**Python语言特性**

**1 Python的函数参数传递**

看两个例子:

|  |  |
| --- | --- |
| 1  2  3  4  5 | a = 1  def fun(a):      a = 2  fun(a)  print a  # 1 |

|  |  |
| --- | --- |
| 1  2  3  4  5 | 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)**

这个非常的不常用,但是像ORM（对象关系映射（英语：(Object Relational Mapping，简称ORM，或O/RM，或O/R mapping），是一种程序技术，用于实现面向对象编程语言里不同类型系统的数据之间的转换）这种复杂的结构还是会需要的,详情请看：《[深刻理解Python中的元类(metaclass)](http://python.jobbole.com/21351/)》

**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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | def foo(x):      print "executing foo(%s)"%(x)    class A(object):      def foo(self,x): #实例方法          print "executing foo(%s,%s)"%(self,x)        @classmethod      def class\_foo(cls,x): #类方法          print "executing class\_foo(%s,%s)"%(cls,x)        @staticmethod #静态方法      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,对于一般的函数来说我们可以这么调用foo(x),这个函数就是最常用的,它的工作跟任何东西(类,实例)无关.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** |
| --- | --- | --- | --- |
| a = A() | a.foo(x) | a.class\_foo(x) | a.static\_foo(x) |
| A | 不可用 | A.class\_foo(x) | A.static\_foo(x) |

**4 类变量和实例变量**

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 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:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 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**

推导式是可以从一个数据序列构建另一个新的数据序列的结构体。 共有三种推导，在Python2和3中都有支持：

* 列表(list)推导式
* 字典(dict)推导式
* 集合(set)推导式

可能你见过列表推导式（又称解析式）,却没有见过字典推导式,在2.7中才加入的:

|  |  |
| --- | --- |
| 1 | d = {key: value for (key, value) in iterable} |

## 7 Python中单下划线和双下划线

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | >>> class MyClass():  ...     def \_\_init\_\_(self):  ...             self.\_\_superprivate = "Hello"  ...             self.\_semiprivate = ", world!"  ...  >>> mc = MyClass()  >>> print mc.\_\_superprivate  Traceback (most recent call last):    File "<stdin>", line 1, in <module>  AttributeError: myClass instance has no attribute '\_\_superprivate'  >>> print mc.\_semiprivate  , world!  >>> print mc.\_\_dict\_\_  {'\_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://stackoverflow.com/questions/1301346/the-meaning-of-a-single-and-a-double-underscore-before-an-object-name-in-python

## 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:

|  |  |
| --- | --- |
| 1 | "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:

|  |  |
| --- | --- |
| 1 | "hi there %s" % (name,)   # 提供一个单元素的数组而不是一个参数 |

但是有点丑..format就没有这些问题.你给的第二个问题也是这样,.format好看多了.

你为什么不用它?

* 不知道它(在读这个之前)
* 为了和Python2.5兼容(譬如logging库建议使用%([issue #4](https://github.com/taizilongxu/interview_python/issues/4)))

## 9 迭代器和生成器

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

这是中文版: http://taizilongxu.gitbooks.io/stackoverflow-about-python/content/1/README.html

#### Iterables

当你创建了一个列表,你可以一个一个的读取它的每一项,这叫做iteration:

>>> mylist = [1, 2, 3]

>>> for i in mylist:

... print(i)

1

2

3

Mylist是可迭代的.当你用列表推导式的时候,你就创建了一个列表,而这个列表也是可迭代的:

>>> mylist = [x\*x for x in range(3)]

>>> for i in mylist:

... print(i)

0

1

4

所有你可以用在for...in...语句中的都是可迭代的:比如lists,strings,files...因为这些可迭代的对象你可以随意的读取所以非常方便易用,但是你必须把它们的值放到内存里,当它们有很多值时就会消耗太多的内存.

#### Generators

生成器也是迭代器的一种,但是你**只能迭代它们一次**.原因很简单,因为它们不是全部存在内存里,它们只在要调用的时候在内存里生成:

>>> mygenerator = (x\*x for x in range(3))

>>> for i in mygenerator:

... print(i)

0

1

4

生成器和迭代器的区别就是用()代替[],还有你不能用for i in mygenerator第二次调用生成器:首先计算0,然后会在内存里丢掉0去计算1,直到计算完4.

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

相似的,**\*\*kwargs允许你使用没有事先定义的参数名**:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | >>> def table\_things(\*\*kwargs):  ...     for name, value in kwargs.items():  ...         print '{0} = {1}'.format(name, value)  ...  >>> table\_things(apple = 'fruit', cabbage = 'vegetable')  cabbage = vegetable  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**可以同时在函数的定义中**,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  2  3  4  5  6  7 | >>> def print\_three\_things(a, b, c):  ...     print 'a = {0}, b = {1}, c = {2}'.format(a,b,c)  ...  >>> mylist = ['aardvark', 'baboon', 'cat']  >>> print\_three\_things(\*mylist)    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,同学面阿里的时候就被问懵了…

面向切面编程（AOP是Aspect Oriented Program的首字母缩写） ，我们知道，面向对象的特点是继承、多态和封装。而封装就要求将功能分散到不同的对象中去，这在软件设计中往往称为职责分配。实际上也就是说，让不同的类设计不同的方法。这样代码就分散到一个个的类中去了。这样做的好处是降低了代码的复杂程度，使类可重用。  
但是人们也发现，在分散代码的同时，也增加了代码的重复性。什么意思呢？比如说，我们在两个类中，可能都需要在每个方法中做日志。按面向对象的设计方法，我们就必须在两个类的方法中都加入日志的内容。也许他们是完全相同的，但就是因为面向对象的设计让类与类之间无法联系，而不能将这些重复的代码统一起来。  
也许有人会说，那好办啊，我们可以将这段代码写在一个独立的类独立的方法里，然后再在这两个类中调用。但是，这样一来，这两个类跟我们上面提到的独立的类就有耦合了，它的改变会影响这两个类。那么，有没有什么办法，能让我们在需要的时候，随意地加入代码呢？这种在运行时，动态地将代码切入到类的指定方法、指定位置上的编程思想就是面向切面的编程。   
一般而言，我们管切入到指定类指定方法的代码片段称为切面，而切入到哪些类、哪些方法则叫切入点。有了AOP，我们就可以把几个类共有的代码，抽取到一个切片中，等到需要时再切入对象中去，从而改变其原有的行为。  
这样看来，AOP其实只是OOP的补充而已。OOP从横向上区分出一个个的类来，而AOP则从纵向上向对象中加入特定的代码。有了AOP，OOP变得立体了。如果加上时间维度，AOP使OOP由原来的二维变为三维了，由平面变成立体了。从技术上来说，AOP基本上是通过代理机制实现的。   
AOP在编程历史上可以说是里程碑式的，对OOP编程是一种十分有益的补充。

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

### Python中如何在一个函数中加入多个装饰器?

怎么做才能让一个函数同时用两个装饰器,像下面这样:

@makebold

@makeitalic

def say():

return "Hello"

我希望得到

<b><i>Hello</i></b>

我只是想知道装饰器怎么工作的!

去看看[文档](https://docs.python.org/2/reference/compound_stmts.html#function),答在下面:

def makebold(fn):

def wrapped():

return "<b>" + fn() + "</b>"

return wrapped

def makeitalic(fn):

def wrapped():

return "<i>" + fn() + "</i>"

return wrapped

@makebold

@makeitalic

def hello():

return "hello world"

print hello() ## returns <b><i>hello world</i></b>

已经知道函数就是对象.因此,对象:

* 可以赋值给一个变量
* 可以在其他函数里定义

这就意味着函数可以返回另一个函数.

装饰器就是[decorator design pattern](http://en.wikipedia.org/wiki/Decorator_pattern)的pythonic的变种.在Python中有许多经典的设计模式来满足开发者.

* 装饰器是Python2.4里引进的,所以确保你的Python解析器的版本>=2.4
* 装饰器使函数调用变慢了.一定要记住.
* **装饰器不能被取消**(有些人把装饰器做成可以移除的但是没有人会用)所以一旦一个函数被装饰了.所有的代码都会被装饰.
* 用装饰器装饰函数将会很难debug(在>=2.5版本将会有所改善;看下面)

##### 怎么使用装饰器?

**现在遇到了大问题**:我们用装饰器干什么?

看起来很黄很暴力,但是如果有实际用途就更好了.好了这里有1000个用途.传统的用法就是用它来为外部的库的函数(你不能修改的)做扩展,或者debug(你不想修改它,因为它是暂时的).

Python自身提供了几个装饰器,像property, staticmethod.

* Django用装饰器管理缓存和试图的权限.
* Twisted用来修改异步函数的调用.

## 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不像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.

[stackoverflow](http://stackoverflow.com/questions/54867/what-is-the-difference-between-old-style-and-new-style-classes-in-python)

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.

### Python中类分两种：旧式类和新式类：

➤新式类都从object继承，经典类不需要。

➤新式类的MRO(method resolution order 基类搜索顺序)算法采用C3算法广度优先搜索，而旧式类的MRO算法是采用深度优先搜索

➤新式类相同父类只执行一次构造函数，经典类重复执行多次。

### 其中：

➤截止到python2.1，只存在旧式类。旧式类中，类名和type是无关的：如果x是一个旧式类，那么x.\_\_class\_\_定义了x的类名，但是type(x)总是返回<type 'instance'>。这反映了所有的旧式类的实例是通过一个单一的叫做instance的内建类型来实现的，这是它和类不同的地方。

➤新式类是在python2.2为了统一类和实例引入的。一个新式类只能由用户自定义。如果x是一个新式类的实例，那么type(x)和x.\_\_class\_\_是一样的结果（尽管这不能得到保证，因为新式类的实例的\_\_class\_\_方法是允许被用户覆盖的）。

➤Python 2.x中默认都是经典类，只有显式继承了object才是新式类

➤Python 3.x中默认都是新式类,经典类被移除，不必显式的继承object

### Python2.x中：

**[python]** [view plain](http://blog.csdn.net/zimou5581/article/details/53053775) [copy](http://blog.csdn.net/zimou5581/article/details/53053775)

1. **class** A:
2. **pass**
3. **class** B:
4. **pass**
5. **class** C(B):
6. **pass**
7. **class** D(C,A):
8. **pass**

执行顺序为：D->C->B,->A

**[python]** [view plain](http://blog.csdn.net/zimou5581/article/details/53053775) [copy](http://blog.csdn.net/zimou5581/article/details/53053775)

1. **class** A(object):
2. **pass**
3. **class** B(object):
4. **pass**
5. **class** C(object):
6. **pass**
7. **class** D(A,B,C):
8. **pass**

执行顺序为： D->A->B->C->Object

## 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\_\_.

[stackoverflow](http://stackoverflow.com/questions/674304/pythons-use-of-new-and-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

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 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  2  3  4  5  6  7  8  9 | class Borg(object):      \_state = {}      def \_\_new\_\_(cls, \*args, \*\*kw):          ob = super(Borg, cls).\_\_new\_\_(cls, \*args, \*\*kw)          ob.\_\_dict\_\_ = cls.\_state          return ob    class MyClass2(Borg):      a = 1 |

### 3 装饰器版本

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | def singleton(cls, \*args, \*\*kw):      instances = {}      def getinstance():          if cls not in instances:              instances[cls] = cls(\*args, \*\*kw)          return instances[cls]      return getinstance    @singleton  class MyClass:    ... |

### 4 import方法

作为python的模块是天然的单例模式

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | # mysingleton.py  class My\_Singleton(object):      def foo(self):          pass    my\_singleton = My\_Singleton()    # to use  from mysingleton import my\_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.

见[Python 最难的问题](http://www.oschina.net/translate/pythons-hardest-problem)

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).

众所周知，python中有全局解释器锁，由于全局解释器锁的存在，所以在同一时间内，python解释器只能运行一个线程的代码，这大大影响了python多线程的性能。

在python，使用的都是操作系统级别的线程，因此在linux中 python使用的是pthread 在windows下也是使用的Windows的原生线程来实现。

python GIL 之所以会影响多线程等性能，是应为在多线程的情况下，只有当线程获得了一个全局锁的时候，那么该线程的代码才能运行，而全局锁只有一个，所以使用python多线程，在同一时刻也只有一个线程在运行，因此在即使在多核的情况下也只能发挥出单核的性能。

既然python在同一时刻下只能运行一个线程的代码，那线程之间是如何调度的呢？   
对于有io操作的线程，当一个线程在做io操作的时候，因为io操作不需要cpu，所以，这个时候，python会释放python全局锁，这样其他需要运行的线程就会使用该锁。   
对于cpu密集型的线程，比如一个线程可能一直需要使用cpu做计算，那么python中会有一个执行指令的计数器，当一个线程执行了一定数量的指令时，该线程就会停止执行并让出当前的锁，这样其他的线程就可以执行代码了。   
由上面可知，至少有两种情况python会做线程切换，一是一但有IO操作时，会有线程切换，二是当一个线程连续执行了一定数量的指令时，会出现线程切换。当然此处的线程切换不一定就一定会切换到其他线程执行，因为如果当前线程 优先级比较高的话，可能在让出锁以后，又继续获得锁，并优先执行。

linux下 python使用的是 pthread 和 有一个互斥量和一个 条件变量来实现全局锁功能。

## 19 协程

知乎被问到了,呵呵哒,跪了

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.

简单来说，编程中提到的 lambda 表达式，通常是在**需要一个函数，但是又不想费神去命名一个函数**的场合下使用，也就是指**匿名函数**。这一用法跟所谓 λ 演算（题目说明里的维基链接）的关系，有点像原子弹和质能方程的关系，差别其实还是挺大的。

不谈形式化的 λ 演算，只说有实际用途的匿名函数。先举一个普通的 Python 例子：将一个 list 里的每个元素都平方：

map( lambda x: x\*x, [y for y in range(10)] )

这个写法要好过

def sq(x):

return x \* x

map(sq, [y for y in range(10)])

，因为后者多定义了一个（污染环境的）函数，尤其如果这个函数只会使用一次的话。而且第一种写法实际上更易读，因为那个映射到列表上的函数具体是要做什么，非常一目了然。如果你仔细观察自己的代码，会发现这种场景其实很常见：你在某处就真的只需要一个能做一件事情的函数而已，连它叫什么名字都无关紧要。Lambda 表达式就可以用来做这件事。

进一步讲，匿名函数本质上就是一个函数，它所抽象出来的东西是一组运算。这是什么意思呢？类比

a = [1, 2, 3]

和

f = lambda x : x + 1

，你会发现，等号右边的东西完全可以脱离等号左边的东西而存在，等号左边的**名字**只是右边之**实体**的标识符。如果你能习惯 [1, 2, 3] 单独存在，那么 lambda x : x + 1 也能单独存在其实也就不难理解了，它的意义就是给「某个数加一」这一运算本身。

现在回头来看 map() 函数，它可以将一个函数映射到一个可枚举类型上面。沿用上面给出的 a 和 f，可以写：

map(f, a)

也就是将函数 f 依次套用在 a 的每一个元素上面，获得结果 [2, 3, 4]。现在用 lambda 表达式来替换 f，就变成：

map( lambda x : x + 1, [1, 2, 3] )

会不会觉得现在很一目了然了？尤其是类比

a = [1, 2, 3]

r = []

for each in a:

r.append(each+1)

这样的写法时，你会发现自己如果能将「遍历列表，给遇到的每个元素都做某种**运算**」的过程从一个循环里抽象出来成为一个函数 map，然后用 lambda 表达式将这种**运算**作为参数传给 map 的话，考虑事情的思维层级会高出一些来，需要顾及的细节也少了一点。Python 之中，类似能用到 lambda 表达式的「高级」函数还有 reduce、filter 等等，很多语言也都有这样的工具（不过这些特性最好不要在 Python 中用太多，原因详见 <http://www.zhihu.com/question/19794855/answer/12987428> 的评论部分）。这种能够接受一个函数作为参数的函数叫做「高阶函数」（higher-order function），是来自函数式编程（functional programming）的思想。

和其他很多语言相比，Python 的 lambda 限制多多，最严重的当属它只能由一条表达式组成。这个限制主要是为了防止滥用，因为当人们发觉 lambda 很方便，就比较容易滥用，可是用多了会让程序看起来不那么清晰，毕竟每个人对于抽象层级的忍耐 / 理解程度都有所不同。

## 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.

推荐: [酷壳](http://coolshell.cn/articles/10822.html)

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。

|  |  |
| --- | --- |
| 1  2  3  4 | >>>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：

|  |  |
| --- | --- |
| 1  2  3 | >>> 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：

|  |  |
| --- | --- |
| 1  2 | >>> reduce(lambda x,y:x\*y,range(1,4))  6 |

**23 Python里的拷贝**

引用和copy(),deepcopy()的区别

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | 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']] |

## 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的List

推荐: http://www.jianshu.com/p/J4U6rR

>>> L = []

>>> L.append(1)

>>> L.append(2)

>>> L.append(3)

>>> L

[1, 2, 3]

>>> for e in L:

... print e

...

1

2

3

list是可以迭代的。

在需要的时候扩容,但又不允许过度的浪费,适当的内存回收是非常必要的。 这个确定调整后的空间大小算法很有意思。 调整后大小 (new\_allocated) = 新元素数量 (newsize) + 预留空间 (new\_allocated) 调整后的空间肯定能存储 newsize 个元素。要关注的是预留空间的增长状况。 将预留算法改成 Python 版就更清楚了:(newsize // 8) + (newsize < 9 and 3 or 6)。 当 newsize >= allocated,自然按照这个新的长度 "扩容" 内存。 而如果 newsize < allocated,且利用率低于一半呢? allocated newsize new\_size + new\_allocated 10 4 4 + 3 20 9 9 + 7 很显然,这个新长度小于原来的已分配空间长度,自然会导致 realloc 收缩内存。(不容易啊) 引自《深入Python编程》

## 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

推荐：《[Python 2.7.x 和 3.x 版本的重要区别](http://python.jobbole.com/80006/)》

### 目录

* [使用\_\_future\_\_模块](http://python.jobbole.com/80006/#future)
* [print函数](http://python.jobbole.com/80006/#print)
* [整数除法](http://python.jobbole.com/80006/#division)
* [Unicode](http://python.jobbole.com/80006/#unicode)
* [xrange](http://python.jobbole.com/80006/#xrange)
* [触发异常](http://python.jobbole.com/80006/#raise)
* [处理异常](http://python.jobbole.com/80006/#handle)
* [next()函数和.next()方法](http://python.jobbole.com/80006/#next)
* [For循环变量与全局命名空间泄漏](http://python.jobbole.com/80006/#for)
* [比较无序类型](http://python.jobbole.com/80006/#compare)
* [使用input()解析输入内容](http://python.jobbole.com/80006/#input)
* [返回可迭代对象，而不是列表](http://python.jobbole.com/80006/#iterable)
* [更多关于Python 2和Python 3的文章](http://python.jobbole.com/80006/#more)

### \_\_future\_\_模块

Python 3.x引入了一些与Python 2不兼容的关键字和特性，在Python 2中，可以通过内置的\_\_future\_\_模块导入这些新内容。如果你希望在Python 2环境下写的代码也可以在Python 3.x中运行，那么建议使用\_\_future\_\_模块。例如，如果希望在Python 2中拥有Python 3.x的整数除法行为，可以通过下面的语句导入相应的模块。

|  |  |
| --- | --- |
| 1 | from \_\_future\_\_ import division |

下表列出了\_\_future\_\_中其他可导入的特性：

|  |  |  |  |
| --- | --- | --- | --- |
| 特性 | 可选版本 | 强制版本 | 效果 |
| nested\_scopes | 2.1.0b1 | 2.2 | [PEP 227](http://www.python.org/dev/peps/pep-0227): Statically Nested Scopes |
| generators | 2.2.0a1 | 2.3 | [PEP 255](http://www.python.org/dev/peps/pep-0255): Simple Generators |
| division | 2.2.0a2 | 3.0 | [PEP 238](http://www.python.org/dev/peps/pep-0238): Changing the Division Operator |
| absolute\_import | 2.5.0a1 | 3.0 | [PEP 328](http://www.python.org/dev/peps/pep-0328): Imports: Multi-Line and Absolute/Relative |
| with\_statement | 2.5.0a1 | 2.6 | [PEP 343](http://www.python.org/dev/peps/pep-0343): The “with” Statement |
| print\_function | 2.6.0a2 | 3.0 | [PEP 3105](http://www.python.org/dev/peps/pep-3105): Make print a function |
| unicode\_literals | 2.6.0a2 | 3.0 | [PEP 3112](http://www.python.org/dev/peps/pep-3112): Bytes literals in Python 3000 |

|  |  |
| --- | --- |
| 1 | from platform import python\_version |

### print函数

虽然print语法是Python 3中一个很小的改动，且应该已经广为人知，但依然值得提一下：Python 2中的print语句被Python 3中的print()函数取代，这意味着在Python 3中必须用括号将需要输出的对象括起来。

在Python 2中使用额外的括号也是可以的。但反过来在Python 3中想以Python2的形式不带括号调用print函数时，会触发SyntaxError。

Python 2

|  |  |
| --- | --- |
| 1  2  3  4 | print 'Python', python\_version()  print 'Hello, World!'  print('Hello, World!')  print "text", ; print 'print more text on the same line' |

|  |  |
| --- | --- |
| 1  2  3  4 | Python 2.7.6  Hello, World!  Hello, World!  text print more text on the same line |

Python 3

|  |  |
| --- | --- |
| 1  2  3  4  5 | print('Python', python\_version())  print('Hello, World!')    print("some text,", end="")  print(' print more text on the same line') |

|  |  |
| --- | --- |
| 1  2  3 | Python 3.4.1  Hello, World!  some text, print more text on the same line |

|  |  |
| --- | --- |
| 1 | print 'Hello, World!' |

|  |  |
| --- | --- |
| 1  2  3  4 | File "<ipython-input-3-139a7c5835bd>", line 1  print 'Hello, World!'  ^  SyntaxError: invalid syntax |

注意：

在Python中，带不带括号输出”Hello World”都很正常。但如果在圆括号中同时输出多个对象时，就会创建一个元组，这是因为**在Python 2中，print是一个语句，而不是函数调用**。

|  |  |
| --- | --- |
| 1  2  3 | print 'Python', python\_version()  print('a', 'b')  print 'a', 'b' |

|  |  |
| --- | --- |
| 1  2  3 | Python 2.7.7  ('a', 'b')  a b |

### 整数除法

由于人们常常会忽视Python 3在整数除法上的改动（写错了也不会触发Syntax Error），所以在移植代码或在Python 2中执行Python 3的代码时，需要特别注意这个改动。

所以，我还是会**在Python 3的脚本中尝试用float(3)/2或 3/2.0代替3/2**，以此来避免代码在Python 2环境下可能导致的错误（或与之相反，在Python 2脚本中用from \_\_future\_\_ import division来使用Python 3的除法）。

Python 2

|  |  |
| --- | --- |
| 1  2  3  4  5 | print 'Python', python\_version()  print '3 / 2 =', 3 / 2  print '3 // 2 =', 3 // 2  print '3 / 2.0 =', 3 / 2.0  print '3 // 2.0 =', 3 // 2.0 |

|  |  |
| --- | --- |
| 1  2  3  4  5 | Python 2.7.6  3 / 2 = 1 #Python2 整数除法是整除  3 // 2 = 1  3 / 2.0 = 1.5 #Python2的小数除法结果有小数  3 // 2.0 = 1.0 |

Python 3

|  |  |
| --- | --- |
| 1  2  3  4  5 | print('Python', python\_version())  print('3 / 2 =', 3 / 2)  print('3 // 2 =', 3 // 2)  print('3 / 2.0 =', 3 / 2.0)  print('3 // 2.0 =', 3 // 2.0) |

|  |  |
| --- | --- |
| 1  2  3  4  5 | Python 3.4.1  3 / 2 = 1.5  3 // 2 = 1  3 / 2.0 = 1.5  3 // 2.0 = 1.0 |

### Unicode

**Python 2**有基于**ASCII的str()类型**，其可通过单独的unicode()函数转成**unicode类型**，但**没有byte类型**。

而在**Python 3**中，终于有了**Unicode（utf-8）字符串**，以及两个字节类：**bytes和bytearrays**。

Python 2

|  |  |
| --- | --- |
| 1 | print 'Python', python\_version() |

|  |  |
| --- | --- |
| 1 | Python 2.7.6 |

|  |  |
| --- | --- |
| 1 | print type(unicode('this is like a python3 str type')) |

|  |  |
| --- | --- |
| 1 | <type 'unicode'> |

|  |  |
| --- | --- |
| 1 | print type(b'byte type does not exist') |

|  |  |
| --- | --- |
| 1 | <type 'str'> |

|  |  |
| --- | --- |
| 1 | print 'they are really' + b' the same' |

|  |  |
| --- | --- |
| 1 | they are really the same |

|  |  |
| --- | --- |
| 1 | print type(bytearray(b'bytearray oddly does exist though')) |

|  |  |
| --- | --- |
| 1 | <type 'bytearray'> |

Python 3

|  |  |
| --- | --- |
| 1  2 | print('Python', python\_version())  print('strings are now utf-8 u03BCnicou0394é!') |

|  |  |
| --- | --- |
| 1  2 | Python 3.4.1  strings are now utf-8 μnicoΔé! |

|  |  |
| --- | --- |
| 1  2 | print('Python', python\_version(), end="")  print(' has', type(b' bytes for storing data')) |

|  |  |  |
| --- | --- | --- |
| 1 | Python 3.4.1 has <class 'bytes'> | |
| 1  2 | print('and Python', python\_version(), end="")  print(' also has', type(bytearray(b'bytearrays'))) |

|  |  |
| --- | --- |
| 1 | and Python 3.4.1 also has <class 'bytearray'> |

|  |  |
| --- | --- |
| 1 | 'note that we cannot add a string' + b'bytes for data' |

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | ---------------------------------------------------------------------------  TypeError Traceback (most recent call last)  <ipython-input-13-d3e8942ccf81> in <module>()  ----> 1 'note that we cannot add a string' + b'bytes for data'    TypeError: Can't convert 'bytes' object to str implicitly |

### xrange

在**Python 2.x**中，经常会用**xrange()创建一个可迭代对象**，通常出现在“for循环”或“列表/集合/字典推导式”中。

这种行为与生成器非常相似（如”惰性求值“），但这里的xrange-iterable无尽的，意味着可能在这个xrange上无限迭代。

由于**xrange的“惰性求知“**特性，如果只需迭代一次（如for循环中），range()通常比xrange()快一些。不过**不建议在多次迭代中使用range()**，因为**range()每次都会在内存中重新生成一个列表**。

在Python 3中，range()的实现方式与xrange()函数相同，所以就不存在专用的xrange()（在**Python 3中使用xrange()会触发NameError**）。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | import timeit    n = 10000  def test\_range(n):      return for i in range(n):      pass    def test\_xrange(n):      for i in xrange(n):      pass |

Python 2

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | print 'Python', python\_version()    print 'ntiming range()'  %timeit test\_range(n)      print 'nntiming xrange()'  %timeit test\_xrange(n) |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | Python 2.7.6    timing range()  1000 loops, best of 3: 433 µs per loop    timing xrange()  1000 loops, best of 3: 350 µs per loop |

Python 3

|  |  |
| --- | --- |
| 1  2  3  4 | print('Python', python\_version())    print('ntiming range()')  %timeit test\_range(n) |

|  |  |
| --- | --- |
| 1  2  3  4 | Python 3.4.1    timing range()  1000 loops, best of 3: 520 µs per loop |

|  |  |
| --- | --- |
| 1 | print(xrange(10)) |

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | ---------------------------------------------------------------------------  NameError Traceback (most recent call last)  in ()  ----> 1 print(xrange(10))    NameError: name 'xrange' is not defined |

### Python 3中的range对象中的\_\_contains\_\_方法

另一个值得一提的是，在Python 3.x中，**range**有了一个新的\_\_contains\_\_方法。**\_\_contains\_\_方法可以有效的加快Python 3.x中整数和布尔型的“查找”速度**。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | x = 10000000  def val\_in\_range(x, val):      return val in range(x)    def val\_in\_xrange(x, val):      return val in xrange(x)    print('Python', python\_version())  assert(val\_in\_range(x, x/2) == True)  assert(val\_in\_range(x, x//2) == True)  %timeit val\_in\_range(x, x/2)    %timeit val\_in\_range(x, x//2) |

|  |  |
| --- | --- |
| 1  2  3 | Python 3.4.1  1 loops, best of 3: 742 ms per loop  1000000 loops, best of 3: 1.19 µs per loop |

根据上面的timeit的结果，查找整数比查找浮点数要快大约6万倍。但由于Python 2.x中的range或xrange没有\_\_contains\_\_方法，所以在Python 2中的整数和浮点数的查找速度差别不大。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | print 'Python', python\_version()    assert(val\_in\_xrange(x, x/2.0) == True)  assert(val\_in\_xrange(x, x/2) == True)  assert(val\_in\_range(x, x/2) == True)  assert(val\_in\_range(x, x//2) == True)  %timeit val\_in\_xrange(x, x/2.0)    %timeit val\_in\_xrange(x, x/2)    %timeit val\_in\_range(x, x/2.0)    %timeit val\_in\_range(x, x/2) |

|  |  |
| --- | --- |
| 1  2  3  4  5 | Python 2.7.7  1 loops, best of 3: 285 ms per loop  1 loops, best of 3: 179 ms per loop  1 loops, best of 3: 658 ms per loop  1 loops, best of 3: 556 ms per loop |

下面的代码证明了Python 2.x中没有\_\_contain\_\_方法：

|  |  |
| --- | --- |
| 1  2 | print('Python', python\_version())  range.\_\_contains\_\_ |

|  |  |
| --- | --- |
| 1  2 | Python 3.4.1  <slot wrapper '\_\_contains\_\_' of 'range' objects |

|  |  |
| --- | --- |
| 1  2 | print('Python', python\_version())  range.\_\_contains\_\_ |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Python 2.7.7  ---------------------------------------------------------------------------  AttributeError Traceback (most recent call last)  <ipython-input-7-05327350dafb> in <module>()  1 print 'Python', python\_version()  ----> 2 range.\_\_contains\_\_    AttributeError: 'builtin\_function\_or\_method' object has no attribute '\_\_contains\_\_' |

|  |  |
| --- | --- |
| 1  2 | print('Python', python\_version())  xrange.\_\_contains\_\_ |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | Python 2.7.7    ---------------------------------------------------------------------------  AttributeError Traceback (most recent call last)  in ()  1 print 'Python', python\_version()  ----> 2 xrange.\_\_contains\_\_    AttributeError: type object 'xrange' has no attribute '\_\_contains\_\_' |

关于Python 2中xrange()与Python 3中range()之间的速度差异的一点说明：

有读者指出了Python 3中的range()和Python 2中xrange()执行速度有差异。由于这两者的实现方式相同，因此理论上执行速度应该也是相同的。这里的速度差别仅仅是因为Python 3的总体速度就比Python 2慢。

|  |  |
| --- | --- |
| 1  2  3  4  5 | def test\_while():      i = 0      while i < 20000:          i += 1      return |

|  |  |
| --- | --- |
| 1  2 | print('Python', python\_version())  %timeit test\_while() |

|  |  |
| --- | --- |
| 1  2  3  4 | Python 3.4.1  %timeit test\_while()    100 loops, best of 3: 2.68 ms per loop |

|  |  |
| --- | --- |
| 1  2 | print 'Python', python\_version()  %timeit test\_while() |

|  |  |
| --- | --- |
| 1  2 | Python 2.7.6  1000 loops, best of 3: 1.72 ms per loop |

### 触发异常

**Python 2支持新旧两种异常触发语法，而Python 3只接受带括号的的语法（不然会触发SyntaxError）：**

Python 2

|  |  |
| --- | --- |
| 1 | print 'Python', python\_version() |

|  |  |
| --- | --- |
| 1 | Python 2.7.6 |

|  |  |
| --- | --- |
| 1 | raise IOError, "file error" |

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | ---------------------------------------------------------------------------  IOError Traceback (most recent call last)  <ipython-input-8-25f049caebb0> in <module>()  ----> 1 raise IOError, "file error"    IOError: file error |

|  |  |
| --- | --- |
| 1 | raise IOError("file error") |

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | ---------------------------------------------------------------------------  IOError Traceback (most recent call last)  <ipython-input-9-6f1c43f525b2> in <module>()  ----> 1 raise IOError("file error")    IOError: file error |

Python 3

|  |  |
| --- | --- |
| 1 | print('Python', python\_version()) |

|  |  |
| --- | --- |
| 1 | Python 3.4.1 |

|  |  |
| --- | --- |
| 1 | raise IOError, "file error" |

|  |  |
| --- | --- |
| 1  2  3  4  5 | File "<ipython-input-10-25f049caebb0>", line 1  raise IOError, "file error"  ^  SyntaxError: invalid syntax  The proper way to raise an exception in Python 3: |

|  |  |
| --- | --- |
| 1  2 | print('Python', python\_version())  raise IOError("file error") |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | Python 3.4.1    ---------------------------------------------------------------------------  OSError Traceback (most recent call last)  <ipython-input-11-c350544d15da> in <module>()  1 print('Python', python\_version())  ----> 2 raise IOError("file error")    OSError: file error |

### 异常处理

Python 3中的异常处理也发了一点变化。在Python 3中必须使用“as”关键字。

Python 2

|  |  |
| --- | --- |
| 1  2  3  4  5 | print 'Python', python\_version()  try:      let\_us\_cause\_a\_NameError  except NameError, err:      print err, '--> our error message' |

|  |  |
| --- | --- |
| 1  2 | Python 2.7.6  name 'let\_us\_cause\_a\_NameError' is not defined --> our error message |

Python 3

|  |  |
| --- | --- |
| 1  2  3  4  5 | print('Python', python\_version())  try:      let\_us\_cause\_a\_NameError  except NameError as err:      print(err, '--> our error message') |

|  |  |
| --- | --- |
| 1  2 | Python 3.4.1  name 'let\_us\_cause\_a\_NameError' is not defined --> our error message |

### next()函数和.next()方法

由于会经常用到next()（.next()）函数（方法），所以还要提到另一个语法改动（实现方面也做了改动）：在Python 2.7.5中，函数形式和方法形式都可以使用，而在**Python 3中，只能使用next()函数**（试图调用.next()方法会触发AttributeError）。

Python 2

|  |  |
| --- | --- |
| 1  2  3  4 | print 'Python', python\_version()  my\_generator = (letter for letter in 'abcdefg')  next(my\_generator)  my\_generator.next() |

|  |  |
| --- | --- |
| 1  2 | Python 2.7.6  'b' |

Python 3

|  |  |
| --- | --- |
| 1  2  3 | print('Python', python\_version())  my\_generator = (letter for letter in 'abcdefg')  next(my\_generator) |

|  |  |
| --- | --- |
| 1  2 | Python 3.4.1  'a' |

|  |  |
| --- | --- |
| 1 | my\_generator.next() |

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | ---------------------------------------------------------------------------  AttributeError Traceback (most recent call last)  <ipython-input-14-125f388bb61b> in <module>()  ----> 1 my\_generator.next()    AttributeError: 'generator' object has no attribute 'next' |

### For循环变量与全局命名空间泄漏

好消息是：在**Python 3.x中，for循环中的变量不再会泄漏到全局命名空间中了！**

这是Python 3.x中做的一个改动，在“What’s New In Python 3.0”中有如下描述：

“列表推导不再支持[… for var in item1, item2, …]这样的语法，使用[… for var in (item1, item2, …)]代替。还要注意列表推导有不同的语义：现在**列表推导更接近list()构造器中的生成器表达式这样的语法糖，特别要注意的是，循环控制变量不会再泄漏到循环周围的空间中了。”**

Python 2

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | print 'Python', python\_version()    i = 1  print 'before: i =', i    print 'comprehension: ', [i for i in range(5)]    print 'after: i =', i |

|  |  |
| --- | --- |
| 1  2  3  4 | Python 2.7.6  before: i = 1  comprehension: [0, 1, 2, 3, 4]  after: i = 4 |

Python 3

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | print('Python', python\_version())    i = 1  print('before: i =', i)    print('comprehension:', [i for i in range(5)])    print('after: i =', i) |

|  |  |
| --- | --- |
| 1  2  3  4 | Python 3.4.1  before: i = 1  comprehension: [0, 1, 2, 3, 4]  after: i = 1 |

### 比较无序类型

**Python 3中另一个优秀的改动是，如果我们试图比较无序类型，会触发一个TypeError。**

Python 2

|  |  |
| --- | --- |
| 1  2  3  4 | print 'Python', python\_version()  print "[1, 2] > 'foo' = ", [1, 2] > 'foo'  print "(1, 2) > 'foo' = ", (1, 2) > 'foo'  print "[1, 2] > (1, 2) = ", [1, 2] > (1, 2) |

|  |  |
| --- | --- |
| 1  2  3  4 | Python 2.7.6  [1, 2] > 'foo' = False  (1, 2) > 'foo' = True  [1, 2] > (1, 2) = False |

Python 3

|  |  |
| --- | --- |
| 1  2  3  4 | print('Python', python\_version())  print("[1, 2] > 'foo' = ", [1, 2] > 'foo')  print("(1, 2) > 'foo' = ", (1, 2) > 'foo')  print("[1, 2] > (1, 2) = ", [1, 2] > (1, 2)) |

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | Python 3.4.1  ---------------------------------------------------------------------------  TypeError Traceback (most recent call last)  <ipython-input-16-a9031729f4a0> in <module>()  1 print('Python', python\_version())  ----> 2 print("[1, 2] > 'foo' = ", [1, 2] > 'foo')  3 print("(1, 2) > 'foo' = ", (1, 2) > 'foo')  4 print("[1, 2] > (1, 2) = ", [1, 2] > (1, 2))  TypeError: unorderable types: list() > str() |

### 通过input()解析用户的输入

幸运的是，**Python 3改进了input()函数，这样该函数就会总是将用户的输入存储为str对象**。**在Python 2中，为了避免读取非字符串类型会发生的一些危险行为，不得不使用raw\_input()代替input()。**

Python 2

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | Python 2.7.6  [GCC 4.0.1 (Apple Inc. build 5493)] on darwin  Type "help", "copyright", "credits" or "license" for more information.    >>> my\_input = input('enter a number: ')    enter a number: 123    >>> type(my\_input)  <type 'int'>    >>> my\_input = raw\_input('enter a number: ')    enter a number: 123    >>> type(my\_input)  <type 'str'> |

Python 3

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | Python 3.4.1  [GCC 4.2.1 (Apple Inc. build 5577)] on darwin  Type "help", "copyright", "credits" or "license" for more information.    >>> my\_input = input('enter a number: ')  enter a number: 123  >>> type(my\_input)  <class 'str'> |

### 返回可迭代对象，而不是列表

在xrange一节中可以看到，某些函数和方法在Python中返回的是可迭代对象，而不像在Python 2中返回列表。

由于通常对这些对象只遍历一次，所以这种方式会节省很多内存。然而，如果通过生成器来多次迭代这些对象，效率就不高了。

此时我们的确需要列表对象，可以通过list()函数简单的将可迭代对象转成列表。

Python 2

|  |  |
| --- | --- |
| 1  2  3  4 | print 'Python', python\_version()    print range(3)  print type(range(3)) |

|  |  |
| --- | --- |
| 1  2  3 | Python 2.7.6  [0, 1, 2]  <type 'list'> |

Python 3

|  |  |
| --- | --- |
| 1  2  3  4 | print('Python', python\_version())  print(range(3))  print(type(range(3)))  print(list(range(3))) |

|  |  |
| --- | --- |
| 1  2  3  4 | Python 3.4.1  range(0, 3)  <class 'range'>  [0, 1, 2] |

下面列出了Python 3中其他不再返回列表的常用函数和方法：

* zip()
* map()
* filter()
* 字典的.key()方法
* 字典的.value()方法
* 字典的.item()方法

# 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.

[selec,poll和epoll区别总结](http://www.cnblogs.com/Anker/p/3265058.html)

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.

关于epoll的: http://www.cnblogs.com/my\_life/articles/3968782.html

## 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

推荐: http://www.ruanyifeng.com/blog/2014/11/compiler.html

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 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。  
链接的主要过程包块 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。

### 4 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)

## 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

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

[MySQL索引背后的数据结构及算法原理](http://blog.jobbole.com/24006/)

聚集索引,非聚集索引,B-Tree,B+Tree,最左前缀原理

## 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

## 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。他是它支持“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

## 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

[GET和POST有什么区别？及为什么网上的多数答案都是错的](http://www.cnblogs.com/nankezhishi/archive/2012/06/09/getandpost.html)

get: [RFC 2616 – Hypertext Transfer Protocol — HTTP/1.1](http://tools.ietf.org/html/rfc2616#section-9.3)  
post: [RFC 2616 – Hypertext Transfer Protocol — HTTP/1.1](http://tools.ietf.org/html/rfc2616#section-9.5)

## 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Bcrypy(好像)加密

## 9 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

HTTPS握手,对称加密,非对称加密,TLS/SSL,RSA

## 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 幂等 Idempotence

HTTP方法的幂等性是指一次和多次请求某一个资源应该具有同样的**副作用**。(注意是副作用)

GET http://www.bank.com/account/123456，不会改变资源的状态，不论调用一次还是N次都没有副作用。请注意，这里强调的是一次和N次具有相同的副作用，而不是每次GET的结果相同。GET http://www.news.com/latest-news这个HTTP请求可能会每次得到不同的结果，但它本身并没有产生任何副作用，因而是满足幂等性的。

DELETE方法用于删除资源，有副作用，但它应该满足幂等性。比如：DELETE http://www.forum.com/article/4231，调用一次和N次对系统产生的副作用是相同的，即删掉id为4231的帖子；因此，调用者可以多次调用或刷新页面而不必担心引起错误。

POST所对应的URI并非创建的资源本身，而是资源的接收者。比如：POST http://www.forum.com/articles的语义是在http://www.forum.com/articles下创建一篇帖子，HTTP响应中应包含帖子的创建状态以及帖子的URI。两次相同的POST请求会在服务器端创建两份资源，它们具有不同的URI；所以，POST方法不具备幂等性。

PUT所对应的URI是要创建或更新的资源本身。比如：PUT http://www.forum/articles/4231的语义是创建或更新ID为4231的帖子。对同一URI进行多次PUT的副作用和一次PUT是相同的；因此，PUT方法具有幂等性。

## 12 RESTful架构(SOAP,RPC)

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

## 13 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。

## 14 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

## 15 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。

官方说明：[PEP-3333](https://www.python.org/dev/peps/pep-3333/)

## 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）。  
推荐: http://www.kegel.com/c10k.html

## 18 socket

推荐: http://www.cnblogs.com/bingyun84/archive/2009/10/16/1584387.html

Socket=Ip address+ TCP/UDP + port

## 19 浏览器缓存

推荐: <http://web.jobbole.com/84367/>

304 Not Modified

## 20 HTTP1.0和HTTP1.1

推荐: http://blog.csdn.net/elifefly/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。

# \*NIX

## unix进程间通信方式(IPC)

1. 管道（Pipe）：管道可用于具有亲缘关系进程间的通信，允许一个进程和另一个与它有共同祖先的进程之间进行通信。
2. 命名管道（named pipe）：命名管道克服了管道没有名字的限制，因此，除具有管道所具有的功能外，它还允许无亲缘关系进程间的通信。命名管道在文件系统中有对应的文件名。命名管道通过命令mkfifo或系统调用mkfifo来创建。
3. 信号（Signal）：信号是比较复杂的通信方式，用于通知接受进程有某种事件发生，除了用于进程间通信外，进程还可以发送信号给进程本身；linux除了支持Unix早期信号语义函数sigal外，还支持语义符合Posix.1标准的信号函数sigaction（实际上，该函数是基于BSD的，BSD为了实现可靠信号机制，又能够统一对外接口，用sigaction函数重新实现了signal函数）。
4. 消息（Message）队列：消息队列是消息的链接表，包括Posix消息队列system V消息队列。有足够权限的进程可以向队列中添加消息，被赋予读权限的进程则可以读走队列中的消息。消息队列克服了信号承载信息量少，管道只能承载无格式字节流以及缓冲区大小受限等缺
5. 共享内存：使得多个进程可以访问同一块内存空间，是最快的可用IPC形式。是针对其他通信机制运行效率较低而设计的。往往与其它通信机制，如信号量结合使用，来达到进程间的同步及互斥。
6. 内存映射（mapped memory）：内存映射允许任何多个进程间通信，每一个使用该机制的进程通过把一个共享的文件映射到自己的进程地址空间来实现它。
7. 信号量（semaphore）：主要作为进程间以及同一进程不同线程之间的同步手段。
8. 套接口（Socket）：更为一般的进程间通信机制，可用于不同机器之间的进程间通信。起初是由Unix系统的BSD分支开发出来的，但现在一般可以移植到其它类Unix系统上：Linux和System V的变种都支持套接字。

# 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。

# 编程题

## 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。

Python



|  |  |
| --- | --- |
| 1 | fib = lambda n: n if n <= 2 else fib(n - 1) + fib(n - 2) |

第二种记忆方法

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | def memo(func):      cache = {}      def wrap(\*args):          if args not in cache:              cache[args] = func(\*args)          return cache[args]      return wrap      @ memo  def fib(i):      if i < 2:          return 1      return fib(i-1) + fib(i-2) |

第三种方法

Python



|  |  |
| --- | --- |
| 1  2  3  4  5 | def fib(n):      a, b = 0, 1      for \_ in xrange(n):          a, b = b, a + b      return b |

## 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。

Python



|  |  |
| --- | --- |
| 1 | fib = lambda n: n if n < 2 else 2 \* fib(n - 1) |

## 3 矩形覆盖

我们可以用2\*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\*1的小矩形无重叠地覆盖一个2\*n的大矩形，总共有多少种方法？

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

Python



|  |  |
| --- | --- |
| 1 | f = lambda n: 1 if n < 2 else f(n - 1) + f(n - 2) |

## 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

Python



|  |  |
| --- | --- |
| 1 | list(set(l)) |

用字典

Python



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

用字典并保持顺序

Python



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

列表推导式

Python



|  |  |
| --- | --- |
| 1  2  3 | l1 = ['b','c','d','b','c','a','a']  l2 = []  [l2.append(i) for i in l1 if not i in l2] |

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.

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | class ListNode:      def \_\_init\_\_(self, x):          self.val = x          self.next = None    class Solution:      # @param a ListNode      # @return a ListNode      def swapPairs(self, head):          if head != None and head.next != None:              next = head.next              head.next = self.swapPairs(next.next)              next.next = head              return next          return head |

## 7 创建字典的方法

### 1 直接创建

Python



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

### 2 工厂方法

Python



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

### 3 fromkeys()方法

Python



|  |  |
| --- | --- |
| 1  2  3  4 | dict1={}.fromkeys(('x','y'),-1)  dict={'x':-1,'y':-1}  dict2={}.fromkeys(('x','y'))  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

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | 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])              del l1[0]          else:              tmp.append(l2[0])              del l2[0]          return \_recursion\_merge\_sort2(l1, l2, tmp)    def recursion\_merge\_sort2(l1, l2):      return \_recursion\_merge\_sort2(l1, l2, []) |

循环算法

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | def loop\_merge\_sort(l1, l2):      tmp = []      while len(l1) > 0 and len(l2) > 0:          if l1[0] < l2[0]:              tmp.append(l1[0])              del l1[0]          else:              tmp.append(l2[0])              del l2[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.

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | class ListNode:      def \_\_init\_\_(self, x):          self.val = x          self.next = None  def node(l1, l2):      length1, lenth2 = 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 > lenth2:          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

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19 | def binarySearch(l, t):      low, high = 0, len(l) - 1      while low < high:          print low, high          mid = (low + high) / 2          if l[mid] > t:              high = mid          elif l[mid] < t:              low = mid + 1          else:              return mid      return low if l[low] == t else False    if \_\_name\_\_ == '\_\_main\_\_':      l = [1, 4, 12, 45, 66, 99, 120, 444]      print binarySearch(l, 12)      print binarySearch(l, 1)      print binarySearch(l, 13)      print binarySearch(l, 444) |

## 11 快排

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | def qsort(seq):      if seq==[]:          return []      else:          pivot=seq[0]          lesser=qsort([x for x in seq[1:] if x<pivot])          greater=qsort([x for x in seq[1:] if x>=pivot])          return lesser+[pivot]+greater    if \_\_name\_\_=='\_\_main\_\_':      seq=[5,6,78,9,0,-1,2,3,-65,12]      print(qsort(seq)) |

## 12 找零问题

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | 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      for cents in range(1, money+1):          minCoins = cents     #从第一个开始到money的所有情况初始          for value in values:              if value <= cents:                  temp = coinsUsed[cents - value] + 1                  if temp < 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:0 for i in range(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

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | ## 14 二叉树节点  class Node(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  def lookup(root):      stack = [root]      while stack:          current = stack.pop(0)          print current.data          if current.left:              stack.append(current.left)          if current.right:              stack.append(current.right)  ## 16 深度遍历  def deep(root):      if not root:          return      print root.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

Python



|  |  |
| --- | --- |
| 1  2  3  4 | def maxDepth(root):          if not root:              return 0          return max(maxDepth(root.left), maxDepth(root.right)) + 1 |

## 19 求两棵树是否相同

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | def isSameTree(p, q):      if p == None and q == None:          return True      elif p and q :          return p.val == q.val and isSameTree(p.left,q.left) and isSameTree(p.right,q.right)      else :          return False |

## 20 前序中序求后序

推荐: http://blog.csdn.net/hinyunsin/article/details/6315502

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | def rebuild(pre, center):      if not pre:          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:])      return cur    def deep(root):      if not root:          return      deep(root.left)      deep(root.right)      print root.data |

## 21 单链表逆置

Python



|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22 | class Node(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)))))))))    def rev(link):      pre = link      cur = link.next      pre.next = None      while cur:          tmp = cur.next          cur.next = pre          pre = cur          cur = tmp      return pre    root = rev(link)  while root:      print root.data      root = root.next |