

人生苦短  
我用 *Python*





# 芝麻开门！

python 进阶	1
python 经典面试题	9
python 面试 100 例	21
python 面试题总结	101



# python 进阶

# Python 进阶

Auth: zhangxu@1000phone.com

Date: 2017-12-11

## 1. PEP8 编码规范, 以及开发中的一些惯例和建议

- 代码编排: 缩进 4 个空格, 行长 80, 空行(函数间, 函数内, 文件结尾)
- import: import 顺序; 单行不要 import 多个库; 模块内用不到的不要去 import
- 空格
- 注释
  - 行注释
  - 块注释
  - 引入外来算法或者配置时须在注释中添加源连接, 标明出处
  - 函数和类尽可能添加 `__doc__`
- 命名
  - 包名、模块名、函数名、方法名全部使用小写, 单词间用下划线连接
  - 类名、异常名使用CapWords的方式, 异常名结尾加 `Error` 或 `Warning` 后缀
  - 全局变量尽量使用大写, 一组同类型的全局变量要加上统一前缀
  - 常量全部使用大写, 单词用下划线连接
- 字符串拼接尽量使用 `join` 方式
- 语意明确、直白
  - `not xx in yy` VS `xx not in yy`
  - `a not is b` VS `a is not b`
- 一个函数只做一件事情, 并把这件事做好, 大的功能用小函数之间灵活组合来完成 (《Unix 编程艺术》)
- 函数名必须有动词, 最好是 `do_something` 的句式, 或者 `somebody_do_something` 句式
- 自定义的变量名、函数名不要与标准库中的名字冲突
- `pip install pep8`
- 练习: 规范化这段代码

```
from django.conf import settings
import sys, os
MOD=0xffffffff
def foo ( a , b = 123 ) :
    c = { 'x' : 111 , 'y' : 222 } # 定义一个字典
    d = [ 1, 3 , 5 ]
    return a , b , c
def bar(x):
    if x%2 ==0: return true
```

## 2. Python 的赋值和引用

- `==`, `is`
- `copy`, `deepcopy`
- 练习1: 说出执行结果

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

list1 = extendList(10)
list2 = extendList(123,[])
list3 = extendList('a')
```

- 练习2: 说出下面执行结果

```
from copy import copy, deepcopy
from pickle import dumps, loads

a = [1, 2, 3]
b = [a] * 3
c = copy(b)
```

```

d = deepcopy(b)
e = loads(dumps(b, 4))

b[1].append(999)
c[1].append(999)
d[1].append(999)
e[1].append(999)

```

- 自定义 deepcopy: `my_deepcopy = lambda item: loads(dumps(item, 4))`

### 3. 迭代器, 生成器, itertools, yield, 列表/字典/集合的推导

- iterator: 任何实现了 `__iter__` 和 `__next__` (python2中实现`next()`) 方法的对象都是迭代器.
  - `__iter__` 返回迭代器自身
  - `__next__` 返回容器中的下一个值
  - 如果容器中没有更多元素, 则抛出`StopIteration`异常
- generator: 生成器其实是一种特殊的迭代器, 不需要自定义 `__iter__` 和 `__next__`
  - 生成器函数 (yield)
  - 生成器表达式
- 练习1: 自定义一个迭代器, 实现斐波那契数列

```

class Fib(object):
    def __init__(self, max):
        self.x = 0
        self.y = 1
        self.max = max

    def __iter__(self):
        return self

    def __next__(self):
        n_next = self.y
        self.x, self.y = self.y, self.x + self.y
        if self.max > self.x:
            return n_next
        else:
            raise StopIteration()

```

- 练习2: 自定义一个生成器函数, 实现斐波那契数列

```

def fib(max):
    x = 0
    y = 1
    while y < max:
        yield y
        x, y = y, x + y

```

- 练习3: 定义一个随机数迭代器, 随机范围为 [1, 50], 最大迭代次数 30
- 迭代器、生成器有什么好处?
  - 节省内存
  - 惰性求值

### 4. method, classmethod 和 staticmethod

- method: 通过实例调用时, 可以引用类内部的任何属性和方法
- classmethod: 无需实例化, 可以调用类属性和类方法, 无法取到普通的成员属性和方法
- staticmethod: 无论用类调用还是用实例调用, 都无法取到类内部的属性和方法, 完全独立的一个方法
- 练习: 说出下面代码的运行结果

```

class Test(object): x = 123

```

```

def __init__(self):
    self.y = 456

def bar1(self):
    print('Hello world')

@classmethod
def bar2(cls):
    print('Bad world')

@staticmethod
def bar3():
    print('=====' )

def foo1(self):
    print(self.x)
    print(self.y)
    self.bar1()
    self.bar2()
    self.bar3()

@classmethod
def foo2(cls):
    print(cls.x)
    print(cls.y)
    cls.bar1()
    cls.bar2()
    cls.bar3()

@staticmethod
def foo3(obj):
    print(obj.x)
    print(obj.y)
    obj.bar1()
    obj.bar2()
    obj.bar3()

t = Test()
t.foo1()
t.foo2()
t.foo3()

```

## 5. Python 魔术方法

i. `__str__` , `__repr__`

ii. `__init__` 和 `__new__`

- `__new__` 返回一个对象的实例, `__init__` 无返回值
- `__new__` 是一个类方法
  - a. 单例模式

```

class A(object):
    '''单例模式'''
    obj = None
    def __new__(cls, *args, **kwargs):
        if cls.obj is None:
            cls.obj = object.__new__(cls)
            return cls.obj
        else:
            return cls.obj

```

iii. 比较运算、数学运算

- 比较运算符的重载: `__eq__`, `__ge__`, `__le__`, `__gt__`, `__lt__`, `__ne__`
- 练习: 完成一个类, 实现负无穷的概念

```

class A(object):
    def __le__(self, other):
        return True
    def __lt__(self, other):
        return True

```



```

def __eq__(self, other):
    return False
def __ge__(self, other):
    return False
def __gt__(self, other):
    return False

```

- 运算符重载: `__add__`, `__sub__`, `__mul__`, `__mod__`
- 练习: 实现字典的 `__add__` 方法, 作用相当于 `d.update(other)`

```

class D(dict):
    def __add__(self, other):
        if isinstance(other, dict):
            new = deepcopy(self)
            new.update(other)
            return new
        else:
            raise TypeError('not a dict')

```

#### iv. 容器方法

- `__len__`, `__iter__`, `__next__`, `__contains__`
- `__getitem__` 对 list, tuple, dict 有效
- `__setitem__` 对 list, dict 有效
- `__missing__` 对 dict 有效

#### v. 可执行对象: `__call__`

#### vi. with:

- `__enter__` 进入 with 代码块前的准备操作
- `__exit__` 退出时的善后操作

#### vii. `__setattr__`, `__getattr__`, `__getattr__`, `__dict__`

#### viii. 描述器: `__set__`, `__get__`

- <http://pyzh.readthedocs.io/en/latest/Descriptor-HOW-TO-Guide.html>

#### ix. `__slots__` 和内存优化

### 6. Python 性能之困

#### i. 计算密集型

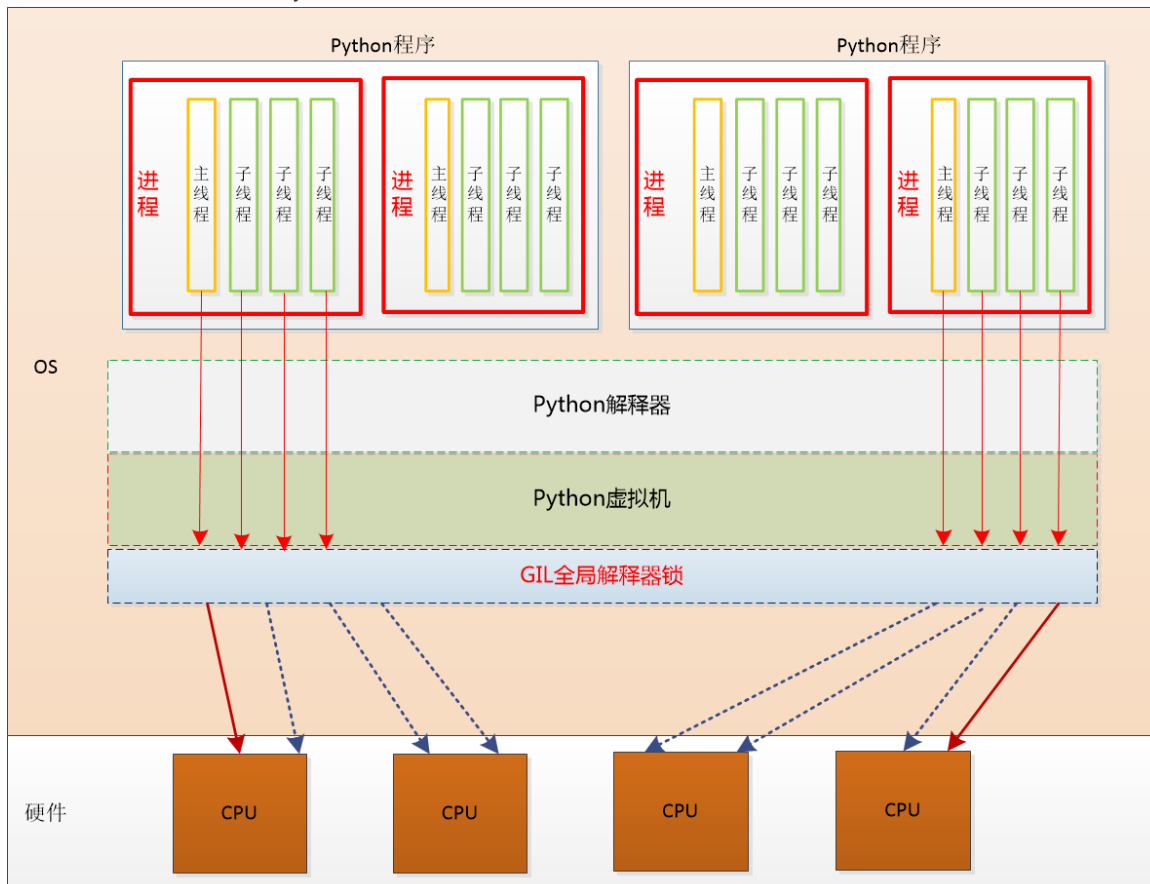
- CPU 长时间满负荷运行, 如图像处理、圆周率计算等
- 计算密集型: 用 C 语言补充
- Profile, timeit

#### ii. I/O 密集型: 网络 IO, 文件 IO, 设备 IO 等

- 多线程 / 多进程 / 协程
- 阻塞 -> 非阻塞
- 同步 -> 异步

#### iii. GIL 全局解释器锁

- 它确保任何时候都只有一个Python线程执行。



#### iv. 什么是进程、线程、协程？

- 进程: 资源消耗大, 系统整体开销大, 数据通信不方便
- 线程: 资源消耗小, 可共享数据。上下文开销大。按时间片强制切换, 不够灵活
- 协程: 内存开销更小, 上下文切换开销更小。可根据事件切换, 更加有效的利用 CPU

#### v. 什么是同步、异步、阻塞、非阻塞？

- 同步, 异步: 客户端调用服务器接口时
- 阻塞, 非阻塞: 服务端发生等待

#### vi. 事件驱动 + 多路复用

- 轮询: select, poll
- 事件驱动: epoll 有效轮询

#### vii. Greenlets / gevent | tornado / asyncio

#### viii. 线程安全, 锁

- 获得锁之后, 一定要释放, 避免死锁
- 获得锁之后, 执行的语句, 只跟被锁资源有关
- 区分普通锁 Lock, 可重入锁 RLock
- 线程之间的数据交互尽量使用 Queue

#### ix. gevent

- monkey.patch
- gevent.sleep 非阻塞式等待
- Queue 协程间数据交互, 避免竞争

### 7. 装饰器

#### ◦ 检查、预处理

- 练习: 写一个 timer 装饰器, 计算出被装饰函数调用一次花多长时间, 并把时间打印出来

#### ◦ 缓存

#### ◦ 其他

#### ◦ 闭包

```

local namespace
|
v
global namespace
|

```

## 8. 一些技巧和误区

### i. 格式化打印

- `json.dumps(obj, indent=4)`
- json 压缩: `json.dumps(obj, separators=(',', ':'))`
- `pprint`

### ii. 确保能取到有效值

- `d.get(k, default)`
- `d.setdefault`
- `defaultdict`
- `a or b`

### iii. try...except... 的滥用

- 不要把所有东西全都包住, 程序错误需要报出来
- 使用 `try...except` 要指明具体错误, `try` 结构不是用来隐藏错误的, 而是用来有方向的处理错误的

### iv. 利用 dict 做模式匹配

```
def do1():
    print('i am do1')

def do2():
    print('i am do2')

def do3():
    print('i am do3')

def do4():
    print('i am do4')

mod = random.randint(1, 10)
mapping = {1: do1, 2: do2, 3: do3, 4: do4}
func = mapping.get(mod, do4)
func()
```

## V. inf, -inf, nan

### vi. \*args, \*\*kwargs, foo(x, y, \*, z, \*\*kwargs), 强制显式参数 \*

### vii. pyenv, venv, 命名空间

- `pyenv`: 管理 Python 版本
- `venv`: 创建虚拟环境, 做环境隔离, `venv` 目录直接放到项目的目录里

### viii. property: 把一个方法属性化

```
class C(object):
    @property
    def x(self):
        "I am the 'x' property."
        return self._x
    @x.setter
    def x(self, value):
        self._x = value
    @x.deleter
    def x(self):
        del self._x
```

### ix. 要批量修改、存取, 避免循环内单步获取: `for range(): db.get()` VS `db.mget()`

### x. 三元运算: `x = a if foo() else b`

### xi. else 字句: if, for, while, try

### xii. collections

- `defaultdict`
- `namedtuple`
- `Counter`

## 9. 项目中的代码管理

### i. venv

### ii. 代码结构管理, 目录树

### iii. git

- clone
- status
- add
- commit
- push
- pull
- merg
- checkout
- branch

## 10. 补充

- 分享一些图书
- Django -> json

## 11. Python 环境加载

- sys.argv
- sys.path
- os.environ

# python 经典面试题



## 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：lambda 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 float 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: Python 如何定义一个函数

函数的定义形式如

下：

```
def <name>(arg1, arg2, ... argN):
```

```
<statements>
```

函数的名字也必须以字母开头，可以包括下划线“\_”，但不能把 Python 的关键字定义成函数的名字。函数内的语句数量是任意的，每个语句至少有一个空格的缩进，以表示此语句属于这个函数的。缩进结束的地方，函数自然结束。

下面定义了一个两个数相加的函数：



```
>>> def add(p1, p2):
print p1, "+", p2, "=", p1+p2
>>> add(1, 2)
1 + 2 = 3
```

函数的目的是把一些复杂的操作隐藏，来简化程序的结构，使其容易阅读。函数在调用前，必须先定义。也可以在一个函数内部定义函数，内部函数只有在外函数调用时才能够被执行。程序调用函数时，转到函数内部执行函数内部的语句，函数执行完毕后，返回到它离开程序的地方，执行程序的下一条语句。

#### 5: [Python 是如何进行内存管理的？](#)

Python 的内存管理是由 Python 得解释器负责的，开发人员可以从内存管理事务中解放出来，致力于应用程序的开发，这样就使得开发的程序错误更少，程序更健壮，开发周期更短

#### 6: 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>
```

#### 7: Python 里面如何实现 tuple 和 list 的转换？

函数 tuple(seq) 可以把所有可迭代的(iterable)序列转换成一个 tuple，元素不变，排序也不变。

例如，tuple([1,2,3]) 返回 (1,2,3)，tuple(' abc' ) 返回 (' a' . ' b', ' c')。如果参数已经是一个 tuple 的话，函数不做任何拷贝而直接返回原来的对象，所以在不确定对象是不是 tuple 的时候来调用 tuple() 函数也不是很耗费的。

函数 list(seq) 可以把所有的序列和可迭代的对象转换成一个 list，元素不变，排序也不变。

例如 list([1,2,3]) 返回 (1,2,3)，list(' abc' ) 返回 [' a', ' b', ' c']。如果参数是一个 list，她会像 set[:] 一样做一个拷贝

8: Python 面试题: 请写出一段 Python 代码实现删除一个 list 里面的重复元素

可以先把 list 重新排序, 然后从 list 的最后开始扫描, 代码如下:

```
if List:
    List.sort()
    last = List[-1]
    for i in range(len(List)-2, -1, -1):
        if last==List[i]: del List[i]
        else: last=List[i]
```

9: Python 文件操作的面试题

1. 如何用 Python 删除一个文件?

使用 `os.remove(filename)` 或者 `os.unlink(filename)`;

2. Python 如何 copy 一个文件?

`shutil` 模块有一个 `copyfile` 函数可以实现文件拷贝

10: Python 里面如何生成随机数?

标准库 `random` 实现了一个随机数生成器, 实例代码如下:

```
import random
```

```
random.random()
```

它会返回一个随机的 0 和 1 之间的浮点数

11: 如何用 Python 来发送邮件?

可以使用 `smtplib` 标准库。

以下代码可以在支持 SMTP 监听器的服务器上执行。

```
import sys, smtplib

fromaddr = raw_input(" From: ")
toaddrs = raw_input(" To: ").split(' ,')
print "Enter message, end with ^D:"
msg = ""
while 1:
    line = sys.stdin.readline()
    if not line:
        break
    msg = msg + line
```

```
# 发送邮件部分
server = smtplib.SMTP('localhost')
server.sendmail(fromaddr, toaddrs, msg)
server.quit()
```

12: Python 里面如何拷贝一个对象?

一般来说可以使用 `copy.copy()` 方法或者 `copy.deepcopy()` 方法, 几乎所有的对象都可以被拷贝

一些对象可以更容易的拷贝, Dictionaries 有一个 `copy` 方法:

```
newdict = olddict.copy()
```

13: 有没有一个工具可以帮助查找 python 的 bug 和进行静态的代码分析?

有, PyChecker 是一个 python 代码的静态分析工具, 它可以帮助查找 python 代码的 bug, 会对代码的复杂度和格式提出警告

PyLint 是另外一个工具可以进行 coding standard 检查。

14: 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
```

14: 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 加到 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
```

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

15: 用 Python 匹配 HTML tag 的时候, <.\*>和<.\*?>有什么区别?

当重复匹配一个正则表达式时候, 例如<.\*>, 当程序执行匹配的时候, 会返回最大的匹配值

例如：

```
import re
s = '<html><head><title>Title</title>'
print(re.match('<.*>', s).group())
```

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

而

```
import re
s = '<html><head><title>Title</title>'
print(re.match('<.*?>', s).group())
```

则会返回<html>

<.\*>这种匹配称作贪心匹配 <.\*?>称作非贪心匹配

16: Python 里面 search()和 match()的区别？

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

例如：

```
print(re.match('super', 'superstition').span()) 会返回 (0, 5)
而 print(re.match('super', 'insuperable')) 则返回 None
```

search() 会扫描整个字符串并返回第一个成功的匹配

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

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

17: 如何用 Python 来进行查询和替换一个文本字符串？

可以使用 sub() 方法来进行查询和替换，sub 方法的格式为： sub(replacement, string[, count=0])

replacement 是被替换成的文本

string 是需要被替换的文本

count 是一个可选参数，指最大被替换的数量

例子:

```
import re
p = re.compile(' (blue|white|red)' )
print(p.sub(' colour' , 'blue socks and red shoes' ))
print(p.sub(' colour' , 'blue socks and red shoes' , count=1))
```

输出:

```
colour socks and colour shoes
colour socks and red shoes
```

subn() 方法执行的效果跟 sub() 一样, 不过它会返回一个二维数组, 包括替换后的新的字符串和总共替换的数量

例如:

```
import re
p = re.compile(' (blue|white|red)' )
print(p.subn(' colour' , 'blue socks and red shoes' ))
print(p.subn(' colour' , 'blue socks and red shoes' , count=1))
```

输出

```
(' colour socks and colour shoes' , 2)
(' colour socks and red shoes' , 1)
```

18: 介绍一下 except 的用法和作用?

Python 的 except 用来捕获所有异常, 因为 Python 里面的每次错误都会抛出一个异常, 所以每个程序的错误都被当作一个运行时错误。

一下是使用 except 的一个例子:

```
try:

foo = opne(" file" ) #open 被错写为 opne

except:

sys.exit(" could not open file!" )
```

因为这个错误是由于 open 被拼写成 opne 而造成的, 然后被 except 捕获, 所以 debug 程序的时候很容易不知道出了什么问题

下面这个例子更好点:

```
try:
foo = open(" file" ) # 这时候 except 只捕获 IOError
except IOError:
sys.exit(" could not open file" )
```

19: Python 中 pass 语句的作用是什么？

pass 语句什么也不做，一般作为占位符或者创建占位程序，pass 语句不会执行任何操作，比如：

```
while False:
```

```
pass
```

pass 通常用来创建一个最简单的类：

```
class MyEmptyClass:
```

```
pass
```

pass 在软件设计阶段也经常用来作为 TODO，提醒实现相应的实现，比如：

```
def initlog(*args):
```

```
pass #please implement this
```

20: 介绍一下 Python 下 range()函数的用法？

如果需要迭代一个数字序列的话，可以使用 range() 函数，range() 函数可以生成等差级数。

如例：

```
for i in range(5)
```

```
print(i)
```

这段代码将输出 0, 1, 2, 3, 4 五个数字

range(10) 会产生 10 个值，也可以让 range() 从另外一个数字开始，或者定义一个不同的增量，甚至是负数增量

range(5, 10) 从 5 到 9 的五个数字

range(0, 10, 3) 增量为三，包括 0, 3, 6, 9 四个数字

range(-10, -100, -30) 增量为-30，包括-10, -40, -70

可以一起使用 range() 和 len() 来迭代一个索引序列



例如：

```
a = ['Nina', 'Jim', 'Rainman', 'Hello']  
for i in range(len(a)):  
    print(i, a[i])
```

# python 面试 100 例

# Python 语言特性

## 1 Python 的函数参数传递

看两个如下例子，分析运行结果：

代码一：

```
a = 1
def fun(a):
    a = 2
fun(a)
print(a) # 1
```

代码二：

```
a = []
def fun(a):
    a.append(1)
fun(a)
print(a) # [1]
```

所有的变量都可以理解是内存中一个对象的“引用”，或者，也可以看似 c 中 void\* 的感觉。

这里记住的是类型是属于对象的，而不是变量。而对象有两种，“可更改”（mutable）与“不可更改”（immutable）对象。在 python 中，strings, tuples, 和 numbers 是不可更改的对象，而 list,dict 等则是可以修改的对象。（这就是这个问题的重点）

当一个引用传递给函数的时候，函数自动复制一份引用，这个函数里的引用和外边的引用没有半毛关系了。所以第一个例子里函数把引用指向了一个不可变对象，当函数

返回的时候 ,外面的引用没半毛感觉 .而第二个例子就不一样了 ,函数内的引用指向的是可变对象 ,对它的操作就和定位了指针地址一样 ,在内存里进行修改 .

## 2 Python 中的元类 (metaclass)

元类就是用来创建类的“东西”。你创建类就是为了创建类的实例对象，但是我们已经学习到了 Python 中的类也是对象。好吧，元类就是用来创建这些类（对象）的，元类就是类的类

这个非常的不常用 ,详情请看：《 深刻理解 Python 中的元类 (metaclass) 》

## 3 @staticmethod 和 @classmethod

Python 其实有 3 个方法 ,即静态方法 (staticmethod), 类方法 (classmethod) 和实例方法 ,如下：

```
class A(object):
    def foo(self,x):
        print "executing foo(%s,%s)"%(self,x)

    @classmethod
    def class_foo(cls,x):
        print("executing class_foo(%s,%s)"%(cls,x))

    @staticmethod
    def static_foo(x):
        print ("executing static_foo(%s)"%x)

a=A()
```

这里先理解下函数参数里面的 self 和 cls. 这个 self 和 cls 是对类或者实例的绑定 .对于实例方法 ,我们知道在类里每次定义方法的时候都需要绑定这个实例 ,就是 foo(self, x) ,为什么要这么做呢 ?因为实例方法的调用离不开实例 ,我们需要把实

例自己传给函数 ,调用的时候是这样的 `a.foo(x)` (其实是 `foo(a, x)` ).类方法一样 ,只不过它传递的是类而不是实例 ,`A.class_foo(x)` .注意这里的 `self` 和 `cls` 可以替换别的参数 ,但是 python 的约定是这俩 ,还是不要改的好 .

对于静态方法其实和普通的方法一样 ,不需要对谁进行绑定 ,唯一的区别是调用的时候需要使用 `a.static_foo(x)` 或者 `A.static_foo(x)` 来调用 .

\	实例方法	类方法	静态方法
<code>a = A()</code>	<code>a.foo(x)</code>	<code>a.class_foo(x)</code>	<code>a.static_foo(x)</code>
<code>A</code>	不可用	<code>A.class_foo(x)</code>	<code>A.static_foo(x)</code>

## 4 类变量和实例变量

```
class Person:
    name="aaa"

p1=Person()
p2=Person()
p1.name="bbb"
print(p1.name) # bbb
print(p2.name) # aaa
print(Person.name) # aaa
```

类变量就是供类使用的变量 ,实例变量就是供实例使用的 .

这里 `p1.name="bbb"` 是实例调用了类变量 ,这其实和上面第一个问题一样 ,就是函数传参的问题 ,`p1.name` 一开始是指向的类变量 `name="aaa"` ,但是在实例的作用域里把类变量的引用改变了 ,就变成了一个实例变量 ,`self.name` 不再引用 `Person` 的类变量 `name` 了.

可以看看下面的例子 :

```
class Person:
    name=[]

p1=Person()
p2=Person()
p1.name.append(1)
print(p1.name) # [1]
print(p2.name) # [1]
print(Person.name) # [1]
```

## 5 Python 自省

这个也是 python 彪悍的特性 .

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 .简单一句就是运行时能够获得对象的类型 .比如 type(),dir(),getattr(),hasattr(),isinstance().

## 6 字典推导式

可能你见过列表推导时 ,却没有见过字典推导式 ,在 2.7 中才加入的 :

```
d = {key: value for (key, value) in iterable}
```

## 7 Python 中单下划线和双下划线

```
1 >>> class MyClass():
2 ...     def __init__(self):
3 ...         self.__superprivate = "Hello"
4 ...         self._semiprivate = ", world!"
5 ...
6 >>> mc = MyClass()
7 >>> print(mc.__superprivate)
8 Traceback (most recent call last):
9   File "<stdin>", line 1, in <module>
10 AttributeError: myClass instance has no attribute '__superprivate'
11 >>> print(mc._semiprivate)
```

```
12 , world!
13 >>> print mc.__dict__
14 {'_MyClass__superprivate': 'Hello', '_semiprivate': ', world!'}
```

`__foo__` :一种约定 ,Python 内部的名字 ,用来区别其他用户自定义的命名 ,以防冲突 .

`_foo` :一种约定 ,用来指定变量私有 .程序员用来指定私有变量的一种方式 .

`__foo` :这个有真正的意义 :解析器用 `_classname__foo` 来代替这个名字 ,以区别和其他类相同的命名 .

详情见 :

<http://www.zhihu.com/question/19754941>

## 8 字符串格式化 :% 和 .format

`.format` 在许多方面看起来更便利 .对于 % 最烦人的是它无法同时传递一个变量和元组 .你可能会想下面的代码不会有什么问题 :

Python:

```
"hi there %s" % name
```

但是 ,如果 `name` 恰好是 (1,2,3), 它将会抛出一个 `TypeError` 异常 .为了保证它总是正确的 ,你必须这样做 :

```
"hi there %s" % (name,)          # 提供一个单元素的数组而不是一个参数
```

## 9 迭代器和生成器

在 Python 中 ,这种一边循环一边计算的机制 ,称为生成器 : `generator` 。

可以被 `next()` 函数调用并不断返回下一个值的对象称为迭代器 : `Iterator` 。

这个是 stackoverflow 里 python 排名第一的问题 ,值得一看 :

<http://stackoverflow.com/questions/231767/what-does-the-yield-keyword>

-do-in-python

## 10 \*args and \*\*kwargs

用 \*args 和 \*\*kwargs 只是为了方便并没有强制使用它们 .

当你不确定你的函数里将要传递多少参数时你可以用 \*args .例如 ,它可以传递任意数量的参数 :

```
1 >>> def print_everything(*args):
2     for count, thing in enumerate(args):
3         print '{0}. {1}'.format(count, thing)
4     ...
5 >>> print_everything('apple', 'banana', 'cabbage')
6 0. apple
7 1. banana
8 2. cabbage
```

相似的 ,\*\*kwargs 允许你使用没有事先定义的参数名 :

```
1 >>> def table_things(**kwargs):
2     for name, value in kwargs.items():
3         print '{0} = {1}'.format(name, value)
4     ...
5 >>> table_things(apple = 'fruit', cabbage = 'vegetable')
6 cabbage = vegetable
7 apple = fruit
```

你也可以混着用 .命名参数首先获得参数值然后所有的其他参数都传递给 \*args 和

\*\*kwargs .命名参数在列表的最前端 .例如 :

```
1 def table_things(titlestring, **kwargs)
```

\*args 和 \*\*kwargs 可以同时函数的定义中 ,但是 \*args 必须在 \*\*kwargs 前面 .

当调用函数时你也可以用 \* 和 \*\* 语法 .例如 :



```
1 >>> def print_three_things(a, b, c):
2 ...     print 'a = {0}, b = {1}, c = {2}'.format(a,b,c)
3 ...
4 >>> mylist = ['aardvark', 'baboon', 'cat']
5 >>> print_three_things(*mylist)
6
7 a = aardvark, b = baboon, c = cat
```

就像你看到的一样 ,它可以传递列表 (或者元组 )的每一项并把它们解包 .注意必须与它们在函数里的参数相吻合 .当然,你也可以在函数定义或者函数调用时用 \*.

<http://stackoverflow.com/questions/3394835/args-and-kwargs>

## 11 面向切面编程 AOP 和装饰器

这个 AOP 一听起来有点懵 ,同学面试的时候就被问懵了...

装饰器是一个很著名的设计模式 , 经常被用于有切面需求的场景 , 较为经典的有插入日志、性能测试、事务处理等。装饰器是解决这类问题的绝佳设计 , 有了装饰器 , 我们就可以抽离出大量函数中与函数功能本身无关的雷同代码并继续重用。概括的讲 , 装饰器的作用就是为已经存在的对象添加额外的功能。

这个问题比较大 , 推荐 :

<http://stackoverflow.com/questions/739654/how-can-i-make-a-chain-of-function-decorators-in-python>

中文 :

<http://taizilongxu.gitbooks.io/stackoverflow-about-python/content/3/README.html>

## 12 鸭子类型

“ 当看到一只鸟走起来像鸭子、游泳起来像鸭子、叫起来也像鸭子，那么这只鸟就可以被称为鸭子。 ”

我们并不关心对象是什么类型，到底是不是鸭子，只关心行为。

比如在 python 中，有很多 file-like 的东西，比如 StringIO,GzipFile,socket 。它们有很多相同的方法，我们把它们当作文件使用。

又比如 list.extend() 方法中 ,我们并不关心它的参数是不是 list, 只要它是可迭代的 ,所以它的参数可以是 list/tuple/dict/ 字符串 / 生成器等 。

鸭子类型在动态语言中经常使用，非常灵活，使得 python 不想 java 那样专门去弄一大堆的设计模式。

## 13 Python 中重载

引自知乎 :<http://www.zhihu.com/question/20053359>

函数重载主要是为了解决两个问题。

1. 可变参数类型。
2. 可变参数个数。

另外，一个基本的设计原则是， 仅仅当两个函数除了参数类型和参数个数不同以外，其功能是完全相同的，此时才使用函数重载，如果两个函数的功能其实不同，那么不应当使用重载，而应当使用一个名字不同的函数。

好吧，那么对于情况 1，函数功能相同，但是参数类型不同， python 如何处理？答案是根本不需要处理， 因为 python 可以接受任何类型的参数， 如果函数的功能相同，那么不同的参数类型在 python 中很可能是相同的代码， 没有必要做成两个不同函数。

那么对于情况 2，函数功能相同，但参数个数不同， python 如何处理？大家知道，答案就是缺省参数。对那些缺少的参数设定为缺省参数即可解决问题。因为你假设函数功能相同，那么那些缺少的参数终归是需要用的。

好了，鉴于情况 1 跟 情况 2 都有了解决方案， python 自然就不需要函数重载了。

## 14 新式类和旧式类

这个面试官问了，我说了老半天，不知道他问的真正意图是什么。

这篇文章很好的介绍了新式类的特性：

<http://www.cnblogs.com/btchenguang/archive/2012/09/17/2689146.html>

新式类很早在 2.2 就出现了，所以旧式类完全是兼容的问题，Python3 里的类全部都是新式类。这里有一个 MRO 问题可以了解下（新式类是广度优先，旧式类是深度优先），<Python 核心编程>里讲的也很多。

## 15 `__new__` 和 `__init__` 的区别

这个 `__new__` 确实很少见到 ,先做了解吧 .

1. `__new__` 是一个静态方法 ,而 `__init__` 是一个实例方法 .
2. `__new__` 方法会返回一个创建的实例 ,而 `__init__` 什么都不返回 .
3. 只有在 `__new__` 返回一个 `cls` 的实例时后面的 `__init__` 才能被调用 .
4. 当创建一个新实例时调用 `__new__` ,初始化一个实例时用 `__init__` .

ps: `__metaclass__` 是创建类时起作用 .所以我们可以分别使用

`__metaclass__` , `__new__` 和 `__init__` 来分别在类创建 ,实例创建和实例初始化的时候做一些小手脚 .

## 16 单例模式

这个绝对常考啊 .绝对要记住 1~2 个方法 ,当时面试官是让手写的 .

### 1 使用 `__new__` 方法

```
class Singleton(object):
    def __new__(cls, *args, **kw):
        if not hasattr(cls, '_instance'):
            orig = super(Singleton, cls)
            cls._instance = orig.__new__(cls, *args, **kw)
        return cls._instance
```

```
class MyClass(Singleton):
    a = 1
```

### 2 共享属性

创建实例时把所有实例的 `__dict__` 指向同一个字典 ,这样它们具有相同的属性和方法 .

```
1 class Borg(object):
2     _state = {}
3     def __new__(cls, *args, **kw):
```

```

4     ob=super(Borg,cls).__new__(cls,*args,**kw)
5     ob.__dict__=cls._state
6     returnob
7

```

```

8 classMyClass2(Borg):
9     a=1

```

### 3 装饰器版本

```

1
2 defsingleton(cls,*args,**kw):
3     instances={}
4     defgetinstance():
5         ifcls notininstances:
6             instances[cls]=cls(*args,**kw)
7         returninstances[cls]
8     returngetinstance
9
10 @singleton
11 classMyClass:

```

## 4 import 方法

作为 python 的模块是天然的单例模式

```

# mysingleton.py
classMy_Singleton(object):
    deffoo(self):
        pass

my_singleton=My_Singleton()

# to use
frommysingleton importmy_singleton

my_singleton.foo()

```

## 17 Python 中的作用域

Python 中，一个变量的作用域总是由在代码中被赋值的地方所决定的。

当 Python 遇到一个变量的话他会按照这样的顺序进行搜索：

本地作用域（ Local ） 当前作用域被嵌入的本地作用域（ Enclosing locals ） 全局/模块作用域（ Global ） 内置作用域（ Built-in ）

## 18 GIL 线程全局锁

线程全局锁 (Global Interpreter Lock), 即 Python 为了保证线程安全而采取的独立线程运行的限制,说白了就是一个核只能在同一时间运行一个线程。

解决办法就是多进程和下面的协程 (协程也只是单 CPU,但是能减小切换代价提升性能)。

## 19 协程

简单点说协程是进程和线程的升级版,进程和线程都面临着内核态和用户态的切换问题而耗费许多切换时间,而协程就是用户自己控制切换的时机,不再需要陷入系统的内核态。

Python 里最常见的 yield 就是协程的思想!可以查看第九个问题。

## 20 闭包

闭包 (closure) 是函数式编程的重要的语法结构。闭包也是一种组织代码的结构,它同样提高了代码的可重复使用性。

当一个内嵌函数引用其外部作用域的变量 ,我们就会得到一个闭包 . 总结一下 ,创建一个闭包必须满足以下几点 :

1. 必须有一个内嵌函数
2. 内嵌函数必须引用外部函数中的变量
3. 外部函数的返回值必须是内嵌函数

感觉闭包还是有难度的 ,几句话是说不明白的 ,还是查查相关资料 .

重点是函数运行后并不会被撤销 ,就像 16 题的 instance 字典一样 ,当函数运行完后 ,instance 并不被销毁 ,而是继续留在内存空间里 .这个功能类似类里的类变量 ,只不过迁移到了函数上 .

闭包就像个空心球一样 ,你知道外面和里面 ,但你不知道中间是什么样 .

## 21 lambda 函数

其实就是一个匿名函数 ,为什么叫 lambda? 因为和后面的函数式编程有关 .

## 22 Python 函数式编程

这个需要适当的了解一下吧 ,毕竟函数式编程在 Python 中也做了引用 .

python 中函数式编程支持 :

filter 函数的功能相当于过滤器。调用一个布尔函数 bool\_func 来迭代遍历每个 seq 中的元素 ; 返回一个使 bool\_seq 返回值为 true 的元素的序列。

```
>>>a=[1,2,3,4,5,6,7]
>>>b=filter(lambdax:x>5,a)
>>>printb
>>>[6,7]
```

map 函数是对一个序列的每个项依次执行函数， 下面是对一个序列每个项都乘以 2：

```
>>>a=map(lambdax:x*2,[1,2,3])
>>>list(a)
[2,4,6]
```

reduce 函数是对一个序列的每个项迭代调用函数，下面是求 3 的阶乘：

```
>>>reduce(lambdax,y:x*y,range(1,4))
6
```

## 23 Python 里的拷贝

引用和 copy(),deepcopy() 的区别

```
1 importcopy
2 a=[1,2,3,4,['a','b']] # 原始对象
3
4 b=a #赋值，传对象的引用
5 c=copy.copy(a) # 对象拷贝，浅拷贝
6 d=copy.deepcopy(a) # 对象拷贝，深拷贝
7
8 a.append(5) #修改对象 a
9 a[4].append('c') # 修改对象 a 中的 ['a', 'b'] 数组对象
10
11 print'a = ',a
12 print'b = ',b
13 print'c = ',c
14 print'd = ',d
15
16 输出结果：
17 a= [1,2,3,4,['a','b','c'],5]
18 b= [1,2,3,4,['a','b','c'],5]
19 c= [1,2,3,4,['a','b','c']]
```



```
20 d= [1,2,3,4,['a','b']]
```

## 24 Python 垃圾回收机制

Python GC 主要使用引用计数（reference counting）来跟踪和回收垃圾。在引用计数的基础上，通过“标记 - 清除”（mark and sweep）解决容器对象可能产生的循环引用问题，通过“分代回收”（generation collection）以空间换时间的方法提高垃圾回收效率。

### 1 引用计数

PyObject 是每个对象必有的内容，其中 ob\_refcnt 就是做为引用计数。当一个对象有新的引用时，它的 ob\_refcnt 就会增加，当引用它的对象被删除，它的 ob\_refcnt 就会减少。引用计数为 0 时，该对象生命就结束了。

优点：

1. 简单
2. 实时性

缺点：

1. 维护引用计数消耗资源
2. 循环引用

## 2 标记 - 清除机制

基本思路是先按需分配，等到没有空闲内存的时候从寄存器和程序栈上的引用出发，遍历以对象为节点、以引用为边构成的图，把所有可以访问到的对象打上标记，然后清扫一遍内存空间，把所有没标记的对象释放。

## 3 分代技术

分代回收的整体思想是：将系统中的所有内存块根据其存活时间划分为不同的集合，每个集合就成为一个“代”，垃圾收集频率随着“代”的存活时间的增大而减小，存活时间通常利用经过几次垃圾回收来度量。

Python 默认定义了三代对象集合，索引数越大，对象存活时间越长。

举例：

当某些内存块 M 经过了 3 次垃圾收集的清洗之后还存活时，我们就将内存块 M 划到一个集合 A 中去，而新分配的内存都划分到集合 B 中去。当垃圾收集开始工作时，大多数情况都只对集合 B 进行垃圾回收，而对集合 A 进行垃圾回收要隔相当长一段时间后才进行，这就使得垃圾收集机制需要处理的内存少了，效率自然就提高了。

在这个过程中，集合 B 中的某些内存块由于存活时间长而会被转移到集合 A 中，当然，集合 A 中实际上也存在一些垃圾，这些垃圾的回收会因为这种分代的机制而被延迟。

25 Python 里面如何实现 tuple 和 list 的转换？

答：tuple，可以说是不可变的 list，访问方式还是通过索引下标的方式。

当你明确定义个 tuple 是，如果仅有一个元素，必须带有逗号，例如：(1,)。

当然，在 2.7 以后的版本，python 里还增加了命名式的 tuple ！

至于有什么用，首先第一点，楼主玩过 python 都知道，python 的函数可以有多返回值的，而 python

里，多返回值，就是用 tuple 来表示，这是用的最广的了，

比如说，你需要定义一个常量的列表，但你又不想使用 list，那也可以是要你管 tuple，例如：

```
if a in ('A','B','C'):pass
```

## 26 Python 的 is

is 是对比地址，== 是对比值

## 27 read,readline 和 readlines

- read 读取整个文件
- readline 读取下一行，使用生成器方法
- readlines 读取整个文件到一个迭代器以供我们遍历

## 28 Python2 和 3 的区别

大部分 Python 库都同时支持 Python 2.7.x 和 3.x 版本的，所以不论选择哪个版本

都是可以的。但为了在使用 Python 时避开某些版本中一些常见的陷阱，或需要移

植某个 Python 项目

使用 `__future__` 模块

`print` 函数

整数除法

Unicode

`xrange`

触发异常

处理异常

`next()` 函数和 `.next()` 方法

For 循环变量与全局命名空间泄漏

比较无序类型

使用 `input()` 解析输入内容

返回可迭代对象，而不是列表

推荐：《 [Python 2.7.x 和 3.x 版本的重要区别](#) 》

## 29 到底什么是 Python ？你可以在回答中与其他技术进行对比

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。

## 30 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

31 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。

## 32 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` 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。

## 33 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！

## 34 “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, 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 原则吗？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。

## 35 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。

36 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。

## 36. 介绍一下 `except` 的用法和作用？

答： `try ...except ...except ...[else ...][finally ...]`

执行 `try` 下的语句，如果引发异常，则执行过程会跳到 `except` 语句。对每个 `except` 分支顺序尝试执行，如果引发的异常与 `except` 中的异常组匹配，执行相应的语句。如果所有的 `except` 都不匹配，则异常会传递到下一个调用本代码的最高层 `try` 代码中。

`try` 下的语句正常执行，则执行 `else` 块代码。如果发生异常，就不会执行

如果存在 `finally` 语句，最后总是会执行。

## 37. Python 中 `pass` 语句的作用是什么？

答： `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，

`while False: pass`

## 38. 介绍一下 Python 下 `range()` 函数的用法？

答：列出一组数据，经常用在 `for` `in range()` 循环中

## 39. 如何用 Python 来进行查询和替换一个文本字符串？

答：可以使用 `re` 模块中的 `sub()` 函数或者 `subn()` 函数来进行查询和替换，

格式：`sub(replacement, string[, count=0])` （`replacement` 是被替换成的文本，

string 是需要被替换的文本， count 是一个可选参数，指最大被替换的数量)

```
>>> import re
```

```
>>> p=re.compile( ' blue|white|red ' )
```

```
>>> print(p.sub( ' colour ', 'blue socks and red shoes ' ))
```

```
colour socks and colourshoes
```

```
>>> print(p.sub( ' colour ', 'blue socks and red shoes ', count=1))
```

```
colour socks and redshoes
```

subn() 方法执行的效果跟 sub() 一样，不过它会返回一个二维数组，包括替换后的新的字符串和总共替换的数量

## 40. Python 里面 match() 和 search() 的区别？

答：re 模块中 match(pattern,string[,flags])， 检查 string 的开头是否与 pattern 匹配。

re 模块中 research(pattern,string[,flags])， 在 string 搜索 pattern 的第一个匹配值。

```
>>> print(re.match( ' super ', ' superstition ').span())
```

```
(0, 5)
```

```
>>> print(re.match( ' super ', ' insuperable ' ))
```

```
None
```

```
>>> print(re.search( ' super ', ' superstition ').span())
```

```
(0, 5)
```

```
>>>print(re.search( ' super ' , ' insuperable ' ).span())
```

(2, 7)

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

答 : 术语叫贪婪匹配 ( <.\*> ) 和非贪婪匹配 ( <.\*?> )

例如 :

test

<.\*> :

test

<.\*?> :

42. Python 里面如何生成随机数 ?

答 : random 模块

随机整数 : random.randint(a,b) : 返回随机整数 x, a<=x<=b

random.randrange(start,stop,[,step]) : 返回一个范围在 (start,stop,step) 之间的随机整数 , 不包括结束值。

随机实数 : random.random( ) : 返回 0 到 1 之间的浮点数

random.uniform(a,b): 返回指定范围内的浮点数。

43. 有没有一个工具可以帮助查找 python 的 bug 和进行静态的代码分析？

答：PyChecker 是一个 python 代码的静态分析工具，它可以帮助查找 python 代码的 bug，会对代码的复杂度和格式提出警告

Pylint 是另外一个工具可以进行 codingstandard 检查

44. 如何在一个 function 里面设置一个全局的变量？

答：解决方法是在 function 的开始插入一个 global 声明：

```
def f()

global x
```

45. 单引号，双引号，三引号的区别

答：单引号和双引号是等效的，如果要换行，需要符号 (\), 三引号则可以直接换行，并且可以包含注释

如果要表示 Let 's go 这个字符串

单引号：s4 = ' 's go '

双引号：s5 = " Let 's go "

s6 = ' I realy like " python " ! '

这就是单引号和双引号都可以表示字符串的原因了

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

47 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(l1n):
```

```
    l1 = sorted(l1n)
```

```
    l2 = [i for i in l1 if i<0.5]
```

```
    return [i*i for i in l2]
```

```
def f2(l1n):
```

```
    l1 = [i for i in l1n if i<0.5]
```

```
    l2 = sorted(l1)
```

```
    return [i*i for i in l2]
```

```
def f3(l1n):
```

```
    l1 = [i*i for i in l1n]
```

```
    l2 = sorted(l1)
```

```
    return [i for i in l2 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: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(lln)')

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。

## 48. 如何用 Python 来进行查询和替换一个文本字符串？

可以使用 `sub()` 方法来进行查询和替换， `sub` 方法的格式为：`sub(replacement, string[, count=0])`

`replacement` 是被替换成的文本

`string` 是需要被替换的文本

`count` 是一个可选参数，指最大被替换的数量

## 49. Python 里面 `search()` 和 `match()` 的区别？

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

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

前者是贪婪匹配，会从头到尾匹配 <a>xyz</a> ，而后者是非贪婪匹配，只匹配到第一个 >。

51. Python 里面如何生成随机数？

```
import random
```

```
random.random()
```

它会返回一个随机的 0 和 1 之间的浮点数

## 操作系统

### 1 select,poll 和 epoll

其实所有的 I/O 都是轮询的方法，只不过实现的层面不同罢了。

这个问题可能有点深入了，但相信能回答出这个问题是对 I/O 多路复用有很好的了解了。其中 tornado 使用的就是 epoll 的。

基本上 select 有 3 个缺点：

1. 连接数受限
2. 查找配对速度慢
3. 数据由内核拷贝到用户态

poll 改善了第一个缺点

epoll 改了三个缺点 .

## 2 调度算法

1. 先来先服务 (FCFS, First Come First Serve)
2. 短作业优先 (SJF, Shortest Job First)
3. 最高优先权调度 (Priority Scheduling)
4. 时间片轮转 (RR, Round Robin)
5. 多级反馈队列调度 (multilevel feedback queue scheduling)

实时调度算法 :

1. 最早截至时间优先 EDF
2. 最低松弛度优先 LLF

## 3 死锁

原因:

1. 竞争资源
2. 程序推进顺序不当

必要条件：

1. 互斥条件
2. 请求和保持条件
3. 不剥夺条件
4. 环路等待条件

处理死锁基本方法：

1. 预防死锁 (摒弃除 1 以外的条件 )
2. 避免死锁 (银行家算法 )
3. 检测死锁 (资源分配图 )
4. 解除死锁

1. 剥夺资源
2. 撤销进程

## 4 程序编译与链接

Bulid 过程可以分解为 4 个步骤 :预处理 (Prepressing), 编译 (Compilation) 、汇编 (Assembly) 、链接 (Linking)

以 c 语言为例：

### 1 预处理

预编译过程主要处理那些源文件中的以 “ # ” 开始的预编译指令，主要处理规则有：

1. 将所有的“`#define`”删除，并展开所用的宏定义
2. 处理所有条件预编译指令，比如“`#if`”、“`#ifdef`”、“`#elif`”、“`#endif`”
3. 处理“`#include`”预编译指令，将被包含的文件插入到该编译指令的位置，  
注：此过程是递归进行的
4. 删除所有注释
5. 添加行号和文件名标识，以便于编译时编译器产生调试用的行号信息以及用于编译时产生编译错误或警告时可显示行号
6. 保留所有的 `#pragma` 编译器指令。

## 2 编译

编译过程就是把预处理完的文件进行一系列的词法分析、语法分析、语义分析及优化后生成相应的汇编代码文件。这个过程是整个程序构建的核心部分。

## 3 汇编

汇编器是将汇编代码转化成机器可以执行的指令，每一条汇编语句几乎都是一条机器指令。经过编译、链接、汇编输出的文件成为目标文件 (Object File)

## 4 链接

链接的主要内容就是把各个模块之间相互引用的部分处理好，使各个模块可以正确的拼接。

链接的主要过程包括 地址和空间的分配 ( Address and Storage Allocation )、符号决议 (Symbol Resolution) 和重定位 (Relocation) 等步骤。



## 5 静态链接和动态链接

静态链接方法：静态链接的时候，载入代码就会把程序会用到的动态代码或动态代码的地址确定下来

静态库的链接可以使用静态链接，动态链接库也可以使用这种方法链接导入库

动态链接方法：使用这种方式的程序并不在一开始就完成动态链接，而是直到真正调用动态库代码时，载入程序才计算（被调用的那部分）动态代码的逻辑地址，然后等到某个时候，程序又需要调用另外某块动态代码时，载入程序又去计算这部分代码的逻辑地址，所以，这种方式使程序初始化时间较短，但运行期间的性能比不上静态链接的程序

## 6 虚拟内存技术

虚拟存储器是值具有请求调入功能和置换功能，能从逻辑上对内存容量加以扩充的一种存储系统。

## 7 分页和分段

分页：用户程序的地址空间被划分成若干固定大小的区域，称为“页”，相应地，内存空间分成若干个物理块，页和块的大小相等。可将用户程序的任一页放在内存的任一块中，实现了离散分配。

分段：将用户程序地址空间分成若干个大小不等的段，每段可以定义一组相对完整的逻辑信息。存储分配时，以段为单位，段与段在内存中可以不相邻接，也实现了离散分配。

## 分页与分段的主要区别

1. 页是信息的物理单位 ,分页是为了实现非连续分配 ,以便解决内存碎片问题 ,或者说分页是由于系统管理的需要 .段是信息的逻辑单位 ,它含有一组意义相对完整的信息 ,分段的目的是为了更好地实现共享 ,满足用户的需要 .
2. 页的大小固定 ,由系统确定 ,将逻辑地址划分为页号和页内地址是由机器硬件实现的 .而段的长度却不固定 ,决定于用户所编写的程序 ,通常由编译程序在对源程序进行编译时根据信息的性质来划分 .
3. 分页的作业地址空间是一维的 .分段的地址空间是二维的 .

## 8 页面置换算法

1. 最佳置换算法 OPT:不可能实现
2. 先进先出 FIFO
3. 最近最久未使用算法 LRU:最近一段时间里最久没有使用过的页面予以置换 .
4. clock 算法

## 9 边沿触发和水平触发

边缘触发是指每当状态变化时发生一个 io 事件 , 条件触发是只要满足条件就发生一个 io 事件

## 数据库

### 1 事务

数据库事务 (Database Transaction) ,是指作为单个逻辑工作单元执行的一系列操作，要么完全地执行，要么完全地不执行。

## 2 数据库索引

索引是对数据库表中一列或多列的值进行排序的一种结构，使用索引可快速访问数据库表中的特定信息。

索引分为聚簇索引和非聚簇索引两种，聚簇索引是按照数据存放的物理位置为顺序的，而非聚簇索引就不一样了；聚簇索引能提高多行检索的速度，而非聚簇索引对于单行的检索很快。

推荐：<http://tech.meituan.com/mysql-index.html>

## 3 Redis 原理

## 4 乐观锁和悲观锁

悲观锁：假定会发生并发冲突，屏蔽一切可能违反数据完整性的操作

乐观锁：假设不会发生并发冲突，只在提交操作时检查是否违反数据完整性。

## 5 MVCC

大多数的 MySQL 事务型存储引擎，如 InnoDB，Falcon 以及 PBXT 都不使用一种简单的行锁机制。事实上，他们都和另外一种用来增加并发性的被称为“多版本并发控制（MVCC）”的机制来一起使用。MVCC 不只使用在 MySQL 中，Oracle、PostgreSQL，以及其他一些

数据库系统也同样使用它。

## 6 MyISAM 和 InnoDB

MyISAM 适合于一些需要大量查询的应用，但其对于有大量写操作并不是很好。甚至你只是需要 `update` 一个字段，整个表都会被锁起来，而别的进程，就算是读进程都无法操作直到读操作完成。另外，MyISAM 对于 `SELECT COUNT(*)` 这类的计算是超快无比的。

InnoDB 的趋势会是一个非常复杂的存储引擎，对于一些小的应用，它会比 MyISAM 还慢。但是它支持“行锁”，于是在写操作比较多的时候，会更优秀。并且，他还支持更多的高级应用，比如：事务。

## 网络

### 1 三次握手

1. 客户端通过向服务器端发送一个 `SYN` 来创建一个主动打开，作为三路握手的一部分。客户端把这段连接的序号设定为随机数 `A`。
2. 服务器端应当为一个合法的 `SYN` 回送一个 `SYN/ACK`。ACK 的确认码应为 `A+1`，`SYN/ACK` 包本身又有一个随机序号 `B`。
3. 最后，客户端再发送一个 `ACK`。当服务端受到这个 `ACK` 的时候，就完成了三路握手，并进入了连接创建状态。此时包序号被设定为收到的确认号 `A+1`，而响应则为 `B+1`。

## 2 四次挥手

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

- （1）客户端 A 发送一个 FIN，用来关闭客户 A 到服务器 B 的数据传送。
- （2）服务器 B 收到这个 FIN，它发回一个 ACK，确认序号为收到的序号加 1。和 SYN 一样，一个 FIN 将占用一个序号。
- （3）服务器 B 关闭与客户端 A 的连接，发送一个 FIN 给客户端 A。
- （4）客户端 A 发回 ACK 报文确认，并将确认序号设置为收到序号加 1。

## 3 ARP 协议

地址解析协议 (Address Resolution Protocol): 根据 IP 地址获取物理地址的一个 TCP/IP 协议

## 4 urllib 和 urllib2 的区别

这个面试官确实问过，当时答的 urllib2 可以 Post 而 urllib 不可以。

1. urllib 提供 urlencode 方法用来 GET 查询字符串的产生，而 urllib2 没有。  
这是为何 urllib 常和 urllib2 一起使用的原因。
2. urllib2 可以接受一个 Request 类的实例来设置 URL 请求的 headers，urllib 仅可以接受 URL。这意味着，你不可以伪装你的 User Agent 字符串等。

## 5 Post 和 Get 区别

GET 后退按钮 / 刷新无害，POST 数据会被重新提交（浏览器应该告知用户数据会被重新提交）。

GET 书签可收藏，POST 为书签不可收藏。

GET 能被缓存，POST 不能缓存。

GET 编码类型 `application/x-www-form-urlencoded`，POST 编码类型 `application/x-www-form-urlencoded` 或 `multipart/form-data`。为二进制数据使用多重编码。

GET 历史参数保留在浏览器历史中。POST 参数不会保存在浏览器历史中。

GET 对数据长度有限制，当发送数据时，GET 方法向 URL 添加数据；URL 的长度是受限制的（URL 的最大长度是 2048 个字符）。POST 无限制。

GET 只允许 ASCII 字符。POST 没有限制。也允许二进制数据。

与 POST 相比，GET 的安全性较差，因为所发送的数据是 URL 的一部分。在发送密码或其他敏感信息时绝不要使用 GET！POST 比 GET 更安全，因为参数不会被保存在浏览器历史或 web 服务器日志中。

GET 的数据在 URL 中对所有人都是可见的。POST 的数据不会显示在 URL 中。

## 6 Cookie 和 Session

	Cookie	Session
储存位置	客户端	服务器端
目的	跟踪会话，也可以保存用户偏好设置或者保存用户名密码等	跟踪会话
安全性	不安全	安全

session 技术是要使用到 cookie 的，之所以出现 session 技术，主要是为了安全。

## 7 apache 和 nginx 的区别

nginx 相对 apache 的优点：

- 轻量级，同样起 web 服务，比 apache 占用更少的内存及资源
- 抗并发，nginx 处理请求是异步非阻塞的，支持更多的并发连接，而 apache 则是阻塞型的，在高并发下 nginx 能保持低资源低消耗高性能
- 配置简洁
- 高度模块化的设计，编写模块相对简单
- 社区活跃

apache 相对 nginx 的优点：

- rewrite，比 nginx 的 rewrite 强大

- 模块超多，基本想到的都可以找到
- 少 bug ，nginx 的 bug 相对较多
- 超稳定

## 8 网站用户密码保存

1. 明文保存
2. 明文 hash 后保存 ,如 md5
3. MD5+Salt 方式 ,这个 salt 可以随机
4. 知乎使用了 Bcrypt( 好像 )加密

## 9 HTTP 和 HTTPS

HTTPS( 全称 : Hypertext Transfer Protocol over Secure Socket Layer ) ,是以安全为目标的 HTTP 通道 ,简单讲是 HTTP 的安全版。即 HTTP 下加入 SSL 层 , HTTPS 的安全基础是 SSL ,因此加密的详细内容就需要 SSL。 它是一个 URI scheme ( 抽象标识符体系 ) ,句法类同 http: 体系。用于安全的 HTTP 数据传输。 https:URL 表明它使用了 HTTP ,但 HTTPS 存在不同于 HTTP 的默认端口及一个加密 / 身份验证层 ( 在 HTTP 与 TCP 之间 ) 。这个系统的最初研发由网景公司进行 , 提供了身份验证与加密通讯方法 , 现在它被广泛用于万维网上安全敏感的通讯 , 例如交易支付方面。

超文本传输协议 (HTTP-Hypertext transfer protocol) 是一种详细规定了浏览器和万维网服务器之间互相通信的规则 , 通过因特网传送万维网文档的数据传送协议。

## 10 XSRF 和 XSS

- CSRF(Cross-site request forgery) 跨站请求伪造



- XSS(Cross Site Scripting) 跨站脚本攻击

CSRF 重点在请求 ,XSS 重点在脚本

## 11 RESTful 架构(SOAP,RPC)

推荐 : <http://www.ruanyifeng.com/blog/2011/09/restful.html>

## 12 SOAP

SOAP ( 原为 Simple Object Access Protocol 的首字母缩写 , 即简单对象访问协议 ) 是交换数据的一种协议规范 , 使用在计算机网络 Web 服务 ( web service ) 中 , 交换带结构信息。 SOAP 为了简化网页服务器 ( Web Server ) 从 XML 数据库中提取数据时 , 节省去格式化页面时间 , 以及不同应用程序之间按照 HTTP 通信协议 , 遵从 XML 格式执行资料互换 , 使其抽象于语言实现、平台和硬件。

## 13 RPC

RPC ( Remote Procedure Call Protocol ) —— 远程过程调用协议 , 它是一种通过网络从远程计算机程序上请求服务 , 而不需要了解底层网络技术的协议。 RPC 协议假定某些传输协议的存在 , 如 TCP 或 UDP , 为通信程序之间携带信息数据。在 OSI 网络通信模型中 , RPC 跨越了传输层和应用层。 RPC 使得开发包括网络分布式多程序在内的应用程序更加容易。

总结 : 服务提供的两大流派 . 传统意义以方法调用为导向通称 RPC。为了企业 SOA, 若干厂商联合推出 webservice, 制定了 wsdl 接口定义 , 传输 soap. 当互联网时代 , 臃

肿 SOA 被简化为 http+xml/json. 但是简化出现各种混乱。以资源为导向 ,任何操作无非是对资源的增删改查 ,于是统一的 REST 出现了 .

进化的顺序 : RPC -> SOAP -> RESTful

## 14 CGI 和 WSGI

CGI 是通用网关接口 ,是连接 web 服务器和应用程序的接口 ,用户通过 CGI 来获取动态数据或文件等。

CGI 程序是一个独立的程序 , 它可以用几乎所有语言来写 , 包括 perl ,c ,lua ,python 等等。

WSGI, Web Server Gateway Interface , 是 Python 应用程序或框架和 Web 服务器之间的一种接口 , WSGI 的其中一个目的就是让用户可以用统一的语言 (Python) 编写前后端。

## 16 中间人攻击

在 GFW 里屡见不鲜的 ,呵呵 .

中间人攻击 ( Man-in-the-middle attack , 通常缩写为 MITM ) 是指攻击者与通讯的两端分别创建独立的联系 , 并交换其所收到的数据 , 使通讯的两端认为他们正在通过一个私密的连接与对方直接对话 , 但事实上整个会话都被攻击者完全控制。

## 17 c10k 问题

所谓 c10k 问题，指的是服务器同时支持成千上万个客户端的问题，也就是 concurrent 10 000 connection （这也是 c10k 这个名字的由来）。

## 18 socket

Socket=Ip address+ TCP/UDP + port

## 19 浏览器缓存

推荐 : <http://web.jobbole.com/84367/>

浏览器缓存机制，其实主要就是 HTTP 协议定义的缓存机制（如：Expires ； Cache-control 等）

Expires 策略

Expires 是 Web 服务器响应消息头字段，在响应 http 请求时告诉浏览器在过期时间前浏览器可以直接从浏览器缓存取数据，而无需再次请求。

Cache-control 策略（重点关注）

Cache-Control 与 Expires 的作用一致，都是指明当前资源的有效期，控制浏览器是否直接从浏览器缓存取数据还是重新发请求到服务器取数据。只不过 Cache-Control 的选择更多，设置更细致，如果同时设置的话，其优先级高于 Expires

## 20 HTTP1.0 和 HTTP1.1

推荐 : <http://blog.csdn.net/elifely/article/details/3964766>

1. 请求头 Host 字段,一个服务器多个网站

2. 长链接
3. 文件断点续传
4. 身份认证 ,状态管理 ,Cache 缓存

## 21 Ajax

AJAX,Asynchronous JavaScript and XML （异步的 JavaScript 和 XML ），是与在不重新加载整个页面的情况下，与服务器交换数据并更新部分网页的技术。

## 数据结构

### 1 红黑树

红黑树与 AVL 的比较：

AVL 是严格平衡树，因此在增加或者删除节点的时候，根据不同情况，旋转的次数比红黑树要多；

红黑是用非严格的平衡来换取增删节点时候旋转次数的降低；

所以简单说，如果你的应用中，搜索的次数远远大于插入和删除，那么选择 AVL ，如果搜索，插入删除次数几乎差不多，应该选择 RB。

### 1 台阶问题 / 斐波纳挈

一只青蛙一次可以跳上 1 级台阶，也可以跳上 2 级。求该青蛙跳上一个 n 级的台阶总共有多少种跳法。

```
1 fib=lambdan:nifn<=2elsefib(n-1)+fib(n-2)
```

## 第二种记忆方法

```
defmemo(func):
    cache={}
    defwrap(*args):
        ifargs notincache:
            cache[args]=func(*args)
        returncache[args]
    returnwrap
```

```
@memo
deffib(i):
    ifi<2:
        return1
    returnfib(i-1)+fib(i-2)
```

## 第三种方法

```
1 deffib(n):
2     a,b=0,1
3     for_inxrange(n):
4         a,b=b,a+b
5     returnb
```

## 2 变态台阶问题

一只青蛙一次可以跳上 1 级台阶，也可以跳上 2 级.....它也可以跳上 n 级。求该青蛙跳上一个 n 级的台阶总共有多少种跳法。

```
1 fib=lambdan:nifn<2else2*fib(n-1)
```

## 3 矩形覆盖

我们可以用  $2 \times 1$  的小矩形横着或者竖着去覆盖更大的矩形。 请问用  $n$  个  $2 \times 1$  的小矩形无重叠地覆盖一个  $2 \times n$  的大矩形，总共有多少种方法？

第  $2 \times n$  个矩形的覆盖方法等于第  $2 \times (n-1)$  加上第  $2 \times (n-2)$  的方法。

```
f=lambdan:1ifn<2elsef(n-1)+f(n-2)
1
```

## 4 杨氏矩阵查找

在一个  $m$  行  $n$  列二维数组中， 每一行都按照从左到右递增的顺序排序， 每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

## 5 去除列表中的重复元素

用集合

```
1 list(set(l))
```

用字典

```
1 l1=['b','c','d','b','c','a','a']
2 l2={}.fromkeys(l1).keys()
3 printl2
```

用字典并保持顺序

```
1 l1=['b','c','d','b','c','a','a']
2 l2=list(set(l1))
3 l2.sort(key=l1.index)
4 printl2
```

```
1 l1=['b','c','d','b','c','a','a']
2 l2=[]
3 [l2.append(i)foriinl1 ifnotiinl2]
```

面试官提到的 ,先排序然后删除 .

## 6 链表成对调换

1->2->3->4 转换成 2->1->4->3 .

```
classListNode:
    def__init__(self,x):
        self.val=x
        self.next=None

classSolution:
    # @param a ListNode
    # @return a ListNode
    defswapPairs(self,head):
        ifhead!=Noneandhead.next!=None:
            next=head.next
            head.next=self.swapPairs(next.next)
            next.next=head
            returnnext
        returnhead
```

## 7 创建字典的方法

### 1 直接创建

```
1 dict={'name':'earth','port':'80'}
```

### 2 工厂方法

```
1 items=[('name','earth'),('port','80')]
2 dict2=dict(items)
3 dict1=dict([('name','earth'],['port','80']))
```

### 3 fromkeys() 方法

```
1 dict1={}.fromkeys(('x','y'),-1)
2 dict={'x':-1,'y':-1}
3 dict2={}.fromkeys(('x','y'))
4 dict2={'x':None,'y':None}
```

## 8 合并两个有序列表

知乎远程面试要求编程

尾递归

```
def _recursion_merge_sort2(l1,l2,tmp):
    if len(l1)==0 or len(l2)==0:
        tmp.extend(l1)
        tmp.extend(l2)
        return tmp
    else:
        if l1[0]<l2[0]:
            tmp.append(l1[0])
            dell1[0]
        else:
            tmp.append(l2[0])
            dell2[0]
    return _recursion_merge_sort2(l1,l2,tmp)

def recursion_merge_sort2(l1,l2):
    return _recursion_merge_sort2(l1,l2,[])
```

循环算法

```
def loop_merge_sort(l1,l2):
    tmp=[]
    while len(l1)>0 and len(l2)>0:
        if l1[0]<l2[0]:
            tmp.append(l1[0])
            dell1[0]
        else:
            tmp.append(l2[0])
            dell2[0]
```



```
tmp.extend(l1)
tmp.extend(l2)
return tmp
```

## 9 交叉链表求交点

去哪儿的面试 ,没做出来 .

```
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None
def node(l1, l2):
    length1, length2 = 0, 0
    # 求两个链表长度
    while l1.next:
        l1 = l1.next
        length1 += 1
    while l2.next:
        l2 = l2.next
        length2 += 1
    # 长的链表先走
    if length1 > length2:
        for _ in range(length1 - length2):
            l1 = l1.next
    else:
        for _ in range(length2 - length1):
            l2 = l2.next
    while l1 and l2:
        if l1.next == l2.next:
            return l1.next
        else:
            l1 = l1.next
            l2 = l2.next
```

## 10 二分查找

```
def binarySearch(l, t):
    low, high = 0, len(l) - 1
    while low < high:
        print low, high
```

```

        mid=(low+high)/2
        ifl[mid]>t:
            high=mid
        elifl[mid]<t:
            low=mid+1
        else:
            returnmid
    returnlow ifl[low]==telseFalse

if__name__=='__main__':
    l=[1,4,12,45,66,99,120,444]
    printbinarySearch(l,12)
    printbinarySearch(l,1)
    printbinarySearch(l,13)
    printbinarySearch(l,444)

```

## 11 快排

```

1 defqsort(seq):
2     ifseq==[]:
3         return[]
4     else:
5         pivot=seq[0]
6         lesser=qsort([xforxinseq[1:]ifx<pivot])
7         greater=qsort([xforxinseq[1:]ifx>=pivot])
8         returnlesser+[pivot]+greater
9
10 if__name__=='__main__':
11     seq=[5,6,78,9,0,-1,2,3,-65,12]
12     print(qsort(seq))

```

## 12 找零问题

```

def coinChange(values,money,coinsUsed):
    #values T[1:n] 数组
    #valuesCounts 钱币对应的种类数
    #money 找出来的总钱数
    #coinsUsed 对应于目前钱币总数 i 所使用的硬币数目
    forcents inrange(1,money+1):
        minCoins=cents #从第一个开始到 money 的所有情况初始
        forvalue invalues:
            ifvalue<=cents:
                temp=coinsUsed[cents-value]+1

```

```

        iftemp<minCoins:
            minCoins=temp
        coinsUsed[cents]=minCoins
    print(' 面值为： {0}  的最小硬币数目为：    {1} '.format(cents,coinsUsed[cents]))

if__name__=='__main__':
    values=[25,21,10,5,1]
    money=63
    coinsUsed={i:0foriinrange(money+1)}
    coinChange(values,money,coinsUsed)

```

## 13 广度遍历和深度遍历二叉树

给定一个数组，构建二叉树，并且按层次打印这个二叉树

```

## 14  二叉树节点
classNode(object):
    def__init__(self,data,left=None,right=None):
        self.data=data
        self.left=left
        self.right=right

tree=Node(1,Node(3,Node(7,Node(0)),Node(6)),Node(2,Node(5),Node(4)))

## 15  层次遍历
deflookup(root):
    stack=[root]
    whilestack:
        current=stack.pop(0)
        printcurrent.data
        ifcurrent.left:
            stack.append(current.left)
        ifcurrent.right:
            stack.append(current.right)

## 16  深度遍历
defdeep(root):
    ifnotroot:
        return
    printroot.data
    deep(root.left)
    deep(root.right)

if__name__=='__main__':

```

```
lookup(tree)
deep(tree)
```

## 17 前中后序遍历

深度遍历改变顺序就 OK 了

## 18 求最大树深

```
1 defmaxDepth(root):
2     ifnotroot:
3         return0
4     returnmax(maxDepth(root.left),maxDepth(root.right))+1
```

## 19 求两棵树是否相同

```
1 defisSameTree(p,q):
2     ifp==Noneandq==None:
3         returnTrue
4     elifpandq:
5         returnp.val==q.val andisSameTree(p.left,q.left)andisSameTree(p.right,q.right)
6     else:
7         returnFalse
```

## 20 前序中序求后序

```
defrebuild(pre,center):
    ifnotpre:
        return
    cur=Node(pre[0])
    index=center.index(pre[0])
    cur.left=rebuild(pre[1:index+1],center[:index])
    cur.right=rebuild(pre[index+1:],center[index+1:])
    returncur
```

```
defdeep(root):
    ifnotroot:
        return
    deep(root.left)
    deep(root.right)
```

```
printroot.data
```

## 21 单链表逆置

```
classNode(object):
    def__init__(self,data=None,next=None):
        self.data=data
        self.next=next

link=Node(1,Node(2,Node(3,Node(4,Node(5,Node(6,Node(7,Node(8,Node(9))))))))))

defrev(link):
    pre=link
    cur=link.next
    pre.next=None
    whilecur:
        tmp=cur.next
        cur.next=pre
        pre=cur
        cur=tmp
    returnpre

root=rev(link)
whileroot:
    printroot.data
    root=root.next
```

## Python Web 相关

解释一下 **WSGI** 和 **FastCGI** 的关系？

CGI 全称是“公共网关接口”(CommonGateway Interface)，HTTP 服务器与你的或其它机器上的程序进行“交谈”的一种工具，其程序须运行在网络服务器上。 CGI 可以用任何一种语言编写，只要这种语言具有标准输入、输出和环境变量。如 php,perl,tcl 等。

FastCGI 像是一个常驻 (long-live) 型的 CGI , 它可以一直执行着 , 只要激活后 , 不会每次都要花费时间去 fork 一次 (这是 CGI 最为人诟病的 fork-and-execute 模式) 。它还支持分布式的运算 , 即 FastCGI 程序可以在网站服务器以外的主机上执行并且接受来自其它网站服务器来的请求。

FastCGI 是语言无关的、可伸缩架构的 CGI 开放扩展 , 其主要行为是将 CGI 解释器进程保持在内存中并因此获得较高的性能。众所周知 , CGI 解释器的反复加载是 CGI 性能低下的主要原因 , 如果 CGI 解释器保持在内存中并接受 FastCGI 进程管理器调度 , 则可以提供良好的性能、伸缩性、Fail- Over 特性等等。

WSGI 的全称为 : PythonWeb Server Gateway Interface v1.0 ( Python Web 服务器网关接口 ) ,

它是 Python 应用程序和 WEB 服务器之间的一种接口。

它的作用 , 类似于 FCGI 或 FASTCGI 之类的协议的作用。

WSGI 的目标 , 是要建立一个简单的普遍适用的服务器与 WEB 框架之间的接口。

Flup 就是使用 Python 语言对 WSGI 的一种实现 , 是可以用于 Python 的应用开发中的一种工具或者说是一种库。

Spawn-fcgi 是一个小程序 , 这个程序的作用是管理 fast-cgi 进程 , 那么管理 wsgi 进程也是没有问题的 , 功能和 php-fpm 类似。

故，简单地说，WSGI 和 FastCGI 都是一种 CGI，用于连接 WEB 服务器与应用程序，而 WSGI 专指 Python 应用程序。而 flup 是 WSGI 的一种实现，Spawn-fcgi 是用于管理 flup 进程的一个工具，可以启动多个 wsgi 进程，并管理它们。

## 解释一下 Django 和 Tornado 的关系、差别

Django 源自一个在线新闻 Web 站点，于 2005 年以开源的形式被释放出来。

Django 框架的核心组件有：

用于创建模型的对象关系映射为最终用户设计的完美管理界面一流的 URL 设计设计者友好的模板语言缓存系统等等

它鼓励快速开发，并遵循 MVC 设计。Django 遵守 BSD 版权，最新发行版本是

Django

1.4，于 2012 年 03 月 23 日发布。Django 的主要目的是简便、快速的开发数据库驱动的网站。它强调代码复用，多个组件可以很方便的以“插件”形式服务于整个框架，Django 有许多功能强大的第三方插件，你甚至可以很方便的开发出自己的工具包。

这使得 Django 具有很强的可扩展性。它还强调快速开发和 DRY(Do Not Repeat Yourself) 原则。

Tornado 是 FriendFeed 使用的可扩展的非阻塞式 web 服务器及其相关工具的开源版本。这个 Web 框架看起来有些像 web.py 或者 Google 的 webapp，不过为了能有效利用非阻塞式服务器环境，这个 Web 框架还包含了一些相关的有用工具和优化。

Tornado 和现在的主流 Web 服务器框架（包括大多数 Python 的框架）有着明显的区别：它是非阻塞式服务器，而且速度相当快。得益于其非阻塞的方式和对 epoll 的运用，Tornado 每秒可以处理数以千计的连接，这意味着对于实时 Web 服务来说，Tornado 是一个理想的 Web 框架。我们开发这个 Web 服务器的主要目的就是为了处理 FriendFeed 的实时功能——在 FriendFeed 的应用里每一个活动用户都会保持着一个服务器连接。（关于如何扩容服务器，以处理数以千计的客户端的连接的问题。

## 解释下 django-debug-toolbar 的使用

使用 django 开发站点时，可以使用 django-debug-toolbar 来进行调试。在 settings.py 中添加 'debug\_toolbar.middleware.DebugToolbarMiddleware'，到项目的 MIDDLEWARE\_CLASSES 内。

## 解释下 Django 使用 redis 缓存服务器

为了能在 Django 中使用 redis，还需要安装 redis for Django 的插件。然后在 Django 的 settings 中配置了。现在连接和配置都已经完成了，接下来是一个简单的例子：

```
from django.conf import settings
from django.core.cache import cache
#read cache user id
def read_from_cache(self,user_name):
    key='user_id_of_'+user_name
    value=cache.get(key)
    if value==None:
        data=None
    else:
```



```

        data=json.loads(value)
    returndata
#write cache user id
def write_to_cache(self,user_name):
    key='user_id_of_'+user_name
    cache.set(key,json.dumps(user_name),settings.NEVER_REDIS_TIMEOUT)

```

## 如何进行 Django 单元测试

Django 的单元测试使用 python 的 unittest 模块，这个模块使用基于类的方法来定义测试。类名为 `django.test.TestCase`，继承于 python 的 `unittest.TestCase`。

```

from django.test import TestCase
from myapp.models import Animal

class AnimalTestCase(TestCase):
    def setUp(self):
        Animal.objects.create(name="lion",sound="roar")
        Animal.objects.create(name="cat",sound="meow")

    def test_animals_can_speak(self):
        """Animals that can speak are correctly identified"""
        lion=Animal.objects.get(name="lion")
        cat=Animal.objects.get(name="cat")
        self.assertEqual(lion.speak(),'The lion says "roar"')
        self.assertEqual(cat.speak(),'The cat says "meow"')

```

执行目录下所有的测试（所有的 `test*.py` 文件）：运行测试的时候，测试程序会在所有以 `test` 开头的文件中查找所有的 `test cases`(`unittest.TestCase` 的子类),自动建立测试集然后运行测试。

```
1 $python manage.py test
```

执行 animals 项目下 tests 包里的测试：

```
$python manage.py testanimals.tests
```

执行 animals 项目里的 test 测试：

```
1 $python manage.py testanimals
```

单独执行某个 test case：

```
1 $python manage.py testanimals.tests.AnimalTestCase
```

单独执行某个测试方法：

```
1 $python manage.py testanimals.tests.AnimalTestCase.test_animals_can_speak
```

为测试文件提供路径：

```
1 $python manage.py testanimals/
```

通配测试文件名：

```
1 $python manage.py test--pattern="tests_*.py"
```

启用 warnings 提醒：

```
1 $python-Wall manage.py test
```

## 解释下 Http 协议

HTTP 是一个属于应用层的面向对象的协议，由于其简捷、快速的方式，适用于分布式超媒体信息系统。

HTTP 协议的主要特点可概括如下：

1.支持客户 / 服务器模式。

2.简单快速：客户向服务器请求服务时，只需传送请求方法和路径。请求方法常用的有 GET、HEAD、POST。每种方法规定了客户与服务器联系的类型不同。由于 HTTP 协议简单，使得 HTTP 服务器的程序规模小，因而通信速度很快。

3.灵活：HTTP 允许传输任意类型的数据对象。正在传输的类型由 Content-Type 加以标记。

4.无连接：无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间。

5.无状态：HTTP 协议是无状态协议。无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快。

## 解释下 Http 请求头和常见响应状态码

**Accept:** 指浏览器或其他客户可以接爱的 MIME 文件格式。可以根据它判断并返回适当的文件格式。

**Accept-Charset** : 指出浏览器可以接受的字符编码。英文浏览器的默认值是 ISO-8859-1.

**Accept-Language** : 指出浏览器可以接受的语言种类, 如 en 或 en-us , 指英语。

**Accept-Encoding** : 指出浏览器可以接受的编码方式。编码方式不同于文件格式, 它是为了压缩文件并加速文件传递速度。浏览器在接收到 Web 响应之后先解码, 然后再检查文件格式。

**Cache-Control** : 设置关于请求被代理服务器存储的相关选项。一般用不到。

**Connection** : 用来告诉服务器是否可以维持固定的 HTTP 连接。HTTP/1.1 使用 **Keep-Alive** 为默认值, 这样, 当浏览器需要多个文件时 (比如一个 HTML 文件和相关的图形文件), 不需要每次都建立连接。

**Content-Type** : 用来表明 request 的内容类型。可以用 `HttpServletRequest` 的 `getContentType()` 方法取得。

**Cookie** : 浏览器用这个属性向服务器发送 Cookie 。Cookie 是在浏览器中寄存的小型数据体, 它可以记载和服务相关的用户信息, 也可以用来实现会话功能。

状态代码有三位数字组成，第一个数字定义了响应的类别，且有五种可能取值：

1xx：指示信息 – 表示请求已接收，继续处理

2xx：成功 – 表示请求已被成功接收、理解、接受

3xx：重定向 – 要完成请求必须进行更进一步的操作

4xx：客户端错误 – 请求有语法错误或请求无法实现

5xx：服务器端错误 – 服务器未能实现合法的请求

常见状态代码、状态描述、说明：

200 OK // 客户端请求成功

400 Bad Request // 客户端请求有语法错误，不能被服务器所理解

401 Unauthorized // 请求未经授权，这个状态代码必须和 WWW-Authenticate 报头域一起使用

403 Forbidden // 服务器收到请求，但是拒绝提供服务

404 Not Found // 请求资源不存在， eg：输入了错误的 URL

500 Internal Server Error // 服务器发生不可预期的错误

503 Server Unavailable // 服务器当前不能处理客户端的请求，一段时间后可能恢复正常

eg : HTTP/1.1 200 OK    ( CRLF )

# 爬虫

一、试列出至少三种目前流行的大型数据库的名称 :\_\_\_\_\_、  
\_\_\_\_\_,\_\_\_\_\_ 其中您最熟悉的是 \_\_\_\_\_ 从 \_\_\_\_\_ 年开始使用。

Oracle , Mysql , SQLServer Oracle  根据自己情况

二、有表 List , 并有字段 A、B、C , 类型都是整数。表中有如下几条记录 :

A	B	C
2	7	9
5	6	4
3	11	9

现在对该表一次完成以下操作 :

查询出 B 和 C 列的值 , 要求按 B 列升序排列

写出一条新的记录 , 值为 {7,9,8}

查询 C 列 , 要求消除重复的值 , 按降序排列

写出完成完成以上操作的标准的    SQL 语句 , 并且写出操作 3 的结果。

create table List(A int,B int,C int)

Select B,C from List order by B

Insert into List values(7,9,8)

Select distinct(C) from List order by desc;

984

### 三、请简要说明视图的作用

- 1.数据库视图隐藏了数据的复杂性。
- 2.数据库视图有利于控制用户对表中某些列的访问。
- 3.数据库视图使用户查询变得简单。

### 四、列举您使用过的 python 网络爬虫所用到的网络数据包（最熟悉的在前）：

requests 、 urllib 、 urllib2 、 httplib2

### 五、列举您使用过的 python 网络爬虫所用到的解析数据包（最熟悉的在前）：

BeautifulSoup 、 pyquery 、 Xpath 、 lxml

### 六、列举您使用过的 python 中的编码方式（最熟悉的在前）：

UTF-8 , ASCII , gbk



### 七、python3.5 语言中 enumerate 的意思是 \_\_\_\_\_

☐ 对于一个可迭代的 ( iterable ) / 可遍历的对象 ( 如列表、字符串 ) ,  
enumerate 将其组成一个索引序列, 利用它可以同时获得索引和值  
enumerate 多用于在 for 循环中得到计数

八、99 的八进制表示是 \_\_\_\_\_

☐ 143

九、请举出三种常用的排序算法

☐ 冒泡、选择、快速

十、列出比较熟悉的爬虫框架

☐ Scrapy

十一、用 4、9、2、7 四个数字, 可以使用 +、-、\* 和 / , 每个数字使用一次, 使表达式的结果为 24 , 表达式是

☐  $(9+7-4) * 2$

十二、对你最有影响的或是您认为最有价值的软件方面的几本书是?

十三、您最熟悉的 Unix 环境是 \_\_\_\_\_. Unix 下查询环境变量的命令是 \_\_\_\_\_ 查询脚本定时任务的命令是 \_\_\_\_\_

☐ AIX , envcrontab

十四、写出在网络爬虫爬取数据的过程中, 遇到的防爬虫问题的解决方案

☐ 通过 headers 反爬虫: 解决策略, 伪造 headers

基于用户行为反爬虫: 动态变化去爬取数据, 模拟普通用户的行为



基于动态页面的反爬虫：跟踪服务器发送的 ajax 请求，模拟 ajax 请求

十五、阅读以下 Python 程序

```
for i in range(5,0,-1):
```

```
    print(i)
```

请在下面写出打印结果

54321

十六、在某系统中一个整数占用两个八位字节，使用 Python 按下面的要求编写完整程序。

接收从标准输入中依次输入的五個数字，将其组合成为一个整数，放入全局变量 n 中，随后在标准输出输出这个整数。（ord(char) 获取字符 ASCII 值的函数）

人，从刚出生来到这个世界，便开始探索这个世界。累了就歇会，精神了就继续探索，直至死亡。

# python 面试题总结



### 1. Python 线程池原理？

我理解为线程池是一个存放很多线程的单位，同时还有一个对应的任务队列。整个执行过程其实就是使用线程池中已有有限的线程把任务队列中的任务做完。这样做的好处就是你不需为每个任务都创建一个线程，因为当你创建第 100 个线程来执行第 100 个任务的时候，可能前面已经有 50 个线程结束工作了。超过最大值的线程可以排队，但他们要等到其他线程完成后才启动。因此重复利用线程来执行任务，减少系统资源的开销。

### 2. Python 合并字典，相同 key 的 value 如何相加？

利用 collections.Counter 可轻松办到

```
>>> x = { 'apple': 1, 'banana': 2 }
>>> y = { 'banana': 10, 'pear': 11 }
>>> from collections import Counter
>>> XY = Counter(x), Counter(y)
>>> z = dict(X+Y)
>>> z
{'apple': 1, 'banana': 12, 'pear': 11}
```

另一种写法

```
>>> from collections import Counter
>>> dict(Counter(x)+Counter(y))
```

( 合并两个字典的方法

方法 1：

```
dictMerged1=dict(dict1.items()+dict2.items())
```

方法 2：

```
dictMerged2=dict(dict1, **dict2)
```

方法 2 等同于：

```
dictMerged=dict1.copy()
```

```
dictMerged.update(dict2)
```

或者

```
dictMerged=dict(dict1)
```

```
dictMerged.update(dict2)
```

### 3. 解释 GUI 和 GPL？

GUI 图形用户 界面 ( Graphical User Interface ，简称 GUI，又称图形 用户接口 ) 是指采用图形方式显示的计算机操作用户 界面。

## GPL(GNU通用公共许可证)

GPL 同其它的自由软件许可证一样，许可社会公众享有：运行、复制软件的自由，发行传播软件的自由，获得软件源码的自由，改进软件并将自己作出的改进版本向社会发行传播的自由。

### 4.简述爬虫的基本步骤和流程？

网络爬虫的基本工作流程如下：

- 1.首先选取一部分精心挑选的 URL；
- 2.将这些 URL 放入待抓取 URL 队列；
- 3.从待抓取 URL 队列中取出待抓取在 URL (解析 DNS ,并且得到主机的 ip ,) 并将 URL 对应的网页下载下来，存储进已下载网页库中。此外，将这些 URL 放进已抓取 URL 队列。
- 4.分析已抓取 URL 队列中的 URL 和其中的有价值的数据，将新的 URL，放入待抓取 URL 队列，将数据存储起来，从而进入下一个循环。

### 5.你们公司的业务中，并发能达到多少？

### 6. 如何循环抓取一个网站的 1000 张图片？

流程大概是这样

找到所有页数

- 遍历所有的页数
- 遍历当前页的所有相册（给每个相册建立一个目录）
- 遍历当前相册的所有图片（遍历此相册的所有页（遍历当前页的所有照片并找到图片的 url））
- 获得图片 url 就存起来，然后通过图片 url 下载图片。
- 引用计数当图片下载量达到 1000 张时，停止爬取。

### 7. 如果对方网站可以反爬取，封 ip 怎么办？

- 1、放慢抓取速度，减小对于目标网站造成的压力。但是这样会减少单位时间类的抓取量。
- 2、第二种方法是通过设置代理 IP 等手段，突破反爬虫机制继续高频率抓取。但是这样需要多个稳定的代理 IP。  
代理 IP 可以搜索到免费的，但是可能不太稳定，也有收费的，但是不一定划算，也可能不是长久之计。

### 普通的基于 ADSL拨号的解决办法

通常，在抓取过程中遇到禁止访问，可以重新进行 ADSL 拨号，获取新的 IP，从而可以继续抓取。但是这样在多网站多线程抓取的时候，如果某一个网站的抓

取被禁止了，  
同时也影响到了其他网站的抓取，整体来说也会降低抓取速度。

一种可能的解决办法

同样也是基于 ADSL 拨号，不同的是，需要两台能够进行 ADSL 拨号的服务器，  
抓取过程中使用这两台服务器作为代理。

假设有 A、B 两台可以进行 ADSL 拨号的服务器。爬虫程序在 C 服务器上运行，  
使用 A 作为代理访问外网，如果在抓取过程中遇到禁止访问的情况，立即将代理  
切换为 B，然后将 A 进行重新拨号。如果再遇到禁止访问就切换为 A 做代理，  
B 再拨号，如此反复。如下图：

使用 A 为代理，B 拨号：

#### 8. 简述 移动端 app 数据的抓取流程

- 1、例如：爬取手机淘宝，核心还是一个 WEB 页面：<https://m.taobao.com/#index>
- 2、有很多资讯类的 APP，核心都是一个 WEB 页面。直接用爬虫的方法抓就可以了。
- 3、非 WEB 页面的 APP，用 APP 自动化的一些东西试试
- 4、不然就只有抓包了。

#### 9. 是否了解 Python 的函数参数传递的机制？

python 的参数传递机制具有值传递（int、float 等值数据类型）和引用传递（以字典、列表等非值对象数据类型为代表）两种基本机制以及方便的关键字传递特性（直接使用函数的形参名指定实参的传递目标，如函数定义为 `def f(a,b,c)`，那么在调用时可以采用 `f(b=1,c=2,a=3)` 的指定形参目标的传递方式，而不必拘泥于 c 语言之类的形参和实参按位置对应）

除此之外，python 中还允许包裹方式的参数传递，这为不确定参数个数和参数类型的函数调用提供了基础：

```
def f(*a,**b)
```

包裹参数传递的实现是在定义函数时在形参前面加上 \* 或 \*\*，\* 所对应的形参（如上面的 a）会被解释为一个元组（tuple），而 \*\* 所对应的形参（如上面的 b）会被解释为一个字典。

#### 10. 是否了解 \*args 和 \*\*kwargs 的用法？

当函数的参数不确定时，可以使用 \*args 和 \*\*kwargs，\*args 没有 key 值（列表），\*\*kwargs 有 key 值（字典）。

#### 11. 是否了解 Python 中的单例模式。有哪几种实现？

单例模式是一种常用的软件设计模式。在它的核心结构中只包含一个被称为单例类的特殊类。通过单例模式可以保证系统中一个类只有一个实例而且该实例易于

外界访问，从而方便对实例个数的控制并节约系统资源。如果希望在系统中某个类的对象只能存在一个，单例模式是最好的解决方案。

单例模式的要点有三个；一是某个类只能有一个实例；二是它必须自行创建这个实例；三是它必须自行向整个系统提供这个实例。在 Python 中，单例模式有以下几种实现方式。

方法一、实现 `__new__` 方法，然后将类的一个实例绑定到类变量 `_instance` 上；如果 `cls._instance` 为 `None`，则说明该类还没有被实例化过，`new` 一个该类的实例，并返回；如果 `cls._instance` 不为 `None`，直接返回 `_instance`，代码如下：

```
?
1 class Singleton(object):
2
3     def __new__(cls, *args, **kwargs):
4         if not hasattr(cls, '_instance'):
5             orig = super(Singleton, cls)
6             cls._instance = orig.__new__(cls, *args, **kwargs)
7         return cls._instance
8
9 class MyClass(Singleton):
10     a = 1
11
12 one = MyClass()
13 two = MyClass()
14
15 # one 和 two 完全相同，可以用 id(), ==, is 检测
16 print id(one)      # 29097904
17 print id(two)      # 29097904
18 print one == two   # True
19 print one is two   # True
```

方法二、本质上是方法一的升级版，使用 `__metaclass__` (元类) 的高级 python 用法，具体代码如下：

```
?
1 class Singleton2(type):
2
3     def __init__(cls, name, bases, dict):
4         super(Singleton2, cls).__init__(name, bases, dict)
5         cls._instance = None
6
7     def __call__(cls, *args, **kwargs):
```

```

8         if cls._instance is None:
9             cls._instance = super(Singleton2, cls).__call__(*arg
1 s, **kwargs)
0         return cls._instance
1
1 class MyClass2(object):
1     __metaclass__ = Singleton2
2     a = 1
1
3 one = MyClass2()
1 two = MyClass2()
4
1 print id(one)      # 31495472
5 print id(two)      # 31495472
1 print one == two   # True
6 print one is two   # True
1
7
1
8
1
9
2
0
2
1
2
2

```

方法三、使用 Python 的装饰器 (decorator) 实现单例模式，这是一种更 Pythonic 的方法；单利类本身的代码不是单例的，通装饰器使其单例化，代码如下：

?

```

1 def singleton(cls, *args, **kwargs):
2     instances = {}
3     def _singleton():
4         if cls not in instances:
5             instances[cls] = cls(*args, **kwargs)
6         return instances[cls]

```



```

7     return _singleton
8
9 @singleton
10 class MyClass3(object):
11     a = 1
12
13 one = MyClass3()
14 two = MyClass3()
15
16 print id(one)      # 29660784
17 print id(two)      # 29660784
18 print one == two   # True
19 print one is two   # True

```

12.是否了解 Python 垃圾回收机制？

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。引用计数最大缺陷就是循环引用的问题，如果一个对象的引用数为 0，Python 虚拟机就会回收这个对象的内存。所以 Python 采用了辅助方法。

13.是否了解 read,readline和 readlines三者的区别是什么？

read 读取整个文件

readline 读取下一行

readlines 读取整个文件到一个迭代器以供我们遍历（读取到一个 list 中，以供使用，比较方便）

14.是否了解 Post和 Get的区别是什么？

（Http 定义了与服务器交互的不同方法，最基本的方法有 4 种，分别是 GET,POST, PUT, DELETE URL全称是资源描述符，我们可以这样认为：一个 URL地址，它用于描述一个网络上的资源，而 HTTP中的 GET, POST, PUT, DELET就对应着对这个资源的查，改，增，删 4 个操作。）

1. 根据 HTTP规范，GET用于信息获取，而且应该是安全的和幂等的。

(1).所谓安全的意味着该操作用于获取信息而非修改信息。换句话说，GET 请求一般不应产生副作用。就是说，它仅仅是获取资源信息，就像数据库查询一样，不会修改，增加数据，不会影响资源的状态。

\* 注意：这里安全的含义仅仅是指是非修改信息。

(2).幂等的意味着对同一 URL 的多个请求应该返回同样的结果

2.根据 HTTP 规范，POST 表示可能修改变服务器上的资源的请求。

以新闻网站为例，读者对新闻发表自己的评论应该通过 POST 实现，因为在评论提交后站点的资源已经不同了，或者说资源被修改了。

上面大概说了一下 HTTP 规范中 GET 和 POST 的一些原理性的问题。但在实际的做的时候，很多人却没有按照 HTTP 规范去做，导致这个问题的原因有很多，比如说：

1.很多人贪方便，更新资源时用了 GET，因为用 POST 必须要到 FORM(表单)，这样会麻烦一点。

2.对资源的增，删，改，查操作，其实都可以通过 GET/POST 完成，不需要用到 PUT 和 DELETE。

3.另外一个，早期的 Web MVC 框架设计者们并没有有意识地将 URL 当作抽象的资源来看待和设计，所以导致一个比较严重的问题是传统的 Web MVC 框架基本上都只支持 GET 和 POST 两种 HTTP 方法，而不支持 PUT 和 DELETE 方法。

区别：

(1.GET 请求的数据会附在 URL 之后(就是把数据放置在 HTTP 协议头中)，以？分割 URL 和传输数据，参数之间以 &相连，如：

login.action?name=hyddd&password=idontknow&verify=%E4%BD%A0%E5%A5%BD。如果数据是英文字母/数字，原样发送，如果是空格，转换为 +，如果是中文/其他字符，则直接把字符串用 BASE64 加密，得出

如：%E4%BD%A0%E5%A5%BD，其中 %XX 中的 XX 为该符号以 16 进制表示的 ASCII。

POST 把提交的数据则放置在是 HTTP 包的包体中。

2."GET 方式提交的数据最多只能是 1024 字节，理论上 POST 没有限制，可传较大量的数据)

以上这句是我从其他文章转过来的，其实这样说是错误的，不准确的：

(1).首先是"GET 方式提交的数据最多只能是 1024 字节"，因为 GET 是通过 URL 提交数据，那么 GET 可提交的数据量就跟 URL 的长度有直接关系了。而实际上，URL 不存在参数上限的问题，HTTP 协议规范没有对 URL 长度进行限制。这个限制是特定的浏览器及服务器对它的限制。IE 对 URL 长度的限制是 2083 字节 (2K+35)。对于其他浏览器，如 Netscape Firefox 等，理论上没有长度限制，其限制取决于操作系统的支持。

注意这是限制是整个 URL 长度，而不仅仅是你的参数值数据长度。 [见参考资料 5]

(2).理论上讲，POST 是没有大小限制的，HTTP 协议规范也没有进行大小限制，说“POST 数据量存在 80K/100K 的大小限制”是不准确的，POST 数据是没有限制的，起限制作用的是服务器的处理程序的处理能力。

15.是否了解 Cookie 和 Session 的区别是什么？

(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，如果主要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

你服务器的性能，如果主要考虑到减轻服务器性能方面，应当使用

COOKIE

(4) 单个 cookie 在客户端的限制是 3K ,就是说一个站点在客户端存放的

COOKIE不能 3K。

(5) 所以：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

1，session在服务器端， cookie 在客户端（浏览器）

2，session默认被存在在服务器的一个文件里（不是内存）

3，session的运行依赖 session id, 而 session id是存在 cookie 中的，也就是说，如果浏览器禁用了 cookie，同时 session也会失效（但是可以通过其它方式实现，比如在 url 中传递 session\_id)

4，session可以放在 文件、数据库、或内存中都可以。

5，用户验证这种场合一般会用 session

因此，维持一个会话的核心就是客户端的唯一标识，即 session id

16.是否了解 Python2.7.x与 Python3.x 的主要差异？

- [使用 \\_\\_future\\_\\_ 模块](#)
- [print 函数](#)
- [Integer division](#)
- [Unicode](#)
- [xrange](#)
- [Raising exceptions](#)
- [Handling exceptions](#)
- [next\(\) 函数 和 .next\(\) 方法](#)
- [For 循环变量和全局命名空间泄漏](#)
- [比较不可排序类型](#)
- [通过 input\(\) 解析用户的输入](#)
- [返回可迭代对象，而不是列表](#)

## 17.Python的数据类型

Python 提供的基本数据类型主要有：布尔类型、整型、浮点型、字符串、列表、元组、集合、字典等等。

## 18.tuple 和 list 的区别

列表里的内容是可以改变的，增删改都可以，tuple 则不行

不可变的 tuple 意义就在于因为不可变，所以代码更安全。如果可能，能用 tuple 就尽量用 tuple。

## 19.Yield 的用法

带有 yield 的函数在 Python 中被称之为 generator（生成器）。当这个函数在被调用的时候，这个函数在遇到 yield 的时候会停止运行，并返回一个迭代器。

其具体功能是可以当 return 使用，从函数里返回一个值，不同之处是用 yield 返回之后，可以让函数从上回 yield 返回的地点继续执行。也就是说，yield 返回函数，交给调用者一个返回值，然后再“瞬移”回去，让函数继续运行，直到下一跳 yield 语句再返回一个新的值。

## 20.If I want to debug my python scripts,which module I should import (C)

(如果我想调试我的 python 脚本,我应该导入模块 )

- A. pydebug
- B. debug
- C. pdb
- D. pydb

## 21.Which one is correct for the following script. (C)

(下列哪一个是正确的脚本 )

```
fruitlist = [,apple?,?mango?,?banana?,?orange?]  
print fruitlist[1:3]
```

- A. [,apple?,?mango?,?banana?]
- B. [,mango?,?banana?,?orange?]
- C. [,mango?,?banana?]
- D. [,apple?,?mango?]

## 22.Which of following options is not python data type? (A)

(以下哪个选项不是 python 数据类型 )

- A. char
- B. int
- C. float
- D. list

23.what king of data that cPickle.dump store in file? D  
(cpickle.dump 存储在文件中的什么样的数据 )

- A. Object
- B. String
- C. Array
- D. All of above

注 :在 python 中 ,一般可以使用 pickle 类来进行 python 对象的序列化 , 而 cPickle 提供了一个更快速简单的接口 , 如 python 文档所说的 : “ cPickle- A faster pickle 。” cPickle 可以对任意一种类型的 python 对象进行序列化操作 , 比如 list , dict , 甚至是一个类的对象等。 而所谓的序列化 , 我的粗浅的理解就是为了能够完整的保存并能够完全可逆的恢复

24.which of following options is python mutable data type? (D)  
(下列选项哪一个是 python 的可变数据类型 )

- A.int
- B.float
- C.tuple
- D.list

25.which of following options is unable to create dict ?(C)  
(以下哪个选项是无法创建 dict 类型 )

- A.dict1={}
- B.dict2={3:3}
- C.dict3={1,2,3:?uestc?} #TypeError: unhashable type: 'list'
- D.dict4={(1,2,3):?usetc?}

26.How to interpret the following implementation of the result? ( B )  
(如何解释以下的实现结果吗 )

```
>>>print 1.2-1.0 == 0.2  
>>>False
```

- A. The realization of Python error #Python的实现错误
- B. Float could not be accurately expressed浮点数不能准确表达
- C. Boolean operation can not be used for floating-data type compar

#布尔运算不能用于浮点数据类型进行了

- D. Python will be non-zero number as False

#Python 将非零数字是错误的

27.what would the following code yield? ()

(下面的代码会产生什么 )

```
>>>word = ,abcdefghij?
>>>print word[:3] + word[3:]
```

abcdefghij

28.Create a new list that converts the following list(num\_string) of number string to a list of number?

(创建一个新的列表，将列表 ( num\_string) 数的字符串列表的数字 )

Note:Please use List Comprehension Expression

(注意：请使用列表理解表达式 )

```
>>>num_strings = [,1?,?21?,?53?,?84?,?50?,?66?,?7?,?38?,?9?]
```

```
New_num = [ int(i) for i in num_strings ]
```

29.what will be the output of the following?

(写出下面的输出 )

```
>>>def func(x,*y,**z) :
```

Print z

```
>>>func(1,2,3)
```

答： {}

30.How to import method A and B from module M.

(如何从模块 M 导入方法 A 和 B)

```
from M import A , B
```

31.How to print “ Thisprogram can print hello world on screen. ”

(如何打印出 “ Thisprogram can print hello world on screen. ” )

```
def printHello():
```

```
,??Thisprogram can print hello world on screen???
```

```
Print “ helløworld ”
```

答： printHello.\_\_doc\_\_

32.建立一个用于下载文件的线程池，活动线程数为 5.

```
Download_urls = [  
    " http://test.com/test1.txt      " ,  
    " http://test.com/test2.txt      " ,  
    " http://test.com/test3.txt      " ,  
]
```

33.有多少种设计模式，实现其中一个设计模式？

## 设计模式 共 23 种 分三类

设计模式主要分三个类型：创建型、结构型和行为型。

其中

创建型 有：

- 一、 Singleton ，单例模式：保证一个类只有一个实例，并提供一个访问它的全局访问点
- 二、 Abstract Factory ，抽象工厂：提供一个创建一系列相关或相互依赖对象的接口，而无须指定它们的具体类。
- 三、 Factory Method ，工厂方法：定义一个用于创建对象的接口，让子类决定实例化哪一个类，Factory Method 使一个类的实例化延迟到了子类。
- 四、 Builder ，建造模式：将一个复杂对象的构建与他的表示相分离，使得同样的构建过程可以创建不同的表示。
- 五、 Prototype ，原型模式：用原型实例指定创建对象的种类，并且通过拷贝这些原型来创建新的对象。

行为型 有：

- 六、 Iterator ，迭代器模式：提供一个方法顺序访问一个聚合对象的各个元素，而又不需要暴露该对象的内部表示。
- 七、 Observer ，观察者模式：定义对象间一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知自动更新。
- 八、 Template Method ，模板方法：定义一个操作中的算法的骨架，而将一些步骤延迟到子类中，TemplateMethod 使得子类可以不改变一个算法的结构即可以重定义该算法得某些特定步骤。
- 九、 Command ，命令模式：将一个请求封装为一个对象，从而使你可以用不同的请求对客户进行参数化，对请求排队和记录请求日志，以及支持可撤销的操作。
- 十、 State ，状态模式：允许对象在其内部状态改变时改变他的行为。对象看起来似乎改变了他的类。
- 十一、 Strategy ，策略模式：定义一系列的算法，把他们一个个封装起来，并使他们可以互相替换，本模式使得算法可以独立于使用它们的客户。

十二、 Chain of Responsibility ，职责链模式：使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系

十三、 Mediator ，中介者模式：用一个中介对象封装一些列的对象交互。

十四、 Visitor ，访问者模式：表示一个作用于某对象结构中的各元素的操作，它使你可以在不改变各元素类的前提下定义作用于这个元素的新操作。

十五、 Interpreter ，解释器模式： 给定一个语言， 定义他的文法的一个表示， 并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

十六、 Memento ，备忘录模式：在不破坏对象的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。

结构型 有：

十七、 Composite ，组合模式：将对象组合成树形结构以表示部分整体的关系， Composite 使得用户对单个对象和组合对象的使用具有一致性。

十八、 Facade ，外观模式： 为子系统中的一组接口提供一致的界面， facade 提供了一高层接口，这个接口使得子系统更容易使用。

十九、 Proxy ，代理模式：为其他对象提供一种代理以控制对这个对象的访问

二十、 Adapter, 适配器模式：将一类的接口转换成客户希望的另外一个接口， Adapter 模式使得原本由于接口不兼容而不能一起工作那些类可以一起工作。

二十一、 Decorator ，装饰模式：动态地给一个对象增加一些额外的职责，就增加的功能来说， Decorator 模式相比生成子类更加灵活。

二十二、 Bridge ，桥模式：将抽象部分与它的实现部分相分离，使他们可以独立的变化。

二十三、 Flyweight ，享元模式： 使用共享物件，用来尽可能减少内存使用量以及分享资讯给尽可能多的相似物件； 它适合于当大量物件只是重复因而导致无法令人接受的使用大量内存。 通常物件中的部分状态是可以分享。 常见做法是把它们放在外部数据结构， 当需要使用时再将它们传递给享元。

代理模式 ,Python 实现：

```
class Interface:
    def Request(self):
        return 0
class RealSubject(Interface):
    def Request(self):
        print "Real request."
class Proxy(Interface):
    def Request(self):
        self.real = RealSubject()
        self.real.Request()
if __name__ == "__main__":
    p = Proxy()
```



p.Request()

34.用匿名函数写一个功能？

- `print (lambda x:x*x)(4)`

### 35. 爬取淘宝的流程

36.MySQL和 mongodb的区别

**mongodb**与 **mysql**命令对比 传统的关系数据库一般由数据库 ( **database**)、表( **table**)、记录 ( **record**) 三个层次概念组成，

**MongoDB** 是由数据库 ( **database**)、集合 ( **collection**)、文档对象 ( **document**) 三个层次组成。

**MongoDB** 对于关系型数据库里的表，但是集合中没有列、行和关系概念，这体现了模式自由的特点。

37.分析日志的过程

先考虑最简单的情况，在数据规模比较小的时候，也许是几十 MB 几百 MB或者几十 GB, 总之就是在单机处理尚能忍受的时候。一切都很好办，现成的各种 Unix/Linux 工具——awk、grep、sort、join 等都是日志分析的利器，如果仅仅是想知道某个页面的 PV，一个 wc+grep 就能搞定。如果有稍复杂的逻辑，那就使用 Python，配合正则表达式，基本就可以解决所有的问题。

38.爬虫的主要模块

urllib urllib2 BeautifulSoup requests scrapy

39.怎样验证 URL 是否存在

URL 不存在，可能时两种情况

网页在服务器上不存在

服务器不存在

第一种异常发生时，爬虫程序 ( urllib 模块 ) 会返回 HTTP 错误，urlopen 函数会抛出 " HTTPError异常

使用异常处理

try:

    Html =urlopen(URL)

except HTTPError as e:

    print(e)

第二中情况 urlopen 会返回一个 None 对象，可以增加一个判断语句检测返回的 html 是不是 None

If html is None:

```
    print( " URLnot Found " )
else:
    #程序继续
```

#### 40.什么是进程池和线程池

这两个问题有一定的相似度，在面向对象程序编程中，对象的创建与析构都是一个较为复杂的过程，较费时间，所以为了提高程序的运行效率尽可能减少创建和销毁对象的次数，特别是一些很耗资源的对象创建和销毁。

所以我们可以创建一个进程池（线程池），预先放一些进程（线程）进去，要用的时候就直接调用，用完之后再把进程归还给进程池，省下创建删除进程的时间，不过当然就需要额外的开销了

利用线程池与进程池可以是管理进程与线程的工作交给系统管理，不需要程序员

对立面的线程、进程进行管理

#### 41.解释负载均衡和集群

网络的负载均衡是一种动态均衡技术，常见的实现方式是通过一些工具实时地分析数据包，掌握网络中的数据流量状况，把任务合理均衡地分配出去。这种技术基于现有网络结构，提供了一种扩展服务器带宽和增加服务器吞吐量的廉价有效的方法，加强了网络数据处理能力，提高了网络的灵活性和可用性。日常生活中到处都能看到“负载均衡”，一个超市的收营员高峰期只能服务 10 位顾客，当做活动时 20 位顾客需要服务的话可能会排长队，这样购物体验将会很差（就像客户抱怨系统 / 网站访问太慢）。最简单的办法就是再招个营业员，重新开通一个出口。负载均衡的核心就是“分摊压力”。

所谓集群是指一组独立的计算机系统构成的一个松耦合的多处理器系统，它们之间通过网络实现进程间的通信。应用程序可以通过网络共享内存进行消息传送，实现分布式计算机。通俗一点来说，就是让若干台计算机联合起来工作（服务），可以是并行的，也可以是做备份。

1、以下 print 将输出什么，为什么

```
a = 1
def fun(a):
    a = 2
```

```

fun(a)
print a

a = []
def fun(a):
    a.append(1)
fun(a)
print a

```

对象有两种，可更改（mutable）与‘不可更改’（immutable）对象。在 python 中，strings, tuples, 和 numbers 是不可更改的对象，而 list,dict 等则是可以修改的对象。

第一个例子里函数把引用指向了一个不可变对象，当函数返回的时候，外面的引用没半毛感觉。

而第二个例子就不一样了，函数内的引用指向的是可变对象，对它的操作就和定位了指针地址一样，在内存里进行修改。

## 2、Python 中单下划线和双下划线分别代表什么

`__init__`: 一种约定，Python 内部的名字，用来区别其他用户自定义的命名，以防冲突。

`_foo`: 一种约定，用来指定变量私有。程序员用来指定私有变量的一种方式。

`__foo`: 这个有真正的意义：解析器用 `_classname__foo` 来代替这个名字，以区别和其他类相同的命名

## 3、什么是闭包

当一个内嵌函数引用其外部作用域的变量，我们就会得到一个闭包。总结一下，创建一个闭包必须满足以下几点：

必须有一个内嵌函数

内嵌函数必须引用外部函数中的变量

外部函数的返回值必须是内嵌函数

## 4、自己写一个闭包的实例

```
def lazy_sum(*args):
```

```
def sum():
    ax = 0
    for n in args:
        ax = ax + n
    return ax
return sum
```

## 5、三次握手

客户端通过向服务器端发送一个 SYN 来创建一个主动打开，作为三次握手的一部分。

客户端把这段连接的序号设定为随机数 A。

服务器端应当为一个合法的 SYN 回送一个 SYN/ACK。ACK 的确认码应为 A+1，SYN/ACK 包本身又有一个随机序号 B。

最后，客户端再发送一个 ACK。当服务端受到这个 ACK 的时候，就完成了三路握手，并进入了连接创建状态。此时包序号被设定为收到的确认号 A+1，而响应则为 B+1。

## 6、class Person:

```
    name="aaa"
```

```
p1=Person()
```

```
p2=Person()
```

```
p1.name="bbb"
```

```
print p1.name
```

```
print p2.name
```

```
print Person.name
```

p1.name="bbb" 是实例调用了类变量，这其实和上面第一个问题一样，就是函数传参的问题，p1.name 一开始是指向的类变量 name="aaa"，

但是在实例的作用域里把类变量的引用改变了，就变成了一个实例变量，self.name 不再引用 Person 的类变量 name 了。

## 7、GIL 线程全局锁

线程全局锁 (Global Interpreter Lock)，即 Python 为了保证线程安全而采取的独立线程运行的限制，说白了就是一个核只能在同一时间运行一个线程。

8、利用 filter 过滤出列表中大于 5 的数字

```
a = [1,2,3,4,5,6,7]
```

```
b = filter(lambda x: x > 5, a)
```

9、利用 map 让列表每项都乘以 2

```
a = map(lambda x:x*2,[1,2,3])
```

10、MyISAM 和 InnoDB

MyISAM 适合于一些需要大量查询的应用，但其对于有大量写操作并不是很好。甚至你只是需要 update 一个字段，整个表都会被锁起来，而别的进程，就算是读进程都无法操作直到读操作完成。另外，MyISAM 对于 SELECT COUNT(\*) 这类的计算是超快无比的。

InnoDB 的趋势会是一个非常复杂的存储引擎，对于一些小的应用，它会比 MyISAM 还慢。它是它支持“行锁”，于是在写操作比较多的时候，会更优秀。并且，他还支持更多的高级应用，比如：事务。

1.现在有一个 dict 对象 adict，里面包含了一百万个元素，查找其中的某个元素的平均需要多少次比较？一千万个元素呢？

2.下述两个对象有何区别？

```
a=,你好？
```

```
a=u,你好？
```

3.在 python 2 中，dict 对象的 items（）方法和 iteritems() 方法有什么区别？

4.分析下述代码的输出：

```
def foo(count):
```

```
    a,b = 0, 1
```

```
    while a < count:
```

```
        yield a
```

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

```
for i in foo(10):
```

```
    print i
```

5.分析下述代码的输出：

```
def change_int(a):
```

```
    a=10
```

```
def change_list(a):
```

```
    a[0]=10
```

```
m=2
```

```
n=[2]
```

```
change_int(m)
```

```
change_list(n)
```

6.分析下述代码的输出：

```
import re
```

```
s = '<html><head><title>Title</title></head></html>'
```

```
print re.match('<.*>', s).group()
```

```
print re.match('<.*?>', s).group()
```

7.尝试写一段代码，逐行读入一个文本文件（ ASCII 文件，不必考虑代码），并在屏幕上打印出来：

```
def print_file(file_name):
```

8.尝试写一段代码，爬取 HTTP 协议的网站，获取 HTTP 状态码，HTTP 消息头和网页内容：（使用熟悉的网络模块）

```
import
```

```
def crawl_webpage(url = "http://example.org/"):

```

9.尝试写一段代码，从 list 对象中删除重复的项，如输入 [1,2,3,3,4,2,3,4,5,6,1]，则输出为[1,2,3,4,5,6]，不必保持顺序。尽量使用标准库和内键类型以简化代码。

```
ids = [1,2,3,3,4,2,3,4,5,6,1]
```

```
news_ids = []
```

```
for id in ids:
```

```
    if id not in news_ids:
```

```
        news_ids.append(id)
```

```
print news_ids
```

```
ids = [1,4,3,3,4,2,3,4,5,6,1]
```

```
ids = list(set(ids))
```

## GIL 线程全局锁

线程全局锁 (Global Interpreter Lock), 即 Python 为了保证线程安全而采取的独立线程运行的限制,说白了就是一个核只能在同一时间运行一个线程。

## 协程

简单点说协程是进程和线程的升级版,进程和线程都面临着内核态和用户态的切换问题而耗费许多切换时间,而协程就是用户自己控制切换的时机,不再需要陷入系统的内核态。

Python 里最常见的 `yield` 就是协程的思想!

## 闭包

闭包 (closure) 是函数式编程的重要的语法结构。闭包也是一种组织代码的结构,它同样提高了代码的可重复使用性。

当一个内嵌函数引用其外部作用域的变量,我们就会得到一个闭包。总结一下,创建一个闭包必须满足以下几点:

必须有一个内嵌函数

内嵌函数必须引用外部函数中的变量

外部函数的返回值必须是内嵌函数

感觉闭包还是有难度的,几句话是说不明白的,还是查查相关资料。

重点是函数运行后并不会被撤销,就像 16 题的 `instance` 字典一样,当函数运行完后,`instance` 并不被销毁,而是继续留在内存空间里。这个功能类似类里的类变量,只不过迁移到了函数上。

闭包就像个空心球一样,你知道外面和里面,但你不知道中间是什么样。

python 中函数式编程支持:

filter 函数的功能相当于过滤器。 调用一个布尔函数 bool\_func 来迭代遍历每个 seq 中的元素；返回一个使 bool\_seq 返回值为 true 的元素的序列。

```
>>>a = [1,2,3,4,5,6,7]
>>>b = filter(lambda x: x > 5, a)
>>>print b
>>>[6,7]
```

map 函数是对一个序列的每个项依次执行函数，下面是对一个序列每个项都乘以 2：

```
>>> a = map(lambda x:x*2,[1,2,3])
>>> list(a)
[2, 4, 6]
```

reduce 函数是对一个序列的每个项迭代调用函数，下面是求 3 的阶乘：

```
>>> reduce(lambda x,y:x*y,range(1,4))
6
```

## Python 里的拷贝

### 引用和 copy(),deepcopy() 的区别

```
import copy
a = [1, 2, 3, 4, ['a', 'b']]    #原始对象

b = a    #赋值，传对象的引用
c = copy.copy(a)    #对象拷贝，浅拷贝
d = copy.deepcopy(a)    #对象拷贝，深拷贝

a.append(5)    #修改对象 a
a[4].append('c')    #修改对象 a 中的 ['a', 'b'] 数组对象

print 'a = ', a
print 'b = ', b
print 'c = ', c
print 'd = ', d
```

输出结果：



```
a = [1, 2, 3, 4, ['a', 'b', 'c'], 5]
b = [1, 2, 3, 4, ['a', 'b', 'c'], 5]
c = [1, 2, 3, 4, ['a', 'b', 'c']]
d = [1, 2, 3, 4, ['a', 'b']]
```

## Python 垃圾回收机制

Python GC 主要使用引用计数 ( reference counting ) 来跟踪和回收垃圾。在引用计数的基础上, 通过 “标记 - 清除” ( mark and sweep ) 解决容器对象可能产生的循环引用问题, 通过 “分代回收” ( generation collection ) 以空间换时间的方法提高垃圾回收效率。

### 1 引用计数

PyObject 是每个对象必有的内容, 其中 ob\_refcnt 就是做为引用计数。当一个对象有新的引用时, 它的 ob\_refcnt 就会增加, 当引用它的对象被删除, 它的 ob\_refcnt 就会减少. 引用计数为 0 时, 该对象生命就结束了。

优点:

简单

实时性

缺点:

维护引用计数消耗资源

循环引用

### 标记 - 清除机制

基本思路是先按需分配, 等到没有空闲内存的时候从寄存器和程序栈上的引用出发, 遍历以对象为节点、以引用为边构成的图, 把所有可以访问到的对象打上标记, 然后清扫一遍内存空间, 把所有没标记的对象释放。

### 分代技术

分代回收的整体思想是：将系统中的所有内存块根据其存活时间划分为不同的集合，每个集合就成为一个“代”，垃圾收集频率随着“代”的存活时间的增大而减小，存活时间通常利用经过几次垃圾回收来度量。

Python 默认定义了三代对象集合，索引数越大，对象存活时间越长。

举例：当某些内存块 M 经过了 3 次垃圾收集的清洗之后还存活时，我们就将内存块 M 划到一个集合 A 中去，而新分配的内存都划分到集合 B 中去。当垃圾收集开始工作时，大多数情况都只对集合 B 进行垃圾回收，而对集合 A 进行垃圾回收要隔相当长一段时间后才进行，这就使得垃圾收集机制需要处理的内存少了，效率自然就提高了。在这个过程中，集合 B 中的某些内存块由于存活时间长而会被转移到集合 A 中，当然，集合 A 中实际上也存在一些垃圾，这些垃圾的回收会因为这种分代的机制而被延迟。

Python 的 is

is 是对比地址, == 是对比值

HTTP 和 HTTPS

状态码 定义

1xx 报告 接收到请求，继续进程

2xx 成功 步骤成功接收，被理解，并被接受

3xx 重定向 为了完成请求，必须采取进一步措施

4xx 客户端出错 请求包括错的顺序或不能完成

5xx 服务器出错 服务器无法完成显然有效的请求

403: Forbidden 404: Not Found

Ajax

AJAX, Asynchronous JavaScript and XML (异步的 JavaScript 和 XML)，是与在不重新加载整个页面的情况下，与服务器交换数据并更新部分网页的技术

去除列表中的重复元素

用集合

```
list(set(l))
```

用字典

```
l1 = ['b','c','d','b','c','a','a']  
l2 = {}.fromkeys(l1).keys()  
print l2  
用字典并保持顺序
```

```
l1 = ['b','c','d','b','c','a','a']  
l2 = list(set(l1))  
l2.sort(key=l1.index)  
print l2  
列表推导式
```

```
l1 = ['b','c','d','b','c','a','a']  
l2 = []  
[l2.append(i) for i in l1 if not i in l2]
```

## Linux Shell 笔试题

1.How to enable debug mode in shell scripts? (B)  
(如何在 shell 脚本中启用调试模式 )

- A. set + x
- B. set - x
- C. set debug
- D. set --

注：set -x 与 set +x 指令

用于脚本调试。 set 是把它下面的命令打印到屏幕

set -x 是开启 set +x 是关闭 set -o 是查看 (xtrace) , set 去追中一段代码的显示情况。

执行 set -x 后，对整个脚本有效。

2.Display all lines that begin with the letter “ A from a file (B)

(显示所有行开头字母 “ A从文件中 )

- A. ls A\*

- B. Cat file|grep ^A
- C. Cat file|grep A\$
- D. More |grep A\*

3.What does this sign mean in the Shell script“ \$# **B)**(

(这个标志是什么意思在 Shell 脚本 ” \$# 添加到 Shell 的参数个数 )

- A. The return value of last command  
#最后一个命令的返回值
- B. The number of script arguments  
#脚本参数的个数
- C. All of the arguments values  
#所有参数值
- D. The script itself  
#脚本本身

4.Which does console get after run following script command :**B)**

(控制台获取在运行以下脚本命令后 )

echo -n Red && echo -n Go || echo -n Hat

- A. RedHat
- B. RedGo
- C. RedHatGo
- D. GoHat

5.Which of the following statements assign the name of the current directory to variable var? A C

(下列哪个陈述当前目录的名称分配给变量 var ? )

- A. var=`pwd`
- B. var=\${pwd}
- C. var=\$(pwd)
- D. var=@pwd

6.Which of the following answers state string s is of zero length?

- A. [, \$s? = “ ” ]
- B. [ “ \$eq ” ” ]
- C. [ ! -n “ \$s ” ]
- D. [ -z “ \$s ” ]

7.Given a log file as follows:(A)

\$cat a.log

1 5

2 6

3 7

4 8

What would the output of following command?

```
$awk ,{y=x+$0;x=$0};END{p rint y+NR}? a.log
```

A. 9

B. 11

C. 14

D. 30

## Web 笔试

1.Javascript的基本数据类型？

1. Number 数字类型
2. String 字符串类型
3. Boolean 布尔类型
4. Function 函数
5. Object 对象
6. Null
7. Undefined 没有定义类型

2.[ " 1 " , " 2 " , " 3 " ].map(parseInt)

[1, NaN, NaN]

3.Javascript中如何将 JSON 转成对象 /数组？怎么转回来？

JSON.stringify 把一个对象转换成 json 字符串，

JSON.parse 把一个 json 字符串解析成对象。

4.Javascript中的 var 的含义及作用，

在定义全局变量时加或不加 var 关键字没什么影响；但在定义局部变量时如果不加 var 关键字 javascript 解释程序会将其解释为全局变量。

5.Document.write 和 innerHTML 区别？

document.write 是直接将内容写入页面的内容流，会导致页面全部重绘，

innerHTML 将内容写入某个 DOM 节点，不会导致页面全部重绘

6.下面的输出结果？ (My object)

```
var name = " ThisWindow " ;
```

```

var object = {
    name : " Myobject  ",
    getNameFunc : function(){
        var that = this;
        return function(){
            Return that.name;
        }
    }
};
alert(object.getNameFunc());

```

1.请举例说明如下选择器：

- A.id 选择器    #idname{} 选择 id= " idname"的所有元素
- B.类选择器        .classname{} 选择 class= " classname"的所有元素
- C.标签选择器        p{} 选择所有 <p>标签的
- D.相邻选择器        h1 + p {margin-top:60px} 选择紧邻 h1 后面的 p 元素
- E.子选择器            h1 > p {} 选择 h1 的子元素 p
- F.后代选择器    h1 em {} h1 元素中的所有 em 元素
- G.通配符选择器    \* {} 所有的元素
- H.属性选择器    a[href] {color:red;}

2.display 如下值的含义？

- A.block 此元素为块级元素，此元素前后会带有换行符
- B.none 此元素不会被显示
- C.inline-block 行内块元素。
- D.list-item 此元素会作为列表显示

3.postion 如下值的含义？

- A.absolute 生成绝对定位元素，相对于 static 定位之外第一个父元素进行定位
- B.fixed 生成绝对定位的元素，相对于浏览器窗口进行定位。
- C.relative 生成相对定位的元素，相对其正常元素进行定位。
- D.static 默认值，没有定位。
- E.inherit 规定应该从父类继承 position 属性的值。

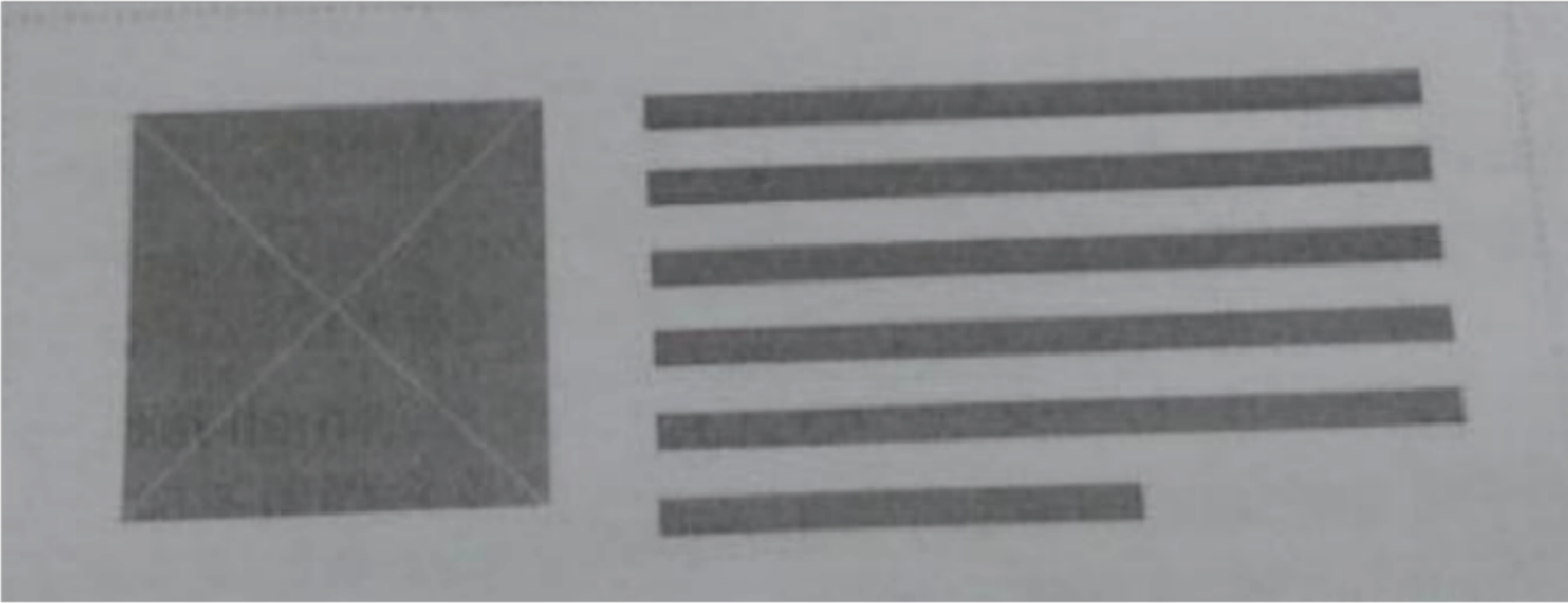
4.分别居中 div 和浮动元素

```

<div class= " divcss1 " ></div>
<div class= " divcss2 " ></div>
<style>
.divcss1 {

```

```
margin:0 auto;
}
.divcss2{
float: right;
}
</style>
5.使用 html 和 css写出下面的效果
```



HTML

1.HTML5 离线存储方式

Cache Manifest

基础

如需启用应用程序缓存，请在文档的 <html> 标签中包含 manifest 属性：

```
<!DOCTYPE HTML>

<html manifest="demo.appcache">

...

</html>
```

每个指定了 manifest 的页面在用户对其访问时都会被缓存。如果未指定 manifest 属性，则页面不会被缓存（除非在 manifest 文件中直接指定了该页面）。

manifest 文件的建议的文件扩展名是： ".appcache" 。

请注意， manifest 文件需要配置 正确的 MIME-type ，即 "text/cache-manifest" 。必须在 web 服务器上 进行配置。

## 2.Iframe 缺点

使用 ifame 框架的弊端是无法被搜索引擎所爬行抓取

## 3.[开放问题 ]写一段语义化的 HTML



