经典7python面试题

今天是三月第一天，也是金三银四的第一天，对于正在找工作的人来说，正是好的时候。最近刚好看到这篇文章，感觉原作者总结的不错，所以就借此机会分享给大家，希望可以帮助那些打算或者正在找工作的朋友们。

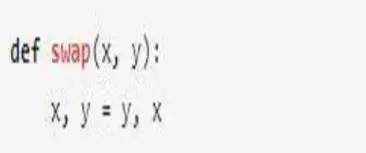
**Python面试一：交换变量值**

平时时不时会面面实习生，大多数的同学在学校里都已经掌握了Python。面试的时候要求同学们实现一个简单的函数，交换两个变量的值，大多数的同学给出的都是如下的答案

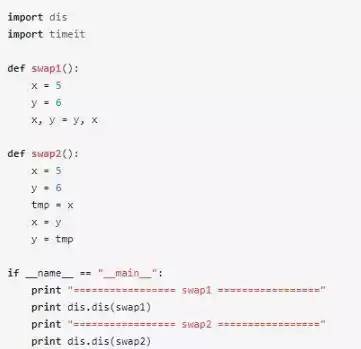


实际上，Python中还有更简洁的更具Python风格的实现，如下

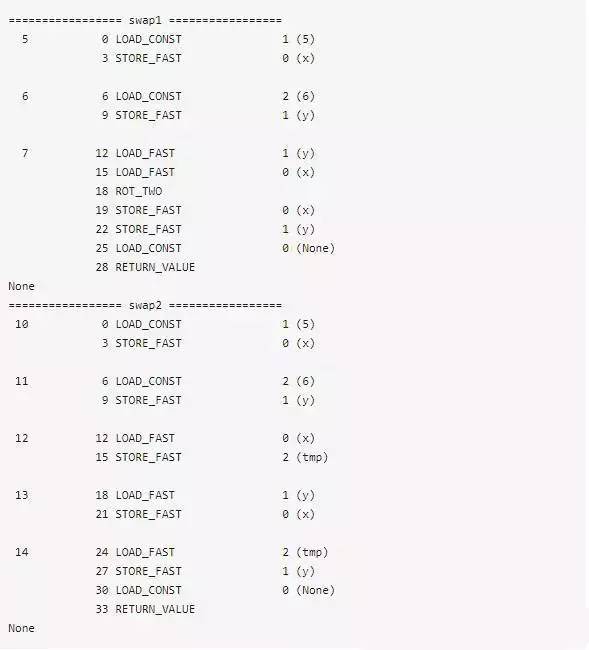
相比前一种方法，后一种方法节省一个中间变量，在性能上也优于前一种方法。



我们从Python的字节码来深入分析一下原因。



dis是个反汇编工具，将Python代码翻译成字节码指令。这里的输出如下

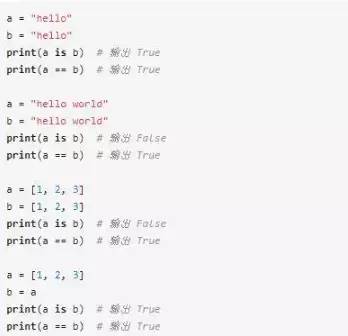


通过字节码可以看到，swap1和swap2最大的区别在于，swap1中通过ROT\_TWO交换栈顶的两个元素实现x和y值的互换，swap2中引入了tmp变量，多了一次LOAD\_FAST, STORE\_FAST的操作。执行一个ROT\_TWO指令比执行一个LOAD\_FAST+STORE\_FAST的指令快，这也是为什么swap1比swap2性能更好的原因。

**Python面试 二：is 和 == 的区别**

面试实习生的时候，当问到 is 和 == 的区别时，很多同学都答不上来，搞不清两者什么时候返回一致，什么时候返回不一致。本文我们来看一下这两者的区别。

我们先来看几个例子：



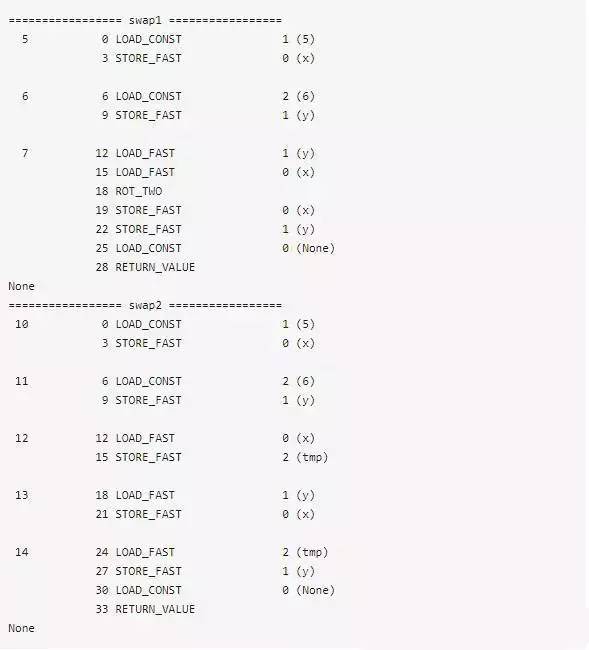
上面的输出结果中为什么有的 is 和 == 的结果相同，有的不相同呢？我们来看下官方文档中对于 is 和 == 的解释。

官方文档中说 is 表示的是对象标示符（object identity），而 == 表示的是相等（equality）。is 的作用是用来检查对象的标示符是否一致，也就是比较两个对象在内存中的地址是否一样，而 == 是用来检查两个对象是否相等。

我们在检查 a is b 的时候，其实相当于检查 id(a) == id(b)。而检查 a == b 的时候，实际是调用了对象 a 的 \_\_eq()\_\_ 方法，a == b 相当于 a.\_\_eq\_\_(b)。

一般情况下，如果 a is b 返回True的话，即 a 和 b 指向同一块内存地址的话，a == b 也返回True，即 a 和 b 的值也相等。

好了，看明白上面的解释后，我们来看下前面的几个例子



打印出 id(a) 和 id(b) 后就很清楚了。只要 a 和 b 的值相等，a == b 就会返回True，而只有 id(a) 和 id(b) 相等时，a is b 才返回 True。

这里还有一个问题，为什么 a 和 b 都是 "hello" 的时候，a is b 返回True，而 a 和 b都是 "hello world" 的时候，a is b 返回False呢？

这是因为前一种情况下Python的字符串驻留机制起了作用。对于较小的字符串，为了提高系统性能Python会保留其值的一个副本，当创建新的字符串的时候直接指向该副本即可。所以 "hello" 在内存中只有一个副本，a 和 b 的 id 值相同，而 "hello world" 是长字符串，不驻留内存，Python中各自创建了对象来表示 a 和 b，所以他们的值相同但 id 值不同。

同学指出：intern机制和字符串长短无关，在交互模式下，每行字符串字面量都会申请一个新字符串，但是只含大小写字母、数字和下划线的会被intern，也就是维护了一张dict来使得这些字符串全局唯一）

总结一下，is 是检查两个对象是否指向同一块内存空间，而 == 是检查他们的值是否相等。可以看出，is 是比 == 更严格的检查，is 返回True表明这两个对象指向同一块内存，值也一定相同。

看到这里，大家是不是搞懂了 is 和 == 的区别呢？

那我们深入一步来思考一下下面这个问题：

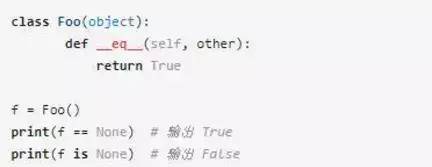
Python里和None比较时，为什么是 is None 而不是 == None 呢？

**Python面试三：可变对象和不可变对象**

上一个面试题：Python面试之 is 和 == 的区别的最后留了一个问题：

Python里和None比较时，为什么是 is None 而不是 == None 呢？

这是因为None在Python里是个单例对象，一个变量如果是None，它一定和None指向同一个内存地址。而 == None背后调用的是\_\_eq\_\_，而\_\_eq\_\_可以被重载，下面是一个 is not None但 == None的例子



好了，解答就到这里，我们开始本篇的正题。

Python中有可变对象和不可变对象之分。可变对象创建后可改变但地址不会改变，即变量指向的还是原来的变量；不可变对象创建之后便不能改变，如果改变则会指向一个新的对象。

Python中dict、list是可变对象，str、int、tuple、float是不可变对象。

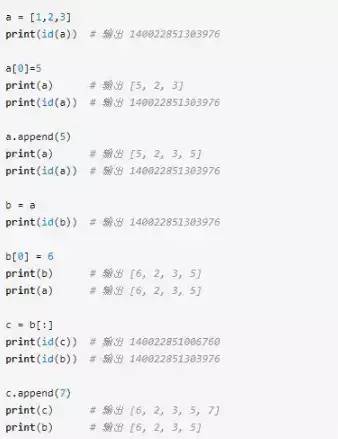
来看一个字符串的例子



上面的例子里，修改a指向的对象的值会导致抛出异常。

执行 a = a + " world"时，先计算等号右边的表达式，生成一个新的对象赋值到变量a，因此a指向的对象发生了改变，id(a) 的值也与原先不同。

再来看一个列表的例子



上面对a修改元素、添加元素，变量a还是指向原来的对象。

将a赋值给b后，变量b和a都指向同一个对象，因此修改b的元素值也会影响a。

变量c是对b的切片操作的返回值，切片操作相当于浅拷贝，会生成一个新的对象，因此c指向的对象不再是b所指向的对象，对c的操作不会改变b的值。

理解了上面不可变对象和可变对象的区别后，我们再来看一个有趣的问题



明明group1和group2是不同的对象（id值不同），为什么调用group2的add\_member方法会影响group1的members？

其中的奥妙就在于\_\_init\_\_函数的第二个参数是默认参数，默认参数的默认值在函数创建的时候就生成了，每次调用都是用了这个对象的缓存。我们检查id(group1.mebers)和id(group2.members)，可以发现他们是相同的

print(id(group1.members)) # 输出 140127132522040

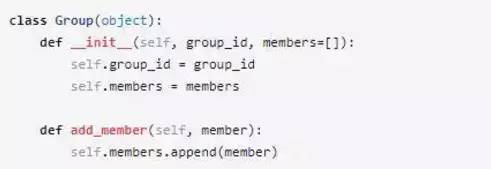
print(id(group2.members)) # 输出 140127132522040

所以，group1.members和group2.members指向了同一个对象，对group2.members的修改也会影响group1.members。

那么问题来了，怎样修改代码才能解决上面默认参数的问题呢？

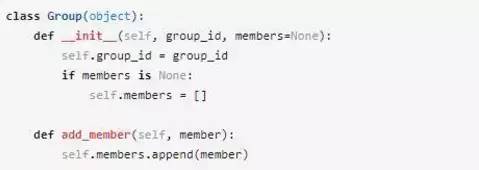
**Python面试四：连接字符串用join还是+**

上一个面试题：Python面试之可变对象和不可变对象的最后留了一个问题



上述代码中默认参数值对象会被缓存，造成Group类型的对象共享同一个members列表，怎样才能解决这个问题呢？

其实很简单，只要传入None作为默认参数，在创建对象的时候动态生成列表，如下



这样对于不同的group对象，它们的members也是不同的对象，所以不会再出现更新一个group对象的members也会更新另外一个group对象的members了。

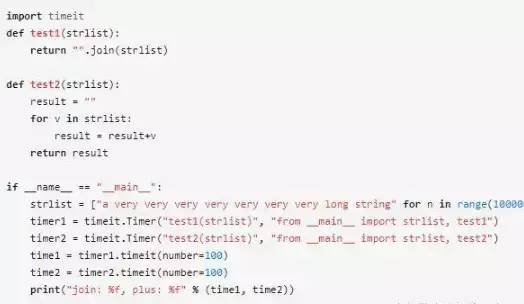
本篇要讲的是，连接字符串的时候可以用join也可以用+，但这两者有没有区别呢？

我们先来看一下用join和+连接字符串的例子



两者的结果是一样，那么考虑这样一个问题，这两者在性能上有区别吗？

我们来做个实验，比较下join和+的性能



上面的程序有如下的输出

join: 0.116944, plus: 0.394379

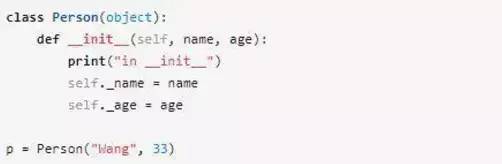
可以看到，join的性能明显好于+。这是为什么呢？

原因是这样的，上一篇Python面试之可变对象和不可变对象中讲过字符串是不可变对象，当用操作符+连接字符串的时候，每执行一次+都会申请一块新的内存，然后复制上一个+操作的结果和本次操作的右操作符到这块内存空间，因此用+连接字符串的时候会涉及好几次内存申请和复制。而join在连接字符串的时候，会先计算需要多大的内存存放结果，然后一次性申请所需内存并将字符串复制过去，这是为什么join的性能优于+的原因。所以在连接字符串数组的时候，我们应考虑优先使用join。

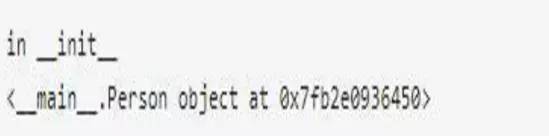
**Python面试五：理解\_\_new\_\_和\_\_init\_\_的区别**

很多同学都以为Python中的\_\_init\_\_是构造方法，但其实不然，Python中真正的构造方法是\_\_new\_\_。\_\_init\_\_和\_\_new\_\_有什么区别？本文就来探讨一下。

我们先来看一下\_\_init\_\_的用法

****

上面的代码会输出如下的结果

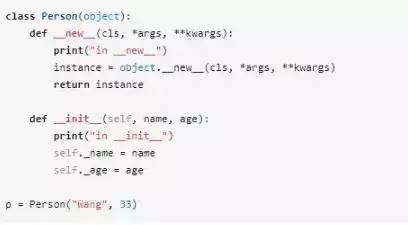


那么我们思考一个问题，Python中要实现Singleton怎么实现，要实现工厂模式怎么实现？

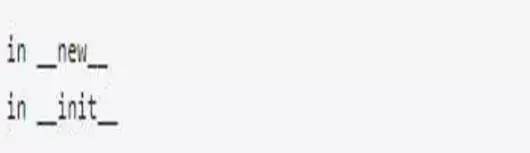
用\_\_init\_\_函数似乎没法做到呢~

实际上，\_\_init\_\_函数并不是真正意义上的构造函数，\_\_init\_\_方法做的事情是在对象创建好之后初始化变量。真正创建实例的是\_\_new\_\_方法。

我们来看下面的例子



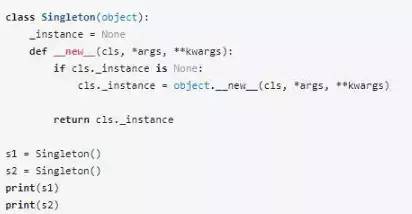
上面的代码输出如下的结果



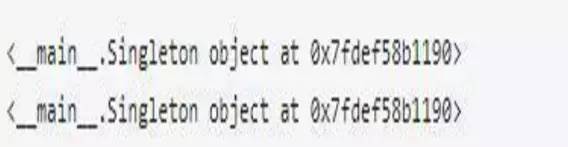
上面的代码中实例化了一个Person对象，可以看到\_\_new\_\_和\_\_init\_\_都被调用了。\_\_new\_\_方法用于创建对象并返回对象，当返回对象时会自动调用\_\_init\_\_方法进行初始化。\_\_new\_\_方法是静态方法，而\_\_init\_\_是实例方法。

好了，理解\_\_new\_\_和\_\_init\_\_的区别后，我们再来看一下前面提出的问题，用Python怎么实现Singleton，怎么实现工厂模式？

先来看Singleton



上面的代码输出



可以看到s1和s2都指向同一个对象，实现了单例模式。

再来看下工厂模式的实现



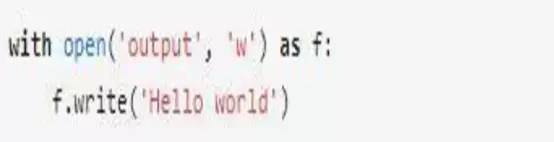
上面的代码输出



看完上面两个例子，大家是不是对\_\_new\_\_和\_\_init\_\_的区别有了更深入的理解？

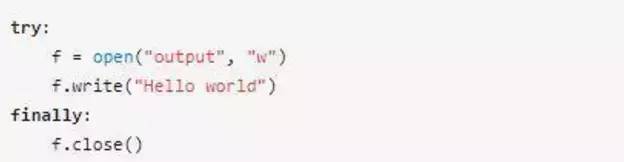
**Python面试六：with与上下文管理器With基本语法**

Python老司机应该对下面的语法不陌生



上面的代码往output文件写入了Hello world字符串，with语句会在执行完代码块后自动关闭文件。这里无论写文件的操作成功与否，是否有异常抛出，with语句都会保证文件被关闭。

如果不用with，我们可能要用下面的代码实现类似的功能



可以看到使用了with的代码比上面的代码简洁许多。

上面的with代码背后发生了些什么？我们来看下它的执行流程

首先执行open('output', 'w')，返回一个文件对象

调用这个文件对象的\_\_enter\_\_方法，并将\_\_enter\_\_方法的返回值赋值给变量f

执行with语句体，即with语句包裹起来的代码块

不管执行过程中是否发生了异常，执行文件对象的\_\_exit\_\_方法，在\_\_exit\_\_方法中关闭文件。

这里的关键在于open返回的文件对象实现了\_\_enter\_\_和\_\_exit\_\_方法。一个实现了\_\_enter\_\_和\_\_exit\_\_方法的对象就称之为上下文管理器。

上下文管理器

上下文管理器定义执行 with 语句时要建立的运行时上下文，负责执行 with 语句块上下文中的进入与退出操作。\_\_enter\_\_方法在语句体执行之前进入运行时上下文，\_\_exit\_\_在语句体执行完后从运行时上下文退出。

在实际应用中，\_\_enter\_\_一般用于资源分配，如打开文件、连接数据库、获取线程锁；\_\_exit\_\_一般用于资源释放，如关闭文件、关闭数据库连接、释放线程锁。

自定义上下文管理器

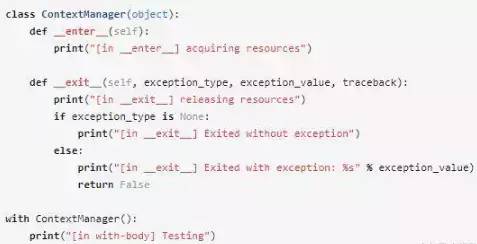
既然上下文管理器就是实现了\_\_enter\_\_和\_\_exit\_\_方法的对象，我们能不能定义自己的上下文管理器呢？答案是肯定的。

我们先来看下\_\_enter\_\_和\_\_exit\_\_方法的定义：

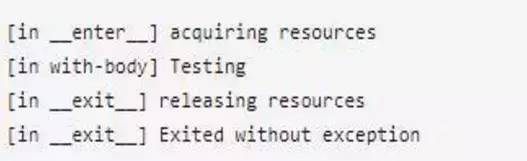
\_\_enter\_\_() - 进入上下文管理器的运行时上下文，在语句体执行前调用。如果有as子句，with语句将该方法的返回值赋值给 as 子句中的 target。

\_\_exit\_\_(exception\_type, exception\_value, traceback) - 退出与上下文管理器相关的运行时上下文，返回一个布尔值表示是否对发生的异常进行处理。如果with语句体中没有异常发生，则\_\_exit\_\_的3个参数都为None，即调用 \_\_exit\_\_(None, None, None)，并且\_\_exit\_\_的返回值直接被忽略。如果有发生异常，则使用 sys.exc\_info 得到的异常信息为参数调用\_\_exit\_\_(exception\_type, exception\_value, traceback)。出现异常时，如果\_\_exit\_\_(exception\_type, exception\_value, traceback)返回 False，则会重新抛出异常，让with之外的语句逻辑来处理异常；如果返回 True，则忽略异常，不再对异常进行处理。

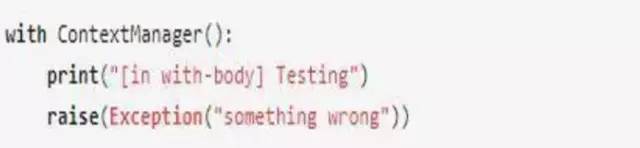
理解了\_\_enter\_\_和\_\_exit\_\_方法后，我们来自己定义一个简单的上下文管理器。这里不做实际的资源分配和释放，而用打印语句来表明当前的操作。



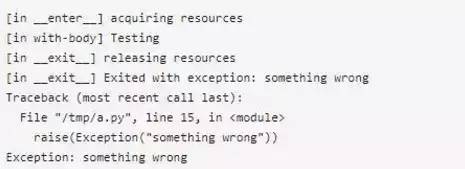
运行上面的代码，会得到如下的输出



我们在with语句体中人为地抛出一个异常



会得到如下的输出



如我们所期待，with语句体中抛出异常，\_\_exit\_\_方法中exception\_type不为None，\_\_exit\_\_方法返回False，异常被重新抛出。

以上，我们通过实现\_\_enter\_\_和\_\_exit\_\_方法来实现了一个自定义的上下文管理器。

contextlib库

除了上面的方法，我们也可以使用contextlib库来自定义上下文管理器。如果用contextlib来实现，可以用下面的代码来实现类似的上下文管理器



上面的代码涉及到装饰器（@contextmanager），生成器（yield），有点难读。这里yield之前的代码相当于\_\_enter\_\_方法，在进入with语句体之前执行，yield之后的代码相当于\_\_exit\_\_方法，在退出with语句体的时候执行。

**Python面试七：你真的理解finally了吗？**

无论try语句中是否抛出异常，finally中的语句一定会被执行。我们来看下面的例子：



不论try中写文件的过程中是否有异常，finally中关闭文件的操作一定会执行。由于finally的这个特性，finally经常被用来做一些清理工作。

我们再来看下面的例子



这个例子中 func1() 和 func2() 返回什么呢？

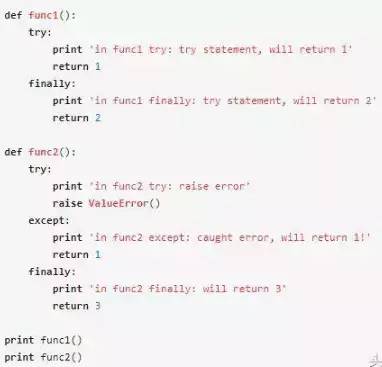
答案是 func1() 返回2， func2() 返回3。为什么是这样的呢？我们先来看一段Python官网上对于finally的解释：

A finally clause is always executed before leaving the try statement, whether an exception has occurred or not. When an exception has occurred in the try clause and has not been handled by an except clause (or it has occurred in a except or else clause), it is re-raised after the finally clause has been executed. The finally clauseis also executed “on the way out” when any other clause of the try statement is left via a break, continue or return statement.

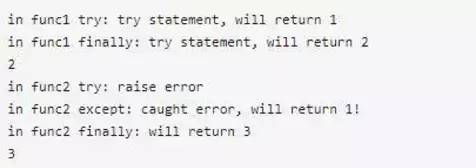
重点部分用粗体标出了，翻成中文就是try块中包含break、continue或者return语句的，在离开try块之前，finally中的语句也会被执行。

所以在上面的例子中，func1() 中，在try块return之前，会执行finally中的语句，try中的return被忽略了，最终返回的值是finally中return的值。func2() 中，try块中抛出异常，被except捕获，在except块return之前，执行finally中的语句，except中的return被忽略，最终返回的值是finally中return的值。

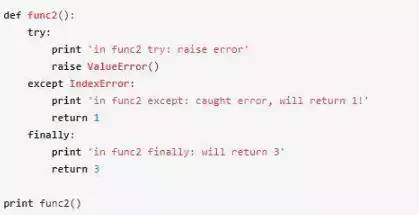
我们在上面的例子中加入print语句，可以更清楚地看到过程



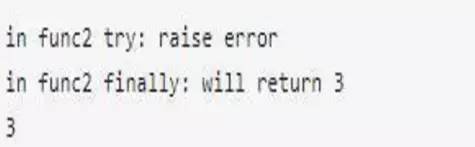
上面的代码输出



我们对上面的func2做一些修改，如下



输出如下



try中抛出的异常是ValueError类型的，而except中定位的是IndexError类型的，try中抛出的异常没有被捕获到，所以except中的语句没有被执行，但不论异常有没有被捕获，finally还是会执行，最终函数返回了finally中的返回值3。

这里还可以看到另外一个问题。try中抛出的异常没有被捕获到，按理说当finally执行完毕后，应该被再次抛出，但finally里执行了return，导致异常被丢失。

可以看到在finally中使用return会导致很多问题。实际应用中，不推荐在finally中使用return返回。