Python面试必须要看的15个问题

3 年前 19946

> 本文由EarlGrey@编程派独家编译，转载请务必注明作者及出处。 原文：Sheena@codementor 译文：编程派

引言

想找一份Python开发工作吗？那你很可能得证明自己知道如何使用Python。下面这些问题涉及了与Python相关的许多技能，问题的关注点主要是语言本身，不是某个特定的包或模块。每一个问题都可以扩充为一个教程，如果可能的话。某些问题甚至会涉及多个领域。

我之前还没有出过和这些题目一样难的面试题，如果你能轻松地回答出来的话，赶紧去找份工作吧！

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）。

推荐阅读 热门文章 随机

- 20天持续压测，云存储性能哪家更强?
- 国内公有云大幅降价后，首份一手云计算产品评测报告
- Python进阶、求职必看的前辈经验分享
- 硅谷码农用Python写了个机器人，租到了让女友满意的房子
- 使用 Python 进行科学计算：NumPy入门
- 十分钟入门Matplotlib
- 从零开发一个小游戏：PyGame 入门
- 好用！在 Notebook 中使用 Sublime Text 快捷键
- 十张GIFs让你弄懂递归等概念

热门标签

IDE PyCon 编译 Flask Codewars Postgresql Django Docker Git 程序员 开发库 漫画 编码风格

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:
A6 = [[0, 0], [1, 1], [2, 4], [3, 9], [4, 16], [5, 25]
```

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 call
for i in range(2):
    l.append(i*i)

print l             # [0, 1]

l = [3,2,1]         # the second call
for i in range(3):
    l.append(i*i)

print l             # [3, 2, 1, 0, 1, 4]

l = l_mem           # the third call
for 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 datetime
datetime.datetime.now = lambda: datetime.datetime(2012
```

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':
f(a=1,b=2,c=3,zzz="hi")     # () {'a': 1, 'c': 3, 'b':
f(1,2,3,a=1,b=2,c=3)        # (1, 2, 3) {'a': 1, 'c':

f(*l,**d)                   # (1, 2, 3) {'a': 7, 'c':
f(*t,**d)                   # (4, 5, 6) {'a': 7, 'c':
f(1,2,*t)                   # (1, 2, 4, 5, 6) {}
```

```
f(q="winning",**d)           # () {'a': 7, 'q': 'winnin
f(1,2,*t,q="winning",**d)    # (1, 2, 4, 5, 6) {'a': 7,

def f2(arg1,arg2,*args,**kwargs): print arg1,arg2, arg

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
f2(1,2,3,a=1,b=2,c=3)            # 1 2 (3,) {'a': 1, 'c

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

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_decorator
def my_func(stuff):
    do_things
Is 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(
    @classmethod
    def class_method(*args,**kwargs):
        print "calling class_method({0},{1})".format(a
    @staticmethod
    def static_method(*args,**kwargs):
        print "calling static_method({0},{1})".format(
    @property
    def some_property(self,*args,**kwargs):
        print "calling some_property getter({0},{1},{2
        return self._some_property
    @some_property.setter
    def some_property(self,*args,**kwargs):
        print "calling some_property setter({0},{1},{2
        self._some_property = args[0]
    @property
    def some_other_property(self,*args,**kwargs):
        print "calling some_other_property getter({0},
        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

o.normal_method
# <bound method MyClass.normal_method of <__main__.MyC

o.normal_method()
# normal_method((<__main__.MyClass instance at 0x7fdd2

o.normal_method(1,2,x=3,y=4)
# normal_method((<__main__.MyClass instance at 0x7fdd2
```

```
# 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

o.class_method()
# class_method((<class __main__.MyClass at 0x7fdd2536a

o.class_method(1,2,x=3,y=4)
# class_method((<class __main__.MyClass at 0x7fdd2536a

# 静态方法（static method）中除了你调用时传入的参数以外，没有

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
# “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
# 'properties are nice'
# “属性”是很好的功能

o.some_property()
# calling some_property getter(<__main__.MyClass insta
# 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
# 'VERY nice'

# o.some_other_property()
# calling some_other_property getter(<__main__.MyClass
# 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 objec

o.some_property
# calling some_property getter(<__main__.MyClass objec
# '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
```

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)
oChild1.append(oChild4)
oChild1.append(oChild5)
oChild2.append(oChild6)
oChild4.append(oChild7)
oChild3.append(oChild8)
oChild3.append(oChild9)
oChild6.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 cProfile
lIn = [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
        1    0.009    0.009    0.044    0.044 <stdin>:
        1    0.001    0.001    0.045    0.045 <string>
        1    0.000    0.000    0.000    0.000 {method
        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
        1    0.008    0.008    0.023    0.023 <stdin>:
        1    0.001    0.001    0.024    0.024 <string>
        1    0.000    0.000    0.000    0.000 {method
        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
        1    0.016    0.016    0.054    0.054 <stdin>:
        1    0.001    0.001    0.055    0.055 <string>
        1    0.000    0.000    0.000    0.000 {method
        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。

结语

我给出的这些问题时，有意涉及了多个领域。而且答案也是特意写的较为啰嗦。在编程面试中，你需要展示你对语言的理解，如果你能简要地说清楚，那请务必那样做。我尽量在答案中提供了足够的信息，即使是你之前从来没有了解过这些领域，你也可以从答案中学到些东西。我希望本文能够帮助你找到满意的工作。

加油!

上一篇 下一篇

相关文章