```
'''
                    /题目目录/

        (一)、这两个参数是什么意思：*args，**kwargs？
        (二)、谈一谈Python的装饰器（decorator）
        (三)、简要描述Python的垃圾回收机制（garbage collection）
        (四)：Python多线程（multi-threading）。这是个好主意吗？
        (五)、 说明os,sys模块不同，并列举常用的模块方法？
        (六)、什么是lambda表达式？它有什么好处？
        (七)、Python中pass语句的作用是什么？
        (八)、Python是如何进行类型转换的？
        (九)、Python里面如何拷贝一个对象？
        (十)、__new__和__init__的区别。
        (十一)、Python中单下划线和双下划线分别是什么？
        (十二)、说一说Python自省。
        ......
'''
```

（一）、这两个参数是什么意思：*args，kwargs？我们为什么要使用它们？**

答：如果我们不确定往一个函数中传入多少参数，或者我们希望以元组（tuple）或者列表（list）的形式传参数的时候，我们可以使用*args（单星号）。如果我们不知道往函数中传递多少个关键词参数或者想传入字典的值作为关键词参数的时候我们可以使用**kwargs（双星号），args、kwargs 两个标识符是约定俗成的用法。

另一种答法：当函数的参数前面有一个星号*号的时候表示这是一个可变的位置参数，两个星号**表示这个是一个可变的关键词参数。星号*把序列或者集合解包（unpack）成位置参数，两个星号**把字典解包成关键词参数。

```
tempList = [1,2,3]
tempTuple = (2,3,4)
tempDict = {'s':3,'m':4,'c':5}

def testFunc(*args,**kwargs):
    print args,kwargs

testFunc() #() {}
testFunc(*tempList) #(1, 2, 3) {}
testFunc(*tempTuple) #(2,3,4) {}
testFunc(*tempDict) #('s','m','c'),{}
testFunc(**tempDict) #(),{'s':3,'m':4,'c':5}
testFunc(*tempList,**tempDict) #(1,2,3) {'s':3,'m':4,'c':5}
testFunc(0) #(0,) {}
testFunc(0,*tempList) #(0,1,2,3) {}
testFunc(0,**tempDict) #(0,) {'s': 3, 'm': 4, 'c': 5}
testFunc(0,*tempList,tempName = 'bye',**tempDict)
#(0, 1, 1, 2, 3) {'s': 3, 'm': 4, 'c': 5, 'tempName': 'bye'}
```

（二）、谈一谈 Python 的装饰器（decorator）

装饰器本质上是一个 Python 函数，它可以让其它函数在不作任何变动的情况下增加额外功能，装饰器的返回值也是一个函数对象。它经常用于有切面需求的场景。比如：插入日志、性能测试、事务处理、缓存、权限校验等。有了装饰器我们就可以抽离出大量的与函数功能无关的雷同代码进行重用。

有关于具体的装饰器的用法看这里：[装饰器 - 廖雪峰的官方网站]

```
import functools
def log(text):
    if isinstance(text, basestring):
        def decorator(func):
            functools.wraps(func)
            def wrapper(*args,**kwargs):
                func(*args,**kwargs)
                print '%s %s'%(text,func.__name__)
            print '%s %s'%('start',func.__name__)
            return wrapper
        return decorator
    else:
        functools.wraps(func)
        def wrapper(*args,**kwargs):
            print 'Call %s:'%(func.__name__)
            return func(*args,**kwargs)
        return wrapper

@log('end')
def now():
    print '2016-11-10'
now()
```

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

Python 中的垃圾回收是以引用计数为主，标记-清除和分代收集为辅。

引用计数：Python 在内存中存储每个对象的引用计数，如果计数变成 0，该对象就会消失，分配给该对象的内存就会释放出来。

标记-清除：一些容器对象，比如 list、dict、tuple，instance 等可能会出现引用循环，对于这些循环，垃圾回收器会定时回收这些循环（对象之间通过引用（指针）连在一起，构成一个有向图，对象构成这个有向图的节点，而引用关系构成这个有向图的边）。

分代收集：Python 把内存根据对象存活时间划分为三代，对象创建之后，垃圾回收器会分配它们所属的代。每个对象都会被分配一个代，而被分配更年轻的代是被优先处理的，因此越晚创建的对象越容易被回收。

如果你想要深入了解 Python 的 GC 机制，点击这里：[转载]Python 垃圾回收机制--完美讲解!

（四）、Python 多线程（multi-threading）。这是个好主意吗？

Python 并不支持真正意义上的多线程，Python 提供了多线程包。Python 中有一个叫 Global Interpreter Lock（GIL）的东西，它能确保你的代码中永远只有一个线程在执行。经过 GIL 的处理，会增加执行的开销。这就意味着如果你先要提高代码执行效率，使用 threading 不是一个明智的选择，当然如果你的代码是 IO 密集型，多线程可以明显提高效率，相反如果你的代码是 CPU 密集型的这种情况下多线程大部分是鸡肋。

想要深入详细了解多线程，点击这里：详解 Python 中的多线程编程_python

想了解一下 IO 密集和 CPU 密集可以点击这里：CPU-bound(计算密集型) 和 I/O bound(I/O 密集型)

（五）、 说明 os, sys 模块不同，并列举常用的模块方法？

官方文档：

os 模板提供了一种方便的使用操作系统函数的方法

sys 模板可供访问由解释器使用或维护的变量和与解释器交互的函数

另一种回答：

os 模块负责程序与操作系统的交互，提供了访问操作系统底层的接口。sys 模块负责程序与 Python 解释器的交互，提供了一系列的函数和变量用户操作 Python 运行时的环境。一些常用的方法：

```
    os.remove()           删除文件
    os.rename()           重命名文件
    os.walk()             生成目录树下的所有文件名
    os.chdir()            改变目录
    os.mkdir/makedirs     创建目录/多层目录
    os.rmdir/removedirs   删除目录/多层目录
    os.listdir()          列出指定目录的文件
    os.getcwd()           取得当前工作目录
    os.chmod()            改变目录权限
    os.path.basename()    去掉目录路径，返回文件名
    os.path.dirname()     去掉文件名，返回目录路径
    os.path.join()        将分离的各部分组合成一个路径名
    os.path.split()       返回（dirname(),basename())元组
    os.path.splitext()    (返回filename,extension)元组
    os.path.getatime\ctime\mtime分别返回最近访问、创建、修改时间
    os.path.getsize()     返回文件大小
    os.path.exists()      是否存在
    os.path.isabs()       是否为绝对路径
    os.path.isdir()       是否为目录
    os.path.isfile()      是否为文件
'''
    sys.argv           命令行参数List，第一个元素是程序本身路径
    sys.modules.keys() 返回所有已经导入的模块列表
    sys.exc_info()     获取当前正在处理的异常类,exc_type、exc_value、exc_traceback当前处理的异常详细信息
    sys.exit(n)        退出程序，正常退出时exit(0)
    sys.hexversion     获取Python解释程序的版本值，16进制格式如：0x020403F0
    sys.version        获取Python解释程序的版本信息
    sys.maxint         最大的Int值
    sys.maxunicode     最大的Unicode值
    sys.modules        返回系统导入的模块字段，key是模块名，value是模块
    sys.path           返回模块的搜索路径，初始化时使用PYTHONPATH环境变量的值
    sys.platform       返回操作系统平台名称
    sys.stdout         标准输出
    sys.stdin          标准输入
    sys.stderr         错误输出
    sys.exc_clear()    用来清除当前线程所出现的当前的或最近的错误信息
    sys.exec_prefix    返回平台独立的python文件安装的位置
    sys.byteorder      本地字节规则的指示器，big-endian平台的值是'big',little-endian平台的值是'little'
    sys.copyright      记录python版权相关的东西
    sys.api_version    解释器的C的API版本
    sys.version_info
'''
```

（六）、什么是 lambda 表达式？它有什么好处？

简单来说，lambda 表达式通常是当你需要使用一个函数，但是又不想费脑袋去命名一个函数的时候使用，也就是通常所说的匿名函数。
lambda 表达式一般的形式是：关键词 lambda 后面紧接一个或多个参数，紧接一个冒号“：”，紧接一个表达式。lambda 表达式是一个表达式不是一个语句。

```
f = lambda x,y,z : z + y + z
print f(4,2,6)

L = {'f1':(lambda x,y:x**2+y**2),
     'f2':(lambda x,y:x**3+y**3),
     'f3':(lambda x,y:x**4+y**3)}
print L['f2'](3,2)
```

想更加详细的了解 Python 中的 Lamdba 表达式可以点击这里：Lambda 表达式有何用处？如何使用？ - Python

（七）、Python 中 pass 语句的作用是什么？

pass 语句不会执行任何操作，一般作为占位符或者创建占位程序

（八）、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为另一种类型的内置方法。

```
119 int(x [,base])          将x转换为一个整数
120 long(x [,base])         将x转换为一个长整数
121 float(x)                将x转换到一个浮点数
122 complex(real [,imag])   创建一个复数
123 str(x)                  将对象 x 转换为字符串
124 repr(x)                 将对象 x 转换为表达式字符串
125 eval(str)               用来计算在字符串中的有效Python表达式,并返回一个对象
126 tuple(s)                将序列 s 转换为一个元组
127 list(s)                 将序列 s 转换为一个列表
128 chr(x)                  将一个整数转换为一个字符
129 unichr(x)               将一个整数转换为Unicode字符
130 ord(x)                  将一个字符转换为它的整数值
131 hex(x)                  将一个整数转换为一个十六进制字符串
132 oct(x)                  将一个整数转换为一个八进制字符串
133 '''
134 print int(0),int('1')
135 print long(0),long('123456')
136 print float(0),float('12')
137 print str(0),str('12'),str([1,2]),str({'1':1,'2':2})
138 print list((1,2,3,4,5))
```

```
0 1
0 123456
0.0 12.0
0 12 [1, 2] {'1': 1, '2': 2}
[1, 2, 3, 4, 5]
```

（九）、Python 里面如何拷贝一个对象？

Python 中对象之间的赋值是按引用传递的，如果要拷贝对象需要使用标准模板中的 copy
copy. copy：浅拷贝，只拷贝父对象，不拷贝父对象的子对象。
copy. deepcop y ：深拷贝，拷贝父对象和子对象。

```
142 import copy
143 tempList = [0,1,2,[3,4]]
144 testList = tempList
145 testCopyList = copy.copy(tempList)
146 testDeepCopyList = copy.deepcopy(testList)
147
148 tempList.append('sign')
149 print testList,testCopyList,testDeepCopyList
150
151 testList[3].append('sign')
152 print testList,testCopyList,testDeepCopyList
```

```
[0, 1, 2, [3, 4], 'sign'] [0, 1, 2, [3, 4]] [0, 1, 2, [3, 4]]
[0, 1, 2, [3, 4, 'sign'], 'sign'] [0, 1, 2, [3, 4, 'sign']] [0, 1, 2, [3, 4]]
[Finished in 0.1s]
```

（十）、__new__和__init__的区别。

__init__为初始化方法，__new__方法是真正的构造函数。
__new__是实例创建之前被调用，它的任务是创建并返回该实例，是静态方法
__init__是实例创建之后被调用的，然后设置对象属性的一些初始值。
总结：__new__方法在__init__方法之前被调用，并且__new__方法的返回值将传递给__init__方法作为第一个参数，最后__init__给这个实例设置一些参数。

```
155 class TestClass(object):
156
157     def __new__(cls,*args,**kwargs):
158         print '__new__'
159         return object.__new__(cls,*args,**kwargs)
160
161     def __init__(self,testName):
162         print '__init__'
163         self.testName = testName
164
165 print TestClass('Json').testName
166
```

```
__new__
__init__
Json
[Finished in 0.1s]
```

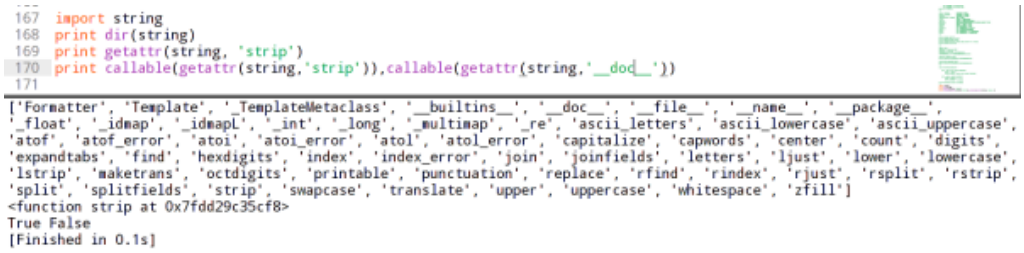
想要更加详细的了解这两个方法，请点击：Python 中的__new__及其用法

（十一）、Python 中单下划线和双下划线分别是什么？

__name__：一种约定，Python 内部的名字，用来与用户自定义的名字区分开，防止冲突
_name：一种约定，用来指定变量私有
__name：解释器用_classname__name 来代替这个名字用以区别和其他类相同的命名
想要更加详细的了解这两者的区别，请点击：Python 中的下划线（译文）

（十二）、说一说 Python 自省。

自省就是面向对象的语言所写的程序在运行时，所能知道对象的类型。简单一句话就是运行时能够获得对象的类型。比如：type()、dir()、getattr()、hasattr()、isinstance()



```
167 import string
168 print dir(string)
169 print getattr(string, 'strip')
170 print callable(getattr(string,'strip')),callable(getattr(string,'__doc__'))
171
```

```
['Formatter', 'Template', '_TemplateMetaclass', '__builtins__', '__doc__', '__file__', '__name__', '__package__',
'_float', '_idmap', '_idmapL', '_int', '_long', '_multimap', '_re', 'ascii_letters', 'ascii_lowercase', 'ascii_uppercase',
'atof', 'atof_error', 'atoi', 'atoi_error', 'atol', 'atol_error', 'capitalize', 'capwords', 'center', 'count', 'digits',
'expandtabs', 'find', 'hexdigits', 'index', 'index_error', 'join', 'joinfields', 'letters', 'ljust', 'lower', 'lowercase',
'lstrip', 'maketrans', 'octdigits', 'printable', 'punctuation', 'replace', 'rfind', 'rindex', 'rjust', 'rsplit', 'rstrip',
'split', 'splitfields', 'strip', 'swapcase', 'translate', 'upper', 'uppercase', 'whitespace', 'zfill']
<function strip at 0x7fdd29c35cf8>
True False
[Finished in 0.1s]
```

想要完整的理解 Python 自省，请点击：Python 自省（反射）指南
有关于元类以及单例模式会在后面文章中做详细的解释说明。

更多 Python 面试题：

- 七、PYTHON 一些基础面试题目总结
- Python 面试必须要看的 15 个问题