语言特性

1. 谈谈对 Python 和其他语言的区别

答：Python 是一门语法简洁优美,功能强大无比,应用领域非常广泛,具有强大完备的第三方库，他是一门强类型的可移植、可扩展，可嵌入的解释型编程语言，属于动态语言。

拿 C 语言和 Python 比： Python 的第三方类库比较齐全并且使用简洁,很少代码就能实现一些功能，如果用 C 去实现相同的功能可能就比较复杂。但是对于速度来说 Python 的运行速度相较于 C 就比较慢了。所以有利的同时也有弊端，毕竟我们的学习成本降低了。

2. 简述解释型和编译型编程语言

答：解释型语言是在运行程序的时候才翻译，每执行一次，要翻译一次，效率较低。 编译型就是直接编译成机型可以执行的，只翻译一次，所以效率相对来说较高。

3. Python 的解释器种类以及相关特点?

答:

- CPython c 语言开发的，使用最广的解释器
- IPython 基于 cPython 之上的一个交互式计时器，交互方式增强功能和 cPython 一样
- PyPy 目标是执行效率，采用 JIT 技术。对 Python 代码进行动态编译，提高执行效率
- JPython 运行在 Java 上的解释器，直接把 Python 代码编译成 Java 字节码执行
- IronPython 运行在微软 .NET 平台上的解释器，把 Python 编译成 . NET 的字节码。

4. Python3 和 Python2 的区别?

答： 这里例举 5 条

1. print 在 Python3 中是函数必须加括号，Python2 中 print 为 class。
2. Python2 中使用 xrange，Python3 使用 range。
3. Python2 中默认的字符串类型默认是 ASCII，Python3 中默认的字符串类型是 Unicode。
4. Python2 中/的结果是整型，Python3 中是浮点类型。
5. Python2 中声明元类：*metaclass* = MetaClass,Python3 中声明元类：class newclass(metaclass=MetaClass)：pass。

5. Python3 和 Python2 中 int 和 long 区别?

答：Python2 有 int 和 long 类型。int 类型最大值不能超过 sys.maxint，而且这个最大值是平台相关的。可以通过在数字的末尾附上一个 L 来定义长整型，显然，它比 int 类型表示的数字范围更大。在 Python3 里，只有一种整数类型 int，大多数情况下，和 Python 2 中的长整型类似。

6. xrange 和 range 的区别?

答：xrange 是在 Python2 中的用法，Python3 中只有 range xrange 用法与 range 完全相同，所不同的是生成的不是一个 list 对象，而是一个生成器。

编码规范

7. 什么是 PEP8?

答：PEP8 通常会听别人提到，但是具体的指什么内容呢，简单介绍下。《Python Enhancement Proposal #8》（8 号 Python 增强提案）又叫 PEP8，他针对的 Python 代码格式而编订的风格指南。

8. 了解 Python 之禅么?

答：通过 import this 语句可以获取其具体的内容。它告诉大家如何写出高效整洁的代码。

9. 了解 DocStrings 么?

答：DocStrings 文档字符串是一个重要工具，用于解释文档程序，帮助你的程序文档更加简单易懂。主要是解释代码作用的。

10. 了解类型注解么?

答：PEP 484 引入了类型提示，这使得可以对 Python 代码进行静态类型检查。 在使用 Ide 的时候可以获取到参数的类型，更方便传入参数。使用格式如下

```
def foo(num:  int) -> None:
    print(f"接收到的数字是：{num}")
```

介绍下这个简单例子，我们可以在函数的参数部分使用参数名+：+类型，来指定参数可以接受的类型，这里的话就是 num 参数为 int 类型，然后后面->接的是返回值的类型。这里返回值为 None，然后通过 fstring 格式化字符串输出传入的数字。

11. 例举你知道 Python 对象的命名规范，例如方法或者类等

答：

类：总是使用首字母大写单词串，如 MyClass。内部类可以使用额外的前导下划线。 变量：小写，由下划线连接各个单词。方法名类似 常量：常量名所有字母大写 等

12. Python 中的注释有几种?

答：总体来说分为两种，单行注释和多行注释。

1. 单行注释在行首是 `#`。
2. 多行注释可以使用三个单引号或三个双引号，包括要注释的内容。

13. 如何优雅的给一个函数加注释?

答：可以使用 docstring 配合类型注解

14. 如何给变量加注释?

答：可以通过变量名：类型的方式如下

```
a:  str = "this is string type"
```

15. Python 代码缩进中是否支持 Tab 键和空格混用。

答：不允许 tab 键和空格键混用，这种现象在使用 sublime 的时候尤为明显。

一般推荐使用 4 个空格替代 tab 键。

16. 是否可以在一句 import 中导入多个库?

答：可以是可以，但是不推荐。因为一次导入多个模块可读性不是很好，所以一行导入一个模块会比较好。同样的尽量少用 from modulename import *，因为判断某个函数或者属性的来源有些困难，不方便调试，可读性也降低了。

17. 在给 Py 文件命名的时候需要注意什么?

答：给文件命名的时候不要和标准库库的一些模块重复，比如 abc。 另外要名字要有意义，不建议数字开头或者中文命名。

18. 例举几个规范 Python 代码风格的工具

答：pylint 和 flake8

数据类型-字符串

19. 列举 Python 中的基本数据类型?

答： Python3 中有六个标准的数据类型：字符串（String）、数字（Digit）、列表（List）、元组（Tuple）、集合（Sets）、字典（Dictionary）。

20. 如何区别可变数据类型和不可变数据类型

答： 从对象内存地址方向来说

1. 可变数据类型：在内存地址不变的情况下，值可改变（列表和字典是可变类型，但是字典中的 key 值必须是不可变类型）
2. 不可变数据类型：内存改变，值也跟着改变。（数字，字符串，布尔类型，都是不可变类型）可以通过 id() 方法进行内存地址的检测。

21. 将"hello world"转换为首字母大写"Hello World"

答： 这个得看清题目是要求两个单词首字母都要大写，如果只是第一个单词首字母大小的话，只使用 capitalize 即可，但是这里是两个单词，所以用下面的方法。

```
arr = "hello world".split(" ")
new_str = f"{arr[0].capitalize()} {arr[1].capitalize()}"
print(new_str)
```

后来评论中有朋友提到了下面的方法，这里感谢这位朋友提醒。方案如下

```
"hello world".title()
```

非常简单一句话搞定。

22. 如何检测字符串中只含有数字?

答：可以通过 isdigit 方法，例子如下

```
s1 = "12223".isdigit()
print(s1)

s2 = "12223a".isdigit()
print(s2)

#结果如下:
#True
#False
```

23. 将字符串"ilovechina"进行反转

答：

```
s1 = "ilovechina"[: : -1]
print(s1)
```

24. Python 中的字符串格式化方式你知道哪些?

答：%s，format，fstring(Python3.6 开始才支持，现在推荐的写法)

25. 有一个字符串开头和末尾都有空格，比如“ adabdw ”，要求写一个函数把这个字符串的前后空格都去掉。

答：因为题目要是写一个函数所以我们不能直接使用 strip，不过我们可以把它封装到函数啊

```
def strip_function(s1):
    return s1.strip()

s1 = " adabdw "
print(strip_function(s1))
```

26. 获取字符串”123456“最后的两个字符。

答：切片使用的考察，最后两个即开始索引是 -2，代码如下

```
a = "123456"
print(a[-2: : ])
```

27. 一个编码为 GBK 的字符串 S，要将其转成 UTF-8 编码的字符串，应如何操作?

答：

```
a= "S".encode("gbk").decode("utf-8",'ignore')
print(a)
```

28. （1）s="info：xiaoZhang 33 shandong"，用正则切分字符串输出['info', 'xiaoZhang', '33', 'shandong']。（2）a = "你好 中国 "，去除多余空格只留一个空格。

答:

（1）我们需要根据冒号或者空格切分

```
import re

s = "info: xiaoZhang 33 shandong"
res = re.split(r": | ", s)
print(res)
```

（2）

```
s = "你好      中国   "
print(" ".join(s.split()))
```

29. (1) 怎样将字符串转换为小写。 (2) 单引号、双引号、三引号的区别?

答： (1) 使用字符串的 lower() 方法。

(2)单独使用单引号和双引号没什么区别，但是如果引号里面还需要使用引号的时候，就需要这两个配合使用了，然后说三引号，同样的三引号也分为三单引号和三双引号，两个都可以声名长的字符串时候使用，如果使用 docstring 就需要使用三双引号。

数据类型 - 列表

30. 已知 AList = [1,2,3,1,2]，对 AList 列表元素去重，写出具体过程。

答:

```
list(set(AList))
```

31. 如何实现 "1,2,3" 变成 ["1","2","3"]

答:

```
s = "1,2,3"
print(s.split(","))
```

32. 给定两个 list，A 和 B，找出相同元素和不同元素

答:

```
A、B 中相同元素: print(set(A)&set(B))
A、B 中不同元素: print(set(A)^set(B))
```

33. [[1,2],[3,4],[5,6]] 一行代码展开该列表，得出 [1,2,3,4,5,6]

答:

```
l = [[1,2],[3,4],[5,6]]
x=[j for i in l for j in i]
print(x)
```

34. 合并列表 [1,5,7,9] 和 [2,2,6,8]

答：使用 extend 和 + 都可以。

```
a = [1,5,7,9]
b = [2,2,6,8]
a.extend(b)
print(a)
```

35. 如何打乱一个列表的元素?

答:

```
import random

a = [1, 2, 3, 4, 5]
random.shuffle(a)
print(a)
```

数据类型 - 字典

36. 字典操作中 del 和 pop 有什么区别

答：del 可以根据索引（元素所在位置）来删除的，没有返回值。 pop 可以根据索引弹出一个值，然后可以接收它的返回值。

37. 按照字典的内的年龄排序

```
d1 = [
    {'name': 'alice', 'age': 38},
    {'name': 'bob', 'age': 18},
    {'name': 'Carl', 'age': 28},
]
```

答:

```
d1 = [
    {'name': 'alice', 'age': 38},
    {'name': 'bob', 'age': 18},
    {'name': 'Carl', 'age': 28},
]

print(sorted(d1, key=lambda x: x["age"]))
```

38. 请合并下面两个字典 a = {"A"： 1,"B"： 2},b = {"C"： 3,"D"： 4}

答： 合并字典方法很多，可以使用 a.update(b) 或者下面字典解包的方式

```
a = {"A": 1,"B": 2}
b = {"C": 3,"D": 4}
print({**a,**b})
```

39. 如何使用生成式的方式生成一个字典，写一段功能代码。

答:

```
# 需求 3: 把字典的 key 和 value 值调换;
d = {'a': '1', 'b': '2'}

print({v: k for k,v in d.items()})
```

40. 如何把元组 ("a","b") 和元组 (1,2)，变为字典 {"a"： 1,"b"： 2}

答 zip 的使用，但是最后记得把 zip 对象再转换为字典。

```
a = ("a", "b")
b = (1, 2)
print(dict(zip(a, b)))
```

数据类型 - 综合

41. 下列字典对象键类型不正确的是?

```
A: {1: 0,2: 0,3: 0}
B: {"a": 0, "b": 0, "c": 0}
C:  {(1,2): 0, (2,3): 0}
D:  {[1,2]: 0, [2,3]: 0}
```

答：D 因为只有可 hash 的对象才能做字典的键，列表是可变类型不是可 hash 对象，所以不能用列表做为字典的键。

42. 如何交换字典 {"A"： 1,"B"： 2}的键和值

答:

```
s =  {"A": 1,"B": 2}

#方法一:
dict_new = {value:key for key, value in s.items()}

# 方法二:
new_s= dict(zip(s.values(), s.keys()))
```

43. Python 里面如何实现 tuple 和 list 的转换?

答： Python 中的类型转换，一般通过类型强转即可完成 tuple 转 list 是 list() 方法 list 转 tuple 使用 tuple() 方法

44. 我们知道对于列表可以使用切片操作进行部分元素的选择，那么如何对生成器类型的对象实现相同的功能呢?

答： 这个题目考察了 Python 标准库的 itertools 模快的掌握情况，该模块提供了操作生成器的一些方法。 对于生成器类型我们使用 islice 方法来实现切片的功能。例子如下

```
from itertools import islice
gen = iter(range(10)) #iter()函数用来生成迭代器
#第一个参数是迭代器，第二个参数起始索引，第三个参数结束索引，不支持负数索引
for i in islice(gen,0,4):
    print(i)
```

45. 请将 [i for i in range(3)] 改成生成器

答：通过把列表生产式的中括号，改为小括号我们就实现了生产器的功能即，

```
(i for i in range(3))
```

46. a="hello" 和 b="你好" 编码成 bytes 类型

答： 这个题目一共三种方式，第一种是在字符串的前面加一个 b，第二种可以使用 bytes 方法，第三种使用字符串 encode 方法。具体代码如下，abc 代表三种情况

```
a = b"hello"
b = bytes("你好", "utf-8")
c = "你好".encode("utf-8")
print(a, b, c)
```

47. 下面的代码输出结果是什么?

```
a = (1,2,3,[4,5,6,7],8)
a[2] = 2
```

答： 我们知道元组里的元素是不能改变的所以这个题目的答案是出现异常。

48. 下面的代码输出的结果是什么?

```
a = (1,2,3,[4,5,6,7],8)
a[3][0] = 2
```

答：前面我说了元组的里元素是不能改变的，这句话严格来说是不准确的，如果元组里面元素本身就是可变类型，比如列表，那么在操作这个元素里的对象时，其内存地址也是不变的。a[3] 对应的元素是列表，然后对列表第一个元素赋值，所以最后的结果是：(1,2,3,[2,5,6,7],8)

操作类题目

49. Python 交换两个变量的值

答：在 Python 中交换两个对象的值通过下面的方式即可

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

但是需要强调的是这并不是元组解包，通过 dis 模块可以发现，这是交换操作的字节码是 ROT_TWO，意思是在栈的顶端做两个值的互换操作。

50. 在读文件操作的时候会使用 read、readline 或者 readlines，简述它们各自的作用

答：.read() 每次读取整个文件，它通常用于将文件内容放到一个字符串变量中。如果希望一行一行的输出那么就可以使用 readline()，该方法会把文件的内容加载到内存，所以对于对于大文件的读取操作来说非常的消耗内存资源，此时就可以通过 readlines 方法，将文件的句柄生成一个生产器，然后去读就可以了。

51. json 序列化时，可以处理的数据类型有哪些？如何定制支持 datetime 类型？

答： 可以处理的数据类型是 str、int、list、tuple、dict、bool、None, 因为 datetime 类不支持 json 序列化，所以我们对它进行拓展。

```
# 自定义时间序列化
import json
from datetime import datetime, date

# JSONEncoder 不知道怎么去把这个数据转换成 json 字符串的时候
# , 它就会去调 default()函数,所以都是重写这个函数来处理它本身不支持的数据类型,
# default()函数默#认是直接抛异常的。
class DateToJson(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, datetime):
            return obj.strftime('%Y-%m-%d %H: %M: %S')
        elif isinstance(obj, date):
            return obj.strftime('%Y-%m-%d')
        else:
            return json.JSONEncoder.default(self, obj)


d = {'name':  'cxa', 'data':  datetime.now()}
print(json.dumps(d, cls=DateToJson))
```

52. 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？

答：可以通过 json.dumps 的 ensure_ascii 参数解决，代码示例如下：

```
import json
a=json.dumps({"name": "张三"},ensure_ascii=False)
print(a)
```

53. 有两个磁盘文件 A 和 B，各存放一行字母，要求把这两个文件中的信息合并(按字母顺序排列)，输出到一个新文件 C 中。

答：

```
#文件 A.txt 内容为 ASDCF
#文件 B.txt 内容为 EFGGTG
with open("A.txt") as f1:
    f1_txt = f1.readline()
with open("B.txt") as f2:
    f2_txt = f2.readline()
f3_txt = f1_txt + f2_txt

f3_list = sorted(f3_txt)

with open("C.txt", "a+") as f:
     f.write("".join(f3_list))
```

输出的文件 C 的内容为 ACDEFFGGGST

54. 如果当前的日期为 20190530，要求写一个函数输出 N 天后的日期，(比如 N 为 2，则输出 20190601)。

答：这个题目考察的是 datetime 里的 timedelta 方法的使用，参数可选、默认值都为 0：datetime.timedelta(days=0, seconds=0, microseconds=0, milliseconds=0, minutes=0, hours=0, weeks=0) 通过这个参数可以指定不同的日期类型进行加减操作，这里我们需要改的是 days，代码如下

```
import datetime


def datetime_operate(n:  int):
    now = datetime.datetime.now()  # 获取当前时间
    _new_date = now + datetime.timedelta(days=n)  # 获取指定天数后的新日期
    new_date = _new_date.strftime("%Y%m%d")  # 转换为指定的输出格式
    return new_date


if __name__ == '__main__':
    print(datetime_operate(4))
```

55. 写一个函数，接收整数参数 n，返回一个函数，函数的功能是把函数的参数和 n 相乘并把结果返回。

答：这个题目考查了闭包的使用代码示例如下，返回函数之类型是函数对象。

```
def mul_operate(num):
    def g(val):
        return num * val

    return g


m = mul_operate(8)
print(m(5))
```

56. 下面代码会存在什么问题，如何改进？

```
def strappend(num):
    str='first'
    for i in range(num):
        str+=str(i)
    return str
```

答： 首先不应该使用 Python 的内置类似 str 作为变量名这里我把它改为了 s,另外在Python,str 是个不可变对象，每次迭代都会生成新的存储空间，num 越大，创建的 str 对象就会越多，内存消耗越大。使用 yield 改成生成器即可, 还有一点就是命名规范的位置，函数名改为_分割比较好，完整的代码如下:

```
def str_append(num):
    s = 'first'
    for i in range(num):
        s += str(i)
        yield s

if __name__ == '__main__':
    for i in str_append(3):
        print(i)
```

57. 一行代码输出 1-100 之间的所有偶数。

答：可以通过列表生成式，然后使用与操作如果如 1 与之后结果为 0 则表明为偶数，等于 1 则为奇数。

```
# 方法1
print([i for i in range(1, 101) if i & 0x1 == 0])
# 方法2：测试发现方法二效率更高
print(list(range(2, 101, 2)))
```

58. with 语句的作用，写一段代码？

with 语句适用于对资源进行访问的场合，确保不管使用过程中是否发生异常都会执行必要的“清理”操作，释放资源，比如文件使用后自动关闭、线程中锁的自动获取和释放等。

其他的内容看下面我之前写的代码。

```
#一般访问文件资源时我们会这样处理：
```

```
f = open(
    'c: \test.txt', 'r')
data = f.read()
f.close()
# 这样写没有错，但是容易犯两个毛病：
# 1. 如果在读写时出现异常而忘了异常处理。
# 2. 忘了关闭文件句柄

#以下的加强版本的写法：

f = open('c: \test.txt', 'r')
try:
    data = f.read()
finally:
    f.close()

#以上的写法就可以避免因读取文件时异常的发生而没有关闭问题的处理了。代码长了一些。
#但使用 with 有更优雅的写法：

with open(r'c: \test.txt', 'r') as f:
    data = f.read()
#with 的实现

class Test:
    def __enter__(self):
        print('__enter__() is call!')
        return self

    def dosomething(self):
        print('dosomethong!')

    def __exit__(self, exc_type, exc_value, traceback):
        print('__exit__() is call!')
        print(f'type: {exc_type}')
        print(f'value: {exc_value}')
        print(f'trace: {traceback}')
        print('__exit()__ is call!')

with Test() as sample:
      pass

#当对象被实例化时，就会主动调用__enter__()方法，任务执行完成后就会调用__exit__()方法，
#另外，注意到，__exit__()方法是带有三个参数的(exc_type, exc_value, traceback)，
#依据上面的官方说明：如果上下文运行时没有异常发生，那么三个参数都将置为 None，
#这里三个参数由于没有发生异常，的确是置为了 None，与预期一致.

# 修改后不出异常了
class Test:
    def __enter__(self):
        print('__enter__() is call!')
        return self

    def dosomething(self):
        x = 1/0
```

```
        print('dosomethong!')

    def __exit__(self, exc_type, exc_value, traceback):
        print('__exit__() is call!')
        print(f'type: {exc_type}')
        print(f'value: {exc_value}')
        print(f'trace: {traceback}')
        print('__exit()__ is call!')
        return True


with Test() as sample:
```

59. Python 字典和 json 字符串相互转化方法

答:

在 Python 中使用 dumps 方法 将 dict 对象转为 Json 对象，使用 loads 方法可以将 Json 对象转为 dict 对象。

```
dic = {'a': 123, 'b': "456", 'c': "liming"}
json_str = json.dumps(dic)
dic2 = json.loads(json_str)
print(dic2)
打印:
'{"a": 123, "b": "456", "c": "liming"}'
{'a': 123, 'b': '456', 'c': 'liming'}
```

我们再来看一个特殊的例子

```
import json
dic = {'a':  123, 'b':  "456", 'c':  "liming"}
dic_str = json.loads(str(dic).replace("'", "\""))
print(dic_str)
```

下面我解释下上面代码是测试什么:

```
首先 json.loads(jsonstr) 这里面的参数只能是 jsonstr 格式的字符串.
当我们使用 str 将字典 dic 转化为字符串以后，得到的结果为:"{'a': 123, 'b': '456', 'c':
'liming'}"。
如果直接使用 json.loads(str(dic)) 你会发现出现错误，原因就是，单引号的字符串不符合Json的
标准格式所以再次使用了 replace("'", "\"")。得到字典
其实这个例子主要目的是告诉大家 Json 的标准格式是不支持单引号型字符串的，否则会出现以下错误。
json.decoder.JSONDecodeError: Expecting property name enclosed in double quotes:
line 1 column 2 (char 1)
```

60. 请写一个 Python 逻辑，计算一个文件中的大写字母数量

答:

```
with open('A.txt') as fs:
    count = 0
    for i in fs.read():
        if i.isupper():
            count += 1
print(count)
```

61. 请写一段 Python连接Mongo数据库，然后的查询代码。

答:

```
# -*- coding: utf-8 -*-
# @Author : 陈祥安
import pymongo
db_configs = {
    'type': 'mongo',
    'host': '地址',
    'port': '端口',
    'user': 'spider_data',
    'passwd': '密码',
    'db_name': 'spider_data'
}


class Mongo():
    def __init__(self, db=db_configs["db_name"], username=db_configs["user"],
                 password=db_configs["passwd"]):
        self.client =
pymongo.MongoClient(f'mongodb://{db_configs["host"]}:db_configs["port"]')
        self.username = username
        self.password = password
        if self.username and self.password:
            self.db1 = self.client[db].authenticate(self.username, self.password)
        self.db1 = self.client[db]

    def find_data(self):
        # 获取状态为0的数据
        data = self.db1.test.find({"status": 0})
        gen = (item for item in data)
        return gen

if __name__ == '__main__':
    m = Mongo()
    print(m.find_data())
```

62.说一说Redis的基本类型

答: Redis 支持五种数据类型： string（字符串） 、 hash（哈希）、list（列表） 、 set（集合） 及 zset(sorted set： 有序集合)。

63. 请写一段 Python连接Redis数据库的代码。

答:

```
from redis import StrictRedis, ConnectionPool
redis_url="redis://:xxxx@112.27.10.168:6379/15"
pool = ConnectionPool.from_url(redis_url, decode_responses=True)
r= StrictRedis(connection_pool=pool)
```

64. 请写一段 Python连接Mysql数据库的代码。

答:

```
conn = pymysql.connect(host='localhost',
port=3306, user='root',
passwd='1234', db='user', charset='utf8mb4')#声明mysql连接对象
cursor=conn.cursor(cursor=pymysql.cursors.DictCursor)#查询结果以字典的形式
cursor.execute(sql语句字符串)#执行sql语句
conn.close()#关闭链接
```

65.了解Redis的事务么

答: 简单理解，可以认为 redis 事务是一些列 redis 命令的集合，并且有如下两个特点： 1.事务是一个单独的隔离操作：事务中的所有命令都会序列化、按顺序地执行。事务在执行的过程中，不会被其他客户端发送来的命令请求所打断。 2.事务是一个原子操作：事务中的命令要么全部被执行，要么全部都不执行。 一般来说，事务有四个性质称为ACID，分别是原子性，一致性，隔离性和持久性。 一个事务从开始到执行会经历以下三个阶段：

- 开始事务
- 命令入队
- 执行事务 代码示例:

```
import redis
import sys
def run():
    try:
        conn=redis.StrictRedis('192.168.80.41')
       # Python中redis事务是通过pipeline的封装实现的
        pipe=conn.pipeline()
        pipe.sadd('s001','a')
        sys.exit()
        #在事务还没有提交前退出，所以事务不会被执行。
        pipe.sadd('s001','b')
        pipe.execute()
        pass
    except Exception as err:
        print(err)
        pass
if __name__=="__main__":
      run()
```

66.了解数据库的三范式么?

答: 经过研究和对使用中问题的总结，对于设计数据库提出了一些规范，这些规范被称为范式 一般需要遵守下面3范式即可: 第一范式（1NF）：强调的是列的原子性，即列不能够再分成其他几列。 第二范式（2NF）：首先是 1NF，另外包含两部分内容，一是表必须有一个主键；二是没有包含在主键中的列必须完全依赖于主键，而不能只依赖于主键的一部分。 第三范式（3NF）：首先是 2NF，另外非主键列必须直接依赖于主键，不能存在传递依赖。即不能存在：非主键列 A 依赖于非主键列 B，非主键列 B 依赖于主键的情况。

67.了解分布式锁么

答: 分布式锁是控制分布式系统之间的同步访问共享资源的一种方式。 对于分布式锁的目标，我们必须首先明确三点：

- 任何一个时间点必须只能够有一个客户端拥有锁。
- 不能够有死锁，也就是最终客户端都能够获得锁，尽管可能会经历失败。
- 错误容忍性要好，只要有大部分的Redis实例存活，客户端就应该能够获得锁。 分布式锁的条件 互斥性：分布式锁需要保证在不同节点的不同线程的互斥 可重入性：同一个节点上的同一个线程如果获取了锁之后，能够再次获取这个锁。 锁超时：支持超时释放锁，防止死锁 高效，高可用：加锁和解锁需要高效，同时也需要保证高可用防止分布式锁失效，可以增加降级。 支持阻塞和非阻塞：可以实现超时获取失败，tryLock(long timeOut) 支持公平锁和非公平锁

分布式锁的实现方案 1、数据库实现（乐观锁） 2、基于zookeeper的实现 3、基于Redis的实现（推荐）

68.用 Python 实现一个 Reids 的分布式锁的功能

答:REDIS分布式锁实现的方式：SETNX + GETSET,NX是Not eXists的缩写，如SETNX命令就应该理解为：SET if Not eXists。 多个进程执行以下Redis命令：

```
SETNX lock.foo <current Unix time + lock timeout + 1>
```

如果 SETNX 返回1，说明该进程获得锁，SETNX将键 lock.foo 的值设置为锁的超时时间（当前时间 + 锁的有效时间）。 如果 SETNX 返回0，说明其他进程已经获得了锁，进程不能进入临界区。进程可以在一个循环中不断地尝试 SETNX 操作，以获得锁。

```
import time
import redis
from conf.config import REDIS_HOST, REDIS_PORT, REDIS_PASSWORD

class RedisLock:
    def __init__(self):
        self.conn = redis.Redis(host=REDIS_HOST, port=REDIS_PORT,
password=REDIS_PASSWORD, db=1)
        self._lock = 0
        self.lock_key = ""
    @staticmethod
    def my_float(timestamp):
        """
        Args:
            timestamp:
        Returns:
            float或者0
            如果取出的是None，说明原本锁并没人用，getset已经写入，返回0，可以继续操作。
        """
```

```
        if timestamp:
            return float(timestamp)
        else:
            #防止取出的值为None，转换float报错
            return 0

    @staticmethod
    def get_lock(cls, key, timeout=10):
        cls.lock_key = f"{key}_dynamic_lock"
        while cls._lock != 1:
            timestamp = time.time() + timeout + 1
            cls._lock = cls.conn.setnx(cls.lock_key, timestamp)
            # if 条件中，可能在运行到or之后被释放，也可能在and之后被释放
            # 将导致 get到一个None，float失败。
            if cls._lock == 1 or (
                            time.time() > cls.my_float(cls.conn.get(cls.lock_key))
and
                            time.time() >
cls.my_float(cls.conn.getset(cls.lock_key, timestamp))):
                break
            else:
                time.sleep(0.3)

    @staticmethod
    def release(cls):
        if cls.conn.get(cls.lock_key) and time.time() <
cls.conn.get(cls.lock_key):
            cls.conn.delete(cls.lock_key)


def redis_lock_deco(cls):
    def _deco(func):
        def __deco(*args, **kwargs):
            cls.get_lock(cls, args[1])
            try:
                return func(*args, **kwargs)
            finally:
                cls.release(cls)
        return __deco
    return _deco


@redis_lock_deco(RedisLock())
def my_func():
    print("myfunc() called.")
    time.sleep(20)

if __name__ == "__main__":
    my_func()
```

69.写一段 Python 使用 mongo 数据库创建索引的代码:

答:

```python
# -*- coding: utf-8 -*-
# @Time : 2018/12/28 10:01 AM
# @Author : cxa
import pymongo
db_configs = {
    'type': 'mongo',
    'host': '地址',
    'port': '端口',
    'user': 'spider_data',
    'passwd': '密码',
    'db_name': 'spider_data'
}


class Mongo():
    def __init__(self, db=db_configs["db_name"], username=db_configs["user"],
                 password=db_configs["passwd"]):
        self.client = pymongo.MongoClient(f'mongodb://{db_configs["host"]}:
{db_configs["port"]}')
        self.username = username
        self.password = password
        if self.username and self.password:
            self.db1 = self.client[db].authenticate(self.username, self.password)
        self.db1 = self.client[db]

    def add_index(self):
        """
          通过create_index添加索引
        """
        self.db1.test.create_index([('name', pymongo.ASCENDING)], unique=True)

    def get_index(self,):
        """
          查看索引列表
        """
        indexlist=self.db1.test.list_indexes()
        for index in indexlist:
            print(index)

if __name__ == '__main__':
    m = Mongo()
    m.add_index()
    print(m.get_index())
```

高级特性

70. 函数装饰器有什么作用？请列举说明？

答： 装饰器就是一个函数，它可以在不需要做任何代码变动的前提下给一个函数增加额外功能，启动装饰的效果。 它经常用于有切面需求的场景，比如：插入日志、性能测试、事务处理、缓存、权限校验等场景。 下面是一个日志功能的装饰器

```python
from functools import wraps
```

```
def log(label):
    def decorate(func):
       @wraps(func)
       def _wrap(*args,**kwargs):
        try:
          func(*args,**kwargs)
          print("name",func.__name__)
        except Exception as e:
           print(e.args)
       return _wrap
    return decorate

@log("info")
def foo(a,b,c):
     print(a+b+c)
     print("in foo")

#decorate=decorate(foo)

if __name__ == '__main__':
    foo(1,2,3)
     #decorate()
```

71. Python 垃圾回收机制?

答：Python 不像 C++，Java 等语言一样，他们可以不用事先声明变量类型而直接对变量进行赋值。对 Python 语言来讲，对象的类型和内存都是在运行时确定的。这也是为什么我们称 Python 语言为动态类型的原因。

主要体现在下面三个方法：

1.引用计数机制 2.标记-清除 3.分代回收

72. 魔法函数 *call*怎么使用?

答： *call* 可以把类实例当做函数调用。 使用示例如下

```
class Bar:
    def __call__(self, *args, **kwargs):
        print('in call')


if __name__ == '__main__':
    b = Bar()
    b()
```

73. 如何判断一个对象是函数还是方法?

答：看代码已经结果就懂了

```
from types import MethodType, FunctionType
```

```
class Bar:
    def foo(self):
        pass


def foo2():
    pass


def run():
    print("foo 是函数", isinstance(Bar().foo, FunctionType))
    print("foo 是方法", isinstance(Bar().foo, MethodType))
    print("foo2 是函数", isinstance(foo2, FunctionType))
    print("foo2 是方法", isinstance(foo2, MethodType))


if __name__ == '__main__':
    run()
```

输出

```
foo 是函数 False
foo 是方法 True
foo2 是函数 True
foo2 是方法 False
```

74. @classmethod 和 @staticmethod 用法和区别

答： 相同之处：@staticmethod 和@classmethod 都可以直接类名.方法名()来调用，不用在示例化一个类。 @classmethod 我们要写一个只在类中运行而不在实例中运行的方法。如果我们想让方法不在实例中运行，可以这么做：

```
def iget_no_of_instance(ins_obj):
    return ins_obj.__class__.no_inst


class Kls(object):
    no_inst = 0

    def __init__(self):
        Kls.no_inst = Kls.no_inst + 1


ik1 = Kls()
ik2 = Kls()
print(iget_no_of_instance(ik1))
```

@staticmethod 经常有一些跟类有关系的功能但在运行时又不需要实例和类参与的情况下需要用到静态方法

```
IND = 'ON'
```

```
class Kls(object):
    def __init__(self, data):
        self.data = data

    @staticmethod
    def check_ind():
        return (IND == 'ON')

    def do_reset(self):
        if self.check_ind():
            print('Reset done for: ', self.data)

    def set_db(self):
        if self.check_ind():
            self.db = 'New db connection'
        print('DB connection made for:  ', self.data)


ik1 = Kls(12)
ik1.do_reset()
ik1.set_db()
```

75. Python 中的接口如何实现?

答： 接口提取了一群类共同的函数，可以把接口当做一个函数的集合，然后让子类去实现接口中的函数。但是在 Python 中根本就没有一个叫做 interface 的关键字，如果非要去模仿接口的概念，可以使用抽象类来实现。抽象类是一个特殊的类，它的特殊之处在于只能被继承，不能被实例化。使用 abc 模块来实现抽象类。

76. Python 中的反射了解么?

答：Python 的反射机制设定较为简单，一共有四个关键函数分别是 getattr、hasattr、setattr、delattr。

77. metaclass 作用？以及应用场景?

答： metaclass 即元类，metaclass 是类似创建类的模板，所有的类都是通过他来 create 的(调用**new**)，这使得你可以自由的控制创建类的那个过程，实现你所需要的功能。 我们可以使用元类创建单例模式和实现 ORM 模式。

78. hasattr()、getattr()、setattr() 的用法

答：这三个方法属于 Python 的反射机制里面的，hasattr 可以判断一个对象是否含有某个属性，getattr 可以充当 get 获取对象属性的作用。而 setattr 可以充当 person.name = "liming"的赋值操作。代码示例如下：

```
class Person():
    def __init__(self):
        self.name = "liming"
        self.age = 12

    def show(self):
```

```
        print(self.name)
        print(self.age)

    def set_name(self):
        setattr(Person, "sex", "男")

    def get_name(self):
        print(getattr(self, "name"))
        print(getattr(self, "age"))
        print(getattr(self, "sex"))


def run():
    if hasattr(Person, "show"):
        print("判断 Person 类是否含有 show 方法")


    Person().set_name()
    Person().get_name()


if __name__ == '__main__':
    run()
```

79. 请列举你知道的 Python 的魔法方法及用途。

答：

```
1 __init__:
类的初始化方法。它获取任何传给构造器的参数（比如我们调用 x = SomeClass(10, ‘foo’)，
__init__就会接到参数 10 和 ‘foo’ 。 __init__在 Python 的类定义中用的最多。

2 __new__:
__new__是对象实例化时第一个调用的方法，它只取下 cls 参数，并把其他参数传给 __init__ 。
__new__很少使用，但是也有它适合的场景，尤其是当类继承自一个像元组或者字符串这样不经常改变的
类型的时候.

3 __del__:
__new__和 __init__是对象的构造器， __del__是对象的销毁器。它并非实现了语句 del x (因此该
语句不等同于 x.__del__())。而是定义了当对象被垃圾回收时的行为。 当对象需要在销毁时做一些处
理的时候这个方法很有用，比如 socket 对象、文件对象。但是需要注意的是，当 Python 解释器退出
但对象仍然存活的时候，__del__并不会 执行。 所以养成一个手工清理的好习惯是很重要的，比如及时
关闭连接。
```

80. 如何知道一个 Python 对象的类型?

答：可以通过 type 方法

81. Python 的传参是传值还是传址?

答：Python 中的传参即不是传值也不是传地址，传的是对象的引用。

82. Python 中的元类 (metaclass) 使用举例

答：可以使用元类实现一个单例模式，代码如下

```
class Singleton(type):
    def __init__(self, *args, **kwargs):
        print("in __init__")
        self.__instance = None
        super(Singleton, self).__init__(*args, **kwargs)

    def __call__(self, *args, **kwargs):
        print("in __call__")
        if self.__instance is None:
            self.__instance = super(Singleton, self).__call__(*args, **kwargs)
        return self.__instance

class Foo(metaclass=Singleton):
    pass  # 在代码执行到这里的时候，元类中的__new__方法和__init__方法其实已经被执行了，而不是在 Foo 实例化的时候执行。且仅会执行一次。

foo1 = Foo()
foo2 = Foo()
print(foo1 is foo2)
```

83. 简述 any() 和 all() 方法

答： any(x)：判断 x 对象是否为空对象，如果都为空、0、false，则返回 false，如果不都为空、0、false，则返回 true。 all(x)：如果 all(x) 参数 x 对象的所有元素不为 0、''、False 或者 x 为空对象，则返回 True，否则返回 False。

84. filter 方法求出列表所有奇数并构造新列表，a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

答

```
a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(list(filter(lambda x:  x % 2 == 1, a)))
```

其实现在不推荐使用 filter,map 等方法了，一般列表生成式就可以搞定了。

85. 什么是猴子补丁?

答： 猴子补丁（monkey patching）：在运行时动态修改模块、类或函数，通常是添加功能或修正缺陷。猴子补丁在代码运行时内存中）发挥作用，不会修改源码，因此只对当前运行的程序实例有效。因为猴子补丁破坏了封装，而且容易导致程序与补丁代码的实现细节紧密耦合，所以被视为临时的变通方案，不是集成代码的推荐方式。大概是下面这样的一个效果

```
def post():
    print("this is post")
    print("想不到吧")

class Http():
    @classmethod
    def get(self):
```

```
        print("this is get")


def main():
    Http.get=post #动态的修改了 get 原因的功能,

if __name__ == '__main__':
    main()
    Http.get()
```

86. 在 Python 中是如何管理内存的?

答： 垃圾回收：Python 不像 C++，Java 等语言一样，他们可以不用事先声明变量类型而直接对变量进行赋值。对 Python 语言来讲，对象的类型和内存都是在运行时确定的。这也是为什么我们称 Python 语言为动态类型的原因（这里我们把动态类型可以简单的归结为对变量内存地址的分配是在运行时自动判断变量类型并对变量进行赋值）。

引用计数：Python 采用了类似 Windows 内核对象一样的方式来对内存进行管理。每一个对象，都维护这一个对指向该对对象的引用的计数。当变量被绑定在一个对象上的时候，该变量的引用计数就是 1，(还有另外一些情况也会导致变量引用计数的增加)，系统会自动维护这些标签，并定时扫描，当某标签的引用计数变为 0 的时候，该对就会被回收。

- 内存池机制 Python 的内存机制以金字塔行，1、2 层主要有操作系统进行操作
- 第 0 层是 C 中的 malloc，free 等内存分配和释放函数进行操作
- 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
- 第 3 层是最上层，也就是我们对 Python 对象的直接操作
- 在 C 中如果频繁的调用 malloc 与 free 时,是会产生性能问题的.再加上频繁的分配与释放小块的内存会产生内存碎片。Python 在这里主要干的工作有：
  - 如果请求分配的内存在 1~256 字节之间就使用自己的内存管理系统,否则直接使用 malloc。
  - 这里还是会调用 malloc 分配内存，但每次会分配一块大小为 256k 的大块内存。
- 经由内存池登记的内存到最后还是会回收到内存池，并不会调用 C 的 free 释放掉以便下次使用。对于简单的 Python 对象，例如数值、字符串，元组（tuple 不允许被更改)采用的是复制的方式(深拷贝?)，也就是说当将另一个变量 B 赋值给变量 A 时，虽然 A 和 B 的内存空间仍然相同，但当 A 的值发生变化时，会重新给 A 分配空间，A 和 B 的地址变得不再相同。

87. 当退出 Python 时是否释放所有内存分配?

答：不是的，循环引用其他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

正则表达式

88. （1）使用正则表达式匹配出 `www.baidu.com` 中的地址（2）a="张明 98 分"，用 re.sub，将 98 替换为 100

答： 第一问答案

```
import re

source = "<html><h1>www.baidu.com</h1></html>"
pat = re.compile("<html><h1>(.*?)</h1></html>")
print(pat.findall(source)[0])
```

第二问答案

```
import re
s = "张明 98 分"
print(re.sub(r"\d+","100",s))
```

89. 正则表达式匹配中(.)和(.?)匹配区别?

答：(.) 为贪婪模式极可能多的匹配内容，(.?) 为非贪婪模式又叫懒惰模式，一般匹配到结果就好，匹配字符的少为主，示例代码如下

```
import re

s = "<html><div>文本 1</div><div>文本 2</div></html>"

pat1 = re.compile(r"\<div>(.*?)\</div>")
print(pat1.findall(s))

pat2 = re.compile(r"\<div>(.*)\</div>")
print(pat2.findall(s))
```

输出

```
['文本 1', '文本 2']
['文本 1</div><div>文本 2']
```

90. 写一段匹配邮箱的正则表达式

答：关于邮箱的匹配这个还真的是一个永恒的话题。

电子邮件地址有统一的标准格式：用户名@服务器域名。用户名表示邮件信箱、注册名或信件接收者的用户标识，@符号后是你使用的邮件服务器的域名。@可以读成“at”，也就是“在”的意思。整个电子邮件地址可理解为网络中某台服务器上的某个用户的地址。

1. 用户名，可以自己选择。由字母 a～z(不区分大小写)、数字 0～9、点、减号或下划线组成；只能以数字或字母开头和结尾。
2. 与你使用的网站有关，代表邮箱服务商。例如网易的有@163.com 新浪有@vip.sina.com 等。

网上看到了各种各样的版本，都不确定用哪个，于是自己简单的总结了一个。大家有更好的欢迎留言。

```
r"^[a-zA-Z0-9]+[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$"
```

下面解释上面的表达式

1. 首先强调一点关于\w 的含义，\w 匹配英文字母和俄语字母或数字或下划线或汉字。

2. 注意^[]和[^]的区别，[]表示字符集合，^[]表示已[]内的任意字符集开始，[^]表示。
3. ^[a-zA-Z0-9]+：这里注意^[]和[^]的,第一个^表示已什么开头，第二个[]的^表示不等于[]内。所以这段表示以英文字母和数字开头，后面紧跟的+，限定其个数>=1 个。
4. [a-zA-Z0-9.*+-]+：表示匹配英文字母和数字开头以及.+-, 的任意一个字符，并限定其个数>=1 个。为了考虑@前面可能出现.+-（但是不在开头出现）。
5. @就是邮箱必备符号了
6. @[a-zA-Z0-9-]+.：前面的不用说了，后面的.表示.转义了,也是必备符号。
7. [a-zA-Z0-9-.]+：$符表示以什么结束,这里表示以英文字和数字或 -. 1 个或多个结尾。

来个例子验证一波：

```
import re
plt=re.compile(r"^[a-zA-Z0-9]+[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$")
b=plt.findall('adas+fefe.we@qq.com.cn')
print(b)
```

网上找了个验证邮件地址的通用正则表达式（符合 RFC 5322 标准）

```
(?: [a-z0-9!#$%&'*+/=?^_`{|}~-]+(?: \.[a-z0-9!#$%&'*+/=?^_`{|}~-]+)*|"(?: [\x01-
\x08\x0b\x0c\x0e-\x1f\x21\x23-\x5b\x5d-\x7f]|\\[\x01-\x09\x0b\x0c\x0e-
\x7f])*")@(?: (?: [a-z0-9](?: [a-z0-9-]*[a-z0-9])?\.)+[a-z0-9](?: [a-z0-9-]*[a-z0-
9])?|\[(?: (?: 25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?: 25[0-5]|2[0-4][0-9]|
[01]?[0-9][0-9]?|[a-z0-9-]*[a-z0-9]: (?: [\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x5a\x53-
\x7f]|\\[\x01-\x09\x0b\x0c\x0e-\x7f])+)\])
```

其他内容

91. 解释一下 Python 中 pass 语句的作用?

答：pass 实际上就是一个占位符，在写一个函数但是不确定里面写啥的时候，这个时候可以使用 pass。示例如下

```
def foo():
 pass
```

92. 简述你对 input()函数的理解

答：在 Python3 中 input 函数可以接收用户输入的字符串。

然后根据程序的需要转换成所需格式即可。

93. Python 中的 is 和==

答：先说==它的作用是判断两个对象的值是否相同，然后说 is。is 表示的谁是谁，这也就意味着对象完全相等。我们知道一个对象有各自的内存地址和对应的值，当内存地址和值都相同的时候使用 is 可以得到结果 True。另外需要注意的下面两点特殊的情况。

这些变量很可能在许多程序中使用。 通过池化这些对象，Python 可以防止对一致使用的对象进行内存分配调用。

```
1.介于数字-5 和 256 之间的整数
2.字符串仅包含字母、数字或下划线
```

94. Python 中的作用域

答：

```
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)
```

95. 三元运算写法和应用场景?

答：Python 中的三元运算又称三目运算，是对简单的条件语句的简写。 是一种比较 Pythonic 的学法，形式为：val = 1 if 条件成立 else 2 代码示例如下：

```python
a = 2
b = 5

# 普通写法
if a > b:
    val = True
else:
    val = False
# 改为三元运算符后
val = a if a > b else b
print(val)  # 5
```

96. 了解 enumerate 么?

答：enumerate 可以在迭代一个对象的时候，同时获取当前对象的索引和值。 代码示例如下

```python
from string import ascii_lowercase

s = ascii_lowercase

for index, value in enumerate(s):
    print(index, value)
```

97. 列举 5 个 Python 中的标准模块

答： pathlib：路径操作模块，比 os 模块拼接方便。 urllib：网络请求模块，包括对 url 的结构解析。 asyncio： Python 的异步库，基于事件循环的协程模块。 re：正则表达式模块。 itertools：提供了操作生成器的一些模块。

98. 如何在函数中设置一个全局变量

答：

```
# 通过使用 global 对全局变量进行修改。
n = 0
def foo():
    global n
    n = 100
foo()
print(n)
x = 0
```

之前我在视频教程中对这块做了个讲解，具体点击下方链接 https：//www.bilibili.com/video/av50865713

99. pathlib 的用法举例

答：pathlib 可以对文件以及文件的其他属性进行操作。比较喜欢的一点是路径拼接符"/"的使用，之前在公众号中写过 pathlib 一些其他的用法这里就不一一例举了。

100. Python 中的异常处理，写一个简单的应用场景

答： 比如在计算除法中出现为 0 的情况出现异常

```
try:
    1 / 0
except ZeroDivisionError as e:
    print(e.args)
```

101. Python 中递归的最大次数，那如何突破呢？

答：Python 有递归次数限制，默认最大次数为 1000。通过下面的代码可以突破这个限制

```
import sys
sys.setrecursionlimit(1500) # set the maximum depth as 1500
```

另外需要注意的是 sys.setrecursionlimit() 只是修改解释器在解释时允许的最大递归次数，此外，限制最大递归次数的还和操作系统有关。

102. 什么是面向对象的 mro

答：Python 是支持面向对象编程的，同时也是支持多重继承的。一般我们通过调用类对象的 mro()方法获取其继承关系。

103. isinstance 作用以及应用场景？

答：isinstance 是判断一个对象是否为另一个对象的子类的，例如我们知道在 Python3 中 bool 类型其实是 int 的子类，所以我们可以对其检测。

```
print(isinstance(True,int))
```

104. 什么是断言？应用场景？

答：在 Python 中是断言语句 assert 实现此功能，一般在表达式为 True 的情况下，程序才能通过。

```python
#author: 陈祥安
#公众号: Python 学习开发

#assert () 方法，断言成功，则程序继续执行，断言失败，则程序报错
# 断言能够帮助别人或未来的你理解代码，
# 找出程序中逻辑不对的地方。一方面，
# 断言会提醒你某个对象应该处于何种状态，
# 另一方面，如果某个时候断言为假，
# 会抛出 AssertionError 异常，很有可能终止程序。

def foo(a):
    assert a==2,Exception("不等于 2")
    print("ok",a)

if __name__ == '__main__':
    foo(1)
```

105. lambda 表达式格式以及应用场景?

答：lambda 表达式其实就是一个匿名函数,在函数编程中经常作为参数使用。 例子如下

```python
a = [('a',1),('b',2),('c',3),('d',4)]
a_1 = list(map(lambda x: x[0],a))
```

106. 新式类和旧式类的区别

答：Python 2.x 中默认都是经典类，只有显式继承了 object 才是新式类，Python 3.x 中默认都是新式类，经典类被移除，不必显式的继承 object。 新式类都从 object 继承，经典类不需要。 新式类的 MRO(method resolution order 基类搜索顺序)算法采用 C3 算法广度优先搜索，而旧式类的 MRO 算法是采用深度优先搜索。 新式类相同父类只执行一次构造函数，经典类重复执行多次。

107. dir()是干什么用的?

答：当在使用某一个对象不知道有哪些属性或者方法可以使用时，此时可以通过 dir() 方法进行查看。

108. 一个包里有三个模块，demo1.py、demo2.py、demo3.py，但使用 from tools import *导入模块时，如何保证只有 demo1、demo3 被导入了。

答： 增加*init*.py 文件，并在文件中增加：

```python
__all__ = ['demo1','demo3']
```

109. 列举 5 个 Python 中的异常类型以及其含义

答：

```
AttributeError 对象没有这个属性

NotImplementedError 尚未实现的方法

StopIteration 迭代器没有更多的值

TypeError 对类型无效的操作

IndentationError 缩进错误
```

110. copy 和 deepcopy 的区别是什么?

答： copy.copy()浅拷贝，只拷贝父对象，不会拷贝对象的内部的子对象。 copy.deepcopy()深拷贝，拷贝对象及其子对象。

111. 代码中经常遇到的*args, **kwargs 含义及用法。

答： 在函数定义中使用 *args 和**kwargs 传递可变长参数。 *args 用来将参数打包成 tuple 给函数体调用。** kwargs 打包关键字参数成 dict 给函数体调用。

112. Python 中会有函数或成员变量包含单下划线前缀和结尾，和双下划线前缀结尾，区别是什么?

答： "单下划线" 开始的成员变量叫做保护变量，意思是只有类对象和子类对象自己能访问到这些变量； "双下划线" 开始的是私有成员，意思是只有类对象自己能访问，连子类对象也不能访问到这个数据。

以单下划线开头（_foo）的代表不能直接访问的类属性，需通过类提供的接口进行访问，不能用“from xxx import *”而导入；以双下划线开头的（__foo）代表类的私有成员；

以双下划线开头和结尾的（_*foo）**代表 Python 里特殊方法专用的标识，如 _*init**（）代表类的构造函数。

代码示例

```python
class Person:
    """docstring for ClassName"""
    def __init__(self):
       self.__age = 12
       self._sex = 12
    def _sex(self):
        return "男"
    def set_age(self,age):
        self.__age = age

    def get_age(self):
        return self.__age

if __name__ == '__main__':
    p=Person()
    print(p._sex)
    #print(p.__age)
    #Python 自动将__age 解释成 _Person__age,于是我们用 _Person__age 访问，这次成功。
```

```
    print(p._Person__age)
```

113. w、a+、wb 文件写入模式的区别

答： w 表示写模式支持写入字符串，如果文件存在则覆盖。 a+ 和 w 的功能类型不过如果文件存在的话内容不会覆盖而是追加。 wb 是写入二进制字节类型的数据。

114. 举例 sort 和 sorted 的区别

答： 相同之处 sort 和 sorted 都可以对列表元素排序，sort() 与 sorted() 的不同在于，sort 是在原位重新排列列表，而 sorted() 是产生一个新的列表。 sort 是应用在 list 上的方法，sorted 可以对所有可迭代的对象进行排序操作。

list 的 sort 方法返回的是对已经存在的列表进行操作，而内建函数 sorted 方法返回的是一个新的 list，而不是在原来的基础上进行的操作。

115. 什么是负索引?

答：负索引一般表示的是从后面取元素。

116. pprint 模块是干什么的?

答：pprint 是 print 函数的美化版，可以通过 import pprint 导入。示例如下

```
import pprint
pprint.pprint("this is pprint")
```

117. 解释一下 Python 中的赋值运算符

答：通过下面的代码列举出所有的赋值运算符

```
a=7
a+=1
print(a)
a-=1
print(a)
a*=2
print(a)
a/=2
print(a)
a**=2
print(a)
a//=3
print(a)
a%=4
print(a)
```

118. 解释一下 Python 中的逻辑运算符

答：Python 中有三个逻辑运算符：and、or、not

```
print(False and True) #False
print(7<7 or True) #True
print(not 2==2) #False
```

119. 讲讲 Python 中的位运算符

答：按位运算符是把数字看作二进制来进行计算的。Python 中的按位运算法则如下：

下表中变量 a 为 60，b 为 13，二进制格式如下：

```
a = 0011 1100
b = 0000 1101
-----------------
a&b = 0000 1100
a|b = 0011 1101
a^b = 0011 0001
~a  = 1100 0011
```

运算符	描述	实例
&	按位与运算符：参与运算的两个值,如果两个相应位都为1,则该位的结果为1,否则为0	(a & b) 输出结果 12 ，二进制解释： 0000 1100
|	按位或运算符：只要对应的二个二进位有一个为1时，结果位就为1。	(a | b) 输出结果 61 ，二进制解释： 0011 1101
^	按位异或运算符：当两对应的二进位相异时，结果为1	(a ^ b) 输出结果 49 ，二进制解释： 0011 0001
~	按位取反运算符：对数据的每个二进制位取反,即把1变为0,把0变为1	(~a) 输出结果 -61 ，二进制解释： 1100 0011
<<	左移动运算符：运算数的各二进位全部左移若干位，由"<<"右边的数指定移动的位数，高位丢弃，低位补0。	a << 2 输出结果 240 ，二进制解释： 1111 0000
>>	右移动运算符：把">>"左边的运算数的各二进位全部右移若干位，">>"右边的数指定移动的位数	a >> 2 输出结果 15 ，二进制解释： 0000 1111

120. 在 Python 中如何使用多进制数字?

答： 我们在 Python 中，除十进制外还可以使用二进制、八进制和十六进制

- 二进制数字由 0 和 1 组成，我们使用 0b 或 0B 前缀表示二进制数

```
print(int(0b1010))#10
```

- 使用 bin()函数将一个数字转换为它的二进制形式

```
print(bin(0xf))#0b1111
```

- 八进制数由数字 0-7 组成，用前缀 0o 或 0O 表示 8 进制数

```
print(oct(8))#0o10
```

- 十六进数由数字 0-15 组成，用前缀 0x 或者 0X 表示 16 进制数

```
print(hex(16))#0x10
print(hex(15))#0xf
```

121. 怎样声明多个变量并赋值?

答：Python 是支持多个变量赋值的，代码示例如下

```
#对变量 a,b,c 声明并赋值
a,b,c = 1,2,3
```

算法和数据结构

122. 已知：

```
AList = [1,2,3]
BSet = {1,2,3}
```

(1) 从 AList 和 BSet 中 查找 4，最坏时间复杂度哪个大？ (2) 从 AList 和 BSet 中 插入 4，最坏时间复杂度哪个大?

答： (1) 对于查找，列表和集合的最坏时间复杂度都是 O(n)，所以一样的。 (2) 列表操作插入的最坏时间复杂度为 o(n),集合为 o(1)，所以 Alist 大。 set 是哈希表所以操作的复杂度基本上都是 o(1)。

123. 用 Python 实现一个二分查找的函数

答：

```
def binary_search(arr, target):
    n = len(arr)
    left = 0
    right = n-1
    while left <= right :
        mid = (left + right)//2
        if arr[mid] < target:
            left = mid + 1
        elif arr[mid] > target:
            right = mid - 1
        else:
            print(f"index: {mid},value: {arr[mid]}")
            return True
    return False
```

```
if __name__ == '__main__':
     l = [1,3,4,5,6,7,8]
     binary_search(l,8)
```

124. Python 单例模式的实现方法

答：实现单例模式的方法有多种，之前再说元类的时候用 call 方法实现了一个单例模式，另外 Python 的模块就是一个天然的单例模式，这里我们使用 new 关键字来实现一个单例模式。

```
"""
通过 new 函数实现简单的单例模式。
"""
class Book:
    def __new__(cls, title):
        if not hasattr(cls, "_ins"):
            cls._ins = super().__new__(cls)
        print('in __new__')
        return cls._ins

    def __init__(self, title):
        print('in __init__')
        super().__init__()
        self.title = title


if __name__ == '__main__':
    b = Book('The Spider Book')
    b2 = Book('The Flask Book')
    print(id(b))
    print(id(b2))
    print(b.title)
    print(b2.title)
```

125. 使用 Python 实现一个斐波那契数列

答： 斐波那契数列：数列从第 3 项开始，每一项都等于前两项之和。

```
def fibonacci(num):
    """
    获取指定位数的列表
    : param num:
    : return:
    """
    a, b = 0, 1
    l = []
    for i in range(num):
        a, b = b, a + b
        l.append(b)
    return l


if __name__ == '__main__':
    print(fibonacci(10))
```

126. 找出列表中的重复数字

答：

```
"""
从头扫到尾，只要当前元素值与下标不同，就做一次判断,numbers[i]与 numbers[numbers[i]],
相等就认为找到了重复元素，返回 true,否则就交换两者，继续循环。直到最后还没找到认为没找到重复
元素。
"""


# -*- coding: utf-8 -*-
class Solution:
    def duplicate(self, numbers):
        """

        :param numbers:
        :return:
        """
        if numbers is None or len(numbers) <= 1:
            return False
        use_set = set()
        duplication = {}
        for index, value in enumerate(numbers):
            if value not in use_set:
                use_set.add(value)
            else:
                duplication[index] = value
        return duplication


if __name__ == '__main__':
    s = Solution()
    d = s.duplicate([1, 2, -3, 4, 4, 95, 95, 5, 2, 2, -3, 7, 7, 5])
    print(d)
```

127. 找出列表中的单个数字

答：

```
def find_single(l : list):
    result = 0
    for v in l:
        result ^= v
    if result == 0:
        print("没有落单元素")
    else:
        print("落单元素" ,result)

if __name__ == '__main__':
    l = [1,2,3,4,5,6,2,3,4,5,6]
    find_single(l)
```

128. 写一个冒泡排序

答：

```
"""
冒泡排序
"""
def bubble_sort(arr):
    n = len(arr)
    for i in range(n - 1):
        for j in range(n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]


if __name__ == '__main__':
    l = [1, 2, 3, 4, 5, 55, 6, 3, 4, 5, 6]
    bubble_sort(l)
    print(l)
```

129. 写一个快速排序

答：

```
"""
快速排序
"""


def quick_sort(arr, first, last):
    if first >= last:
        return
    mid_value = arr[first]
    low = first
    high = last

    while low < high:
        while low < high and arr[high] >= mid_value:
            high -= 1  # 游标左移
        arr[low] = arr[high]

        while low < high and arr[low] < mid_value:
            low += 1
        arr[high] = arr[low]
        arr[low] = mid_value

    quick_sort(arr, first, low - 1)
    quick_sort(arr, low + 1, last)


if __name__ == '__main__':
    l = [1, 2, 3, 4, 5, 55, 6, 3, 4, 5, 6]
    quick_sort(l, 0, len(l) - 1)
    print(l)
```

130. 写一个拓扑排序

答：

```
"""
拓扑排序
对应于该图的拓扑排序。每一个有向无环图都至少存在一种拓扑排序。

"""
import pysnooper
from typing import Mapping


@pysnooper.snoop()
def topological_sort(graph: Mapping):
    # in_degrees = {'a': 0, 'b': 0, 'c': 0, 'd': 0, 'e': 0, 'f': 0}
    in_degrees = dict((u, 0) for u in graph)
    for u in graph:
        for v in graph[u]:   # 根据键找出值也就是下级节点
            in_degrees[v] += 1  # 对获取到的下级节点的入度加 1
    # 循环结束之后的结果: {'a': 0, 'b': 1, 'c': 1, 'd': 2, 'e': 1, 'f': 4}
    Q = [u for u in graph if in_degrees[u] == 0]  # 入度为 0 的节点
    in_degrees_zero = []
    while Q:
        u = Q.pop()  # 默认从最后一个移除
        in_degrees_zero.append(u)  # 存储入度为 0 的节点
        for v in graph[u]:
            in_degrees[v] -= 1  # 删除入度为 0 的节点，以及移除其指向
            if in_degrees[v] == 0:
                Q.append(v)
    return in_degrees_zero


if __name__ == '__main__':
    # 用字典的键值表示图的节点之间的关系，键当前节点。值是后续节点。
    graph_dict = {
        'a': 'bf',  # 表示 a 指向 b 和 f
        'b': 'cdf',
        'c': 'd',
        'd': 'ef',
        'e': 'f',
        'f': ''
    }

    t = topological_sort(graph_dict)
    print(t)
```

131. Python 实现一个二进制计算

答：

```
"""
二进制加法
"""
def binary_add(a: str, b: str):
    return bin(int(a, 2) + int(b, 2))[2:]


if __name__ == '__main__':
    num1 = input("输入第一个数，二进制格式：\n")
    num2 = input("输入第二个数，二进制格式：\n")
    print(binary_add(num1, num2))
```

132. 有一组“+”和“-”符号，要求将“+”排到左边，“-”排到右边，写出具体的实现方法。

答：

```
"""
有一组“+”和“-”符号，要求将“+”排到左边，“-”排到右边，写出具体的实现方法。

如果让+等于 0，-等于 1 不就是排序了么。
"""
from collections import deque
from timeit import Timer

s = "++++++----+++----"


# 方法一
def func1():
    new_s = s.replace("+", "0").replace("-", "1")
    result = "".join(sorted(new_s)).replace("0", "+").replace("1", "-")
    return result


# 方法二
def func2():
    q = deque()
    left = q.appendleft
    right = q.append
    for i in s:
        if i == "+":
            left("+")
        elif i == "-":
            right("-")


def func3():
    data = list(s)
    start_index = 0
    end_index = 0
    count = len(s)
    while start_index + end_index < count:
```

```
        if data[start_index] == '-':
            data[start_index], data[count - end_index - 1] = data[count -
end_index - 1], data[start_index]
            end_index += 1
        else:
            start_index += 1
    return "".join(data)


if __name__ == '__main__':
    timer1 = Timer("func1()", "from __main__ import func1")
    print("func1", timer1.timeit(1000000))
    timer2 = Timer("func2()", "from __main__ import func2")
    print("func2", timer2.timeit(1000000))
    timer3 = Timer("func3()", "from __main__ import func3")
    print("func3", timer3.timeit(1000000))

# 1000000 测试结果
# func1 1.39003764
# func2 1.593012875
# func3 3.3487415590000005
# func1 的方式最优，其次是 func2
```

133. 单链表反转

答:

```
"""
单链表反转
"""


class Node:
    def __init__(self, val=None):
        self.val = val
        self.next = None


class SingleLinkList:
    def __init__(self, head=None):
        """链表的头部"""
        self._head = head

    def add(self, val:  int):
        """
        给链表添加元素
        : param val:  传过来的数字
        : return:
        """
        # 创建一个节点
        node = Node(val)
        if self._head is None:
            self._head = node
        else:
```

```
            cur = self._head
            while cur.next is not None:
                cur = cur.next  # 移动游标
            cur.next = node  # 如果 next 后面没了证明以及到最后一个节点了

    def traversal(self):
        if self._head is None:
            return
        else:
            cur = self._head
            while cur is not None:
                print(cur.val)
                cur = cur.next

    def size(self):
        """
        获取链表的大小
        : return:
        """
        count = 0
        if self._head is None:
            return count
        else:
            cur = self._head
            while cur is not None:
                count += 1
                cur = cur.next
            return count

    def reverse(self):
        """
        单链表反转
        思路:
        让 cur.next 先断开即指向 none，指向设定 pre 游标指向断开的元素，然后
        cur.next 指向断开的元素，再把开始 self._head 再最后一个元素的时候.
        : return:
        """
        if self._head is None or self.size() == 1:
            return
        else:
            pre = None
            cur = self._head
            while cur is not None:
                post = cur.next
                cur.next = pre
                pre = cur
                cur = post
            self._head = pre  # 逆向后的头节点


if __name__ == '__main__':
    single_link = SingleLinkList()
    single_link.add(3)
    single_link.add(5)
    single_link.add(6)
```

```
    single_link.add(7)
    single_link.add(8)
    print("对链表进行遍历")
    single_link.traversal()
    print(f"size: {single_link.size()}")
    print("对链表进行逆向操作之后")
    single_link.reverse()
    single_link.traversal()
```

134. 交叉链表求交点

答:

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None


class Solution:
    def getIntersectionNode(self, headA, headB):
        """
        : tye head1, head1:  ListNode
        : rtye:  ListNode
        """
        if headA is not None and headB is not None:
            cur1, cur2 = headA, headB

            while cur1 != cur2:
                cur1 = cur1.next if cur1 is not None else headA
                cur2 = cur2.next if cur2 is not None else headB

            return cur1
```

cur1、cur2，2 个指针的初始位置是链表 headA、headB 头结点，cur1、cur2 两个指针一直往后遍历。 直到 cur1 指针走到链表的末尾，然后 cur1 指向 headB； 直到 cur2 指针走到链表的末尾，然后 cur2 指向 headA； 然后再继续遍历； 每次 cur1、cur2 指向 None，则将 cur1、cur2 分别指向 headB、headA。 循环的次数越多，cur1、cur2 的距离越接近，直到 cur1 等于 cur2。则是两个链表的相交点。

135. 用队列实现栈

答： 下面代码分别使用 1 个队列和 2 个队列实现了栈。

```
from queue import Queue

#使用 2 个队列实现
class MyStack:

    def __init__(self):
        """
        Initialize your data structure here.
```

```
        """
        # q1 作为进栈出栈，q2 作为中转站
        self.q1 = Queue()
        self.q2 = Queue()

    def push(self, x):
        """
        Push element x onto stack.
        : type x:  int
        : rtype:  void
        """
        self.q1.put(x)

    def pop(self):
        """
        Removes the element on top of the stack and returns that element.
        : rtype:  int
        """

        while self.q1.qsize() > 1:
            self.q2.put(self.q1.get())  # 将 q1 中除尾元素外的所有元素转到 q2 中
        if self.q1.qsize() == 1:
            res = self.q1.get()  # 弹出 q1 的最后一个元素
            self.q1, self.q2 = self.q2, self.q1  # 交换 q1,q2
            return res

    def top(self):
        """
        Get the top element.
        : rtype:  int
        """
        while self.q1.qsize() > 1:
            self.q2.put(self.q1.get())  # 将 q1 中除尾元素外的所有元素转到 q2 中
        if self.q1.qsize() == 1:
            res = self.q1.get()  # 弹出 q1 的最后一个元素
            self.q2.put(res)  # 与 pop 唯一不同的是需要将 q1 最后一个元素保存到 q2 中
            self.q1, self.q2 = self.q2, self.q1  # 交换 q1,q2
            return res

    def empty(self):
        """
        Returns whether the stack is empty.
        : rtype:  bool
        """
        return not bool(self.q1.qsize() + self.q2.qsize())  # 为空返回 True，不为空
返回 False


#使用 1 个队列实现
class MyStack2(object):

    def __init__(self):
        """
        Initialize your data structure here.
        """
```

```
        self.sq1 = Queue()

    def push(self, x):
        """
        Push element x onto stack.
        : type x:  int
        : rtype:  void
        """
        self.sq1.put(x)

    def pop(self):
        """
        Removes the element on top of the stack and returns that element.
        : rtype:  int
        """
        count = self.sq1.qsize()
        if count == 0:
            return False
        while count > 1:
            x = self.sq1.get()
            self.sq1.put(x)
            count -= 1
        return self.sq1.get()

    def top(self):
        """
        Get the top element.
        : rtype:  int
        """
        count = self.sq1.qsize()
        if count == 0:
            return False
        while count:
            x = self.sq1.get()
            self.sq1.put(x)
            count -= 1
        return x

    def empty(self):
        """
        Returns whether the stack is empty.
        : rtype:  bool
        """
        return self.sq1.empty()


if __name__ == '__main__':
    obj = MyStack2()
    obj.push(1)
    obj.push(3)
    obj.push(4)
    print(obj.pop())
    print(obj.pop())
    print(obj.pop())
    print(obj.empty())
```

136. 找出数据流的中位数

答：对于一个升序排序的数组，中位数为左半部分的最大值，右半部分的最小值，而左右两部分可以是无需的，只要保证左半部分的数均小于右半部分即可。因此，左右两半部分分别可用最大堆、最小堆实现。

如果有奇数个数，则中位数放在左半部分；如果有偶数个数，则取左半部分的最大值、右边部分的最小值之平均值。

分两种情况讨论： 当目前有偶数个数字时，数字先插入最小堆，然后选择最小堆的最小值插入最大堆（第一个数字插入左半部分的最小堆）。

当目前有奇数个数字时，数字先插入最大堆，然后选择最大堆的最大值插入最小堆。 最大堆：根结点的键值是所有堆结点键值中最大者，且每个结点的值都比其孩子的值大。 最小堆：根结点的键值是所有堆结点键值中最小者，且每个结点的值都比其孩子的值小。

```
# -*- coding: utf-8 -*-
from heapq import *


class Solution:
    def __init__(self):
        self.maxheap = []
        self.minheap = []

    def Insert(self, num):
        if (len(self.maxheap) + len(self.minheap)) & 0x1:   # 总数为奇数插入最大堆
            if len(self.minheap) > 0:
                if num > self.minheap[0]:   # 大于最小堆里的元素
                    heappush(self.minheap, num)  # 新数据插入最小堆
                    heappush(self.maxheap, -self.minheap[0])  # 最小堆中的最小插入最
大堆
                    heappop(self.minheap)
                else:
                    heappush(self.maxheap, -num)
            else:
                heappush(self.maxheap, -num)
        else:   # 总数为偶数 插入最小堆
            if len(self.maxheap) > 0:   # 小于最大堆里的元素
                if num < -self.maxheap[0]:
                    heappush(self.maxheap, -num)  # 新数据插入最大堆
                    heappush(self.minheap, -self.maxheap[0])  # 最大堆中的最大元素插
入最小堆
                    heappop(self.maxheap)
                else:
                    heappush(self.minheap, num)
            else:
                heappush(self.minheap, num)

    def GetMedian(self, n=None):
        if (len(self.maxheap) + len(self.minheap)) & 0x1:
            mid = self.minheap[0]
        else:
            mid = (self.minheap[0] - self.maxheap[0]) / 2.0
        return mid
```

```
if __name__ == '__main__':
    s = Solution()
    s.Insert(1)
    s.Insert(2)
    s.Insert(3)
    s.Insert(4)
    print(s.GetMedian())
```

137. 二叉搜索树中第 K 小的元素

答： 二叉搜索树(Binary Search Tree)，又名二叉排序树(Binary Sort Tree)。 二叉搜索树是具有有以下性质的二叉树：

1. 若左子树不为空，则左子树上所有节点的值均小于或等于它的根节点的值。
2. 若右子树不为空，则右子树上所有节点的值均大于或等于它的根节点的值。
3. 左、右子树也分别为二叉搜索树。

二叉搜索树按照中序遍历的顺序打印出来正好就是排序好的顺序。所以对其遍历一个节点就进行计数，计数达到 k 的时候就结束。

```
class TreeNode:
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None


class Solution:
    count = 0
    nodeVal = 0

    def kthSmallest(self, root, k):
        """
        : type root:  TreeNode
        : type k:  int
        : rtype:  int
        """
        self.dfs(root, k)
        return self.nodeVal

    def dfs(self, node, k):
        if node != None:
            self.dfs(node.left, k)
            self.count = self.count + 1
            if self.count == k:
                self.nodeVal = node.val
                # 将该节点的左右子树置为 None,来结束递归，减少时间复杂度
                node.left = None
                node.right = None
            self.dfs(node.right, k)
```

爬虫相关

138. 在 requests 模块中，requests.content 和 requests.text 什么区别

答： requests.content 获取的是字节，requests.text 获取的是文本内容。

139. 简要写一下 lxml 模块的使用方法框架

答：

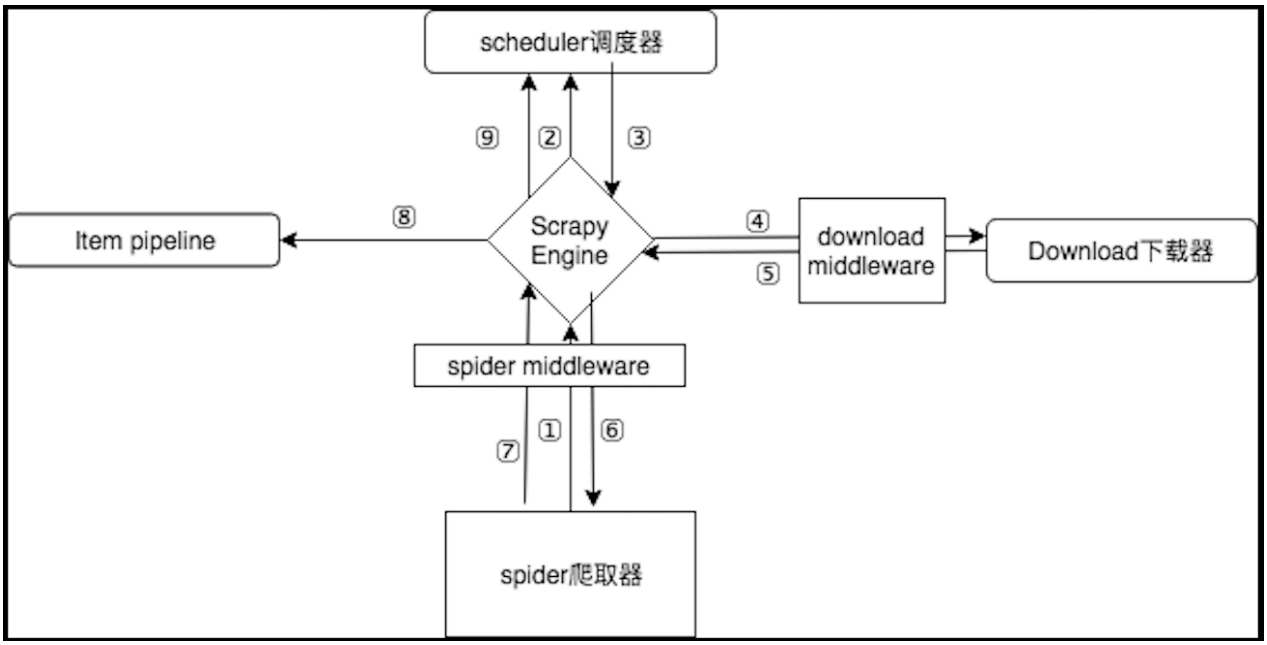
```
from lxml import html
source='''
<div class="nam"><span>中国</span></div>
root=html.fromstring(source)
_content=root.xpath("string(//div[@class='nam'])")

if _content and isinstance(_content,list):
    content=_content[0]
elif isinstance(_content,str):
    content=_content
print(content)
```

140. 说一说 scrapy 的工作流程

答：

首先还是先看张图



已 www.baidu.com 为例： 首先需要知道的事各个模块之间调用都是通过引擎进行的。

1. spider 把百度需要下载的第一个 url：www.baidu.com 交给引擎。
2. 引擎把 url 交给调度器排序入队处理。
3. 调度器把处理好的 request 返回给引擎。
4. 通过引擎调动下载器，按照下载中间件的设置下载这个 request。
5. 下载器下载完毕结果返回给引擎（如果失败：不好意思，这个 request 下载失败，然后引擎告诉调

度器，这个 request 下载失败了，你记录一下，我们待会儿再下载。）
6. 引擎调度 spider，把按照 Spider 中间件处理过了的请求，交给 spider 处理。
7. spider 把处理好的 url 和 item 传给引擎。
8. 引擎根据不同的类型调度不同的模块，调度 Item Pipeline 处理 item。
9. 把 url 交给调度器。 然后从第 4 步开始循环，直到获取到你需要的信息，

注意！只有当调度器中不存在任何 request 了，整个程序才会停止。

141. scrapy 的去重原理

答：scrapy 本身自带一个去重中间件，scrapy 源码中可以找到一个 dupefilters.py 去重器。里面有个方法叫做 request_seen，它在 scheduler(发起请求的第一时间)的时候被调用。它代码里面调用了 request_fingerprint 方法（就是给 request 生成一个指纹）。

就是给每一个传递过来的 url 生成一个固定长度的唯一的哈希值。但是这种量级千万到亿的级别内存是可以应付的。

142. scrapy 中间件有几种类，你用过哪些中间件

答： scrapy 的中间件理论上有三种(Schduler Middleware,Spider Middleware,Downloader Middleware)。在应用上一般有以下两种

1. 爬虫中间件 Spider Middleware：主要功能是在爬虫运行过程中进行一些处理。
2. 下载器中间件 Downloader Middleware：这个中间件可以实现修改 User-Agent 等 headers 信息，处理重定向，设置代理，失败重试，设置 cookies 等功能。

143. 你写爬虫的时候都遇到过什么？反爬虫措施，你是怎么解决的？

答：

- Headers： 从用户的 headers 进行反爬是最常见的反爬虫策略。Headers 是一种区分浏览器行为和机器行为中最简单的方法，还有一些网站会对 Referer （上级链接）进行检测（机器行为不太可能通过链接跳转实现）从而实现爬虫。 相应的解决措施：通过审查元素或者开发者工具获取相应的 headers 然后把相应的 headers 传输给 Python 的 requests，这样就能很好地绕过。
- IP 限制 一些网站会根据你的 IP 地址访问的频率，次数进行反爬。也就是说如果你用单一的 IP 地址访问频率过高，那么服务器会在短时间内禁止这个 IP 访问。

解决措施：构造自己的 IP 代理池，然后每次访问时随机选择代理（但一些 IP 地址不是非常稳定，需要经常检查更新）。

- UA 限制 UA 是用户访问网站时候的浏览器标识，其反爬机制与 ip 限制类似。

解决措施：使用随机 UA

- 验证码反爬虫或者模拟登陆 验证码：这个办法也是相当古老并且相当的有效果，如果一个爬虫要解释一个验证码中的内容，这在以前通过简单的图像识别是可以完成的，但是就现在来讲，验证码的干扰线，噪点都很多，甚至还出现了人类都难以认识的验证码。

相应的解决措施：验证码识别的基本方法：截图，二值化、中值滤波去噪、分割、紧缩重排（让高矮统一）、字库特征匹配识别。（Python 的 PIL 库或者其他），复杂的情况需求接入打码平台。

- Ajax 动态加载 网页的不希望被爬虫拿到的数据使用 Ajax 动态加载，这样就为爬虫造成了绝大的麻烦，如果一个爬虫不具备 js 引擎，或者具备 js 引擎，但是没有处理 js 返回的方案，或者是具备了 js 引擎，但是没办法让站点显示启用脚本设置。基于这些情况，ajax 动态加载反制爬虫还是相当有效的。

Ajax 动态加载的工作原理是：从网页的 url 加载网页的源代码之后，会在浏览器里执行 JavaScript 程序。这些程序会加载出更多的内容，并把这些内容传输到网页中。这就是为什么有些网页直接爬它的 URL 时却没有数据的原因。

处理方法：找对应的 ajax 接口，一般数据返回类型为 json。

- cookie 限制 一次打开网页会生成一个随机 cookie，如果再次打开网页这个 cookie 不存在，那么再次设置，第三次打开仍然不存在，这就非常有可能是爬虫在工作了。

解决措施：在 headers 挂上相应的 cookie 或者根据其方法进行构造（例如从中选取几个字母进行构造）。如果过于复杂，可以考虑使用 selenium 模块（可以完全模拟浏览器行为）。

144. 为什么会用到代理?

答：如果使用同一个 ip 去不断的访问的网站的话,会很容易被封 ip，严重的永久封禁，导致当前的访问不了该网站。不只是通过程序，通过浏览器也无法访问。

145. 代理失效了怎么处理?

答：一般通过大家代理池来实现代理切换等操作，来实现时时使用新的代理 ip，来避免代理失效的问题。

146. 列出你知道 header 的内容以及信息

答： User-Agent：User-Agent 的内容包含发出请求的用户信息。 Accept：指定客户端能够接收的内容类型。 Accept-Encoding：指定浏览器可以支持的 web 服务器返回内容压缩编码类型。 Accept-Language：浏览器可接受的语言。 Connection：表示是否需要持久连接。（HTTP 1.1 默认进行持久连接）。 Content-Length：请求的内容长度。 If-Modified-Since：如果请求的部分在指定时间之后被修改则请求成功，未被修改则返回 304 代码。 Referer：先前网页的地址，当前请求网页紧随其后，即来路。

147. 说一说打开浏览器访问 www.baidu.com 获取到结果，整个流程。

答： 浏览器向 DNS 服务器发送 baidu.com 域名解析请求。 DNS 服务器返回解析后的 ip 给客户端浏览器，浏览器想该 ip 发送页面请求。 DNS 服务器接收到请求后，查询该页面，并将页面发送给客户端浏览器。 客户端浏览器接收到页面后，解析页面中的引用，并再次向服务器发送引用资源请求。 服务器接收到资源请求后，查找并返回资源给客户端。 客户端浏览器接收到资源后，渲染，输出页面展现给用户。

148. 爬取速度过快出现了验证码怎么处理

答：一般在爬取过程中出现了验证码根据不同的情况，处理不一样。 如果在一开始访问就有验证码,那么就想办法绕开验证码,比如通过 wap 端或者 app 去发现其他接口等，如果不行就得破解验证码了，复杂验证码就需要接入第三方打码平台了。 如果开始的时候没有验证码，爬了一段时间才出现验证码，这个情况就要考虑更换代理 ip 了。 可能因为同一个访问频率高导致的。

149. scrapy 和 scrapy-redis 有什么区别？为什么选择 redis 数据库?

答： 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 支持分布式策略，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。

150. 分布式爬虫主要解决什么问题

答：使用分布式主要目的就是为了给爬虫加速。解决了单个 ip 的限制，宽带的影响，以及 CPU 的使用情况和 io 等一系列操作

151. 写爬虫是用多进程好？还是多线程好？ 为什么？

答： 多线程，因为爬虫是对网络操作属于 io 密集型操作适合使用多线程或者协程。

152. 解析网页的解析器使用最多的是哪几个

答：lxml，pyquery

153. 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？

答： 解决限制 IP 可以搭建代理 IP 地址池、adsl 拨号使用等。

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，一般要学会习惯性的先找需要爬取网站的 h5 端页面，看看有没有提供接口，进而简化操作。

154. 验证码的解决？

答： 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。 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。

155. 使用最多的数据库（mysql，mongodb，redis 等），对他的理解？

答： 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。

mongodb 数据库：是非关系型数据库，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。

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。

网络编程

156. TCP 和 UDP 的区别？

答： UDP 是面向无连接的通讯协议，UDP 数据包括目的端口号和源端口号信息。

优点：UDP 速度快、操作简单、要求系统资源较少，由于通讯不需要连接，可以实现广播发送。

缺点：UDP 传送数据前并不与对方建立连接，对接收到的数据也不发送确认信号，发送端不知道数据是否会正确接收，也不重复发送，不可靠。

TCP 是面向连接的通讯协议，通过三次握手建立连接，通讯完成时四次挥手。

优点：TCP 在数据传递时，有确认、窗口、重传、阻塞等控制机制，能保证数据正确性，较为可靠。

缺点：TCP 相对于 UDP 速度慢一点，要求系统资源较多。

157. 简要介绍三次握手和四次挥手

答： 三次握手 第一次握手：主机 A 发送同步报文段（SYN）请求建立连接。 第二次握手：主机 B 听到连接请求，就将该连接放入内核等待队列当中，并向主机 A 发送针对 SYN 的确认 ACK，同时主机 B 也发送自己的请求建立连接（SYN）。 第三次握手：主机 A 针对主机 BSYN 的确认应答 ACK。

四次挥手 第一次挥手：当主机 A 发送数据完毕后，发送 FIN 结束报文段。 第二次挥手：主机 B 收到 FIN 报文段后，向主机 A 发送一个确认序号 ACK（为了防止在这段时间内，对方重传 FIN 报文段）。 第三次挥手：主机 B 准备关闭连接，向主机 A 发送一个 FIN 结束报文段。 第四次挥手：主机 A 收到 FIN 结束报文段后，进入 TIME_WAIT 状态。并向主机 B 发送一个 ACK 表示连接彻底释放。

除此之外经常看的问题还有，为什么 2、3 次挥手不能合在一次挥手中？ 那是因为此时 A 虽然不再发送数据了，但是还可以接收数据，B 可能还有数据要发送给 A，所以两次挥手不能合并为一次。

158. 什么是粘包？ socket 中造成粘包的原因是什么？ 哪些情况会发生粘包现象？

答：TCP 是流式协议，只有字节流，流是没有边界的，根部就不存在粘包一说，一般粘包都是业务上没处理好造成的。

但是在描述这个现象的时候，可能还得说粘包。TCP 粘包通俗来讲，就是发送方发送的多个数据包，到接收方后粘连在一起，导致数据包不能完整的体现发送的数据。

导致 TCP 粘包的原因，可能是发送方的原因，也有可能是接受方的原因。

发送方 由于 TCP 需要尽可能高效和可靠，所以 TCP 协议默认采用 Nagle 算法，以合并相连的小数据包，再一次性发送，以达到提升网络传输效率的目的。但是接收方并不知晓发送方合并数据包，而且数据包的合并在 TCP 协议中是没有分界线的，所以这就会导致接收方不能还原其本来的数据包。

接收方 TCP 是基于“流”的。网络传输数据的速度可能会快过接收方处理数据的速度，这时候就会导致，接收方在读取缓冲区时，缓冲区存在多个数据包。在 TCP 协议中接收方是一次读取缓冲区中的所有内容，所以不能反映原本的数据信息。

一般的解决方案大概下面几种：

1. 发送定长包。如果每个消息的大小都是一样的，那么在接收对等方只要累计接收数据，直到数据等于一个定长的数值就将它作为一个消息。
2. 包尾加上\r\n 标记。FTP 协议正是这么做的。但问题在于如果数据正文中也含有\r\n，则会误判为消息的边界。
3. 包头加上包体长度。包头是定长的 4 个字节，说明了包体的长度。接收对等方先接收包体长度，依据包体长度来接收包体。

并发

159. 举例说明 concurrent.future 的中线程池的用法

答：

```
from concurrent.futures import ThreadPoolExecutor
import requests
URLS = ['http: //www.163.com', 'https: //www.baidu.com/', 'https: //github.com/']
def load_url(url):
        req= requests.get(url, timeout=60)
        print(f'{url} page is {len(req.content)} bytes')
with ThreadPoolExecutor(max_workers=3) as pool:
        pool.map(load_url,URLS)
print('主线程结束')
```

160. 说一说多线程，多进程和协程的区别。

答：概念：

进程：

```
进程是具有一定独立功能的程序关于某个数据集合上的一次运行活动，
进程是系统进行资源分配和调度的一个独立单位。每个进程都有自己的独立内存空间，
不同进程通过进程间通信来通信。由于进程比较重量，占据独立的内存，
所以上下文进程间的切换开销（栈、寄存器、虚拟内存、文件句柄等）比较大，但相对比较稳定安全。
```

线程：

```
线程是进程的一个实体,是 CPU 调度和分派的基本单位，
它是比进程更小的能独立运行的基本单位.
线程自己基本上不拥有系统资源,只拥有一点在运行中必不可少的资源(如程序计数器,一组寄存器和栈),
但是它可与同属一个进程的其他的线程共享进程所拥有的全部资源。
线程间通信主要通过共享内存，上下文切换很快，资源开销较少，但相比进程不够稳定容易丢失数据。
```

协程：

```
协程是一种用户态的轻量级线程，协程的调度完全由用户控制。
协程拥有自己的寄存器上下文和栈。
协程调度切换时，将寄存器上下文和栈保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文
和栈，
直接操作栈则基本没有内核切换的开销，可以不加锁的访问全局变量，所以上下文的切换非常快。
```

区别：进程与线程比较：线程是指进程内的一个执行单元,也是进程内的可调度实体。线程与进程的区别：

```
1) 地址空间：线程是进程内的一个执行单元，进程内至少有一个线程，它们共享进程的地址空间，
而进程有自己独立的地址空间
2) 资源拥有：进程是资源分配和拥有的单位,同一个进程内的线程共享进程的资源
3) 线程是处理器调度的基本单位,但进程不是
4) 二者均可并发执行
5) 每个独立的线程有一个程序运行的入口、顺序执行序列和程序的出口，
但是线程不能够独立执行，必须依存在应用程序中，由应用程序提供多个线程执行控制
```

协程与线程进行比较：

```
1) 一个线程可以多个协程，一个进程也可以单独拥有多个协程，这样 Python 中则能使用多核 CPU。
2) 线程进程都是同步机制，而协程则是异步
3) 协程能保留上一次调用时的状态，每次过程重入时，就相当于进入上一次调用的状态
```

161. 简述 GIL

答： GIL：全局解释器锁。每个线程在执行的过程都需要先获取 GIL，保证同一时刻只有一个线程可以执行代码。

线程释放 GIL 锁的情况：在 IO 操作等可能会引起阻塞的 systemcall 之前,可以暂时释放 GIL，但在执行完毕后, 必须重新获取 GIL，Python3.x 使用计时器（执行时间达到阈值后，当前线程释放 GIL）或 Python2.x，tickets 计数达到 100 。

Python 使用多进程是可以利用多核的 CPU 资源的。

多线程爬取比单线程性能有提升，因为遇到 IO 阻塞会自动释放 GIL 锁。

162. 进程之间如何通信

答： 可以通过队列的形式，示例如下

```python
from multiprocessing import Queue, Process
import time, random

# 要写入的数据
list1 = ["java", "Python", "JavaScript"]


def write(queue):
    """
    向队列中添加数据
    : param queue:
    : return:
    """
    for value in list1:
        print(f"正在向队列中添加数据-->{value}")
        # put_nowait 不会等待队列有空闲位置再放入数据，如果数据放入不成功就直接崩溃,比如数据满了。put 的话就会一直等待
        queue.put_nowait(value)
        time.sleep(random.random())


def read(queue):

    while True:
        # 判断队列是否为空
        if not queue.empty():
            # get_nowait 队列为空，取值的时候不等待，但是取不到值那么直接崩溃了
            value = queue.get_nowait()
            print(f'从队列中取到的数据为-->{value}')
            time.sleep(random.random())
        else:
            break
```

```python
if __name__ == '__main__':
    # 父进程创建出队列，通过参数的形式传递给子进程
    #queue = Queue(2)
    queue = Queue()

    # 创建两个进程 一个写数据 一个读数据
    write_data = Process(target=write, args=(queue,))
    read_data = Process(target=read, args=(queue,))

    # 启动进程 写入数据
    write_data.start()
    # 使用 join 等待写数据结束
    write_data.join()
    # 启动进程  读取数据
    print('*' * 20)
    read_data.start()
    # 使用 join  等待读数据结束
    read_data.join()

    print('所有的数据都写入并读取完成。。。')
```

163. IO 多路复用的作用?

答： 阻塞 I/O 只能阻塞一个 I/O 操作，而 I/O 复用模型能够阻塞多个 I/O 操作，所以才叫做多路复用。

I/O 多路复用是用于提升效率，单个进程可以同时监听多个网络连接 IO。 在 IO 密集型的系统中， 相对于线程切换的开销问题，IO 多路复用可以极大的提升系统效率。

164. select、poll、epoll 模型的区别?

答： select，poll，epoll 都是 IO 多路复用的机制。I/O 多路复用就通过一种机制，可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作。

select 模型： select 目前几乎在所有的平台上支持，其良好跨平台支持也是它的一个优点。select 的一个缺点在于单个进程能够监视的文件描述符的数量存在最大限制，在 Linux 上一般为 1024，可以通过修改宏定义甚至重新编译内核的方式提升这一限制，但 是这样也会造成效率的降低。

poll 模型： poll 和 select 的实现非常类似，本质上的区别就是存放 fd 集合的数据结构不一样。select 在一个进程内可以维持最多 1024 个连接，poll 在此基础上做了加强，可以维持任意数量的连接。

但 select 和 poll 方式有一个很大的问题就是，我们不难看出来 select 是通过轮训的方式来查找是否可读或者可写，打个比方，如果同时有 100 万个连接都没有断开，而只有一个客户端发送了数据，所以这里它还是需要循环这么多次，造成资源浪费。所以后来出现了 epoll 系统调用。

epoll 模型： epoll 是 select 和 poll 的增强版，epoll 同 poll 一样，文件描述符数量无限制。但是也并不是所有情况下 epoll 都比 select/poll 好，比如在如下场景：在大多数客户端都很活跃的情况下，系统会把所有的回调函数都唤醒，所以会导致负载较高。既然要处理这么多的连接，那倒不如 select 遍历简单有效。

165. 什么是并发和并行?

答： “并行是指同一时刻同时做多件事情，而并发是指同一时间间隔内做多件事情”。

并发与并行是两个既相似而又不相同的概念：并发性，又称共行性，是指能处理多个同时性活动的能力；并行是指同时发生的两个并发事件，具有并发的含义，而并发则不一定并行，也亦是说并发事件之间不一定要同一时刻发生。

并发的实质是一个物理 CPU(也可以多个物理 CPU) 在若干道程序之间多路复用，并发性是对有限物理资源强制行使多用户共享以提高效率。 并行性指两个或两个以上事件或活动在同一时刻发生。在多道程序环境下，并行性使多个程序同一时刻可在不同 CPU 上同时执行。

并行，是每个 CPU 运行一个程序。

166. 一个线程 1 让线程 2 去调用一个函数怎么实现

答:

```
import threading


def func1(t2):
    print('正在执行函数func1')
    t2.start()


def func2():
    print('正在执行函数func2')


if __name__ == '__main__':
    t2 = threading.Thread(target=func2)
    t1 = threading.Thread(target=func1, args=(t2,))
    t1.start()
```

167. 解释什么是异步非阻塞?

答： 异步 异步与同步相对，当一个异步过程调用发出后，调用者在没有得到结果之前，就可以继续执行后续操作。当这个调用完成后，一般通过状态、通知和回调来通知调用者。对于异步调用，调用的返回并不受调用者控制。

非阻塞 非阻塞是这样定义的，当线程遇到 I/O 操作时，不会以阻塞的方式等待 I/O 操作的完成或数据的返回，而只是将 I/O 请求发送给操作系统，继续执行下一条语句。当操作系统完成 I/O 操作时，以事件的形式通知执行 I/O 操作的线程，线程会在特定时候处理这个事件。简答理解就是如果程序不会卡住，可以继续执行，就是说非阻塞的。

168. threading.local 的作用?

答： threading.local()这个方法是用来保存一个全局变量，但是这个全局变量只有在当前线程才能访问，如果你在开发多线程应用的时候，需要每个线程保存一个单独的数据供当前线程操作，可以考虑使用这个方法，简单有效。代码示例

```
import threading
import time

a = threading.local() #全局对象
```

```
def worker():
    a.x = 0
    for i in range(200):
        time.sleep(0.01)
        a.x += 1
    print(threading.current_thread(),a.x)

for i in range(20):
    threading.Thread(target=worker).start()
```

Git 面试题

169. 说说你知道的 git 命令

答： git init：该命令将创建一个名为 .git 的子目录,这个子目录含有你初始化的 Git 仓库中所有的必须文件,这些文件是 Git 仓库的骨干

git clone url：将服务器代码下载到本地

git pull：将服务器的代码拉到本地进行同步，如果本地有修改会产生冲突。

git push：提交本地修改的代码到服务器

git checkout -b branch：创建并切换分支

git status：查看修改状态

git add 文件名：提交到暂存区

git commit -m "提交内容"：输入提交的注释内容

git log：查看提交的日志情况

170. git 如何查看某次提交修改的内容

答：我们首先可以 git log 显示历史的提交列表 之后我们用 git show 便可以显示某次提交的修改内容 同样 git show filename 可以显示某次提交的某个内容的修改信息。