1,有一张商品购买记录表T，其中包含字段uid（用户id）pid（商品id），现在需要统计既买了商品A（即pid为A）又买了商品B（即pid为B）的用户uid，请写出对应的SQL语句。

Select distinct(uid) from t where

Uid in (select uid from t where pid=’A’)

And uid in (select uid from t where pid=’B’)

2，有两个有序整型数组：A和B，都是从小到大排列，要求把A和B合并成数组C，且C也是有序的，例如A = [1,3,5,7...]B = [2,4,6,8...],合并后C = [1,2,3,4,5,6,7,8...]，请写一个方法尽可能的高效率来实现（注意：不能使用原系统提供的函数）

list1=[1,3,5,7]  
list2=[2,4,6,8,10]  
a=b=0  
c=[]  
**for** i **in** range(a,len(list1)):  
 **for** j **in** range(b,len(list2)):  
 **if** list2[j]<=list1[i]:  
 c.append(list2[j])  
 b = b + 1  
 **else**:  
 c.append(list1[i])  
 a = a + 1  
 **break  
if** b < len(list2):  
 **for** i **in** range(b,len(list2)):  
 c.append(list2[i])  
**if** a < len(list1):  
 **for** i **in** range(a,len(list1)):  
 c.append(list1[i])  
print(c)

3，请使用python将字符串“1,2,3,4,5”转换成字符串“5|4|3|2|1”

a = **'1,2,3,4,5'**a = a.replace(**','**,**'|'**)  
b = **''  
for** i **in** range(0,len(a)):  
 b += a[len(a)-1-i]  
print(b)

**s1='1,2,3,4,5'  
s2=list[::-1].replace(',','|')  
print(s2)**

4，请分别描述python2和python3中，import包时的路径的搜索顺序

**当你导入一个模块，Python 解析器对模块位置的搜索顺序是：**

**1、当前目录**

**2、如果不在当前目录，Python 则搜索在 shell 变量 PYTHONPATH 下的每个目录。**

**3、如果都找不到，Python会察看默认路径。UNIX下，默认路径一般为/usr/local/lib/python/。**

**模块搜索路径存储在 system 模块的 sys.path 变量中。变量里包含当前目录，PYTHONPATH和由安装过程决定的默认目录。**

5，用python的正则表达式匹配时间的信息

**用正则表达式验证日期的难点有二：一是大小月份的天数不同，二是闰年的考虑。  
对于第一个难点，我们首先不考虑闰年，假设2月份都是28天，这样，月份和日期可以分成三种情况：  
  
1、月份为 1, 3, 5, 7, 8, 10, 12，天数范围为 01 - 31，匹配MM-DD的正则表达式为：**

**(0[13578]|1[02])-(0[1-9]|[12][0-9]|3[01])**

**2、月份为 4, 6, 9, 11，天数范围为 01-30，匹配MM-DD的正则表达式为：**

**(0[469]|11)-(0[1-9]|[12][0-9]|30)**

**3、月份为 2，考虑平年情况，匹配MM-DD的正则表达式为：**

**02-(0[1-9]|[1][0-9]|2[0-8])**

**根据上面的成果，我们可以得到匹配平年日期格式为YYYY-MM-DD的正则表达式：  
  
([0-9]{3}[1-9]|[0-9]{2}[1-9][0-9]{1}|[0-9]{1}[1-9][0-9]{2}|[1-9][0-9]{3})-(((0[13578]|1[02])-(0[1-9]|[12][0-9]|3[01]))|((0[469]|11)-(0[1-9]|[12][0-9]|30))|(02-(0[1-9]|[1][0-9]|2[0-8])))**

6，使用python编写一个装饰器，打印被装饰器的输入和输出

**def** check\_result(function):  
 **def** wrap(\*args, \*\*kwargs):  
 result = function(\*args, \*\*kwargs)  
 **if** result > 100:  
 **return** 0  
 **else**:  
 **return** result  
 **return** wrap  
  
@check\_result  
**def** foo(x, y):  
 **return** x \*\* y  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 **print**(foo(2 ,2))

7，阐述range和xrange的区别，并且用python仿写xrange函数

**range 前面小节已经说明了，range([start,] stop[, step])，根据start与stop指定的范围以及step设定的步长，生成一个序列。**

[**xrange**](http://www.nowamagic.net/academy/tag/xrange) **用法与 range 完全相同，所不同的是生成的不是一个list对象，而是一个生成器。**

**要生成很大的数字序列的时候，用xrange会比range性能优很多，因为不需要一上来就开辟一块很大的内存空间。**

8，列举几种你曾经经常用的python包并且解释其功能以及用法

1. **os模块：**os模块包装了不同操作系统的通用接口，使用户在不同操作系统下，可以使用相同的函数接口，返回相同结构的结果。

  os中有大量文件、路径操作的相关函数

os.path:在不同的操作系统中调用不同的模块，是一个可import的模块，这个模块中提供很多有用的操作：

1. **sys模块：**系统信息和方法模块，提供了很多实用的变量和方法
2. **built-in内置模块：**

4..**time模块**

  这个模块定义的都是和时间、时钟、计时相关的内容：

    clock():返回第一次调用到当前调用时的计时，是以秒为单位的浮点数

**5.re模块**

9，python判断变量函数时，判断变量的类型为什么不使用type=（），而是使用isinstance（）方法

type()不会认为子类是一种父类类型。

isinstance()会认为子类是一种父类类型

10，写一个函数，输入一个字符串，返回倒叙排列的结果：如string\_reverse('abcdef'),返回：‘fedcba’

**def** string\_reverse(a):  
 b = **''** num = len(a)  
 **for** i **in** range(0,num):  
 b += a[num-1-i]  
 **return** b  
print(string\_reverse(**"abcdef"**))

11，请使用自己的算法，按升序合并如下两个list，并去除重复的元素：

list1=[1,7,5,3]  
list2=[2,11,1,7,8,4,9,10]  
list1.sort()  
list2.sort()  
a=b=0  
c=[]  
**for** i **in** range(a,len(list1)):  
 **for** j **in** range(b,len(list2)):  
 **if** list2[j]<=list1[i]:  
 c.append(list2[j])  
 b = b + 1  
 **else**:  
 c.append(list1[i])  
 a = a + 1  
 **break  
if** b < len(list2):  
 **for** i **in** range(b,len(list2)):  
 c.append(list2[i])  
**if** a < len(list1):  
 **for** i **in** range(a,len(list1)):  
 c.append(list1[i])  
print(set(c))

两个无序数列降序排序：

list1=[1,7,5,3]  
list2=[2,11,1,7,8,4,9,10]  
list1.sort()  
list2.sort()  
a=b=0  
c=[]  
**for** i **in** range(a,len(list1)):  
 **for** j **in** range(b,len(list2)):  
 **if** list2[j]<=list1[i]:  
 c.append(list2[j])  
 b = b + 1  
 **else**:  
 c.append(list1[i])  
 a = a + 1  
 **break  
if** b < len(list2):  
 **for** i **in** range(b,len(list2)):  
 c.append(list2[i])  
**if** a < len(list1):  
 **for** i **in** range(a,len(list1)):  
 c.append(list1[i])  
c=set(c)  
c=list(c)  
c.reverse()  
print(c)

12,MYSQL中，delete from表名和truncate table表名的区别

**下面是delete from 表名和truncate table 表名方面的区别，供大家在删除数据表时作个参考了。**

**1、delete不能使自动编号返回为起始值。但是truncate能使自动增长的列的值返回为默认的种子。  
2、truncate只能一次清空，不能按条件删除。但是delete可以按条件清除部分记录。  
3、truncate清空数据表性能（速度）比delete快。  
4、truncate不会记录到系统日志，不会触发delete触发器**

MySQL中有两种删除表中记录的方法，一种是delete from语句，另一种是truncate table语句**。   
（1）delete from语句可以使用where对要删除的记录进行选择。而使用truncate table将删除表中的所有记录。因此，delete语句更灵活。   
（2）如果要清空表中的所有记录，可以使用下面的两种方法：   
delete from tablename   
truncate table tablename   
其中第二条记录中的table是可选的。   
（3）如果要删除表中的部分记录，只能使用delete语句。   
delete FROM table1 WHERE ;   
如果delete不加WHERE子句，那么它和truncate table是一样的，但它们有一点不同，那就是delete可以返回被删除的记录数，而truncate table返回的是0。   
（4）如果一个表中有自增字段，使用truncate table和没有WHERE子句的delete删除所有记录后，这个自增字段将起始值恢复成1.如果你不想这样做的话，可以在delete语句中加上永真的WHERE，如WHERE 1或WHERE true。   
delete FROM table1 WHERE 1;   
上面的语句在执行时将扫描每一条记录。但它并不比较，因为这个WHERE条件永远为true。这样做虽然可以保持自增的最大值，但由于它是扫描了所有的记录，因此，它的执行成本要比没有WHERE子句的delete大得多。**

13，在mysql中，要生成下列结果，SQL语句如何实现？

表名：com\_result

内容：

rq 胜负

2017-04-09 胜

2017-04-09 胜

2017-04-09 负

2017-04-09 负

2017-04-10 胜

2017-04-10 负

2017-04-10 负

生成结果：

rq 胜 负

2017-04-09 2 2

2017-04-10 1 2

Select

14，简述TCP断开握手的过程

**由于TCP连接是全双工的，因此每个方向都必须单独进行关闭。这个原则是当一方完成它的数据发送任务后就能发送一个FIN来终止这个方向的连接。收到一个 FIN只意味着这一方向上没有数据流动，一个TCP连接在收到一个FIN后仍能发送数据。首先进行关闭的一方将执行主动关闭，而另一方执行被动关闭。**

**CP的连接的拆除需要发送四个包，因此称为四次挥手(four-way handshake)。客户端或服务器均可主动发起挥手动作，在socket编程中，任何一方执行close()操作即可产生挥手操作。**

**（1）客户端A发送一个FIN，用来关闭客户A到服务器B的数据传送。**

**（2）服务器B收到这个FIN，它发回一个ACK，确认序号为收到的序号加1。和SYN一样，一个FIN将占用一个序号。**

**（3）服务器B关闭与客户端A的连接，发送一个FIN给客户端A。**

**（4）客户端A发回ACK报文确认，并将确认序号**设置为收到序号加1。

15，简述进程，线程，协程的含义

16，简述静态方法，类方法，实例方法的区别

**# coding=utf-8**

**class A(object):**

**def \_\_init\_\_(self, x=0):**

**self.x = x**

**#类实例方法 # 最常见的实例方法**

**def foo(self, y=0):**

**print self.x, y**

**#类方法 ＃ cls即为类自身**

**@classmethod**

**def class\_foo(cls, x=1, y=1):**

**return cls(x) ＃ 因为类方法可以传参一个cls参数，所以可以调用类的相关信息，而静态方法是一个独立的函数，也是可以调用的，这里为了简单，没有调用静态方法**

**#静态方法 # 不能传递和类或实例相关的参数，如cls或self，但可以传递其他参数**

**@staticmethod**

**def static\_foo(x=2):**

**return A(x)**

17，dict的items（）与itreritems（）方法的不同

**items()返回的是列表对象，而iteritems()返回的是iterator对象。例如：**

**print dic.items() #[('a', 'hello'), ('c', 'you'), ('b', 'how')]**

**print dic.iteritems() #<dictionary-itemiterator object at 0x020E9A50>**

**深究：iteritor是迭代器的意思，一次反悔一个数据项，知道没有为止**

**for i in dic.iteritems():  
  
　　print i  
  
结果：('a', 'hello')  
  
('c', 'you')  
  
('b', 'how')**

18，请简述魔法函数\_\_new\_\_,\_\_call\_\_,\_\_repe\_\_,\_\_str\_\_,\_\_setattr\_\_,\_\_getattr\_\_的含义是什么？

\_**\_call\_\_**

**Python中有一个有趣的语法，只要定义类型的时候，实现\_\_call\_\_函数，这个类型就成为可调用的。**

**换句话说，我们可以把这个类型的对象当作函数来使用，相当于 重载了括号运算符。**

**\_\_call\_\_**

**在Python中，函数其实是一个对象：**

**>>> f = abs**

**>>> f.\_\_name\_\_**

**'abs'**

**>>> f(-123)**

**123**

**由于 f 可以被调用，所以，f 被称为可调用对象。**

**所有的函数都是可调用对象。**

**一个类实例也可以变成一个可调用对象，只需要实现一个特殊方法\_\_call\_\_()。**

**我们把 Person 类变成一个可调用对象：**

**class Person(object):**

**def \_\_init\_\_(self, name, gender):**

**self.name = name**

**self.gender = gender**

**def \_\_call\_\_(self, friend):**

**print 'My name is %s...' % self.name**

**print 'My friend is %s...' % friend**

**现在可以对 Person 实例直接调用：**

**>>> p = Person('Bob', 'male')**

**>>> p('Tim')**

**My name is Bob...**

**My friend is Tim...**

**\_\_repr\_\_和\_\_str\_\_这两个方法都是用于显示的，\_\_str\_\_是面向用户的，而\_\_repr\_\_面向程序员。**

* **打印操作会首先尝试\_\_str\_\_和str内置函数(print运行的内部等价形式)，它通常应该返回一个友好的显示。**
* **\_\_repr\_\_用于所有其他的环境中：用于交互模式下提示回应以及repr函数，如果没有使用\_\_str\_\_，会使用print和str。它通常应该返回一个编码字符串，可以用来重新创建对象，或者给开发者详细的显示。**

**当我们想所有环境下都统一显示的话，可以重构\_\_repr\_\_方法；当我们想在不同环境下支持不同的显示，例如终端用户显示使用\_\_str\_\_，而程序员在开发期间则使用底层的\_\_repr\_\_来显示，实际上\_\_str\_\_只是覆盖了\_\_repr\_\_以得到更友好的用户显示。**

#### \_\_getattr\_\_方法

## 拦截点号运算。当对未定义的属性名称和实例进行点号运算时，就会用属性名作为字符串调用这个方法。如果继承树可以找到该属性，则不调用此方法

1. **class empty:**
2. **def\_\_getattr\_\_(self, attrname):**
3. **ifattrname =="age":**
4. **return40**
5. **else:**
6. **raiseAttributeError, attrname**
8. **x =empty()**
9. **print(x.age)       #40**
10. **print(x.name)      #error text omitted..AttributeError,nam**

### object.\_getattr\_(self, name)

**实例instance通过instance.name访问属性name，只有当属性name没有在实例的\_\_dict\_\_或它构造类的\_\_dict\_\_或基类的\_\_dict\_\_中没有找到，才会调用\_\_getattr\_\_。当属性name可以通过正常机制追溯到时，\_\_getattr\_\_是不会被调用的。如果在\_\_getattr\_\_(self, attr)存在通过self.attr访问属性，会出现无限递归错误。**

**class ClassA(object):**

**def \_\_init\_\_(self, classname):**

**self.classname = classname**

**def \_\_getattr\_\_(self, attr):**

**return('invoke \_\_getattr\_\_', attr)**

**insA = ClassA('ClassA')**

**print(insA.\_\_dict\_\_) # 实例insA已经有classname属性了**

**# {'classname': 'ClassA'}**

**print(insA.classname) # 不会调用\_\_getattr\_\_**

**# ClassA**

**print(insA.grade) # grade属性没有找到，调用\_\_getattr\_\_**

**# ('invoke \_\_getattr\_\_', 'grade')**

**这里empty类和实例x并没有属性age，所以执行x.age时，就会调用\_\_getattr\_\_方法，对于name也是同样。**

#### \_\_setattr\_\_方法

**会拦截所有属性的的赋值语句。如果定义了这个方法，self.arrt = value 就会变成self,\_\_setattr\_\_("attr", value).这个需要注意。当在\_\_setattr\_\_方法内对属性进行赋值是，不可使用self.attr = value,因为他会再次调用self,\_\_setattr\_\_("attr", value),则会形成无穷递归循环，最后导致堆栈溢出异常。应该通过对属性字典做索引运算来赋值任何实例属性，也就是使用self.\_\_dict\_\_['name'] = value.**

### object.\_\_setattr\_\_(self, name, value) 如果类自定义了\_\_setattr\_\_方法，当通过实例获取属性尝试赋值时，就会调用\_\_setattr\_\_。 常规的对实例属性赋值，被赋值的属性和值会存入实例属性字典\_\_dict\_\_中。

**class ClassA(object):**

**def \_\_init\_\_(self, classname):**

**self.classname = classname**

**insA = ClassA('ClassA')**

**print(insA.\_\_dict\_\_)**

**# {'classname': 'ClassA'}**

**insA.tag = 'insA'**

**print(insA.\_\_dict\_\_)**

**# {'tag': 'insA', 'classname': 'ClassA'}**

**如下类自定义了\_\_setattr\_\_,对实例属性的赋值就会调用它。类定义中的self.attr也同样，所以在\_\_setattr\_\_下还有self.attr的赋值操作就会出现无线递归的调用\_\_setattr\_\_的情况。自己实现\_\_setattr\_\_有很大风险，一般情况都还是继承object类的\_\_setattr\_\_方法。**

**class ClassA(object):**

**def \_\_init\_\_(self, classname):**

**self.classname = classname**

**def \_\_setattr\_\_(self, name, value):**

**# self.name = value # 如果还这样调用会出现无限递归的情况**

**print('invoke \_\_setattr\_\_')**

**insA = ClassA('ClassA') # \_\_init\_\_中的self.classname调用\_\_setattr\_\_。**

**# invoke \_\_setattr\_\_**

**print(insA.\_\_dict\_\_)**

**# {}**

**insA.tag = 'insA'**

**# invoke \_\_setattr\_\_**

**print(insA.\_\_dict\_\_)**

**# {}**

19，django中如何读取和保存session，整个session的运行机制是什么？

**平时我们往session中添加的数据实际上保存在request.session.\_session这个字典中的。**

**request.session实际上就是一个字典，所以可以像访问字典一样访问session里面的数据，通过遍历session里面的数据，可以证明上面那个观点：**

**[python]** [**view plain**](http://blog.csdn.net/shanliangliuxing/article/details/8827381)[**copy**](http://blog.csdn.net/shanliangliuxing/article/details/8827381)

1. **session\_tmp = request.session**
2. **for key, value in session\_tmp.items():**
3. **print key + ' : ' + str(value)**

**比方说下面往session里面添加一条记录：request.session['username'] = 'admin'**

**实际上这条数据保存在request.session.\_session这个字典中，用的时候直接像下面这样用就行了：**

1. **request.session['username']**

20，下面代码执行的结果

def f(x,l=[])

for i in range(x):

l.append(i\*i)

print(l)

f(2)

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

f(3)

**[0, 1]**

**[3, 2, 1, 0, 1, 4]**

**[0, 1, 0, 1, 4]**

def f(x,l=[]):

print id(l)

for i in range(x):

l.append(i\*i)

print id(l)

print '#########'

f(2)

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

f(3)

**39443912**

**39443912**

**#########**

**39993928**

**39993928**

**#########**

**39443912**

**39443912**

**#########**

21，请写出一段python代码实现删除一个list里面的重复

**l1 = ['b','c','d','b','c','a','a']   
l2 = list(set(l1))   
print l2**

22，python代码得到列表list的交集和差集

1. **获取两个list 的交集**

**a=[2,3,4,5]**

**b=[2,5,8]**

**tmp = [val for val in a if val in b]**

**print tmp**

**#[2, 5]**

1. **获取两个list 的并集**

**print list(set(a).union(set(b)))**

1. **获取两个 list 的差集**

**print list(set(b).difference(set(a))) # b中有而a中没有的**

23，\_\_new\_\_和\_\_init\_\_的区别

一、\_\_init\_\_ 方法是什么？

**使用Python写过面向对象的代码的同学，可能对 \_\_init\_\_ 方法已经非常熟悉了，\_\_init\_\_ 方法通常用在初始化一个类实例的时候。例如：**

**class Person(object):**

**"""Silly Person"""**

**def \_\_init\_\_(self, name, age):**

**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**

**这样便是\_\_init\_\_最普通的用法了。但\_\_init\_\_其实不是实例化一个类的时候第一个被调用 的方法。当使用 Persion(name, age) 这样的表达式来实例化一个类时，最先被调用的方法 其实是 \_\_new\_\_ 方法**

# ****二、\_\_new\_\_ 方法是什么？****

\_**\_new\_\_方法接受的参数虽然也是和\_\_init\_\_一样，但\_\_init\_\_是在类实例创建之后调用，而 \_\_new\_\_方法正是创建这个类实例的方法。**

**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**

**piglei@macbook-pro:blog$ python new\_and\_init.py**

**\_\_new\_\_ called.**

**\_\_init\_\_ called.**

**<Person: piglei(24)>**

**1.p = Person(name, age)  
2.首先执行使用name和age参数来执行Person类的\_\_new\_\_方法，这个\_\_new\_\_方法会 返回Person类的一个实例（通常情况下是使用 super(Persion, cls).\_\_new\_\_(cls, … …) 这样的方式），  
3.然后利用这个实例来调用类的\_\_init\_\_方法，上一步里面\_\_new\_\_产生的实例也就是 \_\_init\_\_里面的的 self  
所以，\_\_init\_\_ 和 \_\_new\_\_ 最主要的区别在于：  
1.\_\_init\_\_ 通常用于初始化一个新实例，控制这个初始化的过程，比如添加一些属性， 做一些额外的操作，发生在类实例被创建完以后。它是实例级别的方法。  
2.\_\_new\_\_ 通常用于控制生成一个新实例的过程。它是类级别的方法。  
但是说了这么多，\_\_new\_\_最通常的用法是什么呢，我们什么时候需要\_\_new\_\_？**

三、\_\_new\_\_ 的作用

**依照Python官方文档的说法，\_\_new\_\_方法主要是当你继承一些不可变的class时(比如int, str, tuple)， 提供给你一个自定义这些类的实例化过程的途径。还有就是实现自定义的metaclass。  
首先我们来看一下第一个功能，具体我们可以用int来作为一个例子：  
假如我们需要一个永远都是正数的整数类型，通过集成int，我们可能会写出这样的代码。**

**class PositiveInteger(int):**

**def \_\_init\_\_(self, value):**

**super(PositiveInteger, self).\_\_init\_\_(self, abs(value))**

**i = PositiveInteger(-3)**

**print i**

**但运行后会发现，结果根本不是我们想的那样，我们任然得到了-3。这是因为对于int这种 不可变的对象，我们只有重载它的\_\_new\_\_方法才能起到自定义的作用。  
这是修改后的代码：**

**通过重载\_\_new\_\_方法，我们实现了需要的功能。  
另外一个作用，关于自定义metaclass。其实我最早接触\_\_new\_\_的时候，就是因为需要自定义 metaclass，但鉴于篇幅原因，我们下次再来讲python中的metaclass和\_\_new\_\_的关系**

四、用\_\_new\_\_来实现单例

**事实上，当我们理解了\_\_new\_\_方法后，我们还可以利用它来做一些其他有趣的事情，比如实现 设计模式中的 单例模式(singleton) 。  
因为类每一次实例化后产生的过程都是通过\_\_new\_\_来控制的，所以通过重载\_\_new\_\_方法，我们 可以很简单的实现单例模式。**

**class Singleton(object):**

**def \_\_new\_\_(cls):**

**# 关键在于这，每一次实例化的时候，我们都只会返回这同一个instance对象**

**if not hasattr(cls, 'instance'):**

**cls.instance = super(Singleton, cls).\_\_new\_\_(cls)**

**return cls.instance**

**obj1 = Singleton()**

**obj2 = Singleton()**

**obj1.attr1 = 'value1'**

**print obj1.attr1, obj2.attr1**

**print obj1 is obj2**

**输出结果：**



24，python如何生成随机数

**Import random**

**Random.randint(a,b)随机生成一个a到b区间的数，**

**Random.random 生成一个0到1之间的随机浮点数**

25，python中search和match区别

**这是正则表达式里面的函数**：

Span：表示[]前闭后开

1. **match()函数只检测RE是不是在string的开始位置匹配，search()会扫描整个string查找匹配；**
2. **也就是说match()只有在0位置匹配成功的话才有返回，如果不是开始位置匹配成功的话，match()就返回none。**
3. **例如：**

**print(re.match('super', 'superstition').span())   会返回(0, 5)**

**而print(re.match('super', 'insuperable'))   则返回None**

1. **search()会扫描整个字符串并返回第一个成功的匹配：**

**例如：print(re.search('super', 'superstition').span())返回(0, 5)**

**print(re.search('super', 'insuperable').span())返回(2, 7)**

1. **其中span函数定义如下，返回位置信息：**

**span([group]):**

**返回(start(group), end(group))。**

26，python函数中，\*args,\*\*kwargs这个个函数什么意思？为何要使用他们？

**不定长参数：\*args没有key值，kwargs有key值**

27，有两个序列a，b，大小都为n，序列元素的值任意整形数，无序；

要求：通过交换a，b中的元素，使[序列a元素的和]与[序列b元素的和]之间的差最小

28，MyISAM和InnoDB的区别？

**InnoDB可以添加事务，**两**种类型最主要的差别就是Innodb 支持事务处理与外键和行级锁。而MyISAM不支持.所以MyISAM往往就容易被人认为只适合在小项目中使用**

29，如何列出表‘test’内name域值不为‘tecmint’，web\_address域值为‘test.com’的所有数据？

30，平衡点问题

假如一个数组的元素，其前面的部分等于后面的部分，那么这个点的程序就是平衡点

列表 numbers = [1,3,5,7,8,25,4,20],25前面的总和为24,25后面的总和为24，那么25就是这个平衡点。

编写程序，寻找并返回任意一个列表的平衡点

31，用python语言实现

一个数组，其中该数组中的每个元素均为字符串，删除该数组中空白字符串

32，什么是异步调用？并用一段简单的代码实现异步调用

33，给定一个无序的数值序列，找出其中第N大的数值

34，如何衡量两段文本的相似度

35，当你在爬取数据，遇到反扒时，解决思路是什么？

36，写出20个linux命令

37，如何理解python中字符串中的\字符

38，如何理解python中的深拷贝和浅拷贝

**Deepcopy是子类与父类一起拷贝，copy是只copy父类**

39，列表中有n个正整数范围在[0,1000]，请写编程对列表中的数据排序

**import random  
b = []  
def rand(n):  
 for i in range(n):  
 b.append(random.randint(0,1000))  
 return b  
d = rand(10)  
c = len(d)  
for j in range(c):  
 for k in range(j+1,c):  
 if d[j] > d[k]:  
 d[j],d[k] = d[k],d[j]  
print(d)**

40，列表中有n对不重复的正整数，随机取出一个数字后，编程快速找到取出的数字。如：[1,3,4,2,4,1,2,3]->[1,3,4,2,1,2,3]

41，请简述浏览器是如何获取一枚网页的

42，请写出可以一定程度判断网页是否变化的响应头

43，通过http服务器日志如何干掉恶意爬虫

44，简述图片验证码的工作流程

45，list和tuple的区别

**List可变，tuple不可变**。

46，list和dict两种结构的区别

**dict查找速度快。无论是10个还是10万个，速度都是一样的，但是代价是耗费的内存大。List相反，占用内存小，但是查找速度慢。这就好比是数组和链表的区别，数组并不知道要开辟多少空间，所以往往开始就会开辟一个大空间，但是直接通过下标查找速度快；而链表占用的空间小，但是查找的时候必须顺序的遍历导致速度很慢  
没有顺序。Dict是无顺序的，而List是有序的集合，所以不能用Dict来存储有序集合  
Key不可变，Value可变。一旦一个键值对加入dict后，它对应的key就不能再变了，但是Value是可以变化的。所以List不可以当做Dict的Key，但是可以作为Value:**

47，对list进行切片的用法

**取一个list的部分元素是非常常见的操作。比如，一个list如下：**

**>>> L = ['Adam', 'Lisa', 'Bart', 'Paul']**

**取前3个元素，应该怎么做？**

**对这种经常取指定索引范围的操作，用循环十分繁琐，因此，Python提供了切片（Slice）操作符，能大大简化这种操作。**

**对应上面的问题，取前3个元素，用一行代码就可以完成切片：**

**>>> L[0:3]**

**['Adam', 'Lisa', 'Bart']**

**L[0:3]表示，从索引0开始取，直到索引3为止，但不包括索引3。即索引0，1，2，正好是3个元素。**

**只用一个** : **，表示从头到尾：**

**>>> L[:]**

**['Adam', 'Lisa', 'Bart', 'Paul']**

**因此，L[:]实际上复制出了一个新list。**

**切片操作还可以指定第三个参数：**

**>>> L[::2]**

**['Adam', 'Bart']**

**第三个参数表示每N个取一个，上面的 L[::2] 会每两个元素取出一个来，也就是隔一个取一个。**

**把list换成tuple，切片操作完全相同，只是切片的结果也变成了tuple。**

48，什么是lambda函数？它有什么好处？

**匿名函数，比较简洁，代码量少**

**Python允许你定义一种单行的小函数。定义lambda函数的形式如下：labmda 参数：表达式lambda函数默认返回表达式的值。你也可以将其赋值给一个变量。lambda函数可以接受任意个参数，包括可选参数，但是表达式只有一个：  
>>> g = lambda x, y: x\*y  
>>> g(3,4)  
12  
>>> g = lambda x, y=0, z=0: x+y+z  
>>> g(1)  
1  
>>> g(3, 4, 7)  
14  
也能够直接使用lambda函数，不把它赋值给变量：  
>>> (lambda x,y=0,z=0:x+y+z)(3,5,6)  
14  
如果你的函数非常简单，只有一个表达式，不包含命令，可以考虑lambda函数。否则，你还是定义函数才对，毕竟函数没有这么多限制。**

49，利用yield关键字的好处

**yield是python中定义为生成器函数，其本质是封装了  \_\_iter\_\_和\_\_next\_\_方法的迭代器；**

**与return返回的区别：return只能返回一次值，函数就终止了，而yield能多次返回值，每次返回都会将函数暂停，下一次next会从上一次暂停的位置继续执行；**

50，如何利用python输出一个Fibonacci数列？

# -\*- coding:utf-8 -\*-

def fibs(num):

    a=b=1

    for i in range(num):

        yield a

        a,b=b,a+b

print list(fibs(10))

**i, j = 0, 1**

**while i < 10000:**

**print i,**

**i, j = j, i+j**

**def fib(num):  
 a = [0,1]  
 for i in range(2,num):  
 a.append(a[i-1]+a[i-2])  
 return a  
print(fib(10))**

51，介绍一下except的用法和作用？

**try:**

**pass**

**except BaseException as e:**

**print(e)**

**finally:**

**pass**

**捕获try except中间代码发生的异常，如果发生异常执行except的代码，不管是否发生异常都执行finally中的代码**

**except可以有0个或多个，如果有多个从上到下依次根据异常类型匹配，匹配某个Exception这执行对应的except中代码**

52，用python匹配HTML tag的时候，<.\*>和<.\*?>有什么区别？

**Import re**

**S = ‘<html><head><title>Title</title>’**

**Print(re.match(‘<.\*>’),s).group())**

**会返回一个匹配<html><head><title>Title</title>**

**而：**

**Import re**

**S = ‘<html><head><title>Title</title>’**

**Print(re.match(‘<.\*?>’,s).group())**

**<.\*>这种匹配称为贪心匹配<.\*?>称为非贪心匹配**

**53，利用正则表达式从一段HTML文本中提取多个字段值**

54，用两种方式按行读取一个csv文件

**第一种方式使用reader函数：第二种方法使用DictReader**

55，任意一种排序方式的代码实现（尽量选择时间复杂度是O（NIogN））

56，简述程序运行时间和时间复杂度的区别，并举例说明

57，简述数据库的锁机制

58，MySQL有几种存储引擎，简述各自特点，

**MYISAM与InnoDB**

**MYISAM表是独立于操作系统的，这说明可以轻松的将其从windows服务器移植到linux**

**InnoDB是一个健壮的事务型存储引擎**

**Memory使用memory存储引擎的出发点是速度。为了最快响应，采用逻辑存储介质是系统内存**

59，如何理解mysql主从库架构

60，写一个函数，遍历一个文件夹下的所有文件和子文件夹

深度遍历：

**import** os  
rootpath=**r"C:\Users\Tsinghua-yincheng\Desktop\YinchengDay6Down"**mystack=[]  
mystack.append([rootpath,0])*#rootpath,路径***while** len(mystack)!=0:  
 pathlist=mystack.pop()*#弹出的路径* num=pathlist[1] *#取出层次* headstr=**""  
 for** i **in** range(num):  
 headstr+=**"----"** *#每一层的前缀，显示一层* **if** os.path.isdir(pathlist[0]):  
 print(headstr,**"文件夹 "**,pathlist[0])  
  
 num+=1  
 headstr+=**"----"** *#层次 +1，* filenamelist=os.listdir(pathlist[0]) *#列举文件夹下所有文件名* **for** filename **in** filenamelist:*#遍历所有文件名* filepath=os.path.join(pathlist[0],filename)*#取得绝对路径* **if** os.path.isdir(filepath):  
 mystack.append([filepath,num]) *#文件夹，压入栈* **else**:  
 print(headstr,**"文件 "**,filepath)  
  
 **else**:  
 print(headstr,**"文件 "**,pathlist[0])

广度遍历：

**import** os  
**import** collections  
  
path=**r"C:\Users\Tsinghua-yincheng\Desktop\YinchengDay6Down"**myqueue=collections.deque([])*#空队列*myqueue.append(path)  
  
**while** len(myqueue)!=0:  
 path=myqueue.popleft()*#取出路径* filelist=os.listdir(path)*#遍历文件夹，如果是文件就是空集合* **for** filename **in** filelist:  
 filepath=os.path.join(path,filename)*#绝对路径* **if** os.path.isdir(filepath):  
 print(**"文件夹"**,filepath)  
 myqueue.append(filepath)  
 **else**:  
 print(**"文件"**,filepath)

61，请简述多线程爬虫的实现方法，使用的包和相关用途，还有需要注意的问题。

62，自定义一个浏览器兼容的事件绑定方法需要注意那些问题？

63，设计一个布局方案，是的页面在pc端和pad端显示为一行三列，在手机端为一列三行。

64，请解释JSONP的工作原理

65，display和position都有那些值？分别列举一下这些值的用途

Display:none、inline、block、inline-block

Position：static absolute relative fixed

66，地址栏输入一个地址后，输入回车，描述一下这时开始，浏览器做了什么？

67，创建一个JS类，模拟实现方法的重载

**Var showmessage = function(name,value,id){**

**If(id !=’undefined’){**

**Alert(name +value +id);**

**}**

**Else if(value !=’unidefined’){**

**Alert(name +value);**

**}**

**Else{**

**Alert(name);**

**}**

**}**

**Showmessage(‘哈哈’)；**

**Showmessage(‘哈哈’，‘？？’)；**

**Showmessage(‘哈哈’,”??”,12345);**

68，用JS实现一个标准的排序算法，对某个数字数组进行由低到高的排序

**Funcction(array){**

**Var i=0,**

**Len = array.lengthh,**

**J,d**

**For(;i<len;i++){**

**For(j=0;j<len;j++){**

**If(array[i]<array[j]){**

**D = array[j];**

**Array[j] =array[i];**

**Array[i] =d;**

**}**

**}**

**}**

**Return array;**

**}**

69，用html，css和js模拟实现一个下拉框，使得下拉框在各个浏览器下的样式和行为完全一致，说出你的设计方案，并且重点说明功能设计时要考虑的因素

**1.响应式布局**

**1.Meta标签定义**

**2.使用Media Queries适配对应样式**

**2.响应式内容**

**1.响应式图片**

70，分别写出python中基本元祖，列表，字典，集合的操作

定义：

新增：

更改：

删除：