Python 基本面试题|深入解答

分享给大家的 11 道 Python 面试题，好多小伙伴都很积极的去思考分析，给我留言的同学非常多,非常欣慰有这么多好学的小伙伴，大家一起学习，一起加油，把 Python 学好,今天我就把 11 道面试题细细解答一下

1. 单引号，双引号，三引号的区别

分别阐述 3 种引号用的场景和区别

1),单引号和双引号主要用来表示字符串

比如:

单引号:'python'

双引号:"python"

2).三引号

三单引号:'''python ''',也可以表示字符串一般用来输入多行文本,或者用于大段的注释

三双引号："""python"""，一般用在类里面,用来注释类,这样省的写文档,直接用类的对象__doc__访问获得文档

区别:

若你的字符串里面本身包含单引号,必须用双引号

比如:"can't find the log\n"

2. Python 的参数传递是值传递还是引用传递

举例说明 Python 函数参数传递的几种形式，并说明函数传参是值传递还是引用传递

1).Python 的参数传递有：

位置参数

默认参数，

可变参数,

关键字参数

2).函数的传值到底是值传递还是引用传递，要分情况

a.不可变参数用值传递：

像整数和字符串这样的不可变对象，是通过拷贝进行传递的，因为你无论如何都不可能在原处改变不可变对象

b.可变参数是用引用传递的

比如像列表，字典这样的对象是通过引用传递，和 C 语言里面的用指针传递数组很相似，可变对象能在函数内部改变.

3.什么是 lambda 函数？它有什么好处?

举例说明 lambda 的用法，并说明用 lambda 的优点

1).lambda 的用法：

lambda 是匿名函数，用法如下:lambda arg1,arg2..argN:expression using args

2).优点

lambda 能和 def 做同样种类的工作，特别是对于那些逻辑简单的函数，直接用 lambda 会更简洁，而且省去取函数名的麻烦(给函数取名是个技术活)

4.字符串格式化:%和.format 的区别

字符串的 format 函数非常灵活，很强大,**可以接受的参数不限个数，并且位置可以不按顺序**，而且有较为强大的格式限定符(比如:填充，对齐,精度等)

5.Python 是如何进行内存管理的

1).对象的引用计数机制

Python 内部使用引用计数，来保持追踪内存中的对象，所有对象都有引用计数。

引用计数增加的情况：

一个对象分配一个新名称

将其放入一个容器中（如列表、元组或字典）

引用计数减少的情况：

使用 del 语句对对象别名显示的销毁

引用超出作用域或被重新赋值

2).垃圾回收

当一个对象的引用计数归零时，它将被垃圾收集机制处理掉。

3).内存池机制

Python 提供了对内存的垃圾收集机制，但是它将不用的内存放到内存池而不是返回给操作系统:

Pymalloc 机制：为了加速 Python 的执行效率，Python 引入了一个内存池机制，用于管理对小块内存的申请和释放。

对于 Python 对象，如整数，浮点数和 List，都有其独立的私有内存池，对象间不共享他们的内存池。也就是说如果你分配又释放了大量的整数，用于缓存这些整数的内存就不能再分配给浮点数。

6. 写一个函数，输入一个字符串，返回倒序排列的结果

输入: string_reverse(‘abcdef’)，返回: ‘fedcba’,写出你能想到的多种方法

1).利用字符串本身的翻转

```
def string_reverse1(text='abcdef'):

return text[::-1]
```

2).把字符串变成列表，用列表的 reverse 函数

3).新建一个列表，从后往前取

4).利用双向列表 deque 中的 extendleft 函数

5).递归

7. 按升序合并如下两个 list，并去除重复的元素

list1 = [2, 3, 8, 4, 9, 5, 6]

list2 = [5, 6, 10, 17, 11, 2]

1).最简单的方法用 set

list3=list1+list2

print sorted(list(set(list3)))

2).递归

先选一个中间数，然后一边是小的数字，一边是大的数字，然后再循环递归，排完序(是不是想起了 c 里面的冒泡)

8. 以下的代码的输出将是什么？说出你的答案并解释

class Parent(object):

x = 1

class Child1(Parent):

```
pass

class Child2(Parent):

pass

print Parent.x, Child1.x, Child2.x

Child1.x = 2

print Parent.x, Child1.x, Child2.x

Parent.x = 3

print Parent.x, Child1.x, Child2.x

>>

1 1 1

1 2 1

3 2 3
```

解答:

使你困惑或是惊奇的是关于最后一行的输出是 3 2 3 而不是 3 2 1。为什么改变了 Parent.x 的值还会改变 Child2.x 的值，但是同时 Child1.x 值却没有改变？

这个答案的关键是，在 Python 中，类变量在内部是作为字典处理的。如果一个变量的名字没有在当前类的字典中发现，将搜索祖先类（比如父类）直到被引用的变量名被找到.

首先，在父类中设置 x = 1 会使得类变量 x 在引用该类和其任何子类中的值为 1。**这就是因为第一个 print 语句的输出是 1 1 1**

然后，如果任何它的子类重写了该值（例如，我们执行语句 Child1.x = 2）该值仅仅在子类中被改变。**这就是为什么第二个 print 语句的输出是 1 2 1**

最后，如果该值在父类中被改变（例如，我们执行语句 Parent.x = 3），这个**改变会影响到任何未重写该值的子类当中的值**（在这个示例中被影响的子类是 Child2）。这就是为什么第三个 print 输出是 3 2 3

9. 下面的代码会不会报错

```
list = ['a', 'b', 'c', 'd', 'e']

print list[10:]
```

不会报错，而且会输出一个 []，并且不会导致一个 IndexError

解答:

当试图访问一个超过列表索引值的成员将导致 IndexError（比如访问以上列表的 list[10]）。尽管如此，试图访问一个列表的以超出列表长度数作为开始索引的切片将不会导致 IndexError，**并且将仅仅返回一个空列表**

一个讨厌的小问题是它会导致出现 bug ，并且这个问题是难以追踪的，因为它在运行时不会引发错误，吐血啊~~

10.说出下面 list1,list2,list3 的输出值

```
def extendList(val, list=[]):

list.append(val)

return list

list1 = extendList(10)

list2 = extendList(123,[])

list3 = extendList('a')

print "list1 = %s" % list1

print "list2 = %s" % list2

print "list3 = %s" % list3

>>

list1 = [10, 'a']

list2 = [123]

list3 = [10, 'a']
```

许多人会错误的认为 list1 应该等于 [10] 以及 list3 应该等于 ['a']。认为 list 的参数会在 extendList 每次被调用的时候会被设置成它的默认值 []。

尽管如此，实际发生的事情是，新的默认列表仅仅只在函数被定义时创建一次。随后当 extendList 没有被指定的列表参数调用的时候，其使用的是同一个列表。这就是为什么当函数被定义的时候，表达式是用默认参数被计算，而不是它被调用的时候。

因此，list1 和 list3 是操作的相同的列表。而 list2 是操作的它创建的独立的列表（通过传递它自己的空列表作为 list 参数的值）

所以这一点一定要切记切记.下面我们把 list 置为 None 就可以避免一些麻烦了

11. 写出你认为最 Pythonic 的代码

Pythonic 编程风格是 Python 的一种追求的风格，精髓就是追求直观，简洁而容易读.

下面是一些比较好的例子

1).交互变量

非 **Pythonic**

```
temp = a

a = b

b = temp
```

pythonic:

```
a,b=b,a
```

2).判断其值真假

```
name = 'Tim'

langs = ['AS3', 'Lua', 'C']

info = {'name': 'Tim', 'sex': 'Male', 'age':23 }
```

非 **Pythonic**

```
if name != '' and len(langs) > 0 and info != {}:
```

print('All True!')

pythonic:

if name and langs and info:

print('All True!')

3).列表推导式

[x for x in range(1,100) if x%2==0]

4).zip 创建键值对

keys = ['Name', 'Sex', 'Age']

values = ['Jack', 'Male', 23]

dict(zip(keys,values))

。

问题 1

到底什么是 Python？你可以在回答中与其他技术进行对比（也鼓励这样做）。

答案

下面是一些关键点：

- Python 是一种解释型语言。这就是说，与 C 语言和 C 的衍生语言不同，Python 代码在运行之前不需要编译。其他解释型语言还包括 PHP 和 Ruby。
- Python 是动态类型语言，指的是你在声明变量时，不需要说明变量的类型。你可以直接编写类似 x=111 和 x="I'm a string"这样的代码，程序不会报错。
- Python 非常适合面向对象的编程（OOP），因为它支持通过组合（composition）与继承（inheritance）的方式定义类（class）。Python 中没有访问说明符（access specifier，类似 C++中的 public 和 private），这么设计的依据是“大家都是成年人了”。
- 在 Python 语言中，函数是第一类对象（first-class objects）。这指的是它们可以被指定给变量，函数既能返回函数类型，也可以接受函数作为输入。类（class）也是第一类对象。
- Python 代码编写快，但是运行速度比编译语言通常要慢。好在 Python 允许加入基于 C 语言编写的扩展，因此我们能够优化代码，消除瓶颈，这点通常是可以实现的。numpy 就是一个很好地例子，它的运行速度真的非常快，因为很多算术运算其实并不是通过 Python 实现的。
- Python 用途非常广泛——网络应用，自动化，科学建模，大数据应用，等等。它也常被用作“胶水语言”，帮助其他语言和组件改善运行状况。
- Python 让困难的事情变得容易，因此程序员可以专注于算法和数据结构的设计，而不用处理底层的细节。

为什么提这个问题：

如果你应聘的是一个 Python 开发岗位，你就应该知道这是门什么样的语言，以及它为什么这么酷。以及它哪里不好。

问题 2

补充缺失的代码

```
def print_directory_contents(sPath):
    """    这个函数接受文件夹的名称作为输入参数，      返回该文件夹中文件的路径，      以及其包含文件夹中文件的路径。
    """
    # 补充代码
```

答案

```
def print_directory_contents(sPath):
    import os
    for sChild in os.listdir(sPath):
        sChildPath = os.path.join(sPath,sChild)
        if os.path.isdir(sChildPath):
            print_directory_contents(sChildPath)
        else:
            print sChildPath
```

特别要注意以下几点：

- 命名规范要统一。如果样本代码中能够看出命名规范，遵循其已有的规范。
- 递归函数需要递归并终止。确保你明白其中的原理，否则你将面临无休无止的调用栈（callstack）。
- 我们使用 os 模块与操作系统进行交互，同时做到交互方式是可以跨平台的。你可以把代码写成 sChildPath = sPath + '/' + sChild，但是这个在 Windows 系统上会出错。
- 熟悉基础模块是非常有价值的，但是别想破脑袋都背下来，记住 Google 是你工作中的良师益友。
- 如果你不明白代码的预期功能，就大胆提问。
- 坚持 KISS 原则！保持简单，不过脑子就能懂！

为什么提这个问题：

- 说明面试者对与操作系统交互的基础知识
- 递归真是太好用啦

问题 3

阅读下面的代码，写出 A0，A1 至 An 的最终值。

```
A0 = dict(zip(('a','b','c','d','e'),(1,2,3,4,5)))A1 = range(10)A2 = [i for i in A1 if i in A0]A3 = [A0[s] for s in A0]A4 = [i for i in A1 if i in A3]A5 = {i:i*i for i in A1}A6 = [[i,i*i] for i in A1]
```

答案

```
A0 = {'a': 1, 'c': 3, 'b': 2, 'e': 5, 'd': 4}A1 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]A2 = []A3 = [1, 3, 2, 5, 4]A4 = [1, 2, 3, 4, 5]A5 = {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}A6 = [[0, 0], [1, 1], [2, 4], [3, 9], [4, 16], [5, 25], [6, 36], [7, 49], [8, 64], [9, 81]]
```

为什么提这个问题：

- 列表解析（list comprehension）十分节约时间，对很多人来说也是一个大的学习障碍。
- 如果你读懂了这些代码，就很可能可以写下正确地值。
- 其中部分代码故意写的怪怪的。因为你共事的人之中也会有怪人。

问题 4

Python 和多线程（multi-threading）。这是个好主意码？列举一些让 Python 代码以并行方式运行的方法。

答案

Python 并不支持真正意义上的多线程。Python 中提供了多线程包，但是如果你想通过多线程提高代码的速度，使用多线程包并不是个好主意。Python 中有一个被称为 Global Interpreter Lock（GIL）的东西，它会确保任何时候你的多个线程中，只有一个被执行。线程的执行速度非常之快，会让你误以为线程是并行执行的，但是实际上都是轮流执行。经过 GIL 这一道关卡处理，会增加执行的开销。这意味着，如果你想提高代码的运行速度，使用 threading 包并不是一个很好的方法。

不过还是有很多理由促使我们使用 threading 包的。如果你想同时执行一些任务，而且不考虑效率问题，那么使用这个包是完全没问题的，而且也很方便。但是大部分情况下，并不是这么一回事，你会希望把多线程的部分外包给操作系统完成（通过开启多个进程），或者是某些调用你的 Python 代码的外部程序(例如 Spark 或 Hadoop)，又或者是你的 Python 代码调用的其他代码（例如，你可以在 Python 中调用 C 函数，用于处理开销较大的多线程工作）。

为什么提这个问题

因为 GIL 就是个混账东西（A-hole）。很多人花费大量的时间，试图寻找自己多线程代码中的瓶颈，直到他们明白 GIL 的存在。

问题 5

你如何管理不同版本的代码？

答案：

版本管理！被问到这个问题的时候，你应该要表现得很兴奋，甚至告诉他们你是如何使用 Git（或是其他你最喜欢的工具）追踪自己和奶奶的书信往来。我偏向于使用 Git 作为版本控制系统（VCS），但还有其他的选择，比如 subversion（SVN）。

为什么提这个问题：

因为没有版本控制的代码，就像没有杯子的咖啡。有时候我们需要写一些一次性的、可以随手扔掉的脚本，这种情况下不作版本控制没关系。但是如果你面对的是大量的代码，使用版本控制系统是有利的。版本控制能够帮你追踪谁对代码库做了什么操作；发现新引入了什么 bug；管理你的软件的不同版本和发行版；在团队成员中分享源代码；部署及其他自动化处

理。它能让你回滚到出现问题之前的版本，单凭这点就特别棒了。还有其他的好功能。怎么一个棒字了得！

问题 6

下面代码会输出什么：

```
def f(x,l=[]):
    for i in range(x):
        l.append(i*i)
    print l
f(2)f(3,[3,2,1])f(3)
```

答案：

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

呃？

第一个函数调用十分明显，for 循环先后将 0 和 1 添加至了空列表 l 中。l 是变量的名字，指向内存中存储的一个列表。

第二个函数调用在一块新的内存中创建了新的列表。l 这时指向了新生成的列表。之后再往新列表中添加 0、1、2 和 4。很棒吧。

第三个函数调用的结果就有些奇怪了。它使用了之前内存地址中存储的旧列表。这就是为什么它的前两个元素是 0 和 1 了。

不明白的话就试着运行下面的代码吧：

```
l_mem = []
l = l_mem           # the first callfor i in range(2):
    l.append(i*i)
print l             # [0, 1]
l = [3,2,1]         # the second callfor i in range(3):
    l.append(i*i)
print l             # [3, 2, 1, 0, 1, 4]
l = l_mem           # the third callfor i in range(3):
    l.append(i*i)
print l             # [0, 1, 0, 1, 4]
```

问题 7

“猴子补丁”（monkey patching）指的是什么？这种做法好吗？

答案：

“猴子补丁”就是指，在函数或对象已经定义之后，再去改变它们的行为。

举个例子：

```
import datetimedatetime.datetime.now = lambda: datetime.datetime(2012, 12, 12)
```

大部分情况下，这是种很不好的做法 - 因为函数在代码库中的行为最好是都保持一致。打"猴子补丁"的原因可能是为了测试。mock 包对实现这个目的很有帮助。

为什么提这个问题？

答对这个问题说明你对单元测试的方法有一定了解。你如果提到要避免"猴子补丁"，可以说明你不是那种喜欢花里胡哨代码的程序员（公司里就有这种人，跟他们共事真是糟糕透了），而是更注重可维护性。还记得 KISS 原则码？答对这个问题还说明你明白一些 Python 底层运作的方式，函数实际是如何存储、调用等等。

另外：如果你没读过 mock 模块的话，真的值得花时间读一读。这个模块非常有用。

问题 8

这两个参数是什么意思：*args，**kwargs？我们为什么要使用它们？

答案

如果我们不确定要往函数中传入多少个参数，或者我们想往函数中以列表和元组的形式传参数时，那就使要用*args；

如果我们不知道要往函数中传入多少个关键词参数，或者想传入字典的值作为关键词参数时，那就要使用**kwargs。

args 和 kwargs 这两个标识符是约定俗成的用法，你当然还可以用*bob 和**billy，但是这样就并不太妥。

下面是具体的示例：

```
def f(*args,**kwargs): print args, kwargs
l = [1,2,3]t = (4,5,6)d = {'a':7,'b':8,'c':9}
f()f(1,2,3)                    # (1, 2, 3) {}f(1,2,3,"groovy")
# (1, 2, 3, 'groovy') {}f(a=1,b=2,c=3)                # () {'a': 1, 'c':
3, 'b': 2}f(a=1,b=2,c=3,zzz="hi")     # () {'a': 1, 'c': 3, 'b': 2, 'zzz':
'hi'}f(1,2,3,a=1,b=2,c=3)         # (1, 2, 3) {'a': 1, 'c': 3, 'b': 2}
f(*l,**d)                    # (1, 2, 3) {'a': 7, 'c': 9, 'b': 8}f(*t,**d)
# (4, 5, 6) {'a': 7, 'c': 9, 'b': 8}f(1,2,*t)                    # (1, 2,
4, 5, 6) {}f(q="winning",**d)          # () {'a': 7, 'q': 'winning', 'c':
9, 'b': 8}f(1,2,*t,q="winning",**d)   # (1, 2, 4, 5, 6) {'a': 7, 'q':
'winning', 'c': 9, 'b': 8}
def f2(arg1,arg2,*args,**kwargs): print arg1,arg2, args, kwargs
f2(1,2,3)                    # 1 2 (3,) {}f2(1,2,3,"groovy")
# 1 2 (3, 'groovy') {}f2(arg1=1,arg2=2,c=3)           # 1 2 () {'c':
3}f2(arg1=1,arg2=2,c=3,zzz="hi")  # 1 2 () {'c': 3, 'zzz':
'hi'}f2(1,2,3,a=1,b=2,c=3)           # 1 2 (3,) {'a': 1, 'c': 3, 'b': 2}
```

f2(*l,**d)　　# 1 2 (3,) {'a': 7, 'c': 9, 'b': 8}f2(*t,**d)　　# 4 5 (6,) {'a': 7, 'c': 9, 'b': 8}f2(1,2,*t)　　# 1 2 (4, 5, 6) {}f2(1,1,q="winning",**d)　　# 1 1 () {'a': 7, 'q': 'winning', 'c': 9, 'b': 8}f2(1,2,*t,q="winning",**d)　　# 1 2 (4, 5, 6) {'a': 7, 'q': 'winning', 'c': 9, 'b': 8}

为什么提这个问题?

有时候，我们需要往函数中传入未知个数的参数或关键词参数。有时候，我们也希望把参数或关键词参数储存起来，以备以后使用。有时候，仅仅是为了节省时间。

问题 9

下面这些是什么意思：@classmethod, @staticmethod, @property?

回答背景知识

这些都是装饰器（decorator）。装饰器是一种特殊的函数，要么接受函数作为输入参数，并返回一个函数，要么接受一个类作为输入参数，并返回一个类。

@标记是语法糖（syntactic sugar），可以让你以简单易读得方式装饰目标对象。

```
@my_decoratordef my_func(stuff):
    do_thingsIs equivalent to
def my_func(stuff):
    do_things
my_func = my_decorator(my_func)
```

你可以在本网站上找到介绍装饰器工作原理的教材。

真正的答案

@classmethod, @staticmethod 和@property 这三个装饰器的使用对象是在类中定义的函数。下面的例子展示了它们的用法和行为：

```
class MyClass(object):
    def __init__(self):
        self._some_property = "properties are nice"
        self._some_other_property = "VERY nice"
    def normal_method(*args,**kwargs):
        print "calling normal_method({0},{1})".format(args,kwargs)
    @classmethod
    def class_method(*args,**kwargs):
        print "calling class_method({0},{1})".format(args,kwargs)
    @staticmethod
    def static_method(*args,**kwargs):
        print "calling static_method({0},{1})".format(args,kwargs)
    @property
    def some_property(self,*args,**kwargs):
```

```
        print "calling some_property
getter({0},{1},{2})".format(self,args,kwargs)
        return self._some_property
    @some_property.setter
    def some_property(self,*args,**kwargs):
        print "calling some_property
setter({0},{1},{2})".format(self,args,kwargs)
        self._some_property = args[0]
    @property
    def some_other_property(self,*args,**kwargs):
        print "calling some_other_property
getter({0},{1},{2})".format(self,args,kwargs)
        return self._some_other_property
o = MyClass()# 未装饰的方法还是正常的行为方式，需要当前的类实例（self）作为第一个参数。
o.normal_method # <bound method MyClass.normal_method of
<__main__.MyClass instance at 0x7fdd2537ea28>>
o.normal_method() # normal_method((<__main__.MyClass instance at
0x7fdd2537ea28>,),{})
o.normal_method(1,2,x=3,y=4) # normal_method((<__main__.MyClass
instance at 0x7fdd2537ea28>, 1, 2),{'y': 4, 'x': 3})
# 类方法的第一个参数永远是该类
o.class_method# <bound method classobj.class_method of <class
__main__.MyClass at 0x7fdd2536a390>>
o.class_method()# class_method((<class __main__.MyClass at
0x7fdd2536a390>,),{})
o.class_method(1,2,x=3,y=4)# class_method((<class __main__.MyClass at
0x7fdd2536a390>, 1, 2),{'y': 4, 'x': 3})
# 静态方法（static method）中除了你调用时传入的参数以外，没有其他的参数。
o.static_method# <function static_method at 0x7fdd25375848>
o.static_method()# static_method((),{})
o.static_method(1,2,x=3,y=4)# static_method((1, 2),{'y': 4, 'x': 3})
# @property 是实现 getter 和 setter 方法的一种方式。直接调用它们是错误的。
# “只读”属性可以通过只定义 getter 方法，不定义 setter 方法实现。
o.some_property# 调用 some_property 的 getter(<__main__.MyClass
instance at 0x7fb2b70877e8>,(),{})# 'properties are nice'# “属性”是很好的功能
o.some_property()# calling some_property getter(<__main__.MyClass
instance at 0x7fb2b70877e8>,(),{})# Traceback (most recent call last):#
File "<stdin>", line 1, in <module># TypeError: 'str' object is not
callable
```

```
o.some_other_property# calling some_other_property
getter(<__main__.MyClass instance at 0x7fb2b70877e8>,(),{})# 'VERY
nice'
# o.some_other_property()# calling some_other_property
getter(<__main__.MyClass instance at 0x7fb2b70877e8>,(),{})# Traceback
(most recent call last):#   File "<stdin>", line 1, in <module>#
TypeError: 'str' object is not callable
o.some_property = "groovy"# calling some_property
setter(<__main__.MyClass object at 0x7fb2b7077890>,('groovy',),{})
o.some_property# calling some_property getter(<__main__.MyClass object
at 0x7fb2b7077890>,(),{})# 'groovy'
o.some_other_property = "very groovy"# Traceback (most recent call
last):#   File "<stdin>", line 1, in <module># AttributeError: can't set
attribute
o.some_other_property# calling some_other_property
getter(<__main__.MyClass object at 0x7fb2b7077890>,(),{})
```

问题 10

阅读下面的代码，它的输出结果是什么？

```
class A(object):
    def go(self):
        print "go A go!"
    def stop(self):
        print "stop A stop!"
    def pause(self):
        raise Exception("Not Implemented")
class B(A):
    def go(self):
        super(B, self).go()
        print "go B go!"
class C(A):
    def go(self):
        super(C, self).go()
        print "go C go!"
    def stop(self):
        super(C, self).stop()
        print "stop C stop!"
class D(B,C):
    def go(self):
        super(D, self).go()
        print "go D go!"
    def stop(self):
        super(D, self).stop()
```

```
        print "stop D stop!"
    def pause(self):
        print "wait D wait!"
class E(B,C): pass
a = A()b = B()c = C()d = D()e = E()
# 说明下列代码的输出结果
a.go()b.go()c.go()d.go()e.go()
a.stop()b.stop()c.stop()d.stop()e.stop()
a.pause()b.pause()c.pause()d.pause()e.pause()
```

答案

输出结果以注释的形式表示：

```
a.go()# go A go!
b.go()# go A go!# go B go!
c.go()# go A go!# go C go!
d.go()# go A go!# go C go!# go B go!# go D go!
e.go()# go A go!# go C go!# go B go!
a.stop()# stop A stop!
b.stop()# stop A stop!
c.stop()# stop A stop!# stop C stop!
d.stop()# stop A stop!# stop C stop!# stop D stop!
e.stop()# stop A stop!
a.pause()# ... Exception: Not Implemented
b.pause()# ... Exception: Not Implemented
c.pause()# ... Exception: Not Implemented
d.pause()# wait D wait!
e.pause()# ...Exception: Not Implemented
```

为什么提这个问题?

因为面向对象的编程真的真的很重要。不骗你。答对这道问题说明你理解了继承和 Python 中 super 函数的用法。

问题 11

阅读下面的代码，它的输出结果是什么？

```
class Node(object):
    def __init__(self,sName):
        self._lChildren = []
        self.sName = sName
    def __repr__(self):
        return "<Node '{}'>".format(self.sName)
    def append(self,*args,**kwargs):
        self._lChildren.append(*args,**kwargs)
```

```
    def print_all_1(self):
        print self
        for oChild in self._lChildren:
            oChild.print_all_1()
    def print_all_2(self):
        def gen(o):
            lAll = [o,]
            while lAll:
                oNext = lAll.pop(0)
                lAll.extend(oNext._lChildren)
                yield oNext
        for oNode in gen(self):
            print oNode
oRoot = Node("root")oChild1 = Node("child1")oChild2 =
Node("child2")oChild3 = Node("child3")oChild4 = Node("child4")oChild5 =
Node("child5")oChild6 = Node("child6")oChild7 = Node("child7")oChild8 =
Node("child8")oChild9 = Node("child9")oChild10 = Node("child10")
oRoot.append(oChild1)oRoot.append(oChild2)oRoot.append(oChild3)oChild
1.append(oChild4)oChild1.append(oChild5)oChild2.append(oChild6)oChild
4.append(oChild7)oChild3.append(oChild8)oChild3.append(oChild9)oChild
6.append(oChild10)
# 说明下面代码的输出结果
oRoot.print_all_1()oRoot.print_all_2()
```

答案

oRoot.print_all_1()会打印下面的结果：

```
<Node 'root'><Node 'child1'><Node 'child4'><Node 'child7'><Node
'child5'><Node 'child2'><Node 'child6'><Node 'child10'><Node
'child3'><Node 'child8'><Node 'child9'>
```

oRoot.print_all_1()会打印下面的结果：

```
<Node 'root'><Node 'child1'><Node 'child2'><Node 'child3'><Node
'child4'><Node 'child5'><Node 'child6'><Node 'child8'><Node
'child9'><Node 'child7'><Node 'child10'>
```

为什么提这个问题？

因为对象的精髓就在于组合（composition）与对象构造（object construction）。对象需要有组合成分构成，而且得以某种方式初始化。这里也涉及到递归和生成器（generator）的使用。

生成器是很棒的数据类型。你可以只通过构造一个很长的列表，然后打印列表的内容，就可以取得与 print_all_2 类似的功能。生成器还有一个好处，就是不用占据很多内存。

有一点还值得指出，就是 print_all_1 会以深度优先（depth-first）的方式遍历树(tree)，而 print_all_2 则是宽度优先（width-first）。有时候，一种遍历方式比另一种更合适。但这要看你的应用的具体情况。

问题 12

简要描述 Python 的垃圾回收机制（garbage collection）。

答案

这里能说的很多。你应该提到下面几个主要的点：

- Python 在内存中存储了每个对象的引用计数（reference count）。如果计数值变成 0，那么相应的对象就会小时，分配给该对象的内存就会释放出来用作他用。
- 偶尔也会出现引用循环（reference cycle）。垃圾回收器会定时寻找这个循环，并将其回收。举个例子，假设有两个对象 o1 和 o2，而且符合 o1.x == o2 和 o2.x == o1 这两个条件。如果 o1 和 o2 没有其他代码引用，那么它们就不应该继续存在。但它们的引用计数都是 1。
- Python 中使用了某些启发式算法（heuristics）来加速垃圾回收。例如，越晚创建的对象更有可能被回收。对象被创建之后，垃圾回收器会分配它们所属的代（generation）。每个对象都会被分配一个代，而被分配更年轻代的对象是优先被处理的。

问题 13

将下面的函数按照执行效率高低排序。它们都接受由 0 至 1 之间的数字构成的列表作为输入。这个列表可以很长。一个输入列表的示例如下：[random.random() for i in range(100000)]。你如何证明自己的答案是正确的。

```
def f1(lIn):
    l1 = sorted(lIn)
    l2 = [i for i in l1 if i<0.5]
    return [i*i for i in l2]
def f2(lIn):
    l1 = [i for i in lIn if i<0.5]
    l2 = sorted(l1)
    return [i*i for i in l2]
def f3(lIn):
    l1 = [i*i for i in lIn]
    l2 = sorted(l1)
    return [i for i in l1 if i<(0.5*0.5)]
```

答案

按执行效率从高到低排列：f2、f1 和 f3。要证明这个**答案**是对的，你应该知道如何分析自己代码的性能。Python 中有一个很好的程序分析包，可以满足这个需求。

```
import cProfilelIn = [random.random() for i in
range(100000)]cProfile.run('f1(lIn)')cProfile.run('f2(lIn)')cProfile.
run('f3(lIn)')
```

为了向大家进行完整地说明，下面我们给出上述分析代码的输出结果：

```
>>> cProfile.run('f1(lIn)')
         4 function calls in 0.045 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
        1    0.009    0.009    0.044    0.044 <stdin>:1(f1)
        1    0.001    0.001    0.045    0.045 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}
        1    0.035    0.035    0.035    0.035 {sorted}

>>> cProfile.run('f2(lIn)')
         4 function calls in 0.024 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
        1    0.008    0.008    0.023    0.023 <stdin>:1(f2)
        1    0.001    0.001    0.024    0.024 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}
        1    0.016    0.016    0.016    0.016 {sorted}

>>> cProfile.run('f3(lIn)')
         4 function calls in 0.055 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall
filename:lineno(function)
        1    0.016    0.016    0.054    0.054 <stdin>:1(f3)
        1    0.001    0.001    0.055    0.055 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {method 'disable' of
'_lsprof.Profiler' objects}
        1    0.038    0.038    0.038    0.038 {sorted}
```

为什么提这个问题？

定位并避免代码瓶颈是非常有价值的技能。想要编写许多高效的代码，最终都要回答常识上来——在上面的例子中，如果列表较小的话，很明显是先进行排序更快，因此如果你可以在排序前先进行筛选，那通常都是比较好的做法。其他不显而易见的问题仍然可以通过恰当的工具来定位。因此了解这些工具是有好处的。

问题 14

你有过失败的经历吗？

错误的答案

我从来没有失败过！

为什么提这个问题？

恰当地回答这个问题说明你用于承认错误，为自己的错误负责，并且能够从错误中学习。如果你想变得对别人有帮助的话，所有这些都是特别重要的。如果你真的是个完人，那就太糟了，回答这个问题的时候你可能都有点创意了。

问题 15

你有实施过个人项目吗？

真的？

如果做过个人项目，这说明从更新自己的技能水平方面来看，你愿意比最低要求付出更多的努力。如果你有维护的个人项目，工作之外也坚持编码，那么你的雇主就更可能把你视作为会增值的资产。

即使他们不问这个问题，我也认为谈谈这个话题很有帮助。

1：Python 如何实现单例模式？

Python 有两种方式可以实现单例模式，下面两个例子使用了不同的方式实现单例模式：

1.

```
class Singleton(type):
def __init__(cls, name, bases, dict):
super(Singleton, cls).__init__(name, bases, dict)
cls.instance = None

def __call__(cls, *args, **kw):
if cls.instance is None:
cls.instance = super(Singleton, cls).__call__(*args, **kw)

return cls.instance
```

```
class MyClass(object):
__metaclass__ = Singleton

print MyClass()
print MyClass()
```

2. 使用 decorator 来实现单例模式

```
def singleton(cls):
instances = {}
def getinstance():
if cls not in instances:
instances[cls] = cls()
return instances[cls]
return getinstance

@singleton
class MyClass:
…
```

2：什么是 lambda 函数？

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。

3：Python 是如何进行类型转换的？

Python 提供了将变量或值从一种类型转换成另一种类型的内置函数。int 函数能够将符合数学格式数字型字符串转换成整数。否则，返回错误信息。

```
>>> int(”34”)
34
>>> int(”1234ab”) #不能转换成整数
ValueError: invalid literal for int(): 1234ab
```

函数 int 也能够把浮点数转换成整数，但浮点数的小数部分被截去。
>>> int(34.1234)
34
>>> int(-2.46)
-2
函数°oat 将整数和字符串转换成浮点数：
>>> float(”12”)
12.0
>>> float(”1.111111”)
1.111111
函数 str 将数字转换成字符：
>>> str(98)
‘98’
>>> str(”76.765”)
‘76.765’
整数 1 和浮点数 1.0 在 python 中是不同的。虽然它们的值相等的，但却属于不同的类型。这两个数在计算机的存储形式也是不一样。

4：如何反序的迭代一个序列？how do I iterate over a sequence in reverse order

如果是一个 list，最快的解决方案是：

list.reverse()
try:
for x in list:
“do something with x”
finally:
list.reverse()

如果不是 list，最通用但是稍慢的解决方案是：

for i in range(len(sequence)-1, -1, -1):
x = sequence[i]
<do something with x>

如何在一个 function 里面设置一个全局的变量？

解决方法是在 function 的开始插入一个 global 声明：

def f()

global x

5：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。

1. 将两序列合并为一个序列，并排序，为序列 Source

2. 拿出最大元素 Big，次大的元素 Small

3. 在余下的序列 S[:-2]进行平分，得到序列 max，min

4. 将 Small 加到 max 序列，将 Big 加大 min 序列，重新计算新序列和，和大的为 max，小的为 min。

Python 代码

```
def mean( sorted_list ):

if not sorted_list:

return (([],[]))

big = sorted_list[-1]

small = sorted_list[-2]

big_list, small_list = mean(sorted_list[:-2])

big_list.append(small)

small_list.append(big)

big_list_sum = sum(big_list)

small_list_sum = sum(small_list)

if big_list_sum > small_list_sum:

return ( (big_list, small_list))

else:

return (( small_list, big_list))

tests = [     [1,2,3,4,5,6,700,800],

[10001,10000,100,90,50,1],

range(1, 11),
```

```
[12312, 12311, 232, 210, 30, 29, 3, 2, 1, 1]

]

for l in tests:

l.sort()

print

print “Source List:\t”, l

l1,l2 = mean(l)

print “Result List:\t”, l1, l2

print “Distance:\t”, abs(sum(l1)-sum(l2))

print ‘-*’*40
```

输出结果

Python 代码

```
Source List:      [1, 2, 3, 4, 5, 6, 700, 800]

Result List:      [1, 4, 5, 800] [2, 3, 6, 700]

Distance:          99

-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*

Source List:      [1, 50, 90, 100, 10000, 10001]

Result List:      [50, 90, 10000] [1, 100, 10001]

Distance:          38

-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*

Source List:      [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

Result List:      [2, 3, 6, 7, 10] [1, 4, 5, 8, 9]

Distance:          1

-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*-*
```

Source List: [1, 1, 2, 3, 29, 30, 210, 232, 12311, 12312]

Result List: [1, 3, 29, 232, 12311] [1, 2, 30, 210, 12312]

Distance: 21

-*

python 工程师（web 开发和爬虫方向）-面试经历

一、这家公司主要对亚马逊商品进行数据采集，问的问题比较杂。

1. 是否了解线程的同步和异步？
2. 是否了解网络的同步和异步？
3. 链表和顺序表储存时各自有什么优点？
4. 使用 redis 搭建分布式系统时如何处理网络延迟和网络异常？
5. 数据仓库是什么？
6. 假设有一个爬虫，从网络上获取数据的频率快，本地写入数据的频率慢，使用什么数据结构好？
7. 你是否了解谷歌的无头浏览器？
8. 你是否了解 MySQL 数据库的几种引擎？
9. redis 数据库有哪几种数据结构？

二、这家是做网络电视应用（教育/游戏等）的后台（.APK）

1. 是否了解 django 中的 manage.py 自定义的用法？
2. django 的常用功能有哪些？
3. django 有哪些优势？
4. 是否对 django 的 admin 进行定制过？
5. 在 django 中有使用过原生 sql 语句吗？（查了一下可能是涉及到 ORM 的性能优化）

三、做大数据征信业务的公司，招 django 工程师

1. django 有什么优点？
2. 是否了解 django admin 定制？
3. 描述一下你的项目。

四、做银行/金融业决策系统，招 django 工程师

1. 详细描述一下做某个项目的过程，描述的是一个爬虫项目的过程
2. 是否有对爬虫采集结果进行数据分析，有哪些？

3. 假设现在数据已经爬取完成，需要开发一个接口给用户，你会设置哪些筛选维度？（薪资，地点，职位关键词等）
4. 筛选维度设计好后，现在需要在用户输入某个维度的条件后查询与用户查询目标最为匹配的条目，如何设计查询逻辑算法？（拿薪资举例，以 500 为区间建立引索，将薪资的下限和上限存入每个区间里面,不能有冗余空间。以用户查询值为基准，逆序查询指定个数的条目，前端显示最相关的数据。需要处理非 500 倍数的值。）
   据实际例子去想解决思路，喜欢这种面试方式。
5. 决策系统的某个模块的数据结构如何设计？（使用向量集）

共 22 家问过上面五道

Python4 期模拟面试技术面试题答案

一、python 语法

1. 请说一下你对迭代器和生成器的区别？

答：（1）迭代器是一个更抽象的概念，任何对象，如果它的类有 next 方法和 iter 方法返回自己本身。对于 string、list、dict、tuple 等这类容器对象，使用 for 循环遍历是很方便的。在后台 for 语句对容器对象调用 iter()函数，iter()是 python 的内置函数。iter()会返回一个定义了 next()方法的迭代器对象，它在容器中逐个访问容器内元素，next()也是 python 的内置函数。在没有后续元素时，next()会抛出一个 StopIteration 异常

（2）生成器（Generator）是创建迭代器的简单而强大的工具。它们写起来就像是正规的函数，只是在需要返回数据的时候使用 yield 语句。每次 next()被调用时，生成器会返回它脱离的位置（它记忆语句最后一次执行的位置和所有的数据值）

区别：生成器能做到迭代器能做的所有事，而且因为自动创建了__iter__()和 next()方法，生成器显得特别简洁，而且生成器也是高效的，使用生成器表达式取代列表解析可以同时节省内存。除了创建和保存程序状态的自动方法，当发生器终结时，还会自动抛出 StopIteration 异常

2. 什么是线程安全？

线程安全是在多线程的环境下，能够保证多个线程同时执行时程序依旧运行正确，而且要保证对于共享的数据可以由多个线程存取，但是同一时刻只能有一个线程进行存取。多线程环境下解决资源竞争问题的办法是加锁来保证存取操作的唯一性。

3. 你所遵循的代码规范是什么？请举例说明其要求？

PEP8

1 变量

常量:大写加下划线 USER_CONSTANT

私有变量 ： 小写和一个前导下划线 _private_value

Python 中不存在私有变量一说，若是遇到需要保护的变量，使用小写和一个前导下划线。但这只是程序员之间的一个约定，用于警告说明这是一个私有变量，外部类不要去访问它。但实际上，外部类还是可以访问到这个变量。

内置变量 ： 小写，两个前导下划线和两个后置下划线 __class__

两个前导下划线会导致变量在解释期间被更名。这是为了避免内置变量和其他变量产生冲突。用户定义的变量要严格避免这种风格。以免导致混乱。

2 函数和方法

总体而言应该使用，小写和下划线。但有些比较老的库使用的是混合大小写，即首单词小写，之后每个单词第一个字母大写，其余小写。但现在，小写和下划线已成为规范。

私有方法 ：小写和一个前导下划线

这里和私有变量一样，并不是真正的私有访问权限。同时也应该注意一般函数不要使用两个前导下划线(当遇到两个前导下划线时，Python 的名称改编特性将发挥作用)。

特殊方法 ：小写和两个前导下划线，两个后置下划线

这种风格只应用于特殊函数，比如操作符重载等。

函数参数 ： 小写和下划线，缺省值等号两边无空格

3 类

类总是使用驼峰格式命名，即所有单词首字母大写其余字母小写。类名应该简明，精确，并足以从中理解类所完成的工作。常见的一个方法是使用表示其类型或者特性的后缀，例如:

SQLEngine，MimeTypes 对于基类而言，可以使用一个 Base 或者 Abstract 前缀 BaseCookie，AbstractGroup

4 模块和包

除特殊模块 __init__ 之外，模块名称都使用不带下划线的小写字母。

若是它们实现一个协议，那么通常使用 lib 为后缀，例如:

```
import smtplib
import  os
import  sys
```

5 关于参数

5.1 不要用断言来实现静态类型检测。断言可以用于检查参数，但不应仅仅是进行静态类型检测。 Python 是动态类型语言，静态类型检测违背了其设计思想。断言应该用于避免函数不被毫无意义的调用。

5.2 不要滥用 *args 和 **kwargs。*args 和 **kwargs 参数可能会破坏函数的健壮性。它们使签名变得模糊，而且代码常常开始在不应该的地方构建小的参数解析器。

6 其他

6.1 使用 has 或 is 前缀命名布尔元素

```
is_connect  =  True
    has_member  =  False
```

6.2 用复数形式命名序列

```
members  =  ['user_1',  'user_2']
```

6.3 用显式名称命名字典

```
person_address  =  {'user_1':'10  road  WD',  'user_2'  :  '20  street  hua fu'}
```

6.4 避免通用名称

诸如 list, dict, sequence 或者 element 这样的名称应该避免。

6.5 避免现有名称

诸如 os, sys 这种系统已经存在的名称应该避免。

7 一些数字

一行列数 ：PEP 8 规定为 79 列。根据自己的情况，比如不要超过满屏时编辑器的显示列数。

一个函数 ： 不要超过 30 行代码, 即可显示在一个屏幕类，可以不使用垂直游标即可看到整个函数。

一个类 ： 不要超过 200 行代码，不要有超过 10 个方法。一个模块 不要超过 500

行。

8 验证脚本

可以安装一个 pep8 脚本用于验证你的代码风格是否符合 PEP8。

4. Python 中怎么简单的实现列表去重？

Set

5. python 中 yield 的用法？

答：yield 简单说来就是一个生成器，这样函数它记住上次返 回时在函数体中的位置。对生成器第 二次（或 n 次）调用跳转至该函 次）调用跳转至该函 数。

6. 什么是面向对象编程？

面向对象编程是一种解决软件复用的设计和编程方法。 这种方法把软件系统中相近相似的操作逻辑和操作 应用数据、状态,以类的型式描述出来,以对象实例的形式在软件系统中复用,以达到提高软件开发效率的作用。

7. python2 和 python3 的区别?

1.性能

Py3.0 运行 pystone benchmark 的速度比 Py2.5 慢 30%。Guido 认为 Py3.0 有极大的优化空间，在字符串和整形操作上可

以取得很好的优化结果。

Py3.1 性能比 Py2.5 慢 15%，还有很大的提升空间。

2.编码

Py3.X 源码文件默认使用 utf-8 编码

3. 语法

1）去除了<>，全部改用!=

2）去除``，全部改用 repr()

3）关键词加入 as 和 with，还有 True,False,None

4）整型除法返回浮点数，要得到整型结果，请使用//

5）加入 nonlocal 语句。使用 noclocal x 可以直接指派外围（非全局）变量

6）去除 print 语句，加入 print()函数实现相同的功能。同样的还有 exec 语句，已经改为 exec()函数

7）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

8）输入函数改变了，删除了 raw_input，用 input 代替：

2.X:guess = int(raw_input('Enter an integer : ')) # 读取键盘输入的方法

3.X:guess = int(input('Enter an integer : '))

9）去除元组参数解包。不能 def(a, (b, c)):pass 这样定义函数了

10）新式的 8 进制字变量，相应地修改了 oct()函数。

11）增加了 2 进制字面量和 bin()函数

12）扩展的可迭代解包。在 Py3.X 里，a, b, *rest = seq 和 *rest, a = seq 都是合法的，只要求两点：rest 是 list

对象和 seq 是可迭代的。

13）新的 super()，可以不再给 super()传参数，

14）新的 metaclass 语法：

```
class Foo(*bases, **kwds):
    pass
```

15）支持 class decorator。用法与函数 decorator 一样：

4. 字符串和字节串

1）现在字符串只有 str 一种类型，但它跟 2.x 版本的 unicode 几乎一样。

2）关于字节串，请参阅“数据类型”的第 2 条目

5.数据类型

1)Py3.X 去除了 long 类型，现在只有一种整型——int，但它的行为就像 2.X 版本的 long

2）新增了 bytes 类型，对应于 2.X 版本的八位串，定义一个 bytes 字面量的方法如下：

str 对象和 bytes 对象可以使用.encode() (str -> bytes) or .decode() (bytes -> str)方法相互转化。

3）dict 的.keys()、.items 和.values()方法返回迭代器，而之前的 iterkeys()等函数都被废弃。同时去掉的还有

dict.has_key()，用 in 替代它吧

6.面向对象

1）引入抽象基类（Abstraact Base Classes，ABCs)。

2）容器类和迭代器类被 ABCs 化。

3）迭代器的 next()方法改名为__next__()，并增加内置函数 next()，用以调用迭代器的__next__()方法

4）增加了@abstractmethod 和 @abstractproperty 两个 decorator，编写抽象方法（属性）更加方便。

7.异常

1）所以异常都从 BaseException 继承，并删除了 StardardError

2）去除了异常类的序列行为和.message 属性

3）用 raise Exception(args)代替 raise Exception, args 语法

4）捕获异常的语法改变，引入了 as 关键字来标识异常实例

5）异常链，因为__context__在 3.0a1 版本中没有实现

8.模块变动

1)移除了 cPickle 模块，可以使用 pickle 模块代替。最终我们将会有一个透明高效的模块。

2）移除了 imageop 模块

3）移除了 audiodev, Bastion, bsddb185, exceptions, linuxaudiodev, md5, MimeWriter, mimify, popen2,

rexec, sets, sha, stringold, strop, sunaudiodev, timing 和 xmllib 模块

4）移除了 bsddb 模块(单独发布，可以从 http://www.jcea.es/programacion/pybsddb.htm 获取)

5）移除了 new 模块

6）os.tmpnam()和 os.tmpfile()函数被移动到 tmpfile 模块下

7）tokenize 模块现在使用 bytes 工作。主要的入口点不再是 generate_tokens，而是 tokenize.tokenize()

9.其它

1）xrange() 改名为 range()，要想使用 range()获得一个 list，必须显式调用：

```
>>> list(range(10)) [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

2）bytes 对象不能 hash，也不支持 b.lower()、b.strip()和 b.split()方法，但对于后两者可以使用 b.strip(b' \n\t\r \f')和 b.split(b' ')来达到相同目的

3)zip()、map()和 filter()都返回迭代器。而 apply()、callable()、coerce()、execfile()、reduce()和 reload ()函数都被去除了现在可以使用 hasattr()来替换 callable(). hasattr()的语法如：hasattr(string, '__name__')

4）string.letters 和相关的.lowercase 和.uppercase 被去除，请改用 string.ascii_letters 等

5）如果 x < y 的不能比较，抛出 TypeError 异常。2.x 版本是返回伪随机布尔值的

6）__getslice__系列成员被废弃。a[i:j]根据上下文转换为 a.__getitem__(slice(I, j))或 __setitem__和 __delitem__调用

7）file 类被废弃

8.谈谈你对 GIL 锁对 python 多线程的影响？

GIL 的全称是 Global Interpreter Lock(全局解释器锁)，来源是 python 设计之初的考虑，为了数据安全所做的决定。每个 CPU 在同一时间只能执行一个线程（在单核 CPU 下的多线程其实都只是并发，不是并行，并发和并行从宏观上来讲都是同时处理多路请求的概念。但并发和并行又有区别，并行是指两个或者多个事件在同一时刻发生；而并发是指两个或多个事件在同一时间间隔内发生。）

在 Python 多线程下，每个线程的执行方式：

1、获取 GIL

2、执行代码直到 sleep 或者是 python 虚拟机将其挂起。

3、释放 GIL

可见，某个线程想要执行，必须先拿到 GIL，我们可以把 GIL 看作是“通行证”，并且在一个 python 进程中，GIL 只有一个。拿不到通行证的线程，就不允许进入 CPU 执行。

在 Python2.x 里,GIL 的释放逻辑是当前线程遇见 IO 操作或者 ticks 计数达到 100(ticks 可以看作是 Python 自身的一个计数器，专门做用于 GIL，每次释放后归零，这个计数可以通过 sys.setcheckinterval 来调整)，进行释放。而每次释放 GIL 锁，线程进行锁竞争、切换线程，会消耗资源。并且由于 GIL 锁存在，python 里一个进程永远只能同时执行一个线程(拿到 GIL 的线程才能执行)。

IO 密集型代码(文件处理、网络爬虫等)，多线程能够有效提升效率(单线程下有 IO 操作会进行 IO 等待，造成不必要的时间浪费，而开启多线程能在线程 A 等待时，自动切换到线程 B，可以不浪费 CPU 的资源，从而能提升程序执行效率)，所以多线程对 IO 密集型代码

比较友好。

9. python 是如何进行内存管理的？

一、垃圾回收：python 不像 C++，Java 等语言一样，他们可以不用事先声明变量类型而直接对变量进行赋值。对 Python 语言来讲，对象的类型和内存都是在运行时确定的。这也是为什么我们称 Python 语言为动态类型的原因（这里我们把动态类型可以简单的归结为对变量内存地址的分配是在运行时自动判断变量类型并对变量进行赋值）。

二、引用计数：Python 采用了类似 Windows 内核对象一样的方式来对内存进行管理。每一个对象，都维护这一个对指向该对对象的引用的计数。当变量被绑定在一个对象上的时候，该变量的引用计数就是 1，(还有另外一些情况也会导致变量引用计数的增加),系统会自动维护这些标签，并定时扫描，当某标签的引用计数变为 0 的时候，该对就会被回收。

三、内存池机制 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

二、Linux 基础和数据结构与算法

1. 10 个常用的 Linux 命令？

答案：略

2. find 和 grep 的区别？

grep 命令是一种强大的文本搜索工具，grep 搜索内容串可以是正则表达式，允许对文本文件进行模式查找。如果找到匹配模式， grep 打印包含模式的所有行。

find 通常用来在特定的目录下搜索符合条件的文件，也可以用来搜索特定用户属主的文件。

3. 什么是阻塞？什么是非阻塞？

阻塞调用是指调用结果返回之前，当前线程会被挂起。函数只有在得到结果之后才会返回。有人也许会把阻塞调用和同步调用等同起来，实际上他是不同的。对于同步调用来说，

很多时候当前线程还是激活的，只是从逻辑上当前函数没有返回而已。例如，我们在 CSocket 中调用 Receive 函数，如果缓冲区中没有数据，这个函数就会一直等待，直到有数据才返回。而此时，当前线程还会继续处理各种各样的消息。如果主窗口和调用函数在同一个线程中，除非你在特殊的界面操作函数中调用，其实主界面还是应该可以刷新。socket 接收数据的另外一个函数 recv 则是一个阻塞调用的例子。当 socket 工作在阻塞模式的时候，如果没有数据的情况下调用该函数，则当前线程就会被挂起，直到有数据为止。

非阻塞和阻塞的概念相对应，指在不能立刻得到结果之前，该函数不会阻塞当前线程，而会立刻返回。

4. 描述数组、链表、队列、堆栈的区别？

数组与链表是数据存储方式的概念，数组在连续的空间中存储数据，而链表可以在非连续的空间中存储数据；

队列和堆栈是描述数据存取方式的概念，队列是先进先出，而堆栈是后进先出；队列和堆栈可以用数组来实现，也可以用链表实现。

5. 你知道几种排序,讲一讲你最熟悉的一种?

排序方法	平均情况	最好情况	最坏情况	辅助空间	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	O(1)	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	O(1)	稳定
希尔排序	$O(n \log n)$~ $O(n^2)$	$O(n^{1.3})$	$O(n^2)$	O(1)	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	O(1)	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$~ $O(n)$	不稳定

三、Web 框架

1.django 中当一个用户登录 A 应用服务器（进入登录状态），然后下次请求被 nginx 代理到 B 应用服务器会出现什么影响？

如果用户在 A 应用服务器登陆的 session 数据没有共享到 B 应用服务器，纳米之前的登录状态就没有了。

2.跨域请求问题 django 怎么解决的（原理）

启用中间件

post 请求

验证码

表单中添加{%csrf_token%}标签

3.请解释或描述一下 Django 的架构

对于 Django 框架遵循 MVC 设计，并且有一个专有名词：MVT

M 全拼为 Model，与 MVC 中的 M 功能相同，负责数据处理，内嵌了 ORM 框架

V全拼为View,与MVC中的C功能相同,接收HttpRequest,业务处理,返回HttpResponse

T全拼为Template，与MVC中的V功能相同，负责封装构造要返回的html，内嵌了模板引擎

4.django对数据查询结果排序怎么做，降序怎么做，查询大于某个字段怎么做

排序使用order_by()

降序需要在排序字段名前加-

查询字段大于某个值：使用filter(字段名_gt=值)

5.说一下Django，MIDDLEWARES中间件的作用？

答：中间件是介于request与response处理之间的一道处理过程，相对比较轻量级，并且在全局上改变django的输入与输出。

6.你对Django的认识？

Django是走大而全的方向，它最出名的是其全自动化的管理后台：只需要使用起ORM，做简单的对象定义，它就能自动生成数据库结构、以及全功能的管理后台。

Django内置的ORM跟框架内的其他模块耦合程度高。

应用程序必须使用Django内置的ORM,否则就不能享受到框架内提供的种种基于其ORM的便利；理论上可以切换掉其ORM模块，但这就相当于要把装修完毕的房子拆除重新装修，倒不如一开始就去毛胚房做全新的装修。

Django的卖点是超高的开发效率，其性能扩展有限；采用Django的项目，在流量达到一定规模后，都需要对其进行重构，才能满足性能的要求。

Django适用的是中小型的网站，或者是作为大型网站快速实现产品雏形的工具。

Django模板的设计哲学是彻底的将代码、样式分离； Django从根本上杜绝在模板中进行编码、处理数据的可能。

7. Django重定向你是如何实现的？用的什么状态码？

使用HttpResponseRedirect

redirect和reverse

状态码：302,301

8.ngnix的正向代理与反向代理？

答：正向代理 是一个位于客户端和原始服务器(origin server)之间的服务器，为了从原始服务器取得内容，客户端向代理发送一个请求并指定目标(原始服务器)，然后代理向原始服务器转交请求并将获得的内容返回给客户端。客户端必须要进行一些特别的设置才能使用正向代理。

反向代理正好相反，对于客户端而言它就像是原始服务器，并且客户端不需要进行任何特别的设置。客户端向反向代理的命名空间中的内容发送普通请求，接着反向代理将判断向何处(原始服务器)转交请求，并将获得的内容返回给客户端，就像这些内容原本就是它自己的一样。

9. Tornado 的核是什么？

Tornado 的核心是 ioloop 和 iostream 这两个模块，前者提供了一个高效的 I/O 事件循环，后者则封装了 一个无阻塞的 socket 。通过向 ioloop 中添加网络 I/O 事件，利用无阻塞的 socket ，再搭配相应的回调 函数，便可达到梦寐以求的高效异步执行。

10.Django 本身提供了 runserver，为什么不能用来部署？

runserver 方法是调试 Django 时经常用到的运行方式，它使用 Django 自带的

WSGI Server 运行，主要在测试和开发中使用，并且 runserver 开启的方式也是单进程 。

uWSGI 是一个 Web 服务器，它实现了 WSGI 协议、uwsgi、http 等协议。注意 uwsgi 是一种通信协议，而 uWSGI 是实现 uwsgi 协议和 WSGI 协议的 Web 服务器。uWSGI 具有超快的性能、低内存占用和多 app 管理等优点，并且搭配着 Nginx

就是一个生产环境了，能够将用户访问请求与应用 app 隔离开，实现真正的部署 。相比来讲，支持的并发量更高，方便管理多进程，发挥多核的优势，提升性能。

四、网络编程和前端

1.AJAX 是什么，如何使用 AJAX？

ajax(异步的 javascript 和 xml) 能够刷新局部网页数据而不是重新加载整个网页。

第一步，创建 xmlhttprequest 对象，var xmlhttp =new XMLHttpRequest ();XMLHttpRequest 对象用来和服务器交换数据。

第二步，使用 xmlhttprequest 对象的 open（）和 send（）方法发送资源请求给服务器。

第三步，使用 xmlhttprequest 对象的 responseText 或 responseXML 属性获得服务器的响应。

第四步，onreadystatechange 函数，当发送请求到服务器，我们想要服务器响应执行一些功能就需要使用 onreadystatechange 函数，每次 xmlhttprequest 对象的 readyState 发生改变都会触发 onreadystatechange 函数。

2. 常见的 HTTP 状态码有哪些？

200 OK

301 Moved Permanently

302 Found

304 Not Modified

307 Temporary Redirect

400 Bad Request

401 Unauthorized

403 Forbidden

404 Not Found

410 Gone

500 Internal Server Error

501 Not Implemented

3. Post 和 get 区别？

1、GET 请求，请求的数据会附加在 URL 之后，以?分割 URL 和传输数据，多个参数用&连接。URL 的编码格式采用的是 ASCII 编码，而不是 uniclde，即是说所有的非 ASCII 字符都要编码之后再传输。

POST 请求：POST 请求会把请求的数据放置在 HTTP 请求包的包体中。上面的 item=bandsaw 就是实际的传输数据。

因此，GET 请求的数据会暴露在地址栏中，而 POST 请求则不会。

2、传输数据的大小

在 HTTP 规范中，没有对 URL 的长度和传输的数据大小进行限制。但是在实际开发过程中，对于 GET，特定的浏览器和服务器对 URL 的长度有限制。因此，在使用 GET 请求时，传输数据会受到 URL 长度的限制。

对于 POST，由于不是 URL 传值，理论上是不会受限制的，但是实际上各个服务器会规定对 POST 提交数据大小进行限制，Apache、IIS 都有各自的配置。

3、安全性

POST 的安全性比 GET 的高。这里的安全是指真正的安全，而不同于上面 GET 提到的安全方法中的安全，上面提到的安全仅仅是不修改服务器的数据。比如，在进行登录操作，通过 GET 请求，用户名和密码都会暴露再 URL 上，因为登录页面有可能被浏览器缓存以及其他人查看浏览器的历史记录的原因，此时的用户名和密码就很容易被他人拿到了。除此之外，GET 请求提交的数据还可能会造成 Cross-site request frogery 攻击。

4.cookie 和 session 的区别？

1、cookie 数据存放在客户的浏览器上，session 数据放在服务器上。

2、cookie 不是很安全，别人可以分析存放在本地的 COOKIE 并进行 COOKIE 欺骗考虑到安全应当使用 session。

3、session 会在一定时间内保存在服务器上。当访问增多，会比较占用服务器的性能考虑到减轻服务器性能方面，应当使用 COOKIE。

4、单个 cookie 保存的数据不能超过 4K，很多浏览器都限制一个站点最多保存 20 个 cookie。

5、建议：

将登陆信息等重要信息存放为 SESSION

其他信息如果需要保留，可以放在 COOKIE 中

5.创建一个简单 tcp 服务器需要的流程

1.socket 创建一个套接字

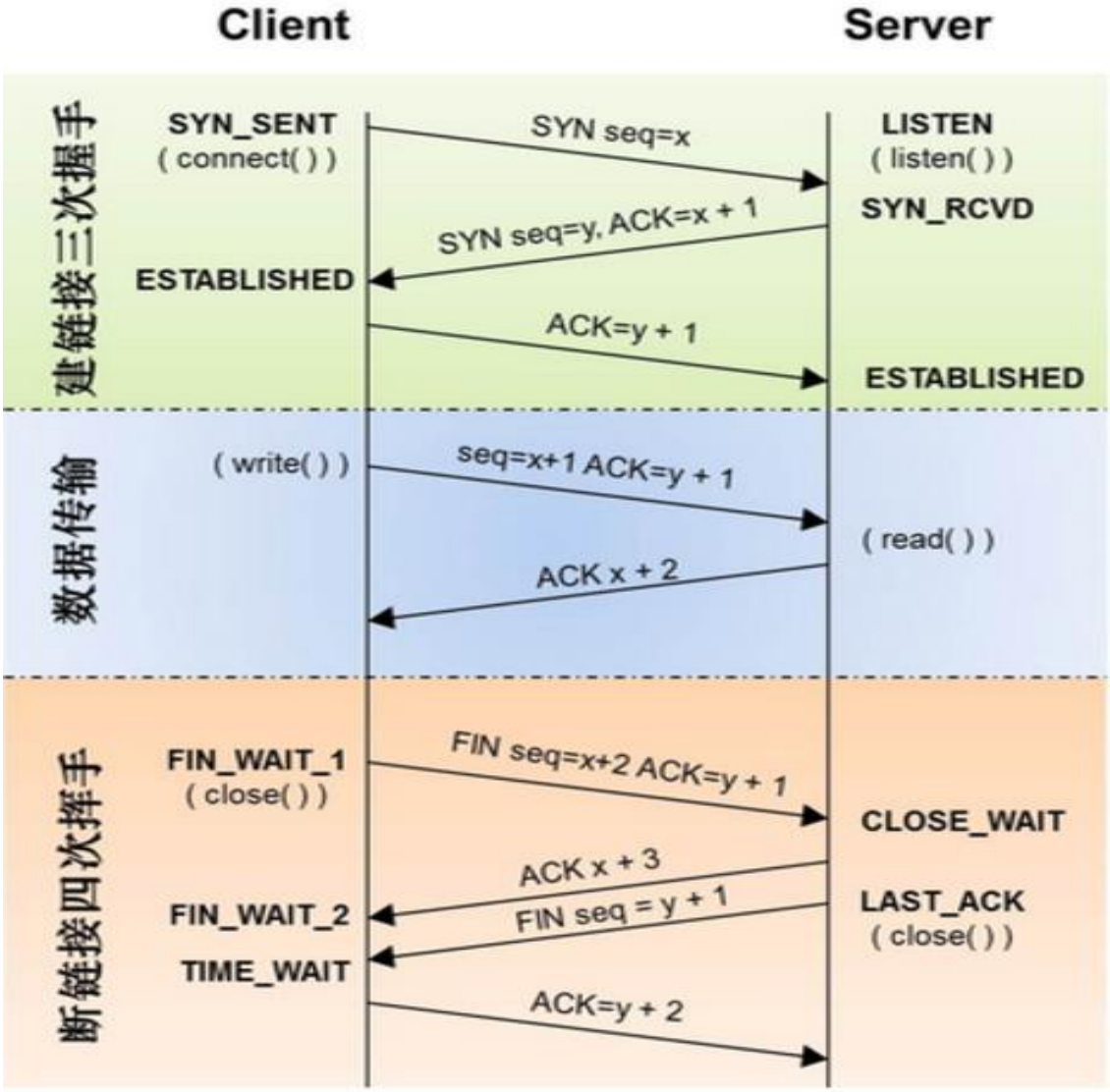
2.bind 绑定 ip 和 port

3.listen 使套接字变为可以被动链接

4.accept 等待客户端的链接

5.recv/send 接收发送数据

6.请简单说一下三次握手和四次挥手？什么是 2msl？为什么要这样做？



2MSL 即两倍的 MSL，TCP 的 TIME_WAIT 状态也称为 2MSL 等待状态，

当 TCP 的一端发起主动关闭，在发出最后一个 ACK 包后，

即第 3 次握 手完成后发送了第四次握手的 ACK 包后就进入了 TIME_WAIT 状态，

必须在此状态上停留两倍的 MSL 时间，

等待 2MSL 时间主要目的是怕最后一个 ACK 包对方没收到，

那么对方在超时后将重发第三次握手的 FIN 包，

主动关闭端接到重发的 FIN 包后可以再发一个 ACK 应答包。

在 TIME_WAIT 状态 时两端的端口不能使用，要等到 2MSL 时间结束才可继续使用。

当连接处于 2MSL 等待阶段时任何迟到的报文段都将被丢弃。

不过在实际应用中可以通过设置 SO_REUSEADDR 选项达到不必等待 2MSL 时间结束再使用此端口。

五、爬虫和数据库

1.scrapy 和 scrapy-redis 有什么区别？为什么选择 redis 数据库？

1）scrapy 是一个 Python 爬虫框架，爬取效率极高，具有高度定制性，但是不支持分布式。而 scrapy-redis 一套基于 redis 数据库、运行在 scrapy 框架之上的组件，可以让 scrapy 支持分布式策略，Slaver 端共享 Master 端 redis 数据库里的 item 队列、请求队列和请求指纹集合。

2）为什么选择 redis 数据库，因为 redis 支持主从同步，而且数据都是缓存在内存中的，所以基于 redis 的分布式爬虫，对请求和数据的高频读取效率非常高。

2. 你用过的爬虫框架或者模块有哪些？谈谈他们的区别或者优缺点？

Python 自带：urllib，urllib2

第 三 方：requests

框　　架：Scrapy

urllib 和 urllib2 模块都做与请求 URL 相关的操作，但他们提供不同的功能。

urllib2.：urllib2.urlopen 可以接受一个 Request 对象或者 url，（在接受 Request 对象时候，并以此可以来设置一个 URL 的 headers），urllib.urlopen 只接收一个 url

urllib 有 urlencode,urllib2 没有，因此总是 urllib，urllib2 常会一起使用的原因

scrapy 是封装起来的框架，他包含了下载器，解析器，日志及异常处理，基于多线程，twisted 的方式处理，对于固定单个网站的爬取开发，有优势，但是对于多网站爬取 100 个网站，并发及分布式处理方面，不够灵活，不便调整与括展。

request 是一个 HTTP 库， 它只是用来，进行请求，对于 HTTP 请求，他是一个强大的库，下载，解析全部自己处理，灵活性更高，高并发与分布式部署也非常灵活，对于功能可以更好实现.

Scrapy 优缺点：

优点：scrapy 是异步的

采取可读性更强的 xpath 代替正则

强大的统计和 log 系统

同时在不同的 url 上爬行

支持 shell 方式，方便独立调试

写 middleware,方便写一些统一的过滤器

通过管道的方式存入数据库

缺点：基于 python 的爬虫框架，扩展性比较差

基于 twisted 框架，运行中的 exception 是不会干掉 reactor，并且异步框架出错后是不会停掉其他任务的，数据出错后难以察觉。

3.你常用的 mysql 引擎有哪些？各引擎间有什么区别？

主要 MyISAM 与 InnoDB 两个引擎，其主要区别如下：

一、InnoDB 支持事务，MyISAM 不支持，这一点是非常之重要。事务是一种高级的处理方式，如在一些列增删改中只要哪个出错还可以回滚还原，而 MyISAM 就不可以了；

二、MyISAM 适合查询以及插入为主的应用，InnoDB 适合频繁修改以及涉及到安全性较高的应用；

三、InnoDB 支持外键，MyISAM 不支持；

四、MyISAM 是默认引擎，InnoDB 需要指定；

五、InnoDB 不支持 FULLTEXT 类型的索引；

六、InnoDB 中不保存表的行数，如 select count(*) from table 时，InnoDB；需要扫描一遍整个表来计算有多少行，但是 MyISAM 只要简单的读出保存好的行数即可。注意的是，当 count(*)语句包含 where 条件时 MyISAM 也需要扫描整个表；

七、对于自增长的字段，InnoDB 中必须包含只有该字段的索引，但是在 MyISAM 表中可以和其他字段一起建立联合索引；

八、清空整个表时，InnoDB 是一行一行的删除，效率非常慢。MyISAM 则会重建表；

九、InnoDB 支持行锁（某些情况下还是锁整表，如 update table set a=1 where user like '%lee%'

4.描述下 scrapy 框架运行的机制？

答：从 start_urls 里获取第一批 url 并发送请求，请求由引擎交给调度器入请求队列，获取完毕后，调度器将请求队列里的请求交给下载器去获取请求对应的响应资源，并将响应交给自己编写的解析方法做提取处理：1. 如果提取出需要的数据，则交给管道文件处理；2. 如果提取出 url，则继续执行之前的步骤（发送 url 请求，并由引擎将请求交给调度器入队列...），直到请求队列里没有请求，程序结束。

5.什么是关联查询，有哪些？

答：将多个表联合起来进行查询，主要有内连接、左连接、右连接、全连接（外连接）

6.写爬虫是用多进程好？还是多线程好？ 为什么？

答：IO 密集型代码(文件处理、网络爬虫等)，多线程能够有效提升效率(单线程下有 IO 操作会进行 IO 等待，造成不必要的时间浪费，而开启多线程能在线程 A 等待时，自动切换到线程 B，可以不浪费 CPU 的资源，从而能提升程序执行效率)。在实际的数据采集过程中，既考虑网速和响应的问题，也需要考虑自身机器的硬件情况，来设置多进程或多线程

7.数据库的优化？

1. 优化索引、SQL 语句、分析慢查询；
2. 设计表的时候严格根据数据库的设计范式来设计数据库；
3. 使用缓存，把经常访问到的数据而且不需要经常变化的数据放在缓存中，能节约磁盘 IO；
4. 优化硬件；采用 SSD，使用磁盘队列技术(RAID0,RAID1,RDID5)等；

5. 采用 MySQL 内部自带的表分区技术，把数据分层不同的文件，能够提高磁盘的读取效率；

6. 垂直分表；把一些不经常读的数据放在一张表里，节约磁盘 I/O；

7. 主从分离读写；采用主从复制把数据库的读操作和写入操作分离开来；

8. 分库分表分机器（数据量特别大），主要的的原理就是数据路由；

9. 选择合适的表引擎，参数上的优化；

10. 进行架构级别的缓存，静态化和分布式；

11. 不采用全文索引；

12. 采用更快的存储方式，例如 NoSQL 存储经常访问的数据

8.常见的反爬虫和应对方法？

1）.通过 Headers 反爬虫

从用户请求的 Headers 反爬虫是最常见的反爬虫策略。很多网站都会对 Headers 的 User-Agent 进行检测，还有一部分网站会对 Referer 进行检测（一些资源网站的防盗链就是检测 Referer)。如果遇到了这类反爬虫机制，可以直接在爬虫中添加 Headers，将浏览器的 User-Agent 复制到爬虫的 Headers 中；或者将 Referer 值修改为目标网站域名。对于检测 Headers 的反爬虫，在爬虫中修改或者添加 Headers 就能很好的绕过。

2）.基于用户行为反爬虫

还有一部分网站是通过检测用户行为，例如同一 IP 短时间内多次访问同一页面，或者同一账户短时间内多次进行相同操作。

大多数网站都是前一种情况，对于这种情况，使用 IP 代理就可以解决。可以专门写一个爬虫，爬取网上公开的代理 ip，检测后全部保存起来。这样的代理 ip 爬虫经常会用到，最好自己准备一个。有了大量代理 ip 后可以每请求几次更换一个 ip，这在 requests 或者 urllib2 中很容易做到，这样就能很容易的绕过第一种反爬虫。

对于第二种情况，可以在每次请求后随机间隔几秒再进行下一次请求。有些有逻辑漏洞的网站，可以通过请求几次，退出登录，重新登录，继续请求来绕过同一账号短时间内不能多次进行相同请求的限制。

3）.动态页面的反爬虫

上述的几种情况大多都是出现在静态页面，还有一部分网站，我们需要爬取的数据是通过 ajax 请求得到，或者通过 JavaScript 生成的。首先用 Fiddler 对网络请求进行分析。如果能够找到 ajax 请求，也能分析出具体的参数和响应的具体含义，我们就能采用上面的方法，直接利用 requests 或者 urllib2 模拟 ajax 请求，对响应的 json 进行分析得到需要的数据。

能够直接模拟 ajax 请求获取数据固然是极好的，但是有些网站把 ajax 请求的所有参数全部加密了。我们根本没办法构造自己所需要的数据的请求。这种情况下就用 selenium+phantomJS，调用浏览器内核，并利用 phantomJS 执行 js 来模拟人为操作以及触发页面中的 js 脚本。从填写表单到点击按钮再到滚动页面，全部都可以模拟，不考虑具体的请求和响应过程，只是完完整整的把人浏览页面获取数据的过程模拟一遍。

用这套框架几乎能绕过大多数的反爬虫，因为它不是在伪装成浏览器来获取数据（上述

的通过添加 Headers 一定程度上就是为了伪装成浏览器），它本身就是浏览器，phantomJS 就是一个没有界面的浏览器，只是操控这个浏览器的不是人。利 selenium+phantomJS 能干很多事情，例如识别点触式（12306）或者滑动式的验证码，对页面表单进行暴力破解等。

9.分布式爬虫主要解决什么问题？

1)ip

2)带宽

3）cpu

4）io

10.爬虫过程中验证码怎么处理？

1.scrapy 自带（但是成功率不高）

2.付费接口（若快 http://www.ruokuai.com/home/pricetype）

六、其他

主观题 答案：略...

python 面试题超纲 20 道

What is python .. (“dot dot”) notation syntax?

Python 的 .. (点 点) 是什么语法?

```
f = 1..__truediv__ # or 1..__div__ for python 2print(f(8)) # prints 0.125
```

真的超纲了喂。..是什么鬼啊摔。真的不是问**args 是什么语法么？那我们用 python 试试好了。

```
>>> f = 1.>>> f1.0>>> f.__floordiv__
<method-wrapper '__floordiv__' of float object at 0x7f9fb4dc1a20>
```

咦，你是不是有看出来什么了。再看一个例子。

```
>>> 1..__add__(2.)3.0
```

大家都看出来了吧，..**本身并不是什么操作符更不是什么语法**。其中，**第一个点是浮点值的一部分，第二个点是点操作符来访问对象属性和方法**。
太简单，下一题。

Why is x**4.0 faster than x**4 in Python 3?

为什么在 Python3 中 x**4.0 比 x**4 运行的快?

```
$ python -m timeit "for x in range(100):" " x**4.0"
  10000 loops, best of 3: 24.2 usec per loop

$ python -m timeit "for x in range(100):" " x**4"
  10000 loops, best of 3: 30.6 usec per loop
```

我先解释下为什么会有这个问题，先不要讨论提问者的电脑怎么这么慢好么。我拿 **python2** 试了一下，结果是这样。

```
$ python -m timeit "for x in range(100):" " x**4.0"
  10000 loops, best of 3: 15.6 usec per loop

$ python -m timeit "for x in range(100):" " x**4"
  10000 loops, best of 3: 4.59 usec per loop
```

可以明显看出 **python2 中 int 型计算明显快于 float**。嗯...其实我已经回答完了不知道大家有没有发现。这个问题是由 python2 与 python3 差别引起的。
先指出答案核心：**python3 的 4 是** `PyLongObject`，**pytohn2 的 4 是** `int`。

- pytohn3 与 python2 中的 float 对象依然是那个原生类。
- python3 中的 int 对象实例化一个支持任意长度的成熟类，叫 PyLongObject。
- python2 中的**整数分为 int 与 long**，比如：

```
# Python 2type(4)  #〈type 'int'〉type(4L) #〈type 'long'〉
```

以上的区别导致 python3 中整数的运算更加繁琐复杂，因为你需要用 PyLongObject 对象的值来执行它的 ob_digit 阵数组。(参考：**Understanding memory allocation for large integers in Python for more on PyLongObjects.**)

> Given a string of a million numbers (Pi for example), write a function/program that returns all repeating 3 digit numbers and number of repetition greater than 1
>
> 给定一个长度为 100 万的数字（比如π），写代码计算出所有连续的三个数字，且要求该连续的三个数字至少重复出现过一次。

```
# For example: if the string was: 123412345123456 then the function/program would return:
# 123 - 3 times# 234 - 3 times# 345 - 2 times
```

提问的人最后还加了一句能不能实现**时间复杂度为常数级**解决方案？这很明显的**面试题**嘛，假如面试时候你遇到这个你会怎么回答？

首先，**在 O(1)情况下无法处理任意大小的数据结构**，在这种情况下，**最好的希望是 O(n)**，其中 n 是字符串的长度，也就是**线性时间复杂度**。

但是这里如果每次输入定长 100w，从**技术的角度来说实现 O(1)是可行的（以为数据源定长嘛就相当于 n=100w，这属于咬文嚼字了没意思）**，但是我相信这绝不会是这题重点。还是先给出 python 实现代码好了：

```
inpStr = '123412345123456'
# O(1) array creation.
freq = [0] * 1000
# O(n) string processing.for val in [int(inpStr[pos:pos+3]) for pos in range(len(inpStr) - 2)]:
    freq[val] += 1
# O(1) output of relevant array values.print ([(num, freq[num]) for num in range(1000) if freq[num] > 1])
```

代码不复杂，相信大家都看的懂，我们继续讨论速度的问题。上述代码可见实现 O(n)是没什么问题的，但是如果再提速呢？

方法也有，**运用归并思想，将输入值切片，多线程运行。**像这样

```
# 123412345123456# 拆成
    vv
123412  vv
    123451
```

```
5123456
```

运用归并的思想去解决问题，但问题来了，因为 **pytohn 有 GIL 的存在，所以用 python 实现该思想可能成本很大**。当然你可以用其他语言实现。我觉得面试时候给出 python 代码提出归并思想就比较好了。

Why in python 0, 0 == (0, 0) equals (0, False)

为什么在 Python 中表达式 0, 0 == (0, False) 的结果是 (0, False)

```
(0, 0) == 0, 0  # results in a two element tuple: (False, 0)
0, 0 == (0, 0)  # results in a two element tuple: (0, False)
(0, 0) == (0, 0) # results in a boolean True # But:
a = 0, 0
b = (0, 0)
a == b # results in a boolean True
```

这题很简单啊，我换种写法,标明优先级大家就懂了。

```
(((0, 0) == 0), 0)  # results in a two element tuple: (False, 0)
(0, (0 == (0, 0)))  # results in a two element tuple: (0, False)
((0, 0) == (0, 0)) # results in a boolean True # ALSO:
a = 0, 0
b = (0, 0)
a == b # results in a boolean True
```

逗号分隔符与相等运算符的优先级是不同的。

Does Python optimize away a variable that's only used as a return value?

Python 会把一个只用做返回值的变量优化掉吗？

```
# Case 1:def func():
    a = 42
    return a
# Case 2:def func2():
    return 42
```

这个问题非常有趣，我是从来没往这方面想过。我对其底层可能理解不多大家轻点喷，我把解决问题的过程帮大家写出来，大家可以看着思考一下：

```
# 用 dis 打印运行过程
from dis import dis# Case 1:
dis(func)
```

```
  2           0 LOAD_CONST               1 (42)
              2 STORE_FAST               0 (a)

  3           4 LOAD_FAST                0 (a)
              6 RETURN_VALUE
# Case 2:
dis(func2)
  2           0 LOAD_CONST               1 (42)
              2 RETURN_VALUE
```

作者：Tony 带不带水
链接：https://www.jianshu.com/p/8d66c5b99642
來源：简书

许多自学爬虫(python)的小伙伴因为没有经历过面试所以在找工作之前难免有些抓不住重点，虽然自己有些技术

但是因为发挥不好而错失工作机会，本人经过 n 次面试以后特总结以下面试常见问题，为想要转爬虫的小

伙伴提供一些参考。

一.项目问题：

一般面试官的第一个问题八成都是问一下以前做过的项目，所以最好准备两个自己最近写的有些技术

含量的项目，当然一定要自己亲手写过的，在别的地方看的源码，就算看的再清楚，总归没有自己敲的

了解的多。以下是抽出的几点

1.你写爬虫的时候都遇到过什么反爬虫措施，你是怎么解决的

2.用的什么框架，为什么选择这个框架(我用的是 scrapy 框架，所以下面的问题也是针对 scrapy)

二.框架问题（scrapy）可能会根据你说的框架问不同的问题，但是 scrapy 还是比较多的

1.scrapy 的基本结构（五个部分都是什么，请求发出去的整个流程）

2.scrapy 的去重原理 （指纹去重到底是什么原理）

3.scrapy 中间件有几种类，你用过那些中间件，

4.scrapy 中间件再哪里起的作用（面向切面编程）

三.代理问题

1.为什么会用到代理

2.代理怎么使用（具体代码，请求在什么时候添加的代理）

3.代理失效了怎么处理

四.验证码处理

1.登陆验证码处理

2.爬取速度过快出现的验证码处理

3.如何用机器识别验证码

五.模拟登陆问题

1.模拟登陆流程

2.cookie 如何处理

3.如何处理网站传参加密的情况

六.分布式

1.分布式原理

2.分布式如何判断爬虫已经停止了

3.分布式去重原理

七.数据存储和数据库问题

1.关系型数据库和非关系型数据库的区别

2.爬下来数据你会选择什么存储方式，为什么

3.各种数据库支持的数据类型，和特点，比如：redis 如何实现持久化，mongodb

是否支持事物等。。

八.python 基础问题

基础问题非常多，但是因为爬虫性质，还是有些问的比较多的，下面是总结

1.python2 和 python3 的区别，如何实现 python2 代码迁移到 python3 环境

2.python2 和 python3 的编码方式有什么差别（工作中发现编码问题还是挺让人不爽的）

3.迭代器，生成器，装饰器

4.python 的数据类型

九.协议问题

爬虫从网页上拿数据肯定需要模拟网络通信的协议

1.http 协议，请求由什么组成，每个字段分别有什么用,https 和 http 有什么差距

2.证书问题

3.TCP,UDP 各种相关问题

十.数据提取问题

1.主要使用什么样的结构化数据提取方式，可能会写一两个例子

2.正则的使用

3.动态加载的数据如何提取

4.json 数据如何提取

十二.算法问题

这个实在不好总结，比较考验代码功力，大部分会让你写出时间复杂度比较低的

算法。小伙伴们要善用 python 的数据类型，对 python 的数据结构深入了解。

以上就是总结内容，欢迎小伙伴们共同探讨。每个公司各有特点，但是这些算是基础，也是常见问题。

祝大家能找到理想工作，不写 bug

1、大数据的文件读取

① 利用生成器 generator

②迭代器进行迭代遍历：for line in file

2、迭代器和生成器的区别

1)迭代器是一个更抽象的概念，任何对象，如果它的类有 next 方法和 iter 方法返回自己本身。对于 string、list、dict、tuple 等这类容器对象，使用 for 循环遍历是很方便的。在后台 for 语句对容器对象调用 iter()函数，iter()是 python 的内置函数。iter()会返回一个定义了 next()方法的迭代器对象，它在容器中逐个访问容器内元素，next()也是 python 的内置函数。在没有后续元素时，next()会抛出一个 StopIteration 异常

2)生成器(Generator)是创建迭代器的简单而强大的工具。它们写起来就像是正规的函数，只是在需要返回数据的时候使用 yield 语句。每次 next()被调用时，生成器会返回它脱离的位置(它记忆语句最后一次执行的位置和所有的数据值)

区别：生成器能做到迭代器能做的所有事,而且因为自动创建了__iter__()和 next()方法,生成器显得特别简洁,而且生成器也是高效的，使用生成器表达式取代列表

解析可以同时节省内存。除了创建和保存程序状态的自动方法,当发生器终结时,还会自动抛出 StopIteration 异常

3、装饰器的作用和功能：

引入日志
函数执行时间统计
执行函数前预备处理
执行函数后的清理功能
权限校验等场景
缓存

4、简单谈下 GIL：

Global Interpreter Lock(全局解释器锁)

Python 代码的执行由 Python 虚拟机(也叫解释器主循环，CPython 版本)来控制，Python 在设计之初就考虑到要在解释器的主循环中，同时只有一个线程在执行，即在任意时刻，只有一个线程在解释器中运行。对 Python 虚拟机的访问由全局解释器锁(GIL)来控制，正是这个锁能保证同一时刻只有一个线程在运行。

在多线程环境中，Python 虚拟机按以下方式执行：

1. 设置 GIL

2. 切换到一个线程去运行

3. 运行：

a. 指定数量的字节码指令，或者

b. 线程主动让出控制(可以调用 time.sleep(0))

4. 把线程设置为睡眠状态

5. 解锁 GIL

6. 再次重复以上所有步骤

在调用外部代码(如 C/C++扩展函数)的时候，GIL 将会被锁定，直到这个函数结束为止(由于在这期间没有 Python 的字节码被运行，所以不会做线程切换)。

5、find 和 grep

grep 命令是一种强大的文本搜索工具，grep 搜索内容串可以是正则表达式，允许对文本文件进行模式查找。如果找到匹配模式，grep 打印包含模式的所有行。

find 通常用来再特定的目录下搜索符合条件的文件，也可以用来搜索特定用户属主的文件。

6、线上服务可能因为种种原因导致挂掉怎么办?

linux 下的后台进程管理利器 supervisor

每次文件修改后再 linux 执行 service supervisord restart

7、如何提高 python 的运行效率

使用生成器;关键代码使用外部功能包(Cython，pylnlne，pypy，pyrex);针对循环的优化--尽量避免在循环中访问变量的属性

8、常用 Linux 命令：

ls,help,cd,more,clear,mkdir,pwd,rm,grep,find,mv,su,date

9、Python 中的 yield 用法

yield 简单说来就是一个生成器，这样函数它记住上次返 回时在函数体中的位置。对生成器第 二次(或 n 次)调用跳转至该函 次)调用跳转至该函数。

10、Python 是如何进行内存管理的

一、垃圾回收：python 不像 C++，Java 等语言一样，他们可以不用事先声明变量类型而直接对变量进行赋值。对 Python 语言来讲，对象的类型和内存都是在运行时确定的。这也是为什么我们称 Python 语言为动态类型的原因(这里我们把动态类型可以简单的归结为对变量内存地址的分配是在运行时自动判断变量类型并对变量进行赋值)。

二、引用计数：Python 采用了类似 Windows 内核对象一样的方式来对内存进行管理。每一个对象，都维护这一个对指向该对对象的引用的计数。当变量被绑定在一个对象上的时候，该变量的引用计数就是 1，(还有另外一些情况也会导致变量引用计数的增加),系统会自动维护这些标签，并定时扫描，当某标签的引用计数变为 0 的时候，该对就会被回收。

三、内存池机制 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

11、描述数组、链表、队列、堆栈的区别?

数组与链表是数据存储方式的概念，数组在连续的空间中存储数据，而链表可以在非连续的空间中存储数据;

队列和堆栈是描述数据存取方式的概念，队列是先进先出，而堆栈是后进先出;队列和堆栈可以用数组来实现，也可以用链表实现。

12、你知道几种排序,讲一讲你最熟悉的一种?

你是最棒的!

web 框架部分

1.django 中当一个用户登录 A 应用服务器(进入登录状态)，然后下次请求被 nginx 代理到 B 应用服务器会出现什么影响?

如果用户在 A 应用服务器登陆的 session 数据没有共享到 B 应用服务器，纳米之前的登录状态就没有了。

2.跨域请求问题 django 怎么解决的(原理)

启用中间件

post 请求

验证码
表单中添加{%csrf_token%}标签

3.请解释或描述一下 Django 的架构

对于 Django 框架遵循 MVC 设计，并且有一个专有名词：MVT

M 全拼为 Model，与 MVC 中的 M 功能相同，负责数据处理，内嵌了 ORM 框架

V 全拼为 View，与 MVC 中的 C 功能相同，接收 HttpRequest，业务处理，返回 HttpResponse

T 全拼为 Template，与 MVC 中的 V 功能相同，负责封装构造要返回的 html，内嵌了模板引擎

4.django 对数据查询结果排序怎么做，降序怎么做，查询大于某个字段怎么做

排序使用 order_by()
降序需要在排序字段名前加-
查询字段大于某个值：使用 filter(字段名_gt=值)

5.说一下 Django，MIDDLEWARES 中间件的作用?

答：中间件是介于 request 与 response 处理之间的一道处理过程，相对比较轻量级，并且在全局上改变 django 的输入与输出。

6.你对 Django 的认识?

Django 是走大而全的方向，它最出名的是其全自动化的管理后台：只需要使用起 ORM，做简单的对象定义，它就能自动生成数据库结构、以及全功能的管理后台。

Django 内置的 ORM 跟框架内的其他模块耦合程度高。

应用程序必须使用 Django 内置的 ORM，否则就不能享受到框架内提供的种种基于其 ORM 的便利;理论上可以切换掉其 ORM 模块，但这就相当于要把装修完毕的房子拆除重新装修，倒不如一开始就去毛胚房做全新的装修。

Django 的卖点是超高的开发效率，其性能扩展有限;采用 Django 的项目，在流量达到一定规模后，都需要对其进行重构，才能满足性能的要求。

Django 适用的是中小型的网站，或者是作为大型网站快速实现产品雏形的工具。

Django 模板的设计哲学是彻底的将代码、样式分离; Django 从根本上杜绝在模板中进行编码、处理数据的可能。

7. Django 重定向你是如何实现的?用的什么状态码?

使用 HttpResponseRedirect

redirect 和 reverse

状态码：302,301

8.ngnix 的正向代理与反向代理?

正向代理 是一个位于客户端和原始服务器(origin server)之间的服务器，为了从原始服务器取得内容，客户端向代理发送一个请求并指定目标(原始服务器)，然后代理向原始服务器转交请求并将获得的内容返回给客户端。客户端必须要进行一些特别的设置才能使用正向代理。

反向代理正好相反，对于客户端而言它就像是原始服务器，并且客户端不需要进行任何特别的设置。客户端向反向代理的命名空间中的内容发送普通请求，接着反向代理将判断向何处(原始服务器)转交请求，并将获得的内容返回给客户端，就像这些内容原本就是它自己的一样。

9. Tornado 的核是什么?

Tornado 的核心是 ioloop 和 iostream 这两个模块，前者提供了一个高效的 I/O 事件循环，后者则封装了 一个无阻塞的 socket 。通过向 ioloop 中添加网络 I/O 事件，利用无阻塞的 socket ，再搭配相应的回调 函数，便可达到梦寐以求的高效异步执行。

10.Django 本身提供了 runserver，为什么不能用来部署?

runserver 方法是调试 Django 时经常用到的运行方式，它使用 Django 自带的

WSGI Server 运行，主要在测试和开发中使用，并且 runserver 开启的方式也是单进程 。

uWSGI 是一个 Web 服务器，它实现了 WSGI 协议、uwsgi、http 等协议。注意 uwsgi 是一种通信协议，而 uWSGI 是实现 uwsgi 协议和 WSGI 协议的 Web 服务器。uWSGI 具有超快的性能、低内存占用和多 app 管理等优点，并且搭配着 Nginx

就是一个生产环境了，能够将用户访问请求与应用 app 隔离开，实现真正的部署 。相比来讲，支持的并发量更高，方便管理多进程，发挥多核的优势，提升性能。

网络编程和前端部分

1.AJAX 是什么，如何使用 AJAX?

ajax(异步的 javascript 和 xml) 能够刷新局部网页数据而不是重新加载整个网页。

第一步，创建 xmlhttprequest 对象，var xmlhttp =new XMLHttpRequest();XMLHttpRequest 对象用来和服务器交换数据。

第二步，使用 xmlhttprequest 对象的 open()和 send()方法发送资源请求给服务器。

第三步，使用 xmlhttprequest 对象的 responseText 或 responseXML 属性获得服务器的响应。

第四步，onreadystatechange 函数，当发送请求到服务器，我们想要服务器响应执行一些功能就需要使用 onreadystatechange 函数，每次 xmlhttprequest 对象的 readyState 发生改变都会触发 onreadystatechange 函数。

2. 常见的 HTTP 状态码有哪些?

200 OK
301 Moved Permanently
302 Found
304 Not Modified
307 Temporary Redirect
400 Bad Request
401 Unauthorized
403 Forbidden
404 Not Found
410 Gone
500 Internal Server Error
501 Not Implemented

3. Post 和 get 区别?

GET 请求，请求的数据会附加在 URL 之后，以?分割 URL 和传输数据，多个参数用&连接。URL 的编码格式采用的是 ASCII 编码，而不是 uniclde，即是说所有的非 ASCII 字符都要编码之后再传输。

POST 请求：POST 请求会把请求的数据放置在 HTTP 请求包的包体中。上面的 item=bandsaw 就是实际的传输数据。

因此，GET 请求的数据会暴露在地址栏中，而 POST 请求则不会。

2、传输数据的大小

在 HTTP 规范中，没有对 URL 的长度和传输的数据大小进行限制。但是在实际开发过程中，对于 GET，特定的浏览器和服务器对 URL 的长度有限制。因此，在使用 GET 请求时，传输数据会受到 URL 长度的限制。

对于 POST，由于不是 URL 传值，理论上是不会受限制的，但是实际上各个服务器会规定对 POST 提交数据大小进行限制，Apache、IIS 都有各自的配置。

3、安全性

POST 的安全性比 GET 的高。这里的安全是指真正的安全，而不同于上面 GET 提到的安全方法中的安全，上面提到的安全仅仅是不修改服务器的数据。比如，在进行登录操作，通过 GET 请求，用户名和密码都会暴露再 URL 上，因为登录页面有可能被浏览器缓存以及其他人查看浏览器的历史记录的原因，此时的用户名和密码就很容易被他人拿到了。除此之外，GET 请求提交的数据还可能会造成 Cross-site request frogery 攻击。

4.cookie 和 session 的区别?

1、cookie 数据存放在客户的浏览器上，session 数据放在服务器上。

2、cookie 不是很安全，别人可以分析存放在本地的 COOKIE 并进行 COOKIE 欺骗考虑到安全应当使用 session。

3、session 会在一定时间内保存在服务器上。当访问增多，会比较占用服务器的性能考虑到减轻服务器性能方面，应当使用 COOKIE。

4、单个 cookie 保存的数据不能超过 4K，很多浏览器都限制一个站点最多保存 20 个 cookie。

5、建议：

将登陆信息等重要信息存放为 SESSION

其他信息如果需要保留，可以放在 COOKIE 中

5.创建一个简单 tcp 服务器需要的流程

1.socket 创建一个套接字

2.bind 绑定 ip 和 port

3.listen 使套接字变为可以被动链接

4.accept 等待客户端的链接

5.recv/send 接收发送数据

爬虫和数据库部分

1.scrapy 和 scrapy-redis 有什么区别?为什么选择 redis 数据库?

1) scrapy 是一个 Python 爬虫框架，爬取效率极高，具有高度定制性，但是不支持分布式。而 scrapy-redis 一套基于 redis 数据库、运行在 scrapy 框架之上的组件，可以让 scrapy 支持分布式策略，Slaver 端共享 Master 端 redis 数据库里的 item 队列、请求队列和请求指纹集合。

2) 为什么选择 redis 数据库，因为 redis 支持主从同步，而且数据都是缓存在内存中的，所以基于 redis 的分布式爬虫，对请求和数据的高频读取效率非常高。

2. 你用过的爬虫框架或者模块有哪些?谈谈他们的区别或者优缺点?

Python 自带：urllib，urllib2

第 三 方：requests

框 架：Scrapy

urllib 和 urllib2 模块都做与请求 URL 相关的操作，但他们提供不同的功能。

urllib2.：urllib2.urlopen 可以接受一个 Request 对象或者 url，(在接受 Request 对象时候，并以此可以来设置一个 URL 的 headers)，urllib.urlopen 只接收一个 url

urllib 有 urlencode,urllib2 没有，因此总是 urllib，urllib2 常会一起使用的原因

scrapy 是封装起来的框架，他包含了下载器，解析器，日志及异常处理，基于多线程， twisted 的方式处理，对于固定单个网站的爬取开发，有优势，但是对于多网站爬取 100 个网站，并发及分布式处理方面，不够灵活，不便调整与括展。

request 是一个 HTTP 库， 它只是用来，进行请求，对于 HTTP 请求，他是一个强大的库，下载，解析全部自己处理，灵活性更高，高并发与分布式部署也非常灵活，对于功能可以更好实现.

Scrapy 优缺点：

优点：

scrapy 是异步的

采取可读性更强的 xpath 代替正则

强大的统计和 log 系统

同时在不同的 url 上爬行

支持 shell 方式，方便独立调试

写 middleware,方便写一些统一的过滤器

通过管道的方式存入数据库

缺点：

基于 python 的爬虫框架，扩展性比较差

基于 twisted 框架，运行中的 exception 是不会干掉 reactor，并且异步框架出错后是不会停掉其他任务的，数据出错后难以察觉。

3.你常用的 mysql 引擎有哪些?各引擎间有什么区别?

主要 MySQL 与 InnoDB 两个引擎，其主要区别如下：

一、InnoDB 支持事务，MySQL 不支持，这一点是非常之重要。事务是一种高级的处理方式，如在一些列增删改中只要哪个出错还可以回滚还原，而 MySQL 就不可以了;

二、MySQL 适合查询以及插入为主的应用，InnoDB 适合频繁修改以及涉及到安全性较高的应用;

三、InnoDB 支持外键，MySQL 不支持;

四、MySQL 是默认引擎，InnoDB 需要指定;

五、InnoDB 不支持 FULLTEXT 类型的索引;

六、InnoDB 中不保存表的行数，如 select count(*) from table 时，InnoDB;需要

扫描一遍整个表来计算有多少行，但是 MySQL 只要简单的读出保存好的行数即

可。注意的是，当 count(*)语句包含 where 条件时 MySQL 也需要扫描整个表;

七、对于自增长的字段，InnoDB 中必须包含只有该字段的索引，但是在 MySQL 表中可以和其他字段一起建立联合索引;

八、清空整个表时，InnoDB 是一行一行的删除，效率非常慢。MySQL 则会重建表;

九、InnoDB 支持行锁(某些情况下还是锁整表，如 update table set a=1 where user like '%lee%'

4.描述下 scrapy 框架运行的机制?

从 start_urls 里获取第一批 url 并发送请求，请求由引擎交给调度器入请求队列，获取完毕后，调度器将请求队列里的请求交给下载器去获取请求对应的响应资源，并将响应交给自己编写的解析方法做提取处理：1. 如果提取出需要的数据，则交给管道文件处理;2. 如果提取出 url，则继续执行之前的步骤(发送 url 请求，并由引擎将请求交给调度器入队列...)，直到请求队列里没有请求，程序结束。

5.什么是关联查询，有哪些?

将多个表联合起来进行查询，主要有内连接、左连接、右连接、全连接(外连接)

6.写爬虫是用多进程好?还是多线程好？为什么?

IO 密集型代码(文件处理、网络爬虫等)，多线程能够有效提升效率(单线程下有 IO 操作会进行 IO 等待，造成不必要的时间浪费，而开启多线程能在线程 A 等待时，自动切换到线程 B，可以不浪费 CPU 的资源，从而能提升程序执行效率)。在实际的数据采集过程中，既考虑网速和响应的问题，也需要考虑自身机器的硬件情况，来设置多进程或多线程

7.数据库的优化?

1. 优化索引、SQL 语句、分析慢查询;

2. 设计表的时候严格根据数据库的设计范式来设计数据库;

3. 使用缓存，把经常访问到的数据而且不需要经常变化的数据放在缓存中，能节约磁盘 IO;

4. 优化硬件;采用 SSD，使用磁盘队列技术(RAID0,RAID1,RDID5)等;

5. 采用 MySQL 内部自带的表分区技术，把数据分层不同的文件，能够提高磁盘的读取效率;

6. 垂直分表;把一些不经常读的数据放在一张表里，节约磁盘 I/O;

7. 主从分离读写;采用主从复制把数据库的读操作和写入操作分离开来;

8. 分库分表分机器(数据量特别大)，主要的的原理就是数据路由;

9. 选择合适的表引擎，参数上的优化;

10. 进行架构级别的缓存，静态化和分布式;

11. 不采用全文索引;

12. 采用更快的存储方式，例如 NoSQL 存储经常访问的数据

8.常见的反爬虫和应对方法?

1).通过 Headers 反爬虫

从用户请求的 Headers 反爬虫是最常见的反爬虫策略。很多网站都会对 Headers 的 User-Agent 进行检测，还有一部分网站会对 Referer 进行检测(一些资源网站的防盗链就是检测 Referer)。如果遇到了这类反爬虫机制，可以直接在爬虫中添加 Headers，将浏览器的 User-Agent 复制到爬虫的 Headers 中;或者将 Referer 值修改为目标网站域名。对于检测 Headers 的反爬虫，在爬虫中修改或者添加 Headers 就能很好的绕过。

2).基于用户行为反爬虫

还有一部分网站是通过检测用户行为，例如同一 IP 短时间内多次访问同一页面，或者同一账户短时间内多次进行相同操作。

大多数网站都是前一种情况，对于这种情况，使用 IP 代理就可以解决。可以专门写一个爬虫，爬取网上公开的代理 ip，检测后全部保存起来。这样的代理 ip 爬虫经常会用到，最好自己准备一个。有了大量代理 ip 后可以每请求几次更换一个 ip，这在 requests 或者 urllib2 中很容易做到，这样就能很容易的绕过第一种反爬虫。

对于第二种情况，可以在每次请求后随机间隔几秒再进行下一次请求。有些有逻辑漏洞的网站，可以通过请求几次，退出登录，重新登录，继续请求来绕过同一账号短时间内不能多次进行相同请求的限制。

3).动态页面的反爬虫

上述的几种情况大多都是出现在静态页面，还有一部分网站，我们需要爬取的数据是通过 ajax 请求得到，或者通过 JavaScript 生成的。首先用 Fiddler 对网络请求进行分析。如果能够找到 ajax 请求，也能分析出具体的参数和响应的具体含义，我们就能采用上面的方法，直接利用 requests 或者 urllib2 模拟 ajax 请求，对响应的 json 进行分析得到需要的数据。

能够直接模拟 ajax 请求获取数据固然是极好的，但是有些网站把 ajax 请求的所有参数全部加密了。我们根本没办法构造自己所需要的数据的请求。这种情况下就用 selenium+phantomJS，调用浏览器内核，并利用 phantomJS 执行 js 来模拟人为操作以及触发页面中的 js 脚本。从填写表单到点击按钮再到滚动页面，全部都可以模拟，不考虑具体的请求和响应过程，只是完完整整的把人浏览页面获取数据的过程模拟一遍。

用这套框架几乎能绕过大多数的反爬虫，因为它不是在伪装成浏览器来获取数据(上述的通过添加 Headers 一定程度上就是为了伪装成浏览器)，它本身就是浏览器，phantomJS 就是一个没有界面的浏览器，只是操控这个浏览器的不是人。利 selenium+phantomJS 能干很多事情，例如识别点触式(12306)或者滑动式的验证码，对页面表单进行暴力破解等。

9.分布式爬虫主要解决什么问题?

1)ip

2)带宽

3)cpu

4)io

10.爬虫过程中验证码怎么处理?

1.scrapy 自带

2.付费接口
