写在之前（必读）

大家好哇，我是 Rocky0429。

对于 Python 的学习，很多小伙伴和我说不知道自己现在什么水平，也不知道现在自己算是到了哪个阶段。其实很简单，那句话怎么说来着，实践是检验真理的唯一标准。怎么实践呢？我感觉就是去面试。

很多同学可能一听面试，就慌忙摇头，觉得自己的水准还不到能面试的时候。可是哪有你想的那么复杂，面试无非就是去问你一些问题，你来回答。回答的上来就过，回答不上来就回去接着学。这就和我们从小到大在学校里，上课被老师提问一样，我们每个人都应该是面试强者才对。

当然了，去面试之前肯定也要准备啦，就和我们考试之前押题一样，所以呢，我结合自己的经验，整理了一些 Python 面试题，取名 **Python 面试之道**，从 Python 基础、Python 进阶、Python 后台开发到爬虫、机器学习，并且附带了详细的答案，无论是准备面试还是自己学习，这份面试题绝对值得你去看，去学习。

这本电子书我会一直更新，后续我看到的好的面试题就会整理到这里面，之后还会新增数据结构与算法、Mysql 数据库、操作系统、计算机网络等模块的内容，可能你手里拿到的不是这本电子书的最新版，为了保证实时性，所以建议你去获得最新版。微信搜索【**Python空间**】，回复【**Python面试之道**】，即可获取最新版的 PDF 版本，扫描下方的二维码可以直达：

再次提醒一下，关注以后，直接回复【**Python面试之道**】，即可获取最新版 PDF。

Python 基础（共 42 题）

1、什么是 Python?

Python 是一种编程语言，它有对象、模块、线程、异常处理和自动内存管理，可以加入其他语言的对比。

Python 是一种解释型语言，Python 在代码运行之前不需要解释。

Python 是动态类型语言，在声明变量时，不需要说明变量的类型。

Python 适合面向对象的编程，因为它支持通过组合与继承的方式定义类。

在 Python 语言中，函数是第一类对象。

Python 代码编写快，但是运行速度比编译型语言通常要慢。

Python 用途广泛，常被用走"胶水语言"，可帮助其他语言和组件改善运行状况。

使用 Python，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。

2、赋值、浅拷贝和深拷贝的区别?

(1) 赋值

在 Python 中，对象的赋值就是简单的对象引用，这点和 C++不同，如下所示：

```
a = [1,2,"hello",['python', 'C++']]
b = a
```

在上述情况下，a 和 b 是一样的，他们指向同一片内存，b 不过是 a 的别名，是引用。

我们可以使用 b is a 去判断，返回 True，表明他们地址相同，内容相同，也可以使用 id() 函数来查看两个列表的地址是否相同。

赋值操作(包括对象作为参数、返回值)不会开辟新的内存空间，它只是复制了对象的引用。也就是说除了 b 这个名字之外，没有其他的内存开销。修改了 a，也就影响了 b，同理，修改了 b，也就影响了 a。

(2) 浅拷贝

浅拷贝会创建新对象，其内容非原对象本身的引用，而是原对象内第一层对象的引用。

浅拷贝有三种形式:切片操作、工厂函数、copy 模块中的 copy 函数。

比如上述的列表 a，切片操作：b = a[:] 或者 b = [x for x in a]；

工厂函数：b = list(a)；

copy 函数：b = copy.copy(a)；

浅拷贝产生的列表 b 不再是列表 a 了，使用 is 判断可以发现他们不是同一个对象，使用 id 查看，他们也不指向同一片内存空间。但是当我们使用 id(x) for x in a 和 id(x) for x in b 来查看 a 和 b 中元素的地址时，可以看到二者包含的元素的地址是相同的。

在这种情况下，列表 a 和 b 是不同的对象，修改列表 b 理论上不会影响到列表 a。

但是要注意的是，浅拷贝之所以称之为浅拷贝，是它仅仅只拷贝了一层，在列表 a 中有一个嵌套的 list，如果我们修改了它，情况就不一样了。

比如：a[3].append('java')，查看列表 b，会发现列表 b 也发生了变化，这是因为，我们修改了嵌套的 list，修改外层元素，会修改它的引用，让它们指向别的位置，修改嵌套列表中的元素，列表的地址并未发生变化，指向的都是用一个位置。

(3) 深拷贝

深拷贝只有一种形式，copy 模块中的 deepcopy() 函数。

深拷贝和浅拷贝对应，深拷贝拷贝了对象的所有元素，包括多层嵌套的元素。因此，它的时间和空间开销要高。

同样的对列表 a，如果使用 b = copy.deepcopy(a)，再修改列表 b 将不会影响到列表 a，即使嵌套的列表具有更深的层次，也不会产生任何影响，因为深拷贝拷贝出来的对象根本就是一个全新的对象，不再与原来的对象有任何的关联。

(4) 注意点

对于非容器类型，如数字、字符，以及其他的“原子”类型，没有拷贝一说，产生的都是原对象的引用。

如果元组变量值包含原子类型对象，即使采用了深拷贝，也只能得到浅拷贝。

3、init 和new的区别?

当我们使用「类名()」创建对象的时候，Python 解释器会帮我们做两件事情：第一件是为对象在内存分配空间，第二件是为对象进行初始化。「分配空间」是**new** 方法，初始化是**init**方法。

new 方法在内部其实做了两件时期：第一件事是为「对象分配空间」，第二件事是「把对象的引用返回给 Python 解释器」。当 Python 的解释器拿到了对象的引用之后，就会把对象的引用传递给 **init** 的第一个参数 self，**init** 拿到对象的引用之后，就可以在方法的内部，针对对象来定义实例属性。

之所以要学习 **new** 方法，就是因为需要对分配空间的方法进行改造，改造的目的就是为了当使用「类名()」创建对象的时候，无论执行多少次，在内存中永远只会创造出一个对象的实例，这样就可以达到单例设计模式的目的。

4、Python 的变量、对象以及引用?

首先把结论抛出来：

- 变量是到内存空间的一个指针，也就是拥有指向对象连接的空间；
- 对象是一块内存，表示它们所代表的值；
- 引用就是自动形成的从变量到对象的指针。

以下是具体解释：

在 Python 中使用变量的时候不需要提前声明变量及其类型，变量还是会正常工作。在 Python 中，这个是以一种非常流畅的方式完成，下面以 a = 1 为例我们来看一下它到底是个什么情况。

首先是怎么知道创建了变量：对于变量 a，或者说是变量名 a，当程序第一次给它赋值的时候就创建了它，其实真实情况是 Python 在代码运行之前就先去检测变量名，我们不去具体深究这些，你只需要当作是「最开始的赋值创建了变量」。

再者是怎么知道变量是什么类型：其实这个很多人都没有搞清楚，「类型」这个概念不是存在于变量中，而是存在于对象中。变量本身就是通用的，它只是恰巧在某个时间点上引用了当时的特定对象而已。就比如说在表达式中，我们用的那个变量会立马被它当时所引用的特定对象所替代。

上面这个是动态语言明显区别于静态语言的地方，其实对于刚开始来说，如果你适应将「变量」和「对象」分开，动态类型你也就可以很容易理解了。

我们还是以 a = 1 为例，其实从上面的讲述中，我们很容易的可以发现对于 a = 1 这个赋值语句 Python 是如何去执行它的：创建一个代表值 1 的对象 --> 创建一个变量 a --> 将变量 a 和对象 1 连接。 下面我用一个图来更清晰的表示一下：

由上图我们可以看出，变量 a 其实变成了对象 1 的一个引用。如果你学过指针的话，你就会发现在内部「变量其实就是到对象内存空间的一个指针」。

同样还是上图，我们还可以看出在 Python 中「引用」是从变量到对象的连接，它就是一种关系，在内存中以指针的形式实现。

5、创建百万级实例如何节省内存?

可以定义类的 **slot** 属性，用它来声明实例属性的列表，可以用来减少内存空间的目的。

具体解释：

首先，我们先定义一个普通的 User 类：

```
class User1:
    def __init__(self, id, name, sex, status):
        self.id = id
        self.name = name
        self.sex = sex
        self.status = status
```

然后再定义一个带 **slot** 的类：

```
class User2:
    __slots__ = ['id', 'name', 'sex', 'status']
    def __init__(self, id, name, sex, status):
        self.id = id
        self.name = name
        self.sex = sex
        self.status = status
```

接下来创建两个类的实例：

```
u1 = User1('01', 'rocky', '男', 1)
u2 = User1('02', 'leey', '男', 1)
```

我们已经知道 u1 比 u2 使用的内存多，我们可以这样来想，一定是 u1 比 u2 多了某些属性，我们分别来看一下 u1 和 u2 的属性：

```
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__', 'id', 'name', 'sex', 'status']
```

```
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__slots__', '__str__', '__subclasshook__', 'id', 'name', 'sex', 'status']
```

乍一看好像差别不大，我们下面具体来看一下差别在哪：

```
set(dir(u1)) - set(dir(u2))
```

通过做集合的差集，我们得到 u1 和 u2 在属性上的具体差别：

```
{'__weakref__', '__dict__'}
```

在我们不使用弱引用的时候，**weakref** 并不占用多少内存，那最终这个锅就要 **dict** 来背了。

下面我们来看一下 **dict**:

```
u1.__dict__
```

输出结果如下所示：

```
{'id': '01', 'name': 'rocky', 'sex': '男', 'status': 1}
```

输出一个字典，在它内部我们发现了刚刚在类里定义的属性，这个字典就是为了实例动态绑定属性的一个字典，我们怎么动态绑定呢？比如我们现在没有 u1.level 这个属性，那么我们可以为它动态绑定一个 level 属性，比如 u1.level = 10，然后我们再来考察这个字典：

```
u1.__dict__
```

现在输出的结果为：

```
{'id': '01', 'name': 'rocky', 'sex': '男', 'status': 1, 'level': 10}
```

这样看到 level 进入到这个字典中。

这样一个动态绑定属性的特性，其实是以牺牲内存为代价的，因为这个 **dict** 它本身是占用内存的，接下来我们来验证这件事情：

```
import sys
sys.getsizeof(u1.__dict__)
```

我们用 sys 模块下的 getsizeof 方法，它可以得到一个对象使用的内存：

```
112
```

我们可以看到这个字典占用了 112 的字节。反观 u2，它没有了 **dict** 这个属性，我们想给它添加一个属性，也是被拒绝的。

```
u2.level = 10
```

显示的结果如下所示：

```
AttributeError: 'User2' object has no attribute 'level'
```

6、Python 里面如何生成随机数?

在 Python 中用于生成随机数的模块是 random，在使用前需要 import. 如下例子可以酌情列举：

random.random()：生成一个 0-1 之间的随机浮点数

random.uniform(a, b)：生成[a,b]之间的浮点数

random.randint(a, b)：生成[a,b]之间的整数

random.randrange(a, b, step)：在指定的集合[a,b)中，以 step 为基数随机取一个数

random.choice(sequence)：从特定序列中随机取一个元素，这里的序列可以是字符串，列表，元组等。

7、Python 是强语言类型还是弱语言类型?

Python 是强类型的动态脚本语言。

> 强类型：不允许不同类型相加。
> 动态：不使用显示数据类型声明，且确定一个变量的类型是在第一次给它赋值的时候。
> 脚本语言：一般也是解释型语言，运行代码只需要一个解释器，不需要编译。

8、谈一下什么是解释性语言，什么是编译性语言?

计算机不能直接理解高级语言，只能直接理解机器语言，所以必须要把高级语言翻译成机器语言，计算机才能执行高级语言编写的程序。

解释性语言在运行程序的时候才会进行翻译。

编译型语言写的程序在执行之前，需要一个专门的编译过程，把程序编译成机器语言（可执行文件）。

9、Python 中有日志吗?怎么使用?

Python 中有日志，Python 自带 logging 模块，调用 logging.basicConfig()方法，配置需要的日志等级和相应的参数，Python 解释器会按照配置的参数生成相应的日志。

补充知识：

Python 的标准日志模块

Python 标准库中提供了 logging 模块供我们使用。在最简单的使用中，默认情况下 logging 将日志打印到屏幕终端，我们可以直接导入 logging 模块，然后调用 debug，info，warn，error 和 critical 等函数来记录日志，默认日志的级别为 warning，级别比 warning 高的日志才会被显示（critical > error > warning > info > debug），「级别」是一个逻辑上的概念，用来区分日志的重要程度。

```
import logging

logging.debug('debug message')
logging.info("info message")
logging.warn('warn message')
logging.error("error message")
logging.critical('critical message')
```

上述代码的执行结果如下所示：

```
WARNING:root:warn message
ERROR:root:error message
CRITICAL:root:critical message
```

我在上面说过，用 print 的话会产生大量的信息，从而很难从中找到真正有用的信息。而 logging 中将日志分成不同的级别以后，我们在大多数时间只保存级别比较高的日志信息，从而提高了日志的性能和分析速度，这样我们就可以很快速的从一个很大的日志文件里找到错误的信息。

配置日志格式

我们在用 logging 来记录日志之前，先来进行一些简单的配置：

```
import logging

logging.basicConfig(filename= 'test.log', level= logging.INFO)

logging.debug('debug message')
logging.info("info message")
logging.warn('warn message')
logging.error("error message")
logging.critical('critical message')
```

运行上面的代码以后，会在当前的目录下新建一个 test.log 的文件，这个文件中存储 info 以及 info 以上级别的日志记录。运行一次的结果如下所示：

```
INFO:root:info message
WARNING:root:warn message
ERROR:root:error message
CRITICAL:root:critical message
```

上面的例子中，我是用 basicConfig 对日志进行了简单的配置，其实我们还可以进行更为复杂些的配置，在此之前，我们先来了解一下 logging 中的几个概念：

```
Logger：日志记录器，是应用程序中可以直接使用的接口。
Handler：日志处理器，用以表明将日志保存到什么地方以及保存多久。
Formatter：格式化，用以配置日志的输出格式。
```

上述三者的关系是：一个 Logger 使用一个 Handler，一个 Handler 使用一个 Formatter。那么概念我们知道了，该如何去使用它们呢？我们的 logging 中有很多种方式来配置文件，简单的就用上面所说的 basicConfig，对于比较复杂的我们可以将日志的配置保存在一个配置文件中，然后在主程序中使用 fileConfig 读取配置文件。

基本的知识我们知道了，下面我们来做一个小的题目：日志文件保存所有 debug 及其以上级别的日志，每条日志中要有打印日志的时间，日志的级别和日志的内容。请先自己尝试着思考一下，如果你已经思考完毕请继续向下看：

```
import logging

logging.basicConfig(
   level= logging.DEBUG,
   format = '%(asctime)s : %(levelname)s : %(message)s',
   filename= "test.log"
)

logging.debug('debug message')
logging.info("info message")
logging.warn('warn message')
logging.error("error message")
logging.critical('critical message')
```

上述代码的一次运行结果如下：

```
2018-10-19 22:50:35,225 : DEBUG : debug message
2018-10-19 22:50:35,225 : INFO : info message
2018-10-19 22:50:35,225 : WARNING : warn message
2018-10-19 22:50:35,225 : ERROR : error message
2018-10-19 22:50:35,225 : CRITICAL : critical message
```

我刚刚在上面说过，对于比较复杂的我们可以将日志的配置保存在一个配置文件中，然后在主程序中使用 fileConfig 读取配置文件。下面我们就来看一个典型的日志配置文件（配置文件名为 logging.conf）：

```
[loggers]
keys = root

[handlers]
keys = logfile

[formatters]
keys = generic

[logger_root]
handlers = logfile

[handler_logfile]
class = handlers.TimedRotatingFileHandler
args = ('test.log', 'midnight', 1, 10)
level = DEBUG
formatter = generic

[formatter_generic]
format = %(asctime)s %(levelname)-5.5s [%(name)s:%(lineno)s] %(message)s
```

在上述的日志配置文件中，首先我们在 [loggers] 中声明了一个叫做 root 的日志记录器（logger），在 [handlers] 中声明了一个叫 logfile 的日志处理器（handler），在 [formatters] 中声明了一个名为 generic 的格式化（formatter）。之后在 [logger_root] 中定义 root 这个日志处理器（logger） 所使用的日志处理器（handler） 是哪个，在 [handler_logfile] 中定义了日志处理器（handler） 输出日志的方式、日志文件的切换时间等。最后在 [formatter_generic] 中定义了日志的格式，包括日志的产生时间，级别、文件名以及行号等信息。

有了上述的配置文件以后，我们就可以在主代码中使用 logging.conf 模块的 fileConfig 函数加载日志配置：

```
import logging
import logging.config

logging.config.fileConfig('logging.conf')

logging.debug('debug message')
logging.info("info message")
logging.warn('warn message')
logging.error("error message")
logging.critical('critical message')
```

上述代码的运行一次的结果如下所示：

```
2018-10-19 23:00:02,809 WARNI [root:8] warn message
2018-10-19 23:00:02,809 ERROR [root:9] error message
2018-10-19 23:00:02,809 CRITI [root:10] critical message
```

10、Python 是如何进行类型转换的?

内建函数封装了各种转换函数，可以使用目标类型关键字强制类型转换，进制之间的转换可以用 int('str'，base='n')将特定进制的字符串转换为十进制，再用相应的进制转换函数将十进制转换为目标进制。

可以使用内置函数直接转换的有：

```
list---->tuple tuple(list)
tuple---->list list(tuple)
```

11、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。当 Python 遇到一个变量的话它会按照这的顺序进行搜索：

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)。

12、什么是 Python 自省?

Python 自省是 Python 具有的一种能力，使程序员面向对象的语言所写的程序在运行时,能够获得对象的类 Python 型。

Python 是一种解释型语言，为程序员提供了极大的灵活性和控制力。

13、什么是 Python 的命名空间?

命名空间，又名 namesapce，是在很多的编程语言中都会出现的术语，趁着这个题顺便给大家仔细介绍一下。

全局变量 & 局部变量

全局变量和局部变量是我们理解命名空间的开始，我们先来看一段代码：

```
x = 2
def func():
   x = 3
   print('func x ---> ',x)

func()
print('out of func x ---> ',x)
```

这段代码输出的结果如下：

```
func x ---> 3
out of func x ---> 2
```

从上述的结果中可以看出，运行 func()，输出的是 func() 里面的变量 x 所引用的对象 3，之后执行的是代码中的最后一行。这里要区分清楚，前一个 x 输出的是函数内部的变量 x，后一个 x 输出的是函数外的变量 x，两个变量互相不影响，在各自的作用域中起作用。

那个只在函数内起作用的变量就叫 “局部变量”，有了 “局部” 就有相应的 “全部”，但是后者听起来有歧义，所以就叫了 “全局”。

```
x = 2
def func():
   global x = 3 #注意此处
   print('func x ---> ',x)

func()
print('out of func x ---> ',x)
```

这段代码中比上段代码多加了一个 global x，这句话的意思是在声明 x 是全局变量，通俗点说就是这个 x 和 函数外的 x 是同一个了，所以结果就成了下面这样：

```
func x ---> 3
out of func x ---> 3
```

这样乍一看好像全局变量好强，可以管着函数内外，但是我们还是要注意，全局变量还是谨慎使用的好，因为毕竟内外有别，不要带来混乱。

作用域

作用域，用比较直白的方式来说，就是程序中变量与对象存在关联的那段程序，比如我在上面说的， x = 2 和 x = 3 是在两个不同的作用域中。

通常的，作用域是被分为静态作用域和动态作用域，虽然我们说 Python 是动态语言，但是它的作用域属于静态作用域，即 Python 中的变量的作用域是由该变量所在程序中的位置所决定的。

在 Python 中作用域被划分成四个层级，分别是：local（局部作用域），enclosing（嵌套作用域），global（全局作用域）和 built - in（内建作用域）。对于一个变量，Python 也是按照之前四个层级依次在不用的作用域中查找，我们在上一段代码中，对于变量 x，首先搜索的是函数体内的局部作用域，然后是函数体外的全局作用域，至于这段话具体怎么来理解，请看下面的例子：

```
def out_func():
   x = 2
   def in_func():
       x = 3
       print('in_func x ---> ',x)
   in_func()
   print('out_func x ---> ',x)

x = 4
out_func()
print('x == ',x)
```

上述代码运行的结果是：

```
in_func x ---> 3
out_func x ---> 2
x == 4
```

仔细观察一下上面的代码和运行的结果，你就会发现变量在不同的范围内进行搜索的规律，是不是感觉这些都是以前被你忽略的呢？

命名空间

《维基百科》中说 “命名空间是对作用域的一种特殊的抽象”，在这里我用一个比方来具体说明一下：

比如张三在公司 A，他的工号是 111，李四在公司 B，他的工号也是 111，因为两个人在不同的公司，他们俩的工号可以相同但是不会引起混乱，这里的公司就表示一个独立的命名空间，如果两个人在一个公司的话，他们的工号就不能相同，否则光看工号也不知道到底是谁。

其实上面举的这个例子的特点就是我们使用命名空间的理由，在大型的计算机程序中，往往会出现成百上千的标识符，命名空间提供隐藏区域标识符的机制。通过将逻辑上相关的标识符构成响应的命名空间，可以使整个系统更加的模块化。

我在开头引用的《维基百科》的那句话说 “命名空间是对作用域的一种特殊的抽象”，它其实包含了处于该作用域内的标识符，且它本身也用一个标识符来表示。在 Python 中，命名空间本身的标识符也属于更外层的一个命名空间，所以命名空间也是可以嵌套的，它们共同生活在 “全局命名空间” 下。

简言之，不同的命名空间可以同时存在，但是彼此独立，互不干扰。当然了，命名空间因为其对象的不同也有所区别，可以分为以下几种：

- 1.本地命名空间：模块中有函数或者类的时候，每个函数或者类所定义的命名空间即是本地命名空间，当函数返回结果或者抛出异常的时候，本地命名空间也就结束了。
- 2.全局命名空间：每个模块创建了自己所拥有的全局命名空间，不同模块的全局命名空间彼此独立，不同模块中相同名称的命名空间也会因为模块的不同而不相互干扰。
- 3.内置命名空间：当 Python 运行起来的时候，它们就存在了，内置函数的命名空间都属于内置命名空间，所以我们可以在任何程序中直接运行它们。

程序查询命名空间的时候也有一套顺序，依次按照本地命名空间 ，全局命名空间，内置命名空间。

```
def fun(like):
   name = 'rocky'
   print(locals())

fun('python')
```

访问本地命名空间使用 locals 完成，我们来看一下结果：

```
{'name': 'rocky', 'like': 'python'}
```

从上面的结果中可以看出，命名空间中的数据存储的结构和字典是一样的。可能你已经猜到了，当我们要访问全局命名空间的时候，可以使用 globals。

关于命名空间还有一个生命周期的问题，就是一个命名空间什么时候出现，什么时候消失，这个很好理解，就是哪部分被读入内存，哪部分的命名空间就存在了，比如我们在上面说的，Python 启动，内置命名空间就建立。

14、你所遵循的代码规范是什么?

PEP 8 编码风格

Python 代码从第一眼看上去，给人的感觉就是简洁优美，可读性强，也就是我们日常所说的「高颜值」。一方面是因为 Python 自身的优秀设计，比如统一的锁进，没有多余的符号从而让代码变的更加简洁；另一方面就是因为它有着一套较为统一的编码风格，当然它本身只是编码风格方面的建议而不是强制，相应的在编写 Python 代码的编辑器自动提供 PFP 8 检查，当你编写的代码违反了 PEP 8 规范的时候，会给出警告信息和修正的建议。与此同时，还有专门的检查工具对 Python 的代码风格进行检查。

由上，还是建议在编写 Python 代码的时候都遵循 PEP 8 编码规范，毕竟你以后不可能是只一个人写代码，未来不论是在公司或者某些开源项目中，作为其中的一份子，肯定还是要在风格上向大众看齐的。

PEP 8 编码规范详细的给出了 Python 编码的指导，包括什么对齐啦，包的导入顺序啦，空格和注释啦还有命名习惯等方方面面，并且还有详细的事例。

下面我以「包」的导入为例，看一下 PEP 8 给出的具体编程指导。在 Python 中， import 应该一次只导入一个模块，不同的模块应该独立一行：

```
import pandas
import numpy
```

反面例子：

```
import pandas,numpy
```

如果想要从一个模块里面导入多个，也可以像下面这样：

```
from subprocess import Popen, PIPE
```

import 语句应该处于源码文件的顶部，位于模块注释和文档字符串之后，全局变量和常量之前。在导入不同的库的时候，应该按照以下的顺序分组，各个分组之间以空行分隔：

- 导入标准库模块
- 导入相关第三方库模块
- 导入当前应用程序/库模块

具体事例如下所示：

```
import os
import time

import psutil

from test import u_test,my_test
```

Python 中还支持相对导入和绝对导入，在这里还是强推绝对导入。因为绝对导入的可读性更好一些，也不容易出错，即使出错了也会给出更加详细的错误信息。具体如下所示：

```
from sub_package import tools
from sub_package.tools  import msg
```

当然除了上述以外还有更多对于包的规范的描述，PEP 8 的编码风格指导比较长，并且写的非常详细，所以我就不在这一一介绍了，详细的可以参考 Python 官网上的资料。

pycodestyle 检查代码规范

我在上面说过 PEP 8 只是官方给出的 Python 编码规范，并没有强制要求大家都遵守，但是又由于大家都在用，所以它也就变成了事实上的 Python 代码风格标准，既然都是标准了，那么就应该有工具来检查这个标准，这样可以帮助 Python 小白规范自己的代码，也可以帮助大家在开源或者工作中形成统一的代码风格。

为了达成上述的目的，官方提供了同名的命令行工具来检查 Python 代码是否违反了 PEP 8 规范，并且对违反规范的地方给出了相应的提示信息。

```
pip install pep8
```

规范的名字是 PEP 8，这个检查代码风格的命令行工具叫 pep8，这个很容易引起大家的困惑，因此 Python 之父建议将 pep8 重新命名为 pycodestyle，下面我们来看一下 pycodestyle 的用法。

首先通过 pip 安装一下：

```
pip install pycodestyle
```

对一个或者多个文件运行 pycodestyle，打印检查报告：

```
liyans-MacBook-Pro:venv rocky$ pycodestyle --first test_search.py
test_search.py:10:37: W292 no newline at end of file
```

通过 --show-source 显示不符合规范的源码，以便程序员进行修改，具体如下所示：

```
liyans-MacBook-Pro:venv rocky$ pycodestyle --show-source --show-pep8 test_search
.py
test_search.py:10:37: W292 no newline at end of file
logging.critical('critical message')
                                    ^
    Trailing blank lines are superfluous.

    Okay: spam(1)
    W391: spam(1)\n
```


autopep8 格式化代码

autopep8 能够将 Python 代码自动格式化为 PEP 8 风格，它使用 pycodestyle 工具来决定代码中的哪部分需要被格式化，这能够修复大部分 pycodestyle 工具中报告的排版问题。autopep8 本身也是一个用 Python 写的工具，所以我们还是可以用 pip 直接安装：

```
pip install autopep8
```

它的使用方式也很简单，具体如下所示：

```
autopep8 --in-place test_search.py
```

上述代码如果不带 --in-place 的话，会将 autopep8 格式化以后的代码直接输出到控制台。我们可以用这种方式检查 autopep8 的修改，使用 --in-place 则会直接将结果保存到源文件中。在这我继续用上面的例子中用到的 py 文件，具体如下所示：

```
liyans-MacBook-Pro:venv rocky$ autopep8 test_search.py
import logging
import logging.config

logging.config.fileConfig('logging.conf')

logging.debug('debug message')
logging.info("info message")
logging.warn('warn message')
logging.error("error message")
logging.critical('critical message')
```


上面的例子中，autopep8 顺利的修复了所有的问题，但是如果你这个时候查看源文件的话，你会发现源文件的内容还是和原来一样，并没有被修改。这个时候我们就要用到 --in-place，加上这个选项将不会有任何输出， autopep8 会直接修改源文件。

```
autopep8 --in-place test_search.py
```

15、关于 Python 程序的运行方面，有什么手段能提升性能?

1、使用多进程，充分利用机器的多核性能

2、对于性能影响较大的部分代码，可以使用 C 或 C++ 编写

3、对于 IO 阻塞造成的性能影响，可以使用 IO 多路复用来解决

4、尽量使用 Python 的内建函数

5、尽量使用局部变量

16、dict 的 items() 方法与 iteritems() 方法的不同?

items方法将所有的字典以列表方式返回，其中项在返回时没有特殊的顺序

iteritems方法有相似的作用，但是返回一个迭代器对象

17、os.path和sys.path的区别?

os.path是module，包含了各种处理长文件名(路径名)的函数。

sys.path是由目录名构成的列表，Python 从中查找扩展模块(Python 源模块, 编译模块,或者二进制扩展). 启动 Python 时,这个列表从根据内建规则,PYTHONPATH 环境变量的内容, 以及注册表(Windows 系统)等进行初始化。

18、4G 内存怎么读取一个 5G 的数据?

方法一：

通过生成器，分多次读取，每次读取数量相对少的数据（比如 500MB）进行处理，处理结束后在读取后面的 500MB 的数据。

方法二：

可以通过 linux 命令 split 切割成小文件，然后再对数据进行处理，此方法效率比较高。可以按照行数切割，可以按照文件大小切割。

> 在Linux下用split进行文件分割：
> 模式一：指定分割后文件行数
> 对与txt文本文件，可以通过指定分割后文件的行数来进行文件分割。
> 命令：split -l 300 large_file.txt new_file_prefix
> 模式二：指定分割后文件大小
> split -b 10m server.log waynelog

19、输入某年某月某日，判断这一天是这一年的第几天?

使用 Python 标准库 datetime

```
import datetime

def dayofyear():
    year = input("请输入年份：")
    month = input("请输入月份：")
    day = input("请输入天：")
    date1 = datetime.date(year=int(year)，month=int(month)，day=int(day))
    date2 = datetime.date(year=int(year)，month=1，day=1)
    return (date1-date2+1).days
```

20、说明一下 os.path 和 sys.path 分别代表什么?

os.path 主要是用于对系统路径文件的操作。

sys.path 主要是对 Python 解释器的系统环境参数的操作（动态的改变 Python 解释器搜索路径）。

21、Python 中的 os 模块常见方法?

os.remove() 删除文件

os.rename() 重命名文件

os.walk() 生成目录树下的所有文件

os.chdir() 改变目录

os.mkdir/makedirs 创建目录/多层目录

os.rmdir/removedirs 删除目录/多层目录

os.listdir() 列出指定目录的文件

os.getcwd() 取得当前工作目录

os.chmod() 改变目录权限

os.path.basename() 去掉目录路径，返回文件名

os.path.dirname() 去掉文件名，返回目录路径

os.path.join() 将分离的各部分组合成一个路径名

os.path.split() 返回（dirname(),basename())元组

os.path.splitext() 返回(filename,extension)元组

os.path.getatime\ctime\mtime 分别返回最近访问、创建、修改时间

os.path.getsize() 返回文件大小

os.path.exists() 是否存在

os.path.isabs() 是否为绝对路径

os.path.isdir() 是否为目录

os.path.isfile() 是否为文件

22、说一下字典和 json 的区别?

字典是一种数据结构，json 是一种数据的表现形式，字典的 key 值只要是能 hash 的就行，json 的必须是字符串。

23、什么是可变、不可变类型?

可变不可变指的是内存中的值是否可以被改变，不可变类型指的是对象所在内存块里面的值不可以改变，有数值、字符串、元组；可变类型则是可以改变，主要有列表、字典。

24、存入字典里的数据有没有先后排序?

存入的数据不会自动排序，可以使用 sort 函数对字典进行排序。

25、lambda 表达式格式以及应用场景?

lambda函数就是可以接受任意多个参数（包括可选参数）并且返回单个表达式值得函数。

语法：lambda [arg1 [,arg2,.....argn]]:expression

```
def calc(x,y):
    return x*y
```

将上述一般函数改写为匿名函数：

```
lambda x,y:x*y
```

应用

(1) lambda函数比较轻便，即用即仍，适合完成只在一处使用的简单功能。

(2) 匿名函数，一般用来给filter，map这样的函数式编程服务

(3) 作为回调函数，传递给某些应用，比如消息处理。

26、如何理解 Python 中字符串中的\字符?

1、转义字符

2、路径名中用来连接路径名

3、编写太长代码手动软换行

27、常用的 Python 标准库都有哪些?

os 操作系统、time 时间、random 随机、pymysql 连接数据库、threading 线程、multiprocessing 进程、queue 队列

第三方库：
django、flask、requests、virtualenv、selenium、scrapy、xadmin、celery、re、hashlib、md5

常用的科学计算库：Numpy，Pandas、matplotlib

28、如何在Python中管理内存?

python中的内存管理由Python私有堆空间管理。所有Python对象和数据结构都位于私有堆中。程序员无权访问此私有堆。python解释器负责处理这个问题。

Python对象的堆空间分配由Python的内存管理器完成。核心API提供了一些程序员编写代码的工具。

Python还有一个内置的垃圾收集器，它可以回收所有未使用的内存，并使其可用于堆空间。

29、介绍一下 except 的作用和用法?

except: 捕获所有异常

except:<异常名>: 捕获指定异常

except:<异常名 1, 异常名 2>: 捕获异常 1 或者异常 2

except:<异常名>,<数据>: 捕获指定异常及其附加的数据

except:<异常名 1,异常名 2>:<数据>: 捕获异常名 1 或者异常名 2,及附加的数据

30、在 except 中 return 后还会不会执行 finally 中的代码？怎么抛出自定义异常?

会继续处理 finally 中的代码;
用 raise 方法可以抛出自定义异常。

31、read、readline 和 readlines 的区别?

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。

32、range 和 xrange 的区别?

两者用法相同，不同的是 range 返回的结果是一个列表，而 xrange 的结果是一个生成器，前者是直接开辟一块内存空间来保存列表，后者是边循环边使用，只有使用时才会开辟内存空间，所以当列表很长时，使用 xrange 性能要比 range 好。

33、请简述你对 input()函数的理解?

在 Python3 中，input()获取用户输入，不论用户输入的是什么，获取到的都是字符串类型的。

在 Python2 中有 raw_input()和 input(), raw_input()和 Python3 中的 input()作用是一样的，input()输入的是什么数据类型的，获取到的就是什么数据类型的。

34、代码中要修改不可变数据会出现什么问题？抛出什么异常?

代码不会正常运行，抛出 TypeError 异常。

35、print 调用 Python 中底层的什么方法?

print 方法默认调用 sys.stdout.write 方法，即往控制台打印字符串。

36、Python 的 sys 模块常用方

sys.argv 命令行参数 List，第一个元素是程序本身路径

sys.modules.keys() 返回所有已经导入的模块列表

sys.exc_info() 获取当前正在处理的异常类,exc_type、exc_value、exc_traceback 当前处理的异常详细信息

sys.exit(n) 退出程序，正常退出时 exit(0) sys.hexversion 获取 Python 解释程序的版本值，16 进制格式如：0x020403F0

sys.version 获取 Python 解释程序的版本信息

sys.maxint 最大的 Int 值

sys.maxunicode 最大的 Unicode 值

sys.modules 返回系统导入的模块字段，key 是模块名，value 是模块

sys.path 返回模块的搜索路径，初始化时使用 PYTHONPATH 环境变量的值

sys.platform 返回操作系统平台名称

sys.stdout 标准输出

sys.stdin 标准输入

sys.stderr 错误输出

sys.exc_clear() 用来清除当前线程所出现的当前的或最近的错误信息

sys.exec_prefix 返回平台独立的 python 文件安装的位置

sys.byteorder 本地字节规则的指示器，big-endian 平台的值是'big',little-endian 平台的值是'little' sys.copyright 记录 python 版权相关的东西

sys.api_version 解释器的 C 的 API 版本

sys.version_info 元组则提供一个更简单的方法来使你的程序具备 Python 版本要求功能

37、unittest 是什么?

在 Python 中，unittest 是 Python 中的单元测试框架。它拥有支持共享搭建、自动测试、在测试中暂停代码、将不同测试迭代成一组等的功能。

38、模块和包是什么?

在 Python 中，模块是搭建程序的一种方式。每一个 Python 代码文件都是一个模块，并可以引用其他的模块，比如对象和属性。

一个包含许多 Python 代码的文件夹是一个包。一个包可以包含模块和子文件夹。

39、什么是正则的贪婪匹配?

```
>>>re.search('ab*c', 'abcaxc')
<_sre.SRE_Match object; span=(0, 3), match='abc'>

>>>re.search('ab\D+c', 'abcaxc')
<_sre.SRE_Match object; span=(0, 6), match='abcaxc'>
```

贪婪匹配：正则表达式一般趋向于最大长度匹配，也就是所谓的贪婪匹配。

非贪婪匹配：就是匹配到结果就好，就少的匹配字符。

40、常用字符串格式化哪几种?

% 格式化字符串操作符

```
print 'hello %s and %s' % ('df', 'another df')
```

字典形式的字符串格式化方法

```
print 'hello %(first)s and %(second)s' % {'first': 'df', 'second': 'another df'}
```

字符串格式化（format）

(1) 使用位置参数

位置参数不受顺序约束，且可以为{}，参数索引从0开始，format里填写{}对应的参数值。

```
>>> msg = "my name is {}, and age is {}"
>>> msg.format("hqs",22)
'my name is hqs, and age is 22'
```

(2) 使用关键字参数

关键字参数值要对得上，可用字典当关键字参数传入值，字典前加**即可

```
>>> hash = {'name':'john' , 'age': 23}
>>> msg = 'my name is {name}, and age is {age}'
>>> msg.format(**hash)
'my name is john,and age is 23'
```

(3) 填充与格式化

:[填充字符][对齐方式 <^>][宽度]

```
>>> '{0:*<10}'.format(10)      # 左对齐
'10********'
```

41、面向对象深度优先和广度优先是什么?

在子类继承多个父类时，属性查找方式分深度优先和广度优先两种。

当类是经典类时，多继承情况下，在要查找属性不存在时，会按照深度优先方式查找下去。

当类是新式类时，多继承情况下，在要查找属性不存在时，会按照广度优先方式查找下去。

42、“一行代码实现 xx”类题目

(1) 一行代码实现 1 - 100 的和

可以利用 sum() 函数。

```
>>> sum(range(1, 101))
5050
```

(2) 一行代码实现数值交换

不用二话，直接换。

```
>>> a = 1
>>> b = 2
>>> a, b = b, a
>>> print(a, b)
2 1
```

(3) 一行代码求奇偶数

使用列表推导式。

```
>>> [x for x in range(10) if x % 2 == 1]
[1, 3, 5, 7, 9]
```

(4) 一行代码展开列表

使用列表推导式，稍微复杂一点，注意顺序。

```
>>> lst = [[1,2,3], [4,5,6], [7,8,9], [10,11,12]]
>>> [j for i in lst for j in i]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
```

(5) 一行代码打乱列表

用到 random 的 shuffle。

```
>>> import random
>>> lst = [1, 2, 3, 4, 5]
>>> random.shuffle(lst)
>>> lst
[1, 5, 3, 4, 2]
```

(6) 一行代码反转字符串

使用切片。

```
>>> name = 'Rocky0429'
>>> name[::-1]
'9240ykcoR'
```

(7) 一行代码查看目录下所有文件

使用 os 的 listdir。

```
>>> import os
>>> os.listdir('.')
[
```

(8) 一行代码去除字符串间的空格

法 1 replace 函数。

```
>>> des = 'My name is Rocky0429'
>>> des.replace(" ", "")
'MynameisRocky0429'
```

法 2 join & split 函数。

```
>>> des = 'My name is Rocky0429'
>>> "".join(des.split(" "))
'MynameisRocky0429'
```

(9) 一行代码实现字符串整数列表变成整数列表

使用 list & map & lambda。

```
>>> a = ['1', '2', '3']
>>> list(map(lambda a : int(a), ['1', '2', '3']))
[1, 2, 3]
```

(10) 一行代码删除列表中重复的值

使用 list & set。

```
>>> lst = [1, 1, 2, 2, 3, 3, 4]
>>> list(set(lst))
[1, 2, 3, 4]
```

**(11) 一行代码实现 9 * 9 乘法表

稍稍复杂的列表推导式，耐心点就行，一点点的搞...

```
>>> print('\n'.join([' '.join(['%s*%s=%-2s' % (j, i, i * j) for j in range(1, i + 1)]) for i in range(1, 10)]))
1*1=1
1*2=2  2*2=4
1*3=3  2*3=6  3*3=9
1*4=4  2*4=8  3*4=12 4*4=16
1*5=5  2*5=10 3*5=15 4*5=20 5*5=25
1*6=6  2*6=12 3*6=18 4*6=24 5*6=30 6*6=36
1*7=7  2*7=14 3*7=21 4*7=28 5*7=35 6*7=42 7*7=49
1*8=8  2*8=16 3*8=24 4*8=32 5*8=40 6*8=48 7*8=56 8*8=64
1*9=9  2*9=18 3*9=27 4*9=36 5*9=45 6*9=54 7*9=63 8*9=72 9*9=81
```


(12) 一行代码找出两个列表中相同的元素

使用 set 和 &。

```
>>> a = [1, 1, 2, 2, 4]
>>> b = [2, 3, 4 , 4]
>>> set(a) & set(b)
{2, 4}
```

(13) 一行代码找出两个列表中不同的元素

使用 set 和 ^。

```
>>> a = [1, 1, 2, 2, 4]
>>> b = [2, 3, 4 , 4]
>>> set(a) ^ set(b)
{1, 3}
```

(14)一行代码合并两个字典

使用 Update 函数。

```
>>> des = {'name': 'Rocky0429'}
>>> age = {'age': '100'}
>>> des.update(age)
>>> des
{'name': 'Rocky0429', 'age': '100'}
```

(15) 一行代码实现字典键从小到大排序

使用 sort 函数。

```
>>> des = {'name': 'Rocky0429', 'age': '100', 'like': 'Python'}
>>> sorted(des.items(), key=lambda x: x[0])
[('age', '100'), ('like', 'Python'), ('name', 'Rocky0429')]
```

想要了解更多，请扫描下方二维码，我在等你呦

Python 进阶（共 47 题）

1、Python 中类方法、类实例方法、静态方法有何区别?

类方法：是类对象的方法，在定义时需要在上方使用“@classmethod”进行装饰，形参为 cls，表示类对象，类对象和实例对象都可调用

类实例方法：是类实例化对象的方法，只有实例对象可以调用，形参为 self，指代对象本身

静态方法：是一个任意函数，在其上方使用“@staticmethod”进行装饰，可以用对象直接调用，静态方法实际上跟该类没有太大关系

2、Python 的内存管理机制及调优手段?

内存管理机制：引用计数、垃圾回收、内存池。

引用计数

引用计数是一种非常高效的内存管理手段， 当一个 Python 对象被引用时其引用计数增加 1， 当其不再被一个变量引用时则计数减 1. 当引用计数等于 0 时对象被删除。

垃圾回收

(1) 引用计数

引用计数也是一种垃圾收集机制，而且也是一种最直观，最简单的垃圾收集技术。当 Python 的某个对象的引用计数降为 0 时，说明没有任何引用指向该对象，该对象就成为要被回收的垃圾了。比如某个新建对象，它被分配给某个引用，对象的引用计数变为 1。如果引用被删除，对象的引用计数为 0，
那么该对象就可以被垃圾回收。不过如果出现循环引用的话，引用计数机制就不再起有效的作用了

(2)标记清除

如果两个对象的引用计数都为 1，但是仅仅存在他们之间的循环引用，那么这两个对象都是需要被回收的，也就是说，它们的引用计数虽然表现为非 0，但实际上有效的引用计数为 0。所以先将循环引用摘掉，就会得出这两个对象的有效计数。

(3) 分代回收

从前面“标记-清除”这样的垃圾收集机制来看，这种垃圾收集机制所带来的额外操作实际上与系统中总的内存块的数量是相关的，当需要回收的内存块越多时，垃圾检测带来的额外操作就越多，而垃圾回收带来的额外操作就越少；反之，当需回收的内存块越少时，垃圾检测就将比垃圾回收带来更少的额外操作。

举个例子：

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。

内存池

(1) Python 的内存机制呈现金字塔形状，-1，-2 层主要有操作系统进行操作

(2) 第 0 层是 C 中的 malloc，free 等内存分配和释放函数进行操作

(3)第 1 层和第 2 层是内存池，有 Python 的接口函数 PyMem_Malloc 函数实现，当对象小于 256K 时有该层直接分配内存

(4) 第 3 层是最上层，也就是我们对 Python 对象的直接操作

Python 在运行期间会大量地执行 malloc 和 free 的操作，频繁地在用户态和核心态之间进行切换，这将严重影响 Python 的执行效率。为了加速 Python 的执行效率，Python 引入了一个内存池机制，用于管理对小块内存的申请和释放。

Python 内部默认的小块内存与大块内存的分界点定在 256 个字节，当申请的内存小于 256 字节时，PyObject_Malloc 会在内存池中申请内存；当申请的内存大于 256 字节时，PyObject_Malloc 的行为将蜕化为 malloc 的行为。当然，通过修改 Python 源代码，我们可以改变这个默认值，从而改变 Python 的默认内存管理行为。

3、内存泄露是什么？如何避免？

由于疏忽或错误造成程序未能释放已经不再使用的内存的情况。

内存泄漏并非指内存在物理上的消失，而是应用程序分配某段内存后，由于设计错误，失去了对该段内存的控制，因而造成了内存的浪费。导致程序运行速度减慢甚至系统崩溃等严重后果。

del() 函数的对象间的循环引用是导致内存泄漏的主凶。

不使用一个对象时使用:del object 来删除一个对象的引用计数就可以有效防止内存泄漏问题。

通过 Python 扩展模块 gc 来查看不能回收的对象的详细信息。

可以通过 sys.getrefcount(obj) 来获取对象的引用计数，并根据返回值是否为 0 来判断是否内存泄漏。

4、Python 函数调用的时候参数的传递方式是值传递还是引用传递?

Python 的参数传递有：位置参数、默认参数、可变参数、关键字参数。函数的传值到底是值传递还是引用传递，要分情况：

不可变参数用值传递

像整数和字符串这样的不可变对象，是通过拷贝进行传递的，因为你无论如何都不可能在原处改变不可变对象

可变参数是引用传递的

比如像列表，字典这样的对象是通过引用传递、和 C 语言里面的用指针传递数组很相似，可变对象能在函数内部改变。

5、对缺省参数的理解?

缺省参数指在调用函数的时候没有传入参数的情况下，调用默认的参数，在调用函数的同时赋值时，所传入的参数会替代默认参数。

*args 是不定长参数，他可以表示输入参数是不确定的，可以是任意多个。

**kwargs 是关键字参数，赋值的时候是以键 = 值的方式，参数是可以任意多对在定义函数的时候不确定会有多少参数会传入时，就可以使用两个参数。

补充

***args**

如果你之前学过 C 或者 C++，看到星号的第一反应可能会认为这个与指针相关，然后就开始方了，其实放宽心，Python 中是没有指针这个概念的。

在 Python 中我们使用星号收集位置参数，请看下面的例子：

```
>>> def fun(x,*args):
...     print(x)
...     res = x
...     print(args)
...     for i in args:
...             res += i
...     return res
...
>>> print(fun(1,2,3,4,5,6))
```

上述例子中，函数的参数有两个，但是我们在输出的时候赋给函数的参数个数不仅仅是两个，让我们来运行这个代码，得到如下的结果：

```
1
(2, 3, 4, 5, 6)
21
```

从上面我们可以看出，参数 x 得到的值是 1，参数 args 得到的是一个元组 (2,3,4,5,6)，由此我们可以得出，如果输入的参数个数不确定，其它的参数全部通过 *args 以元组的方式由 arg 收集起来。

为了更能明显的看出 *args，我们下面用一个简单的函数来表示：

```
>>> def print_args(*args):
...     print(args)
...
```

接下来我传入不同的值，通过参数 *args 得到的结果我们来看一下：

```
>>> print_args(1,2,3)
(1, 2, 3)
>>> print_args('abc','def','ghi')
('abc', 'def', 'ghi')
>>> print_args('abc',['a','b','c'],1,2,3)
('abc', ['a', 'b', 'c'], 1, 2, 3)
```

不管是什么，都可以一股脑的塞进元组里，即使只有一个值，也是用元组收集，所以还记得在元组里一个元素的时候的形式吗？元组中如果只有一个元素，该元素的后面要有一个逗号。

那么如果不给 *args 传值呢？

```
>>> def print_args(*args):
...     print(args)
...
>>> print_args()
()
```

答案就是这时候 *args 收集到的是一个空的元组。

最后提醒一点的是，当使用星号的时候，不一定要把元组参数命名为 args，但这个是 Python 中的一个常见做法。

kwargs**

使用两个星号是收集关键字参数，可以将参数收集到一个字典中，参数的名字是字典的 “键”，对应的参数的值是字典的 “值”。请看下面的例子：

```
>>> def print_kwargs(**kwargs):
...     print(kwargs)
...
>>> print_kwargs(a = 'lee',b = 'sir',c = 'man')
{'a': 'lee', 'b': 'sir', 'c': 'man'}
```

由例子可以看出，在函数内部，kwargs 是一个字典。

看到这的时候，可能聪明的你会想，参数不是具有不确定型吗？如何知道参数到底会用什么样的方式传值？其实这个很好办，把 *args 和 **kwargs 综合起来就好了啊，请看下面的操作：

```
>>> def print_all(x,y,*args,**kwargs):
...     print(x)
...     print(y)
...     print(args)
...     print(kwargs)
...
```

```
>>> print_all('lee',1234)
lee
1234
()
{}
>>> print_all('lee',1,2,3,4,5)
lee
1
(2, 3, 4, 5)
{}
>>> print_all('lee',1,2,3,4,5,like = 'python')
lee
1
(2, 3, 4, 5)
{'like': 'python'}
```

如此这般，我们就可以应对各种各样奇葩无聊的参数请求了。当然在这还是要说的是，这里的关键字参数命名不一定要是 kwargs，但这个是通常做法。

6、为什么函数名字可以当做参数用?

Python 中一切皆对象，函数名是函数在内存中的空间，也是一个对象。

7、Python 中 pass 语句的作用是什么?

在编写代码时只写框架思路，具体实现还未编写就可以用 pass 进行占位，使程序不报错，不会进行任何操作。

8、面向对象中super的作用?

super() 函数是用于调用父类(超类)的一个方法。

super 是用来解决多重继承问题的，直接用类名调用父类方法在使用单继承的时候没问题，但是如果使用多继承，会涉及到查找顺序（MRO）、重复调用（钻石继承）等种种问题。

MRO 就是类的方法解析顺序表, 其实也就是继承父类方法时的顺序表。

作用：

- 根据 mro 的顺序执行方法
- 主动执行 Base 类的方法

9、是否使用过functools中的函数？其作用是什么?

Python的functools模块用以为可调用对象（callable objects）定义高阶函数或操作。

简单地说，就是基于已有的函数定义新的函数。

所谓高阶函数，就是以函数作为输入参数，返回也是函数。

10、json序列化时，默认遇到中文会转换成unicode，如果想要保留中文怎么办?

```
import json

a = json.dumps({"ddf": "你好"}, ensure_ascii=False)
print(a)
# {"ddf": "你好"}
```

11、什么是断言？应用场景?

assert断言——声明其布尔值必须为真判定，发生异常则为假。

```
info = {}
info['name'] = 'egon'
info['age'] = 18

# 用assert取代上述代码：
assert ('name' in info) and ('age' in info)
```

设置一个断言目的就是要求必须实现某个条件。

12、有用过with statement吗？它的好处是什么?

with语句的作用是通过某种方式简化异常处理，它是所谓的上下文管理器的一种

用法举例如下：

```
 with open('output.txt', 'w') as f:
        f.write('Hi there!')
```

当你要成对执行两个相关的操作的时候，这样就很方便，以上便是经典例子，with语句会在嵌套的代码执行之后，自动关闭文件。

这种做法的还有另一个优势就是，无论嵌套的代码是以何种方式结束的，它都关闭文件。

如果在嵌套的代码中发生异常，它能够在外部exception handler catch异常前关闭文件。

如果嵌套代码有return/continue/break语句，它同样能够关闭文件。

13、简述 Python 在异常处理中，else 和 finally 的作用分别是什么?

如果一个 Try - exception 中，没有发生异常，即 exception 没有执行，那么将会执行 else 语句的内容。反之，如果触发了 Try - exception（异常在 exception 中被定义），那么将会执行exception 中的内容，而不执行 else 中的内容。

如果 try 中的异常没有在 exception 中被指出，那么系统将会抛出 Traceback(默认错误代码）,并且终止程序，接下来的所有代码都不会被执行，但如果有 Finally 关键字，则会在程序抛出 Traceback 之前（程序最后一口气的时候），执行 finally 中的语句。这个方法在某些必须要结束的操作中颇为有用，如释放文件句柄，或释放内存空间等。

14、map 函数和 reduce 函数?

(1) 从参数方面来讲：

map()包含两个参数，第一个参数是一个函数，第二个是序列（列表 或元组）。其中，函数（即 map 的第一个参数位置的函数）可以接收一个或多个参数。

reduce()第一个参数是函数，第二个是序列（列表或元组）。但是，其函数必须接收两个参数。

(2) 从对传进去的数值作用来讲：

map()是将传入的函数依次作用到序列的每个元素，每个元素都是独自被函数“作用”一次 。

reduce()是将传人的函数作用在序列的第一个元素得到结果后，把这个结果继续与下一个元素作用（累积计算）。

补充 Python 特殊函数

lambda 函数

lambda 是一个可以只用一行就能解决问题的函数，让我们先看下面的例子：

```
>>> def add(x):
...     x += 1
...     return x
...
>>> numbers = range(5)
>>> list(numbers)
[0, 1, 2, 3, 4]
>>> new_numbers = []
>>> for i in numbers:
...     new_numbers.append(add(i))
...
>>> new_numbers
[1, 2, 3, 4, 5]
```

在上面的这个例子中，函数 add() 充当了一个中间角色，当然上面的例子也可以如下实现：

```
>>> new_numbers = [i+1 for i in numbers]
>>> new_numbers
[1, 2, 3, 4, 5]
```

首先我要说，上面的列表解析式其实是很好用的，但是我偏偏要用 lambda 这个函数代替 add(x)：

```
>>> lamb = lambda x: x+1
>>> new_numbers = []
>>> for i in numbers:
...     new_numbers.append(lamb(i))
...
>>> new_numbers
[1, 2, 3, 4, 5]
```

在这里的 lamb 就相当于 add(x)，lamb = lambda x : x+1 就相当于 add(x) 里的代码块。下面再写几个应用 lambda 的小例子：

```
>>> lamb = lambda x,y : x + y
>>> lamb(1,2)
3
>>> lamb1 = lambda x : x ** 2
>>> lamb1(5)
25
```

由上面的例子我们可以总结一下 lambda 函数的具体使用方法：lambda 后面直接跟变量，变脸后面是冒号，冒号后面是表达式，表达式的计算结果就是本函数的返回值。

在这里有一点需要提醒的是，虽然 lambda 函数可以接收任意多的参数并且返回单个表达式的值，但是 lambda 函数不能包含命令且包含的表达式不能超过一个。如果你需要更多复杂的东西，你应该去定义一个函数。

lambda 作为一个只有一行的函数，在你具体的编程实践中可以选择使用，虽然在性能上没什么提升，但是看着舒服呀。

map 函数

我们在上面讲 lambda 的时候用的例子，其实 map 也可以实现，请看下面的操作：

```
>>> numbers = [0,1,2,3,4]
>>> map(add,numbers)
[1, 2, 3, 4, 5]
>>> map(lambda x: x + 1,numbers)
[1, 2, 3, 4, 5]
```

map 是 Python 的一个内置函数，它的基本格式是：map(func, seq)。

func 是一个函数对象，seq 是一个序列对象，在执行的时候，seq 中的每个元素按照从左到右的顺序依次被取出来，塞到 func 函数里面，并将 func 的返回值依次存到一个列表里。

对于 map 要主要理解以下几个点就好了：

1.对可迭代的对象中的每一个元素，依次使用 fun 的方法（其实本质上就是一个 for 循环）。

2.将所有的结果返回一个 map 对象，这个对象是个迭代器。

我们接下来做一个简单的小题目：将两个列表中的对应项加起来，把结果返回在一个列表里，我们用 map 来做，如果你做完了，请往下看：

```
>>> list1 = [1,2,3,4]
>>> list2 = [5,6,7,8]
>>> list(map(lambda x,y: x + y,list1,list2))
[6, 8, 10, 12]
```

你看上面，是不是很简单？其实这个还看不出 map 的方便来，因为用 for 同样也不麻烦，要是你有这样的想法的话，那么请看下面：

```
>>> list1 = [1,2,3,4]
>>> list2 = [5,6,7,8]
>>> list3 = [9,10,11,12]
>>> list(map(lambda x,y,z : x + y + z,list1,list2,list3))
[15, 18, 21, 24]
```

你看三个呢？是不是用 for 的话就稍显麻烦了？那么我们在想如果是 四个，五个乃至更多呢？这就显示出 map 的简洁优雅了，并且 map 还不和 lambda 一样对性能没有什么提高，map 在性能上的优势也是杠杠的。

filter 函数

filter 翻译过来的意思是 “过滤器”，在 Python 中，它也确实是起到的是过滤器的作用。这个解释起来略微麻烦，还是直接上代码的好，在代码中体会用法是我在所有的文章里一直在体现的：

```
>>> numbers = range(-4,4)
>>> list(filter(lambda x: x > 0,numbers))
[1, 2, 3]
```

上面的例子其实和下面的代码是等价的：

```
>>> [x for x in numbers if x > 0]
[1, 2, 3]
```

然后我们再来写一个例子体会一下：

```
>>> list(filter(lambda x: x != 'o','Rocky0429'))
['R', 'c', 'k', 'y', '0', '4', '2', '9']
```

reduce 函数

我在之前的文章中很多次都说过，我的代码都是用 Python3 版本的。在 Python3 中，reduce 函数被放到 functools 模块里，在 Python2 中还是在全局命名空间。

同样我先用一个例子来跑一下，我们来看看怎么用：

```
>>> reduce(lambda x,y: x+y,[1,2,3,4])
10
```

reduce 函数的第一个参数是一个函数，第二个参数是序列类型的对象，将函数按照从左到右的顺序作用在序列上。如果你还不理解的话，我们下面可以对比一下它和 map 的区别：

```
>>> list1 = [1,2,3,4]
>>> list2 = [5,6,7,8]
>>> list(map(lambda x,y: x + y,list1,list2))
[6, 8, 10, 12]
```

对比上面的两个例子，就知道两者的区别，map 相当于是上下运算的，而 reduce 是从左到右逐个元素进行运算。

15、递归函数停止的条件?

递归的终止条件一般定义在递归函数内部，在递归调用前要做一个条件判断，根据判断的结果选择是继续调用自身，还是 return;返回终止递归。

终止的条件：

(1) 判断递归的次数是否达到某一限定值

(2) 判断运算的结果是否达到某个范围等，根据设计的目的来选择

16、回调函数，如何通信的?

回调函数是把函数的指针(地址)作为参数传递给另一个函数，将整个函数当作一个对象，赋值给调用的函数。

17、 *__setattr__*，_getattr，__delattr函数使用详解?

1.**setattr**(self,name,value)：如果想要给 name 赋值的话，就需要调用这个方法。

2.**getattr**(self,name)：如果 name 被访问且它又不存在，那么这个方法将被调用。

3.**delattr**(self,name)：如果要删除 name 的话，这个方法就要被调用了。

下面我们用例子来演示一下：

```
>>> class Sample:
...     def __getattr__(self,name):
...             print('hello getattr')
...     def __setattr__(self,name,value):
...             print('hello setattr')
...             self.__dict__[name] = value
...
```

上面的例子中类 Sample 只有两个方法，下面让我们实例化一下：

```
>>> s = Sample()
>>> s.x
hello getattr
```

s.x 这个实例属性本来是不存在的，但是由于类中有了 **getattr**(self,name) 方法，当发现属性 x 不存在于对象的 **dict** 中时，就调用了 **getattr**，也就是所谓的「拦截成员」。

```
>>> s.x = 7
hello setattr
```

同理，给对象的属性赋值的时候，调用了 **setattr**(self,name,value) 方法，这个方法中有 self.**dict**[name] = value，通过这个就将属性和数据存到了对象 **dict** 中。如果再调用这个属性的话，会成为下面这样：

```
>>> s.x
7
```

出现这种结果的原因是它已经存在于对象的 **dict** 中了。

看了上面的两个，你是不是觉得上面的方法很有魔力呢？这就是「黑魔法」，但是它具体有什么应用呢？我们接着来看：

```
class Rectangle:
   """
   the width and length of Rectangle
   """

   def __init__(self):
       self.width = 0
       self.length = 0

   def setSize(self,size):
       self.width, self.length = size

   def getSize(self):
       return self.width, self.length

if __name__ == "__main__":
   r = Rectangle()
```

```
    r.width = 3
    r.length = 4
    print(r.getSize())
    print(r.setSize((30,40)))
    print(r.width)
    print(r.length)
```

上面是我根据一个很有名的例子改编的，你可以先想一下结果，想完以后可以接着往下看：

```
(3, 4)
30
40
```

这段代码运行的结果如上面所示，作为一个强迫证的码农，对于这种可以改进的程序当然不能容忍。我们在上面介绍的特殊方法，我们一定要学以致用，虽然重新写的不一定比原来的好，但我们还是要尝试去用：

```
class NewRectangle:
    """
    the width and length of Rectangle
    """

    def __init__(self):
        self.width = 0
        self.length = 0

    def __setattr__(self, name, value):
        if name == 'size':
            self.width, self.length = value
        else:
            self.__dict__[name] = value

    def __getattr__(self, name):
        if name == 'size':
            return self.width, self.length
        else:
            return AttributeError

if __name__ == "__main__":
    r = NewRectangle()
    r.width = 3
    r.length = 4
    print(r.size)
    r.size = 30,40
    print(r.width)
    print(r.length)
```

我们来看一下运行的结果：

```
(3, 4)
30
40
```

我们可以看到，除了类的写法变了以外，调用的方式没有变，结果也没有变。

18、请描述抽象类和接口类的区别和联系?

(1) 抽象类

规定了一系列的方法，并规定了必须由继承类实现的方法。由于有抽象方法的存在，所以抽象类不能实例化。可以将抽象类理解为毛坯房，门窗、墙面的样式由你自己来定，所以抽象类与作为基类的普通类的区别在于约束性更强。

(2) 接口类

与抽象类很相似，表现在接口中定义的方法，必须由引用类实现，但他与抽象类的根本区别在于用途：与不同个体间沟通的规则（方法），你要进宿舍需要有钥匙，这个钥匙就是你与宿舍的接口，你的同室也有这个接口，所以他也能进入宿舍，你用手机通话，那么手机就是你与他人交流的接口。

(3) 区别和关联

- 接口是抽象类的变体，接口中所有的方法都是抽象的。而抽象类中可以有非抽象方法。抽象类是声明方法的存在而不去实现它的类。
- 接口可以继承，抽象类不行。
- 接口定义方法，没有实现的代码，而抽象类可以实现部分方法。
- 接口中基本数据类型为 static 而抽类象不是。
- 接口可以继承，抽象类不行。
- 可以在一个类中同时实现多个接口。
- 接口的使用方式通过 implements 关键字进行，抽象类则是通过继承 extends 关键字进行。

19、请描述方法重载与方法重写?

(1)方法重载

是在一个类里面，方法名字相同，而参数不同。返回类型呢？可以相同也可以不同。重载是让类以统一的方式处理不同类型数据的一种手段。

(2) 方法重写

子类不想原封不动地继承父类的方法，而是想作一定的修改，这就需要采用方法的重写。方法重写又称方法覆盖。

20、什么是 lambda 函数？ 有什么好处?

lambda 函数是一个可以接收任意多个参数(包括可选参数)并且返回单个表达式值的函数

1、lambda 函数比较轻便，即用即仍，很适合需要完成一项功能，但是此功能只在此一处使用，连名字都很随意的情况下

2、匿名函数，一般用来给 filter， map 这样的函数式编程服务

3、作为回调函数，传递给某些应用，比如消息处理

补充

lambda 是一个可以只用一行就能解决问题的函数，让我们先看下面的例子：

```
>>> def add(x):
...     x += 1
...     return x
...
>>> numbers = range(5)
>>> list(numbers)
[0, 1, 2, 3, 4]
>>> new_numbers = []
>>> for i in numbers:
...     new_numbers.append(add(i))
...
>>> new_numbers
[1, 2, 3, 4, 5]
```

在上面的这个例子中，函数 add() 充当了一个中间角色，当然上面的例子也可以如下实现：

```
>>> new_numbers = [i+1 for i in numbers]
>>> new_numbers
[1, 2, 3, 4, 5]
```

首先我要说，上面的列表解析式其实是很好用的，但是我偏偏要用 lambda 这个函数代替 add(x)：

```
>>> lamb = lambda x: x+1
>>> new_numbers = []
>>> for i in numbers:
...     new_numbers.append(lamb(i))
...
>>> new_numbers
[1, 2, 3, 4, 5]
```

在这里的 lamb 就相当于 add(x)，lamb = lambda x : x+1 就相当于 add(x) 里的代码块。下面再写几个应用 lambda 的小例子：

```
>>> lamb = lambda x,y : x + y
>>> lamb(1,2)
3
>>> lamb1 = lambda x : x ** 2
>>> lamb1(5)
25
```

由上面的例子我们可以总结一下 lambda 函数的具体使用方法：lambda 后面直接跟变量，变脸后面是冒号，冒号后面是表达式，表达式的计算结果就是本函数的返回值。

在这里有一点需要提醒的是，虽然 lambda 函数可以接收任意多的参数并且返回单个表达式的值，但是 lambda 函数不能包含命令且包含的表达式不能超过一个。如果你需要更多复杂的东西，你应该去定义一个函数。

lambda 作为一个只有一行的函数，在你具体的编程实践中可以选择使用，虽然在性能上没什么提升，但是看着舒服呀。

21、单例模式的应用场景有哪些?

单例模式应用的场景一般发现在以下条件下：

（1）资源共享的情况下，避免由于资源操作时导致的性能或损耗等。如日志文件，应用配置。

（2）控制资源的情况下，方便资源之间的互相通信。如线程池等。 1.网站的计数器 2.应用配置 3.多线程池 4.数据库配置，数据库连接池 5.应用程序的日志应用....

补充

01.单例设计模式

「单例设计模式」估计对很多人来说都是一个陌生的概念，其实它就环绕在你的身边。比如我们每天必用的听歌软件，同一时间只能播放一首歌曲，所以对于一个听歌的软件来说，负责音乐播放的对象只有一个；再比如打印机也是同样的道理，同一时间打印机也只能打印一份文件，同理负责打印的对象也只有一个。

结合说的听歌软件和打印机都只有唯一的一个对象，就很好理解「单例设计模式」。

> 单例设计模式确保一个类只有一个实例，并提供一个全局访问点。

「单例」就是单个实例，我们在定义完一个类的之后，一般使用「类名()」的方式创建一个对象，而单例设计模式解决的问题就是无论执行多少遍「类名()」，返回的对象内存地址永远是相同的。

02.new 方法

当我们使用「类名()」创建对象的时候，Python 解释器会帮我们做两件事情：第一件是为对象在内存分配空间，第二件是为对象进行初始化。初始化（**init**）我们已经学过了，那「分配空间」是哪一个方法呢？就是我们这一小节要介绍的 **new** 方法。

那这个 **new** 方法和单例设计模式有什么关系呢？单例设计模式返回的对象内存地址永远是相同的，这就意味着在内存中这个类的对象只能是唯一的一份，为达到这个效果，我们就要了解一下为对象分配空间的 **new** 方法。

明确了这个目的以后，接下来让我们看一下 **new** 方法。**new** 方法在内部其实做了两件时期：第一件事是为「对象分配空间」，第二件事是「把对象的引用返回给 Python 解释器」。当 Python 的解释器拿到了对象的引用之后，就会把对象的引用传递给 **init** 的第一个参数 self，**init** 拿到对象的引用之后，就可以在方法的内部，针对对象来定义实例属性。

这就是 **new** 方法和 **init** 方法的分工。

总结一下就是：之所以要学习 **new** 方法，就是因为需要对分配空间的方法进行改造，改造的目的就是为了当使用「类名()」创建对象的时候，无论执行多少次，在内存中永远只会创造出一个对象的实例，这样就可以达到单例设计模式的目的。

03.重写 new 方法

在这里我用一个 **new** 方法的重写来做一个演练：首先定义一个打印机的类，然后在类里重写一下 **new** 方法。通过对这个方法的重写来强化一下 **new** 方法要做的两件事情：在内存中分配内存空间 & 返回对象的引用。同时验证一下，当我们使用「类名()」创建对象的时候，Python 解释器会自动帮我们调用 **new** 方法。

首先我们先定义一个打印机类 Printer，并创建一个实例：

```
class Printer():
    def __init__(self):
        print("打印机初始化")
# 创建打印机对象
printer = Printer()
```

接下来就是重写 **new** 方法，在此之前，先说一下注意事项，只要⚠了这几点，重写 **new** 就没什么难度：

重写 **new** 方法一定要返回对象的引用，否则 Python 的解释器得不到分配了空间的对象引用，就不会调用对象的初始化方法；

new 是一个静态方法，在调用时需要主动传递 cls 参数。

```
# 重写 __new__ 方法
class Printer():
    def __new__(cls, *args, **kwargs):
        # 可以接收三个参数
        # 三个参数从左到右依次是 class，多值元组参数，多值的字典参数
        print("this is rewrite new")
        instance = super().__new__(cls)
        return instance
    def __init__(self):
        print("打印机初始化")
# 创建打印机对象
player = Printer()
print(player)
```

上述代码对 **new** 方法进行了重写，我们先来看一下输出结果：

```
this is rewrite new
打印机初始化
<__main__.Printer object at 0x10fcd2ba8>
```

上述的结果打印出了 **new** 方法和 **init** 方法里的内容，同时还打印了类的内存地址，顺序正好是我们在之前说过的。**new** 方法里的三行代码正好做了在本小节开头所说的三件事：

- print(this is rewrite new)：证明了创建对象时，**new** 方法会被自动调用；
- instance = super().**new**(cls)：为对象分配内存空间（因为 **new** 本身就有为对象分配内存空间的能力，所以在这直接调用父类的方法即可）；
- return instance：返回对象的引用。

04.设计单例模式

说了这么多，接下来就让我们用单例模式来设计一个单例类。乍一看单例类看起来比一般的类更唬人，但其实就是差别在一点：单例类在创建对象的时候，无论我们调用多少次创建对象的方法，得到的结果都是内存中唯一的对象。

可能到这有人会有疑惑：怎么知道用这个类创建出来的对象是同一个对象呢？其实非常的简单，我们只需要多调用几次创建对象的方法，然后输出一下方法的返回结果，如果内存地址是相同的，说明多次调用方法返回的结果，本质上还是同一个对象。

```
class Printer():
    pass

printer1 = Printer()
print(printer1)
printer2 = Printer()
print(printer2)
```

上面是一个一般类的多次调用，打印的结果如下所示：

```
<__main__.Printer object at 0x10a940780>
<__main__.Printer object at 0x10a94d3c8>
```

可以看出，一般类中多次调用的内存地址不同（即 printer1 和 printer2 是两个完全不同的对象），而单例设计模式设计的单例类 Printer()，要求是无论调用多少次创建对象的方法，控制台打印出来的内存地址都是相同的。

那么我们该怎么实现呢？其实很简单，就是多加一个「类属性」，用这个类属性来记录「单例对象的引用」。

为什么要这样呢？其实我们一步一步的来想，当我们写完一个类，运行程序的时候，内存中其实是没有这个类创建的对象的，我们必须调用创建对象的方法，内存中才会有第一个对象。在重写 **new** 方法的时候，我们用 instance = super().**new**(cls)，为对象分配内存空间，同时用 istance 类属性记录父类方法的返回结果，这就是第一个「对象在内存中的返回地址」。当我们再次调用创建对象的方法时，因为第一个对象已经存在了，我们直接把第一个对象的引用做一个返回，而不用再调用 super().**new**(cls) 分配空间这个方法，所以就不会在内存中为这个类的其它对象分配额外的内存空间，而只是把之前记录的第一个对象的引用做一个返回，这样就能做到无论调用多少次创建对象的方法，我们永远得到的是创建的第一个对象的引用。

这个就是使用单例设计模式解决在内存中只创建唯一一个实例的解决办法。下面我就根据上面所说的，来完成单例设计模式。

```
class Printer():
    instance = None
    def __new__(cls, *args, **kwargs):
        if cls.instance is None:
            cls.instance = super().__new__(cls)
    return cls.instance
printer1 = Printer()
print(printer1)
printer2 = Printer()
print(printer2)
```

上述代码很简短，首先给类属性复制为 None，在 **new** 方法内部，如果 instance 为 None，证明第一个对象还没有创建，那么就为第一个对象分配内存空间，如果 instance 不为 None，直接把类属性中保存的第一个对象的引用直接返回，这样在外界无论调用多少次创建对象的方法，得到的对象的内存地址都是相同的。

下面我们运行一下程序，来看一下结果是不是能印证我们的说法：

```
<__main__.Printer object at 0x10f3223c8>
<__main__.Printer object at 0x10f3223c8>
```

上述输出的两个结果可以看出地址完全一样，这说明 printer1 和 printer2 本质上是相同的一个对象。

22、什么是闭包?

我们都知道在数学中有闭包的概念，但此处我要说的闭包是计算机编程语言中的概念，它被广泛的使用于函数式编程。

关于闭包的概念，官方的定义颇为严格，也很难理解，在《Python语言及其应用》一书中关于闭包的解释我觉得比较好 -- 闭包是一个可以由另一个函数动态生成的函数，并且可以改变和存储函数外创建的变量的值。乍一看，好像还是比较很难懂，下面我用一个简单的例子来解释一下：

```
>>> a = 1
>>> def fun():
...     print(a)
...
>>> fun()
1
>>> def fun1():
...     b = 1
...
>>> print(b)
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
```

毋庸置疑，第一段程序是可以运行成功的，a = 1 定义的变量在函数里可以被调用，但是反过来，第二段程序则出现了报错。

在函数 fun() 里可以直接使用外面的 a = 1，但是在函数 fun1() 外面不能使用它里面所定义的 b = 1，如果我们根据作用域的关系来解释，是没有什么异议的，但是如果在某种特殊情况下，我们必须要在函数外面使用函数里面的变量，该怎么办呢?

我们先来看下面的例子：

```
>>> def fun():
...     a = 1
...     def fun1():
...             return a
...     return fun1
...
>>> f = fun()
>>> print(f())
1
```

如果你看过昨天的文章，你一定觉得的很眼熟，上述的本质就是我们昨天所讲的嵌套函数。

在函数 fun() 里面，有 a = 1 和 函数 fun1() ,它们两个都在函数 fun() 的环境里面，但是它们两个是互不干扰的，所以 a 相对于 fun1() 来说是自由变量，并且在函数 fun1() 中应用了这个自由变量 -- 这个 fun1() 就是我们所定义的闭包。

闭包实际上就是一个函数，但是这个函数要具有 1.定义在另外一个函数里面(嵌套函数)；2.引用其所在环境的自由变量。

上述例子通过闭包在 fun() 执行完毕时，a = 1依然可以在 f() 中，即 fun1() 函数中存在，并没有被收回，所以 print(f()) 才得到了结果。

当我们在某些时候需要对事务做更高层次的抽象，用闭包会相当舒服。比如我们要写一个二元一次函数，如果不使用闭包的话相信你可以轻而易举的写出来，下面让我们来用闭包的方式完成这个一元二次方程：

```
>>> def fun(a,b,c):
...     def para(x):
...             return a*x**2 + b*x + c
...     return para
...
>>> f = fun(1,2,3)
>>> print(f(2))
11
```

上面的函数中，f = fun(1,2,3) 定义了一个一元二次函数的函数对象，x^2 + 2x + 3，如果要计算 x = 2，该一元二次函数的值，只需要计算 f(2) 即可，这种写法是不是看起来更简洁一些。

23、什么是装饰器?

「装饰器」作为 Python 高级语言特性中的重要部分，是修改函数的一种超级便捷的方式，适当使用能够有效提高代码的可读性和可维护性，非常的便利灵活。

「装饰器」本质上就是一个函数，这个函数的特点是可以接受其它的函数当作它的参数，并将其替换成一个新的函数（即返回给另一个函数）。

可能现在这么看的话有点懵，为了深入理解「装饰器」的原理，我们首先先要搞明白「什么是函数对象」，「什么是嵌套函数」，「什么是闭包」。关于这三个问题我在很久以前的文章中已经写过了，你只需要点击下面的链接去看就好了，这也是面试中常问的知识哦：

零基础学习 Python 之函数对象

零基础学习 Python 之嵌套函数

零基础学习 Python 之闭包

装饰器

搞明白上面的三个问题，其实简单点来说就是告诉你：函数可以赋值给变量，函数可嵌套，函数对象可以作为另一个函数的参数。

首先我们来看一个例子，在这个例子中我们用到了前面列出来的所有知识：

```
def first(fun):
    def second():
        print('start')
        fun()
        print('end')
        print fun.__name__
    return second

def man():
    print('i am a man()')

f = first(man)
f()
```

上述代码的执行结果如下所示：

```
start
i am a man()
end
man
```

上面的程序中，这个就是 first 函数接收了 man 函数作为参数，并将 man 函数以一个新的函数进行替换。看到这你有没有发现，这个和我在文章刚开始时所说的「装饰器」的描述是一样的。既然这样的话，那我们就把上述的代码改造成符合 Python 装饰器的定义和用法的样子，具体如下所示：

```
def first(func):
    def second():
        print('start')
        func()
        print('end')
        print (func.__name__)
    return second

@first
def man():
    print('i am a man()')

man()
```

上面这段代码和之前的代码的作用一模一样。区别在于之前的代码直接“明目张胆”的使用 first 函数去封装 man 函数，而上面这个是用了「语法糖」来封装 man 函数。至于什么是语法糖，不用细去追究，你就知道是类似「@first」这种形式的东西就好了。

在上述代码中「@frist」在 man 函数的上面，表示对 man 函数使用 first 装饰器。「@」 是装饰器的语法，「first」是装饰器的名称。

下面我们再来看一个复杂点的例子，用这个例子我们来更好的理解一下「装饰器」的使用以及它作为 Python 语言高级特性被人津津乐道的部分：

```
def check_admin(username):
    if username != 'admin':
        raise Exception('This user do not have permission')

class Stack:
    def __init__(self):
        self.item = []

    def push(self,username,item):
        check_admin(username=username)
        self.item.append(item)

    def pop(self,username):
        check_admin(username=username)
        if not self.item:
            raise Exception('NO elem in stack')
        return self.item.pop()
```

上述实现了一个特殊的栈，特殊在多了检查当前用户是否为 admin 这步判断，如果当前用户不是 admin，则抛出异常。上面的代码中将检查当前用户的身份写成了一个独立的函数 check_admin，在 push 和 pop 中只需要调用这个函数即可。这种方式增强了代码的可读性，减少了代码冗余，希望大家在编程的时候可以具有这种意识。

下面我们来看看上述代码用装饰器来写成的效果：

```
def check_admin(func):
    def wrapper(*args, **kwargs):
        if kwargs.get('username') != 'admin':
            raise Exception('This user do not have permission')
        return func(*args, **kwargs)
    return wrapper

class Stack:
    def __init__(self):
        self.item = []

    @check_admin
    def push(self,username,item):
        self.item.append(item)

    @check_admin
    def pop(self,username):
        if not self.item:
            raise Exception('NO elem in stack')
        return self.item.pop()
```

PS：可能很多人对 *args 和 **kwargs 不太熟悉，详情请戳下面的链接：

Python 拓展之 *args & **kwargs

对比一下使用「装饰器」和不使用装饰器的两种写法，乍一看，好像使用「装饰器」以后代码的行数更多了，但是你有没有发现代码看起来好像更容易理解了一些。在没有装饰器的时候，我们先看到的是 check_admin 这个函数，我们得先去想这个函数是干嘛的，然后看到的才是对栈的操作；而使用装饰器的时候，我们上来看到的就是对栈的操作语句，至于 check_admin 完全不会干扰到我们对当前函数的理解，所以使用了装饰器可读性更好了一些。

就和我在之前的文章中所讲的「生成器」那样，虽然 Python 的高级语言特性好用，但也不能乱用。装饰器的语法复杂，通过我们在上面缩写的装饰器就可以看出，它写完以后是很难调试的，并且使用「装饰器」的程序的速度会比不使用装饰器的程序更慢，所以还是要具体场景具体看待。

24、函数装饰器有什么作用?

装饰器本质上是一个 Python 函数，它可以在让其他函数在不需要做任何代码的变动的前提下增加额外的功能。装饰器的返回值也是一个函数的对象，它经常用于有切面需求的场景。 比如：插入日志、性能测试、事务处理、缓存、权限的校验等场景 有了装饰器就可以抽离出大量的与函数功能本身无关的雷同代码并发并继续使用。

25、生成器、迭代器的区别

迭代器是一个更抽象的概念，任何对象，如果它的类有 next 方法和 iter 方法返回自己本身，对于 string、list、dict、tuple 等这类容器对象，使用 for 循环遍历是很方便的。在后台 for 语句对容器对象调用 iter()函数，iter()是 python 的内置函数。iter()会返回一个定义了 next()方法的迭代器对象，它在容器中逐个访问容器内元素，next()也是 python 的内置函数。在没有后续元素时，next()会抛出一个 StopIteration 异常。

生成器（Generator）是创建迭代器的简单而强大的工具。它们写起来就像是正规的函数，只是在需要返回数据的时候使用 yield 语句。每次 next()被调用时，生成器会返回它脱离的位置（它记忆语句最后一次执行的位置和所有的数据值）

区别：生成器能做到迭代器能做的所有事,而且因为自动创建了 iter()和 next()方法,生成器显得特别简洁,而且生成器也是高效的，使用生成器表达式取代列表解析可以同时节省内存。除了创建和保存程序状态的自动方法,当发生器终结时,还会自动抛出 StopIteration 异常。

26、多线程交互，访问数据，如果访问到了就不访问了，怎么避免重读?

创建一个已访问数据列表，用于存储已经访问过的数据，并加上互斥锁，在多线程访问数据的时候先查看数据是否已经在已访问的列表中，若已存在就直接跳过。

27、Python 中 yield 的用法?

yield 就是保存当前程序执行状态。你用 for 循环的时候，每次取一个元素的时候就会计算一次。用 yield 的函数叫 generator，和 iterator 一样，它的好处是不用一次计算所有元素，而是用一次算一次，可以节省很多空间。generator每次计算需要上一次计算结果，所以用 yield，否则一 return，上次计算结果就没了。

补充

在 Python 中，定义生成器必须要使用 yield 这个关键词，yield 翻译成中文有「生产」这方面的意思。在 Python 中，它作为一个关键词，是生成器的标志。接下来我们来看一个例子：

```
>>> def f():
...    yield 0
...    yield 1
...    yield 2
...
>>> f
<function f at 0x00000000004EC1E0>
```

上面是写了一个很简单的 f 函数，代码块是 3 个 yield 发起的语句，下面让我们来看看如何使用它：

```
>>> fa = f()
>>> fa
<generator object f at 0x0000000001DF1660>
>>> type(fa)
<class 'generator'>
```

上述操作可以看出，我们调用函数得到了一个生成器（generator）对象。

```
>>> dir(fa)
['__class__', '__del__', '__delattr__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__lt__', '__name__', '__ne__',
'__new__', '__next__', '__qualname__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'close',
'gi_code', 'gi_frame', 'gi_running', 'gi_yieldfrom', 'send', 'throw']
```

在上面我们看到了 **iter**() 和 **next**()，虽然我们在函数体内没有显示的写 **iter**() 和 **next**()，仅仅是写了 yield，但它就已经是「迭代器」了。既然如此，那我们就可以进行如下操作：

```
>>> fa = f()
>>> fa.__next__()
0
>>> fa.__next__()
1
>>> fa.__next__()
2
>>> fa.__next__()
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
StopIteration
```

从上面的简单操作可以看出：含有 yield 关键词的函数 f() 是一个生成器对象，这个生成器对象也是迭代器。所以就有了这样的定义：把含有 yield 语句的函数称为生成器，生成器是一种用普通函数语法定义的迭代器。

通过上面的例子可以看出，这个生成器（即迭代器）在定义的过程中并没有昨天讲的迭代器那样写 **iter**()，而是只用了 yield 语句，之后一个普普通通的函数就神奇的成了生成器，同样也具备了迭代器的特性。

yield 语句的作用，就是在调用的时候返回相应的值。下面我来逐行的解释一下上面例子的运行过程：

1.fa = f()：fa 引用生成器对象。

2.fa.**next**()：生成器开始执行，遇到了第一个 yield，然后返回后面的 0，并且挂起（即暂停执行）。

3.fa.**next**()：从上次暂停的位置开始，继续向下执行，遇到第二个 yield，返回后面的值 1，再挂起。

4.fa.**next**()：重复上面的操作。

5.fa.**next**()：从上次暂停的位置开始，继续向下执行，但是后面已经没有 yield 了，所以 **next**() 发生异常。

28、谈下 python 的 GIL

GIL 是python的全局解释器锁，同一进程中假如有多个线程运行，一个线程在运行python程序的时候会霸占python解释器（加了一把锁即GIL），使该进程内的其他线程无法运行，等该线程运行完后其他线程才能运行。如果线程运行过程中遇到耗时操作，则解释器锁解开，使其他线程运行。所以在多线程中，线程的运行仍是有先后顺序的，并不是同时进行。

多进程中因为每个进程都能被系统分配资源，相当于每个进程有了一个python解释器，所以多进程可以实现多个进程的同时运行，缺点是进程系统资源开销大

29、Python 中的可变对象和不可变对象?

不可变对象，该对象所指向的内存中的值不能被改变。当改变某个变量时候，由于其所指的值不能被改变，相当于把原来的值复制一份后再改变，这会开辟一个新的地址，变量再指向这个新的地址。

可变对象，该对象所指向的内存中的值可以被改变。变量（准确的说是引用）改变后，实际上是其所指的值直接发生改变，并没有发生复制行为，也没有开辟新的出地址，通俗点说就是原地改变。

Python 中，数值类型（int 和 float）、字符串 str、元组 tuple 都是不可变类型。而列表 list、字典 dict、集合 set 是可变类型。

30、一句话解释什么样的语言能够用装饰器?

函数可以作为参数传递的语言，可以使用装饰器

31、Python 中 is 和==的区别?

is 判断的是 a 对象是否就是 b 对象，是通过 id 来判断的。

==判断的是 a 对象的值是否和 b 对象的值相等，是通过 value 来判断的。

32、谈谈你对面向对象的理解?

面向对象是相对于面向过程而言的。

面向过程语言是一种基于功能分析的、以算法为中心的程序设计方法

面向对象是一种基于结构分析的、以数据为中心的程序设计思想。在面向对象语言中有一个有很重要东西，叫做类。面向对象有三大特性：封装、继承、多态。

33、Python 里 match 与 search 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。

34、用 Python 匹配 HTML g tag 的时候，<.> *和* <.?> 有什么区别?

<.*>是贪婪匹配，会从第一个“<”开始匹配，直到最后一个“>”中间所有的字符都会匹配到，中间可能会包含“<>”。

<.*?>是非贪婪匹配，从第一个“<”开始往后，遇到第一个“>”结束匹配，这中间的字符串都会匹配到，但是
不会有“<>”。

35、Python 中的进程与线程的使用场景?

多进程适合在 CPU 密集型操作(cpu 操作指令比较多，如位数多的浮点运算)。

多线程适合在 IO 密集型操作(读写数据操作较多的，比如爬虫)。

36、解释一下并行（parallel）和并发（concurrency）的区别?

并行（parallel）是指同一时刻，两个或两个以上时间同时发生。

并发（parallel）是指同一时间间隔（同一段时间），两个或两个以上时间同时发生。

37、如果一个程序需要进行大量的 IO 操作，应当使用并行还是并发?

并发

38、如果程序需要进行大量的逻辑运算操作，应当使用并行还是并发?

并行

39、在 Python 中可以实现并发的库有哪些?

- 线程
- 进程
- 协程
- threading

40、Python 如何处理上传文件?

Python 中使用 GET 方法实现上传文件，下面就是用 Get 上传文件的例子，client 用来发 Get 请求，server 用来收请求。

请求端代码

```
import requests #需要安装 requests

with open('test.txt', 'rb') as f:
requests.get('http://服务器 IP 地址:端口', data=f)
```

服务端代码

```
var http = require('http');
var fs = require('fs');
var server = http.createServer(function(req, res){

    var recData = "";
    req.on('data', function(data){
```

```
    recData += data;
    })
    req.on('end', function(data){
    recData += data;
    fs.writeFile('recData.txt', recData, function(err){
    console.log('file received');
        })
    })
    res.end('hello');
    })
server.listen(端口);
```

41、请列举你使用过的 Python 代码检测工具?

- 移动应用自动化测试 Appium
- OpenStack 集成测试 Tempest
- 自动化测试框架 STAF
- 自动化测试平台 TestMaker
- JavaScript 内存泄露检测工具 Leak Finder
- Python 的 Web 应用验收测试 Splinter
- 即插即用设备调试工具 UPnP-Inspector

42、python 程序中文输出问题怎么解决?

方法一

用encode和decode，如：

```
import os.path
import xlrd,sys

Filename='/home/tom/Desktop/1234.xls'
if not os.path.isfile(Filename):
    raise NameError,"%s is not a valid filename"%Filename

bk=xlrd.open_workbook(Filename)
shxrange=range(bk.nsheets)
print shxrange

for x in shxrange:
    p=bk.sheets()[x].name.encode('utf-8')
    print p.decode('utf-8')
```

方法二

在文件开头加上

```
reload(sys)

sys.setdefaultencoding('utf8')
```

43、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

44、如何用Python删除一个文件?

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

45、如何用 Python 来发送邮件?

python实现发送和接收邮件功能主要用到poplib和smtplib模块。

poplib用于接收邮件，而smtplib负责发送邮件。

```
#! /usr/bin/env python
#coding=utf-8
import sys
import time
import poplib
import smtplib

# 邮件发送函数
def send_mail():
     try:
        handle = smtplib.SMTP('smtp.126.com',25)
        handle.login('XXXX@126.com','**********')
        msg = 'To: XXXX@qq.com\r\nFrom:XXXX@126.com\r\nSubject:hello\r\n'
        handle.sendmail('XXXX@126.com','XXXX@qq.com',msg)
        handle.close()
        return 1
    except:
        return 0

# 邮件接收函数
def accpet_mail():
    try:
        p=poplib.POP3('pop.126.com')
        p.user('pythontab@126.com')
        p.pass_('**********')
        ret = p.stat() #返回一个元组:(邮件数,邮件尺寸)
       #p.retr('邮件号码')方法返回一个元组:(状态信息,邮件,邮件尺寸)
    except poplib.error_proto,e:
        print "Login failed:",e
        sys.exit(1)

# 运行当前文件时，执行sendmail和accpet_mail函数
if __name__ == "__main__":
    send_mail()
    accpet_mail()
```

46、当退出 Python 时，是否释放全部内存?

不是。

循环引用其它对象或引用自全局命名空间的对象的模块，在Python退出时并非完全释放。另外，也不会释放C库保留的内存部分。

47、什么是猴子补丁?

在运行期间动态修改一个类或模块。

```
>>> class A:
    def func(self):
        print("Hi")
>>> def monkey(self):
print "Hi, monkey"
>>> m.A.func = monkey
>>> a = m.A()
>>> a.func()
```

运行结果为:

```
Hi, Monkey
```


Python 后台开发（共 44 题）

Flask

1、Flask 中请求上下文和应用上下文的区别和作用?

current_app、g 是应用上下文。

request、session 是请求上下文。

手动创建上下文的两种方法:

```
with app.app_context()
app = current_app._get_current_object()
```

两者区别:

请求上下文：保存了客户端和服务器交互的数据。

应用上下文：flask 应用程序运行过程中，保存的一些配置信息，比如程序名、数据库连接、应用信息等。

两者作用：

请求上下文(request context)：Flask 从客户端收到请求时，要让视图函数能访问一些对象，这样才能处理请求。请求对象是一个很好的例子，它封装了客户端发送的 HTTP 请求。要想让视图函数能够访问请求对象，一个显而易见的方式是将其作为参数传入视图函数，不过这会导致程序中的每个视图函数都增加一个参数，除了访问请求对象,如果视图函数在处理请求时还要访问其他对象，情况会变得更糟。为了避免大量可有可无的参数把视图函数弄得一团糟，Flask使用上下文临时把某些对象变为全局可访问。

应用上下文(application context)：它的字面意思是 应用上下文，但它不是一直存在的，它只是 request context 中的一个对 app 的代理(人)，所谓 local proxy。它的作用主要是帮助 request 获取当前的应用，它是伴 request 而生，随 request 而灭的。

2、对 Flask 蓝图(Blueprint)的理解?

蓝图的定义

蓝图 /Blueprint 是 Flask 应用程序组件化的方法，可以在一个应用内或跨越多个项目共用蓝图。使用蓝图可以极大地简化大型应用的开发难度，也为 Flask 扩展 提供了一种在应用中注册服务的集中式机制。

蓝图的应用场景

(1) 把一个应用分解为一个蓝图的集合。这对大型应用是理想的。一个项目可以实例化一个应用对象，初始化几个扩展，并注册一集合的蓝图。

(2) 以 URL 前缀和/或子域名，在应用上注册一个蓝图。 URL 前缀/子域名中的参数即成为这个蓝图下的所有视图函数的共同的视图参数（默认情况下）。

(3) 在一个应用中用不同的 URL 规则多次注册一个蓝图。

(4) 通过蓝图提供模板过滤器、静态文件、模板和其它功能。一个蓝图不一定要实现应用或者视图函数。

(5) 初始化一个 Flask 扩展时，在这些情况中注册一个蓝图。

蓝图的缺点

不能在应用创建后撤销注册一个蓝图而不销毁整个应用对象。

使用蓝图的三个步骤

(1) 创建一个蓝图对象

```
blue = Blueprint("blue", __name__)
```

(2) 在这个蓝图对象上进行操作 ，例如注册路由、指定静态文件夹、注册模板过滤器...

```
@blue.route('/')
def blue_index():
return 'Welcome to my blueprint'
```

(3) 在应用对象上注册这个蓝图对象

```
app.register_blueprint(blue, url_prefix='/blue')
```

3、Flask 项目中如何实现 session 信息的写入?

Flask 中有三个 session：

(1) 数据库中的 session，例如:db.session.add()

(2)在 flask_session 扩展中的 session，使用：from flask_session importSession，使用第三方扩展的 session 可以把信息存储在服务器中，客户端浏览器中只存储 sessionid

(3) flask 自带的 session，是一个请求上下文， 使用：from flask import session。自带的session 把信息加密后都存储在客户端的浏览器 cookie 中。

4、项目接口实现后路由访问不到怎么办?

可以通过 postman 测试工具测试，或者看 log 日志信息找到错误信息的大概位置。

5、Flask 中 url_for 函数?

URL 反转：根据视图函数名称得到当前所指向的 url。

url_for() 函数最简单的用法是以视图函数名作为参数，返回对应的 url，还可以用作加载静态文件。

```
<link rel="stylesheet" href="{{url_for('static',filename='css/index.css')}}">
```

该条语句就是在模版中加载 css 静态文件。

url_for 和 redirect 区别

url_for 是用来拼接 URL 的，可以使用程序 URL 映射中保存的信息生成 URL。

url_for() 函数最简单的用法是以视图函数名作为参数， 返回对应的 URL。例如，在示例程序中 hello.py 中调用 url_for('index') 得到的结果是 /。

redirect 是重定向函数，输入一个 URL 后，自动跳转到另一个 URL 所在的地址，例如，你在函数中写 return redirect('https://www.baidu.com') 页面就会跳转向百度页面。

```
from flask import Flask,redirect,url_for

app = Flask(__name__)
@app.route('/')

def index():
    login_url = url_for('login')
    return redirect(login_url)
    return u'这是首页'

@app.route('/login/')
def login():
    return u'这是登陆页面'

@app.route('/question/<is_login>/')
def question(is_login):
    if is_login == '1':
        return u'这是发布问答的页面'
    else:
        return redirect(url_for('login'))

if __name__ == '__main__':
```

```
    app.run(debug=True)
```

6、Flask 中请求钩子的理解和应用?

请求钩子是通过装饰器的形式实现的，支持以下四种：

> before_first_request 在处理第一个请求前运行
> before_request 在每次请求前运行
>
> after_request 如果没有未处理的异常抛出，在每次请求后运行
> teardown_request 即使有未处理的异常抛出，在每次请求后运行

应用：

请求钩子

```
@api.after_request
def after_request(response):
"""设置默认的响应报文格式为 application/json"""
    # 如果响应报文 response 的 Content-Type 是以 text 开头，则将其改为
    # 默认的 json 类型
    if response.headers.get("Content-Type").startswith("text"):
        response.headers["Content-Type"] = "application/json"
    return respon
```

7、一个变量后写多个过滤器是如何执行的?

{{ expression | filter1 | filter2 | ... }} 即表达式(expression)使用 filter1 过滤后再使用 filter2 过滤。

8、如何把整个数据库导出来，再导入指定数据库中?

导出

```
mysqldump [-h 主机] -u 用户名 -p 数据库名 > 导出的数据库名.sql
```

导入指定的数据库中:

第一种方法：

```
mysqldump [-h 主机] -u 用户名 -p 数据库名 < 导出的数据库名.sql
```

第二种方法：

先创建好数据库，因为导出的文件里没有创建数据库的语句，如果数据库已经建好，则不用再创建。

```
create database example charset=utf8;（数据库名可以不一样）
```

切换数据库：

```
use example;
```

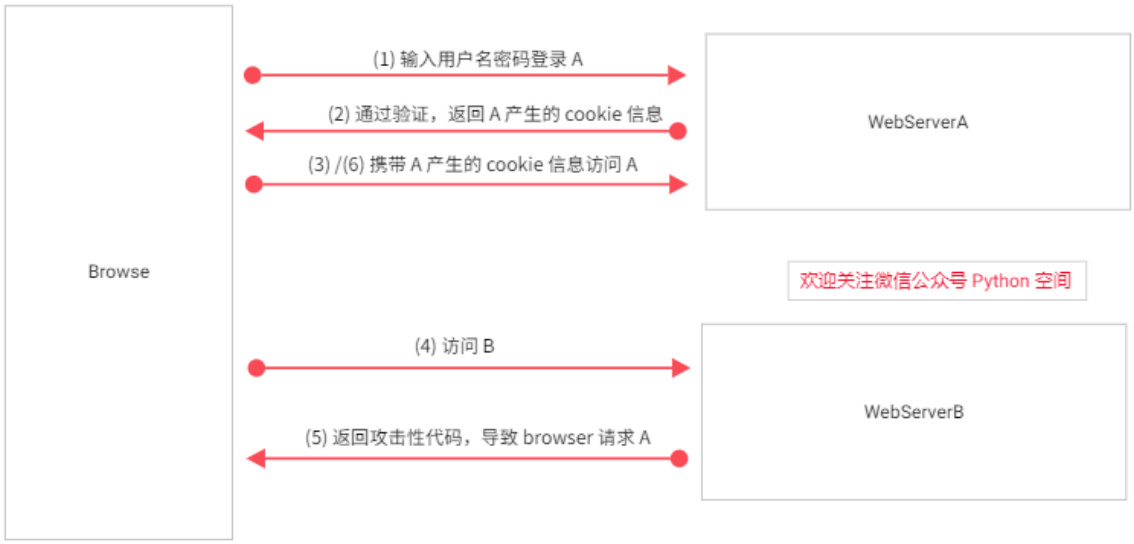
导入指定 sql 文件：

```
mysql>source /path/example.sql;
```

9、Flask 和 Django 路由映射的区别?

在 django 中，路由是浏览器访问服务器时，先访问的项目中的 url，再由项目中的 url 找到应用 url，这些 url 是放在一个列表里，遵从从前往后匹配的规则。在 flask 中，路由是通过装饰器给每个视图函数提供的，而且根据请求方式的不同可以一个 url 用于不同的作用。

10、跨站请求伪造和跨站请求保护的实现?



图中 Browse 是浏览器，WebServerA 是受信任网站/被攻击网站 A，WebServerB 是恶意网站/点击网站 B。

（1）一开始用户打开浏览器，访问受信任网站 A，输入用户名和密码登陆请求登陆网站 A。

（2）网站 A 验证用户信息，用户信息通过验证后，网站 A 产生 Cookie 信息并返回给浏览器。

（3）用户登陆网站 A 成功后，可以正常请求网站 A。

（4）用户未退出网站 A 之前，在同一浏览器中，打开一个 TAB 访问网站 B。

（5）网站 B 看到有人方式后，他会返回一些攻击性代码。

（6）浏览器在接受到这些攻击性代码后，促使用户不知情的情况下浏览器携带 Cookie（包括 sessionId）信息，请求网站 A。这种请求有可能更新密码，添加用户什么的操作。

从上面 CSRF 攻击原理可以看出，要完成一次 CSRF 攻击，需要被攻击者完成两个步骤：

(1) 登陆受信任网站 A，并在本地生成 COOKIE。
(2) 在不登出 A 的情况下，访问危险网站 B。

如果不满足以上两个条件中的一个，就不会受到 CSRF 的攻击，以下情况可能会导致 CSRF：

(1) 登录了一个网站后，打开一个 tab 页面并访问另外的网站。

(2) 关闭浏览器了后，本地的 Cookie 尚未过期，你上次的会话还没有已经结束。（事实上，关闭浏览器不能结束一个会话，但大多数人都会错误的认为关闭浏览器就等于退出登录/结束会话了……）

解决办法：就是在表单中添加 from.csrf_token。

11、Flask(name)中的name可以传入哪些值?

可以传入的参数：

(1) 字符串：‘hello’, 但是‘abc’,不行，因为 abc 是 python 内置的模块

(2) __name__，约定俗成

不可以传入的参数：

(1) python 内置的模块，re,urllib,abc 等

(2) 数字

Django

1、Django 创建项目的命令?

django-admin startproject 项目名称

python manage.py startapp 应用 app 名

2、Django 创建项目后，项目文件夹下的组成部分（对 mvt 的理解）？

项目文件夹下的组成部分：

manage.py 是项目运行的入口，指定配置文件路径。

与项目同名的目录，包含项目的配置文件。

___init.py 是一个空文件，作用是这个目录可以被当作包使用。

settings.py 是项目的整体配置文件。

urls.py 是项目的 URL 配置文件。

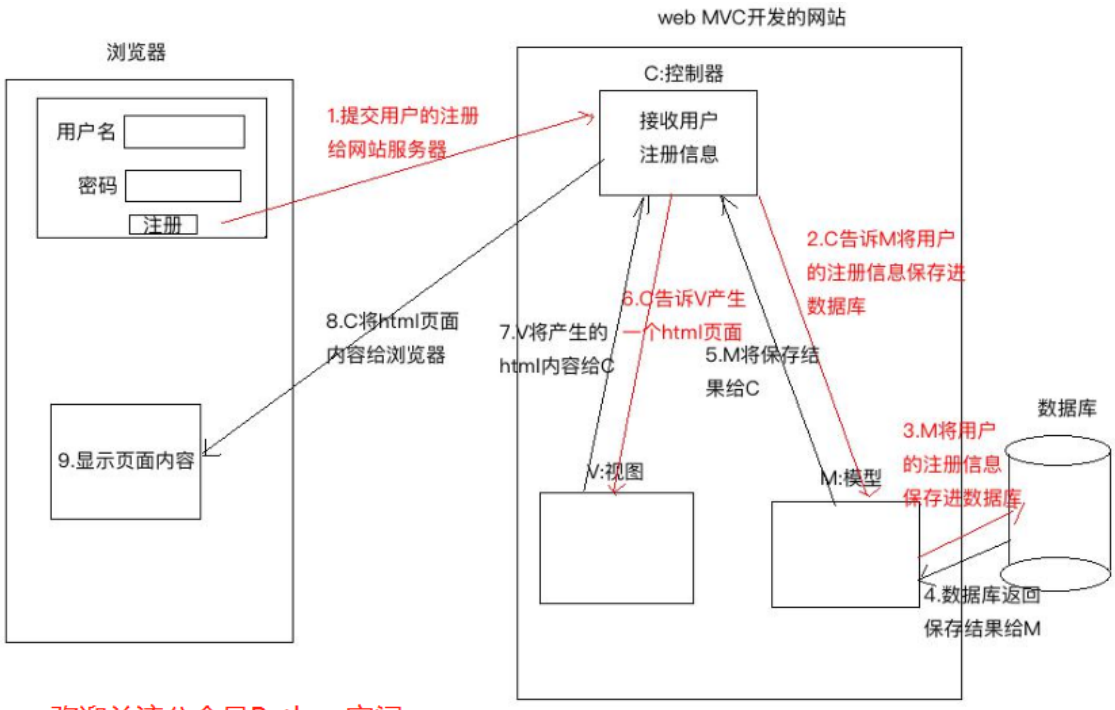
wsgi.py 是项目与 WSGI 兼容的 Web 服务器。

3、对 MVC,MVT 解读的理解?

M：Model，模型，和数据库进行交互

V：View，视图，负责产生 Html 页面

C：Controller，控制器，接收请求，进行处理，与 M 和 V 进行交互，返回应答。



1 用户点击注按钮，将要注册的信息发送给网站服务器。

2 Controller 控制器接收到用户的注册信息，Controller 会告诉 Model 层将用户的注册信息保存到数据库

3 Model 层将用户的注册信息保存到数据库

4 数据保存之后将保存的结果返回给 Model 模型

5 Model 层将保存的结果返回给 Controller 控制器

6 Controller 控制器收到保存的结果之后，或告诉 View 视图，view 视图产生一个 html 页面

7 View 将产生的 Html 页面的内容给了 Controller 控制器

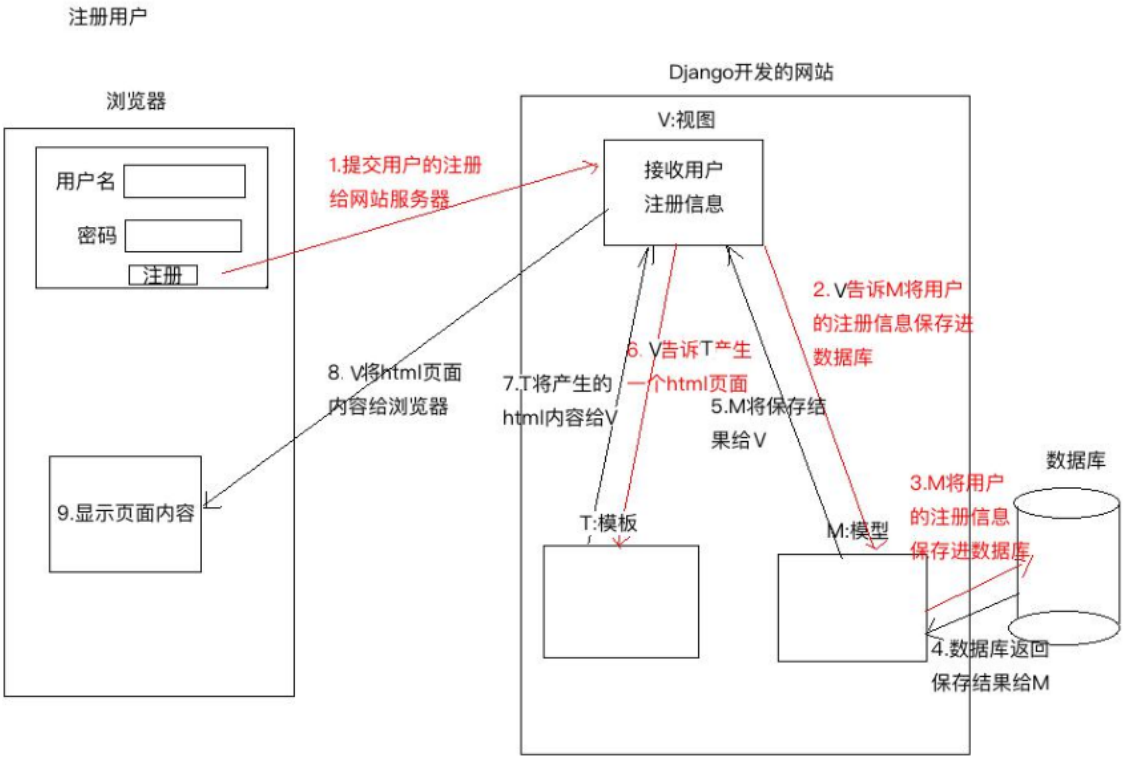
8 Controller 将 Html 页面的内容返回给浏览器

9 浏览器接受到服务器 Controller 返回的 Html 页面进行解析展示

M：Model，模型，和 MVC 中的 M 功能相同，和数据库进行交互

V：view，视图，和 MVC 中的 C 功能相同，接收请求，进行处理，与 M 和 T 进行交互，返回应答

T：Template，模板，和 MVC 中的 V 功能相同，产生 Html 页面



1 用户点击注册按钮，将要注册的内容发送给网站的服务器

2 View 视图，接收到用户发来的注册数据，View 告诉 Model 将用户的注册信息保存进数据库

3 Model 层将用户的注册信息保存到数据库中

4 数据库将保存的结果返回给 Model

5 Model 将保存的结果给 View 视图

6 View 视图告诉 Template 模板去产生一个 Html 页面

7 Template 生成 html 内容返回给 View 视图

8 View 将 html 页面内容返回给浏览器

9 浏览器拿到 view 返回的 html 页面内容进行解析，展示

4、Django 中 models 利用 ORM 对 Mysql 进行查表的语句（多个语句）

字段查询

all():返回模型类对应表格中的所有数据。

get():返回表格中满足条件的一条数据，如果查到多条数据，则抛异常：MultipleObjectsReturned，查询不到数据，则抛异常：DoesNotExist。

filter():参数写查询条件，返回满足条件 QuerySet 集合数据。

条件格式：

模型类属性名__条件名=值

注意：此处是模型类属性名，不是表中的字段名

关于 filter 具体案例如下：

判等 exact

```
BookInfo.object.filter(id=1)
BookInfo.object.filter(id__exact=1)此处的__exact 可以省略
```

模糊查询 like

例：查询书名包含'传'的图书。contains

```
contains BookInfo.objects.filter(btitle__contains='传')
```

空查询 where 字段名 isnull

```
BookInfo.objects.filter(btitle__isnull=False)
```

范围查询 where id in (1，3，5)

```
BookInfo.objects.filter(id__in=[1，3，5])
```

比较查询 gt lt(less than) gte(equal) lte

```
BookInfo.objects.filter(id__gte=3)
```

日期查询

```
BookInfo.objects.filter(bpub_date__year = 1980)
BookInfo.objects.filter(bpub_date__gt = date(1980，1，1))
```

exclude:返回不满足条件的数据。

```
BookInfo.objects.exclude(id=3)
```

F 对象

作用：用于类属性之间的比较条件。

```
from django.db.models import F
例：where bread > bcomment BookInfo.objects.filter(bread__gt =F(‘bcomment’))
例：BookInfo.objects.filter(bread__gt=F(‘bcomment’)*2)
```

Q 对象

作用：用于查询时的逻辑条件。可以对 Q 对象进行&|~操作。

```
from django.db.models import Q
BookInfo.objects.filter(id__gt=3， bread__gt=30)
BooInfo.objects.filter(Q(id__gt=3) & Q(bread__gt=3))
例：BookInfo.objects.filter(Q(id__gt=3) | Q(bread__gt=30))
例：BookInfo.objects.filter(~Q(id=3))
```

order_by 返回 QuerySet

作用：对查询结果进行排序。

```
例： BookInfo.objects.all().order_by('id')
例： BookInfo.objects.all().order_by('-id')
例：BookInfo.objects.filter(id__gt=3).order_by('-bread')
```

聚合函数

作用：对查询结果进行聚合操作。

```
sum count max min avg
```

aggregate：调用这个函数来使用聚合。

```
from django.db.models import Sum，Count，Max，Min，Avg
例：BookInfo.objects.aggregate(Count('id'))
```

{'id__count': 5} 注意返回值类型及键名

```
例：BookInfo.objects.aggregate(Sum(‘bread’))
```

{‘bread__sum’:120} 注意返回值类型及键名

count 函数

作用：统计满足条件数据的数目。

例：统计所有图书的数目。

```
BookInfo.objects.all().count()
```

例：统计 id 大于 3 的所有图书的数目。

```
BookInfo.objects.filter(id__gt = 3).count()
```

模型类关系

一对多关系

例：图书类-英雄类

models.ForeignKey() 定义在多的类中。

2）多对多关系

例：新闻类-新闻类型类

models.ManyToManyField() 定义在哪个类中都可以。

3）一对一关系

例：员工基本信息类-员工详细信息类

models.OneToOneField() 定义在哪个类中都可以。

5、django 中间件的使用?

Django 在中间件中预置了六个方法，这六个方法的区别在于不同的阶段执行，对输入或输出进行干预，方法如下：

1.初始化：无需任何参数，服务器响应第一个请求的时候调用一次，用于确定是否启用当前中间件。

```
def __init__():
    pass
```

2.处理请求前：在每个请求上调用，返回 None 或 HttpResponse 对象。

```
def process_request(request):
    pass
```

3.处理视图前：在每个请求上调用，返回 None 或 HttpResponse 对象。

```
def process_view(request, view_func, view_args, view_kwargs):
    pass
```

4.处理模板响应前：在每个请求上调用，返回实现了 render 方法的响应对象。

```
def process_template_response(request, response):
    pass
```

5.处理响应后：所有响应返回浏览器之前被调用，在每个请求上调用，返回 HttpResponse 对象。

```
def process_response(request, response):
    pass
```

6.异常处理：当视图抛出异常时调用，在每个请求上调用，返回一个 HttpResponse 对象。

```
def process_exception(request,exception):
    pass
```

6、Django 项目的优化?

这个答案 web 通用

(1) 优化数据库查询

- 一次提供所有数据
- 仅提供相关的数据

(2) 代码优化

- 简化代码
- 更新或替代第三方软件包
- 重构代码

7、谈一下你对 uWSGI 和 nginx 的理解?

uWSGI

uWSGI 是一个 Web 服务器，它实现了 WSGI 协议、uwsgi、http 等协议。Nginx 中 HttpUwsgiModule 的作用是与 uWSGI 服务器进行交换。WSGI 是一种 Web 服务器网关接口。它是一个 Web 服务器（如 nginx，uWSGI 等服务器）与 web 应用（如用 Flask 框架写的程序）通信的一种规范。

要注意 WSGI / uwsgi / uWSGI 这三个概念的区分。

(1) WSGI 是一种通信协议。

(2) uwsgi 是一种线路协议而不是通信协议，在此常用于在 uWSGI 服务器与其他网络服务器的数据通信。

(3) uWSGI 是实现了 uwsgi 和 WSGI 两种协议的 Web 服务器。

ngix

nginx 是一个开源的高性能的 HTTP 服务器和反向代理：

(1) 作为 web 服务器，它处理静态文件和索引文件效果非常高；

(2) 它的设计非常注重效率，最大支持 5 万个并发连接，但只占用很少的内存空间；

(3) 稳定性高，配置简洁；

(4) 强大的反向代理和负载均衡功能，平衡集群中各个服务器的负载压力应用。

8、说说 nginx 和 uWISG 服务器之间如何配合工作的?

首先浏览器发起 http 请求到 nginx 服务器，Nginx 根据接收到请求包，进行 url 分析，判断访问的资源类型。

如果是静态资源，直接读取静态资源返回给浏览器。

如果请求的是动态资源就转交给 uwsgi 服务器，uwsgi 服务器根据自身的 uwsgi 和 WSGI 协议，找到对应的 Django 框架，Django 框架下的应用进行逻辑处理后，将返回值发送到 uwsgi 服务器，然后 uwsgi 服务器再返回给 nginx，最后 nginx 将返回值返回给浏览器进行渲染显示给用户。

9、django 开发中数据库做过什么优化?

(1) 设计表时，尽量少使用外键，因为外键约束会影响插入和删除性能

(2) 使用缓存，减少对数据库的访问

(3) 在 orm 框架下设置表时，能用 varchar 确定字段长度时，就别用 text

(4) 可以给搜索频率高的字段属性，在定义时创建索引

(5) Django orm 框架下的 Querysets 本来就有缓存的

(6) 如果一个页面需要多次连接数据库，最好一次性取出所有需要的数据，减少对数据库的查询次数

(7) 若页面只需要数据库里某一个两个字段时，可以用 QuerySet.values()

(8) 在模板标签里使用 with 标签可以缓存 Qset 的查询结果

10、验证码过期时间怎么设置?

将验证码保存到数据库或 session，设置过期时间为 1 分钟，然后页面设置一个倒计时(一般是前端 js 实现 这个计时)的展示，一分钟过后再次点击获取新的信息。

11、Python 中 Django、Flask、Tornado 三大框架各自的应用场景?

Django

主要是用来搞快速开发的，他的亮点就是快速开发，节约成本，正常的并发量不过 10000，如果要实现高并发的话，就要对 Django 进行二次开发，比如把整个笨重的框架给拆掉，自己写 socket 实现 http 的通信，底层用纯 c/c++写提升效率，ORM 框架给干掉，自己编写封装与数据库交互的框架，因为啥呢，ORM 虽然面向对象来操作数据库，但是它的效率很低，使用外键来联系表与表之间的查询

Flask

轻量级，主要是用来写接口的一个框架，实现前后端分离，提升开发效率，Flask 本身相当于一个内核，其他几乎所有的功能都要用到扩展（邮件扩展 Flask-Mail，用户认证 Flask-Login），都需要用第三方的扩展来实现。比如可以用 Flask-extension 加入 ORM、窗体验证工具，文件上传、身份验证等。Flask 没有默认使用的数据库，你可以选择 MySQL，也可以用 NoSQL。

其 WSGI 工具箱采用 Werkzeug（路由模块），模板引擎则使用 Jinja2。这两个也是 Flask 框架的核心。Python 最出名的框架要数 Django，此外还有 Flask、Tornado 等框架。虽然 Flask 不是最出名的框架，但是 Flask 应该算是最灵活的框架之一，这也是 Flask 受到广大开发者喜爱的原因。

Tornado

Tornado 是一种 Web 服务器软件的开源版本。

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。得利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，因此 Tornado 是实时 Web 服务的一个 理想框架。

12、django 如何提升性能?

对一个后端开发程序员来说，提升性能指标主要有两个一个是并发数，另一个是响应时间网站性能的优化一般包括 web 前端性能优化，应用服务器性能优化，存储服务器优化。

对前端的优化主要有：

(1) 减少 http 请求，减少数据库的访问量，比如使用雪碧图。

(2)使用浏览器缓存，将一些常用的 css，js，logo 图标，这些静态资源缓存到本地浏览器，通过设置 http 头中的 cache-control 和 expires 的属性，可设定浏览器缓存，缓存时间可以自定义。

(3) 对 html，css，javascript 文件进行压缩，减少网络的通信量。

对我个人而言，我做的优化主要是以下三个方面：

(1)合理的使用缓存技术，对一些常用到的动态数据，比如首页做一个缓存，或者某些常用的数据做个缓存，设置一定得过期时间，这样减少了对数据库的压力，提升网站性能。

(2) 使用 celery 消息队列，将耗时的操作扔到队列里，让 worker 去监听队列里的任务，实现异步操作，比如发邮件，发短信。

(3) 就是代码上的一些优化，补充：nginx 部署项目也是项目优化，可以配置合适的配置参数，提升效率，增加并发量。

(4) 如果太多考虑安全因素，服务器磁盘用固态硬盘读写，远远大于机械硬盘，这个技术现在没有普及，主要是固态硬盘技术上还不是完全成熟， 相信以后会大量普及。

(5) 另外还可以搭建服务器集群，将并发访问请求，分散到多台服务器上处理。

(6) 最后就是运维工作人员的一些性能优化技术了。

13、什么是 restful api，谈谈你的理解?

REST:Representational State Transfer 的缩写，翻译："具象状态传输"。一般解释为"表现层状态转换"。

REST 是设计风格而不是标准。是指客户端和服务器的交互形式。我们需要关注的重点是如何设计 REST 风格的网络接口。

REST 的特点：

(1) 具象的。一般指表现层，要表现的对象就是资源。比如，客户端访问服务器，获取的数据就是资源。比如文字、图片、音视频等。

(2) 表现：资源的表现形式。txt 格式、html 格式、json 格式、jpg 格式等。浏览器通过 URL 确定资源的位置，但是需要在 HTTP 请求头中，用 Accept 和 Content-Type 字段指定，这两个字段是对资源表现的描述。

(3) 状态转换：客户端和服务器交互的过程。在这个过程中，一定会有数据和状态的转化，这种转化叫做状态转换。其中，GET 表示获取资源，POST 表示新建资源，PUT 表示更新资源，DELETE 表示删除资源。HTTP 协议中最常用的就是这四种操作方式。

RESTful 架构：

(1) 每个 URL 代表一种资源

(2) 客户端和服务器之间，传递这种资源的某种表现层

(3) 客户端通过四个 http 动词，对服务器资源进行操作，实现表现层状态转换

14、启动 Django 服务的方法?

runserver 方法是调试 Django 时经常用到的运行方式，它使用 Django 自带的 WSGI Server 运行，主要在测试和开发中使用，并且 runserver 开启的方式也是单进程 。

15、怎样测试 django 框架中的代码?

在单元测试方面，Django 继承 python 的 unittest.TestCase 实现了自己的 django.test.TestCase，编写测试用例通常从这里开始。测试代码通常位于 app 的 tests.py 文件中(也可以在 models.py 中编写，一般不建议）。在 Django 生成的 depotapp 中，已经包含了这个文件，并且其中包含了一个测试

用例的样例：

```
python manage.py test：执行所有的测试用例
python manage.py test app_name， 执行该 app 的所有测试用例
python manage.py test app_name.case_name: 执行指定的测试用例
```

一些测试工具：unittest 或者 pytest

16、有过部署经验？用的什么技术？可以满足多少压力?

(1) 有部署经验，在阿里云服务器上部署的

(2) 技术有：nginx + uwsgi 的方式来部署 Django 项目

(3) 无标准答案（例：压力测试一两千）

17、Django 中哪里用到了线程?哪里用到了协程?哪里用到了进程?

(1) Django 中耗时的任务用一个进程或者线程来执行，比如发邮件，使用 celery。

(2) 部署 django 项目的时候，配置文件中设置了进程和协程的相关配置。

18、django 关闭浏览器，怎样清除 cookies 和 session?

设置 Cookie

```
1. def cookie_set(request):
2. response = HttpResponse("<h1>设置 Cookie，请查看响应报文头</h1>")
3. response.set_cookie('h1', 'hello django')
4. return response
```

读取 Cookie

```
1. def cookie_get(request):
2. response = HttpResponse("读取 Cookie，数据如下：<br>")
3. if request.COOKIES.has_key('h1'):
4. response.write('<h1>' + request.COOKIES['h1'] + '</h1>')
5. return response
```

以键值对的格式写会话。

```
1. request.session['键']=值
```

根据键读取值。

```
1. request.session.get('键',默认值)
```

清除所有会话，在存储中删除值部分。

```
1. request.session.clear()
```

清除会话数据，在存储中删除会话的整条数据。

```
1. request.session.flush()
```

删除会话中的指定键及值，在存储中只删除某个键及对应的值。

```
1. del request.session['键']
```

设置会话的超时时间，如果没有指定过期时间则两个星期后过期。
如果 value 是一个整数，会话将在 value 秒没有活动后过期。
如果 value 为 0，那么用户会话的 Cookie 将在用户的浏览器关闭时过期。
如果 value 为 None，那么会话永不过期。

```
1. request.session.set_expiry(value)
```

Session 依赖于 Cookie，如果浏览器不能保存 cookie 那么 session 就失效了。因为它需要浏览器的 cookie 值去 session 里做对比。session 就是用来在服务器端保存用户的会话状态。

cookie 可以有过期时间，这样浏览器就知道什么时候可以删除 cookie 了。 如果 cookie 没有设置过期时间，当用户关闭浏览器的时候，cookie 就自动过期了。你可以改变 SESSION_EXPIRE_AT_BROWSER_CLOSE 的设置来控制 session 框架的这一行为。缺省情况下，SESSION_EXPIRE_AT_BROWSER_CLOSE 设置为 False ，这样，会话 cookie 可以在用户浏览器中保持有效达 SESSION_COOKIE_AGE 秒（缺省设置是两周，即 1，209，600 秒）如果你不想用户每次打开浏览器都必须重新登陆的话，用这个参数来帮你。如果 SESSION_EXPIRE_AT_BROWSER_CLOSE 设置为 True，当浏览器关闭时，Django 会使 cookie 失效。

SESSION_COOKIE_AGE：设置 cookie 在浏览器中存活的时间。

19、代码优化从哪些方面考虑？有什么想法？

1、优化算法时间

算法的时间复杂度对程序的执行效率影响最大，在 Python 中可以通过选择合适的数据结构来优化时间复杂度，如 list 和 set 查找某一个元素的时间复杂度分别是 O(n)和 O(1)。不同的场景有不同的优化方式，总得来说，一般有分治，分支界限，贪心，动态规划等思想。

2、循环优化

每种编程语言都会强调需要优化循环。当使用 Python 的时候，你可以依靠大量的技巧使得循环运行得更快。然而，开发者经常漏掉的一个方法是：

避免在一个循环中使用点操作。每一次你调用方法 str.upper，Python 都会求该方法的值。然而，如果你用一个变量代替求得的值，值就变成了已知的，Python 就可以更快地执行任务。优化循环的关键，是要减少 Python 在循环内部执行的工作量，因为 Python 原生的解释器在那种情况下，真的会减缓执行的速度。（注意：优化循环的方法有很多，这只是其中的一个。例如，许多程序员都会说，列表推导是在循环中提高执行速度的最好方式。这里的关键是，优化循环是程序取得更高的执行速度的更好方式之一。）

3、函数选择

在循环的时候使用 xrange 而不是 range；使用 xrange 可以节省大量的系统内存，因为 xrange() 在序列中每次调用只产生一个整数元素。而 range()将直接返回完整的元素列表，用于循环时会有不必要的开销。在 python3 中 xrange 不再存在，里面 range 提供一个可以遍历任意长度的范围的 iterator。

4、并行编程

因为 GIL 的存在，Python 很难充分利用多核 CPU 的优势。但是，可以通过内置的模 multiprocessing 实现下面几种并行模式：

多进程：对于 CPU 密集型的程序，可以使用 multiprocessing 的 Process，Pool 等封装好的类，通过多进程的方式实现并行计算。但是因为进程中的通信成本比较大，对于进程之间需要大量数据交互的程序效率未必有大的提高。

多线程：对于 IO 密集型的程序，multiprocessing.dummy 模块使用 multiprocessing 的接口封装 threading，使得多线程编程也变得非常轻松(比如可以使用 Pool 的 map 接口，简洁高效)。

布式：multiprocessing 中的 Managers 类提供了可以在不同进程之共享数据的方式，可以在此基础上开发出分布式的程序。

不同的业务场景可以选择其中的一种或几种的组合实现程序性能的优化。

5、使用性能分析工具

除了上面在 ipython 使用到的 timeit 模块，还有 cProfile。cProfile 的使用方式也非常简单：python-mcProfilefilename.py，filename.py 是要运行程序的文件名，可以在标准输出中看到每一个函数被调用的次数和运行的时间，从而找到程序的性能瓶颈，然后可以有针对性地优化。

6、set 的用法

set 的 union，intersection，difference 操作要比 list 的迭代要快。因此如果涉及到求 list 交集，并集或者差的问题可以转换为 set 来操作。

7、PyPy

PyPy 是用 RPython(CPython 的子集)实现的 Python，根据官网的基准测试数据，它比 CPython 实现的 Python 要快 6 倍以上。快的原因是使用了 Just-in-Time(JIT)编译器，即动态编译器，与静态编译器(如 gcc，javac 等)不同，它是利用程序运行的过程的数据进行优化。由于历史原因，目前 pypy 中还保留着 GIL，不过正在进行的 STM 项目试图将 PyPy 变成没有 GIL 的 Python。如果 python 程序中含有 C 扩展(非 cffi 的方式)，JIT 的优化效果会大打折扣，甚至比 CPython 慢（比 Numpy）。

所以在 PyPy 中最好用纯 Python 或使用 cffi 扩展。

20、Django 中间件是如何使用的?

中间件不用继承自任何类（可以继承 object），下面一个中间件大概的样子：

```
class CommonMiddleware(object):
    def process_request(self， request):
        return None

def process_response(self， request， response):
    return response
```

还有 process_view， process_exception 和 process_template_response 函数。

(1) 初始化：无需任何参数，服务器响应第一个请求的时候调用一次，用于确定是否启用当前中间件。

```
def __init__(self):
    pass
```

(2) 处理请求前：在每个请求上，request 对象产生之后，url 匹配之前调用，返回 None 或 HttpResponse 对象。

```
def process_request(self， request):
    pass
```

(3) 处理视图前：在每个请求上，url 匹配之后，视图函数调用之前调用，返回 None 或HttpResponse 对象。

```
def process_view(self， request， view_func， *view_args，**view_kwargs):
    pass
```

(4) 处理响应后：视图函数调用之后，所有响应返回浏览器之前被调用，在每个请求上调用，返回 HttpResponse 对象。

```
def process_response(self， request， response):
    pass
```

(5) 异常处理：当视图抛出异常时调用，在每个请求上调用，返回一个 HttpResponse 对象。

```
def process_exception(self， request，exception):
    pass
```

21、ngnix 的正向代理与反向代理?

web 开发中，部署方式大致类似。简单来说，使用 Nginx 主要是为了实现分流、转发、负载均衡，以及分担服务器的压力。Nginx 部署简单，内存消耗少，成本低。Nginx 既可以做正向代理，也可以做反向代理。

正向代理：请求经过代理服务器从局域网发出，然后到达互联网上的服务器。

特点：服务端并不知道真正的客户端是谁。

反向代理：请求从互联网发出，先进入代理服务器，再转发给局域网内的服务器。

特点：客户端并不知道真正的服务端是谁。

区别：正向代理的对象是客户端。反向代理的对象是服务端。

22、简述 Django 下的（内建的）缓存机制?

一个动态网站的基本权衡点就是，它是动态的。每次用户请求页面，服务器会重新计算。从开销处理的角度来看，这比你读取一个现成的标准文件的代价要昂贵的多。

这就是需要缓存的地方。

Django 自带了一个健壮的缓存系统来保存动态页面这样避免对于每次请求都重新计算。方便起见，Django 提供了不同级别的缓存粒度：可以缓存特定视图的输出、可以仅仅缓存那些很难生产出来的部分、或者可以缓存整个网站 Django 也能很好的配合那些“下游”缓存， 比如 Squid 和基于浏览器的缓存。这里有一些缓存不必要直接去控制但是可以提供线索， (via HTTPheaders)关于网站哪些部分需要缓存和如何缓存。

设置缓存：

缓存系统需要一些设置才能使用。 也就是说，你必须告诉他你要把数据缓存在哪里- 是数据库中，文件系统或者直接在内存中。这个决定很重要，因为它会影响你的缓存性能，是的，一些缓存类型要比其他的缓存类型更快速。

你的缓存配置是通过 setting 文件的 CACHES 配置来实现的。这里有 CACHES 所有可配置的变量值。

23、Django HTTP 请求的处理流程?

Django 和其他 Web 框架的 HTTP 处理的流程大致相同，Django 处理一个 Request 的过程是首先通过中间件，然后再通过默认的 URL 方式进行的。我们可以在 Middleware 这个地方把所有 Request 拦截住，用我们自己的方式完成处理以后直接返回 Response。

(1) 加载配置

Django 的配置都在 “Project/settings.py” 中定义，可以是 Django 的配置，也可以是自定义的配置，并且都通过 django.conf.settings 访问，非常方便。

(2) 启动

最核心动作的是通过 django.core.management.commands.runfcgi 的 Command 来启动，它运行 django.core.servers.fastcgi 中的 runfastcgi，runfastcgi 使用了 flup 的 WSGIServer 来启动 fastcgi 。而 WSGIServer 中携带了 django.core.handlers.wsgi 的 WSGIHandler 类的一个实例，通过 WSGIHandler 来处理由 Web 服务器（比如 Apache，Lighttpd 等）传过来的请求，此时才是真正进入 Django 的世界。

(3) 处理 Request

当有 HTTP 请求来时，WSGIHandler 就开始工作了，它从 BaseHandler 继承而来。WSGIHandler 为每个请求创建一个 WSGIRequest 实例，而 WSGIRequest 是从 http.HttpRequest 继承而来。接下来就开始创建 Response 了。

(4) 创建 Response

BaseHandler 的 get_response 方法就是根据 request 创建 response，而具体生成 response 的动作就是执行 urls.py 中对应的 view 函数了，这也是 Django 可以处理“友好 URL ”的关键步骤，每个这样的函数都要返回一个 Response 实例。此时一般的做法是通过 loader 加载 template 并生成页面内容，其中重要的就是通过 ORM 技术从数据库中取出数据，并渲染到 Template 中，从而生成具体的页面了。

(5) 处理 Response

Django 返回 Response 给 flup，flup 就取出 Response 的内容返回给 Web 服务器，由后者返回给浏览器。

总之，Django 在 fastcgi 中主要做了两件事：处理 Request 和创建 Response，而它们对应的核心就是“ urls 分析”、“模板技术”和“ ORM 技术”。

24、Django 里 QuerySet 的 get 和 filter 方法的区别?

(1) 输入参数

get 的参数只能是 model 中定义的那些字段，只支持严格匹配。filter 的参数可以是字段，也可以是扩展的 where 查询关键字，如 in，like 等。

(2) 返回值

get 返回值是一个定义的 model 对象。filter 返回值是一个新的 QuerySet 对象，然后可以对 QuerySet 在进行查询返回新的 QuerySet 对象，支持链式操作，QuerySet 一个集合对象，可使用迭代或者遍历，切片等，但是不等于 list 类型(使用一定要注意)。

(3) 异常

get 只有一条记录返回的时候才正常，也就说明 get 的查询字段必须是主键或者唯一约束的字段。当返回多条记录或者是没有找到记录的时候都会抛出异常 filter 有没有匹配的记录都可以

25、跨域请求问题 django 怎么解决的?

- 启用中间件
- post 请求
- 验证码
- 表单中添加 csrf_token 标签

26、Django 对数据查询结果排序怎么做，降序怎么做，查询大于某个字段怎么做?

- 排序使用 order_by()
- 降序需要在排序字段名前加-

- 查询字段大于某个值：使用 filter(字段名_gt=值)

27、Django 重定向你是如何实现的？用的什么状态码？

- 使用 HttpResponseRedirect
- redirect 和 reverse
- 状态码：302,301

28、生成迁移文件和执行迁移文件的命令是什么？

python manage.py makemigrations

python manage.py migrate

29、关系型数据库的关系包括哪些类型？

- ForeignKey：一对多，将字段定义在多的一端中。
- ManyToManyField：多对对：将字段定义在两端中。
- OneToOneField：一对一，将字段定义在任意一端中。

30、查询集返回列表的过滤器有哪些？

- all()：返回所有的数据
- filter()：返回满足条件的数据
- exclude()：返回满足条件之外的数据，相当于 sql 语句中 where 部分的 not 关键字
- order_by()：排序

31、Django 本身提供了 runserver，为什么不能用来部署？

runserver 方法是调试 Django 时经常用到的运行方式，它使用 Django 自带的 WSGI Server 运行，主要在测试和开发中使用，并且 runserver 开启的方式也是单进程 。

uWSGI 是一个 Web 服务器，它实现了 WSGI 协议、uwsgi、http 等协议。注意 uwsgi 是一种通信协议，而 uWSGI 是实现 uwsgi 协议和 WSGI 协议的 Web 服务器。uWSGI 具有超快的性能、低内存占用和多 app 管理等优点，并且搭配着 Nginx 就是一个生产环境了，能够将用户访问请求与应用 app 隔离开，实现真正的部署。相比来讲，支持的并发量更高，方便管理多进程，发挥多核的优势，提升性能。

32、HttpRequest 和 HttpResponse 是什么？干嘛用的？

HttpRequest 是 django 接受用户发送多来的请求报文后，将报文封装到 HttpRequest 对象中去。

HttpResponse 返回的是一个应答的数据报文。render 内部已经封装好了 HttpResponse 类。

视图的第一个参数必须是 HttpRequest 对象，两点原因：表面上说，他是处理 web 请求的，所以必须是请求对象，根本上说，他是基于请求的一种 web 框架，所以，必须是请求对象。

因为 view 处理的是一个 request 对象，请求的所有属性我们都可以根据对象属性的查看方法来获取具体的信息：格式：request.属性

```
request.path 请求页面的路径，不包含域名
request.get_full_path 获取带参数的路径
request.method 页面的请求方式
request.GET GET 请求方式的数据
request.POST POST 请求方式的数据
request.COOKIES 获取 cookie
request.session 获取 session
request.FILES 上传图片（请求页面有 enctype="multipart/form-data"属性时 FILES 才有数据。？a=10 的键和值时怎么产生的，键是开发人员在编写代码时确定下来的，值时根据数据生成或者用户填写的，总之是不确定的。
```

403 错误：表示资源不可用，服务器理解客户的请求，但是拒绝处理它，通常由于服务器上文件和目录的权限设置导致的 web 访问错误。如何解决：

(1) 把中间件注释。

(2) 在表单内部添加{% scrf_token %}

request.GET.get()取值时如果一键多值情况，get 是覆盖的方式获取的。getlist（）可以获取多值。

在一个有键无值的情况下，该键名 c 的值返回空。有键无值：c: getlist 返回的是列表，空列表在无键无值也没有默认值的情况下，返回的是 None 无键无值：e:None

```
HttpResponse 常见属性：
content： 表示返回的内容
charset: 表示 response 采用的编码字符集，默认是 utf-8
status_code:返回的 HTTP 响应状态码 3XX 是对请求继续进一步处理，常见的是重定向。
```

常见方法：

```
init:创建 httpResponse 对象完成返回内容的初始化
set_cookie：设置 Cookie 信息：格式：
set_cookies('key','value',max_age=None,expires=None)
max_age 是一个整数，表示指定秒数后过期，expires 指定过期时间，默认两个星期后过期。
write 向响应体中写数据
```

应答对象：

方式一：render(request,"index.html") 返回一个模板

render(request,"index.html", context) 返回一个携带动态数据的页面

方式二：render_to_response("index.html") 返回一个模板页面

方式三：redirect("/") 重定向

方式四：HttpResponseRdeirect("/") 实现页面跳转功能

方式五：HttpResponse（"itcast1.0")在返回到额页面中添加字符串内容

方式六：HttpResponseJson() 返回的页面中添加字符串内容。

JsonResponse 创建对象时候接收字典作为参数，返回的对象是一个 json 对象。

能接收 Json 格式数据的场景，都需要使用 view 的 JsonResponse 对象返回一个 json 格式数据

ajax 的使用场景，页面局部刷新功能。ajax 接收 Json 格式的数据。

在返回的应答报文中，可以看到 JsonResponse 应答的 content-Type 内容是 application/json

ajax 实现网页局部刷新功能：ajax 的 get()方法获取请求数据 ajax 的 each()方法遍历输出这些数据

33、Django 日志管理

配置好之后：

```
import logging
logger=logging.getLogger(__name__) # 为 loggers 中定义的名称
logger.info("some info ...)
```

可用函数有：logger.debug() logger.info() logger.warning() logger.error()

Django 文件管理：对于 jdango 老说，项目中的 css，js,图片都属于静态文件，我们一般会将静态文件放到一个单独的目录中，以方便管理，在 html 页面调用时，也需要指定静态文件的路径。静态文件可以放在项目根目录下，也可以放在应用的目录下，由于这些静态文件在项目中是通用的，所以推荐放在项目的根目录下。

在生产中，只要和静态文件相关的，所有访问，基本上没有 django 什么事，一般都是由 nignx 软件代劳了，为什么？因为 nginx 就是干这个的。

爬虫（共 35 题）

常见问题

1、爬取数据后使用哪个数据库存储数据的，为什么？

MongoDB 是使用比较多的数据库，这里以 MongoDB 为例，大家需要结合自己真实开发环境回答。

原因：

1.与关系型数据库相比，MongoDB 的优点如下。

(1) 弱一致性（最终一致），更能保证用户的访问速度

举例来说，在传统的关系型数据库中，一个 COUNT 类型的操作会锁定数据集，这样可以保证得到“当前”情况下的较精确值。这在某些情况下，例 如通过 ATM 查看账户信息的时候很重要，但对于 Wordnik 来说，数据是不断更新和增长的，这种“较精确”的保证几乎没有任何意义，反而会产生很大的延 迟。他们需要的是一个“大约”的数字以及更快的处理速度。

但某些情况下 MongoDB 会锁住数据库。如果此时正有数百个请求，则它们会堆积起来，造成许多问题。我们使用了下面的优化方式来避免锁定。

每次更新前，我们会先查询记录。查询操作会将对象放入内存，于是更新则会尽可能的迅速。在主/从部署方案中，从节点可以使用“-pretouch”参数运行，这也可以得到相同的效果。

使用多个 mongod 进程。我们根据访问模式将数据库拆分成多个进程。

(2) 文档结构的存储方式，能够更便捷的获取数据。

对于一个层级式的数据结构来说，如果要将这样的数据使用扁平式的，表状的结构来保存数据，这无论是在查询还是获取数据时都十分困难。

(3) 内置 GridFS，支持大容量的存储。

GridFS 是一个出色的分布式文件系统，可以支持海量的数据存储。内置了 GridFS 了 MongoDB，能够满足对大数据集的快速范围查询。

(4) 内置 Sharding。

提供基于 Range 的 Auto Sharding 机制：一个 collection 可按照记录的范围，分成若干个段，切分到不同的Shard上。Shards可以和复制结合，配合Replica sets能够实现Sharding+fail-over，不同的 Shard 之间可以负载均衡。查询是对 客户端是透明的。客户端执行查询，统计，MapReduce 等操作，这些会被 MongoDB 自动路由到后端的数据节点。这让我们关注于自己的业务，适当的时候可以无痛的升级。MongoDB 的 Sharding 设计能力较大可支持约 20 petabytes，足以支撑一般应用。

这可以保证 MongoDB 运行在便宜的 PC 服务器集群上。PC 集群扩充起来非常方便并且成本很低，避免了“sharding”操作的复杂性和成本。

(5) 第三方支持丰富。(这是与其他的 NoSQL 相比，MongoDB 也具有的优势)

现在网络上的很多 NoSQL 开源数据库完全属于社区型的，没有官方支持，给使用者带来了很大的风险。而开源文档数据库 MongoDB 背后有商业公司 10gen 为其提供供商业培训和支持。而且 MongoDB 社区非常活跃，很多开发框架都迅速提供了对 MongDB 的支持。不少知名大公司和网站也在生产环境中使用MongoDB，越来越多的创新型企业转而使用 MongoDB 作为和 Django，RoR 来搭配的技术方案。

(6) 性能优越

在使用场合下，千万级别的文档对象，近 10G 的数据，对有索引的 ID 的查询不会比 mysql 慢，而对非索引字段的查询，则是全面胜出。 mysql 实际无法胜任大数据量下任意字段的查询，而 mongodb 的查询性能实在让我惊讶。写入性能同样很令人满意，同样写入百万级别的数据，mongodb 比我以前试用过的couchdb 要快得多，基本 10 分钟以下可以解决。补上一句，观察过程中 mongodb 都远算不上是 CPU 杀手。

2.Mongodb 与 redis 相比较

(1) mongodb 文件存储是 BSON 格式类似 JSON，或自定义的二进制格式。mongodb 与 redis 性能都很依赖内存的大小，mongodb 有丰富的数据表达、索引；最类似于关系数据库，支持丰富的查询语言，redis 数据丰富，较少的 IO ，这方面 mongodb 优势明显。

(2) mongodb 不支持事物，靠客户端自身保证，redis 支持事物，比较弱，仅能保证事物中的操作按顺序执行，这方面 redis 优于 mongodb。

(3) mongodb 对海量数据的访问效率提升，redis 较小数据量的性能及运算,这方面 mongodb 性能优于 redis.monbgodb 有mapredurce功能，提供数据分析，redis 没有，这方面 mongodb 优于 redis。

2、你用过的爬虫框架或者模块有哪些？谈谈他们的区别或者优缺点？

用过的：

(1) Python 自带：urllib、urllib2

urllib 和 urllib2 模块都做与请求 URL 相关的操作，但他们提供不同的功能。

urllib2：urllib2.urlopen 可以接受一个 Request 对象或者 url，（在接受 Request 对象时候，并以此可以来设置一个 URL 的 headers），urllib.urlopen 只接收一个 url。

urllib 有 urlencode,urllib2 没有，因此总是 urllib，urllib2 常会一起使用的原因。

(2) 第三方：requests

request 是一个 HTTP 库， 它只是用来，进行请求，对于 HTTP 请求，他是一个强大的库，下载，解析全部自己处理，灵活性更高，高并发与分布式部署也非常灵活，对于功能可以更好实现

(3) 框架： Scrapy

scrapy 是封装起来的框架，它包含了下载器，解析器，日志及异常处理，基于多线程，twisted 的方式处理，对于固定单个网站的爬取开发，有优势，但是对于多网站爬取，并发及分布式处理方面，不够灵活，不便调整与括展。

Scrapy 优点

scrapy 是异步的

采取可读性更强的 xpath 代替正则

强大的统计和 log 系统

同时在不同的 url 上爬行

支持 shell 方式，方便独立调试

写 middleware,方便写一些统一的过滤器

通过管道的方式存入数据库

Scrapy 缺点

基于 python 的爬虫框架，扩展性比较差

基于 twisted 框架，运行中的 exception 是不会干掉 reactor，并且异步框架出错后是不会停掉其他任务的，数据出错后难以察觉。

3、写爬虫是用多进程好？还是多线程好？ 为什么？

IO 密集型代码(文件处理、网络爬虫等)，多线程能够有效提升效率(单线程下有 IO 操作会进行 IO 等待，造成不必要的时间浪费，而开启多线程能在线程 A 等待时，自动切换到线程 B，可以不浪费 CPU 的资源，从而能提升程序执行效率)。在实际的数据采集过程中，既考虑网速和响应的问题，也需要考虑自身机器的硬件情况，来设置多进程或多线程。

4、常见的反爬虫和应对方法？

(1) 通过 Headers 反爬虫

从用户请求的 Headers 反爬虫是最常见的反爬虫策略。很多网站都会对 Headers 的 User-Agent 进行检测，还有一部分网站会对 Referer 进行检测（一些资源网站的防盗链就是检测 Referer）。如果遇到了这类反爬虫机制，可以直接在爬虫中添加 Headers，将浏览器的 User-Agent 复制到爬虫的 Headers 中；或者将 Referer 值修改为目标网站域名。对于检测 Headers 的反爬虫，在爬虫中修改或者添加 Headers 就能很好的绕过。

(2) 基于用户行为反爬虫

还有一部分网站是通过检测用户行为，例如同一 IP 短时间内多次访问同一页面，或者同一账户短时间内多次进行相同操作。

大多数网站都是前一种情况，对于这种情况，使用 IP 代理就可以解决。可以专门写一个爬虫，爬取网上公开的代理 ip，检测后全部保存起来。这样的代理 ip 爬虫经常会用到，最好自己准备一个。有了大量代理 ip 后可以每请求几次更换一个 ip，这在 requests 或者 urllib2 中很容易做到，这样就能很容易的绕过第一种反爬虫。

对于第二种情况，可以在每次请求后随机间隔几秒再进行下一次请求。有些有逻辑漏洞的网站，可以通过请求几次，退出登录，重新登录，继续请求来绕过同一账号短时间内不能多次进行相同请求的限制。

(3) 动态页面的反爬虫

上述的几种情况大多都是出现在静态页面，还有一部分网站，我们需要爬取的数据是通过 ajax 请求得到，或者通过 JavaScript 生成的。首先用 Fiddler 对网络请求进行分析。如果能够找到 ajax 请求，也能分析出具体的参数和响应的具体含义，我们就能采用上面的方法，直接利用 requests 或者 urllib2模拟 ajax 请求，对响应的 json 进行分析得到需要的数据。

能够直接模拟 ajax 请求获取数据固然是极好的，但是有些网站把 ajax 请求的所有参数全部加密了。我们根本没办法构造自己所需要的数据的请求。这种情况下就用 selenium+phantomJS，调用浏览器内核，并利用 phantomJS 执行 js 来模拟人为操作以及触发页面中的 js 脚本。从填写表单到点击按钮再到滚动页面，全部都可以模拟，不考虑具体的请求和响应过程，只是完完整整的把人浏览页面获取数据的过程模拟一遍。

用这套框架几乎能绕过大多数的反爬虫，因为它不是在伪装成浏览器来获取数据（上述的通过添加 Headers 一定程度上就是为了伪装成浏览器），它本身就是浏览器，phantomJS 就是一个没有界面的浏览器，只是操控这个浏览器的不是人。利 selenium+phantomJS 能干很多事情，例如识别点触式（12306）或者滑动式的验证码，对页面表单进行暴力破解等。

5、需要登录的网页，如何解决同时限制 ip，cookie,session（其中有一些是动态生成的）在不使用动态爬取的情况下?

(1) 解决限制 IP 可以使用代理 IP 地址池、服务器

(2)不适用动态爬取的情况下可以使用反编译 JS 文件获取相应的文件，或者换用其他平台（比如手机端）看看是否可以获取相应的 json 文件。

6、验证码的解决?

(1) 图形验证码

干扰、杂色不是特别多的图片可以使用开源库 Tesseract 进行识别，太过复杂的需要借助第三方打码平台。

(2) 滑块验证码

点击和拖动滑块验证码可以借助 selenium、无图形界面浏览器（chromedirver 或者 phantomjs）和 pillow 包来模拟人的点击和滑动操作，pillow 可以根据色差识别需要滑动的位置。

7、使用最多的数据库（Mysql，Mongodb，redis 等），对他们的理解?

(1) MySQL 数据库

开源免费的关系型数据库，需要实现创建数据库、数据表和表的字段，表与表之间可以进行关联（一对多、多对多），是持久化存储。

(2) Mongodb 数据库

非关系型数据库，数据库的三元素是，数据库、集合、文档，可以进行持久化存储，也可作为内存数据库，存储数据不需要事先设定格式，数据以键值对的形式存储。

(3) redis 数据库

非关系型数据库，使用前可以不用设置格式，以键值对的方式保存，文件格式相对自由，主要用与缓存数据库，也可以进行持久化存储。

8、写一个邮箱地址的正则表达式?

[A-Za-z0-9-*]+@[a-zA-Z0-9*-]+(.[a-zA-Z0-9_-]+)+$

9、如何定时启动你的爬虫项目?

(1) 最简单的方法：直接使用 Timer 类

```
1. import time
2. import os
3.
4. while True:
5. os.system("scrapy crawl News")
6. time.sleep(86400) #每隔一天运行一次 24*60*60=86400s
```

(2) 使用 sched

```
1. import sched
2. #初始化 sched 模块的 scheduler 类
3. #第一个参数是一个可以返回时间戳的函数，第二个参数可以在定时未到达之前阻塞。
4. schedule = sched.scheduler ( time.time, time.sleep )
5.
6. #被周期性调度触发的函数
7. def func():
8. os.system("scrapy crawl News")
9. def perform1(inc):
10. schedule.enter(inc,0,perform1,(inc,))
11. func() # 需要周期执行的函数
12. def mymain():
13. schedule.enter(0,0,perform1,(86400,))
14.
15. if __name__=="__main__":
16. mymain()
17. schedule.run() # 开始运行，直到计划时间队列变成空为止
```

(3) 使用 Crontab

10、爬的那些内容数据量有多大，多久爬一次，爬下来的数据是怎么存储?

京东整站的数据大约在 1 亿左右，爬下来的数据存入数据库，mysql 数据库中如果有重复的 url 建议去重存入数据库，可以考虑引用外键。评分、评论如果做增量，Redis 中 url 去重，评分和评论建议建立一张新表用 id 做关联。

多久爬一次这个问题要根据公司的要求去处理，不一定是每天都爬。

Mongo 建立唯一索引键（id）可以做数据重复 前提是数据量不大， 2 台电脑几百万的情况 数据库需要做分片 （数据库要设计合理）。

例：租房的网站数据量每天大概是几十万条 ，每周固定爬取。

11、cookie 过期的处理问题?

因为 cookie 存在过期的现象，一个很好的处理方法就是做一个异常类，如果有异常的话 cookie 抛出异常类在执行程序。

12、动态加载又对及时性要求很高怎么处理?

Selenium + Phantomjs，尽量不使用 sleep 而使用 WebDriverWait

13、谈一谈你对 Selenium 和 PhantomJS 了解?

Selenium 是一个 Web 的自动化测试工具，可以根据我们的指令，让浏览器自动加载页面，获取需要的数据，甚至页面截屏，或者判断网站上某些动作是否发生。Selenium 自己不带浏览器，不支持浏览器的功能，它需要与第三方浏览器结合在一起才能使用。但是我们有时候需要让它内嵌在代码中运行，所以我们可以用一个叫 PhantomJS 的工具代替真实的浏览器。Selenium 库里有个叫 WebDriver 的 API。WebDriver 有点儿像可以加载网站的浏览器，但是它也可以像 BeautifulSoup 或者其他 Selector 对象一样用来查找页面元素，与页面上的元素进行交互 (发送文本、点击等)，以及执行其他动作来运行网络爬虫。

PhantomJS 是一个基于 Webkit 的“无界面”(headless)浏览器，它会把网站加载到内存并执行页面上的 JavaScript，因为不会展示图形界面，所以运行起来比完整的浏览器要高效。相比传统的 Chrome 或 Firefox 浏览器等，资源消耗会更少。

如果我们把 Selenium 和 PhantomJS 结合在一起，就可以运行一个非常强大的网络爬虫了，这个爬虫可以处理 JavaScript、Cookie、headers，以及任何我们真实用户需要做的事情。

主程序退出后，selenium 不保证 phantomJS 也成功退出，最好手动关闭 phantomJS 进程。（有可能会导致多个 phantomJS 进程运行，占用内存）。

WebDriverWait 虽然可能会减少延时，但是目前存在 bug（各种报错），这种情况可以采用 sleep。phantomJS 爬数据比较慢，可以选择多线程。如果运行的时候发现有的可以运行，有的不能，可以尝试将 phantomJS 改成 Chrome。

14、代理 IP 里的“透明”“匿名”“高匿”分别是指?

(1) 透明代理

它的意思是客户端根本不需要知道有代理服务器的存在，但是它传送的仍然是真实的 IP。你要想隐藏的话，不要用这个。

(2) 普通匿名代理

普通匿名代理能隐藏客户机的真实 IP，但会改变我们的请求信息，服务器端有可能会认为我们使用了代理。不过使用此种代理时，虽然被访问的网站不能知道你的 ip 地址，但仍然可以知道你在使用代理，当然某些能够侦测 ip 的网页仍然可以查到你的 ip。

(3) 高匿名代理

高匿名代理不改变客户机的请求，这样在服务器看来就像有个真正的客户浏览器在访问它，这时客户的真实 IP 是隐藏的，服务器端不会认为我们使用了代理。

设置代理有以下两个好处

1. 让服务器以为不是同一个客户端在请求
2. 防止我们的真实地址被泄露，防止被追究

15、robots 协议有什么用处?

Robots 协议：网站通过 Robots 协议告诉搜索引擎哪些页面可以抓取，哪些页面不能抓取。

16、为什么 requests 请求需要带上 header?

原因是：模拟浏览器，欺骗服务器，获取和浏览器一致的内容

header 的形式：字典

headers = {"User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36(KHTML, like Gecko) Chrome/54.0.2840.99 Safari/537.36"}

用法： requests.get(url,headers=headers)

17、dumps,loads 与 dump,load 的区别?

json.dumps() 将 pyhton 的 dict 数据类型编码为 json 字符串

json.loads() 将 json 字符串解码为 dict 的数据类型

json.dump(x,y) x 是 json 对象, y 是文件对象，最终是将 json 对象写入到文件中

json.load(y) 从文件对象 y 中读取 json 对象

18、平常怎么使用代理的?

(1) 自己维护代理池

(2) 付费购买（目前市场上有很多 ip 代理商，可自行百度了解，建议看看他们的接口文档（API&SDK））

19、IP 存放在哪里？怎么维护 IP？对于封了多个 ip 的，怎么判定 IP 没被封?

存放在数据库(redis、mysql 等)

维护多个代理网站

一般代理的存活时间往往在十几分钟左右，定时任务，加上代理 IP 去访问网页，验证其是否可用，如果返回状态为 200，表示这个代理是可以使用的。

20、怎么获取加密的数据?

(1) Web 端加密可尝试移动端（app）

(2) 解析加密，看能否破解

(3) 反爬手段层出不穷，js 加密较多，只能具体问题具体分析

21、假如每天爬取量在 5、6 万条数据，一般开几个线程，每个线程 ip 需要加锁限定吗?

(1) 5、6 万条数据相对来说数据量比较小，线程数量不做强制要求(做除法得一个合理值即可)

(2) 多线程使用代理，应保证不在同时一刻使用一个代理 IP

22、怎么监控爬虫的状态?

(1) 使用 python 的 STMP 包将爬虫的状态信心发送到指定的邮箱

(2) Scrapyd、pyspider

(3) 引入日志

框架

1、描述下 scrapy 框架运行的机制?

从 start_urls 里获取第一批 url 并发送请求，请求由引擎交给调度器入请求队列，获取完毕后，调度器将请求队列里的请求交给下载器去获取请求对应的响应资源，并将响应交给自己编写的解析方法做提取处理：

(1) 如果提取出需要的数据，则交给管道文件处理；

(2) 如果提取出 url，则继续执行之前的步骤（发送 url 请求，并由引擎将请求交给调度器入队列...），直到请求队列里没有请求，程序结束。

2、谈谈你对 Scrapy 的理解?

scrapy 是一个为了爬取网站数据，提取结构性数据而编写的应用框架，我们只需要实现少量代码，就能够快速的抓取到数据内容。Scrapy 使用了 Twisted 异步网络框架来处理网络通讯，可以加快我们的下载速度，不用自己去实现异步框架，并且包含了各种中间件接口，可以灵活的完成各种需求。

scrapy 框架的工作流程：

(1) 首先 Spiders（爬虫）将需要发送请求的 url(requests)经 ScrapyEngine（引擎）交给 Scheduler（调度器）

(2) Scheduler（排序，入队）处理后，经 ScrapyEngine，DownloaderMiddlewares(可选，主要有 User_Agent， Proxy 代理)交给 Downloader

(3) Downloader 向互联网发送请求，并接收下载响应（response）。将响应（response）经 ScrapyEngine，SpiderMiddlewares(可选)交给 Spiders

(4)Spiders 处理 response，提取数据并将数据经 ScrapyEngine 交给 ItemPipeline 保存（可以是本地，可以是数据库）。提取 url 重新经 ScrapyEngine 交给 Scheduler 进行下一个循环。直到无 Url 请求程序停止结束

3、什么是增量爬取?

增量爬取即保存上一次状态，本次抓取时与上次比对，如果不在上次的状态中，便视为增量，保存下来。对于 scrapy 来说，上一次的状态是抓取的特征数据和上次爬取的 request 队列（url 列表），request 队列可以通过 request 队列可以通过 scrapy.core.scheduler 的 pending_requests 成员得到，在爬虫启动时导入上次爬取的特征数据，并且用上次 request 队列的数据作为 start url 进行爬取，不在上一次状态中的数据便保存。

选用 BloomFilter 原因：对爬虫爬取数据的保存有多种形式，可以是数据库，可以是磁盘文件等，不管是数据库，还是磁盘文件，进行扫描和存储都有很大的时间和空间上的开销，为了从时间和空间上提升性能，故选用 BloomFilter 作为上一次爬取数据的保存。保存的特征数据可以是数据的某几项，即监控这几项数据，一旦这几项数据有变化，便视为增量持久化下来，根据增量的规则可以对保存的状态数据进行约束。比如：可以选网页更新的时间，索引次数或是网页的实际内容，cookie 的更新等。

4、爬虫向数据库存数据开始和结束都会发一条消息，是 scrapy 哪个模块实现的?

Scrapy 使用信号来通知事情发生，因此答案是 signals 模块。

5、爬取下来的数据如何去重，说一下具体的算法依据?

(1)通过 MD5 生成电子指纹来判断页面是否改变

(2) nutch 去重。nutch 中 digest 是对采集的每一个网页内容的 32 位哈希值，如果两个网页内容完全一样，它们的 digest 值肯定会一样。

数据量不大时，可以直接放在内存里面进行去重，python 可以使用 set()进行去重。当去重数据需要持久化时可以使用 redis 的 set 数据结构。

当数据量再大一点时，可以用不同的加密算法先将长字符串压缩成 16/32/40 个字符，再使用上面两种方法去重。

当数据量达到亿（甚至十亿、百亿）数量级时，内存有限，必须用“位”来去重，才能够满足需求。Bloomfilter 就是将去重对象映射到几个内存“位”，通过几个位的 0/1 值来判断一个对象是否已经存在。

然而 Bloomfilter 运行在一台机器的内存上，不方便持久化（机器 down 掉就什么都没啦），也不方便分布式爬虫的统一去重。如果可以在 Redis 上申请内存进行 Bloomfilter，以上两个问题就都能解决了。

simhash 最牛逼的一点就是将一个文档，最后转换成一个 64 位的字节，暂且称之为特征字，然后判断重复只需要判断他们的特征字的距离是不是<n（根据经验这个 n 一般取值为 3），就可以判断两个文档是否相似。

可见 scrapy_redis 是利用 set 数据结构来去重的，去重的对象是 request 的 fingerprint（其实就是用 hashlib.sha1()对 request 对象的某些字段信息进行压缩）。其实 fp 就是 request 对象加密压缩后的一个字符串（40 个字符，0~f）。

6、Scrapy 的优缺点?

优点

(1) scrapy 是异步的

(2) 采取可读性更强的 xpath 代替正则

(3)强大的统计和 log 系统

(4) 同时在不同的 url 上爬行

(5) 支持 shell 方式，方便独立调试

(6) 写 middleware,方便写一些统一的过滤器

(7) 通过管道的方式存入数据库

缺点

(1) 基于 python 的爬虫框架，扩展性比较差

(2) 基于 twisted 框架，运行中的 exception 是不会干掉 reactor（反应器），并且异步框架出错后是不会停掉其他任务的，数据出错后难以察觉。

7、什么是 scrapy-redis 中的指纹？是如何去重的?

指纹

通过 sha1 加密，把请求体，请求方式，请求 url 放在一起。然后进行 16 进制的转义符字符串生成指纹。生成一个字符串，放到数据库中作为唯一标示。

去重

url 中按照 url 去重：

(1) 按照 url 去重，有一个列表，发送请求之前从数据表中看一下这个 url有没有请求过，请求过了就不用看了

(2) 内容判断，从数据库中查数据的表示，如果请求过了就在不在请求了。

8、怎么设置深度爬取?

通过在 settings.py 中设置 depth_limit 的值可以限制爬取深度，这个深度是与 start_urls 中定义 url 的相对值。也就是相对 url 的深度。若定义 url 为 http://www.domz.com/game/,depth_limit=1 那么限制爬取的只能是此 url 下一级的网页。深度大于设置值的将被忽视。

9、scrapy 和 scrapy-redis 有什么区别？为什么选择 redis 数据库?

scrapy 是一个 Python 爬虫框架，爬取效率极高，具有高度定制性，但是不支持分布式。而scrapy-redis 一套基于 redis 数据库、运行在 scrapy 框架之上的组件，可以让 scrapy 支持分布策略，Slaver 端共享 Master 端 redis 数据库里的 item 队列、请求队列和请求指纹集合。

为什么选择 redis 数据库，因为 redis 支持主从同步，而且数据都是缓存在内存中的，所以基于 redis的分布式爬虫，对请求和数据的高频读取效率非常高。

10、分布式爬虫主要解决什么问题?

主要解决一下 4 个问题：

(1) ip

(2) 带宽

(3) cpu

(4) io

11、什么是反向代理？作用是什么?

代理可以假扮 Web 服务器。这些被称为替换物(surrogate)或反向代理(reverse proxy)的代理接收发送给 Web 服务器的真实请求，但与 Web 服务器不同的是，它们可以发起与其他服务器的通信，以便按需定位所请求的内容。

可以用这些反向代理来提高访问慢速 Web 服务器上公共内容的性能。在这种配置中，通常将这些反向代理称为服务器加速器(server accelerator)。还可以将替换物与内容路由功能配合使用，以创建按需复制内容的分布式网络。

12、什么是分布式存储?

传统定义

分布式存储系统是大量 PC 服务器通过 Internet 互联，对外提供一个整体的服务。

分布式存储系统具有以下特性

(1) 可扩展：分布式存储系统可以扩展到几百台甚至几千台这样的一个集群规模，系统的整体性能线性增长。

(2) 低成本：分布式存储系统的自动容错、自动负载均衡的特性，允许分布式存储系统可以构建在低成本的服务器上。另外，线性的扩展能力也使得增加、减少服务器的成本低，实现分布式存储系统的自动运维。

(3) 高性能：无论是针对单台服务器，还是针对整个分布式的存储集群，都要求分布式存储系统具备高性能。

(4) 易用：分布式存储系统需要对外提供方便易用的接口，另外，也需要具备完善的监控、运维工具，并且可以方便的与其他的系统进行集成。分布式存储系统的挑战主要在于数据和状态信息的持久化，要求在自动迁移、自动容错和并发读写的过程中，保证数据的一致性。

(5) 容错：可以快速检测到服务器故障，并自动的将在故障服务器上的数据进行迁移。

(6) 负载均衡：新增的服务器在集群中保障负载均衡？数据迁移过程中保障不影响现有的服务。

(7) 事务与并发控制：实现分布式事务。

(8) 易用性：设计对外接口，使得设计的系统易于使用。

13、你所知道的分布式爬虫方案有哪些?

3 种分布式爬虫策略

(1) Slaver 端从 Master 端拿任务（Request/url/ID）进行数据抓取，在抓取数据的同时也生成新任务，并将任务抛给 Master。Master 端只有一个 Redis 数据库，负责对 Slaver 提交的任务进行去重、加入待爬队列。

优点： scrapy-redis 默认使用的就是这种策略，我们实现起来很简单，因为任务调度等工作 scrapy-redis 都已经帮我们做好了，我们只需要继承 RedisSpider、指定 redis_key 就行了。

缺点： scrapy-redis 调度的任务是 Request 对象，里面信息量比较大（不仅包含 url，还有 callback 函数、headers 等信息），导致的结果就是会降低爬虫速度、而且会占用 Redis 大量的存储空间。当然我们可以重写方法实现调度 url 或者用户 ID。

(2) Master 端跑一个程序去生成任务（Request/url/ID）。Master 端负责的是生产任务，并把任务去重、加入到待爬队列。Slaver 只管从 Master 端拿任务去爬。

优点： 将生成任务和抓取数据分开，分工明确，减少了 Master 和 Slaver 之间的数据交流；Master 端生成任务还有一个好处就是：可以很方便地重写判重策略（当数据量大时优化判重的性能和速度还是很重要的）。

缺点： 像 QQ 或者新浪微博这种网站，发送一个请求，返回的内容里面可能包含几十个待爬的用户 ID，即几十个新爬虫任务。但有些网站一个请求只能得到一两个新任务，并且返回的内容里也包含爬虫要抓取的目标信息，如果将生成任务和抓取任务分开反而会降低爬虫抓取效率。毕竟带宽也是爬虫的一个瓶颈问题，我们要秉着发送尽量少的请求为原则，同时也是为了减轻网站服务器的压力，要做一只有道德的 Crawler。所以，视情况而定。

3.Master 中只有一个集合，它只有查询的作用。Slaver 在遇到新任务时询问 Master 此任务是否已爬，如果未爬则加入 Slaver 自己的待爬队列中，Master 把此任务记为已爬。它和策略一比较像，但明显比策略一简单。策略一的简单是因为有 scrapy-redis 实现了 scheduler 中间件，它并不适用于非 scrapy 框架的爬虫。

优点： 实现简单，非 scrapy 框架的爬虫也适用。Master 端压力比较小，Master 与 Slaver 的数据交流也不大。

缺点：“健壮性”不够，需要另外定时保存待爬队列以实现“断点续爬”功能。各 Slaver 的待爬任务不通用。

如果把 Slaver 比作工人，把 Master 比作工头。策略一就是工人遇到新任务都上报给工头，需要干活的时候就去工头那里领任务；策略二就是工头去找新任务，工人只管从工头那里领任务干活；策略三就是工人遇到新任务时询问工头此任务是否有人做了，没有的话工人就将此任务加到自己的“行程表”。

机器学习（共 32 题）

1、什么是机器学习

简单的说，机器学习就是让机器从数据中学习，进而得到一个更加符合现实规律的模型，通过对模型的使用使得机器比以往表现的更好，这就是机器学习。

对上面这句话的理解：

数据：从现实生活抽象出来的一些事物或者规律的特征进行数字化得到。

学习：在数据的基础上让机器重复执行一套特定的步骤（学习算法）进行事物特征的萃取，得到一个更加逼近于现实的描述（这个描述是一个模型它的本身可能就是一个函数）。我们把大概能够描述现实的这个函数称作我们学到的模型。

更好：我们通过对模型的使用就能更好的解释世界，解决与模型相关的问题。

2、机器学习与数据挖掘的区别?

数据挖掘和机器学习的区别和联系，周志华有一篇很好的论述《机器学习和数据挖掘》可以帮助大家理解。

数据挖掘受到很多学科领域的影响，其中数据库、机器学习、统计学无疑影响最大。简言之，对数据挖掘而言，数据库提供数据管理技术，机器学习和统计学提供数据分析技术。

由于统计学往往醉心于理论的优美而忽视实际的效用，因此，统计学界提供的很多技术通常都要在机器学习界进一步研究，变成有效的机器学习算法之后才能再进入数据挖掘领域。从这个意义上说，统计学主要是通过机器学习来对数据挖掘发挥影响，而机器学习和数据库则是数据挖掘的两大支撑技术。

从数据分析的角度来看，绝大多数数据挖掘技术都来自机器学习领域，但机器学习研究往往并不把海量数据作为处理对象，因此，数据挖掘要对算法进行改造，使得算法性能和空间占用达到实用的地步。同时，数据挖掘还有自身独特的内容，即关联分析。

而模式识别和机器学习的关系是什么呢，传统的模式识别的方法一般分为两种：统计方法和句法方法。句法分析一般是不可学习的，而统计分析则是发展了不少机器学习的方法。也就是说，机器学习同样是给模式识别提供了数据分析技术。

至于，数据挖掘和模式识别，那么从其概念上来区分吧，数据挖掘重在发现知识，模式识别重在认识事物。

机器学习的目的是建模隐藏的数据结构，然后做识别、预测、分类等。因此，机器学习是方法，模式识别是目的。

总结一下吧。只要跟决策有关系的都能叫 AI(人工智能)，所以说 PR（模式识别）、DM（数据挖掘）、IR（信息检索） 属于 AI 的具 体应用应该没有问题。 研究的东西则不太一样， ML(机器学习) 强调自我完善的过程。 Anyway，这些学科都是相通的。

3、什么是机器学习的过度拟合现象？过度拟合产生的原因？如何避免过拟合问题?

如果一味的去提高训练数据的预测能力，所选模型的复杂度往往会很高，这种现象称为过拟合，所表现的就是模型训练时候的误差很小，但在测试的时候误差很大。

过拟合的产生总结一下大概有以下五种原因可能产生:

(1) 建模样本抽取错误，包括（但不限于）样本数量太少，抽样方法错误，抽样时没有足够正确考虑业务场景或业务特点，等等导致抽出的样本数据不能有效足够代表业务逻辑或业务场景

(2) 样本里的噪音数据干扰过大，大到模型过分记住了噪音特征，反而忽略了真实的输入输出间的关系

(3) 在决策树模型搭建中，如果我们对于决策树的生长没有合理的限制和修剪的话，决策树的自由生长有可能每片叶子里只包含单纯的事件数据(event)或非事件数据（no event），可以想象，这种决策树当然可以完美匹配（拟合）训练数据，但是一旦应用到新的业务真实数据时，效果是一塌糊涂。

(4) 建模时的“逻辑假设”到了模型应用时已经不能成立了。任何预测模型都是在假设的基础上才可以搭建和应用的，常用的假设包括：假设历史数据可以推测未来，假设业务环节没有发生显著变化，假设建模数据与后来的应用数据是相似的，等等。如果上述假设违反了业务场景的话，根据这些假设搭建的模型当然是无法有效应用的。

(5) 建模时使用了太多的输入变量，这跟上面第二点（噪音数据）有些类似，数据挖掘新人常常犯这个错误，自己不做分析判断，把所有的变量交给软件或者机器去“撞大运”。须知，一个稳定优良的模型一定要遵循建模输入变量“少而精”的原则的。

上面的原因都是现象，但是其本质只有一个，那就是“业务理解错误造成的”，无论是抽样，还是噪音，还是决策树，神经网络等等，如果我们对于业务背景和业务知识非常了解，非常透彻的话，一定是可以避免绝大多数过拟合现象产生的。因为在模型从确定需求，到思路讨论，到搭建，到业务应用验证，各个环节都是可以用业务敏感来防止过拟合于未然的。

避免过拟合问题：

(1) 重采样Bootstrap

(2) L1，L2 正则化

(3) 决策树的剪枝操作

(4) 交叉验证

4、什么是机器学习的欠拟合？如何避免欠拟合问题？

所谓欠拟合就是模型复杂度低或者数据集太小，对模型数据的拟合程度不高，因此模型在训练集上的效果就不好。

避免欠拟合问题：

(1) 增加样本数量

(2) 增加样本特征的数量

(3) 可以进行特征维度扩展

5、解释有监督和无监督机器学习之间的区别？

监督学习需要训练标记的数据。换句话说，监督学习使用了基本事实，这意味着我们对输出和样本已有知识。这里的目标是学习一个近似输入和输出之间关系的函数。

另一方面，无监督学习不使用标记的输出。此处的目标是推断数据集中的自然结构。

6、偏差和方差之间的权衡是什么？

偏差（模型拟合数据的程度）是指由于ML算法中的假设不正确或过于简单而导致的误差，这会导致过度拟合。

方差（模型基于输入的变化量）指的是由于ML算法的复杂性而导致的错误，从而对训练数据的高水平变化和过度拟合产生了敏感性。

换句话说，简单的模型是稳定的（低方差）但有很大的偏差。复杂的模型容易过拟合，但表达了模型的真实性（低偏差）。误差的最佳减少需要权衡偏差和方差，以避免高方差和高偏差。

7、解释 KNN 和 k-means 聚类之间的区别？

主要区别在于，KNN 需要标记点（分类算法，监督学习），而 k-means 则不需要（聚类算法，无监督学习）。

若要使用 K 最近邻，请使用要分类为未标记点的标记数据。K 均值聚类采用未标记的点，并学习如何使用点之间距离的平均值对它们进行分组。

8、什么是贝叶斯定理？我们为什么用它？

贝叶斯定理是我们在知道其他概率时如何找到概率的方法。换句话说，它提供了先验知识事件的后验概率。该定理是计算条件概率的原则方法。

在 ML 中，贝叶斯定理用于将模型拟合到训练数据集的概率框架中，并用于建立分类预测建模问题（即，朴素贝叶斯，贝叶斯最优分类器）。

9、什么是朴素贝叶斯分类器？我们为什么要使用它们？

朴素贝叶斯分类器是分类算法的集合。这些分类器是具有共同原理的一系列算法。朴素贝叶斯分类器假设某个功能的存在或不影响其他功能的存在或不存在。

换句话说，我们称其为“天真”，因为它假定所有数据集特征都同等重要且独立。

朴素贝叶斯分类器用于分类。当独立性的假设成立时，与其他复杂的预测变量相比，它们更易于实现并产生更好的结果。它们用于垃圾邮件过滤，文本分析和推荐系统。

10、判别模型和生成模型之间有什么区别？

判别模型学习不同类别的数据之间的区别。生成模型学习数据类别。 区分模型通常在分类任务上表现更好。

11、什么是参数模型？提供一个例子

参数模型具有有限数量的参数。您只需要知道模型的参数即可进行数据预测。常见示例如下：线性SVM，线性回归和逻辑回归。

非参数模型具有无限数量的参数以提供灵活性。对于数据预测，您需要模型的参数和观测数据的状态。常见示例如下：k最近邻，决策树和主题模型。

12、如何根据训练集大小选择分类器？

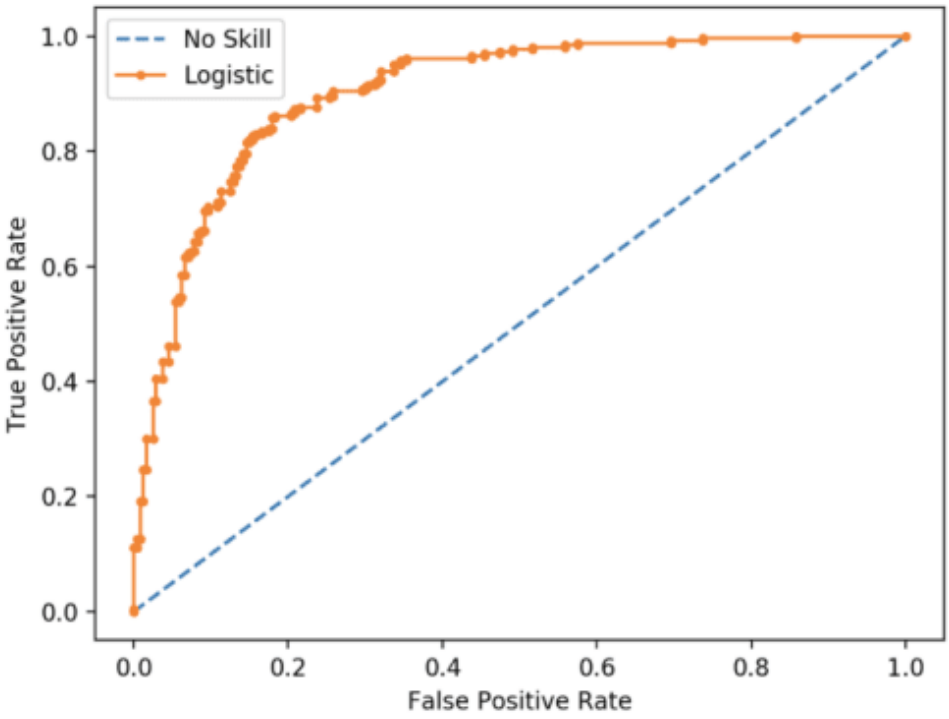
对于小的训练集，具有高偏差和低方差模型的模型更好，因为它不太可能过拟合。一个例子是朴素贝叶斯。

对于大型训练集，具有低偏差和高方差模型的模型更好，因为它表示更复杂的关系。一个例子是逻辑回归。

13、解释ROC曲线和AUC？

ROC曲线是在所有阈值下分类模型的性能的图形表示。它有两个阈值：真阳性率和假阳性率。

简单来说，AUC（ ROC 曲线下方的面积）就是 ROC 曲线下方的面积。AUC 测量 ROC 曲线下从（0,0）到（1,1）的二维区域。它用作评估二进制分类模型的性能指标。



14、如何确保您不会过度拟合模型?

我们可以使用三种方法来防止过度拟合：

(1) 使用交叉验证技术（例如k折交叉验证）

(2) 保持模型简单（即减少变量）以减少方差

(3) 使用正则化技术（例如 LASSO）来惩罚可能导致过度拟合的模型参数

15、说明如何处理数据集中的丢失或损坏的数据?

您需要标识查找数据并删除行/列，或将其替换为其他值。

pandas 提供了执行此操作的有用方法：isnull() 和 dropna()。这些使您可以识别并删除损坏的数据。该 fillna() 方法可用于使用占位符填充无效值。

16、解释如何开发数据管道?

数据管道使我们能够采用数据科学模型并对其进行自动化或扩展。常见的数据管道工具是 Apache Airflow，并且使用 Google Cloud，Azure 和 AWS 托管它们。

对于这样的问题，您想解释所需的步骤并讨论您在构建数据管道方面的实际经验。

对于 Google Cloud 主机，基本步骤如下：

(1) 登录到 Google Cloud Platform

(2) 创建一个计算实例

(3) 从 GitHub 中提取教程内容

(4) 使用 AirFlow 概述管道

(5) 使用 Docker 设置虚拟主机

(6) 开发一个 Docker 容器

(7) 打开 Airflow UI 并运行数据管道

(8) 运行已部署的 Web 应用

17、如何解决模型中的高方差?

如果模型具有低方差和高偏差，我们使用装袋算法，该算法使用随机抽样将数据集划分为子集。我们使用这些样本通过单个学习算法生成一组模型。

此外，我们可以使用正则化技术，其中较高的模型系数会受到惩罚，以降低总体复杂度。

18、什么是超参数?它们与模型参数有何不同?

模型参数是模型内部的变量。参数值是根据训练数据估算的。

超参数是模型外部的变量。该值无法从数据中估算出来，通常用于估算模型参数。

19、你正在处理数据集。如何选择重要变量?

(1) 在选择重要变量之前，请先删除相关变量

(2) 使用随机森林和绘图变量重要性图表

(3) 使用套索回归

(4) 使用线性回归以基于p值选择变量

(5) 使用前向选择，逐步选择和向后选择

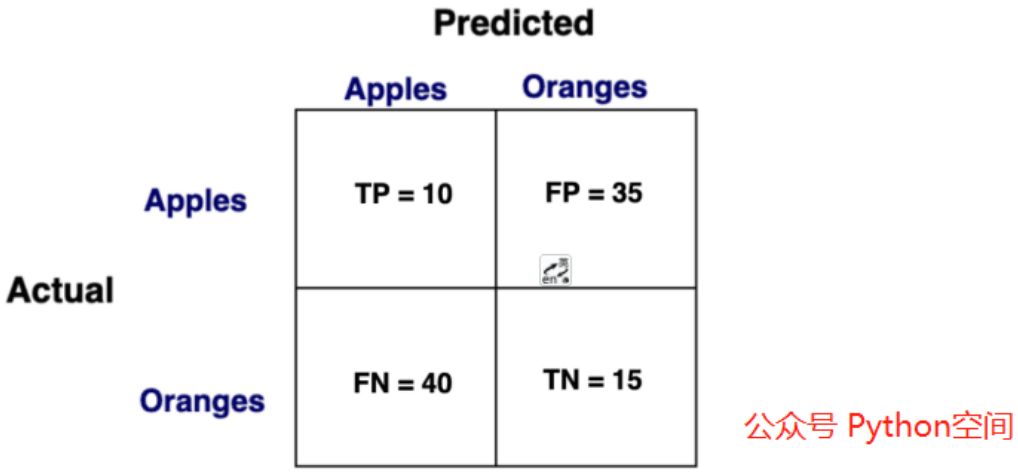
20、为什么 XGBoost 的性能优于 SVM?

XGBoos 是使用许多树的集成方法。这意味着它会随着自身的重复而提高。

SVM 是线性分隔符。因此，如果我们的数据不是线性可分离的，那么 SVM 需要一个内核来使数据达到可以分离的状态。这可能会限制我们，因为每个给定的数据集都没有完美的内核。

21、以下混淆矩阵的召回率，特异性和精确度是多少?

- TP /真正肯定：案例是肯定的，并且被预测为肯定
- TN /真阴性：案件为阴性，预计为阴性
- FN /假阴性：案件为阳性，但预计为阴性
- FP /误报：案例为否定，但预计为肯定



召回率= 20%

特异性= 30%

精度= 22%

补充

> 说明：
>
> 召回率= TP /（TP + FN）= 10/50 = 0.2 = 20%
>
> 特异性= TN /（TN + FP）= 15/50 = 0.3 = 30%
>
> 精度= TP /（TP + FP）= 10/45 = 0.2 = 22%

22、使用 scikit-learn 时，是否确实需要在特征值变化很大时调整特征值?

是。大多数机器学习算法都使用欧几里得距离作为度量标准来测量两个数据点之间的距离。如果值的范围相差很大，则不同功能部件中相同更改的结果将非常不同。

23、你的数据集有 50 个变量，但是8个变量的缺失值高于 30%。如何解决这个问题?

您可以采用 3 种一般方法：

- 只是删除它们（不理想）
- 为缺失值分配唯一的类别，以查看是否有趋势生成此问题
- 检查目标变量的分布。如果找到了模式，请保留缺少的值，将其分配给新类别，然后删除其他类别

24、协方差和相关性有什么区别?

相关性是协方差的标准化格式。协方差本身很难做比较。例如：如果我们计算工资（￥）和年龄（岁）的协方差，因为这两个变量有不同的度量，所以我们会得到不能做比较的不同的协方差。为了解决这个问题，我们计算相关性来得到一个介于-1和1之间的值，就可以忽略它们各自不同的度量。

25、你认为把分类变量当成连续型变量会得到一个更好的预测模型吗?

为了得到更好的预测，只有在分类变量在本质上是有序的情况下才可以被当做连续型变量来处理。

26、在 K-means 或者 KNN，我们是用欧氏距离来计算最近的邻居之间的距离，为什么不用曼哈顿距离?

我们不用曼哈顿距离，因为它只计算水平或者垂直距离，有维度的限制。另一方面，欧氏距离可以用于任何空间的距离计算问题。因为，数据点可以存在于任何空间，欧式距离是更可行的选择。例如：想象一下国际象棋棋盘，象或者车所有的移动的由曼哈顿距离计算的，因为他们是在各自的水平和垂直方向做的运动。

27、给你一个数据集，这个数据集有缺失值，且这些缺失值分布在高中值有1一个标准偏差的的范围内，百分之多少的数据不会受到影响？为什么?

大约有32%的数据将不会受到缺失值的影响。因为，由于数据分布在中位数附近，让我们先假设这是一个正态分布。我们知道，在一个正态分布中，约有68%的数据位于跟平均值（或者众数，中位数）1个标准差范围内，那么剩下的约32%的数据是不受影响的。因此，约有32%的数据将不受缺失值的影响。

28、线程分类器与非线性分类器的区别以及优劣?

如果模型是参数的线性函数，并且存在线性分类面，那么就是线性分类器，否则不是。

常用的线性分类器有:LR ,贝叶斯分类，单层感知器，线性回归

常见的非线性分类器：决策树，RF，GBDT，多层感知机

SVM两种都有（看线性核还是高斯核）

线性分类器速度快，编程方便，但是可能拟合效果不会很好

非线性分类器编程复杂，但是效果拟合能力强

29、文本中的余弦距离是什么，有哪些作用?

余弦距离是两个向量的距离的一种度量方式，其值在-1~1之间，如果为1表示两个向量同相，0表示两个向量正交，-1表示两个向量反向。使用TF-IDF和余弦距离可以寻找内容相似的文章，例如首先用TF-IDF找出两篇文章的关键词，然后每个文章分别取出k个关键词(10-20个)，统计这些关键词的词频，生成两篇文章的词频向量，然后用余弦距离计算其相似度。

30、什么是数据埋点?

数据埋点我们可以分为两类，其一是页面统计，其二是行为统计

页面统计可以帮我们知晓某个页面被多少人访问了多少次

行为统计是指用户在界面上的操作行为，应用最多的是按钮的点击次数

31、请简要说说一个完整的机器学习项目流程

(1) 抽象成数学问题

明确问题是进行机器学习的第一步。机器学习的训练过程通常都是一件非常耗时的事情，胡乱尝试时间成本是非常高的。

这里的抽象成数学问题，指的我们明确我们可以获得什么样的数据，目标是一个分类还是回归或者是聚类的问题，如果都不是的话，如果划归为其中的某类问题。

(2) 获取数据

数据决定了机器学习结果的上限，而算法只是尽可能逼近这个上限。数据要有代表性，否则必然会过拟合。而且对于分类问题，数据偏斜不能过于严重，不同类别的数据数量不要有数个数量级的差距。

而且还要对数据的量级有一个评估，多少个样本，多少个特征，可以估算出其对内存的消耗程度，判断训练过程中内存是否能够放得下。如果放不下就得考虑改进算法或者使用一些降维的技巧了。如果数据量实在太大，那就要考虑分布式了。

(3) 特征预处理与特征选择

良好的数据要能够提取出良好的特征才能真正发挥效力。

特征预处理、数据清洗是很关键的步骤，往往能够使得算法的效果和性能得到显著提高。归一化、离散化、因子化、缺失值处理、去除共线性等，数据挖掘过程中很多时间就花在它们上面。这些工作简单可复制，收益稳定可预期，是机器学习的基础必备步骤。

筛选出显著特征、摒弃非显著特征，需要机器学习工程师反复理解业务。这对很多结果有决定性的影响。特征选择好了，非常简单的算法也能得出良好、稳定的结果。这需要运用特征有效性分析的相关技术，如相关系数、卡方检验、平均互信息、条件熵、后验概率、逻辑回归权重等方法。

(4) 训练模型与调优

直到这一步才用到我们上面说的算法进行训练。现在很多算法都能够封装成黑盒供人使用。但是真正考验水平的是调整这些算法的（超）参数，使得结果变得更加优良。这需要我们对算法的原理有深入的理解。理解越深入，就越能发现问题的症结，提出良好的调优方案。

(5) 模型诊断

如何确定模型调优的方向与思路呢？这就需要对模型进行诊断的技术。

过拟合、欠拟合判断是模型诊断中至关重要的一步。常见的方法如交叉验证，绘制学习曲线等。过拟合的基本调优思路是增加数据量，降低模型复杂度。欠拟合的基本调优思路是提高特征数量和质量，增加模型复杂度。

误差分析 也是机器学习至关重要的步骤。通过观察误差样本，全面分析误差产生误差的原因：是参数的问题还是算法选择的问题，是特征的问题还是数据本身的问题……

诊断后的模型需要进行调优，调优后的新模型需要重新进行诊断，这是一个反复迭代不断逼近的过程，需要不断地尝试， 进而达到最优状态。

(6)模型融合

一般来说，模型融合后都能使得效果有一定提升。而且效果很好。

工程上，主要提升算法准确度的方法是分别在模型的前端（特征清洗和预处理，不同的采样模式）与后端（模型融合）上下功夫。因为他们比较标准可复制，效果比较稳定。而直接调参的工作不会很多，毕竟大量数据训练起来太慢了，而且效果难以保证。

(7) 上线运行

这一部分内容主要跟工程实现的相关性比较大。工程上是结果导向，模型在线上运行的效果直接决定模型的成败。 不单纯包括其准确程度、误差等情况，还包括其运行的速度(时间复杂度)、资源消耗程度（空间复杂度）、稳定性是否可接受。

这些工作流程主要是工程实践上总结出的一些经验。并不是每个项目都包含完整的一个流程。这里的部分只是一个指导性的说明，只有大家自己多实践，多积累项目经验，才会有自己更深刻的认识。

32、我们知道，独热编码（OneHotEncoder）会增加数据集的维度。但是标签编码（LabelEncoder）不会。为什么?

用独热编码(OneHotEncoder)，数据集的维度（也即特征）增加是因为它为分类变量中存在的的每一级都创建了一个变量。例如：假设我们有一个变量“颜色”。这变量有 3 个层级，即红色、蓝色和绿色。

对“颜色”变量进行一位有效编码会生成含0和1值的 Color.Red，Color.Blue 和 Color.Green 三个新变量。在标签编码中，分类变量的层级编码为0，1和2，因此不生成新变量。标签编码主要是用于二进制变量。