致)

```
def fn(a,b,c):
    print(a)
    print(b)
    print(c)
t = (10,20,30)
fn(*t)
10
20
30
```

也可以创建一个字典来解包

```
d = {'a':100, 'b':200, 'c':300}
fn(**d)
100
200
300
```

匿名函数

关键字 lambda 表示匿名函数，语法：lambda 参数列表：返回值

用匿名函数有个好处，因为函数没有名字，不必担心函数名冲突。此外，匿名函数也是一个函数对象，也可以把匿名函数赋值给一个变量，再利用变量来调用该函数：

```
def fn(a, b):
    return a+b
则 fn 函数等价于 lambda a,b : a+b
```

偏函数

把函数的一个参数固定住，生成一个新的函数，

需要语句：import functools 中的 functools.partial(函数名，参数固定)

```
int3 = functools.partial(int, base = 2)
print(int3("111"))
```

71

变量的作用域

作用域:变量的可使用范围。程序的变量不是在所有位置都可以使用的，访问的权限决定于变量赋值的位置。

注意：当我们使用变量时候，会优先在当前作用域中寻找该变量，找到就使用，没有则继续去上一级寻找，依次循环。直到找向全局变量，如果还没找到就会报错。is Not defiend...

如果希望在函数内部改变全局变量，则需要在函数内部用 global 关键字来声明变量。

作用域分为：局部变量、全局变量、内建作用域

命名空间 (namespace)

指的是变量存储的位置，每一个变量都需要存储到指定的命名空间当中。每一个作用域也会有一个对应的命名空间。全局命名空间保存全局变量，函数命名空间保存函数的变量、

命名空间实际上就是一个专门用来存储变量的字典。

用 locals() 函数来获取当前作用域的命名空间。如果在全局作用域调用 locals() 则获取全局命名空间，在函数调用则获取函数命名空间。global() 函数可以在任意的位置获取全局命名空间。

错误处理方式一

1 try...except...else

格式：(最后一个 else 可有可无)

作用：检测 try 中间的语句是否有错误。捕获不同错误做不同处理。

```

try:
    语句t
except 错误码 as e:
    语句1
except 错误码 as e:
    语句2
.....
except 错误码 as e:
    语句n
else:
    语句e
'''

```

逻辑：1：当 try 语句执行出现错误的时候，会匹配第一个错误码，如果匹配上了就执行对应的语句。

2：如果 try 语句执行出现错误，但是 except 没有匹配到异常，错误将会被提交到上一层的 try 语句。或者到程序的最上层。

3：如果 try 语句执行没有出现错误，将会执行 else 下的语句。（当有 else+语句的时候。）

2 也可以使用 except 而不使用任何错误类型。

```

try:
    print(4 / 0)
except:
    print("程序出现了异常")

```

不管代码中有任何异常都会执行 print 语句中的内容。

3 使用 except 带着多种异常。

```

#使用except带着多种异常
try:
    pass
except (NameError, ZeroDivisionError):
    print("出现了")

```

注意 1 错误其实是 class(类)，所有的错误都继承自 BaseException，所以在捕获的时候，它捕获了该类型的错误，并且还将把其所有子类的错误也一网打尽。

2 跨越多层调用，比如，main() 调用了 func2()，func2()调用了 func1()如果是 func1 出现了错误，这时只要 main() 捕获到了就可以处理。

```

def func1(num):
    print(1 / num)
def func2(num):
    func1(num)
def main():
    func2(0)
try:
    main()
except ZeroDivisionError as e:
    print("****")

```

错误处理方式二

try...except...finally 作用：不管语句 t 有没有错误，都会执行最后的 finally 语句。

```

try:
    语句t
except 错误码 as e:
    语句1
except 错误码 as e:
    语句2
.....
except 错误码 as e:
    语句n
finally:
    语句f

```

这里需要注意一点：如果 try 中出现错误，但是 except 没有捕获到错误的话，finally 语句会执行，但是执行完了之后程序就结束了，finally 后面所有语句都不会再执行了。

```

try:
    a = 3/0
    print(a)

finally:
    print('必须执行我')

print('woshi yangcaho')

```

必须执行我

```

Traceback (most recent call last):
  File "C:/Users/Administrator/Pychar
    a = 3/0
ZeroDivisionError: division by zero

```

如果 try 中出现错误，但是 **except** 捕获到了错误的话，finally 语句会执行，程序不会中
断，finally 后面语句照常执行。

```

try:
    a = 3/0
    print(a)
except ZeroDivisionError as e:
    print('为0了')
finally:
    print('必须执行我')

print('woshi yangcaho')

```

为0了

必须执行我

woshi yangcaho

错误处理方式三：断言

```

def func(num, div):

    assert (div != 0), "div不能为0"
    return num / div

print(func(10, 0))

```

```
Traceback (most recent call last):
  File "C:/Users/xlg/Desktop/Python-1704/day07/4-异常处理/断言.py", line 6, in <module>
    print(func(10, 0))
  File "C:/Users/xlg/Desktop/Python-1704/day07/4-异常处理/断言.py", line 3, in func
    assert (div != 0), "div不能为0"
AssertionError: div不能为0
```

断言的意思是在某个地方语言一下，如果和语言相斥的话就会打印自己输出的错误描述。

函数之递归

递归调用：一个函数如果调用了自身，成为递归调用

递归函数：一个会调用自身的函数叫做递归函数。

凡是循环能干的事，递归都能干，但是不是每个人都可以写出来。

老师总结递归函数的写法：

- 1 写出临界条件
- 2 找这一次和上一次的关系
- 3 假设当前函数已经能使用，调用自身计算上一次的结果，再求出本次的计算结果。

```
def add(n):
    if n == 1: # ①临界条件
        return 1
    else:
        return n + add(n-1) #②假设当前函数已经能使用，调用自身计算上一次的结果，再求出本次结果。
print(add(5))
```

代码解析：

$\text{sum}(5) = 5 + \text{sum}(4) = 5 + 4 + \text{sum}(3) = 5 + 4 + 3 + \text{sum}(2) = 5 + 4 + 3 + 2 + \text{sum}(1).$

$\text{sum}(1) = 1$

先计算最基层的临界值，然后层层往上分析，知道最后一个临界值分析与倒数第二个的关系，通过调用自身来将关系表达式写出来。（有点复杂）

高阶函数：

高阶函数的定义：一个函数接受一个函数作为参数，或者返回值为一个函数。那么这个函数就是高阶函数。

函数作为单数类

map

`map(fn, lsd)` 参数 `fn` 是一个函数，参数 `2` 是序列

功能：将传入的函数依次作用在序列中的每一个元素，并把结果作为新的 `Iterator` 返回。

可以将 `Iterator` 看做是一个惰性的列表，如果如果需要转化成列表可以用 `list()` 方法返回 `Iterator` 里面的值。

(插入)巧用函数：字符串转化成整型。

```
def change(str1):  
    return{'0':0, '1':1, '2':2, '3':3, '4':4, '5':5, '6':6, '7':7, '8':8, '9':9}[str1]  
a = '3478944'  
for i in a:  
    print(change(i))
```

#将整数元素的序列，转为字符串型

#[1, 2, 3, 4] -> ["1", "2", "3", "4"]

```
l = map(str, [1, 2, 3, 4])  
print(list(l))
```

reduce(fn, lsd)

不是内置库，需要导入 **from functools import reduce**

参数 fn 为函数，参数 lsd 为一个列表

功能：一个函数作用在序列上，这个函数必须接收两个参数，reduce 把结果继续和序列的下一个元素累计运算。

求列表中所有数的和：

```
from functools import reduce  
def sum(a, b):  
    return a+b  
a = reduce(sum, [1, 2, 3, 4, 5,])  
print(a)  
15
```

filter

filter 英语过滤的意思 (fn, lsd)：功能：用于过滤序列。将传入的函数依次作用的序列的每个元素，根据返回的是 True 还是 False 决定是否保留改元素。


sort

sort()用来对列表中的元素进行排序，默认直接比较列表中元素 ascii 值大小排序。

的大小。可以接收一个关键字参数 key，key 需要一个函数作为参数，然后每次将列表中的一个元素作为该函数的参数调用，并且使用调用后的返回值来比较元素大小。从而对列表排序。

默认排序：

```
l = ['aaa', 'bb', 'cccccc', 'e', 'ffff']  
l.sort()  
print(l)
```

```
['e', 'bb', 'aaa', 'ffff', 'cccccc']  
接收函数排序：  
l = ['aaa', 'bb', 'cccccc', 'e', 'ffff']  
l.sort(key=len)   
print(l)  
['e', 'bb', 'aaa', 'ffff', 'cccccc']
```

sorted

功能与 sort() 基本一致，调用方法不同，sorted() 是将可迭代对象（不仅限于列表）作为参数，key 作为关键字参数调用，然后返回一个新的列表，不改变原对象。（sort 改变了原列表）

如下：按绝对值大小排序方法：

```
list3 = [4, -7, 2, 6, -3]  
list4 = sorted(list3, key=abs)  
print(list3)  
print(list4)  
[2, -3, 4, 6, -7]
```

如果要按照降序排序的话，后面加参数 reverse=True (翻转的意思)

```
list1 = [5, 74, 2, 6, 3, 7, 5,]  
list2 = sorted(list1, reverse=True)  
print(list2)  
[74, 7, 6, 5, 3, 2]
```

函数返回值为函数类（闭包函数）

通过闭包我们可以创建一些只有当前函数能访问的变量，可以将一些私有数据藏到闭包中。详情见函数的装饰器。

闭包函数三个条件：

- ① 函数嵌套
- ② 将内部函数作为返回值
- ③ 内部函数必须要使用到外部函数的变量

```
def MakeAverager():
    nums = []

    def averager(n):
        nums.append(n)
        return sum(nums)/len(nums)
    return averager

s = MakeAverager()
print(s(10))
print(s(20))
10.0
15.0
```

闭包函数的意义：nums 为函数内部变量，只有内部函数 averager 可以调用，就可以防止外层全局变量设置相同的变量名 nums 或对其修改。

函数的装饰器

概念：是一个闭包，把一个函数做为参数，返回一个替代版的函数，本质上就是一个返回函数的函数。

下代码分析:在不改变 func1 的情况下增加功能在打印之前打印 '*****'
所以就定义一个函数以 func1 为参数，在此函数内部调用 func1，inner 的意义是将此装饰器函数的结果以 inner 函数的形式返回出来。

f=outer(func1)意思是 将 outer 调用 func1 之后，用一个变量 'f' 接收 outer 函数的返回值，（即 inner 函数）直接用 f() 就可以调用 inner 了。这样每次调用都需要赋值 f=outer(func1) 所以为了方便就有了一种更加直接的方法。在 func1 的前面 @outer（# 相当于执行 outer(func1)）@ 符号的意义：将装饰器应用到函数。

```
def func1():
    print("sunck is a good man")

def outer(func):
    def inner():
        print("*****")
        func()
    return inner

#f是函数func1的加强版本
f = outer(func1)

f()

*****
sunck is a good man
```

由此可见装饰器的作用：有一个函数在不能修改函数内部的情况下，要增加函数的一些功能，就可以用装饰器做成函数的加强版。

注意：可以同时为一个函数添加多个装饰器，这样函数将会按照从内向外顺序被装饰。可以理解为下面的装饰器又被上面的装饰器装饰。

通用装饰器

因为装饰器的修饰函数 inner() 中的参数需要和被装饰的函数参数相一致，所以可以用一下

装饰器通用装饰各种参数的函数。

```
def outer(func):  
    def inner(*args, **kwargs):  
        #添加修改的功能  
        func(*args, **kwargs)  
    return inner
```

单元测试

作用：用来对一个函数、一个类、或者是对一个模块进行正确性较校验的。

两种结果：单元测试通过，说明我们测试的对象功能正常。测试不通过：要么函数功能存在 bug，要么测试条件有误。

对函数进行单元测试（以下函数对一个定义的和加和减法函数进行测试。）

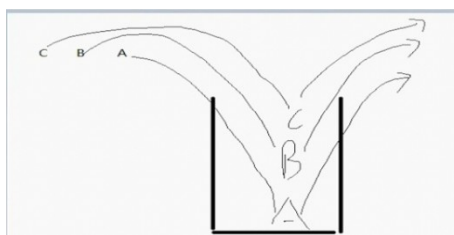
```
import unittest  
from 对函数进行单元测试 import mySum  
from 对函数进行单元测试 import mySub  
  
class Test(unittest.TestCase):  
    def setUp(self):  
        print("开始测试时自动调用")  
    def tearDown(self):  
        print("结束测试时自动调用")  
  
    #为了测试mySum  
    def test_mySum(self):  
        self.assertEqual(mySum(1,2), 3, "加法有误")  
    def test_mySub(self):  
        self.assertEqual(mySub(2,1), 1, "减法有误")  
  
if __name__ == "__main__":  
    unittest.main()
```

栈和队列

栈（stack）理解：如下图，按顺序将 A、B、C 三个数据先后存入栈，然后如果取出 A 的话，必需要先取出 C，B 才能取到 A。这就是栈的‘先进后出’特点。

数据存入栈的过程叫做**压栈**

从栈中取出数据的过程叫做**出栈**



队列：按顺序将 A、B、C 三个数据先后存入栈，然后如果取出 A 的话，只能按照顺序取出 A、B、C。这就是栈的'先进先出'特点。

文件读写

读文件

读一个文件的过程

1：打开文件、

`open(path, flag, [encoding], [errors])`

path:要打开文件的路径。

flag：打开的方式

r：以只读的方式打开文件，文件的指针放在文件的开头。

rb: 以二进制格式打开一个文件用于只读，文件指针在文件的开头。

r+：打开一个文件用于读写，文件的指针在文件开头。

w：打开一个文件只用于写入，如果该文件已经存在会覆盖（原来文件内容都会发生改变），如果不存在则创建一个新文件。

wb: **打开一个文件只用于写入二进制**。如果该文件已经存在会覆盖（原来文件内容都会发生改变），如果不存在则创建一个新文件。

w+：打开一个文件用于读写。如果该文件已经存在会覆盖（原来文件内容都会发生改变），如果不存在则创建一个新文件

a：以追加的方式打开一个文件。如果文件存在，文件指针放到文件末尾，

encoding: 编码格式

默认是 `encoding='utf-8'`

errors :错误处理'

`ignore`'忽略错误。

2：读文件、

1 读取全部文件内容：`f.read()` (适合读取文件比较小的)

2 读取指定长度的文件：`f.read(10)` (读取 10 个字符长度的文件。)

3 读取整行，包括'\n'字符。`f.readline()`

4 读取所有行，并返回一个列表，每行为列表中的一个元素：`r.readlines()`

5 修改指针的位置：`f.seek(0)`方法让指针回到字符转的指定下标（0）处。

3：关闭文件、

f.close()

一个完整的过程：(if f1 解释：如果打开了指针就进入文件了，f1 就肯定有数据了 (True) 就执行 f.close()，如果没打开就不用执行了。)

```
try:
    f1 = open(path, "r", encoding="utf-8")
    print(f1.read())
finally:
    if f1:
        f1.close()
```

一个更加简单的过程：不用close，自动关闭。最常用的方式！

```
with open(path, "r", encoding="utf-8") as f2:
    print(f2.read())
```

写文件

知识点：文件打开之后，用 f.write 写入的内容只是先存内存的缓冲区中间，如果不进行刷新的话不回保存到文件中去，但是有四中方式可以对缓冲区刷新，将在程序中 write 的内容放到文件中去。

- 1 执行文件关闭 f.close() 执行之后缓冲区的内容立刻存到文件中去。
- 2 遇到 '\n'
- 3 手动刷新立即存入。f.flush()
- 4 等到缓冲区自己满了之后就会自动刷新存入。

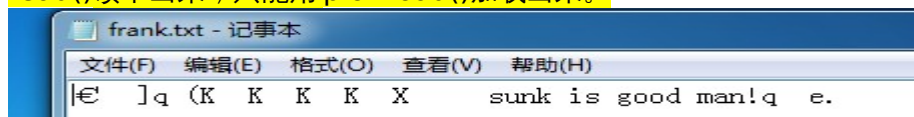
list、tuple、dict、set 的文件操作。

import pickle (数据持久性 (存入磁盘))

```
import pickle
mylist = (1,2,3,4,'sunk is good man!')
path = (r'C:\Users\Administrator\Desktop\frank.txt')
f = open(path,'wb')
pickle.dump(mylist, f)
f.close()

f1 = open(path,'rb')
lsit = pickle.load(f1)
print(lsit)
f1.close()
```

文件打开内容为：PS：对于 pickle.dump 存入的文件，读取的方式不能是 read() 使用 read() 读不出来，只能用 pick.load() 加载出来。




```
(1, 2, 3, 4, 'sunk is good man!')
```

OS 模块

os 包含了普遍操作系统的功能。

os.name 系统名称。（注意后面没有()不是调用 name 函数。）

os.uname 获取系统的详细信息（windows 不支持）

os.environ 获取系统所有的环境变量。

os.environ.get 获取指定环境变量

os.getcwd() 获取当前目录。

os.getcwd() 获取当前工作目录及当前 python 脚本所在的目录。

os.listdir(path) 返回 path 路径下的所有文件。以列表的形式返回。

os.mkdir(str) 在当前目录下创建一个新的目录。str 为目录名字

os.rmdir(str) 在当前目录下删除一个目录。str 为目录名字

os.remove() 文件名删除一个普通文件。

os.stat(文件名) 获取文件的属性。

os.rename(旧名字, 新名字) 重命名

os.system() 运行 shell 命令

- os.system('notepad') 打开记事本

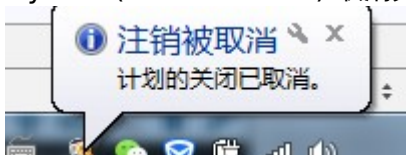
- os.system('write') 打开写字板

- os.system(mspaint) 打开画板

- os.system(msconfig) 打开系统设置

- os.system('shutdown -s -t 500') 设置关机时间 500 秒后关机

- os.system('shutdown -a') 取消关机计划。



- os.system(taskkill /f /im notepad.exe) 关闭记事本

os.path

os.path.abspath('.') 获取当前绝对路径。

os.path.join(p1,p2) 将路径 P1 和路径 p2 拼接起来。在爬取保存文件的时候可用到。

```
p1 = r"C:\Users\xlg\Desktop\Python-1704\day08"
p2 = r"sunck\abc\d"
#注意: 参数2里开始不要有斜杠
#r"C:\Users\xlg\Desktop\Python-1704\day08\sunck"
print(os.path.join(p1, p2))
```



```
C:\Users\xlg\Desktop\Python-1704\day08\sunck\abc\d
```

`os.path.split(path)` 拆分 `path` 路径，将最后一#个和前面拆分成一个元组的两个元素。
`os.path.splitext(path)` 拆分 `path` 最后的扩展名和前面的部分。如果没有扩展名则返回元组的最后一个元素为”

`os.path.isdir(path)` 判断 `filename` 文件名是不是一个目录。

`os.path.isfile(path)` 判 `filename` 文件是否存在。

`os.path.exists(path)` 判断文件目录是否存在。

`os.path.getsize(path)` 判断文件大小。（以字节为单位）

`os.path.dirname(path)` 判断文件所在的目录

`os.path.basename(path)` 获取文件名

这个在爬取网页图片的时候图片命名可以用到，很好用。

```
import os
a = 'http://pic1.sc.chinaz.com/Files/pic/pic9/201809/zzpic13941_s.jpg'
name = os.path.basename(a)
print(name)
zzpic13941_s.jpg
```

时间模块

time

UTC（世界协调时间）：格林尼治天文时间，世界标准时间在中国来讲是 `utc+8`（东八区）

DOS（夏令时）：一种节约能源而认为规定时间制度，在夏季调快 1 时。

时间的表现形式：

1 时间戳：

以整形或浮点型表示时间的一个以秒为单位的时间间隔，这个时间间隔的基础值是从 1970 年 1 月 1 日的零时开始算起。

2 元组：有九个整形内容。

`year`：

`month`：

`day`：

`hours`：

`minutes`：

`seconds`：

`weekday`：

`Julia day`：

`flag(1[夏令时]或-1[根据当前日期自己判断]或 0[正常格式])`：

3 格式化字符串：

`%a` 本地（local）简化星期的名称

`%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 (0-31)

%H 一天中的第几个小时 (24 小时制。0-23)

%I 第几个小时 (12 小时制, 01-)

%J 一年中的第几天

%m 月份 (01-12)

%M 分钟数 (0-59)

%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, 第一个星期天之前所有的天数都放在第 0 周)

%w 一个星期中的第几天 (0-6, 0 是星期天)

%W 和 U 相同, 但是以星期一作为一个星期的开始。

%x 本地相应日期

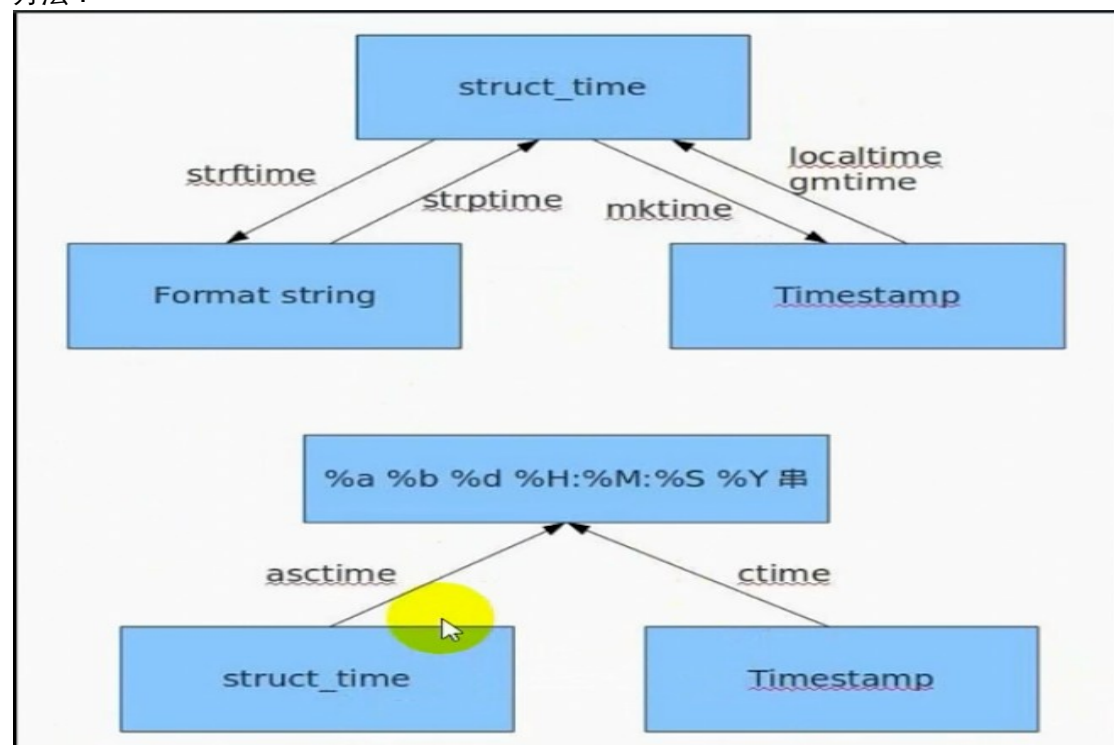
%X 本地相应时间

%y 去掉世纪的年份

%Y 完整的年份

%Z 时区的名字 (如果不存在为空字符)

方法：



1 返回当前时间的时间戳：

```
import time
a = time.time() # 返回当前时间的时间戳
print(a)
1541042051.4165714
```

2 time.localtime()将时间戳转化为元组形式得到的是格林尼治时间。

```
import time
a = time.time() # 返回当前时间的时间戳
print(a)
print(time.gmtime(a))
```

time.struct_time(tm_year=2018, tm_mon=11, tm_mday=1, tm_hour=3, tm_min=14, tm_sec=50, tm_wday=3, tm_yday=305, tm_isdst=0)

time.localtime()得到本地时间的元组

time.struct_time(tm_year=2018, tm_mon=11, tm_mday=1, tm_hour=11, tm_min=17, tm_sec=55, tm_wday=3, tm_yday=305, tm_isdst=0)

time.struct_time(tm_year=2018, tm_mon=11, tm_mday=1, tm_hour=11, tm_min=17, tm_sec=55, tm_wday=3, tm_yday=305, tm_isdst=0)

```
import time
a = time.time() # 返回当前时间的时间戳
print(time.localtime(a))
```

time.struct_time(tm_year=2018, tm_mon=11, tm_mday=1, tm_hour=11, tm_min=17, tm_sec=55, tm_wday=3, tm_yday=305, tm_isdst=0)

3 将本地时间转化为时间戳 time.mktime()

```
import time
a = time.time() # 返回当前时间的时间戳
c = time.localtime(a)
print(time.mktime(c))
```

1541042490.0

4 将时间元组转成字符串：time.asctime()

```
import time
a = time.time() # 返回当前时间的时间戳
c = time.localtime(a)
print(time.asctime(c))
```

Thu Nov 1 11:24:00 2018 (thu 表星期四，nov 十一月 1 表几号)

5 将时间戳转成字符串 time.ctime()相当于time.asctime(time.localtime(time.time()))

```
import time
a = time.time() # 返回当前时间的时间戳
c = time.localtime(a)
print(time.ctime(a))
```

Thu Nov 1 11:26:41 2018

6 自定义格式输出时间：time.strftime('%Y-%m-%d %H:%M:%S',b)将时间元组转成指定格式的字符串，参数2为时间元组，如果没有参数默认转换当前时间。后面的%H:%M:%S可以写成%X，格式一样。

```
import time
a = time.time() # 返回当前时间的时间戳
c = time.localtime(a)
# print(time.ctime(a))
q = time.strftime('%Y-%m-%d %H:%M:%S', c)
print(q)
```

2018-11-01 11:36:21

```
q = time.strftime('%Y-%m-%d %H:%M:%S')
print(q)
```

2018-11-01 11:31:45

- 7 将时间字符串转成时间元组 time.strptime.

```
w = time.strptime(q, "%Y-%m-%d %X")
print(w)
```

```
time.struct_time(tm_year=2017, tm_mon=7, tm_mday=28, tm_hour=15, tm_min=7,
tm_sec=15, tm_wday=4, tm_yday=209, tm_isdst=1)
```

- 8 time.clock()

此函数在 unix 系统下始终返回全部的运行时间

windows 系统从第二次开始到第一次调用此函数的开始时间戳为基数，中间

的

```
y = time.clock()
print(y)
time.sleep(1)
y1 = time.clock()
print(y1)
time.sleep(1)
y2 = time.clock()
print(y2)
```

2.279659510055578e-06

1.000225116376618

2.000057561402629

datetime

datetime 比 time 要高级，可以理解为 datetime 在 time 基础上进行了封装，提供了多种使用的函数，datetime 函数的接口（函数提供的方法）更加直观，更容易调用。

datetime 模块中提供的类

datetime 同时有时间和日期

timedelta 主要用于计算时间的跨度。

tzinfo 时区相关

time 只关注时间

date 只关注日期

datetime 的使用：

- 1 datetime.datetime.now() 获取当前时间

```
import datetime

t1 = datetime.datetime.now()

2018-11-02 08:13:14.756865
```

- 2 获取指定时间(最右边要有个随机六位数)

```
d2 = datetime.datetime(1999, 10, 1, 10, 28, 25, 123456)
print(d2)
1999-10-01 10:28:25.123456
```

- 3 将时间转为字符串。strftime()

```
d3 = d1.strftime("%Y-%m-%d %X")
print(d3)
print(type(d3))
2017-07-28 15:56:04
<class 'str'>
```

- 4 将格式化字符串转化为 datetime 类型 time.time.strptime(格式)
注意转换的格式要与字符串一致

```
d4 = datetime.datetime.strptime(d3, "%Y-%m-%d %X")
print(d4)
2017-07-28 15:57:23
```

- 5 可以直接对 datetime 时间类型进行加减，然后还可以用 d7.days/d7.seconds 查看时间间隔的天数和间隔天数除外的秒数。

```
d5 = datetime.datetime(1999, 10, 1, 10, 28, 25, 123456)
d6 = datetime.datetime.now()
d7 = d6 - d5
print(d7)
print(type(d7))
6510 days, 5:31:07.137009
<class 'datetime.timedelta'>
```

日历模块 (calendar)

1 calendar.month(年, 月)获取指定年的月历, 以日历的形式打印出来。

```
import calendar  
print(calendar.month(2018, 11))
```

```
Mo Tu We Th Fr Sa Su  
      1  2  3  4  
 5  6  7  8  9 10 11  
12 13 14 15 16 17 18  
19 20 21 22 23 24 25  
26 27 28 29 30
```

2 访问整年的日历 calendar.calendar(年)

3 判断是否是闰年 calendar.isleap(year)

4 返回某个月的 weekday 的第一天和这个月天数。
calendar.monthrange(year, month)

5 返回某个月每一周为元素的列表。calendar.monthcalendar(year, month)

模块与类

概述：当代码量比较少的时候，写在一个文件中还看不出来什么缺点，但是随着代码量的增加，代码就会越来越难以维护，所以为了解决这样的问题，我们把很多相似功能的函数分组，分别放到不同的文件中，这样每个文件所包含的内容相对较少，而且对于每一个文件的大致功能可用文件名来体现，很多编程语言都是这样来组织代码的，对于 python 来说，把相同功能的函数封装到一个.py 中去，一个.py 文件就是一个模块。

模块的优点：

- 1 提高代码的可维护性，提高了代码的复用度。
- 2 当一个模块完毕，可以被多个地方引用。
- 3 引用其他模块（内置模块、第三方模块、自定义模块。）
- 4 避免函数名和变量名的重复。

小技巧（将文件拖到 cmd 命令行可以直接显示文件的绝对路径）

SYS 模块

- 1 `sys.argv` (获取命令行参数的列表) 在 cmd 中去运行的时候, 等待用户输入参数。
划线处为我在 cmd 终端输入的参数。

```
import sys
print(sys.argv)
print('*****')
for i in sys.argv:
    print(i)
print('*****')
name = sys.argv[1]
age = sys.argv[2]
hoby = sys.argv[3]
print(name, age, hoby)
```

```
C:\Users\Administrator>python C:\Users\Administrator\PycharmProjects\千锋教程\caogao.py yangchao 18 goodman
['C:\Users\Administrator\PycharmProjects\千锋教程\caogao.py', 'yangchao', '18', 'goodman']
*****
C:\Users\Administrator\PycharmProjects\千锋教程\caogao.py
yangchao
18
goodman
*****
yangchao 18 goodman
```

- 2 `sys.path()` 自动查找所需模块路径的列表。

使用自定义模块

引入模块: `import module([module1],[module2],[module3]..[modulen])`

如: `import time, random, os` 都可以。

PS: 引入自定义模块, 不用加 .py, 一个模块只会被引入一次, 不管你执行了几次 `import`。防止重复引入。

使用模块中的内容:

方法一: 模块名.函数名/变量名。

方法二: `from module import name1[name2, [name3]..[namen]]` PS: 缺点, 如果程序内有引入的同名函数, 会将模块导入的函数覆盖掉。

方法三: `from...import...*` 语句。将一个模块中的所有内容导入进来。PS: 缺点, 如果程序内有引入的同名函数, 会将模块导入的函数覆盖掉。此种方式不推荐使用。

`__name__` 属性。

每一个模块都有一个 `__name__` 属性。当其值 == `'__main__'` 的时候, 表明该模块本身在执行。否则被引入了其他文件。

模块被另一个程序引用的时候, 我们不想让模块中的某些代码执行, 可以用 `__name__` 属性来使程序仅调用模块中的一部分。

包

“如果不同的人做的模块同名怎么办？”

解决：为了解决模块命名的冲突，引入了按目录来组织模块的方法，称之为‘包’

特点：

引入了包以后，只要顶层的包不与其它人发生冲突，那么模块都不会与别人发生冲突。

注意：目录只有包含一个叫做‘__init__.py’的文件，才会被认作为一个包！主要是为了避免一些滥竽充数的单词。但是（学到这里）这个文件中什么也不用写。

安装第三方模块

在 Mac 和 linux 系统下安装 python 的时候，pip 会自动安装。

在 windows 系统下，安装 python 时候需要勾选一个 pip（安装 pip 方法）和 add python....path（添加 python 到环境变量）

安装第三方模块语句（命令行模式下）pip install 模块名

windows 下如果安装报错的话，使用 pip install --upgrade pip 升级 pip

面向对象的编程

软件编程的实质：将我们的思维转变成计算机能够识别的一个过程。

对象的定义：

对象是内存中专门用来存储数据的一块区域。

对象由三部分组成：①对象的标识（ID），②对象的类型（Type），③对象的值（Value）

一：面向过程编程

- 1 自上而下按顺序执行，逐步求精
- 2 其程序结构是按照功能分为若干模块，这些模块形成一个树状结构
- 3 各个模块之间的关系尽可能简单，在功能上相对独立，
- 4 每一个模块内部均是由顺序，选择和循环三种基本结构组成。
- 5 其模块化实现的具体方法是使用子程序。
- 6 程序流程在写程序的时候就已经决定。

二：什么是面向对象

- 1 把数据及对数据的操作方法放到一起，作为一个相互依赖的整体—对象。
- 2 对同类对象抽象出其共性，形成类。
- 3 类中的大多数数据，只能用本类的方法进行处理。
类通过一个简单的外部接口与外界法伤关系，对像与对象之间通过消息进行通信。
- 4 程序流程由用户在使用中决定。

特点：将复杂的事情简单化，面向过程认识执行者，面向对象认识指挥者。所以后者更符合人的需求。

完成需求时：①先去找能完成所需功能的对象②如果没有这样的对象，创建之。

比较：面向过程强调的是功能行为，关注的是解决问题需要哪些步骤；面向对象是将功能

进行封装，强调具备了功能的对象，关注的是解决问题需啊哟哪些对象。（比如盖房子：面向过程就是说房子是怎样由一块砖一块砖码起来的，而面向对象就是看哪些人能盖房子。）

类

面向对象的三大特征：

- ① 封装：确保对象中数据的安全性。（如__属性私有）
- ② 继承：确保对象的可扩展性
- ③ 多态：保证了程序更加灵活

（一）设计类：

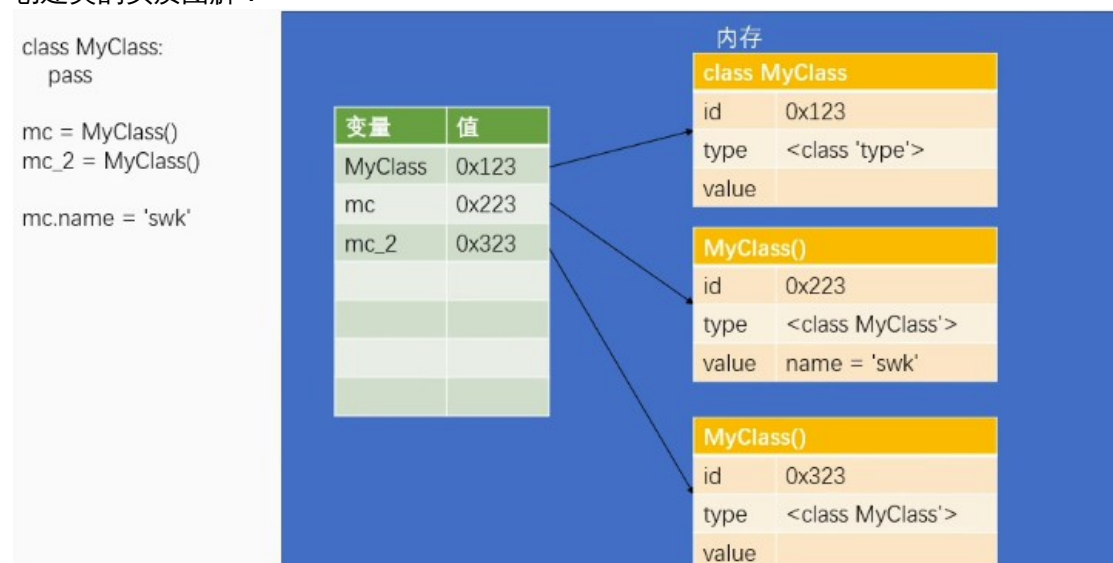
首字母大写，其他遵循驼峰原则（当变量名称需要由多个单词组成时，除第一个单词可大可小外，其他的单词都要大写！像驼峰一样起伏，提高代码可读性）见名知意

属性：见名知意，其他遵循驼峰原则

行为(方法/功能)：见名知意其他遵循驼峰原则

（二）创建类：

创建类的实质图解：



类：类型。本身不占内存空间，与学过的 number、string 等类似。用类创建实例化对象，对象才占用内存空间。

格式：

class 类名（父类列表）：如果父类列表为 object（基类，所有类的父类。一般来说没有合适父类就写 object）

属性：

行为：

关于类的属性有两种：类属性和成员属性

class

```
Dog(object):
    url = 'abc'
    def wangwang(self):
```

```

print('    汪    汪    汪    ')

print(url)
Traceback (most recent call last):
汪汪汪
File "C:/Users/asus/.PyCharmCE2018.3/config/scratches/test4.py", line 10, in
<module>
dog.wangwang()
File "C:/Users/asus/.PyCharmCE2018.3/config/scratches/test4.py", line 7, in
wangwang
print(url)
NameError: name 'url' is not defined

```

分析：此类 Dog 中 url 属性即为类属性，后面的 print (url) 会报错，因为类的实例不能直接访问类属性。可以换成 print(self.url)或者(Dog.url)就可以访问到 url 属性了。

注意：定义类方法的时候，方法的参数必须以 self 作为第一个参数。self 代表类的实例（某个对象）

（三）实例化对象

格式：对象名 = 类名（参数列表）

注意

① 有参数的情况下，小括号不能省略。

② 方法的调用和函数方法调用的不同：函数调用的时候有几个形参就传几个实参，但是类方法调用的时候，第一个参数会由解析器自动传递，就是调用该方法的实例本身。于是类中定义方法至少需要定义一个参数（一般写作 self），但是用类去调用类方法时，不会自动传入参数，需要手动添加。

```

class Animal(object):
    def run(self):
        print('I can run...')

a = Animal()
a.run()
Animal.run()

I can run...
Traceback (most recent call last):
  File "D:/MyCode/Cuiqingcai/test2.py", line 53, in <module>
    Animal.run()
TypeError: run() missing 1 required positional argument: 'self'

```

关于类与实例的属性与方法的关系

③ 方法的查找流程：当调用一个对象的属性时，解析器会先在当前的对象中寻找是否含有该属性，如果有，则返回当前对象属性值，如果没有，则去当前对象的类对象中寻找，如果有则类对象的属性，如果没有就会报错。

④ 在类中定义的方法，不可以直接调用类属性；

·类属性可以通过类或类的实例访问到，但是类属性只能通过类对象来修改，无法通过实例对象修改。；

·实例属性只能通过实例对象来修改，类对象不能修改实例属性。

⑤ @property 的用法，将一个类的方法转换为类的属性，添加@property 装饰器之后，我们就可以像调用类的属性一样调用类的 get 方法了（不用加括号）

⑥ @属性名.setter 使用设置属性的方法对类方法进行设置

⑦ 类中 **@classmethod** 定义的方法叫作类方法，第一个参数为 (cls) 代表该类，类方法可以用类调用，也可以用该类的实例调用，没有区别。

⑧ 类方法前加 **@staticmethod** 表示一个静态方法，静态方法不需要传递任何参数，既可以通过类调用，也可以通过实例调用。静态方法基本上是和类无关的方法，它只是一个保留在类中的函数，一般是一些工具方法。

类的构造函数：

__init__() 在使用创建对象的时候自动调用的。如果构造的对象含有参数，那么参数肯定会被 **__init__()** 调用。如果不显示的写出构造函数，默认会自动添加一个空的函数。

类中 self 的意义：

self 代表类的实例，而非类。

哪个对象调用方法，那么该方法中的 self 就代表那个对象。

self.__class__ 代表类名。

类的析构函数：

__del__() 释放对象时自动调用。（python 中的对象用完之后也可以不手动释放，待程序结束计算机会自动清理内存的，所以析构函数一般用得不多。）

对象释放的方法：del + 对象名

在函数里面定义的对象，会随函数结束时自动释放。可以用来减少内存空间的浪费。

对方一旦释放之后就不能再访问了。

重写 **__repr__** 与 **__str__** 函数。

重写：将函数重新定义写一遍。

__str__()：在调用 print 打印对象时候自动调用，是给用户用的，是一个描述对象的方法。

__repr__()：是给机器用的，在 python 解释器里面直接敲对象名回车之后就可以调用的方法。

注意：在没有 **__str__** 且存在 **__repr__** 时，**__str__ = __repr__**。

用法：当一个对象的属性方法很多的时候，并且都需要打印出来的时候，重写 **__str__** 方法后，可以简化代码。

封装

访问限制：

要让类的内部属性不被外部直接访问。为了数据安全，可以直接在属性的前面加 **__** 即可。属性就变成了私有属性。只内部可以自己调用。可以通过内部的方法实现内部属性的修改
PS：其实 python 内部的私有属性通过解释器变成了 **_Person__money**，所以通过调用 **_Person__money** 还是可以访问私有属性，但是强烈不要这么做，而且不同编辑器变化方式可能有不同。

类的继承

两个类，A类和B类，当A类继承自B类时，A类自定拥有了B类的所有属性和方法。
object是所有类的父类。还可以称为基类或超类。

优点：

- 1 简化了代码，减少冗余。
- 2 提高了代码的健壮性。（在父类中修改方法或属性，所有子类同时都修改了。不需要一个一个去改。）
- 3 提高了代码的安全性。（父类不对外暴露，难以修改。）
- 4 是多态的前提。

缺点：

- 1 耦合与内聚是描述类与类之间的关系的，耦合性越底，内聚性越高，代码越好。但是继承的耦合性挺高的。

（耦合：指两者之间的联系。比如父类这子类的联系就很高，父类变而子类变）

（内聚：内在功能的一个聚合程度。与耦合相对。

注意：如果子类中有和父类同名的方法，则通过子类实例去调用方法时，会调用子类而不是父类的方法，这个叫作类的方法的重写。

单继承的实现：

子类希望可以直接调用父类的__init__来初始化父类中定义的属性时，super()可以用来获取当前类的父类，并且通过super()返回对象调用父类方法时，不需要传self参数。

```
def __init__(self, name, age, money, stuID)
```

调用父类（Father）中的__init__

super().__init__(name, age, money) 等价于 Fathre.__init__(self, name, age, money),推荐用super，可以避免耦合。

多继承的实现

注意括号中调用多个父类的写法。注意：多继承中如果父类中有相同的方法，那么子类在调用该方法时候，默认使用排名考前的父类方法。（没有特殊情况尽量别用）

```
class Father():
    def __init__(self, money):
        self.money = money
    def play(self):
        print('I can play!')
    def func(self):
        print("Father'sfunc")
class Mother():
    def __init__(self, faceValue):
        self.faceValue = faceValue
    def eat(self):
        print('eat')
    def func(self):
        print("Mother's func")
```

```

from father import Father
from mother import Mother

class Child(Father, Mother):
    def __init__(self, money, faceValue):
        Father.__init__(self, money)
        Mother.__init__(self, faceValue)

```

用 `__bases__` 这个属性可以获取当前类的所有父类。

类的多态

概念：一种事物的多种形态。

理解：鸭子类型：如果一个东西，走路像鸭子，叫声像鸭子，那么就认为它是一只鸭子

例子：

`len()`函数式求一个对象的长度，该对象可以是 `list`，`str`，`tuple`，`dict`...

这是因为 `list`，`str`，`tuple`，`dict`...等对象中都有一个叫做 `__len__` 的特殊方法，即只要一个对象有此方法就可以用 `len()` 获取该对象的长度。

对象属性与类属性（类属性由类名来调用，对象属性由创建的对象调用）

对象属性的优先级高于类属性

```

class Person():
    name = 'Person' # 此为类属性
    def __init__(self, name):
        self.name = name # 此为对象属性

print(Person.name)
per = Person('tom')
print(per.name)

```

Person

tom

对比：没有对象属性时默认用类属性

```

class Person():
    name = 'Person' # 此为类属性
    # def __init__(self, name):
    #     self.name = name # 此为对象属性

print(Person.name)
per = Person()
print(per.name)

```

Person

Person

可以动态地给对象添加属性，但是此属性只针对该一个对象有效，对类创建的其他对象无

效。

当用 def 语句删除对象中的某个属性的时候，再调用会使用到同名的类属性。所以以后要注意，对象属性和类属性不要重名，因为对象属性会屏蔽类属性。但是删除对象属性后又可以使用类属性，难以掌控。

动态给实例添加属性和方法

```
class Person():
    pass
#添加属性
per = Person()
# 动态添加属性，体现了动态语言的特点（灵活）
per.name = 'Frank'
print(per.name)
# 动态添加方法。
def say(self):
    print('my name is' + self.name)
per.speak = MethodType(say, per)
per.speak()

Frank
my name isFrank
```

限制对象随意添加属性的方法：

解决：定义类的时候，定义一个特殊的属性（__slots__），可以限制动态添加的属性 slots 后面以元组的形式固定对象的属性。

```
class Person():
    __slots__ = ('age', 'name')

per = Person()
per.weight = 170
print(per.weight)
```

@property（可以让你对受限制访问的属性用.方法获取）要访问类的私有属性 __age 方法除了增加 GetAge() 和 SetAge（）之外可以用以下方法。这样可以直接用对象.属性名获取

```
#方法名为受限制的变量去掉双下划线
@property
def age(self):
    return self.__age
@age.setter #去掉下划线.setter
def age(self, age):
    if age < 0:
        age = 0
    self.__age = age
```



```
per.age = 100 #相当于调用setAge
print(per.age) #相当于调用getAge
```

打印结果 100 理解：此处的 per.age 不是调用 age 属性，而是调用的上面@property 下面的 age 方法。这样实例化对象获取限制访问属性更加方便。

运算符重载

```
class Person():
    def __init__(self, num):
        self.num = num
    def __add__(self, other):
        return Person(self.num + other.num) # 实例中的self.num
    def __str__(self):
        return 'num = ' + str(self.num)

per1 = Person(1)
per2 = Person(2)
print(per1 + per2) # per1 + per2 == per1.__add__(per2)
print(per1)
```

划红线处解释：实例中的 $self.num + other.num = 3$ 所以 $== return Person(3)$ ，所以 $str(self.num = 3)$

垃圾回收

在程序中没有被引用的对象就是垃圾，垃圾将影响程序的运行性能，所以这些垃圾必须被清理（即删除），python 中有自动的垃圾回收机制。会自动将没有被引用的对象删除，可以不用手动处理垃圾，程序结束之后对象也会被删除。

`__del__()` 是一个特殊方法，它会在对象被垃圾回收前调用

例如：

```
class A(object):
    def __init__(self):
        self.name = 'A类'

    def __del__(self):
        print('A对象被删除了...', self)

a = A()
input('回车键退出...')
```

此程序运行之后不按回车键程序未退出，还有 a 变量引用了 A 所有 `__del__()` 方法不会被执行，但是当设置 `a=None` 后，a 指向了 None，对象 A 没有被引用，所以被删除，删除之前执行了 `__del__()` 方法。同样以上代码当按回车之后，因为程序已经结束了，所以 A 类将会被回收，`__del__()` 方法将被执行。

特殊方法（魔术方法）

特殊方法不需要手动调用，在特殊情况下自动执行。

`__str__()`

`__str__()`方法在尝试将对象转换为字符串的时候调用。

作用：指定对象转换为字符串的结果。

当我们打印一个类对象时，实际上打印的就是`__str__()`方法的返回值。

```
class Person(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age

a = Person('Tom', 18)
b = Person('Bob', 24)
print(a)
<__main__.Person object at 0x000000002987390>

class Person(object):
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return 'Hello,World'

a = Person('Tom', 18)
b = Person('Bob', 24)
print(a)
Hello,World
```

`__repr__()`

`__repr__()`这个特殊方法在对象使用`repr()`函数时调用。

作用：指定对象在交互模式中直接输出的效果。

例如：在命令行模式下

```
>>> a = 'hellow,word'
>>> print(a)
hellow,word
>>> a
'hellow,word'
>>>
```

`print(a)`打印的是`__str__()`函数执行的结果而直接写 `a` 执行的就是`__repr__()`函数的结果。

运算方法：

```
# object.__add__(self, other)
# object.__sub__(self, other)
# object.__mul__(self, other)
# object.__matmul__(self, other)
# object.__truediv__(self, other)
# object.__floordiv__(self, other)
# object.__mod__(self, other)
# object.__divmod__(self, other)
# object.__pow__(self, other[, modulo])
# object.__lshift__(self, other)
# object.__rshift__(self, other)
# object.__and__(self, other)
# object.__xor__(self, other)
# object.__or__(self, other)
```

其他

```
# object.__lt__(self, other) 小于
# object.__le__(self, other) 小于等于
# object.__eq__(self, other) 等于
# object.__ne__(self, other) 不等于
# object.__gt__(self, other) 大于
# object.__ge__(self, other) 大于等于

# __gt__会在对象做大于比较的时候调用，该方法的返回值将会作为比较的结果
# 他需要两个参数，一个self表示当前对象，other表示和当前对象比较的对象
# self > other
def __gt__(self, other):
    return self.age > other.age
```

`__bool__(self)` 通过此函数来指定对象转换为布尔值的情况

`__len__()` 返回对象的长度

正则表达式

简介：

正则表达式就是用来简洁表达一组字符串的表达式：

'PN'
'PYN'
'PYTN'
'PYTHN'
'PYTHON'



正则表达式：
`P(Y|YT|YTH|YTHO)?N`

正则的优势：简洁。可以非常简答地表达一组很大字符串的数据特征。‘一行胜前言’
比如表达以 PY 开头但是中间不含由 PY 的不多于 10 个字符的字符串

如：‘PYABDJ’√

‘PYSADFYDF’×

如果要枚举所有符合条件的字符串就会很复杂，但是如果用正则表达式的话就一行：

‘PY[[^]PY]{0,10}’

用处广泛：

- 通用的字符串表达框架
- 简洁表达一组字符串的表达式
- 针对字符串表达‘简洁’和‘特征’思想的工具
- 判断某字符串的特征归属

在文本处理中作用：

- 表达文本类型的特征（病毒、入侵等）
- 同事查找或者替换一组字符串
- 匹配字符串的全部或部分内容

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

在一个字符串的开始位置起匹配正则表达式，**返回 match 对象**。

参数：

`pattern`：匹配的正则表达式

`str` 要匹配的位置

`flag`：标志位，用于控制正则表达式的匹配方式。值如下：

`re.I` 忽略大小写

`re.L` 做本地识别

`re.M` 多行匹配，影响 `^` 和 `$`

`re.S` ‘.’匹配包括换行符在内的任意字符（一般情况下，是不匹配换行符的。）

`re.U` 根据 unicode 字符集解析字符，影响 `\w` `\W` `\b` `\B`

`re.X` 使更灵活的格式理解正则表达式

功能：从字符串**起始位置匹配**一个模式，如果不是起始位置，匹配成功或者没有匹配的话返回 `None`。

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

在一个字符串中搜索匹配正则表达式的第一个位置，**返回 match 对象**。

参数：`pattern`：匹配的正则表达式

`str`：要匹配的位置

`flag`：标志位，用于控制正则表达式的匹配方式。

功能：扫描整个字符串，并返回**第一个匹配**的功能。

`re.findall (pattern, string, flag=0)`

搜索字符串，以**列表的形式返回全部**匹配的字符串。

参数：`pattern`：匹配的正则表达式。

`str`：要匹配的位置。

`flag`：标志位，用于控制正则表达式的匹配方式。

功能：扫描整个字符串，并返回所有可以匹配的结果列表。

`re.split(pattern, string, maxsplit, flags=0)`

注意：与其他函数相比，中间多的 `maxsplit` 参数位最大分割次数。

将一个字符串按照正则表达式匹配的结果进行分割，以**列表类型返回**。

详细看下面的（`re` 深入）

re.finditer()

搜索字符串，返回一个匹配结果的迭代类型，每个迭代元素都是 match 对象。
详细看下面的 (re 深入)

re.sub()

在一个字符串中替换所有匹配正则的子串，返回替换后的字符串。
详细看下面的 (re 深入)

正则表达式元字符

- . 匹配除换行符以外的任意字符。
- . 前一个字符 0 次或者无限次扩展

[] 字符集合。匹配[]中所有包含的任意一个字符。
[^] 非字符集，给单个字符给出排除范围 [^ abc] 表示非 a 或 b 或 c 的单个字符。
{ m } 扩展前一个字符 m 次
{ m, n } 扩展前一个字符 m 次到 n 次 (含 n) 如 : ab{ 1, 2 } c 表示 abc、abbc
+ 前一个字符 1 次或无限次扩展
? 前一个字符 0 次或 1 次扩展 abc? 表示 ab 或者 abc , P (Y|YT|YTH|YTHO) ?N 表示 'PN', 'PYN', 'PYTN', 'PYTHN', 'PYTHON'
| 或左右表达式任意一个
[a-z] 匹配任意小写字母
[A-Z] 匹配任意大写字母
[0-9] 匹配任意数字，类似 [0123456789]
[0-9a-zA-Z] 匹配任意数字和字母
[0-9a-zA-Z _] 匹配任意数字和字母下划线
[^ frank] 匹配除了 frank 这几个字符串的所有字符。^ 叫做脱字符，表示不匹配集合中的字符。
[^ 0-9] 匹配所有非数字字符。
\d 匹配数字，效果同 [0-9]
\D 匹配非数字字符，效果同 [^ 0-9]
\w == [0-9a-zA-Z _] 匹配任意数字和字母下划线
\W 匹配非数字，字母，下划线 == [^ 0-9a-zA-Z _]
\s 匹配任意空白符 (空格，换行 (\n)，回车 (\r\n) 换页，制表符 (\r)) 效果同 [\f\n\r\t]
\S 匹配任意非空白符 == [^ \f\n\r\t]
特殊的 : [\u4e00-\u9fa5] 匹配一个中文字符。

锚字符

^ 行首匹配注意与 [] 里面的 ^ 号不一样。如果加上 re.M 就可以匹配多行的行首
\$ 行尾匹配。abc\$ 表示以 abc 结尾，如果加上 re.M 就可以匹配多行的行尾
^ 最前匹配，只匹配整个字符串的开头，即使在 re.M 模式下也不匹配其它行的行首。
^ 匹配字符串结束，只匹配整个结尾。即使在 re.M 下也不匹配其它的行尾。

\b 匹配一个单词的边界。也就是单词和空格间的位置。

\B 匹配非单词边界。

匹配多个字符

re 库的默认匹配是贪婪的。

```
>>> match = re.search(r'PY.*N', 'PYANBNCNDN')
>>> match.group(0)
'PYANBNCNDN'
```

最小匹配操作：

操作符	说明
*?	前一个字符0次或无限次扩展，最小匹配
+?	前一个字符1次或无限次扩展，最小匹配
??	前一个字符0次或1次扩展，最小匹配
{m,n}?	扩展前一个字符m至n次（含n），最小匹配

(xyz) 匹配小括号内的 xyz(作为一个整体去匹配)

x? 匹配 0 个或者一个 x（非贪婪匹配）

x* 匹配 0 个后者多个 x（贪婪匹配，尽可能多匹配。） .* 表示匹配 0 个或者任意多个字符（除换行符）

x+ 匹配至少一个 x（贪婪匹配尽可能多匹配）

x{n} 匹配确定的 n 个 x（为一个非负整数）（贪婪匹配，尽可能多匹配）

x{n,} 匹配至少 n 个 x

x{n,m} 匹配至少 n 个，最多 m 个 x

x|y 匹配 x 或 y

特殊的匹配方式

*? +? x?:表示最小匹配。通常都是尽可能多的模式，可以使用这种方式解决贪婪匹配。

```
import re
a = 'sunk is a good man sunk is a nice man sunk is a stupid man'
re = re.findall(r'sunk.*?man', a)
print(re)
['sunk is a good man', 'sunk is a nice man', 'sunk is a stupid man']
```

?:x 类似 xyz 但不表示一个组。

re 模块深入

用来做字符串的切割：下面的+表示匹配多个空白符号

```
str1 = "sunk is a good man"
print(str1.split(" "))
print(re.split(r"+", str1))
```

```
['sunck', 'is', 'a', 'good', 'man']
```

`re.finditer (pattern, string, flag=0)`

参数：pattern：匹配的正则表达式

str：要匹配的位置

flag：标志位，用于控制正则表达式的匹配方式。

功能：与 `findall` 类似，扫描整个字符串，返回的是一个迭代器

字符串的替换与修改

`re.sub (pattern, repl, string, count=0, flag=0)`

`re.subn (pattern, repl, string, count=0, flag=0)`

其中 `subn` 返回的时候返回被替换的次数。

参数：pattern 正则表达式

repl 指定的用来替换的字符串

string 目标字符串

count 最多替换次数

flag 标志位

功能：在目标字符串中以正则表达式规格找到目标，然后以 `repl` 参数替换之。可以指定替换的次。如果不指定会替换所有的。

```
s = 'sunk is good man, sunk is nice man, sunk is stupid man'
result = re.sub(r'stupid', 'handsome', s)
print(result)
```

sunk is good man, sunk is nice man, sunk is handsome man

Sub 中 Repl 参数也可以用一个函数表示：

```
import re
def fn(sg):
    ret = int(sg.group()) - 10
    return str(ret)

string = '我 喜 欢 180 身 高 的 女 生 '
pattern = re.compile(r'\d+')
ret = pattern.sub(fn, string)
print(ret)
我喜欢 170 身高的女生
```

分组

除了简单判别是否匹配之外，正则表达式还有提取子串的功能。用 `()` 就是表示提取分组。

```
s = '010-8625825222aaa465'
m = re.match(r'((\d{3})-(\d{8}))', s)
print(m.group(0))# group() 使用序号获取对应组的信息，group(0) 一直代表匹配到的字符串
print(m.group(1))# 打印最外面的组
print(m.group(2))# 打印最外面之后的里面第一个组
print(m.group(3))# 打印最外面之后的里面第二个组
print(m.groups())# groups() 查看匹配到的各组的情况（一个括号括起来的称一个组）
```



```
010-86258252
010-86258252
010
86258252
('010-86258252', '010', '86258252')
```

在括号里面的前面加上?P<>在<>中间给组起名字，然后通过名字访问组的内容(注意 P 大写)

```
s = '010-8625825222aaa465'
m = re.match(r'((?P<first>\d{3})-(?P<second>\d{8}))', s)
print(m.group('first'))
print(m.group('second'))
010
86258252
```

编译

当使用正则时 re 会干两件事：

- 1 编译正则表达式如果正则编译不合法，会报错。
- 2 用编译后的正则去匹配对象。

compile (pattern, flag)

```
s = 'sunk is good man, sunk is nice man, sunk is stupid man'
r = re.compile(r'sunk')
print(r.findall(s))
['sunk', 'sunk', 'sunk']
```

爬虫

url：统一资源定位符组成详情。



参数：是我们的浏览器给服务器传的信息。锚：当前显示的网页位置

简单地打开一个网页获取其信息：

```
import urllib.request
response = urllib.request.urlopen('http://www.baidu.com')
data = response.read().decode('utf-8')
print(data)
```

一般不用 `read()` 全部读取，而是用 `readline()` 读取一行，或者用 `readlines()` 读取全部，但 `readlines` 与 `read()` 不同的是 `readlines` 读取出来的是一个列表，没一个元素就是读取出来的一行。

`response` 还有一些常用属性：

`response.info()` 返回当前环境的有关信息。

```
import urllib.request
response = urllib.request.urlopen('http://www.baidu.com')
print(response.info())
```

`response.getcode()`: 可以返回状态码

```
import urllib.request
response = urllib.request.urlopen('http://www.baidu.com')
print(response.getcode())
200
```

一般后面加语句：

```
if response.getcode() == 200 or response.getcode() == 304:
```

如果得到 200 或者 304 代表访问正常，有数据返回。

状态码对应表：

100	客户必须继续发出请求
101	客户要求服务器根据请求转换HTTP协议版本
200	交易成功
201	提示知道新文件的URL
202	接受和处理, 但处理未完成
203	返回信息不确定或不完整
204	请求收到, 但返回信息为空
205	服务器完成了请求, 用户代理必须复位当前已经浏览过的文件
206	服务器已经完成了部分用户的GET请求
300	请求的资源可在多处得到
301	删除请求数据
302	在其他地址发现了请求数据
303	建议客户访问其他URL或访问方式
304	客户端已经执行了GET, 但文件未变化
305	请求的资源必须从服务器指定的地址得到
306	前一版本HTTP中使用的代码, 现行版本中不再使用
307	申明请求的资源临时性删除
400	错误请求, 如语法错误
401	请求授权失败
402	保留有效ChargeTo头响应
403	请求不允许
404	没有发现文件、查询或URL
405	用户在Request-Line字段定义的方法不允许
406	根据用户发送的Accept头, 请求资源不可访问
407	类似401, 用户必须首先在代理服务器上得到授权
408	客户端没有在用户指定的时间内完成请求
409	对当前资源状态, 请求不能完成
410	服务器上不再有此资源且无进一步的参考地址
411	服务器拒绝用户定义的Content-Length属性请求
412	一个或多个请求头字段在当前请求中错误
413	请求的资源大于服务器允许的大小
414	请求的资源URL长于服务器允许的长度
415	请求资源不支持请求项目格式
416	请求中包含Range请求头字段, 在当前请求资源范围内没有range指示值, 请求也不包含If-Range请求头字段
417	服务器不满足请求Expect头字段指定的期望值, 如果是代理服务器, 可能是下一级服务器不能满足请求
500	服务器产生内部错误
501	服务器不支持请求的函数
502	服务器暂时不可用, 有时是为了防止发生系统过载
503	服务器过载或暂停维修
504	关口过载, 服务器使用另一个关口或服务来响应用户, 等待时间设定值较长
505	服务器不支持或拒绝请求头中指定的HTTP版本

访问当前正在爬取的 url 地址：response.geturl()

```
print(response.geturl())
```

<http://www.baidu.com>

urllib.reuquest.urlretrieve(url, filename)

将获取到的 url 信息放到 filename 这个文件中去, 如果没有, 创建之。

此方法的缺点, 此操作执行过程中会产生缓冲, 如果没结束缓存一直在, 浪费内存, 可以

用 `urllib.request.urlcleanup()` 清空缓存。

模拟浏览器(对付反爬虫方法)

模拟请求头：

```
headers = {  
    "User-Agent": "Mozilla/5.0 (Windows NT 10.0; WOW64)  
    AppleWebKit/537.36 (KHTML, like Gecko) Chrome/59.0.3071.115  
    Safari/537.36"  
}
```

设置一个请求体：

```
#设置一个请求体  
req = urllib.request.Request(url, headers=headers)
```

发起请求：

```
response = urllib.request.urlopen(req)
```

可以设置一个 `agentlist` 保存一些请求头集合。(可以网上百度找) 然后调用 `random.choice()` 随机选择请求头访问，防止服务器封 ip。那么一次完整请求变为：

```
agentStr = random.choice(agentList)  
req = urllib.request.Request(url)  
#向请求体里添加了User-Agent  
req.add_header("User-Agent", agentStr)  
response = urllib.request.urlopen(req)  
print(response.read().decode("utf-8"))
```

设置超时

在 `urllib.request.urlopen(url, timeout=)` 在 `timeout` 处赋值，可以使超时某个时间之后停止运行。

```
for i in range(1, 100):  
    try:  
        response = urllib.request.urlopen(  
            "http://www.baidu.com", timeout=0.5)  
        print(len(response.read().decode("utf-8")))  
    except:  
        print("请求超时，继续下一个爬取")
```

http 请求：

进行客户端与服务端之间的消息传递时使用。

1 **get**：通过 URL 网址传递信息。直接在 url 上添加要传递的信息。(浏览器给服务器的)

- 2 post：可以向服务器传递数据，是一种比较流行的，比较安全的请求方式。也可以修改服务器的数据。
- 3 put：请求服务器存储一个资源，通常要指定存储的位置。
- 4 delete：请求服务器删除一个资源
- 5 head：请求获取对应的 http 包头信息。
- 6 option：可以获取当前 url 所支持的请求类型。

get：

把数据拼接到请求路径的后面传递给服务器。

优点：速度快

缺点：承载的数据量小，不安全。

关于 json 文件知识

概念：json 是一种保存数据的格式。

作用：可以保存本地的 json 文件，也可以将 json 串进行传输，通常将 json 称为轻量级的传输方式（与 XML 相比没有哪些<...>的标签。但是可读性还是不如 XML）

json 文件的组成：

{ } 代表对象(字典)

[] 代表列表

: 代表键值对

, 分隔两个部分

将 json 格式的字符串转为 python 数据类型的对象

```
import json
jsonStr = '{"name": "Frank", "age": 18, "hobby": ["money", "power", "English"], "params": {"a": 1, "b": 2}}'
jsondata = json.loads(jsonStr)
print(jsondata)
print(type(jsondata))
print(jsondata['hobby'])
```

```
{'name': 'Frank', 'age': 18, 'hobby': ['money', 'power', 'English'], 'params': {'a': 1, 'b': 2}}
<class 'dict'>
['money', 'power', 'English']
```

反过来，将 python 字典转化成为 json 字符串形式：用 json.dumps()

```
jsonData2 = {"name": "sunck", "age": 18,
             "hobby": ["money", "power", "english"], "params": {
                 "a": 1, "b": 2}}
jsonStr2 = json.dumps(jsonData2)
print(jsonStr2)
print(type(jsonStr2))
```

```
{"name": "sunck", "age": 18, "hobby": ["money", "power", "english"],
 "params": {"a": 1, "b": 2}}
<class 'str'>
```

用 json 读取本地文件，用 json.load()（注意是 load，不是 loads）智能转化为字典类型。

```

path1 = r"C:\Users\xlg\Desktop\Python-1704\day18
\Jsos\caidanJsos.json"
with open(path1, "rb") as f:
    data = json.load(f)
    print(data)
    #字典类型
    print(type(data))

```

将 json 写入本地文件：用 json.dump()

```

path2 = r"C:\Users\xlg\Desktop\Python-1704\day18
\Jsos\test.json"
jsonData3 = {"name": "sunck", "age": 18,
    "hobby": ["money", "power", "english"], "parames": {
    "a": 1, "b": 2}}
with open(path2, "w") as f:
    json.dump(jsonData3, f)

```

POST 请求

特点：将参数进行打包，单独传输，难以截获，承载量大，一般来说比较安全。（当要对服务器进行修改时，建议使用 post）

缺点：速度比较慢

要用 post 的时候，首先需要分析网页。

```

import urllib.request
import urllib.parse
url = 'http://www.baidu.com'
# 将要发送的数据合成一个字典，字典的键去网址找，一般为input标签的name属性的值
data = {'Username': 'sunck', 'passwd': '666'}
# 对数据进行打包：（注意要编码）
postData = urllib.parse.urlencode(data).encode('utf-8')
# 创建请求体
req = urllib.request.Request(url, data=postData)
req.add_header('User-Agent', (加入请求头))
# 发起请求
response = urllib.request.urlopen(req)
print(response.read().decode('utf-8'))

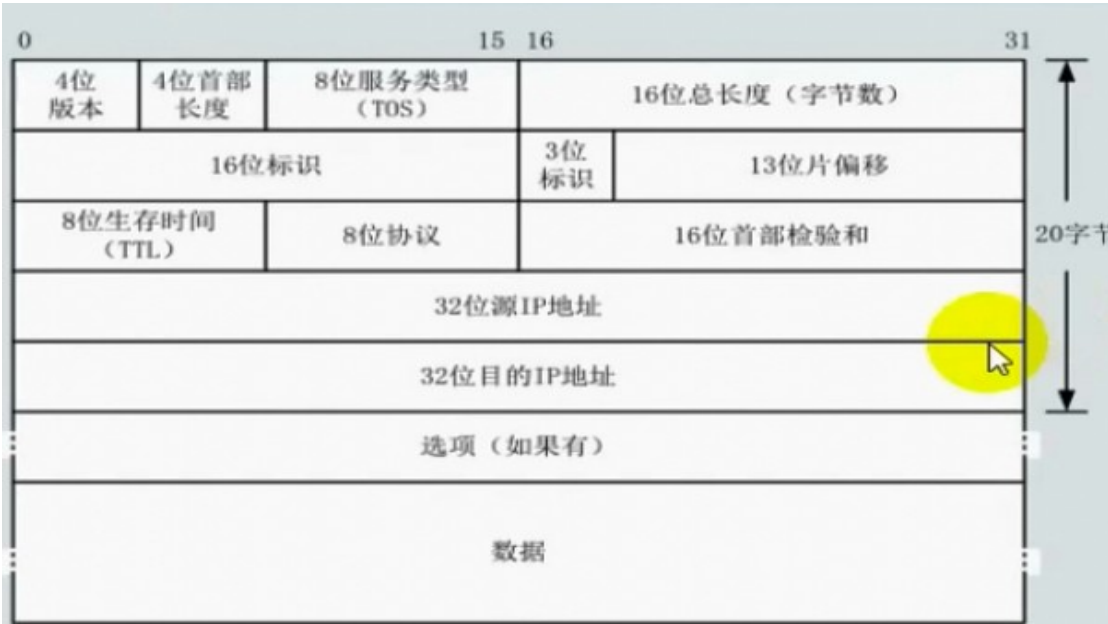
```

网络编程中的计算机知识

用 python 进行网络编程实际上就是 python 程序本身这个进程内，链接别的服务器进程的通信端口进行通信。

ip4 包头结构：

一个包头有 20 个字节。

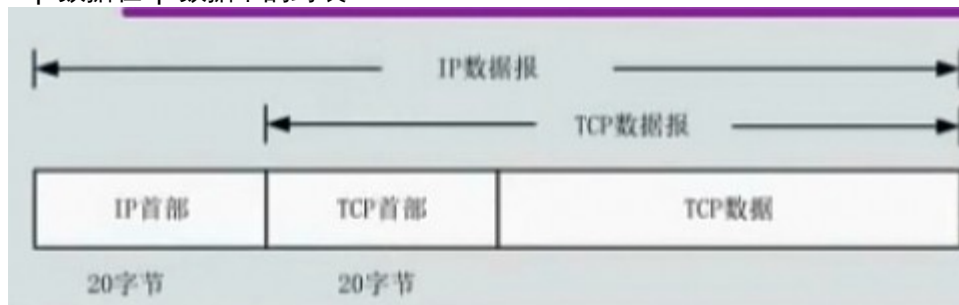


- 1 版本：---IP 头中的前四位标识了 IP 的操作版本，比如版本 4 或者版本 6
- 2 首部长度即 internet 头长度--- 头中下面 4 位包括头长度，以 32 位为单位表示。
- 3 标识 (identifier)：每个 ip 报文被赋予一个唯一的 16 位标识，用于标识数据报的分段

- 4 分段标识：下一个域包括 3 个 1 位标志，标识报文是否允许被分段和是否使用了这些域。
- 5 分段偏移：8 位的域指出分段报文相对于整个报文开始处的偏移，整个值以 64 位为单位递增。
- 6 生存时间：ip 报文不允许在广域的网络中永久漫游。它必须限制在一定的生存时间内。
- 7 协议：8 位域指示 ip 头之后的协议，如 VINES/TCP/UDP 等。
- 8 校验和：校验和是 16 位的错误检验域。目的机、网络中的每个网关要重新计算报文头的校验和，就如同源机器所做的一样。
- 9 源 ip 地址：谁发过来的，ip 地址
- 10 目的 ip 地址：接受者的 ip 地址。
- 11 填充，为了保证 ip 头的长度是 32 位的整数倍，额外的需要用 0 填充。

TCP 包头结构：

tcp 数据在 ip 数据中的封装：



TCP 包头首部结构：



- 1 16 位的源端口域包含初始化通信的端口号。源端口的源 IP 地址的作用是标识报文的返回地址。
- 2 TCP 序列号：32 位的序列号由接收端计算机使用，重组分段的报文成最初形式。
- 3 TCP 应答号:使用 32 位的应答（ACK）域标识下一个希望收到的报文的第一个字节。
- 4 数据偏移：整个 4 位域包括 TCP 头大小以 32 位数据结构或称为“双字”单位。
- 5 保留：6 位恒置 0 的域，为将来丁一新的用途保留。
- 6 标志：6 位标志域，没 1 位标志可以打开一个控制功能：这六个标志分别是：紧急标

- 志、有意义的应答标志、推、重置连接标志、同步序列号标志、完成发送数据标志
- 7 窗口大小：目的机使用 16 位的域告诉源主机，它想收到的每个 TCP 数据段的大小。
 - 8 校验和：TCP 也包括 16 位的错误检查域
 - 9 紧急：紧急指针域是一个可选的 16 位指针，指向段内的最后一个字节位置，这个域只在 URG 标志设置了时才有效。
 - 10 选项：至少一个字节的可变长域标识哪个选项。
 - 11 数据：域的大小是最大的 MSS，MSS 可以在源和谜底机器之间协商。
 - 12 填充：目的是确保空间的可预测性，定时和规范大小。这个域中加入额外的零可以保证 TCP 头是 32 位的整数倍。

TCP/IP 简介：

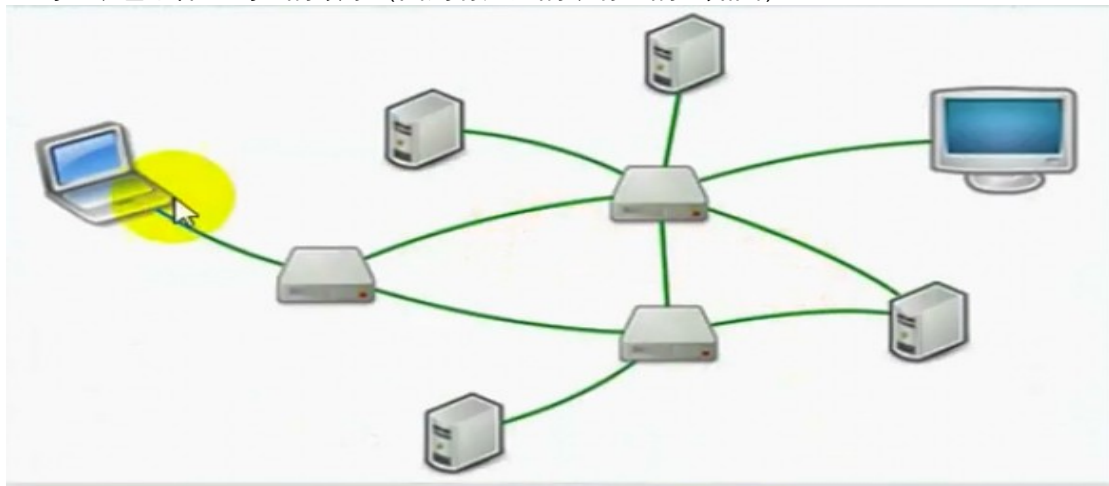
计算机网络出现比互联网要早很多。

计算机为了联网，就必须规定通信协议，早期的计算机网络，都是由各厂商自己规定的一套协议，IBM、App、和 Microsoft 都由各自的网络协议，各不兼容，就好比一群人由的说英语，由的说中文，由的说德语，说同一种语言的人可以交流，不同语言的人就不行了。为了把汉斯杰的所有不同类型的计算机都链接起来，就必须规定一套全球通用的协议，为了实现互联网的这个目标，互联网协议簇(Internet Protocol Suite)就是通用协议标准。Internet 是由 inter 和 net 两个单词组成的，原意就是“联通网络”由于 internet 任何私有网络，只要支持这个协议，就可以进入互联网。

因为互联网协议包含上百种协议标准，但是最重要的是 TCP 和 IP 协议，所以，大家把互联网的洗衣简称为 TCP/IP 协议。

通信的时候，双方必须知道对方的标识，好比发邮件的时候必须知道对方的邮件地址，在互联网上每个计算机的唯一标识就是 IP 地址，类似 123.123.123.123 如果一台计算机同事接入到两个或者更多的网络，比如路由器，它就会有多个 IP 地址，所以，IP 地址对应的实际上是计算机网络的接口。通常是网卡。

IP 协议负责把数据从一台计算机发送到另一台计算机，数据被分成很多块，然后通过 ip 包发送出去，由于互联网链路复杂，两台计算机之间经常由多条路线，因此路由器就负责决定如何把一个 IP 包转发出去，IP 包的特点就是按块发送，途径多个路由器，但不能保证到达，也不保证到达的顺序（因为有走短的，有绕的。如图）



IP 地址实际上是一个 32 位证书（成为 ipv4），以字符串表示的 IP 地址如 192.168.0.1 实际上是把 32 位证书按照 8 位分组后的数字表示，目的是便于阅读。

IP6 得知实际上是一个 128 位的证书，它是目前使用 IPv4 的升级版，以字符串表示类似于 2001:0db8:85a3:0042:1000:8a2e:0370:7334

TCP 协议是建立在 IP 协议之上的，它负责在两台计算机之间建立可靠的连接，保证数

据包的顺序到达，TCP 协议会通过握手建立连接，然后对每个 IP 包编号，确保对方按顺序收到，如果包掉了，就自动重发。



模拟一个包发送的过程：

```
ip包tcp包 客-服 薛延美我要跟你聊天
ip包tcp包 服-客 好滴呀
ip包tcp包 客-服 注意啊，我要开始说话了
经过以上tcp链接就建立了
ip包tcp包[sunck] 客-服
ip包tcp包[is] 客-服
ip包tcp包[a] 客-服
ip包tcp包[good] 客-服
ip包tcp包[man] 客-服

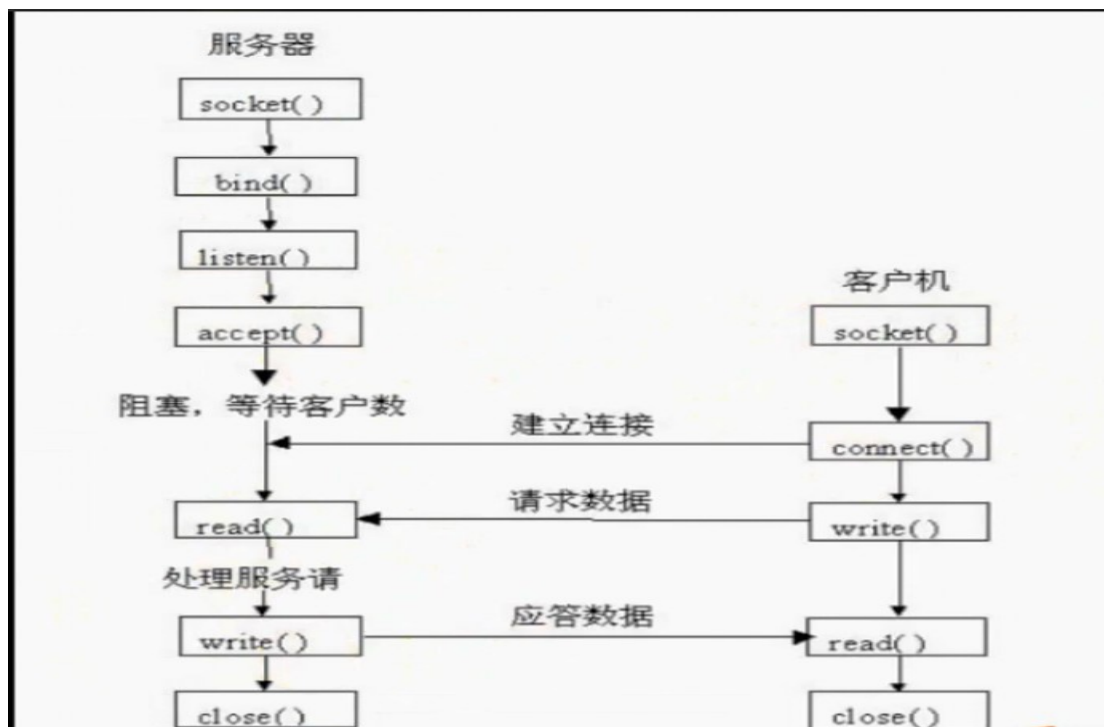
ip包tcp包[yes,you are] 服-客
ip包tcp包[ a good man] 服-客

ip包tcp包 服-客 FIN 我不想跟你聊了
ip包tcp包 客-服 ACK 我知道了
ip包tcp包 客-服 FIN 我早就不想跟你聊了
ip包tcp包 服-客 ACK 我知道了，咱俩断了
```

客户端：创建 TCP 连接时，主动发起连接的叫做客户端。

服务端：接受客户端的连接

TCP 通信原理：



代码实现：import socket (socket 包含了网络编程所有的东西) 英语中的意思位套接字使程序可以收发信息。下面代码是与新浪服务器的交互的一个例子：

```
import socket
# 1 创建一个socket
sk = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# 参数1: 指定协议(AF_INET为IPV4协议, AF_INET6为IPV6协议)
# 参数2: SOCK_STREAM执行使用面向流的TCP协议。
# 2 建立连接# connect参数是一个address, 为一个两个元素的元组前面为ip地址(域名), 后面位端口号。
sk.connect(('www.sina.com.cn', 80))
# 3 信息交流
sk.send(b'GET / HTTP/1.1\r\nHost: www.sina.com.cn\r\nConnection: close\r\n\r\n')
# 等待接受数据
data = []
while True:
    # 每次接受1K数据
    temdata = sk.recv(1024)
    if temdata:
        data.append(temdata)
    else:
        break
dataStr = (b''.join(data)).decode('utf-8')
# 断开连接
sk.close()
print(dataStr)
```

客户端与服务器的信息交换示例代码：(单个连接) 自己连接自己电脑的。

服务器代码：

```

import socket
# 创建一个socket
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# 绑定ip端口
server.bind(('192.168.2.101', 8081))
# 监听
server.listen(5)
print('服务器启动成功...')
clientSocket, clientAddress = server.accept()
while True:
    data = clientSocket.recv(1024)
    print('客户端说: ', data.decode('utf-8'))
    sendData = input('输入给返回客户端的数据: ')
    clientSocket.send(sendData.encode('utf-8'))

```

服务器启动成功...

客户端说: 现在可以聊天了吗?

输入给返回客户端的数据: 可以了。

客户端说: 你可以看到吗?

输入给返回客户端的数据: 恩恩是的, 我们在聊天了。

客户端代码:

```

import socket

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(('192.168.2.101', 8081))

while True:
    data = input('请输入给服务器发送的数据: ')
    client.send(data.encode('utf-8'))# 不能给服务器发送字符串, 需要编码成二进制才可以。
    info = client.recv(1024)
    print('服务器说: ', info.decode('utf-8'))

```

请输入给服务器发送的数据: 现在可以聊天了吗?

服务器说: 可以了。

请输入给服务器发送的数据: 你可以看到吗?

服务器说: 恩恩是的, 我们在聊天了。

请输入给服务器发送的数据:

UDP 通信 (主要可以做广播)

相关概念: TCP 是建立可靠的连接, 并且通信双方都可以以流的形式发送数据, 相对于 TCP, UDP 是面向无连接的协议。使用 UDP 协议时, 不需要建立连接, 只需要知道对方的 IP 地址和端口号, 就可以直接发送数据包, 但是能不能到达就不知道了。

虽然 UDP 传输数据不可靠, 但是比 TCP 速度快, 对于要求不高的数据可以用 UDP。



客户端：

```
import socket
```

```
client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

```
while True:
```

```
    data = input("请输入要发送的数据: ")
```

```
    client.sendto(data.encode('utf-8'), ('192.168.2.101', 8090))
```

```
客户端说: 111
```

```
客户端说: 1235
```

服务端：

```
import socket
```

```
udpServer = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

```
udpServer.bind(('192.168.2.101', 8090))
```

```
while True:
```

```
    data, addr = udpServer.recvfrom(1024)
```

```
    print('客户端说: ', data.decode('utf-8'))
```

```
客户端说: 111
```

```
客户端说: 1235
```

多任务原理

现代的操作系统，Windows Mac OS X Linux UNIX 等都支持多任务的。

多任务：操作系统可以同时运行多个任务。

早期的 cpu 都是单核 cpu 多任务原理：操作系统轮流让任务交替执行。比如说 QQ 执行 0.02 秒，然后切换到酷狗 0.02 秒，在切换到微信 0.02 秒…表面上看每个任务都是反复执行下去的，但是 cpu 调度执行速度太快了，导致感觉所有任务在同时执行一样。

多核心 cpu 实现多任务原理：真正的并行执行多任务只能是在多核 cpu 上面实现，但是由于任务数量远多于核心数量，所以操作系统也会自动把很多任务轮流到每一个核心上执行。

并发：看上去一起执行，任务多于 cpu 核心数

并行：真正一起执行，任务数少于或等于 cpu 核心数。

实现多任务的方式：

- 1 多进程模式
- 2 多线程模式
- 3 协程模式（相对来说比较少用）
- 4 多进程加多线程模式（一般不建议使用，容易乱）

进程

对于操作系统来说，一个任务就是一个进程。

进程是系统中程序执行和资源分配的基本单位，每个进程都有自己的数据段，代码段，和堆栈段。（堆：存储对象的，需要手动开辟和释放内存，栈：普通的变量，比如收某个程序内部的变量。对其他程序不影响。因为每个进程都有自己的堆栈。）

单任务现象：

```
import time
def run():
    while True:
        print('sunck is a nice man!')
        time.sleep(1.2)

if __name__ == '__main__':
    while True:
        print('suck is a good man!')
        time.sleep(1)
    # 不会执行到run(), 只有让上面的while循环结束才可以执行。
    run()
suck is a good man!
suck is a good man!
suck is a good man!
```

所以如果想让 run() 执行就必需要多进程。

启动多进程实现多任务：

multiprocessing 库（跨平台版本的多进程模块，提供一个 Process 类来代表一个进程对象。）

Process (target, args)

参数 target 为要运行的进程（函数）

参数 args 为函数的参数（必须为元组的形式，如果只有一个要记得后面加上',')
' ,')

创建进程之后用.start()开始一个进程。

```
from multiprocessing import Process
import time
def run(str):
    while True:
        print('sunck is a %s man!' %(str))
        time.sleep(1.2)

if __name__ == '__main__':
    print('主（父）进程启动...')
    # 创建子进程
    p = Process(target=run, args=('Nice',))
    p.start()

    while True:
        print('suck is a good man!')
        time.sleep(1)
```

父子进程的先后顺序：

父进程的结束不能影响子进程

```
from multiprocessing import Process
import time
def run(str):
    print('子进程开始')
    time.sleep(1.2)
    print('子进程结束')

if __name__ == '__main__':
    print('主（父）进程启动...')
    p = Process(target=run, args=('Nice',))
    p.start()
    print('主（父）进程结束...')
```

```
主（父）进程启动...
主（父）进程结束...
子进程开始
子进程结束
```

一般情况下需要让父进程等待子进程结束只有再结束比较容易了解进程的情况。只需要在.start()后面加上.join()方法。

```

from multiprocessing import Process
import time
def run(str):
    print('子进程开始')
    time.sleep(3)
    print('子进程结束')

if __name__ == '__main__':
    print('主（父）进程启动...')
    p = Process(target=run, args=('Nice',))
    p.start()
    p.join()
    print('主（父）进程结束...')

```

主（父）进程启动...
子进程开始
子进程结束
主（父）进程结束...

这样看着好像又变成了一个进程了，但是在 p.start() 前面还可以创建多个 p1, p2.. 进程，而父进程可以不用干任何事，只是作为所有进程结束的标志。

全局变量在多个进程中不能共享（对应上面进程中的堆栈说明，每个进程由自己的堆栈）

下代码中的 global num == num = 100

```

from multiprocessing import Process
num = 100
def run():
    print('子进程开始')
    global num # global 申明 num 为全局变量
    num += 1
    print(num)
    print('子进程结束')

if __name__ == '__main__':
    print('父进程开始')
    p = Process(target=run)
    p.start()
    p.join()
    print('父进程结束----%d' % num)

```

父进程开始
子进程开始
101
子进程结束
父进程结束----100

利用多进程创建文件拷贝的代码：

单进程代码：

```

import os
def copyFile(rpath, wpath):
    fr = open(rpath, 'rb')
    fw = open(wpath, 'wb')
    context = fr.read()
    fw.write(context)
    fr.close()
    fw.close()

if __name__ == '__main__':
    path = r'C:\Users\Administrator\Desktop\file'
    topath = r'C:\Users\Administrator\Desktop\tofile'
    filelist = os.listdir(path)
    for filename in filelist:
        copyFile(os.path.join(path, filename), os.path.join(topath, filename))

```

多进程代码：

```

from multiprocessing import Pool
import os
def copyFile(opath, topath):
    rf = open(opath, 'rb')
    wf = open(topath, 'wb')
    contents = rf.read()
    wf.write(contents)
    rf.close()
    wf.close()
if __name__ == '__main__':
    path1 = r'C:\Users\Administrator\Desktop\file'
    path2 = r'C:\Users\Administrator\Desktop\tofile'
    pp = Pool(2)
    filelist = os.listdir(path1)
    for filename in filelist:
        pp.apply_async(copyFile, args= (os.path.join(path1, filename), os.path.join(path2, filename)))
    pp.close()
    pp.join()

```

为了使代码更加简洁，可以把进程需要运行的方法封装进一个进程对象。

自定义自己的进程继承自 Process：

```

from multiprocessing import Process
import os
class FrankProcess(Process):
    def __init__(self, name):
        Process.__init__(self)
        self.name = name
    def run(self):
        print('子进程 (%s---%s) 启动' % (self.name, os.getpid()))
        print("I'm Frank")
        print('子进程 (%s---%s) 结束' % (self.name, os.getpid()))

```

创建主程序调用自己的类进程

```

from FrankProcess import FrankProcess
if __name__ == '__main__':
    print('父进程开始启动')
    p = FrankProcess('test')
    p.start()
    p.join()
    print('父进程结束')

```

```

父进程开始启动
子进程 (test---7416) 启动
I'm Frank
子进程 (test---7416) 结束
父进程结束

```

进程间的通信：Queue

队列代码示例：

```

from multiprocessing import Process, Queue
import os, time, random

# 写数据进程执行的代码
def proc_write(q, urls):
    print('Process(%) is writing...' % os.getpid())
    for url in urls:
        q.put(url) # put意思为从队的尾部插入一个队列
        print('Put %s to queue...' % url)
        time.sleep(random.random())

# 读取进程执行的代码
def proc_read(q):
    print('Process(%) is reading...' % os.getpid())
    while True:
        # while not q.empty():
        url = q.get(True) # get意思为从队头部读出列
        print('Get %s from queue.' % url)

if __name__ == '__main__':
    # 父进程创建queue, 并传给各个子进程
    q = Queue() # 创建一个空的队列
    proc_writer1 = Process(target=proc_write, args=(q, ['url_1', 'url_2', 'url_3']))
    proc_writer2 = Process(target=proc_write, args=(q, ['url_4', 'url_5', 'url_6']))
    proc_reader = Process(target=proc_read, args=(q,))
    # 启动子进程proc_writer 写入
    proc_writer1.start()
    proc_writer2.start()
    # 启动子进程proc_reader, 读取
    proc_reader.start()
    # 等待proc_writer结束
    proc_writer1.join()
    proc_writer2.join()
    # proc_reader进程里面是死循环, 无法等待其结束, 只能终止
    proc_reader.terminate()

Process(7328) is writing...
Put url_4 to queue...
Process(7168) is reading...
Get url_4 from queue.
Process(7140) is writing...
Put url_1 to queue...
Get url_1 from queue.
Put url_5 to queue...
Get url_5 from queue.
Get url_2 from queue.
Put url_2 to queue...
Put url_3 to queue...
Get url_3 from queue.
Put url_6 to queue...
Get url_6 from queue.

```

线程

一个进程需要同时干多件事，就需要同时运行多个‘子任务’，这些子任务就是线程。

线程通常也可以叫轻量级的进程。线程是共享内存空间并发执行的多任务。每个线程共享同一个进程的资源。线程是最小的执行单元，每个进程至少由一个线程。如何调动进程和线程全部由操作系统决定。程序自己不能决定什么时候执行，执行多长时间。

模块：

1 _thread 低级模块

2 threading 高级模块对_thread 进行了封装。

线程的实现代码：

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-
import random
import time
import threading
# 新线程执行的代码
def thread_run(urls):
    print('Current %s is running...' % threading.current_thread().name)
    for url in urls:
        print('%s----->>>%s' % (threading.current_thread().name, url))
        time.sleep(random.random())

    print('%s ended.' % threading.current_thread().name)
print('%s is running...' % threading.current_thread().name)
t1 = threading.Thread(target=thread_run, name='Thread_1', args=(['url_1', 'url_2', 'url_3'],))
t2 = threading.Thread(target=thread_run, name='thread_2', args=(['url4', 'url5', 'url6'],))
t1.start()
t2.start()
t1.join()
t2.join()
print('%s ended.' % threading.current_thread().name)
Thread_1----->>>url_1
Current thread_2 is running...
thread_2----->>>url4
thread_2----->>>url5
Thread_1----->>>url_2
thread_2----->>>url6
thread_2 ended.
Thread_1----->>>url_3
Thread_1 ended.
MainThread ended.
```

对比线程与进程：多进程与多线程最大的不同在于：多进程中，同一个变量，各自有一份拷贝存在每个进程中，互不影响。而多线程中所有的变量都由所有线程共享。所以，任何一个变量都可以被任意线程所修改。因此线程之间共享数据最大的危险在于多个线程同时修改一个变量，容易把内容改乱。

利用线程锁

解决多个线程同时修改全局变量，导致变量变乱的问题：（一下结果不用锁就会变乱，用锁则恒为0）

```

import time, threading
balance = 0
lock = threading.Lock()
def change_it(n):
    global balance
    balance += n
    balance -= n
def run_thread(n):
    for i in range(10000000):
        lock.acquire() # 给线程上锁，确保一次只进行一个线程。
        try:
            change_it(n)
        finally:
            lock.release() # 释放线程，必须要释放不然后面就是阻塞成为死循环
t1 = threading.Thread(target=run_thread, args=(5,))
t2 = threading.Thread(target=run_thread, args=(8,))
t1.start()
t2.start()
t1.join()
t2.join()
print(balance)
0

```

此段代码上锁还可以用 `with lock :` 的形式写用 `with lock :` 代替 `try: lock.acquire()``finally: lock.release()` 同样可以给代码上锁解锁，原理与 `with open() as f :` 自动关闭文件类似。

分析：给线程上锁可以确保上锁的代码只能由一个线程从头到尾执行，组织了多线程的并发，所以说包含锁的某段代码实际上只能以单线程的模式执行。所以效率大大滴降低了。

利用 Thread.Local

解决线程变量变乱问题：相当于每个线程都有一个 `Local.x`（局部变量），互不影响。

作用：为每一个线程绑定一个数据库链接，HTTP 请求，用户身份信息等等，这样一个线程的所有调用到的处理函数都可以非常方便地访问这些资源。

`num = 0`

```

def run(x, n):
    x = x + n
    x = x - n

def func(n):
    local.x = num
    for i in range(1000000):
        run(local.x, n)
    print("%s-%d"%(threading.current_thread().name, local.x))

if __name__ == "__main__":
    t1 = threading.Thread(target=func, args=(6,))
    t2 = threading.Thread(target=func, args=(9,))
    t1.start()
    t2.start()
    t1.join()
    t2.join()

    print("num =", num)

```

信号量控制线程数量

```
import threading,time
sem = threading.Semaphore(2) #控制一次只有几个线程可以执行。
def run():
    with sem:
        for i in range(5):
            print('%s---%d'% (threading.current_thread().name, i))
            time.sleep(1)

if __name__ == '__main__':
    for i in range(5):
        threading.Thread(target=run).start()
```

凑够一定数量才能一起执行

看打印结果：因为前面 5 个开始了，后面只有还有五个，但是要凑够 3 个才运行，所以 4,5 没有 end。

```
import threading,time
bar = threading.Barrier(3) #表示要凑够3个才执行。
def run():
    print('%s---start'% threading.current_thread().name)
    time.sleep(1)
    bar.wait()
    print('%s---end' % threading.current_thread().name)
if __name__ == '__main__':
    for i in range(5):
        threading.Thread(target=run).start()

Thread-1---start
Thread-2---start
Thread-3---start
Thread-4---start
Thread-5---start
Thread-1---end
Thread-3---end
Thread-2---end
```

定时线程：

```
import threading
def run():
    print('sunck is good man!')
# 延迟执行线程。
t = threading.Timer(5, run)# 表示线程创建之后调用start不立即执行，要等5秒后执行。
print('父进程开始')
t.start()
t.join()
print('父进程结束')
父进程开始
sunck is good man!
父进程结束
```

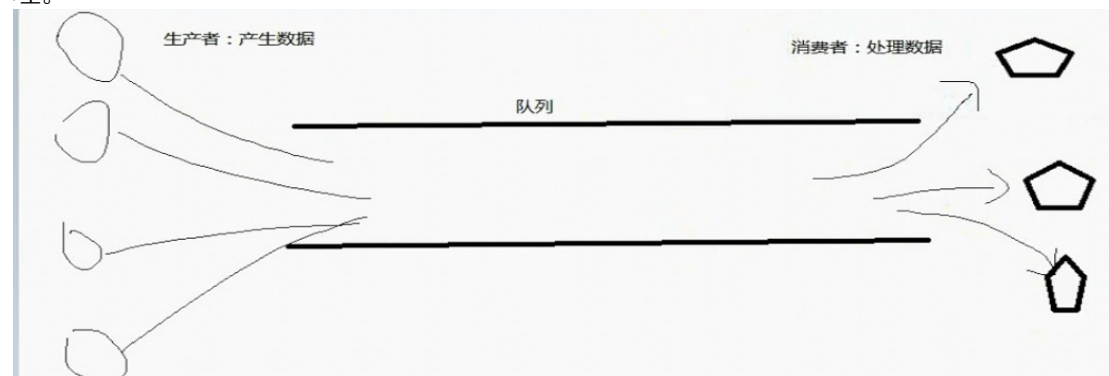

线程通信：

```
import time
def fun():
    # 事件对象
    event = threading.Event()
    def run():
        for i in range(5):
            # 阻塞等待事件的触发
            event.wait()
            event.clear()#启动后之后的重置
            print('sunck is a good man!! %d' % i)
    t = threading.Thread(target=run).start()
    return event

e = fun()
for i in range(5):
    time.sleep(2)
    e.set()#触发事件
每隔 2 秒触发运行一次。
sunck is a good man!! 0
sunck is a good man!! 1
sunck is a good man!! 2
sunck is a good man!! 3
sunck is a good man!! 4
```

生产者和消费者

简单理解：生产者（线程）产生的数据放到 Queue（队列）里，然后消费者将其拿出来处理。



```
import threading,queue,time,random
# 生产者
def product(id, q):
    while True:
        num = random.randint(0,1000)
        q.put(num)
        print('生产者%d生产了%d数据放入了队列'% (id, num))
        time.sleep(3)
    #任务完成
    q.task_done()
```

```

# 消费者
def customer(id, q):
    while True:
        item = q.get()
        if item is None:
            break
        print('消费者%d消费了%d数据'%(id, item))
        time.sleep(2)
    # 任务完成
    p.task_done()
if __name__ == '__main__':
    # 消息队列
    q = queue.Queue()
    # 启动生产者
    for i in range(4):
        threading.Thread(target=product, args=(i,q)).start()
    # 启动消费者
    for i in range(3):
        threading.Thread(target=customer, args=(i, q)).start()

```

生产者0生产了979数据放入了队列
 生产者1生产了712数据放入了队列
 生产者2生产了793数据放入了队列
 生产者3生产了499数据放入了队列
 消费者0消费了979数据
 消费者1消费了712数据
 消费者2消费了793数据

线程的调度:

cond.wait 相当停一下，让下面的先运行，下面运行又 cond.wait()，然后 cond.notify()相当于告诉上面的我运行完了，你来吧！然后上面又运行。此方法两个的时候较为容易理清楚，但是线程一旦多了就比较复杂。

```

import threading
import time
# 线程条件变量
cond = threading.Condition()
def run1():
    with cond:
        for i in range(0, 10, 2):
            print(threading.current_thread().name, i)
            time.sleep(1)
            cond.wait()
            cond.notify()
def run2():
    with cond:
        for i in range(1, 10, 2):
            print(threading.current_thread().name, i)
            time.sleep(1)
            cond.notify()
            cond.wait()
threading.Thread(target=run1).start()
threading.Thread(target=run2).start()

```

进程 VS 线程

多任务的实现原理：通常会设计 Master-Worker 模式，Master 负责分配任务

Worker 负责执行任务，因此多任务环境下，通常是一个 Master，多个 Worker，主进程就是 Master，其他进程就是 Worker。

多进程优点：稳定性高，一个子进程崩溃，不会影响主进程和其他子进程看，当然主进程崩溃所有进程都会挂掉看，但是 Master 只负责分配任务，一般不容易出现问题。

缺点：创建进程的代价大，操作系统同时运行进程的数量是有限的（在内存和 CPU 的限制下，如果由几千个进程，操作系统连调度都会成问题。），在 Unix/Linux 系统下用 fork 调用还行，在 windows 下创建进程开销巨大。

多线程：主线程就是 Master，其他线程都是 Worker

优点通常比多进程要快一点（快不了多少）在 windows 下多线程的效率比多进程要高

缺点任何一个线程出问题，可能造成整个进程的崩溃（因为其共享内存）

计算密集型与IO 密集型

计算密集型：要进行大量的计算，消耗 cpu 资源，比如**计算圆周率，对视频进行高清解码等**，全靠 cpu 的运算能力，这种任务虽然也可以用多任务完成，但是任务越多，花在任务切换的时间就越多，cpu 效率就越底，所以要最高效利用 cpu，计算密集型任务同事进行的数量应当等于 cpu 核心数。

IO 密集型：设计到网络、磁盘 IO 任务的都是 IO 密集型，特点是 cpu 消耗很少，任务的大部分时间都在等待 IO 操作完成（因为 IO 操作速度远低于 CPU 和内存速度）对于 IO 密集型任务，任务越多，cpu 效率越高，但也有一个限度。常见的由 Web 应用，

协程

基础协程

```
def C():
    print('C----star')
    print('C----end')

def B():
    print('B----star')
    C()
    print('B----end')

def A():
    print('A----star')
    B()
    print('A----end')
A----star
B----star
C----star
C----end
B----end
A----end
```

协程：看上去也是子程序，但执行过程中，在子程序的内部可中断，然后转而去执行别的子程序，不是函数调用，有点类似 CPU 中断。

与线程相比：协程的执行效率极高，因为只有一个线程，不存在变量的冲突，在协程中共享资源不加锁。只需要判断状态就可以了。

协程原理：python 通过 generator 实现协程

```
def run():|
    print(1)
    yield 10
    print(2)
    yield 20
    print(3)
    yield 30
m = run()
print(m)
print(next(m))
print(next(m))
print(next(m))
<generator object run at 0x01350240>
1
10
2
20
3
30
```

协程的生产者和消费者

协程的理解：（在命令行模式下可以看到返回值，更加直观）

```
>>> def simple_coro(a):
...     print('协程开始了...')
...     b = yield a
...     print('接收到b=', b)
...     c = yield a + b
...     print('接收到c=', c)
...
>>> coro = simple_coro(10)
>>> next(coro)
协程开始了...
10
>>> coro.send(20)
接收到b= 20
30
>>> coro.send(30)
接收到c= 30
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

协程在运行过程中有四个状态：

1. GEN_CREATE:等待开始执行
2. GEN_RUNNING:解释器正在执行，这个状态一般看不到
3. GEN_SUSPENDED:在 yield 表达式处暂停
4. GEN_CLOSED:执行结束

刚开始先调用了 next(...)是因为这个时候生成器还没有启动，没有停在 yield 那里，这个时候也是无法通过 send 发送数据。所以当我们通过 next(...)激活协程后，程序就会运行到 x = yield，这里有个问题我们需要注意，x = yield 这个表达式的计算过程是先计算等号右边的内容，然后在进行赋值，所以当激活生成器后，程序会停在 yield 这里，但并没有给 x 赋值。

yield 左边的是接受过来的 send()的值，右边的是整个函数返回的值。

```

import time
def consumer():
    r = ''
    while True:
        n = yield r
        if not n:
            return
        print('[CONSUMER] Consuming %s...' % n)
        time.sleep(1)
        r = '200 OK'

def produce(c):
    c.__next__()
    n = 0
    while n < 5:
        n = n + 1
        print('[PRODUCER] Producing %s...' % n)
        r = c.send(n)
        print('[PRODUCER] Consumer return: %s' % r)
    c.close()

if __name__ == '__main__':
    c = consumer()
    produce(c)

[PRODUCER] Producing 1...
[CONSUMER] Consuming 1...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 2...
[CONSUMER] Consuming 2...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 3...
[CONSUMER] Consuming 3...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 4...
[CONSUMER] Consuming 4...
[PRODUCER] Consumer return: 200 OK
[PRODUCER] Producing 5...
[CONSUMER] Consuming 5...
[PRODUCER] Consumer return: 200 OK

```

代码理解：注意到 consumer 函数是一个 generator（生成器），把一个 consumer 传入 produce 后：

1. 首先调用 c.next() 启动生成器；
2. 然后，一旦生产了东西，通过 c.send(n) 切换到 consumer 执行；
3. consumer 通过 yield 拿到消息，处理，又通过 yield 把结果传回；
4. produce 拿到 consumer 处理的结果，继续生产下一条消息；
5. produce 决定不生产了，通过 c.close() 关闭 consumer，整个过程结束。

整个流程无锁，由一个线程执行，produce 和 consumer 协作完成任务，所以称为“协程”，而非线程的抢占式多任务。最后套用 Donald Knuth 的一句话总结协程的特点：“子程序就是协程的一种特例。”

greenlet

一个 greenlet 是一个独立的微小线程。可以将它想想成一个堆栈，栈底是初始功能，而栈顶是当前 greenlet 的暂停位置。可以通过创建多个这样的堆栈，然后再它们之间跳跃执行。跳转不是绝对的，一个 greenlet 必须跳到另一个已经选择好的 greenlet，这将导致前者挂起，后者恢复。Greenlets 之间的跳转成为（switch）

当你创建一个 greenlet 的时候，得到一个初始为空的栈，第一次切换到它，就会开始运行指定函数，然后可以切换出 greenlet 当最终底部函数结束时，greenlet 堆栈再次

为空，greenlet 就 dead 了。greenlets 也会因为一个未捕获的异常而 dead 例如：

```
from greenlet import greenlet
def test1():
    print (12)
    gr2.switch()
    print (34)
def test2():
    print (56)
    gr1.switch()
    print (78)
gr1 = greenlet(test1)
gr2 = greenlet(test2)
gr1.switch()
12
56
34
```

代码解释：最后一行跳转到 test1，打印 12，跳转到 test2，打印 56，跳回 test1，打印 34；然后 test1 完成并且 gr1 死亡。此时，执行返回到原始 gr1.switch() 呼叫。请注意，永远不会打印 78。

关于父母 greenlet

每一个 Greenlet 都有一个 parent，一个新的 greenlet 在哪里创生，*当前环境的 greenlet 就是这个新 greenlet 的 parent*。所有的 greenlet 构成一棵树，其跟节点就是还没有手动创建 greenlet 时候的“main” greenlet（事实上，在首次 import greenlet 的时候实例化）。当一个协程 正常结束，执行流程回到其对应的 parent；或者在一个协程中抛出未被捕获的异常，该异常也是传递到其 parent。

在上述代码中，gr1 和 gr2 都将 main greenlet 作为父级，当其中一个死亡，执行就会回到 main greentlet

切换

对于 greenlet，最常用的写法是 `x = gr.switch(y)`。这句话的意思是切换到 gr，传入参数 y。当从其他协程（不一定是这个 gr）切换回来的时候，将值付给 x。或者当 greenlet 死掉时，执行会跳转到父 greenlet。在切换期间，对象或异常被“发送”到目标 greenlet；这可以用作在 greenlet 之间传递信息的便捷方式。

```
import greenlet
def test1(x, y):
    z = gr2.switch(x+y)
    print('test1 ', z)
def test2(u):
    print('test2 ', u)
    gr1.switch(10)
gr1 = greenlet.greenlet(test1)
gr2 = greenlet.greenlet(test2)
gr1.switch("hello", " world")
'test2 ', 'hello', 'world'
'test1 ', 10
```

代码理解：第 12 行从 main greenlet 切换到了 gr1，test1 第 3 行切换到了 gr2，然后 gr1 挂起，第 8 行从 gr2 切回 gr1 时，第 6 行将值（10）返回值给了 z。

子程序/子函数：所有语言中都是层级调用，比如 A 调用 B 在 B 执行过程中又可以调用 C，C 执行完毕返回，B 执行完毕返回，最后是 A 执行完毕。是通过栈实现的，一个线程就是执行一个子程序，子程序调用总是一个入口，一次返回。调用的顺序是明确的。

gevent

对比 gevent 协程，以下代码需要至少 30 秒执行完毕。

```
import time
def show_wait(name, n):
    for i in range(n):
        print(name, ' 等待了 ', i+1, ' 秒 ')
        time.sleep(1)
show_wait('庞子卓', 10)
show_wait('韩海飞', 10)
show_wait('李海宝', 10)
```

协程优化后代码：三个一起执行，10 秒左右搞定。

```
import gevent
def show_wait(name, n):
    for i in range(n):
        print(name, ' 等待了 ', i+1, ' 秒 ')
        gevent.sleep(1)
# 下面一定要注意运行的函数与参数，参数是不需要带括号的。如果带括号就变成了函数调用，不是协程调用了，速度就会没有变化。
g1 = gevent.spawn(show_wait, '庞子卓', 10)
g2 = gevent.spawn(show_wait, '韩海飞', 10)
g3 = gevent.spawn(show_wait, '李海宝', 10)
# gevent.joinall([g1, g2, g3]) 此条语句等价于下面三条语句
g1.join()
g2.join()
g3.join()
```

gevent 协程网页下载

import gevent.monkey #此代码的作用是当有 IO 等待的时候，无需手动激活，协程自动切换，提高效率把 import gevent, from gevent import monkey, monkey.patch_all() 三行语句放在其他所有的 import 语句之前，可以避免出现警告或者报错信息，导致程序不能正常运行。

```
import requests
def down(url):
    print(url, 'start...')
    data = requests.get(url).text
    print('length:', len(data))
gevent.joinall([gevent.spawn(down, 'http://www.baidu.com'),
                gevent.spawn(down, 'http://www.qq.com'),
                gevent.spawn(down, 'http://www.163.com')])
http://www.baidu.com start...
http://www.qq.com start...
http://www.163.com start...
length: 2381
length: 679086
length: 224266
```


python 2 与 python3 对比

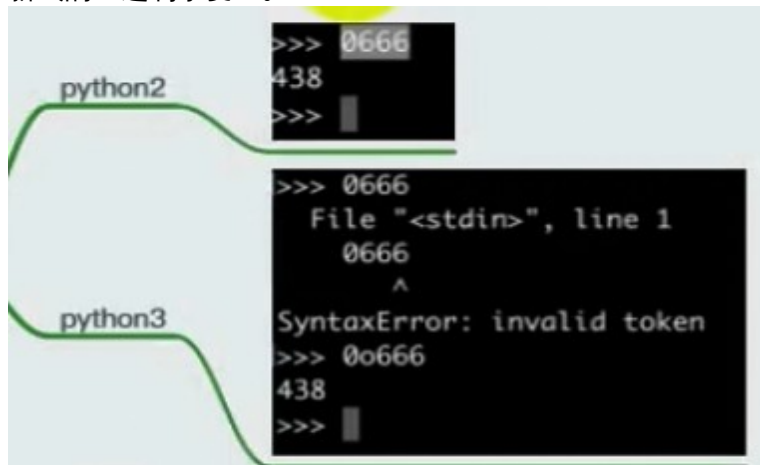
性能：python3 起始比 python2 效率低，但是 python3 有极大的优化空间，效率追赶。

编码：python3 源码文件默认使用 utf-8 使变量名更为广阔。

- 1 去除了 <>，改用 !=
- 2 加入 as 和 with 关键字，还有 True，False，None
- 3 整型触发返回浮点数，整除请使用 //
- 4 加入了 nonlocal 语句。
- 5 去除了 print 语句加入 print() 函数。
- 6 去除了 raw_input，加入 input() 函数。
- 7 新的 super()，可以不给 super() 传参数。

```
class C(object):
    def __init__(self, a):
        print('C', a)
class D(C):
    def __init__(self, a):
        super().__init__(a) # 无参数调用 super()
```

- 8 改变了顺序操作符的行为，例如：x<y，当 x 和 y 类型不匹配时抛出 TypeError 而不是返回随即的 bool 值。
- 9 新式的 8 进制字变量。



- 10 字符串和字节串：py2：字符串以 8-bit 字符串存储
py3 字符串以 16-bit Unicode 字符串存储，现在字符串只有 str 一种类型。
- 11 py3 去除了 long 类型，只有一种 int 但他的行为就像 py2 版本的 long；新增了 bytes 类，对应 py2 中的八位串。str 和 bytes 对象可用 encode() or decode() 方法相互转化。
- 12 面向对象：引入抽象基类。
- 13 异常：所有异常都是从 BaseException 继承，并删除了 StandardError
python2：try.....except exception, e;
python3: try.....except Exception as e:
- 14 xrange() 改为 range()，要想使用 range() 获得一个 list，必须显示调用。
- 15 file 类型被废弃 python2 打开文件：file (path) 和 open (path) 都可以
python3 open (path)