面试整理：分享50道硬核Python编程题，面试前过一遍

目录

题目28：说一下你对闭包的理解。
题目29：说一下Python中的多线程和多进程的应用场景和优缺点。
题目30：说一下Python 2和Python 3的区别。
题目31：谈谈你对“猴子补丁”（monkey patching）的理解。
题目32：阅读下面的代码说出运行结果。
题目33：编写一个函数实现对逆波兰表达式求值，不能使用Python的内置函数。
题目34：Python中如何实现字符串替换操作?
题目35：如何剖析Python代码的执行性能?
题目36：如何使用`random`模块生成随机数、实现随机乱序和随机抽样?
题目37：解释一下线程池的工作原理。
题目38：举例说明什么情况下会出现`KeyError`、`TypeError`、`ValueError`。
题目39：说出下面代码的运行结果。
题目40：如何读取大文件，例如内存只有4G，如何读取一个大小为8G的文件?
题目41：说一下你对Python中模块和包的理解。
题目42：说一下你知道的Python编码规范。
题目43：运行下面的代码是否会报错，如果报错请说明哪里有什么样的错，如果不报错请说出代码的执行结果。
题目44：对下面给出的字典按值从大到小对键进行排序。
题目45：说一下`namedtuple`的用法和作用。
题目46：按照题目要求写出对应的函数。
题目47：按照题目要求写出对应的函数。
题目48：按照题目要求写出对应的装饰器。
题目49：写一个函数实现字符串反转，尽可能写出你知道的所有方法。
题目50：按照题目要求写出对应的函数。

题目001: 在Python中如何实现 单例 模式。

点评：单例模式是指让一个类只能创建出唯一的实例，这个题目在面试中出现的频率极高，因为它考察的不仅仅是单例模式，更是对Python语言到底掌握到何种程度，建议大家用装饰器和元类这两种方式来实现单例模式，因为这两种方式的通用性最强，而且也可以顺便展示自己对装饰器和元类中两个关键知识点的理解。

方法一：使用装饰器实现单例模式。

```
from functools import wraps


def singleton(cls):
```

```
    """单例类装饰器"""
    instances = {}

    @wraps(cls)
    def wrapper(*args, **kwargs):
        if cls not in instances:
            instances[cls] = cls(*args, **kwargs)
        return instances[cls]

    return wrapper


@singleton
class President:
    pass
```

扩展：装饰器是Python中非常有特色的语法，用一个函数去装饰另一个函数或类，为其添加额外的能力。通常通过装饰来实现的功能都属横切关注功能，也就是跟正常的业务逻辑没有必然联系，可以动态添加或移除的功能。装饰器可以为代码提供缓存、代理、上下文环境等服务，它是对设计模式中代理模式的践行。在写装饰器的时候，带装饰功能的函数（上面代码中的wrapper函数）通常都会用functools模块中的wraps再加以装饰，这个装饰器最重要的作用是给被装饰的类或函数动态添加一个__wrapped__属性，这个属性会将被装饰之前的类或函数保留下来，这样在我们不需要装饰功能的时候，可以通过它来取消装饰器，例如可以使用President = President.__wrapped__来取消对President类做的单例处理。需要提醒大家的是：上面的单例并不是线程安全的，如果要做到线程安全，需要对创建对象的代码进行加锁的处理。在Python中可以使用threading模块的RLock对象来提供锁，可以使用锁对象的acquire和release方法来实现加锁和解锁的操作。当然，更为简便的做法是使用锁对象的with上下文语法来进行隐式的加锁和解锁操作。

方法二：使用元类实现单例模式。

```
class SingletonMeta(type):
    """自定义单例元类"""

    def __init__(cls, *args, **kwargs):
        cls.__instance = None
        super().__init__(*args, **kwargs)

    def __call__(cls, *args, **kwargs):
        if cls.__instance is None:
            cls.__instance = super().__call__(*args, **kwargs)
        return cls.__instance
```

```
class President(metaclass=SingletonMeta):
    pass
```

扩展：Python是面向对象的编程语言，在面向对象的世界中，一切皆为对象。对象是通过类来创建的，而类本身也是对象，类这样的对象是通过元类来创建的。我们在定义类时，如果没有给一个类指定父类，那么默认的父类是object，如果没有给一个类指定元类，那么默认的元类是type。通过自定义的元类，我们可以改变一个类默认的行为，就如同上面的代码中，我们通过元类的__call__魔术方法，改变了President类的构造器那样。

补充：关于单例模式，在面试中还有可能被问到它的应用场景。通常一个对象的状态是被其他对象共享的，就可以将其设计为单例，例如项目中使用的数据库连接池对象和配置对象通常都是单例，这样才能保证所有地方获取到的数据库连接和配置信息是完全一致的；而且由于对象只有唯一的实例，因此从根本上避免了重复创建对象造成的时间和空间上的开销，也避免了对资源的多重占用。再举个例子，项目中的日志操作通常也会使用单例模式，这是因为共享的日志文件一直处于打开状态，只能有一个实例去操作它，否则在写入日志的时候会产生混乱。

题目002：不使用中间变量，交换两个变量 a 和 b 的值。

点评：典型的送人头的题目，通常交换两个变量需要借助一个中间变量，如果不允许使用中间变量，在其他编程语言中可以使用异或运算的方式来实现交换两个变量的值，但是Python中有更为简单明了的做法。

方法一：

```
a = a ^ b
b = a ^ b
a = a ^ b
```

方法二：

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

扩展：需要注意，a, b = b, a这种做法其实并不是元组解包，虽然很多人都这样认为。Python字节码指令中有ROT_TWO指令来支持这个操作，类似的还有ROT_THREE，对于3个以上的元素，如a, b, c, d = b, c, d, a，才会用到创建元组和元组解包。想知道你的代码对应的字节码指令，可以使用Python标准库中dis模块的dis函数来反汇编你的Python代码。

题目003：写一个删除列表中重复元素的函数，要求去重后元素相对位置保持不变。

点评：这个题目在初中级Python岗位面试的时候经常出现，题目源于《Python Cookbook》这本书第一章的第10个问题，有很多面试题其实都是这本书上的原题，所以建议大家有时间好好研读一下这本书。

```python
def dedup(items):
    no_dup_items = []
    seen = set()
    for item in items:
        if item not in seen:
            no_dup_items.append(item)
            seen.add(item)
    return no_dup_items
```

如果愿意也可以把上面的函数改造成一个生成器，代码如下所示。

```python
def dedup(items):
    seen = set()
    for item in items:
        if item not in seen:
            yield item
            seen.add(item)
```

扩展：由于Python中的集合底层使用哈希存储，所以集合的in和not in成员运算在性能上远远优于列表，所以上面的代码我们使用了集合来保存已经出现过的元素。集合中的元素必须是hashable对象，因此上面的代码在列表元素不是hashable对象时会失效，要解决这个问题可以给函数增加一个参数，该参数可以设计为返回哈希码或hashable对象的函数。

题目004：假设你使用的是官方的 CPython ，说出下面代码的运行结果。

点评：下面的程序对实际开发并没有什么意义，但却是CPython中的一个大坑，这道题旨在考察面试者对官方的Python解释器到底了解到什么程度。

```python
a, b, c, d = 1, 1, 1000, 1000
print(a is b, c is d)

def foo():
    e = 1000
    f = 1000
    print(e is f, e is d)
    g = 1
    print(g is a)

foo()
```

运行结果：

```
True False
True False
True
```

上面代码中a is b的结果是True但c is d的结果是False，这一点的确让人费解。CPython解释器出于性能优化的考虑，把频繁使用的整数对象用一个叫small_ints的对象池缓存起来造成的。small_ints缓存的整数值被设定为[-5, 256]这个区间，也就是说，在任何引用这些整数的地方，都不需要重新创建int对象，而是直接引用缓存池中的对象。如果整数不在该范围内，那么即便两个整数的值相同，它们也是不同的对象。

CPython底层为了进一步提升性能还做了另一个设定，对于同一个代码块中值不在small_ints缓存范围内的整数，如果同一个代码块中已经存在一个值与其相同的整数对象，那么就直接引用该对象，否则创建新的int对象。需要大家注意的是，这条规则对数值型适用，但对字符串则需要考虑字符串的长度，这一点大家可以自行证明。

扩展：如果你用PyPy（另一种Python解释器实现，支持JIT，对CPython的缺点进行了改良，在性能上优于CPython，但对三方库的支持略差）来运行上面的代码，你会发现所有的输出都是True。

题目005：Lambda函数 是什么，举例说明的它的应用场景。

点评：这个题目主要想考察的是Lambda函数的应用场景，潜台词是问你在项目中有没有使用过Lambda函数，具体在什么场景下会用到Lambda函数，借此来判断你写代码的能力。因为Lambda函数通常用在高阶函数中，主要的作用是通过向函数传入函数或让函数返回函数最终实现代码的解耦合。

Lambda函数也叫匿名函数，它是功能简单用一行代码就能实现的小型函数。Python中的Lambda函数只能写一个表达式，这个表达式的执行结果就是函数的返回值，不用写return关键字。Lambda函数因为没有名字，所以也不会跟其他函数发生命名冲突的问题。

扩展：面试的时候有可能还会考你用Lambda函数来实现一些功能，也就是用一行代码来实现题目要求的功能，例如：用一行代码实现求阶乘的函数，用一行代码实现求最大公约数的函数等。

```
fac = lambda x: __import__('functools').reduce(int.__mul__, range(1, x + 1), 1)
gcd = lambda x, y: y % x and gcd(y % x, x) or x
```

Lambda函数其实最为主要的用途是把一个函数传入另一个高阶函数（如Python内置的filter、map等）中来为函数做解耦合，增强函数的灵活性和通用性。下面的例子通过使用filter和map函数，实现了从列表中筛选出奇数并求平方构成新列表的操作，因为用到了高阶函数，过滤和映射数据的规则都是函数的调用者通过另外一个函数传入的，因此这filter和map函数没有跟特定的过滤和映射数据的规则耦合在一起。

```
items = [12, 5, 7, 10, 8, 19]
items = list(map(lambda x: x ** 2, filter(lambda x: x % 2, items)))
print(items)    # [25, 49, 361]
```

扩展：用列表的生成式来实现上面的代码会更加简单明了，代码如下所示。

```
items = [12, 5, 7, 10, 8, 19]
items = [x ** 2 for x in items if x % 2]
print(items)    # [25, 49, 361]
```

题目006：说说Python中的浅拷贝和深拷贝。

点评：这个题目本身出现的频率非常高，但是就题论题而言没有什么技术含量。对于这种面试题，在回答的时候一定要让你的答案能够超出面试官的预期，这样才能获得更好的印象分。所以回答这个题目的要点不仅仅是能够说出浅拷贝和深拷贝的区别，深拷贝的时候可能遇到的两大问题，还要说出Python标准库对浅拷贝和深拷贝的支持，然后可以说说列表、字典如何实现拷贝操作以及如何通过序列化和反序列的方式实现深拷贝，最后还可以提到设计模式中的原型模式以及它在项目中的应用。

浅拷贝通常只复制对象本身，而深拷贝不仅会复制对象，还会递归的复制对象所关联的对象。深拷贝可能会遇到两个问题：一是一个对象如果直接或间接的引用了自身，会导致无休止的递归拷贝；二是深拷贝可能对原本设计为多个对象共享的数据也进行拷贝。Python通过copy模块中的copy和deepcopy函数来实现浅拷贝和深拷贝操作，其中deepcopy可以通过memo字典来保存已经拷贝过的对象，从而避免刚才所说的自引用递归问题；此外，可以通过copyreg模块的pickle函数来定制指定类型对象的拷贝行为。

deepcopy函数的本质其实就是对象的一次序列化和一次返回序列化，面试题中还考过用自定义函数实现对象的深拷贝操作，显然我们可以使用pickle模块的dumps和loads来做到，代码如下所示。

```
import pickle

my_deep_copy = lambda obj: pickle.loads(pickle.dumps(obj))
```

列表的切片操作[:]相当于实现了列表对象的浅拷贝，而字典的copy方法可以实现字典对象的浅拷贝。对象拷贝其实是更为快捷的创建对象的方式。在Python中，通过构造器创建对象属于两阶段构造，首先是分配内存空间，然后是初始化。在创建对象时，我们也可以基于“原型”对象来创建新对象，通过对原型对象的拷贝（复制内存）就完成了对象的创建和初始化，这种做法更加高效，这也就是设计模式中的原型模式。在Python中，我们可以通过元类的方式来实现原型模式，代码如下所示。

```
import copy


class PrototypeMeta(type):
    """实现原型模式的元类"""

    def __init__(cls, *args, **kwargs):
        super().__init__(*args, **kwargs)
        # 为对象绑定clone方法来实现对象拷贝
        cls.clone = lambda self, is_deep=True: \
            copy.deepcopy(self) if is_deep else copy.copy(self)

```

```
13
14
15  class Person(metaclass=PrototypeMeta):
        pass
```

题目007：Python是如何实现内存管理的?

点评：当面试官问到这个问题的时候，一个展示自己的机会就摆在面前了。你要先反问面试官：“你说的是官方的CPython解释器吗？”。这个反问可以展示出你了解过Python解释器的不同的实现版本，而且你也知道面试官想问的是CPython。当然，很多面试官对不同的Python解释器底层实现到底有什么差别也没有概念。所以，千万不要觉得面试官一定比你强，怀揣着这份自信可以让你更好的完成面试。

Python提供了自动化的内存管理，也就是说内存空间的分配与释放都是由Python解释器在运行时自动进行的，自动管理内存功能极大的减轻程序员的工作负担，也能够帮助程序员在一定程度上解决内存泄露的问题。以CPython解释器为例，它的内存管理有三个关键点：引用计数、标记清理、分代收集。

引用计数：对于CPython解释器来说，Python中的每一个对象其实就是PyObject结构体，它的内部有一个名为ob_refcnt 的引用计数器成员变量。程序在运行的过程中ob_refcnt的值会被更新并藉此来反映引用有多少个变量引用到该对象。当对象的引用计数值为0时，它的内存就会被释放掉。

```
1  typedef struct _object {
2      _PyObject_HEAD_EXTRA
3      Py_ssize_t ob_refcnt;
4      struct _typeobject *ob_type;
5  } PyObject;
```

以下情况会导致引用计数加1：

对象被创建

对象被引用

对象作为参数传入到一个函数中

对象作为元素存储到一个容器中

以下情况会导致引用计数减1：

用del语句显示删除对象引用

对象引用被重新赋值其他对象

一个对象离开它所在的作用域

持有该对象的容器自身被销毁

持有该对象的容器删除该对象

可以通过sys模块的getrefcount函数来获得对象的引用计数。引用计数的内存管理方式在遇到循环引用的时候就会出现致命伤，因此需要其他的垃圾回收算法对其进行补充。

标记清理：CPython使用了“标记-清理”（Mark and Sweep）算法解决容器类型可能产生的循环引用问题。该算法在垃圾回收时分为两个阶段：标记阶段，遍历所有的对象，如果对象是可达的（被其他对象引用），那么就标记该对象为可达；清除阶段，再次遍历对象，如果发现某个对象没有标记为可达，则就将其回收。CPython底层维护了两个双端链表，一个链表存放着需要被扫描的容器对象（姑且称之为链表A），另一个链表存放着临时不可达对象（姑且称之为链表B）。为了实现“标记-清理”算法，链表中的每个节点除了有记录当前引用计数的ref_count变量外，还有一个gc_ref变量，这个gc_ref是ref_count的一个副本，所以初始值为ref_count的大小。执行垃圾回收时，首先遍历链表A中的节点，并且将当前对象所引用的所有对象的gc_ref减1，这一步主要作用是解除循环引用对引用计数的影响。再次遍历链表A中的节点，如果节点的gc_ref值为0，那么这个对象就被标记为“暂时不可达”（GC_TENTATIVELY_UNREACHABLE）并被移动到链表B中；如果节点的gc_ref不为0，那么这个对象就会被标记为“可达“（GC_REACHABLE），对于”可达“对象，还要递归的将该节点可以到达的节点标记为”可达“；链表B中被标记为”可达“的节点要重新放回到链表A中。在两次遍历之后，链表B中的节点就是需要释放内存的节点。

分代回收：在循环引用对象的回收中，整个应用程序会被暂停，为了减少应用程序暂停的时间，Python 通过分代回收（空间换时间）的方法提高垃圾回收效率。分代回收的基本思想是：对象存在的时间越长，是垃圾的可能性就越小，应该尽量不对这样的对象进行垃圾回收。CPython将对象分为三种世代分别记为0、1、2，每一个新生对象都在第0代中，如果该对象在一轮垃圾回收扫描中存活下来，那么它将被移到第1代中，存在于第1代的对象将较少的被垃圾回收扫描到；如果在对第1代进行垃圾回收扫描时，这个对象又存活下来，那么它将被移至第2代中，在那里它被垃圾回收扫描的次数将会更少。分代回收扫描的门限值可以通过gc模块的get_threshold函数来获得，该函数返回一个三元组，分别表示多少次内存分配操作后会执行0代垃圾回收，多少次0代垃圾回收后会执行1代垃圾回收，多少次1代垃圾回收后会执行2代垃圾回收。需要说明的是，如果执行一次2代垃圾回收，那么比它年轻的代都要执行垃圾回收。如果想修改这几个门限值，可以通过gc模块的set_threshold函数来做到。

题目008：说一下你对Python中迭代器和生成器的理解。

点评：很多人面试者都会写迭代器和生成器，但是却无法准确的解释什么是迭代器和生成器。如果你也有同样的困惑，可以参考下面的回答。

迭代器是实现了迭代器协议的对象。跟其他编程语言不通，Python中没有用于定义协议或表示约定的关键字，像interface、protocol这些单词并不在Python语言的关键字列表中。Python语言通过魔法方法来表示约定，也就是我们所说的协议，而__next__和__iter__这两个魔法方法就代表了迭代器协议。可以通过for-in循环从迭代器对象中取出值，也可以使用next函数取出迭代器对象中的下一个值。生成器是迭代器的语法升级版本，可以用更为简单的代码来实现一个迭代器。

扩展：面试中经常让写生成斐波那契数列的迭代器，大家可以参考下面的代码。

```
class Fib(object):

    def __init__(self, num):
        self.num = num
        self.a, self.b = 0, 1
        self.idx = 0


```

```
    def __iter__(self):
        return self

    def __next__(self):
        if self.idx < self.num:
            self.a, self.b = self.b, self.a + self.b
            self.idx += 1
```

如果用生成器的语法来改写上面的代码，代码会简单优雅很多。

```
def fib(num):
    a, b = 0, 1
    for _ in range(num):
        a, b = b, a + b
        yield a
```

题目009：正则表达式的match方法和search方法有什么区别?

点评：正则表达式是字符串处理的重要工具，所以也是面试中经常考察的知识点。在Python中，使用正则表达式有两种方式，一种是直接调用re模块中的函数，传入正则表达式和需要处理的字符串；一种是先通过re模块的compile函数创建正则表达式对象，然后再通过对象调用方法并传入需要处理的字符串。如果一个正则表达式被频繁的使用，我们推荐用re.compile函数创建正则表达式对象，这样会减少频繁编译同一个正则表达式所造成的开销。

match方法是从字符串的起始位置进行正则表达式匹配，返回Match对象或None。search方法会扫描整个字符串来找寻匹配的模式，同样也是返回Match对象或None。

题目010：下面这段代码的执行结果是什么。

```
def multiply():
    return [lambda x: i * x for i in range(4)]

print([m(100) for m in multiply()])
```

运行结果：

```
[300, 300, 300, 300]
```

上面代码的运行结果很容易被误判为[0, 100, 200, 300]。首先需要注意的是multiply函数用生成式语法返回了一个列表，列表中保存了4个Lambda函数，这4个Lambda函数会返回传入的参数乘以i的结果。需要注意的是这里有闭包（closure）现象，multiply函数中的局部变量i的生命周期被延展了，由于i最终的值是3，所以通过m(100)调列表中的Lambda函数时会返回300，而且4个调用都是如此。

如果想得到[0, 100, 200, 300]这个结果，可以按照下面几种方式来修改multiply函数。

方法一：使用生成器，让函数获得i的当前值。

```
def multiply():
    return (lambda x: i * x for i in range(4))

print([m(100) for m in multiply()])
```

或者

```
def multiply():
    for i in range(4):
        yield lambda x: x * i

print([m(100) for m in multiply()])
```

方法二：使用偏函数，彻底避开闭包。

```
from functools import partial
from operator import __mul__

def multiply():
    return [partial(__mul__, i) for i in range(4)]

print([m(100) for m in multiply()])
```

题目011：Python中为什么没有函数重载?

点评：C++、Java、C#等诸多编程语言都支持函数重载，所谓函数重载指的是在同一个作用域中有多个同名函数，它们拥有不同的参数列表（参数个数不同或参数类型不同或二者皆不同），可以相互区分。重载也是一种多态性，因为通常是在编译时通过参数的个数和类型来确定到底调用哪个重载函数，所以也被称为编译时多态性或者叫前绑定。这个问题的潜台词其实是问面试者是否有其他编程语言的经验，是否理解Python是动态类型语言，是否知道Python中函数的可变参数、关键字参数这些概念。

首先Python是解释型语言，函数重载现象通常出现在编译型语言中。其次Python是动态类型语言，函数的参数没有类型约束，也就无法根据参数类型来区分重载。再者Python中函数的参数可以有默认值，可以使用可变参数和关键字参数，因此即便没有函数重载，也要可以让一个函数根据调用者传入的参数产生不同的行为。

题目012：用Python代码实现Python内置函数max。

点评：这个题目看似简单，但实际上还是比较考察面试者的功底。因为Python内置的max函数既可以传入可迭代对象找出最大，又可以传入两个或多个参数找出最大；最为关键的是还可以通过命名关键字参数key来指定一个用于元素比较的函数，还可以通过default命名关键字参数来指定当可迭代对象为空时返回的默认值。

下面的代码仅供参考：

```
def my_max(*args, key=None, default=None):
    """
    获取可迭代对象中最大的元素或两个及以上实参中最大的元素
    :param args: 一个可迭代对象或多个元素
    :param key: 提取用于元素比较的特征值的函数，默认为None
    :param default: 如果可迭代对象为空则返回该默认值，如果没有给默认值则引发ValueError异常
    :return: 返回可迭代对象或多个元素中的最大元素
    """
    if len(args) == 1 and len(args[0]) == 0:
        if default:
            return default
        else:
            raise ValueError('max() arg is an empty sequence')
    items = args[0] if len(args) == 1 else args
    max_elem, max_value = items[0], items[0]
```

题目013：写一个函数统计传入的列表中每个数字出现的次数并返回对应的字典。

点评：送人头的题目，不解释。

```
def count_letters(items):
    result = {}
    for item in items:
        if isinstance(item, (int, float)):
            result[item] = result.get(item, 0) + 1
    return result
```

也可以直接使用Python标准库中collections模块的Counter类来解决这个问题，Counter是dict的子类，它会将传入的序列中的每个元素作为键，元素出现的次数作为值来构造字典。

```
from collections import Counter

def count_letters(items):
    counter = Counter(items)
```

```
    return {key: value for key, value in counter.items() \
            if isinstance(key, (int, float))}
```

题目014：使用Python代码实现遍历一个文件夹的操作。

点评：基本也是送人头的题目，只要用过os模块就应该知道怎么做。

Python标准库os模块的walk函数提供了遍历一个文件夹的功能，它返回一个生成器。

```
import os

g = os.walk('/Users/Hao/Downloads/')
for path, dir_list, file_list in g:
    for dir_name in dir_list:
        print(os.path.join(path, dir_name))
    for file_name in file_list:
        print(os.path.join(path, file_name))
```

说明：os.path模块提供了很多进行路径操作的工具函数，在项目开发中也是经常会用到的。如果题目明确要求不能使用os.walk函数，那么可以使用os.listdir函数来获取指定目录下的文件和文件夹，然后再通过循环遍历用os.isdir函数判断哪些是文件夹，对于文件夹可以通过递归调用进行遍历，这样也可以实现遍历一个文件夹的操作。

题目015：现有2元、3元、5元共三种面额的货币，如果需要找零99元，一共有多少种找零的方式?

点评：还有一个非常类似的题目：“一个小朋友走楼梯，一次可以走1个台阶、2个台阶或3个台阶，问走完10个台阶一共有多少种走法？”，这两个题目的思路是一样，如果用递归函数来写的话非常简单。

```
from functools import lru_cache


@lru_cache()
def change_money(total):
    if total == 0:
        return 1
    if total < 0:
        return 0
    return change_money(total - 2) + change_money(total - 3) + \
        change_money(total - 5)
```

说明：在上面的代码中，我们用lru_cache装饰器装饰了递归函数change_money，如果不做这个优化，上面代码的渐近时间复杂度将会是图片，而如果参数total的值是99，这个运算量是非常巨大的。lru_cache装饰器会缓存函数的执行结果，这样就可以减少重复运算所造成的开销，这是空间换时间的策略，也是动态规划的编程思想。

题目016：写一个函数，给定矩阵的阶数 n ，输出一个螺旋式数字矩阵。

例如：n = 2，返回：

```
1 2
4 3
```

例如：n = 3，返回：

```
1 2 3
8 9 4
7 6 5
```

这个题目本身并不复杂，下面的代码仅供参考。

```
def show_spiral_matrix(n):
    matrix = [[0] * n for _ in range(n)]
    row, col = 0, 0
    num, direction = 1, 0
    while num <= n ** 2:
        if matrix[row][col] == 0:
            matrix[row][col] = num
            num += 1
        if direction == 0:
            if col < n - 1 and matrix[row][col + 1] == 0:
                col += 1
            else:
                direction += 1
        elif direction == 1:
            if row < n - 1 and matrix[row + 1][col] == 0:
```

题目017：阅读下面的代码，写出程序的运行结果。

```
items = [1, 2, 3, 4]
print([i for i in items if i > 2])
print([i for i in items if i % 2])
```

```
print([(x, y) for x, y in zip('abcd', (1, 2, 3, 4, 5))])
print({x: f'item{x ** 2}' for x in (2, 4, 6)})
print(len({x for x in 'hello world' if x not in 'abcdefg'}))
```

点评：生成式（推导式）属于Python的特色语法之一，几乎是面试必考内容。Python中通过生成式字面量语法，可以创建出列表、集合、字典。

```
[3, 4]
[1, 3]
[('a', 1), ('b', 2), ('c', 3), ('d', 4)]
{2: 'item4', 4: 'item16', 6: 'item36'}
6
```

题目018：说出下面代码的运行结果。

```
class Parent:
    x = 1

class Child1(Parent):
    pass

class Child2(Parent):
    pass

print(Parent.x, Child1.x, Child2.x)
Child1.x = 2
print(Parent.x, Child1.x, Child2.x)
Parent.x = 3
print(Parent.x, Child1.x, Child2.x)
```

点评：运行上面的代码首先输出1 1 1，这一点大家应该没有什么疑问。接下来，通过Child1.x = 2给类Child1重新绑定了属性x并赋值为2，所以Child1.x会输出2，而Parent和Child2并不受影响。执行Parent.x = 3会重新给Parent类的x属性赋值为3，由于Child2的x属性继承自Parent，所以Child2.x的值也是3；而之前我们为Child1重新绑定了x属性，那么它的x属性值不会受到Parent.x = 3的影响，还是之前的值2。

```
1 1 1
1 2 1
3 2 3
```

题目19：说说你用过Python标准库中的哪些模块。

点评：Python标准库中的模块非常多，建议大家根据自己过往的项目经历来介绍你用过的标准库和三方库，因为这些是你最为熟悉的，经得起面试官深挖的。

模块名	介绍
sys	跟Python解释器相关的变量和函数，例如：`sys.version`、`sys.exit()`
os	和操作系统相关的功能，例如：`os.listdir()`、`os.remove()`
re	和正则表达式相关的功能，例如：`re.compile()`、`re.search()`
math	和数学运算相关的功能，例如：`math.pi`、`math.e`、`math.cos`
logging	和日志系统相关的类和函数，例如：`logging.Logger`、`logging.Handler`
json / pickle	实现对象序列化和反序列的模块，例如：`json.loads`、`json.dumps`
hashlib	封装了多种哈希摘要算法的模块，例如：`hashlib.md5`、`hashlib.sha1`
urllib	包含了和URL相关的子模块，例如：`urllib.request`、`urllib.parse`
itertools	提供各种迭代器的模块，例如：`itertools.cycle`、`itertools.product`
functools	函数相关工具模块，例如：`functools.partial`、`functools.lru_cache`
collections / heapq	封装了常用数据结构和算法的模块，例如：`collections.deque`
threading / multiprocessing	多线程/多进程相关类和函数的模块，例如：`threading.Thread`

concurrent.futures / asyncio	并发编程/异步编程相关的类和函数的模块，例如：`ThreadPoolExecutor`
base64	提供BASE-64编码相关函数的模块，例如：`bas64.encode`
csv	和读写CSV文件相关的模块，例如：`csv.reader`、`csv.writer`
profile / cProfile / pstats	和代码性能剖析相关的模块，例如：`cProfile.run`、`pstats.Stats`
unittest	和单元测试相关的模块，例如：`unittest.TestCase`

题目20：`__init__`和`__new__`方法有什么区别?

Python中调用构造器创建对象属于两阶段构造过程，首先执行__new__方法获得保存对象所需的内存空间，再通过__init__执行对内存空间数据的填充（对象属性的初始化）。__new__方法的返回值是创建好的Python对象（的引用），而__init__方法的第一个参数就是这个对象（的引用），所以在__init__中可以完成对对象的初始化操作。__new__是类方法，它的第一个参数是类，__init__是对象方法，它的第一个参数是对象。

题目21：输入年月日，判断这个日期是这一年的第几天。

方法一：不使用标准库中的模块和函数。

```
def is_leap_year(year):
    """判断指定的年份是不是闰年，平年返回False，闰年返回True"""
    return year % 4 == 0 and year % 100 != 0 or year % 400 == 0

def which_day(year, month, date):
    """计算传入的日期是这一年的第几天"""
    # 用嵌套的列表保存平年和闰年每个月的天数
    days_of_month = [
        [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31],
        [31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]
    ]
    days = days_of_month[is_leap_year(year)][:month - 1]
    return sum(days) + date
```

方法二：使用标准库中的datetime模块。

```
import datetime

def which_day(year, month, date):
    end = datetime.date(year, month, date)
    start = datetime.date(year, 1, 1)
    return (end - start).days + 1
```

题目22：平常工作中用什么工具进行静态代码分析。

点评：静态代码分析工具可以从代码中提炼出各种静态属性，这使得开发者可以对代码的复杂性、可维护性和可读性有更好的了解，这里所说的静态属性包括：

代码是否符合编码规范，例如：PEP-8。

代码中潜在的问题，包括：语法错误、缩进问题、导入缺失、变量覆盖等。

代码中的坏味道。

代码的复杂度。

代码的逻辑问题。

工作中静态代码分析主要用到的是Pylint和Flake8。Pylint可以检查出代码错误、坏味道、不规范的代码等问题，较新的版本中还提供了代码复杂度统计数据，可以生成检查报告。Flake8封装了Pyflakes（检查代码逻辑错误）、McCabe（检查代码复杂性）和Pycodestyle（检查代码是否符合PEP-8规范）工具，它可以执行这三个工具提供的检查。

题目23：说一下你知道的Python中的魔术方法。

点评：魔术方法也称为魔法方法，是Python中的特色语法，也是面试中的高频问题。

魔术方法	作用
`__new__`、`__init__`、`__del__`	创建和销毁对象相关
`__add__`、`__sub__`、`__mul__`、`__div__`、`__floordiv__`、`__mod__`	算术运算符相关
`__eq__`、`__ne__`、`__lt__`、`__gt__`、`__le__`、`__ge__`	关系运算符相关
`__pos__`、`__neg__`、`__invert__`	一元运算符相关
`__lshift__`、`__rshift__`、`__and__`、`__or__`、`__xor__`	位运算相关
`__enter__`、`__exit__`	上下文管理器协议
`__iter__`、`__next__`、`__reversed__`	迭代器协议
`__int__`、`__long__`、`__float__`、`__oct__`、`__hex__`	类型/进制转换相关
`__str__`、`__repr__`、`__hash__`、`__dir__`	对象表述相关
`__len__`、`__getitem__`、`__setitem__`、`__contains__`、`__missing__`	序列相关
`__copy__`、`__deepcopy__`	对象拷贝相关
`__call__`、`__setattr__`、`__getattr__`、`__delattr__`	其他魔术方法

题目24：函数参数 `arg` 和 `*kwargs` 分别代表什么？

Python中，函数的参数分为位置参数、可变参数、关键字参数、命名关键字参数。*args代表可变参数，可以接收0个或任意多个参数，当不确定调用者会传入多少个位置参数时，就可以使用可变参数，它会将传入的参数打包成一个元组。**kwargs代表关键字参数，可以接收用参数名=参数值的方式传入的参数，传入的参数的会打包成一个字典。定义函数时如果同时使用*args和**kwargs，那么函数可以接收任意参数。

题目25：写一个记录函数执行时间的装饰器。

点评：高频面试题，也是最简单的装饰器，面试者必须要掌握的内容。

方法一：用函数实现装饰器。

```
from functools import wraps
from time import time


def record_time(func):

    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time()
        result = func(*args, **kwargs)
        print(f'{func.__name__}执行时间: {time() - start}秒')
        return result

    return wrapper
```

方法二：用类实现装饰器。类有__call__魔术方法，该类对象就是可调用对象，可以当做装饰器来使用。

```
from functools import wraps
from time import time


class Record:

    def __call__(self, func):

        @wraps(func)
        def wrapper(*args, **kwargs):
            start = time()
            result = func(*args, **kwargs)

```

```
            print(f'{func.__name__}执行时间: {time() - start}秒')
            return result
```

说明：装饰器可以用来装饰类或函数，为其提供额外的能力，属于设计模式中的代理模式。

扩展：装饰器本身也可以参数化，例如上面的例子中，如果不希望在终端中显示函数的执行时间而是希望由调用者来决定如何输出函数的执行时间，可以通过参数化装饰器的方式来做到，代码如下所示。

```
from functools import wraps
from time import time


def record_time(output):
    """可以参数化的装饰器"""

    def decorate(func):

        @wraps(func)
        def wrapper(*args, **kwargs):
            start = time()
            result = func(*args, **kwargs)
            output(func.__name__, time() - start)
            return result
```

题目26：什么是鸭子类型（duck typing）？

鸭子类型是动态类型语言判断一个对象是不是某种类型时使用的方法，也叫做鸭子判定法。简单的说，鸭子类型是指判断一只鸟是不是鸭子，我们只关心它游泳像不像鸭子、叫起来像不像鸭子、走路像不像鸭子就足够了。换言之，如果对象的行为跟我们的预期是一致的（能够接受某些消息），我们就认定它是某种类型的对象。

在Python语言中，有很多bytes-like对象（如：bytes、bytearray、array.array、memoryview）、file-like对象（如：StringIO、BytesIO、GzipFile、socket）、path-like对象（如：str、bytes），其中file-like对象都能支持read和write操作，可以像文件一样读写，这就是所谓的对象有鸭子的行为就可以判定为鸭子的判定方法。再比如Python中列表的extend方法，它需要的参数并不一定要是列表，只要是可迭代对象就没有问题。

说明：动态语言的鸭子类型使得设计模式的应用被大大简化。

题目27：说一下Python中变量的作用域。

Python中有四种作用域，分别是局部作用域（Local）、嵌套作用域（Embedded）、全局作用域（Global）、内置作用域（Built-in），搜索一个标识符时，会按照LEGB的顺序进行搜索，如果所有的作用域中都没有找到这个标识符，就会引发NameError异常。

题目28：说一下你对闭包的理解。

闭包是支持一等函数的编程语言（Python、JavaScript等）中实现词法绑定的一种技术。当捕捉闭包的时候，它的自由变量（在函数外部定义但在函数内部使用的变量）会在捕捉时被确定，这样即便脱离了捕捉时的上下文，它也能照常运行。简单的说，可以将闭包理解为能够读取其他函数内部变量的函数。正在情况下，函数的局部变量在函数调用结束之后就结束了生命周期，但是闭包使得局部变量的生命周期得到了延展。使用闭包的时候需要注意，闭包会使得函数中创建的对象不会被垃圾回收，可能会导致很大的内存开销，所以闭包一定不能滥用。

题目29：说一下Python中的多线程和多进程的应用场景和优缺点。

线程是操作系统分配CPU的基本单位，进程是操作系统分配内存的基本单位。通常我们运行的程序会包含一个或多个进程，而每个进程中又包含一个或多个线程。多线程的优点在于多个线程可以共享进程的内存空间，所以进程间的通信非常容易实现；但是如果使用官方的CPython解释器，多线程受制于GIL（全局解释器锁），并不能利用CPU的多核特性，这是一个很大的问题。使用多进程可以充分利用CPU的多核特性，但是进程间通信相对比较麻烦，需要使用IPC机制（管道、套接字等）。

多线程适合那些会花费大量时间在I/O操作上，但没有太多并行计算需求且不需占用太多内存的I/O密集型应用。多进程适合执行计算密集型任务（如：视频编码解码、数据处理、科学计算等）、可以分解为多个并行子任务并能合并子任务执行结果的任务以及在内存使用方面没有任何限制且不强依赖于I/O操作的任务。

扩展：Python中实现并发编程通常有多线程、多进程和异步编程三种选择。异步编程实现了协作式并发，通过多个相互协作的子程序的用户态切换，实现对CPU的高效利用，这种方式也是非常适合I/O密集型应用的。

题目30：说一下Python 2和Python 3的区别。

点评：这种问题千万不要背所谓的参考答案，说一些自己最熟悉的就足够了。

Python 2中的print和exec都是关键字，在Python 3中变成了函数。

Python 3中没有long类型，整数都是int类型。

Python 2中的不等号<>在Python 3中被废弃，统一使用!=。

Python 2中的xrange函数在Python 3中被range函数取代。

Python 3对Python 2中不安全的input函数做出了改进，废弃了raw_input函数。

Python 2中的file函数被Python 3中的open函数取代。

Python 2中的/运算对于int类型是整除，在Python 3中要用//来做整除除法。

Python 3中改进了Python 2捕获异常的代码，很明显Python 3的写法更合理。

Python 3生成式中循环变量的作用域得到了更好的控制，不会影响到生成式之外的同名变量。

Python 3中的round函数可以返回int或float类型，Python 2中的round函数返回float类型。

Python 3的str类型是Unicode字符串，Python 2的str类型是字节串，相当于Python 3中的bytes。

Python 3中的比较运算符必须比较同类对象。

Python 3中定义类的都是新式类，Python 2中定义的类有新式类（显式继承自object的类）和旧式类（经典类）之分，新式类和旧式类在MRO问题上有非常显著的区别，新式类可以使用class__ 属性获取自身类型，新式类可以使用 __slots魔法。

Python 3对代码缩进的要求更加严格，如果混用空格和制表键会引发TabError。

Python 3中字典的keys、values、items方法都不再返回list对象，而是返回view object，内置的map、filter等函数也不再返回list对象，而是返回迭代器对象。

Python 3标准库中某些模块的名字跟Python 2是有区别的；而在三方库方面，有些三方库只支持Python 2，有些只能支持Python 3。

题目31：谈谈你对“猴子补丁”（monkey patching）的理解。

“猴子补丁”是动态类型语言的一个特性，代码运行时在不修改源代码的前提下改变代码中的方法、属性、函数等以达到热补丁（hot patch）的效果。很多系统的安全补丁也是通过猴子补丁的方式来实现的，但实际开发中应该避免对猴子补丁的使用，以免造成代码行为不一致的问题。

在使用gevent库的时候，我们会在代码开头的地方执行gevent.monkey.patch_all()，这行代码的作用是把标准库中的socket模块给替换掉，这样我们在使用socket的时候，不用修改任何代码就可以实现对代码的协程化，达到提升性能的目的，这就是对猴子补丁的应用。

另外，如果希望用ujson三方库替换掉标准库中的json，也可以使用猴子补丁的方式，代码如下所示。

```
import json, ujson

json.__name__ = 'ujson'
json.dumps = ujson.dumps
json.loads = ujson.loads
```

单元测试中的Mock技术也是对猴子补丁的应用，Python中的unittest.mock模块就是解决单元测试中用Mock对象替代被测对象所依赖的对象的模块。

题目32：阅读下面的代码说出运行结果。

```
class A:
    def who(self):
        print('A', end='')

class B(A):
```

```
    def who(self):
        super(B, self).who()
        print('B', end='')

class C(A):
    def who(self):
        super(C, self).who()
        print('C', end='')

class D(B, C):
```

点评：这道题考查到了两个知识点：

Python中的MRO（方法解析顺序）。在没有多重继承的情况下，向对象发出一个消息，如果对象没有对应的方法，那么向上（父类）搜索的顺序是非常清晰的。如果向上追溯到object类（所有类的父类）都没有找到对应的方法，那么将会引发AttributeError异常。但是有多重继承尤其是出现菱形继承（钻石继承）的时候，向上追溯到底应该找到那个方法就得确定MRO。Python 3中的类以及Python 2中的新式类使用C3算法来确定MRO，它是一种类似于广度优先搜索的方法；Python 2中的旧式类（经典类）使用深度优先搜索来确定MRO。在搞不清楚MRO的情况下，可以使用类的mro方法或mro属性来获得类的MRO列表。

super()函数的使用。在使用super函数时，可以通过super(类型, 对象)来指定对哪个对象以哪个类为起点向上搜索父类方法。所以上面B类代码中的super(B, self).who()表示以B类为起点，向上搜索self（D类对象）的who方法，所以会找到C类中的who方法，因为D类对象的MRO列表是D --> B --> C --> A --> object。

```
ACBD
```

题目33：编写一个函数实现对逆波兰表达式求值，不能使用Python的内置函数。

点评：逆波兰表达式也称为“后缀表达式”，相较于平常我们使用的“中缀表达式”，逆波兰表达式不需要括号来确定运算的优先级，例如5 * (2 + 3)对应的逆波兰表达式是5 2 3 + *。逆波兰表达式求值需要借助栈结构，扫描表达式遇到运算数就入栈，遇到运算符就出栈两个元素做运算，将运算结果入栈。表达式扫描结束后，栈中只有一个数，这个数就是最终的运算结果，直接出栈即可。

```
import operator


class Stack:
    """栈（FILO）"""

    def __init__(self):
        self.elems = []

    def push(self, elem):
```

```
        """入栈"""
        self.elems.append(elem)

    def pop(self):
        """出栈"""
```

题目34：Python中如何实现字符串替换操作?

Python中实现字符串替换大致有两类方法：字符串的replace方法和正则表达式的sub方法。

方法一：使用字符串的replace方法。

```
message = 'hello, world!'
print(message.replace('o', '0').replace('l', 'L').replace('he', 'HE'))
```

方法二：使用正则表达式的sub方法。

```
import re

message = 'hello, world!'
pattern = re.compile('[aeiou]')
print(pattern.sub('#', message))
```

扩展：还有一个相关的面试题，对保存文件名的列表排序，要求文件名按照字母表和数字大小进行排序，例如对于列表filenames = [‘a12.txt’, ‘a8.txt’, ‘b10.txt’, ‘b2.txt’, ‘b19.txt’, ‘a3.txt’]，排序的结果是[‘a3.txt’, ‘a8.txt’, ‘a12.txt’, ‘b2.txt’, ‘b10.txt’, ‘b19.txt’]。提示一下，可以通过字符串替换的方式为文件名补位，根据补位后的文件名用sorted函数来排序，大家可以思考下这个问题如何解决。

题目35：如何剖析Python代码的执行性能?

剖析代码性能可以使用Python标准库中的cProfile和pstats模块，cProfile的run函数可以执行代码并收集统计信息，创建出Stats对象并打印简单的剖析报告。Stats是pstats模块中的类，它是一个统计对象。当然，也可以使用三方工具line_profiler和memory_profiler来剖析每一行代码耗费的时间和内存，这两个三方工具都会用非常友好的方式输出剖析结构。如果使用PyCharm，可以利用“Run”菜单的“Profile”菜单项对代码进行性能分析，PyCharm中可以用表格或者调用图（Call Graph）的方式来显示性能剖析的结果。

下面是使用cProfile剖析代码性能的例子。

example.py

```
import cProfile

```

```
def is_prime(num):
    for factor in range(2, int(num ** 0.5) + 1):
        if num % factor == 0:
            return False
    return True


class PrimeIter:

    def __init__(self, total):
        self.counter = 0
        self.current = 1
```

如果使用line_profiler三方工具，可以直接剖析is_prime函数每行代码的性能，需要给is_prime函数添加一个profiler装饰器，代码如下所示。

```
@profiler
def is_prime(num):
    for factor in range(2, int(num ** 0.5) + 1):
        if num % factor == 0:
            return False
    return True
```

安装line_profiler。

```
pip install line_profiler
```

使用line_profiler。

```
kernprof -lv example.py
```

运行结果如下所示。

```
Line #    Hits    Time      Per Hit  % Time  Line Contents
==============================================================
     1                                    @profile
     2                                    def is_prime(num):
     3    86624   48420.0   0.6      50.5        for factor in range(2, int(num ** 0.5) + 1):
     4    85624   44000.0   0.5      45.9            if num % factor == 0:
```

```
5    6918    3080.0   0.4      3.2                return False
6    1000     430.0   0.4      0.4        return True
```

题目36：如何使用 random 模块生成随机数、实现随机乱序和随机抽样?

点评：送人头的题目，因为Python标准库中的常用模块应该是Python开发者都比较熟悉的内容，这个问题回如果答不上来，整个面试基本也就砸锅了。

random.random()函数可以生成[0.0, 1.0)之间的随机浮点数。

random.uniform(a, b)函数可以生成[a, b]或[b, a]之间的随机浮点数。

random.randint(a, b)函数可以生成[a, b]或[b, a]之间的随机整数。

random.shuffle(x)函数可以实现对序列x的原地随机乱序。

random.choice(seq)函数可以从非空序列中取出一个随机元素。

random.choices(population, weights=None, *, cum_weights=None, k=1)函数可以从总体中随机抽取（有放回抽样）出容量为k的样本并返回样本的列表，可以通过参数指定个体的权重，如果没有指定权重，个体被选中的概率均等。

random.sample(population, k)函数可以从总体中随机抽取（无放回抽样）出容量为k的样本并返回样本的列表。

扩展：random模块提供的函数除了生成均匀分布的随机数外，还可以生成其他分布的随机数，例如random.gauss(mu, sigma)函数可以生成高斯分布（正态分布）的随机数；random.paretovariate(alpha)函数会生成帕累托分布的随机数；random.gammavariate(alpha, beta)函数会生成伽马分布的随机数。

题目37：解释一下线程池的工作原理。

点评：池化技术就是一种典型空间换时间的策略，我们使用的数据库连接池、线程池等都是池化技术的应用，Python标准库currrent.futures模块的ThreadPoolExecutor就是线程池的实现，如果要弄清楚它的工作原理，可以参考下面的内容。

线程池是一种用于减少线程本身创建和销毁造成的开销的技术，属于典型的空间换时间操作。如果应用程序需要频繁的将任务派发到线程中执行，线程池就是必选项，因为创建和释放线程涉及到大量的系统底层操作，开销较大，如果能够在应用程序工作期间，将创建和释放线程的操作变成预创建和借还操作，将大大减少底层开销。线程池在应用程序启动后，立即创建一定数量的线程，放入空闲队列中。这些线程最开始都处于阻塞状态，不会消耗CPU资源，但会占用少量的内存空间。当任务到来后，从队列中取出一个空闲线程，把任务派发到这个线程中运行，并将该线程标记为已占用。当线程池中所有的线程都被占用后，可以选择自动创建一定数量的新线程，用于处理更多的任务，也可以选择让任务排队等待直到有空闲的线程可用。在任务执行完毕后，线程并不退出结束，而是继续保持在池中等待下一次的任务。当系统比较空闲时，大部分线程长时间处于闲置状态时，线程池可以自动销毁一部分线程，回收系统资源。基于这种预创建技术，线程池将线程创建和销毁本身所带来的开销分摊到了各个具体的任务上，执行次数越多，每个任务所分担到的线程本身开销则越小。

一般线程池都必须具备下面几个组成部分：

线程池管理器：用于创建并管理线程池。

工作线程和线程队列：线程池中实际执行的线程以及保存这些线程的容器。

任务接口：将线程执行的任务抽象出来，形成任务接口，确保线程池与具体的任务无关。

任务队列：线程池中保存等待被执行的任务的容器。

题目38：举例说明什么情况下会出现`KeyError`、`TypeError`、`ValueError`。

举一个简单的例子，变量a是一个字典，执行int(a['x'])这个操作就有可能引发上述三种类型的异常。如果字典中没有键x，会引发KeyError；如果键x对应的值不是str、float、int、bool以及bytes-like类型，在调用int函数构造int类型的对象时，会引发TypeError；如果a[x]是一个字符串或者字节串，而对应的内容又无法处理成int时，将引发ValueError。

题目39：说出下面代码的运行结果。

```
def extend_list(val, items=[]):
    items.append(val)
    return items

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

点评：Python函数在定义的时候，默认参数items的值就被计算出来了，即[]。因为默认参数items引用了对象[]，每次调用该函数，如果对items引用的列表进行了操作，下次调用时，默认参数还是引用之前的那个列表而不是重新赋值为[]，所以列表中会有之前添加的元素。如果通过传参的方式为items重新赋值，那么items将引用到新的列表对象，而不再引用默认的那个列表对象。这个题在面试中经常被问到，通常不建议使用容器类型的默认参数，像PyLint这样的代码检查工具也会对这种代码提出质疑和警告。

```
[10, 'a']
[123]
[10, 'a']
```

题目40：如何读取大文件，例如内存只有4G，如何读取一个大小为8G的文件?

很显然4G内存要一次性的加载大小为8G的文件是不现实的，遇到这种情况必须要考虑多次读取和分批次处理。在Python中读取文件可以先通过open函数获取文件对象，在读取文件时，可以通过read方法的size参数指定读取的大小，也可以通过seek方法的offset参数指定读取的位置，这样就可以控制单次读取数据的字节数和总字节数。除此之外，可以使用内置函数iter将文件对象处理成迭代器对象，每次只读取少量的数据进行处理，代码大致写法如下所示。

```
with open('...', 'rb') as file:
    for data in iter(lambda: file.read(2097152), b''):
        pass
```

在Linux系统上，可以通过split命令将大文件切割为小片，然后通过读取切割后的小文件对数据进行处理。例如下面的命令将名为filename的大文件切割为大小为512M的多个文件。

```
split -b 512m filename
```

如果愿意， 也可以将名为filename的文件切割为10个文件，命令如下所示。

```
split -n 10 filename
```

扩展：外部排序跟上述的情况非常类似，由于处理的数据不能一次装入内存，只能放在读写较慢的外存储器（通常是硬盘）上。“排序-归并算法”就是一种常用的外部排序策略。在排序阶段，先读入能放在内存中的数据量，将其排序输出到一个临时文件，依此进行，将待排序数据组织为多个有序的临时文件，然后在归并阶段将这些临时文件组合为一个大的有序文件，这个大的有序文件就是排序的结果。

题目41：说一下你对Python中模块和包的理解。

每个Python文件就是一个模块，而保存这些文件的文件夹就是一个包，但是这个作为Python包的文件夹必须要有一个名为__init__.py的文件，否则无法导入这个包。通常一个文件夹下还可以有子文件夹，这也就意味着一个包下还可以有子包，子包中的__init__.py并不是必须的。模块和包解决了Python中命名冲突的问题，不同的包下可以有同名的模块，不同的模块下可以有同名的变量、函数或类。在Python中可以使用import或from … import …来导入包和模块，在导入的时候还可以使用as关键字对包、模块、类、函数、变量等进行别名，从而彻底解决编程中尤其是多人协作团队开发时的命名冲突问题。

题目42：说一下你知道的Python编码规范。

点评：企业的Python编码规范基本上是参照PEP-8或谷歌开源项目风格指南来制定的，后者还提到了可以使用Lint工具来检查代码的规范程度，面试的时候遇到这类问题，可以先说下这两个参照标准，然后挑重点说一下Python编码的注意事项。

空格的使用

使用空格来表示缩进而不要用制表符（Tab）。

和语法相关的每一层缩进都用4个空格来表示。

每行的字符数不要超过79个字符，如果表达式因太长而占据了多行，除了首行之外的其余各行都应该在正常的缩进宽度上再加上4个空格。

函数和类的定义，代码前后都要用两个空行进行分隔。

在同一个类中，各个方法之间应该用一个空行进行分隔。

二元运算符的左右两侧应该保留一个空格，而且只要一个空格就好。

标识符命名

变量、函数和属性应该使用小写字母来拼写，如果有多个单词就使用下划线进行连接。

类中受保护的实例属性，应该以一个下划线开头。

类中私有的实例属性，应该以两个下划线开头。

类和异常的命名，应该每个单词首字母大写。

模块级别的常量，应该采用全大写字母，如果有多个单词就用下划线进行连接。

类的实例方法，应该把第一个参数命名为self以表示对象自身。

类的类方法，应该把第一个参数命名为cls以表示该类自身。

表达式和语句

采用内联形式的否定词，而不要把否定词放在整个表达式的前面。例如：if a is not b就比if not a is b更容易让人理解。

不要用检查长度的方式来判断字符串、列表等是否为None或者没有元素，应该用if not x这样的写法来检查它。

就算if分支、for循环、except异常捕获等中只有一行代码，也不要将代码和if、for、except等写在一起，分开写才会让代码更清晰。

import语句总是放在文件开头的地方。

引入模块的时候，from math import sqrt比import math更好。

如果有多个import语句，应该将其分为三部分，从上到下分别是Python标准模块、第三方模块和自定义模块，每个部分内部应该按照模块名称的字母表顺序来排列。

题目43：运行下面的代码是否会报错，如果报错请说明哪里有什么样的错，如果不报错请说出代码的执行结果。

```
class A:
    def __init__(self, value):
        self.__value = value

    @property
    def value(self):
```

```
        return self.__value

obj = A(1)
obj.__value = 2
print(obj.value)
print(obj.__value)
```

点评：这道题有两个考察点，一个考察点是对_和__开头的对象属性访问权限以及@property装饰器的了解，另外一个考察的点是对动态语言的理解，不需要过多的解释。

```
1
2
```

扩展：如果不希望代码运行时动态的给对象添加新属性，可以在定义类时使用__slots__魔法。例如，我们可以在上面的A中添加一行__slots__ = ('__value',)，再次运行上面的代码，将会在原来的第10行处产生AttributeError错误。

题目44：对下面给出的字典按值从大到小对键进行排序。

```
prices = {
    'AAPL': 191.88,
    'GOOG': 1186.96,
    'IBM': 149.24,
    'ORCL': 48.44,
    'ACN': 166.89,
    'FB': 208.09,
    'SYMC': 21.29
}
```

点评：sorted函数的高阶用法在面试的时候经常出现，key参数可以传入一个函数名或一个Lambda函数，该函数的返回值代表了在排序时比较元素的依据。

```
sorted(prices, key=lambda x: prices[x], reverse=True)
```

题目45：说一下 `namedtuple` 的用法和作用。

点评：Python标准库的collections模块提供了很多有用的数据结构，这些内容并不是每个开发者都清楚，就比如题目问到的namedtuple，在我参加过的面试中，90%的面试者都不能准确的说出它的作用和应用场景。此外，deque也是一个非常有用但又经常被忽视的类，还有Counter、OrderedDict 、defaultdict 、UserDict等类，大家清楚它们的用法吗?

在使用面向对象编程语言的时候，定义类是最常见的一件事情，有的时候，我们会用到只有属性没有方法的类，这种类的对象通常只用于组织数据，并不能接收消息，所以我们把这种类称为数据类或者退化的类，就像C语言中的结构体那样。我们并不建议使用这种退化的类，在Python中可以用namedtuple（命名元组）来替代这种类。

```
from collections import namedtuple

Card = namedtuple('Card', ('suite', 'face'))
card1 = Card('红桃', 13)
card2 = Card('草花', 5)
print(f'{card1.suite}{card1.face}')
print(f'{card2.suite}{card2.face}')
```

命名元组与普通元组一样是不可变容器，一旦将数据存储在namedtuple的顶层属性中，数据就不能再修改了，也就意味着对象上的所有属性都遵循“一次写入，多次读取”的原则。和普通元组不同的是，命名元组中的数据有访问名称，可以通过名称而不是索引来获取保存的数据，不仅在操作上更加简单，代码的可读性也会更好。

命名元组的本质就是一个类，所以它还可以作为父类创建子类。除此之外，命名元组内置了一系列的方法，例如，可以通过_asdict方法将命名元组处理成字典，也可以通过_replace方法创建命名元组对象的浅拷贝。

```
class MyCard(Card):

    def show(self):
        faces = ['', 'A', '2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K']
        return f'{self.suite}{faces[self.face]}'


print(Card)    # <class '__main__.Card'>
card3 = MyCard('方块', 12)
print(card3.show())    # 方块Q
print(dict(card1._asdict()))    # {'suite': '红桃', 'face': 13}
print(card2._replace(suite='方块'))    # Card(suite='方块', face=5)
```

总而言之，命名元组能更好的组织数据结构，让代码更加清晰和可读，在很多场景下是元组、字典和数据类的替代品。在需要创建占用空间更少的不可变类时，命名元组就是很好的选择。

题目46：按照题目要求写出对应的函数。

要求：写一个函数，传入一个有若干个整数的列表，该列表中某个元素出现的次数超过了50%，返回这个元素。

```python
def more_than_half(items):
    temp, times = None, 0
    for item in items:
        if times == 0:
            temp = item
            times += 1
        else:
            if item == temp:
                times += 1
            else:
                times -= 1
    return temp
```

点评：LeetCode上的题目，在Python面试中出现过，利用元素出现次数超过了50%这一特征，出现和temp相同的元素就将计数值加1，出现和temp不同的元素就将计数值减1。如果计数值为0，说明之前出现的元素已经对最终的结果没有影响，用temp记下当前元素并将计数值置为1。最终，出现次数超过了50%的这个元素一定会被赋值给变量temp。

题目47：按照题目要求写出对应的函数。

要求：写一个函数，传入的参数是一个列表（列表中的元素可能也是一个列表），返回该列表最大的嵌套深度。例如：列表[1, 2, 3]的嵌套深度为1，列表[[1], [2, [3]]]的嵌套深度为3。

```python
def list_depth(items):
    if isinstance(items, list):
        max_depth = 1
        for item in items:
            max_depth = max(list_depth(item) + 1, max_depth)
        return max_depth
    return 0
```

点评：看到题目应该能够比较自然的想到使用递归的方式检查列表中的每个元素。

题目48：按照题目要求写出对应的装饰器。

要求：有一个通过网络获取数据的函数（可能会因为网络原因出现异常），写一个装饰器让这个函数在出现指定异常时可以重试指定的次数，并在每次重试之前随机延迟一段时间，最长延迟时间可以通过参数进行控制。

方法一：

```python
from functools import wraps
from random import random
from time import sleep


def retry(*, retry_times=3, max_wait_secs=5, errors=(Exception, )):

    def decorate(func):

        @wraps(func)
        def wrapper(*args, **kwargs):
            for _ in range(retry_times):
                try:
                    return func(*args, **kwargs)
                except errors:
```

方法二：

```python
from functools import wraps
from random import random
from time import sleep


class Retry(object):

    def __init__(self, *, retry_times=3, max_wait_secs=5, errors=(Exception, )):
        self.retry_times = retry_times
        self.max_wait_secs = max_wait_secs
        self.errors = errors

    def __call__(self, func):

        @wraps(func)
```

点评：我们不止一次强调过，装饰器几乎是Python面试必问内容，这个题目比之前的题目稍微复杂一些，它需要的是一个参数化的装饰器。

题目49：写一个函数实现字符串反转，尽可能写出你知道的所有方法。

点评：烂大街的题目，基本上算是送人头的题目。

方法一：反向切片

```
def reverse_string(content):
    return content[::-1]
```

方法二：反转拼接

```
def reverse_string(content):
    return ''.join(reversed(content))
```

方法三：递归调用

```
def reverse_string(content):
    if len(content) <= 1:
        return content
    return reverse_string(content[1:]) + content[0]
```

方法四：双端队列

```
from collections import deque

def reverse_string(content):
    q = deque()
    q.extendleft(content)
    return ''.join(q)
```

方法五：反向组装

```
from io import StringIO

def reverse_string(content):
    buffer = StringIO()
    for i in range(len(content) - 1, -1, -1):
        buffer.write(content[i])
    return buffer.getvalue()
```

方法六：反转拼接

```
def reverse_string(content):
    return ''.join([content[i] for i in range(len(content) - 1, -1, -1)])
```

方法七：半截交换

```
def reverse_string(content):
    length, content= len(content), list(content)
    for i in range(length // 2):
        content[i], content[length - 1 - i] = content[length - 1 - i], content[i]
    return ''.join(content)
```

方法八：对位交换

```
def reverse_string(content):
    length, content= len(content), list(content)
    for i, j in zip(range(length // 2), range(length - 1, length // 2 - 1, -1)):
        content[i], content[j] = content[j], content[i]
    return ''.join(content)
```

扩展：这些方法其实都是大同小异的，面试的时候能够给出几种有代表性的就足够了。给大家留一个思考题，上面这些方法，哪些做法的性能较好呢？我们之前提到过剖析代码性能的方法，大家可以用这些方法来检验下你给出的答案是否正确。

题目50：按照题目要求写出对应的函数。

要求：列表中有1000000个元素，取值范围是[1000, 10000)，设计一个函数找出列表中的重复元素。

```
def find_dup(items: list):
    dups = [0] * 9000
    for item in items:
        dups[item - 1000] += 1
    for idx, val in enumerate(dups):
        if val > 1:
            yield idx + 1000
```

点评：这道题的解法和计数排序的原理一致，虽然元素的数量非常多，但是取值范围[1000, 10000)并不是很大，只有9000个可能的取值，所以可以用一个能够保存9000个元素的dups列表来记录每个元素出现的次数，dups列表所有元素的初始值都是0，通过对items列表中元素的遍历，当出现某个元素时，将dups列表对应位置的值加1，最后dups列表中值大于1的元素对应的就是items列表中重复出现过的元素。