## 1、为什么学习Python？

http://www.cnblogs.com/clement-jiao/p/8719323.html#autoid-8-0-0

1. 编译型语言：一次性，将全部的程序编译成二进制文件，然后在运行。（c，c++ ，go）
   * 1. 优点：运行速度快。
     2. 缺点：开发效率低，不能跨平台。
2. 解释型语言：当你的程序运行时，一行一行的解释，并运行。（python ， PHP）
   * 1. 优点：调试代码很方便，开发效率高，并且可以跨平台。
     2. 缺点：运行速度慢。
3. 我对程序的定义是人可以读懂，而机器刚好可以执行的一段代码，注重于代码的可读性。
   1. 而Python的定位是“优雅”、“明确”、“简单”，用它编写的程序简单易懂，这与我当初的想法不谋而合。

## 2、****谈谈你对python和其他语言的区别****

1. python 中，变量是以内容为基准而不是像 c 中以变量名为基准；
2. python 中，一个变量可以以多个名称访问；
3. python 中，数字类型的值是不可变的；
4. python 中，编译器会有一个小整数池的概念

## 3、Python和Java、PHP、C、C#、C++等其他语言的对比？

## 4、简述解释型和编译型编程语言？

python  解释型（函数写在调用前）

java   编译型（请求次数多）+解释型（请求次数少）

**解释性语言定义：**

程序不需要编译，在运行程序的时候才翻译，每个语句都是执行的时候才翻译。这样解释性语言每执行一次就需要逐行翻译一次，效率比较低。

现代解释性语言通常把源程序编译成中间代码，然后用解释器把中间代码一条条翻译成目标机器代码，一条条执行。

**编译性语言定义：**

编译性语言写的程序在被执行之前，需要一个专门的编译过程，把程序编译成为机器语言的文件，比如exe文件，以后要运行的话就不用重新翻译了，直接使用编译的结果就行了（exe文件），因为翻译只做了一次，运行时不需要翻译，所以编译型语言的程序执行效率高。

**Python工作过程：**

python 是解释型的编程语言。也可以把python脚本编译成pyc文件，不然编译后也是一种python虚拟指令，在python中运行。

Python先把代码编译成字节码，在对字节码解释执行。字节码在python虚拟机程序里对应的是PyCodeObject对象，pyc文件是字节码在磁盘上的表现形式。

pyc的创建时机是模块加载的时候，即import。

python test.py会对test.py进行编译成字节码并解释执行，但是不会生成test.pyc。

**Java工作过程：**

JAVA的第一道工序是javac编译，当然目标文件是BYTECODE（字节码）。后续可能有三种处理方式：

1. 运行时，BYTECODE由JVM逐条解释执行；

2. 运行时，部分代码可能由JIT翻译为目标机器指令（以method为翻译单位，还会保存起来，第二次执行就不用翻译了）直接执行；

3. RTSJ。继JAVAC之后执行AOT二次编译，生成静态的目标平台代码（典型的就是IBM WEBSHPERE REAL TIME）。

其实，现在用编译型、解释型来分类编程语言已经有点力不从心了。总体来讲，java的编译结果是被jvm“解释执行”的，所以这么说也能说通，而其实这个“是编译还是解释”这个概念在这里已经有点模糊了，理解它的过程就行了，不必下一个“精确”的定义;

而我自己仍然赞成“java是编译型语言”的说法，因为“编译”其本质就是“把一个相对高级的语言转换为另一个相对低级的语言”

**字节码和机器码：**

字节码和机器码（或者native code）的区别：  
C代码被编译成机器码，将在处理器上直接执行。每一条指令控制CPU工作。

Java代码被编译成字节码，将在Java虚拟机(JVM)这个抽象的计算机上执行。每一条指令由JVM处理，JVM同计算机本身之间交互。

简而言之：机器码快的多，但字节码更易迁移，也更安全。

## 5、Python解释器种类以及特点？

Python的种类：

- Cpython：使用 C 解释器来解释扩展名：.pyc C解释器 -> .pyc(字节码) -> 机器码 -> cpu

- Jpython：使用 Java 解释器 java 字节码->机器码 -> cpu

- ironpython：使用C#解释器 C# 字节码 -> 机器码 -> cpu

PYPY：运行速度快，原理：解释器->字节码->编译成机器码->直接运行机器码，用户运行时直接使用机器码，所以运行速度快。

## 6、位和字节的关系？

一个字节 8位，计算机计算时是以字节为单位，存储是以位为单位。

## 7、b、B、KB、MB、GB 的关系？

1G=1024M   
B是一个电脑存储的基本单位(字节),1个英文字符是1个字节,也就是1B，1个汉字为2个字符，也就是2B。   
K 是千的意思， KB也就是1000字节，但计算机的运算和数学有所不同，是1024字节为 1KB，所以说 1024B=1KB   
M 是兆的意思，运算也是类似 ， 以1024进一位， 也就是说1024KB=1MB   
依此类推 ， 1024 MB = 1 GB   
B——比特   
KB——千比特   
MB——兆比特（一般计算机的内存条都是用此来计算单位的）   
GB——吉比特（一般计算机的CPU都是用此来计算单位的）  
KB 千字节   
MB 兆字节   
GB 吉字节

## 8、PEP8 Python 编码规范

代码编排

- 缩进。4个空格的缩进

- 每行最大长度79，换行可以使用反斜杠

- 类和top-level函数定义之间空两行；类中的方法定义之间空一行；函数内逻辑无关段落之间空一行；其他地方尽量不要再空行。

文档编排

- 模块内容的顺序：按标准、三方和自己编写顺序依次排放，之间空一行

- 不要在一句import中多个库

空格的使用

注释

文档描述

网址：http://blog.sae.sina.com.cn/archives/4781

一 代码编排

1 缩进。4个空格的缩进（编辑器都可以完成此功能），不使用Tap，更不能混合使用Tap和空格。

2 每行最大长度79，换行可以使用反斜杠，最好使用圆括号。换行点要在操作符的后边敲回车。

3 类和top-level函数定义之间空两行；类中的方法定义之间空一行；函数内逻辑无关段落之间空一行；其他地方尽量不要再空行。

二 文档编排

1 模块内容的顺序：模块说明和docstring—import—globals&constants—其他定义。其中import部分，又按标准、三方和自己编写顺序依次排放，之间空一行。

2 不要在一句import中多个库，比如import os, sys不推荐。

3 如果采用from XX import XX引用库，可以省略‘module.’，都是可能出现命名冲突，这时就要采用import XX。

三 空格的使用

总体原则，避免不必要的空格。

1 各种右括号前不要加空格。

2 逗号、冒号、分号前不要加空格。

3 函数的左括号前不要加空格。如Func(1)。

4 序列的左括号前不要加空格。如list[2]。

5 操作符左右各加一个空格，不要为了对齐增加空格。

6 函数默认参数使用的赋值符左右省略空格。

7 不要将多句语句写在同一行，尽管使用‘；’允许。

8 if/for/while语句中，即使执行语句只有一句，也必须另起一行。

四 注释

总体原则，错误的注释不如没有注释。所以当一段代码发生变化时，第一件事就是要修改注释！

注释必须使用英文，最好是完整的句子，首字母大写，句后要有结束符，结束符后跟两个空格，开始下一句。如果是短语，可以省略结束符。

1 块注释，在一段代码前增加的注释。在‘#’后加一空格。段落之间以只有‘#’的行间隔。比如：

# Description : Module config.

#

# Input : None

#

# Output : None

2 行注释，在一句代码后加注释。比如：x = x + 1 # Increment x

但是这种方式尽量少使用。

3 避免无谓的注释。

五 文档描述

1 为所有的共有模块、函数、类、方法写docstrings；非共有的没有必要，但是可以写注释（在def的下一行）。

2 如果docstring要换行，参考如下例子,详见PEP 257

"""Return a foobang

Optional plotz says to frobnicate the bizbaz first.

"""

六 命名规范

总体原则，新编代码必须按下面命名风格进行，现有库的编码尽量保持风格。

1 尽量单独使用小写字母‘l’，大写字母‘O’等容易混淆的字母。

2 模块命名尽量短小，使用全部小写的方式，可以使用下划线。

3 包命名尽量短小，使用全部小写的方式，不可以使用下划线。

4 类的命名使用CapWords的方式，模块内部使用的类采用\_CapWords的方式。

5 异常命名使用CapWords+Error后缀的方式。

6 全局变量尽量只在模块内有效，类似C语言中的static。实现方法有两种，一是\_\_all\_\_机制;二是前缀一个下划线。

7 函数命名使用全部小写的方式，可以使用下划线。

8 常量命名使用全部大写的方式，可以使用下划线。

9 类的属性（方法和变量）命名使用全部小写的方式，可以使用下划线。

9 类的属性有3种作用域public、non-public和subclass API，可以理解成C++中的public、private、protected，non-public属性前，前缀一条下划线。

11 类的属性若与关键字名字冲突，后缀一下划线，尽量不要使用缩略等其他方式。

12 为避免与子类属性命名冲突，在类的一些属性前，前缀两条下划线。比如：类Foo中声明\_\_a,访问时，只能通过Foo.\_Foo\_\_a，避免歧义。如果子类也叫Foo，那就无能为力了。

13 类的方法第一个参数必须是self，而静态方法第一个参数必须是cls。

七 编码建议

1 编码中考虑到其他python实现的效率等问题，比如运算符‘+’在CPython（Python）中效率很高，都是Jython中却非常低，所以应该采用.join()的方式。

2 尽可能使用‘is’‘is not’取代‘==’，比如if x is not None 要优于if x。

3 使用基于类的异常，每个模块或包都有自己的异常类，此异常类继承自Exception。

4 异常中不要使用裸露的except，except后跟具体的exceptions。

5 异常中try的代码尽可能少。比如：

try:

value = collection[key]

except KeyError:

return key\_not\_found(key)

else:

return handle\_value(value)

要优于

try:

# Too broad!

return handle\_value(collection[key])

except KeyError:

# Will also catch KeyError raised by handle\_value()

return key\_not\_found(key)

6 使用startswith() and endswith()代替切片进行序列前缀或后缀的检查。比如

Yes: if foo.startswith(‘bar’):优于

No: if foo[:3] == ‘bar’:

7 使用isinstance()比较对象的类型。比如

Yes: if isinstance(obj, int): 优于

No: if type(obj) is type(1):

8 判断序列空或不空，有如下规则

Yes: if not seq:

if seq:

优于

No: if len(seq)

if not len(seq)

9 字符串不要以空格收尾。

10 二进制数据判断使用 if boolvalue的方式。

## 9、通过代码实现如下转换：

## 10、请编写一个函数实现将IP地址转换成一个整数。

|  |
| --- |
| import socket, struct |

|  |
| --- |
|  |

|  |
| --- |
| def ip2long(ip): |

|  |
| --- |
| """ |

|  |
| --- |
| Convert an IP string to long |

|  |
| --- |
| """ |

|  |
| --- |
| packedIP = socket.inet\_aton(ip) |

|  |
| --- |
| return struct.unpack("!L", packedIP)[0] |

## 11、python递归的最大层数？

**def** foo(n):  
 print(n)  
 n = n+1  
 foo(n)  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 foo(1)

## 12、求结果：

v1 = 1 or 3

v2 = 1 and 3

v3 = 0 and 2 and 1

v4 = 0 and 2 or 1

v5 = 0 and 2 or 1 or 4

v6 = 0 or Flase and 1

1

3

0

1

1

False

其一, 在不加括号时候, **and优先级大于or**

其二, x or y 的值**只可能是x或y**. x为真就是x, x为假就是y

第三, x and y 的值**只可能是x或y**. x为真就是y, x为假就是x

显然,

对于, 1 or 5 and 4: 先算5 and 4, 5为真, 值为4. 再算1 or 4, 1 为真,**值为1**

对于, (1 or 5) and 4: 先算1 or 5, 1为真, 值为1. 再算1 and 4, 1为真,**值为4**

## 13、ascii、unicode、utf-8、gbk 区别？

答: ASCII编码（American Standard Code for Information Interchange，美国信息互换标准代码），使用127个8进制字节表示英文和半角字符。

　　GBK （Guo Biao Kuozhan，国标扩展），当在ASCII 码的范围时，就用一个字节表示，然后每两个字节表示一个汉语和全角字符。

　　UCS（Universal Multiple-Octet Coded Character Set）俗称UNICODE，所有的字符都是两个字节。ASCII字符前面全部补零，其他字符重新编码。

　　UTF（UCS Transfer Format，万国码），当在ASCII 码的范围时，就用一个字节表示，一个中文字符占3个字节，从unicode到uft-8并不是直接的对应。

## 14、字节码和机器码的区别？

了解字节码和机器码（或者native code）的区别是很重要的，最好的办法或许是看看例子：

C代码被编译成机器码，将在处理器上直接执行。每一条指令控制CPU工作。

Java代码被编译成字节码，将在Java虚拟机(JVM)这个抽象的计算机上执行。每一条指令由JVM处理，JVM同计算机本身之间交互。

简而言之：机器码快的多，但字节码更易迁移，也更安全。

机器码随机器的变化而变化，但字节码在所有的机器上都是一样的。有人可能会认为机器码是对特定环境优化了的。

回到CPython，工具链的执行过程如下：

CPython编译你的Python源代码，生成字节码。

字节码随后在CPython虚拟机上执行。

初学者常常因为看到.pyc文件而假设Python是编译型的。这也有一些合理性：.pyc文件正式之后要解释的字节码文件。所以，你若之前运行过你的Python代码，生成了.pyc文件，再次运行时就要快得多，因为不需要再次编译生成字节码了。

## 15、三元运算规则以及应用场景？

thon中的三元运算规则 为真时的结果 if 判定条件 else为假时的结果 举个栗子 >>> a = 5 >>> b = 3 >>> c = a if a > b else b >>> c 5 >>> c = a if a < b else b >>>

## 16、列举 Python2和Python3的区别？

网址：https://blog.csdn.net/u010694764/article/details/55189681

使用\_\_future\_\_模块

print函数

整数除法

Unicode

xrange

触发异常

处理异常

next()函数和.next()方法

For循环变量与全局命名空间泄漏

比较无序类型

使用input()解析输入内容

返回可迭代对象，而不是列表

## 17、用一行代码实现数值交换：

a = 1

b = 2

## 18、Python3和Python2中 int 和 long的区别？

int <= 32 位整数  
long > 32 位整数

## 19、xrange和range的区别？

两者的区别是xrange返回的是一个可迭代的对象，range返回的则是一个列表. 同时效率更高，更快。

- 原因是实现的时候使用了 yield（此为python2.x回答，python3已删除xrange）

## 20、文件操作时：xreadlines和readlines的区别？

xreadlines = 返回一个生成器对象，

readlines = 遍历文件所有行

## 21、列举布尔值为False的常见值？

## 22、字符串、列表、元组、字典每个常用的5个方法？

## 23、lambda表达式格式以及应用场景？

## 24、pass的作用？

1. 不做任何事情，一般用做占位语句。

- 2. pass是空语句，是为了保持程序结构的完整性。

## 25、\*arg和\*\*kwarg作用

位置参数（positional argument）

关键词参数（keyword argument）

- \*args表示任何多个无名参数，它本质是一个 tuple ；

- \*\*kwargs表示关键字参数，它本质上是一个 dict ；

- 并且同时使用\*args和\*\*kwargs时，必须\*args参数列要在\*\*kwargs前。

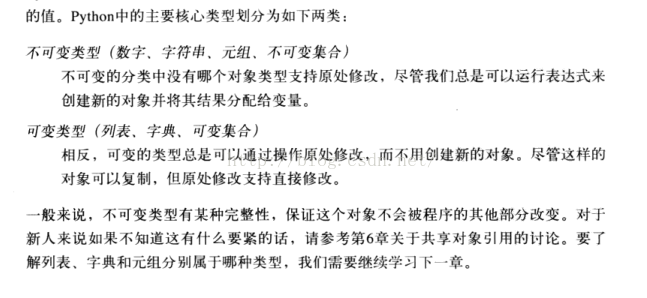
## 26、is和==的区别

- is 比较的是两个实例对象是不是完全相同，它们是不是同一个对象，占用的内存地址是否相同。莱布尼茨说过：“世界上没有两片完全相同的叶子”，这个is正是这样的比较，  
比较是不是同一片叶子（即比较的id是否相同，这id类似于人的身份证标识）。

- == 比较的是两个对象的内容是否相等，即内存地址可以不一样，内容一样就可以了。这里比较的并非是同一片叶子，可能叶子的种类或者脉络相同就可以了。

默认会调用对象的 \_\_eq\_\_()方法。

## 27、Python的可变类型和不可变类型？



总之，用一句话来概括上述过程就是：“python中的不可变数据类型，不允许变量的值发生变化，如果改变了变量的值，相当于是新建了一个对象，而对于相同的值的对象，在内存中则只有一个对象，内部会有一个引用计数来记录有多少个变量引用这个对象；可变数据类型，允许变量的值发生变化，即如果对变量进行append、+=等这种操作后，只是改变了变量的值，而不会新建一个对象，变量引用的对象的地址也不会变化，不过对于相同的值的不同对象，在内存中则会存在不同的对象，即每个对象都有自己的地址，相当于内存中对于同值的对象保存了多份，这里不存在引用计数，是实实在在的对象。”

## 28、简述Python的深浅拷贝以及应用场景？

网址：https://www.cnblogs.com/lst1010/p/5844705.html

　　我们已经详细了解了变量赋值的过程。对于复杂的数据结构来说，赋值就等于完全共享了资源，一个值的改变会完全被另一个值共享。然而有的时候，我们偏偏需要将一份数据的原始内容保留一份，再去处理数据，这个时候使用赋值就不够明智了。python为这种需求提供了copy模块。提供了两种主要的copy方法，一种是普通的copy，另一种是deepcopy。我们称前者是浅拷贝，后者为深拷贝。

　　深浅拷贝一直是所有编程语言的重要知识点，下面我们就从内存的角度来分析一下两者的区别。

**浅拷贝**

　　首先，我们来了解一下浅拷贝。浅拷贝：不管多么复杂的数据结构，浅拷贝都只会copy一层。

#浅拷贝

a = [[1,2],3,4]

b = a.copy()

b[2] = 5

b[0][0] = 7

print(a,b)

**深拷贝**

　　刚刚我们了解了浅拷贝的意义，但是在写程序的时候，我们就是希望复杂的数据结构之间完全copy一份并且它们之间又没有一毛钱关系，应该怎么办呢？

　　我们引入一个深拷贝的概念，深拷贝——即python的copy模块提供的另一个deepcopy方法。深拷贝会完全复制原变量相关的所有数据，在内存中生成一套完全一样的内容，在这个过程中我们对这两个变量中的一个进行任意修改都不会影响其他变量。

#深拷贝

import copy

a = [[1,2],3,4]

b=copy.deepcopy(a)

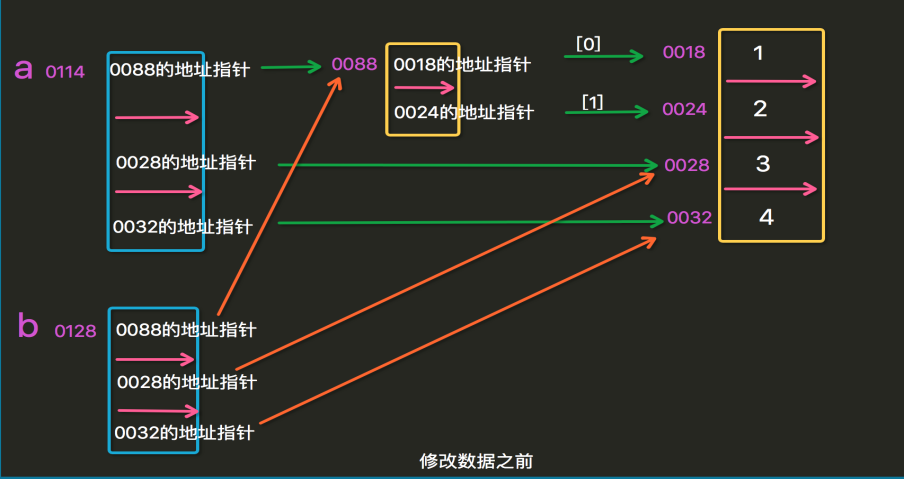
b[1] = 5

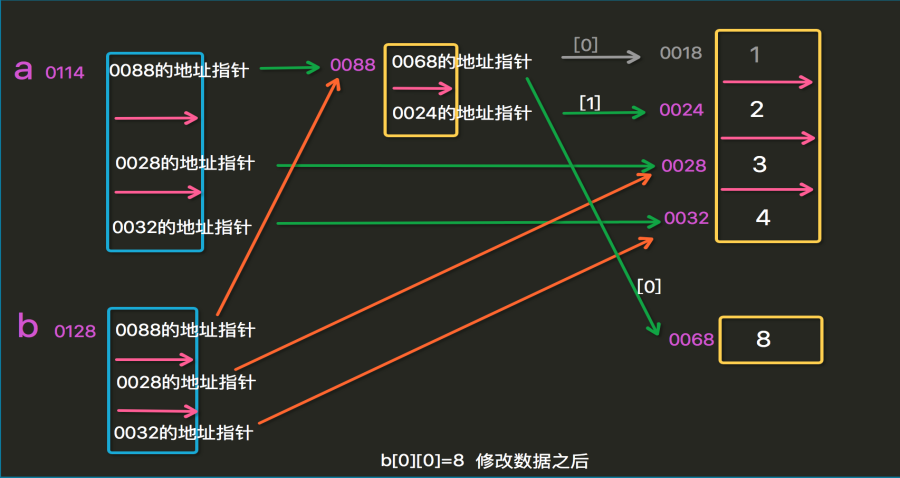
b[0][1] = 7

print(a,b)

深拷贝就是在内存中重新开辟一块空间，不管数据结构多么复杂，只要遇到可能发生改变的数据类型，就重新开辟一块内存空间把内容复制下来，直到最后一层，不再有复杂的数据类型，就保持其原引用。这样，不管数据结构多么的复杂，数据之间的修改都不会相互影响。这就是深拷贝。

深浅拷贝修改前后的关系图





## 29、Python垃圾回收机制和内存池机制？

### 29.1 垃圾回收

网址：https://www.cnblogs.com/George1994/p/7349871.html

Python的GC模块主要运用了引用计数来跟踪和回收垃圾。在引用计数的基础上，还可以通过“标记－清除”解决容器对象可能产生的**循环引用**的问题。通过分代回收以空间换取时间进一步提高垃圾回收的效率。

引用计数

**原理**：当一个对象的引用被创建或者复制时，对象的引用计数加1；当一个对象的引用被销毁时，对象的引用计数减1，当对象的引用计数减少为0时，就意味着对象已经再没有被使用了，可以将其内存释放掉。

**优点**：引用计数有一个很大的优点，即实时性，任何内存，一旦没有指向它的引用，就会被立即回收，而其他的垃圾收集技术必须在某种特殊条件下才能进行无效内存的回收。

**缺点**：但是它也有弱点，引用计数机制所带来的维护引用计数的额外操作与Python运行中所进行的内存分配和释放，引用赋值的次数是成正比的，这显然比其它那些垃圾收集技术所带来的额外操作只是与待回收的内存数量有关的效率要低。同时，引用技术还存在另外一个很大的问题－循环引用，因为对象之间相互引用，每个对象的引用都不会为0，所以这些对象所占用的内存始终都不会被释放掉。如下：

a = []

b = []

a.append(b)

b.append(a)

**print** a

[[[…]]]

**print** b

[[[…]]]

标记清除

标记－清除只关注那些可能会产生循环引用的对象，显然，像是PyIntObject、PyStringObject这些不可变对象是不可能产生循环引用的，因为它们内部不可能持有其它对象的引用。Python中的循环引用总是发生在container对象之间，也就是能够在内部持有其它对象的对象，比如list、dict、class等等。这也使得该方法带来的开销只依赖于container对象的的数量？？？

**原理**：1. 寻找跟对象（root object）的集合作为垃圾检测动作的起点，跟对象也就是一些全局引用和函数栈中的引用，这些引用所指向的对象是不可被删除的；2. 从root object集合出发，沿着root object集合中的每一个引用，如果能够到达某个对象，则说明这个对象是可达的，那么就不会被删除，这个过程就是垃圾检测阶段；3. 当检测阶段结束以后，所有的对象就分成可达和不可达两部分，所有的可达对象都进行保留，其它的不可达对象所占用的内存将会被回收，这就是垃圾回收阶段。（底层采用的是**链表**将这些集合的对象连接在一起）

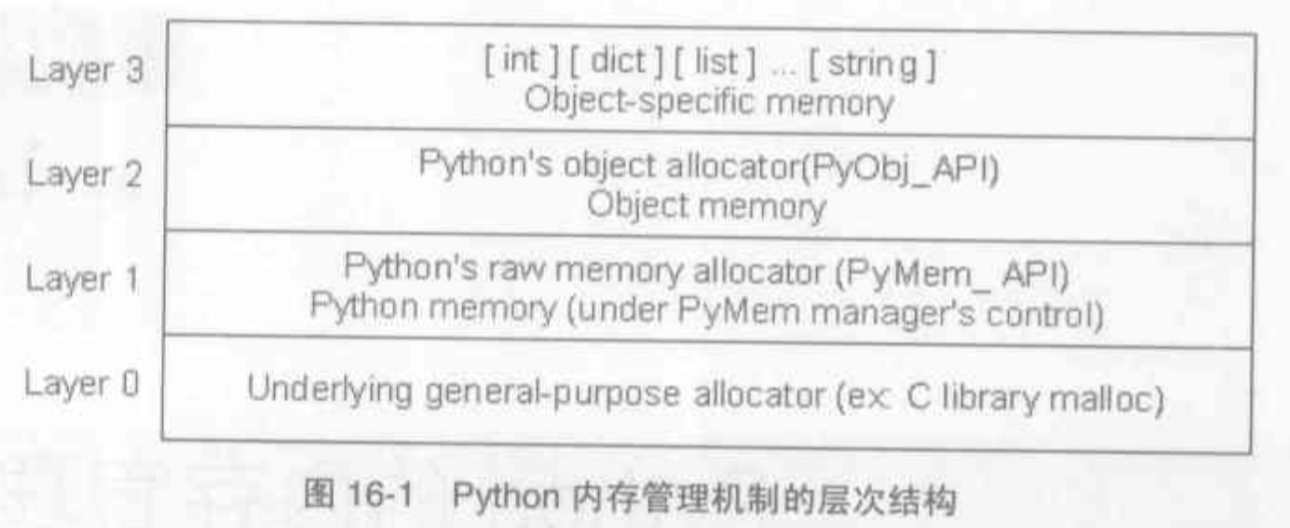
**缺点**：标记和清除的过程效率不高。

分代回收

**原理**：将系统中的所有内存块根据其存活时间划分为不同的集合，每一个集合就成为一个“代”，Python默认定义了三代对象集合，垃圾收集的频率随着“代”的存活时间的增大而减小。也就是说，活得越长的对象，就越不可能是垃圾，就应该减少对它的垃圾收集频率。那么如何来衡量这个存活时间：通常是利用几次垃圾收集动作来衡量，如果一个对象经过的垃圾收集次数越多，可以得出：该对象存活时间就越长

### 29.2 内存管理

Python中的内存管理机制的层次结构提供了4层，其中最底层则是C运行的malloc和free接口，往上的三层才是由Python实现并且维护的。



第一层则是在第0层的基础之上对其提供的接口进行了统一的封装，这是因为虽然不同的操作系统都提供标准定义的内存管理接口，但是对于某些特殊的情况不同的操作系统都不同的行为，比如说调用malloc(0)，有的操作系统会返回NULL，表示内存申请失败；然而有的操作系统会返回一个貌似正常的指针，但是这个指针所指的内存并不是有效的。为了广泛的移植性，Python必须保证相同的语义一定代表相同的运行行为。

在第二层内存管理机制上，Python构建了更高抽象的内存管理策略，比如说一些常用对象，包括整数对象、字符串对象等等。

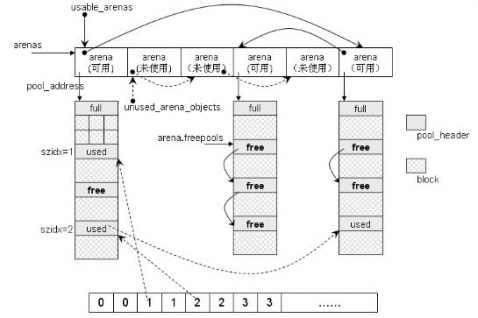
第三层主要是对象缓冲池机制，它基于在第二层的内存池。

### 29.3 内存池

Python为了避免频繁的申请和删除内存所造成系统切换于用户态和核心态的开销，从而引入了**内存池机制**，专门用来管理**小内存**的申请和释放。整个小块内存的内存池可以视为一个层次结构，其一共分为4层，从下之上分别是block、pool、arena和内存池。需要说明的是：block、pool和area都是代码中可以找到的实体，而最顶层的内存池只是一个概念上的东西，表示Python对于整个小块内存分配和释放行为的内存管理机制。

注意，内存大小以256字节为界限，大于则通过malloc进行分配，小于则通过内存池分配。

如下图：



block：最小的内存单元，大小为8的整数倍。有很多种类的block，不同种类的block都有不同的内存大小，申请内存的时候只需要找到适合自身大小的block即可，当然申请的内存也是存在一个上限，如果超过这个上限，则退化到使用最底层的malloc进行申请。

pool：一个pool管理着一堆有固定大小的内存块，其大小通常为一个系统内存页的大小。

arena：多个pool组合成一个arena。

内存池：一个整体的概念。

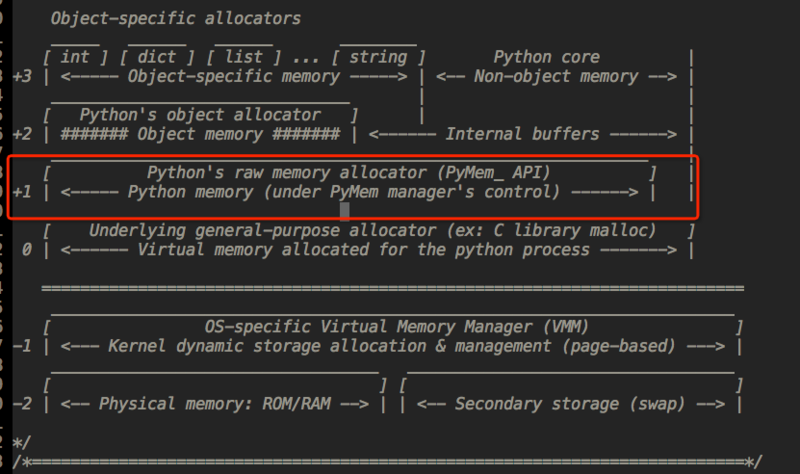
Python的内存机制以金字塔行，-1，-2层主要有操作系统进行操作，

第0层是C中的malloc，free等内存分配和释放函数进行操作；

第1层和第2层是内存池，有Python的接口函数PyMem\_Malloc函数实现，当对象小于256K时有该层直接分配内存；

第3层是最上层，也就是我们对Python对象的直接操作；

在 C 中如果频繁的调用 malloc 与 free 时,是会产生性能问题的.再加上频繁的分配与释放小块的内存会产生内存碎片. Python 在这里主要干的工作有:如果请求分配的内存在1~256字节之间就使用自己的内存管理系统,否则直接使用 malloc.这里还是会调用 malloc 分配内存,但每次会分配一块大小为256k的大块内存.经由内存池登记的内存到最后还是会回收到内存池,并不会调用 C 的 free 释放掉.以便下次使用.对于简单的Python对象，例如数值、字符串，元组（tuple不允许被更改)采用的是复制的方式(深拷贝?)，也就是说当将另一个变量B赋值给变量A时，虽然A和B的内存空间仍然相同，但当A的值发生变化时，会重新给A分配空间，A和B的地址变得不再相同



30、stack：

1. Stack() 创建一个新的空栈

2. push(item) 添加一个新的元素item到栈顶

3. pop() 弹出栈顶元素

4. peek() 返回栈顶元素

5. is\_empty() 判断栈是否为空

6. size() 返回栈的元素个数

class Stack(object):

"""栈"""

def \_\_init\_\_(self):

self.items = []

def is\_empty(self):

"""判断是否为空"""

return self.items == []

def push(self, item):

"""加入元素"""

self.items.append(item)

def pop(self):

"""弹出元素"""

return self.items.pop()

def peek(self):

"""返回栈顶元素"""

return self.items[len(self.items)-1]

def size(self):

"""返回栈的大小"""

return len(self.items)

if \_\_name\_\_ == "\_\_main\_\_":

stack = Stack()

stack.push("hello")

stack.push("world")

stack.push("lcg")

print stack.size()

print stack.peek()

print stack.pop()

print stack.pop()

print stack.pop()

## 31、functools模块

1. 是否使⽤过functools中的函数？其作⽤是什么？

[复制代码](javascript:void(0);)

# 用于修复装饰器

import functools

def deco(func):

@functools.wraps(func) # 加在最内层函数正上方

def wrapper(\*args, \*\*kwargs):

return func(\*args, \*\*kwargs)

return wrapper

@deco

def index():

'''哈哈哈哈'''

x = 10

print('from index')

print(index.\_\_name\_\_)

print(index.\_\_doc\_\_)

# 加@functools.wraps

# index

# 哈哈哈哈

# 不加@functools.wraps

# wrapper

# None

## ****32、reduce****

http://www.cnblogs.com/clement-jiao/p/8719323.html#autoid-8-0-0

[复制代码](javascript:void(0);)

在Python 3里,reduce()函数已经被从全局名字空间里移除了,它现在被放置在fucntools模块里 用的话要 先引入：

>>> from functools import reduce

reduce函数，reduce函数会对参数序列中元素进行累积。

reduce函数的定义：

reduce(function, sequence[, initial]) -> value

function参数是一个有两个参数的函数，reduce依次从sequence中取一个元素，和上一次调用function的结果做参数再次调用function。

第一次调用function时，如果提供initial参数，会以sequence中的第一个元素和initial作为参数调用function，否则会以序列sequence中的前两个元素做参数调用function。

reduce(lambda x, y: x + y, [2, 3, 4, 5, 6], 1)

结果为21( (((((1+2)+3)+4)+5)+6) )

reduce(lambda x, y: x + y, [2, 3, 4, 5, 6])

结果为20

## 33、filter、map、reduce的作用

map：遍历序列，对序列中每个元素进行操作，最终获取新的序列。

- 每个元素增加100：

- li = [11, 22, 33]

- new\_list = map(lambda a: a + 100, li)

- 两个列表对应元素相加

- li = [11, 22, 33]

- sl = [1, 2, 3, 4]

- new\_list = map(lambda a, b: a + b, li, sl)

filter：对于序列中的元素进行筛选，最终获取符合条件的序列。

- 获取列表中大于12的所有元素集合

- li = [11, 22, 33]

- new\_list = filter(lambda arg: arg > 22, li)

- # filter第一个参数为空，将获取原来序列

reduce：对于序列内所有元素进行累计操作。

- 获取序列所有元素的和

- li = [11, 22, 33]

- result = reduce(lambda arg1, arg2: arg1 + arg2, li)

- # reduce的第一个参数，函数必须要有两个参数

- # reduce的第二个参数，要循环的序列

- # reduce的第三个参数，初始值

## 34、一行代码实现9\*9乘法表

print('\n'.join([' '.join(['%s\*%s=%-2s' % (j, i, i \* j) for j in range(1, i + 1)]) for i in range(1, 10)]))

## 35、\_\_new\_\_是一个静态方法,而\_\_init\_\_是一个实例方法.

1. \_\_new\_\_方法会返回一个创建的实例,而\_\_init\_\_什么都不返回.
2. 只有在\_\_new\_\_返回一个cls的实例时后面的\_\_init\_\_才能被调用.
3. 当创建一个新实例时调用\_\_new\_\_,初始化一个实例时用\_\_init\_\_.
4. 继承自object的新式类才有\_\_new\_\_
5. \_\_new\_\_至少要有一个参数cls，代表当前类，此参数在实例化时由Python解释器自动识别
6. \_\_new\_\_必须要有返回值，返回实例化出来的实例，这点在自己实现\_\_new\_\_时要特别注意，  
   可以return父类（通过super(当前类名, cls)）\_\_new\_\_出来的实例，或者直接是object的\_\_new\_\_出来的实例
7. \_\_init\_\_有一个参数self，就是这个\_\_new\_\_返回的实例，\_\_init\_\_在\_\_new\_\_的基础上可以完成一些其它初始化的动作，\_\_init\_\_不需要返回值
8. 如果\_\_new\_\_创建的是当前类的实例，会自动调用\_\_init\_\_函数，通过return语句里面调用的\_\_new\_\_函数的第一个参数是cls来保证是当前类实例，
9. 如果是其他类的类名，；那么实际创建返回的就是其他类的实例，其实就不会调用当前类的\_\_init\_\_函数，也不会调用其他类的\_\_init\_\_函数。

## 36、常用的模块

1. copy ： 用于深浅拷贝
2. os ：与操作系统交互的一个接口 比如用来处理文件和目录
3. random ：随机模块
4. sys：负责程序与python解释器的交互，提供了一系列的函数和变量，用于操控python的运行时环境。
5. json：序列化
6. re ： 正则模块
7. logging ： 日志模块
8. requests： 爬取数据
9. timeit ：
10. subprocess：与OS模块相同，区别与异步提交命令不等待输出。[**与OS模块的区别**](https://blog.csdn.net/bcfdsagbfcisbg/article/details/78134172)

## 37、你知道哪些OS模块的方法

os.remove(‘path/filename’) # 删除文件

os.rename(oldname, newname) # 重命名文件

os.walk() # 生成目录树下的所有文件名

os.chdir('dirname') # 改变目录

os.getcwd() # 取得当前工作目录

os.path.getsize() # 返回文件大小

## 38、如何使⽤python删除⼀个⽂件

import os

file = r'D:\test.txt'

if os.path.exists(file):

os.remove(file)

print('delete success')

else:

print('no such file:%s' % file)

## 39、- re.match 和re.resercah

re.match只匹配字符串的开始，如果字符串开始不符合正则表达式，则匹配失败，函数返回None；

- re.search：匹配整个字符串，直到找到一个匹配。

应用示例：

[复制代码](javascript:void(0);)

import re

s = "fnfffidvvgf"

m = re.match("fi",s)

print(m)  
>>> #None

s = re.search("fi",s).group()

print(s)  
>>> #fi

## 40、贪婪匹配与非贪婪匹配

- 匹配0次或多次 <.\*>

- 非贪婪匹配：匹配0次或1次 <.?>

## 41、[你知道几种设计模式](https://blog.csdn.net/jason0539/article/details/44956775)

设计模式分为三大类：

创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。

结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式。

行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。

其实还有两类：并发型模式和线程池模式。

## 42、如何实现 “1,2,3” 变成 [‘1’,’2’,’3’] ?

 list("1,2,3".split(','))

## 43、如何实现[‘1’,’2’,’3’]变成[1,2,3] ?

>>>a = ['1', '2', '3']

>>>b = [int(i) for i in a]

>>>b

## 44、比较： a = [1,2,3] 和 b = [(1),(2),(3) ] 以及 b = [(1,),(2,),(3,) ] 的区别？

## 45、如何用一行代码生成[1,4,9,16,25,36,49,64,81,100] ?

print ( [i\*i **for** i **in** range(1,11)])

## 46、一行代码实现删除列表中重复的值 ?

print (list(set(a)))

## 47、如何在函数中设置一个全局变量 ?

x = 1

def func():

global x

print x

x = 2

print x

print x

func()

print x

## 48、logging模块的作用？以及应用场景？

Python的logging模块提供了通用的日志系统，可以方便第三方模块或者是应用使用。这个模块提供不同的日志级别，并可以采用不同的方式记录日志，比如文件，HTTP GET/POST，SMTP，Socket等，甚至可以自己实现具体的日志记录方式。

模块提供logger，handler，filter，formatter。

1.  logger：提供日志接口，供应用代码使用。logger最长用的操作有两类：配置和发送日志消息。可以通过logging.getLogger(name)获取logger对象，如果不指定name则返回root对象，多次使用相同的name调用getLogger方法返回同一个logger对象。

2.  handler：将日志记录（logrecord）发送到合适的目的地（destination），比如文件，socket等。一个logger对象可以通过addHandler方法添加0到多个handler，每个handler又可以定义不同日志级别，以实现日志分级过滤显示。

3.   filter：提供一种优雅的方式决定一个日志记录是否发送到handler。

4.   formatter：指定日志记录输出的具体格式。formatter的构造方法需要两个参数：消息的格式字符串和日期字符串，这两个参数都是可选的。

备注：

一般在大型的项目中，由于有多个文件需要输出log，所以需要重写logging模块，达到自定义文件输出格式，输出路径等一系列统一的要求，能省去多次重复定义logging。

49、请用代码简答实现stack 。

## 50、常用字符串格式化哪几种？

1.占位符%

%d 表示那个位置是整数；%f 表示浮点数；%s 表示字符串

- print('Hello,%s' % 'Python')

- print('Hello,%d%s%.2f' % (666, 'Python', 9.99)) # 打印：Hello,666Python10.00

2.format

- print('{k} is {v}'.format(k='python', v='easy')) # 通过关键字

- print('{0} is {1}'.format('python', 'easy')) # 通过关键字

[复制代码](javascript:void(0);)

      刚入门python的同学，特别是，没有系统的学习过python，而是学过别的语言，直接上手python的同学，怕是还不是很了解python强大的字符串格式化方法

**1.最方便的**

**print** 'hello %s and %s' % ('df', 'another df')

但是，有时候，我们有很多的参数要进行格式化，这个时候，一个一个一一对应就有点麻烦了，于是就有了第二种，字典形式的。上面那种是tuple形式的。

2.最好用的

**print** 'hello %(first)s and %(second)s' % {'first': 'df', 'second': 'another df'}

这种字典形式的字符串格式化方法，有一个最大的好处就是，字典这个东西可以和json文件相互转换，所以，当配置文件使用字符串设置的时候，就显得相当方便。

**3.最先进的**

**print** 'hello {first} and {second}'.format(first='df', second='another df')

## 51、简述 生成器、迭代器、可迭代对象 以及应用场景？

网址：<https://www.cnblogs.com/yyds/p/6281453.html>

**一 三元表达式**

name=input('姓名>>: ')

res='mm' if name == 'hahah' else 'NB'

print(res)

**二 列表推导式**

#1、示例

egg\_list=[]

for i in range(10):

egg\_list.append('鸡蛋%s' %i)

egg\_list=['鸡蛋%s' %i for i in range(10)]

#2、语法

[expression for item1 in iterable1 if condition1

for item2 in iterable2 if condition2

...

for itemN in iterableN if conditionN

]

类似于

res=[]

for item1 in iterable1:

if condition1:

for item2 in iterable2:

if condition2

...

for itemN in iterableN:

if conditionN:

res.append(expression)

#3、优点：方便，改变了编程习惯，可称之为声明式编程

**三 生成器表达式**

#1、把列表推导式的[]换成()就是生成器表达式

#2、示例：生一筐鸡蛋变成给你一只老母鸡，用的时候就下蛋，这也是生成器的特性

>>> chicken=('鸡蛋%s' %i for i in range(5))

>>> chicken

<generator object <genexpr> at 0x10143f200>

>>> next(chicken)

'鸡蛋0'

>>> list(chicken) #因chicken可迭代，因而可以转成列表

['鸡蛋1', '鸡蛋2', '鸡蛋3', '鸡蛋4',]

#3、优点：省内存，一次只产生一个值在内存中

把一个列表[]改成()就创建了一个生成器：generator，generator保存的是算法。

可以用于for循环的数据类型：

1、集合类型：list tuple dict set str

2、generator

可以直接作用于for循环的对象统称为：可迭代对象（Iterable）

from collections import Iterable

print(isinstance([],Iterable))

print(isinstance({},Iterable))

print(isinstance((x for x in range(10)),Iterable))

print(isinstance(100, Iterable))

结果：

True

True

True

False

可以被next调用并不断返回下一个值的对象称为：迭代器（Iterator）

from collections import Iterator

print(isinstance([],Iterator))

print(isinstance({},Iterator))

print(isinstance((x for x in range(10)),Iterator))

print(isinstance(100, Iterator))

结果：

from collections import Iterator

print(isinstance([],Iterator))

print(isinstance({},Iterator))

print(isinstance((x for x in range(10)),Iterator))

print(isinstance(100, Iterator))

把list、dict、str等Iterable变成Iterator可以使用iter()函数：

复制代码

#!/usr/bin/env python

# -\*- coding: utf-8 -\*-

from collections import Iterator

print(isinstance(iter([]),Iterator))

print(isinstance(iter({}),Iterator))

print(isinstance((x for x in range(10)),Iterator))

print(isinstance(iter('abc'), Iterator))

复制代码

显示结果：

True

True

True

True

* 语法糖的概念
* 列表生成式
* 生成器（Generator）
* 可迭代对象（Iterable）
* 迭代器（Iterator）
* Iterable、Iterator与Generator之间的关系

**一、语法糖的概念**

“语法糖”，从字面上看应该是一种语法。“糖”，可以理解为简单、简洁。其实我们也已经意识到，没有这些被称为“语法糖”的语法，我们也能实现相应的功能，而 ***“语法糖”使我们可以更加简洁、快速的实现这些功能。*** 只是Python解释器会把这些特定格式的语法翻译成原本那样复杂的代码逻辑而已，没有什么太高深的东西。

到目前为止，我们使用和介绍过的语法糖有：

* ***if...else 三元表达式：*** 可以简化分支判断语句，如 x = y.lower() if isinstance(y, str) else y
* ***with语句：*** 用于文件操作时，可以帮我们自动关闭文件对象，使代码变得简洁；
* ***装饰器：*** 可以在不改变函数代码及函数调用方式的前提下，为函数增加增强性功能；

这里会再介绍两个：

* ***列表生成式：*** 用于生成一个新的列表
* ***生成器：*** 用于“惰性”地生成一个无限序列

**二、列表生成式**

顾名思义，列表生成式就是一个用来生成列表的特定语法形式的表达式。

**1. 语法格式：**

**基础语法格式**

[exp for iter\_var in iterable]

***工作过程：***

* 迭代iterable中的每个元素；
* 每次迭代都先把结果赋值给iter\_var，然后通过exp得到一个新的计算值；
* 最后把所有通过exp得到的计算值以一个新列表的形式返回。

相当于这样的过程：

L = []

for iter\_var in iterable:

L.append(exp)

**带过滤功能语法格式**

[exp for iter\_var in iterable if\_exp]

***工作过程：***

* 迭代iterable中的每个元素，每次迭代都先判断if\_exp表达式结果为真，如果为真则进行下一步，如果为假则进行下一次迭代；
* 把迭代结果赋值给iter\_var，然后通过exp得到一个新的计算值；
* 最后把所有通过exp得到的计算值以一个新列表的形式返回。

相当于这样的过程：

L = []

for iter\_var in iterable:

if\_exp:

L.append(exp)

**循环嵌套语法格式**

[exp for iter\_var\_A in iterable\_A for iter\_var\_B in iterable\_B]

***工作过程：***  
每迭代iterable\_A中的一个元素，就把ierable\_B中的所有元素都迭代一遍。

相当于这样的过程：

L = []

for iter\_var\_A in iterable\_A:

for iter\_var\_B in iterable\_B:

L.append(exp)

**2. 应用场景**

其实列表生成式也是Python中的一种“语法糖”，也就是说列表生成式应该是Python提供的一种生成列表的简洁形式，应用列表生成式可以快速生成一个新的list。它最主要的应用场景是：***根据已存在的可迭代对象推导出一个新的list。***

**3. 使用实例**

我们可以对几个生成列表的要求分别通过“不使用列表生成式”和“使用列表生成式”来实现，然后做个对比总结。

**实例1：生成一个从3到10的数字列表**

# 不使用列表生成式实现

list1 = list(range(3, 11))

# 使用列表生成式实现

list2 = [x for x in range(3, 11)]

**实例2：生成一个2n+1的数字列表，n为从3到11的数字**

# 不使用列表生成式实现

list3 = []

for n in range(3, 11):

list3.append(2\*n + 1)

# 使用列表生成式实现

list4 = [2\*n + 1 for n in range(3, 11)]

**实例3：过滤出一个指定的数字列表中值大于20的元素**

L = [3， 7， 11， 14，19， 33， 26， 57， 99]

# 不使用列表生成式实现

list5 = []

for x in L:

if x < 20:

list5.append(x)

# 使用列表生成式实现

list6 = [x for x in L if x > 20]

**实例4：计算两个集合的全排列，并将结果作为保存至一个新的列表中**

L1 = ['香蕉', '苹果', '橙子']

L2 = ['可乐', '牛奶']

# 不使用列表生成式实现

list7 = []

for x in L1:

for y in L2:

list7.append((x, y))

# 使用列表生成式实现

list8 = [(x, y) for x in L1 for y in L2]

**实例5：将一个字典转换成由一组元组组成的列表，元组的格式为(key, value)**

D = {'Tom': 15, 'Jerry': 18, 'Peter': 13}

# 不使用列表生成式实现

list9 = []

for k, v in D.items():

list9.append((k, v))

# 使用列表生成式实现

list10 = [(k, v) for k, v in D.items()]

可见，使用列表生成式确实要方便、简洁很多，使用一行代码就搞定了。

**4. 列表生成式与map()、filter()等高阶函数功能对比**

我觉得，大家应该已经发现这里说的列表生成式的功能与之前 [这篇文章](http://www.cnblogs.com/yyds/p/6263957.html) 中讲到的map()和filter()高阶函数的功能很像，比如下面两个例子：

**实例1：把一个列表中所有的字符串转换成小写，非字符串元素原样保留**

L = ['TOM', 'Peter', 10, 'Jerry']

# 用列表生成式实现

list1 = [x.lower() if isinstance(x, str) else x for x in L]

# 用map()函数实现

list2 = list(map(lambda x: x.lower() if isinstance(x, str) else x, L))

**实例2：把一个列表中所有的字符串转换成小写，非字符串元素移除**

L = ['TOM', 'Peter', 10, 'Jerry']

# 用列表生成式实现

list3 = [x.lower() for x in L if isinstance(x, str)]

# 用map()和filter()函数实现

list4 = list(map(lambda x: x.lower(), filter(lambda x: isinstance(x, str), L)))

对于大部分需求来讲，使用列表生成式和使用高阶函数都能实现。但是map()和filter()等一些高阶函数在Python3中的返回值类型变成了Iteraotr（迭代器）对象（在Python2中的返回值类型为list），这对于那些元素数量很大或无限的可迭代对象来说显然是更合适的，因为可以避免不必要的内存空间浪费。关于迭代器的概念，下面会单独进行说明。

**三、生成器（Generator）**

从名字上来看，生成器应该是用来生成数据的。

**1. 生成器的作用**

按照某种算法不断生成新的数据，直到满足某一个指定的条件结束。

**2. 生成器的构造方式**

构造生成器的两种方式：

* 使用类似列表生成式的方式生成 (2\*n + 1 for n in range(3, 11))
* 使用包含yield的函数来生成

如果计算过程比较简单，可以直接把列表生成式改成generator；但是，如果计算过程比较复杂，就只能通过包含yield的函数来构造generator。

***说明：*** Python 3.3之前的版本中，不允许迭代函数法中包含return语句。

**3. 生成器构造实例**

# 使用类似列表生成式的方式构造生成器

g1 = (2\*n + 1 for n in range(3, 6))

# 使用包含yield的函数构造生成器

def my\_range(start, end):

for n in range(start, end):

yield 2\*n + 1

g2 = my\_range(3, 6)

print(type(g1))

print(type(g2))

输出结果：

<class 'generator'>

<class 'generator'>

**4. 生成器的执行过程与特性**

**生成器的执行过程：**

在执行过程中，遇到yield关键字就会中断执行，下次调用则继续从上次中断的位置继续执行。

**生成器的特性：**

* 只有在调用时才会生成相应的数据
* 只记录当前的位置
* 只能next，不能prev

**5. 生成器的调用方式**

要调用生成器产生新的元素，有两种方式：

* 调用内置的next()方法
* 使用循环对生成器对象进行遍历（推荐）
* 调用生成器对象的send()方法

**实例1：使用next()方法遍历生成器**

print(next(g1))

print(next(g1))

print(next(g1))

print(next(g1))

输出结果：

7

9

11

Traceback (most recent call last):

File "\*\*\*/generator.py", line 26, in <module>

print(next(g1))

StopIteration

print(next(g2))

print(next(g2))

print(next(g2))

print(next(g2))

输出结果：

7

9

11

Traceback (most recent call last):

File "\*\*\*/generator.py", line 31, in <module>

print(next(g2))

StopIteration

可见，使用next()方法遍历生成器时，最后是以抛出一个StopIeration异常终止。

**实例2：使用循环遍历生成器**

for x in g1:

print(x)

for x in g2:

print(x)

两个循环的输出结果是一样的：

7

9

11

可见，使用循环遍历生成器时比较简洁，且最后不会抛出一个StopIeration异常。因此使用循环的方式遍历生成器的方式才是被推荐的。

需要说明的是：如果生成器函数有返回值，要获取该返回值的话，只能通过在一个while循环中不断的next()，最后通过捕获StopIteration异常

**实例3：调用生成器对象的send()方法**

def my\_range(start, end):

for n in range(start, end):

ret = yield 2\*n + 1

print(ret)

g3 = my\_range(3, 6)

print(g3.send(None))

print(g3.send('hello01'))

print(g3.send('hello02'))

输出结果：

7

hello01

9

hello02

11

print(next(g3))

print(next(g3))

print(next(g3))

输出结果：

7

None

9

None

11

结论：

* next()会调用yield，但不给它传值
* send()会调用yield，也会给它传值（该值将成为当前yield表达式的结果值）

需要注意的是：第一次调用生成器的send()方法时，参数只能为None，否则会抛出异常。当然也可以在调用send()方法之前先调用一次next()方法，目的是让生成器先进入yield表达式。

**6. 生成器与列表生成式对比**

***既然通过列表生成式就可以直接创建一个新的list，那么为什么还要有生成器存在呢？***

因为列表生成式是直接创建一个新的list，它会一次性地把所有数据都存放到内存中，这会存在以下几个问题：

* 内存容量有限，因此列表容量是有限的；
* 当列表中的数据量很大时，会占用大量的内存空间，如果我们仅仅需要访问前面有限个元素时，就会造成内存资源的极大浪费；
* 当数据量很大时，列表生成式的返回时间会很慢；

而生成器中的元素是按照指定的算法推算出来的，只有调用时才生成相应的数据。这样就不必一次性地把所有数据都生成，从而节省了大量的内存空间，这使得其生成的元素个数几乎是没有限制的，并且操作的返回时间也是非常快速的（仅仅是创建一个变量而已）。

我们可以做个试验：对比一下生成一个1000万个数字的列表，分别看下用列表生成式和生成器时返回结果的时间和所占内存空间的大小：

import time

import sys

time\_start = time.time()

g1 = [x for x in range(10000000)]

time\_end = time.time()

print('列表生成式返回结果花费的时间： %s' % (time\_end - time\_start))

print('列表生成式返回结果占用内存大小：%s' % sys.getsizeof(g1))

def my\_range(start, end):

for x in range(start, end):

yield x

time\_start = time.time()

g2 = my\_range(0, 10000000)

time\_end = time.time()

print('生成器返回结果花费的时间： %s' % (time\_end - time\_start))

print('生成器返回结果占用内存大小：%s' % sys.getsizeof(g2))

输出结果：

列表生成式返回结果花费的时间： 0.8215489387512207

列表生成式返回结果占用内存大小：81528056

生成器返回结果花费的时间： 0.0

生成器返回结果占用内存大小：88

可见，生成器返回结果的时间几乎为0，结果所占内存空间的大小相对于列表生成器来说也要小的多。

**四、可迭代对象（Iterable）**

我们经常在Python的文档中看到“Iterable”这个此，它的意思是“可迭代对象”。那么什么是可迭代对象呢？  
***可直接用于for循环的对象统称为可迭代对象（Iterable）。***

目前我们已经知道的可迭代（可用于for循环）的数据类型有：

* 集合数据类型：如list、tuple、dict、set、str等
* 生成器（Generator）

可以使用isinstance()来判断一个对象是否是Iterable对象：

from collections import Iterable

print(isinstance([], Iterable))

**五、迭代器（Iterator）**

**1. 迭代器的定义**

***可以被next()函数调用并不断返回下一个值的对象称为迭代器：Iterator***。

很明显上面讲的生成器也是迭代器。当然，我们可以使用isinstance()来验证一下：

from collections import Iterator

print(isinstance((x for x in range(5)), Iterator))

输出结果为：True

**2. 对迭代器的理解**

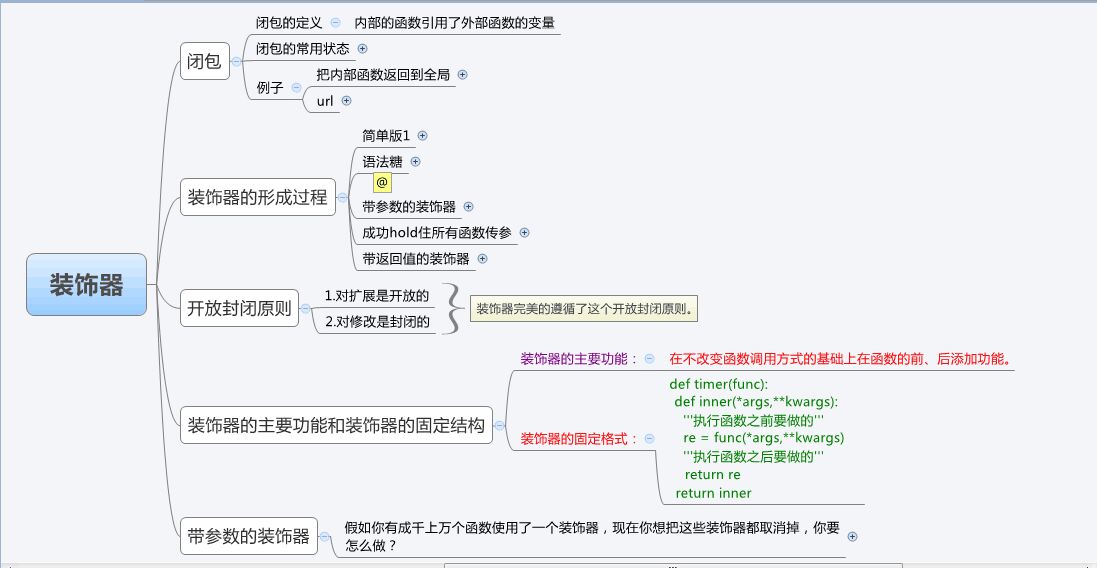
实际上，Python中的Iterator对象表示的是一个数据流，Iterator可以被next()函数调用被不断返回下一个数据，直到没有数据可以返回时抛出StopIteration异常错误。可以把这个数据流看做一个有序序列，但我们无法提前知道这个序列的长度。同时，Iterator的计算是惰性的，只有通过next()函数时才会计算并返回下一个数据。（此段内容来自 [这里](http://www.liaoxuefeng.com/wiki/0014316089557264a6b348958f449949df42a6d3a2e542c000/00143178254193589df9c612d2449618ea460e7a672a366000)）

生成器也是这样的，因为生成器也是迭代器。

**六、Iterable、Iterator与Generator之间的关系**

* ***生成器对象既是可迭代对象，也是迭代器：*** 我们已经知道，生成器不但可以作用与for循环，还可以被next()函数不断调用并返回下一个值，直到最后抛出StopIteration错误表示无法继续返回下一个值了。也就是说，生成器同时满足可迭代对象和迭代器的定义；
* ***迭代器对象一定是可迭代对象，反之则不一定：*** 例如list、dict、str等集合数据类型是可迭代对象，但不是迭代器，但是它们可以通过iter()函数生成一个迭代器对象。

也就是说：迭代器、生成器和可迭代对象都可以用for循环去迭代，生成器和迭代器还可以被next()方函数调用并返回下一个值。



## 52、用Python实现一个二分查找的函数。

二分查找

二分查找又叫折半查找，二分查找应该属于减治技术的成功应用。所谓减治法，就是将原问题分解成若干个子问题后，利用了规模为n的原问题的解与较小规模（通常是n/2）的子问题的解之间的关系。

二分查找利用了记录按关键码有序的特点，其基本思想为：在有序表中，取中间记录作为比较对象，若给定值与中间记录的关键码相等，则查找成功；若给定值小于中间记录的关键码，则在中间记录的左半边继续查找；若给定值大于中间记录的关键码，则在中间记录右半边区继续查找。不断重复上述过程，直到查找成功，或所查找的区域无记录，查找失败。

二分查找的时间复杂度是O(log(n))，最坏情况下的时间复杂度是O(n)。

下图是二分查找的减治思想：

二分查找

如果k< rmid查找这边 如果k> rmid查找这边

例如：

在有序列表list1中[1, 3, 8, 12, 23, 31, 37, 42, 48, 58]中查找值为8的记录的。

伪代码：

1.low = 0; high = len(list1]-1 #设置初识查找区间

2.测试查找区间[low, high]是否存在，若不存在，则查找失败；否则

3.取中间mid=(low + high)/2；比较k与list1[mid]，有以下三种情况：

3.1 若k<r[mid]，则high=mid-1；查找在左半区进行，转2；

3.2 若k>r[mid]，则low=mid+1；查找在右半边区进行，转2；

3.3 若k=r[mid]，则查找成功，返回记录在表中位置mid；

Python实现二分查找算法，代码如下：

#!/usr/bin/python

#coding=utf-8

#自定义函数，实现二分查找，并返回查找结果

def binary\_search(find, list1) :

low = 0

high = len(list1)

while low <= high :

mid = (low + high) / 2

if list1[mid] == find :

return mid

#左半边

elif list1[mid] > find :

high = mid -1

#右半边

else :

low = mid + 1

#未找到返回-1

return -1

list1 = [1,2,3,7,8,9,10,5]

#进行二分查找算法前必须保证要查找的序列时有序的，这里假设是升序列表

list1.sort()

print "原有序列表为:",list1

try :

find = int(raw\_input("请输入要查找的数："))

except :

print "请输入正整数！"

exit()

result = binary\_search(find, list1)

if result != -1 :

print "要找的元素%d的序号为：%d" %(find,result)

else :

print "未找到！"

## 53、谈谈你对闭包的理解？

　闭包是内层函数对层函数局部变量的引用。简单来说，闭包就是内部包含函数的函数。

　闭包的好处：如果python检测到闭包，它有一个机制，局部作用域不会随着函数的结束而结束。可以在局部作用域添加缓存机制，使得对于计算量较大时能够提高效率。闭包也是装饰器的前提

在函数编程中经常用到闭包。闭包是什么，它是怎么产生的及用来解决什么问题呢。给出字面的定义先：**闭包是由函数及其相关的引用环境组合而成的实体(即：闭包=函数+引用环境)(想想Erlang的外层函数传入一个参数a, 内层函数依旧传入一个参数b, 内层函数使用a和b, 最后返回内层函数)**。这个从字面上很难理解，特别对于一直使用命令式语言进行编程的程序员们。本文将结合实例代码进行解释。  
函数是什么  
地球人都知道：函数只是一段可执行代码，编译后就“固化”了，每个函数在内存中只有一份实例，得到函数的入口点便可以执行函数了。在函数式编程语言中，函 数是一等公民（First class value：第一类对象，我们不需要像命令式语言中那样借助函数指针，委托操作函数），函数可以作为另一个函数的参数或返回值，可以赋给一个变量。函数可 以嵌套定义，即在一个函数内部可以定义另一个函数，有了嵌套函数这种结构，便会产生闭包问题。如：

python中的闭包从表现形式上定义（解释）为：如果在一个内部函数里，对在外部作用域（但不是在全局作用域）的变量进行引用，那么内部函数就被认为是闭包(closure).这个定义是相对直白的，好理解的，不像其他定义那样学究味道十足（那些学究味道重的解释，在对一个名词的解释过程中又充满了一堆让人抓狂的其他陌生名词，不适合初学者）。下面举一个简单的例子来说明。

>>>def addx(x):

>>> def adder(y): return x + y

>>> return adder

>>> c = addx(8)

>>> type(c)

<type 'function'>

>>> c.\_\_name\_\_

'adder'

>>> c(10)

结合这段简单的代码和定义来说明闭包：

如果在一个内部函数里：adder(y)就是这个内部函数，

对在外部作用域（但不是在全局作用域）的变量进行引用：x就是被引用的变量，x在外部作用域addx里面，但不在全局作用域里，

则这个内部函数adder就是一个闭包。

再稍微讲究一点的解释是，闭包=函数块+定义函数时的环境，adder就是函数块，x就是环境，当然这个环境可以有很多，不止一个简单的x。

按照命令式语言的规则，ExFunc函数只是返回了内嵌函数InsFunc的地址，在执行InsFunc函数时将会由于在其作用域内找不到sum变量而出 错。而在函数式语言中，当内嵌函数体内引用到体外的变量时，将会把定义时涉及到的引用环境和函数体打包成一个整体（**闭包**）返回。现在给出引用环境的定义就 容易理解了：引用环境是指在程序执行中的某个点所有处于活跃状态的约束（一个变量的名字和其所代表的对象之间的联系）所组成的集合。闭包的使用和正常的函 数调用没有区别。

由于闭包把函数和运行时的引用环境打包成为一个新的整体，所以就解决了函数编程中的嵌套所引发的问题。如上述代码段中，当每次调用ExFunc函数 时都将返回一个新的闭包实例，这些实例之间是隔离的，分别包含调用时不同的引用环境现场。不同于函数，闭包在运行时可以有多个实例，不同的引用环境和相同 的函数组合可以产生不同的实例。

## 54、os和sys模块的作用？

## 55、如何生成一个随机数？

# -\*- coding: UTF-8 -\*- # Filename : test.py # author by : www.runoob.com # 生成 0 ~ 9 之间的随机数 # 导入 random(随机数) 模块 import random print(random.randint(0,9))

假设你对在Python生成随机数与random模块中最经常使用的几个函数的关系与不懂之处。以下的文章就是对Python生成随机数与random模块中最经常使用的几个函数的关系，希望你会有所收获，以下就是这篇文章的介绍。

random.random()用于生成

用于生成一个指定范围内的随机符点数，两个參数当中一个是上限。一个是下限。

假设a > b，则生成随机数

1

n: a <= n <=

b。

假设 a <b， 则 b <= n <= a。

1

2

3

4

5

6

print random.uniform(10,

20)

print random.uniform(20,

10)

#----

#18.7356606526

#12.5798298022

random.randint

用于生成一个指定范围内的整数。当中參数a是下限，參数b是上限。Python生成随机数

1

2

3

print random.randint(12,

20) #生成的随机数n: 12 <= n <= 20

print random.randint(20,

20) #结果永远是20

#print random.randint(20, 10) #该语句是错误的。

下限必须小于上限。

random.randrange

从指定范围内。按指定基数递增的集合中 ，这篇文章就是对python生成随机数的应用程序的部分介绍。

随机整数：

>>> import random

>>> random.randint(0,99)

21

随机选取0到100间的偶数：

>>> import random

>>> random.randrange(0, 101, 2)

42

## 56、如何使用python删除一个文件？

python可以通过os包对文件进行操作。以下代码分别实现一文件夹下所有文件名的读取和文件删除操作

**import** os

#读取path目录下的文件名，返回文件名list列表

**def** readFileName(path):

    lists = []

**for** root,dirs,files **in** os.walk(path):

**for** file **in** files:

            lists.append(os.path.join(root,file))

**return** lists

#删除路径为filepath的文件

**def** delFile(filepath):

    os.remove(filepath)

**print** "ok"

## 57、谈谈你对面向对象的理解？

把数据和功能结合起来，用称为对象的东西包裹起来组织程序的方法。这种方法称为，面向对象的编程理念  
对象可以使用普通的属于对象的变量存储数据。属于一个对象或类的变量被称为域。对象也可以使用属于类的函数来具有功能。这样的函数  
被称为类的方法。这些术语帮助我们把它们与孤立的函数和变量区分开来。域和方法可以合称为类的属性。  
域有两种类型-----属于每个实例/类的对象或属于类本身。它们分别被称为实例变量和类变量。  
类使用class关键字创建。类的域和方法被列在一个缩进块中。  
类的方法与普通的函数只有一个特别的区别-----它们必须有一个额外的第一个参数名称，但是在调用这个方法的时候你不必  
为这个参数赋值，python会提供这个值。这个特别的变量指向对象本身，按照惯例它的名称是self。  
  
你一定很奇怪python如何给self赋值以及为何你不需要给他赋值。举一个例子会使此变得清晰。假如你有一个类称为MyClass和这个类  
的一个实例MyObject。当你调用这个对象的方法MyObject.method(arg1, arg2)的时候，这会由python自动转为  
MyClass.method(MyObject, arg1, arg2)------这就是self的原理了  
这也意味着如果你有一个不需要参数的方法，你还是得给这个方法定义一个self参数。

面向对象的数据它们只是与类和对象的名称空间绑定的普通变量，即这些名称只在这些类与对象的前提下有效。  
有两种类型的域-----类的变量和对象的变量，它们根据是类还是对象拥有这个变量而区分。  
类的变量，由一个类的所有对象（实例）共享使用。只有一个类变量的拷贝，所以当某个对象对类的变量做了改动的时候，这个  
改动会放映在所有其他的实例上。  
对象的变量 由类的每个对象/实例拥有。因此每个对象有自己对这个域的一份拷贝，即他们不是共享的，在同一个类的不同  
实例中，虽然对象的变量有相同的名称，但是是互不相关的。

## 58、Python面向对象中的继承有什么特点？

本文介绍面向对象的3个特点：

1》封装

封装是从业务逻辑中抽象对象时，要赋予对象相关数据与操作，将一些数据和操作打包在一起的过程。封装是使用对象的主要魅力之一，它提供了一个简单方法来创建复杂方案，解决了世界是如何工作的这一问题，我们自然的认为周围的世界是由相互作用的对象组成，每个对象都有自己相关的数据，并能完成一定的功能，从设计的角度来看，封装还提供了一个重要的服务，它分开了是什么和怎么做这两个问题。对象的实现与使用是相互独立的，封装的另外一个优势是支持代码复用，它可以将常用功能以组件方式打包起来。

在上一篇博客（python通过面向对象程序设计方法 实现铅球运行轨迹的计算 网址：http://blog.csdn.net/sxingming/article/details/51280118）中，Projectile类封装了投射体的属性和方法，当主程序对铅球对象处理时，不必关心铅球内部的实现，而是关心这个铅球对象能够做什么，另外Projectile类可被多个程序和多个对象使用，而不需要重复的编写代码。

2》多态

多态意味着多种形式，当用面向对象时，它是指对象是怎么回应一个依赖于对象类型或种类的消息。多态的作用是让程序在不同情况下用一个函数名启用不同的方法。

多态举例：在屏幕上有一个图形对象列表objects，包括circle,rectangle,polygon等，使用相同的代码，可以画出列表中所有的图形：

for obj in objects:

obj.draw(win)

上述for循环，分别执行了circle.draw()方法,rectangle.draw()方法,polygon.draw()方法.

多态给予了面向对象系统极大的灵活性，对象可以用该对象应该用的方式来执行动作，如果没有面向对象，这种灵活性很难实现。

3》继承

一个类（subclass）可以继承另一个类（superclass）.

举例：

建立一个系统以记录员工信息，需要一个Employee类，它包含所有员工都具有的一般信息，其中一个方法是homeAddress()，该方法可返回员工的住址信息。员工分为正式员工(按月发工资)和临时员工（按天发工资），为正式员工定义一个MonthlyEmployee类（Employee类的子类），为临时员工定义一个DaylyEmployee类（Employee类的子类），这两个子类都继承了Employee类，因此都有homeAddress()方法，然而，不同类型的员工，发工资的方法不同，正式员工有monthlyPay()方法，临时员工有daylyPay()方法。

继承的优点：1>建造系统中的类，避免重复操作，例如，我们不必为两个子类分别写一个homeAddress()方法，从Employee类继承即可。2>新类经常是基于已经存在的类，这样就可以提升代码的复用程度。

以上便是，python面向对象程序设计的3个特点：封装，多态，继承。

## 59、面向对象深度优先和广度优先是什么？

Python中分为经典类和新式类：

　　经典类：

　　　　　　class A():

　　　　　　　　pass

　　新式类：

　　　　　　class A(object):

　　　　　　　　pass

　　所以经典类和新式类的区别就是，在声明类的时候，新式类需要加上object关键字。

Python中经典类和新式类的区别：

　　区别主要体现在继承上：

　　　　Python的类可以继承多个类，Java和C#中则只能继承一个类

　　　　Python的类如果继承了多个类，那么其寻找方法的方式有两种

　　　　　　　　当类是经典类时，多继承情况下，会按照深度优先方式查找

　　　　　　　　当类是新式类时，多继承情况下，会按照广度优先方式查找

**简单点说就是：经典类是纵向查找，新式类是横向查找**

首先要理解搜索步，一个完整的搜索步包括两个处理：

a) 获得当前位置上，有几条路可供选择

b) 根据选择策略，选择其中一条路，并走到下个位置

相当于在漆黑的夜里，你只能看清你站的位置和你前面的路，但你不知道每条路能够通向哪里。  
 搜索的任务就是，给出初始位置和目标位置，要求找到一条到达目标的路径。

**深度优先**就是，从初始点出发，不断向前走，如果碰到死路了，就往回走一步，尝试另一条路，直到发现了目标位置。这种不撞南墙不回头的方法，即使成功也不一定找到一条好路，但好处是需要记住的位置比较少。

**广度优先**就是，从初始点出发，把所有可能的路径都走一遍，如果里面没有目标位置，则尝试把所有两步能够到的位置都走一遍，看有没有目标位置；如果还不行，则尝试所有三步可以到的位置。这种方法，一定可以找到一条最短路径，但需要记忆的内容实在很多，要量力而行。

## 60、面向对象中super的作用？

Python中对象方法的定义很怪异，第一个参数一般都命名为self（相当于其它语言的this，比如：C#），用于传递对象本身，而在调用的时候则不

必显式传递，系统会自动传递。

今天我们介绍的主角是super(), 在类的继承里面super()非常常用， 它解决了子类调用父类方法的一些问题， 父类多次被调用时只执行一次， 优化了执行逻辑，下面我们就来详细看一下。

举一个例子：

1. **class** Foo:
2. **def** bar(self, message):
3. **print**(message)
4. >>> Foo().bar("Hello, Python.")
5. Hello, Python.

当存在继承关系的时候，有时候需要在子类中调用父类的方法，此时最简单的方法是把对象调用转换成类调用，需要注意的是这时self参数需要显式传递，例如：

1. **class** FooParent:
2. **def** bar(self, message):
3. **print**(message)
4. **class** FooChild(FooParent):
5. **def** bar(self, message):
6. FooParent.bar(self, message)
7. >>> FooChild().bar("Hello, Python.")
8. Hello, Python.

这样做有一些缺点，比如说如果修改了父类名称，那么在子类中会涉及多处修改，另外，Python是允许多继承的语言，如上所示的方法在多继承时就需要重复写多次，显得累赘。为了解决这些问题，Python引入了super()机制，例子代码如下：

1. **class** FooParent:
2. **def** bar(self, message):
3. **print**(message)
4. **class** FooChild(FooParent):
5. **def** bar(self, message):
6. super(FooChild, self).bar(message)
8. >>> FooChild().bar("Hello, Python.")
9. Hello, Python

表面上看 super(FooChild, self).bar(message)方法和FooParent.bar(self, message)方法的结果是一致的，实际上

## 61、是否使用过functools中的函数？其作用是什么？

Python装饰器（decorator）在实现的时候，有一些细节需要被注意。例如，被装饰后的函数其实已经是另外一个函数了（函数名等函数属性会发生改变）。这样有时候会对程序造成一些不便，例如笔者想对unittest框架中的一些函数添加自定义的decorator，添加后由于函数名和函数的doc发生了改变，对测试结果有一些影响。

所以，Python的functools包中提供了一个叫wraps的decorator来消除这样的副作用。写一个decorator的时候，最好在实现之前加上functools的wrap，它能保留原有函数的名称和docstring。

## 62、列举面向对象中带爽下划线的特殊方法，如：\_\_new\_\_、\_\_init\_\_

*# -\*- coding: utf-8 -\*-***class** Person(object):  
 *"""Silly Person"""* **def** \_\_new\_\_(cls, name, age):  
 print (**'\_\_new\_\_ called.'**)  
 **return** super(Person, cls).\_\_new\_\_(cls, name, age)  
  
 **def** \_\_init\_\_(self, name, age):  
 print (**'\_\_init\_\_ called.'**)  
 self.name = name  
 self.age = age  
 **def** \_\_str\_\_(self):  
 **return '<Person: %s(%s)>'** % (self.name, self.age)  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 piglei = Person(**'piglei'**, 24)  
 print (piglei)

## 63、如何判断是函数还是方法？

一、函数和方法的区别

1、函数要手动传self，方法不用传

2、如果是一个函数，用类名去调用，如果是一个额方法，用对象去调用

举例说明：

class Foo(object):

def \_\_init\_\_(self):

self.name="haiyan"

def func(self):

print(self.name)

obj = Foo()

obj.func()

Foo.func(obj)

判断函数和方法的方式

from types import FunctionType,MethodType

obj = Foo()

print(isinstance(obj.func,FunctionType)) #False

print(isinstance(obj.func,MethodType)) #True #说明这是一个方法

print(isinstance(Foo.func,FunctionType)) #True #说明这是一个函数。

print(isinstance(Foo.func,MethodType)) #False

## 64、静态方法和类方法区别？

面相对象程序设计中，类方法和静态方法是经常用到的两个术语。

逻辑上讲：类方法是只能由类名调用；静态方法可以由类名或对象名进行调用。

## 65、列举面向对象中的特殊成员以及应用场景

## 66、1、2、3、4、5 能组成多少个互不相同且无重复的三位数

sum =0  
**for** i **in** range(1,6):  
 **for** j **in** range(1,6):  
 **for** k **in** range(1,6):  
 **if**(i != j **and** i != k **and** j != k):  
 print(i,j,k)  
 sum += 1

## 68、metaclass作用？以及应用场景？

## 69、用尽量多的方法实现单例模式。

## 70、装饰器的写法以及应用场景。

## 71、异常处理写法以及如何主动跑出异常（应用场景）

## 72、什么是面向对象的mro

## 73、isinstance作用以及应用场景？

## 74、写代码并实现：

## 75、json序列化时，可以处理的数据类型有哪些？如何定制支持datetime类型？

## 76、json序列化时，默认遇到中文会转换成unicode，如果想要保留中文怎么办？

## 77、什么是断言？应用场景？

## 78、有用过with statement吗？它的好处是什么？

## 79、使用代码实现查看列举目录下的所有文件。

## 80、简述 yield和yield from关键字。

## 81、[PYTHON的RE模块理解（RE.COMPILE、RE.MATCH、RE.SEARCH）](https://www.cnblogs.com/papapython/p/7482349.html)

**import re  
help(re.compile)  
'''  
输出结果为：  
Help on function compile in module re:  
  
compile(pattern, flags=0)  
 Compile a regular expression pattern, returning a pattern object.  
通过help可知：编译一个正则表达式模式，返回一个模式对象。  
'''  
  
'''  
第二个参数flags是匹配模式，可以使用按位或’|’表示同时生效，也可以在正则表达式字符串中指定。  
Pattern对象是不能直接实例化的，只能通过compile方法得到。匹配模式有：   
1).re.I(re.IGNORECASE): 忽略大小写  
2).re.M(MULTILINE): 多行模式，改变’^’和’$’的行为  
3).re.S(DOTALL): 点任意匹配模式，改变’.’的行为  
4).re.L(LOCALE): 使预定字符类 \w \W \b \B \s \S 取决于当前区域设定  
5).re.U(UNICODE): 使预定字符类 \w \W \b \B \s \S \d \D 取决于unicode定义的字符属性  
6).re.X(VERBOSE): 详细模式。这个模式下正则表达式可以是多行，忽略空白字符，并可以加入注释   
'''  
  
text="JGod is a handsome boy ,but he is a ider"  
print re.findall(r'\w\*o\w\*',text) #查找有o的单词  
#输出结果为：['JGod', 'handsome', 'boy']  
  
#利用compile生成一个规则模式吧，然后利用findall将某一个对象内容进行匹配。，合适则输出符合规则的内容  
regex=re.compile(r'\w\*o\w\*')  
print regex.findall(text)  
#>>> ['JGod', 'handsome', 'boy']  
  
test1="who you are,what you do,When you get get there? What is time you state there?"  
regex1=re.compile(r'\w\*wh\w\*',re.IGNORECASE)  
wh=regex1.findall(test1)  
print wh  
#>>> ['who', 'what', 'When', 'What']**

'''  
re正则表达式模块还包括一些有用的操作正则表达式的函数。下面主要介绍match函数以及search函数。  
定义： re.match 尝试从字符串的开始匹配一个模式。  
原型：  
re.match(pattern, string, flags)  
第一个参数是正则表达式,如果匹配成功，则返回一个Match，否则返回一个None；  
第二个参数表示要匹配的字符串；  
第三个参数是标致位，用于控制正则表达式的匹配方式，如：是否区分大小写，多行匹配等等。  
  
函数的返回值为真或者假。  
例如：match(‘p’,’python’)返回值为真；match(‘p’,’www.python.org’)返回值为假。  
  
定义：re.search会在给定字符串中寻找第一个匹配给定正则表达式的子字符串。  
  
函数的返回值：如果查找到则返回查找到的值，否则返回为None。  
  
原型：  
re.search(pattern, string, flags)  
  
每个参数的含意与re.match一样。  
'''  
#re.match的例子1  
**import re  
your\_love=re.match("wh","What are you doing? who is you mate?",re.I)  
if your\_love:  
 print "you are my angle"  
else:  
 print "i lose you "  
#相当于：  
print "\*"\*100 #便于区分  
import re  
content="What are you doing? who is your mate?"  
regu\_cont=re.compile("\w\*wh\w\*",re.I)  
yl=regu\_cont.match(content)  
if yl:  
 print yl.group(0)  
else:  
 print "what happen?"  
解析：首先创造了需要正则表达式匹配的字符串content;  
接着利用re.compile()来创建了我们所需要的匹配规则，创建了模式对象regu\_cont；  
yl用来接收对内容content字符串进行regu\_cont正则表达式实现match函数的结果  
如果有yl不为空，则使用m.group(index)输出查找到的子字符串**

**否则（返回值为None） print “what happen?”  
  
  
match例子2**

'''  
match如果查找到结果， 将返回一个 MatchObject，你可以查询 MatchObject 关于匹配字符串的相关信息了。MatchObject 实例也有几个方法和属性；最重要的那些如下所示：  
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  
'''  
**import re  
content="What are you doing? who is your mate?"  
regu\_cont=re.compile("\w\*wh\w\*",re.I)  
yl=regu\_cont.match(content)  
if yl:  
 print yl.group(0)  
else:  
 print "pass the test"  
print yl.group()  
print yl.start()  
print yl.end()  
print yl.span()  
  
执行结果为：  
What  
What  
0  
4  
(0, 4)**

**#search()方法与match()方法类似**

import re

content='Where are you from? You look so hansome.'

regex=re.compile(r'\w\*som\w\*')

m=regex.search(content)

if m:

print m.group(0)

else:

print "Not found"

#相当于：  
**import re  
m=re.search(r'\w\*som\w\*','Where are you from? You look so handsome.',re.I)  
if m:  
 print m.group(0)  
else:  
 print "not found"**

findall

在string中查找所有 **匹配成功的组**, 即用括号括起来的部分。返回list对象，每个list item是由每个匹配的所有组组成的list。

finditer

## 82、python代码

master\_fired\_items = [p **for** sub\_list **in** chain\_items\_list **for** p **in** sub\_list **if** p.type **in** (**'PRESENT'**, **'TRADE'**, **'NORMAL'**, **'MEAL\_BUNDLE'**, **'MEAL\_CUSTOM'**)]  
child\_fired\_items = [p **for** sub\_list **in** chain\_items\_list **for** p **in** sub\_list **if** p.type **not in** (**'PRESENT'**, **'TRADE'**, **'NORMAL'**, **'MEAL\_BUNDLE'**, **'MEAL\_CUSTOM'**)]

from decimal import Decimal

from decimal import getcontext

d\_context = getcontext()

d\_context.prec = 6

print(d\_context)

d = Decimal(1) / Decimal(3)

print(type(d), d)

amount = format(Decimal(amount), **".2f"**)

order\_dis\_sum\_amt = sum([Decimal(str(d.amount)) **for** d **in** self.\_order\_discounts **if** self.\_\_is\_entire\_discount(d)])

item\_lines\_list = [i **for** i **in** self.\_order\_data.items **if** str(i.item\_id) **in** present\_promotion.fired\_items]

adapter\_package = \_\_import\_\_(processors.\_\_name\_\_, fromlist=**"dummy"**)

 \_\_import\_\_(package.module)相当于from package import name，如果fromlist不传入值，则返回package对应的模块，如果fromlist传入值，则返回package.module对应的模块。

order\_data = order\_builder.get\_order().encode(**'utf-8'**)

dept\_class\_name=  
[c.get(**'name'**) **for** c **in** SALES\_CHANNEL\_MAP\_LIST **if** c[**'code'**] == this\_pos\_material.get(**'model'**)][  
 0] **if** any(  
 [c **for** c **in** SALES\_CHANNEL\_MAP\_LIST **if** c[**'code'**] == this\_pos\_material.get(**'model'**)]) **else ''**,  
pay\_amount=parse\_float(this\_pos\_material.get(**'grantTotal'**, 0)),  
pos\_name=this\_pos\_material.get(**'storeInfo'**, {}).get(**'pos\_code'**),  
pos\_id=this\_pos\_material.get(**'storeInfo'**, {}).get(**'pos\_id'**),  
close\_user\_name=this\_pos\_material.get(**'user'**, {}).get(**'name'**),  
order\_unique\_no=this\_pos\_material.get(**'ticket\_id'**),  
close\_time=this\_pos\_material.get(**'endTime'**),  
customer\_name=**''**,  
disc\_style=**''**,  
open\_user\_name=this\_pos\_material.get(**'user'**, {}).get(**'name'**),  
card\_no=**''**,  
disc\_name=**''**,  
dis\_amount=0, *# 总折扣金额*pay\_method=**''**,  
address=**''**,  
chargehostid=this\_pos\_material.get(**'voidTicket'**),  
peoples=this\_pos\_material.get(**'peoples'**),  
item\_amount=parse\_float(this\_pos\_material.get(**'subTotal'**, 0)),  
pay\_channel=this\_pos\_material.get(**'sub\_channel'**, {}).get(**'code'**), *# 小渠道*open\_user\_id=this\_pos\_material.get(**'user'**, {}).get(**'code'**, **''**),  
is\_cancel=True **if** this\_pos\_material.get(**'isVoid'**) == 1 **else** False,  
discounts=[],  
items=[]

this\_trans\_order.sign = hashlib.md5(json.dumps(this\_pos\_data)).hexdigest()

**if not** isinstance(value, (int, long, float, Decimal)):  
 **return** value

self.created = datetime.datetime.now()  
self.updated = datetime.datetime.now()

super(TransactionManagementChangeLog, self).\_\_init\_\_(ticket\_id=ticket\_id)

**if** isinstance(value, basestring) **or** isinstance(value, str):  
 list\_value = value.split(**','**)

**basestring()** 方法是 str 和 unicode 的超类（父类），也是抽象类，因此不能被调用和实例化，但可以被用来判断一个对象是否为 str 或者 unicode 的实例，isinstance(obj, basestring) 等价于 isinstance(obj, (str, unicode))。

以下是 basestring() 方法的语法:

basestring()

以下展示了使用 basestring 函数的实例：

>>>isinstance("Hello world", str) True >>> isinstance("Hello world", basestring) True

exclude\_data = [v **for** k, v **in** methods\_data.items() **if** k **not in** value\_list]  
self.method = exclude\_data **if** len(exclude\_data) **else** None

**def** bind(self, topic=None, channel=None, error\_handler=None):  
 **def** decorator(f):  
 **if** topic **is** None **or** topic == **''**:  
 **raise** AttributeError(**'topic is empty'**)  
 **if** channel **is** None **or** channel == **''**:  
 **raise** AttributeError(**'channel is empty'**)  
 key = **'%s:%s'** % (topic, channel)  
 **if** key **in** self.\_\_tasks:  
 **raise** AttributeError(**'the combination of topic:%s and channel:%s is existing'** % (topic, channel))  
 self.\_\_tasks[key] = dict(message\_handler=f, error\_handler=error\_handler)  
 **return** f  
  
 **return** decorator

*# querystring查询条件***def** query\_search(func):  
 @functools.wraps(func)  
 **def** wrapper(\*args, \*\*kw):  
 searchs = {}  
 **for** k, v **in** request.values.items():  
 searchs[k] = v  
 g.query\_search = searchs  
 g.query\_search[**'limit'**] = g.limit  
 g.query\_search[**'offset'**] = g.offset  
 **if** hasattr(g, **'user'**):  
 g.query\_search[**'user'**] = g.user  
 **return** func(\*args, \*\*kw)  
  
 **return** wrapper

**def** sa\_rule(f):  
 @wraps(f)  
 **def** decorator(\*args, \*\*kw):  
 uid = session[**'id'**]  
 user = User.query.get(uid)  
 partner\_role = [r **for** r **in** user.roles **if** r.name == **'sa' or** r.name == **'partner manager'**]  
 **if** len(partner\_role) > 0:  
 **return** f(\*args, \*\*kw)  
 **else**:  
 **return** jsonify(  
 error\_message=**'user has no permission to create application'** )  
  
 **return** decorator

**class** StatusCode(object):  
 @property  
 **def** SUCCESS(self):  
 **return** 0  
  
 @property  
 **def** ERROR(self):  
 **return** 1  
  
 @property  
 **def** WARINING(self):  
 **return** 2  
  
  
StatusCode = StatusCode()

有没有既能检查参数，又可以用类似属性这样简单的方式来访问类的变量呢？对于追求完美的Python程序员来说，这是必须要做到的！

还记得装饰器（decorator）可以给函数动态加上功能吗？对于类的方法，装饰器一样起作用。Python内置的@property装饰器就是负责把一个方法变成属性调用的：

cols = line.strip().split(**','**)  
**if** len(cols) < len(clms):  
 **raise** Exception(**"%s row's Data error"** % rowNumber)  
**for** i **in** range(len(clms)):  
 rowData[clms[i]] = cols[i].strip()  
arr.append(rowData)  
rowNumber = rowNumber + 1

**if not** os.path.exists(downloadpath):  
 os.makedirs(downloadpath)  
**if** local\_success **is not** None **and** len(local\_success) > 0:  
 **if not** os.path.exists(local\_path + local\_success):  
 os.makedirs(local\_path + local\_success)  
**if** local\_error **is not** None **and** len(local\_error) > 0:  
 **if not** os.path.exists(local\_path + local\_error):  
 os.makedirs(local\_path + local\_error)  
logging.debug(**"\*\* download txt files start\*\*"**)  
ftp = FtpClient(ftp\_address, ftp\_username, ftp\_password, ftp\_port, ftp\_min, ftp\_filePrefix, ftp\_timeout)  
**try**:  
 ftp.login()  
 ftp.download\_files(downloadpath, romoteFilePath)  
**except** Exception **as** e:

pop()函数一般用来删除list列表的末尾元素，同样，kwargs.pop()用来删除关键字参数中的末尾元素，比如：  
kwargs = {'Michael': 95, 'Bob': 75, 'Tracy': 85}  
kwargs.pop()= {'Michael': 95, 'Bob': 75}

**class** EinvoiceTransactionOrder(db.Model, AsDict):  
 \_\_bind\_key\_\_ = **'pgbi'** id = db.Column(db.BigInteger, primary\_key=True, default=get\_guid, index=True)  
 order\_code = db.Column(db.String(50), nullable=False, index=True)  
 order\_type = db.Column(db.String(20), nullable=False,default=**'S'**)  
 sale\_method = db.Column(db.String(5))  
 original\_order\_code = db.Column(db.String(50))  
 store\_code = db.Column(db.String(20), nullable=False, index=True)  
 member\_code = db.Column(db.String(50))  
 guest\_count = db.Column(db.Integer, nullable=False,default=1)  
 guest\_type = db.Column(db.String(10), nullable=False,default=**'N'**)  
 sales\_id = db.Column(db.String(20))  
 sales\_date = db.Column(db.DateTime)  
 gross\_amount = db.Column(db.DECIMAL(precision=16, scale=2, decimal\_return\_scale=2), default=0.0)  
 net\_amount = db.Column(db.DECIMAL(precision=16, scale=2, decimal\_return\_scale=2), default=0.0)  
 discount\_amount = db.Column(db.DECIMAL(precision=16, scale=2, decimal\_return\_scale=2), default=0.0)  
 item\_count = db.Column(db.Integer, default=1)  
 payee = db.Column(db.String(20))  
 checker = db.Column(db.String(20))  
 remark = db.Column(db.String(255))  
 items = db.Column(JSON)  
 discounts = db.Column(JSON)  
 payments = db.Column(JSON)

## 83、加密算法

博客：https://www.cnblogs.com/xiao-apple36/p/8744408.html

*# encoding: utf-8***from** Cryptodome.Cipher **import** AES  
**from** binascii **import** b2a\_hex, a2b\_hex  
  
AES\_LENGTH = 16  
  
**class** prpcrypt():  
 **def** \_\_init\_\_(self, key):  
 self.key = key  
 self.mode = AES.MODE\_ECB  
 self.cryptor = AES.new(self.pad\_key(self.key).encode(), self.mode)  
  
 *# 加密函数，如果text不是16的倍数【加密文本text必须为16的倍数！】，那就补足为16的倍数  
 # 加密内容需要长达16位字符，所以进行空格拼接* **def** pad(self,text):  
 **while** len(text) % AES\_LENGTH != 0:  
 text += **' '  
 return** text  
  
 *# 加密密钥需要长达16位字符，所以进行空格拼接* **def** pad\_key(self,key):  
 **while** len(key) % AES\_LENGTH != 0:  
 key += **' '  
 return** key  
  
 **def** encrypt(self, text):  
  
 *# 这里密钥key 长度必须为16（AES-128）、24（AES-192）、或32（AES-256）Bytes 长度.目前AES-128足够用  
 # 加密的字符需要转换为bytes  
 # print(self.pad(text))* self.ciphertext = self.cryptor.encrypt(self.pad(text).encode())  
 *# 因为AES加密时候得到的字符串不一定是ascii字符集的，输出到终端或者保存时候可能存在问题  
 # 所以这里统一把加密后的字符串转化为16进制字符串* **return** b2a\_hex(self.ciphertext)  
  
 *# 解密后，去掉补足的空格用strip() 去掉* **def** decrypt(self, text):  
 plain\_text = self.cryptor.decrypt(a2b\_hex(text)).decode()  
 **return** plain\_text.rstrip(**' '**)  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 pc = prpcrypt(**'abcdef'**) *# 初始化密钥* e = pc.encrypt(**"0123456789ABCDEF"**)  
 d = pc.decrypt(e)  
 print(e)  
 print(d)  
 e = pc.encrypt(**"00000000000000000000000000"**)  
 d = pc.decrypt(e)  
 print(e, d)  
 e = pc.encrypt(**"YQC\*&!@@#"**)  
 d = pc.decrypt(e)  
 print(e)  
 print()

## 84、ToKEN

**import** time  
**import** base64  
**import** hmac  
  
**def** generate\_token(key, expire=3600):  
 *r'''  
 @Args:  
 key: str (用户给定的key，需要用户保存以便之后验证token,每次产生token时的key 都可以是同一个key)  
 expire: int(最大有效时间，单位为s)  
 @Return:  
 state: str  
 '''* ts\_str = str(time.time() + expire)  
 ts\_byte = ts\_str.encode(**"utf-8"**)  
 sha1\_tshexstr = hmac.new(key.encode(**"utf-8"**),ts\_byte,**'sha1'**).hexdigest()  
 token = ts\_str+**':'**+sha1\_tshexstr  
 b64\_token = base64.urlsafe\_b64encode(token.encode(**"utf-8"**))  
 **return** b64\_token.decode(**"utf-8"**)  
  
**def** certify\_token(key, token):  
 *r'''  
 @Args:  
 key: str  
 token: str  
 @Returns:  
 boolean  
 '''* token\_str = base64.urlsafe\_b64decode(token).decode(**'utf-8'**)  
 token\_list = token\_str.split(**':'**)  
 **if** len(token\_list) != 2:  
 **return False** ts\_str = token\_list[0]  
 **if** float(ts\_str) < time.time():  
 *# token expired* **return False** known\_sha1\_tsstr = token\_list[1]  
 sha1 = hmac.new(key.encode(**"utf-8"**),ts\_str.encode(**'utf-8'**),**'sha1'**)  
 calc\_sha1\_tsstr = sha1.hexdigest()  
 **if** calc\_sha1\_tsstr != known\_sha1\_tsstr:  
 *# token certification failed* **return False** *# token certification success* **return True**key = **"JD98Dskw=23njQndW9D"***# 一小时后过期*token = generate\_token(key, 3600)  
print(token)  
print(certify\_token(key, token))

其一, 在不加括号时候,**and优先级大于or**

其二, x or y 的值**只可能是x或y**. x为真就是x, x为假就是y

第三, x and y 的值**只可能是x或y**. x为真就是y, x为假就是x

显然

对于, 1 or 5 and 4: 先算5 and 4, 5为真, 值为4. 再算1 or 4, 1 为真,值为1

对于, (1 or 5) and 4: 先算1 or 5, 1为真, 值为1. 再算1 and 4, 1为真,值为4