1. Python 基础知识

1.1 语言特征及编码规范

1.1.1 Python 的解释器有哪些?

- CPython：采用 C 语言开发的的一种解释器，目前最通用也是使用最多的解释器。
- IPython：是基于 CPython 之上的一个交互式解释器器，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，提高执行的速度。
- JPython：基于 Java 语言的解释器，可以直接将 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，然后执行。

1.1.2 列举至少 5 条 Python 3 和 Python 2 的区别?

1. Python 3 中默认使用是 UTF-8 编码，在 Python 3 版本中的中文字符是合法的。
2. Python 2 中默认使用 xrange，Python 3 已经改成了 range。
3. Python 3 中的 print 输出函数，需要使用括号，Python 2 中 print 为 class，不需要使用。
4. Python 2 中默认的字符串类型默认是 ASCII，Python 3 中默认的字符串类型是 Unicode。
5. Python 2 中除法/的返回的结果是整型，Python 3 中返回的结果是浮点类型。
6. Python 2 中声明元类：`_metaclass_ = MetaClass`，Python 3 中声明元类：`class newclass(metaclass=MetaClass):pass`。
7. 异常：在 Python 3 中处理异常也轻微的改变了，在 Python 3 中我们现在使用 as 作为关键词。捕获异常的语法由 `except exc, var` 改为 `except exc as var`。
8. 不等运算符：Python 2.x 中不等于有两种写法 `!=` 和 `<>`，Python 3.x中去掉了 `<>`，只有 `!=` 一种写法，还好，我从来没有使用 `<>` 的习惯。
9. zip 方法在 Python 2 和 Python 3 中的不同：在 Python 3.x 中为了减少内存，zip() 返回的是一个对象。如需展示列表，需手动 list() 转换。
10. Python 2 和 Python 3 中还有很多方法返回的是一个可迭代对象，并不是列表。

1.1.3 Python 中新式类和经典类的区别是什么?

在 Python 3 版本中已经取消了经典类，默认都是新式类，并且不必显式的继承 object，以下三种类的写法一样：

```
class Fruit(object):
    pass

class Fruit():
    pass

class Fruit:
    pass
```

在 Python 2 版本中，默认都是经典类，只有继承了 object 才是新式类，参考以下三种写法：

```
# 新式类写法
class Fruit(object):
    pass

# 以下两种为经典类写法
class Fruit():
    pass

class Fruit:
    pass
```

在 Python 2 版本中，经典类和新式类的主要区别：

- 经典类是采用的深度优先算法：当子类继承多个父类的情况时，如果继承的多个父类有属性相同的，根据深度优先，会以继承的第一个父类的属性为主；
- 新式类是采用的广度优先算法：当出现子类继承多个父类的属性相同时，由于采用广度优先算法，后面继承的属性会覆盖前面已经继承的属性。

1.1.4 Python 之禅是什么，Python 中如何获取 Python 之禅?

- 新建一个 Python 文件，通过 `import this` 可以获取 Python 之禅的具体内容；
- Python 之禅目的是告诉我们如何写出高效整洁的代码；
- 其实也是 Python 语言的一种宗旨——简洁高效。

```
import this

# 具体输出的结果就是Python之禅

The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

1.1.5 python中的DocStrings（解释文档）有什么作用?

DocStrings 的主要作用就是解释代码的作用，让其他人读你的代码时候，能更快速理解代码的作用是什么。

当定义一个函数后，我们可以在函数的第一行使用一对三个单引号 `'''` 或者一对三个双引号 `"""` 来定义一个文档字符串，该文档字符串就是该函数的解释文档。

使用 **doc**（注意双下划线）调用函数中的文档字符串属性（注意，文中出现的反斜杠是转义符，去除一些符号的特殊格式）。

```
def get_max(x,y):
    '''
    比较取出两个int类型中较大的那个数
    :param x: int类型数字
    :param y: int类型数字
    :return: 输出较大值
    '''
    if x > y:
        return x
    else:
        return y

get_max(3, 5)
print(get_max.__doc__) # 输出解释文档的具体内容
```

1.1.6 Python 3 中的类型注解有什么好处？如何使用？

Python 是一种高级语言，在我们编写代码的时候，我们定义的变量和定义函数里面的参数的时候，是不用区分它们的类型。但是作为作者我们知道函数的参数应该传入什么类型的数据，但是作为读者并不能快速知道变量或者参数的类型是什么。

针对上述问题，我们就可以使用类型注解，类型注解的好处就是让读者快速知道我们变量或者参数的类型是什么。

参考以下代码

```
# 通用写法
def func(x, y):
    return x + y
# 函数注解写法,表明传入的两个数字是int类型
def func(x:int, y:int) -> int:
    return x + y
```

1.1.7 Python 语言中的命名规范有哪些？

基本原则：Python 语言的标识符必须以字母、下划线 `_` 开头，数字不能作为开头，后面可以跟任意数目的字母、数字和下划线 `_`。此处的字母并不局限于 26 个英文字母，可以包含中文字符、日文字符等。

禁忌：Python 包含一系列关键字和内置函数，不建议使用它们作为变量名，防止发生冲突。

常用命名规则：

- 项目名：首字母大写、其余单词小写，多单词组合则用下划线分割
- 包名、模块名：全用小写字母
- 类名：首字母大写、其他字母小写，多单词采用驼峰
- 方法：小写单词
- 函数：若函数的参数名与保留关键字冲突，则在参数后加一个下划线，比拼音好太多

- 全局变量：采用全大写，多单词用下划线分割

注意：

1. 不论是类成员变量还是全局变量，均不使用 m 或 g 前缀。
2. 私有类成员使用单一下划线前缀标识，多定义公开成员，少定义私有成员。
3. 变量名不应带有类型信息，因为 Python 是动态类型语言。如 iValue、names_list、dict_obj 等都是不好的命名。
4. 开头、结尾一般为 Python 的自有变量，不要以这种方式命名
5. 以 `__` 开头（2 个下划线），是私有实例变量（外部不嫩直接访问），依照情况进行命名。

> 部分内容引用来源：https://www.jianshu.com/p/24f1abefbf72

1.1.8 Python 中各种下划线的作用？

关于下划线：

1. 一个前导下划线：表示非公有，也叫做保护变量，表示类对象和子类对象自己才能访问这些变量。采用 `from somemodulename import *` 的方法导入模块时，被保护的变量不会被导入。
2. 一个后缀下划线：为了避免关键字冲突，采用的一种命名方法。
3. 两个前导下划线：私有属性，当命名一个类属性可能引起名称冲突时使用。避免与子类中的属性命名冲突，无法在外部直接访问（名字已经被重整所以访问不到），类对象和子类可以访问（公有方法可以间接访问，使用重整后的名称访问）。
4. 两个前导和后缀下划线：内置的的魔法对象或者属性（有特殊用途），例如 **init** 或者 **file**。我们不要自己创造类似的名称，只需要使用他们即可。

补充：

- 私有属性和私有方法使用双前置下划线，私有属性和方法类内部，类的对象和子类可以访问
- 私有属性和私有方法外部不能直接访问
- 单前置下划线是普通方法
- 父类的私有属性和私有方法
  - 子类对象不能在自己的方法内部，直接访问父类的私有属性或私有方法
  - 子类对象可以通过父类的公有方法间接访问到私有属性或私有方法

私有属性本质：

- 类创建的时候，在 **init** 方法中，采用双前置下划线创建的属性，该属性创建后，类内部实际上对该属性进行了名字重整（改名了，私有方法和属性在外部不可以直接用属性或方法名调用，内部将私有方法和属性在前面增加了 `"_类名"`）。
- 因此实例化对象后，外界访问不到，但是使用重整后的名字可以访问。

1.1.9 单引号、双引号、三引号有什么区别？

- 单引号和双引号：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。一般建议外部使用单引号，内部使用双引号。
- 三引号：三引号也分为三单引号和三双引号，一般用于函数功能描述。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。
- 推荐：三个单引号或者双引号常用于文档注释，推荐使用三个单引号。
- 推荐：双引号，纯粹的字符串，用于打印显示，字符串的拼接，使用双引号；单引号，除了纯粹的字符串，其它的字符串、参数等都用单引号，比如字典的键值、类传入的参数等等。

1.2 文件 I/O 操作

1.2.1 Python 中打开文件有哪些模式?

参考以下表格，'+' 可读写模式（可添加到其他模式中使用）：

type	info
r	以只读方式打开文件。文件的指针将会放在文件的开头。这是默认模式。
w	打开一个文件只用于写入。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
a	打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。
rb	以二进制格式打开一个文件用于只读。文件指针将会放在文件的开头。这是默认模式。
wb	以二进制格式打开一个文件只用于写入。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
ab	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。也就是说，新的内容将会被写入到已有内容之后。如果该文件不存在，创建新文件进行写入。
r+	打开一个文件用于读写。文件指针将会放在文件的开头。
w+	打开一个文件用于读写。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
a+	打开一个文件用于读写。如果该文件已存在，文件指针将会放在文件的结尾。文件打开时会是追加模式。如果该文件不存在，创建新文件用于读写。
rb+	以二进制格式打开一个文件用于读写。文件指针将会放在文件的开头。
wb+	以二进制格式打开一个文件用于读写。如果该文件已存在则将其覆盖。如果该文件不存在，创建新文件。
ab+	以二进制格式打开一个文件用于追加。如果该文件已存在，文件指针将会放在文件的结尾。如果该文件不存在，创建新文件用于读写。

> 引用来源（CSDN博客）：https://blog.csdn.net/kobe_academy/article/details/90629723

1.2.2 Python 中 read 、readline 和 readlines 的区别?

- read()：read 表示一次性将文件的所有内容都读取到内存之中，如果文件占用空间很大，一次性读取出出现内存不足的问题。针对文件过大，内存不足的问题，我们可以使用，read(size) 一次最多读取多少的 size 字节，读取大文件时候可以多次调用该方法。
- readline()：每次只读取一行的内容，如果是文本文件，建议使用该方法，一次读取一行，逐行读取。
- readlines()：和 read() 方法一样，也是一次读取所有的内容，但是返回结果不同，该方法是按行读取后返回的一个列表，可以使用for循环取出列表中每个元素，一个元素就代表一行。

1.2.3 大文件只需读取部分内容，或者避免读取时候内存不足的解决方法?

方法 1：直接 for 循环迭代协议，避免内存不足，但是使用 with 打开文件后会自动关闭，需要再次打开。

```
# 注意：filetest.txt是一个有多行文字的一个文档，可以自己创建一个txt文档打开
with open('filetest.txt', 'r') as f:
    for line in f:
    print(line)
```

方法 2：迭代器切片操作，使用 islice。

```
# islice返回一个生成器函数，进行切片操作,取出索引为101到105的值
with open('filetest.txt', 'r') as f:
    for line in islice(f, 101, 105):
        print(line)
    for line in islice(f, 5): #只给一个参数，指定的是结束的位置
        print(line)
```

1.2.4 什么是上下文？with 上下文管理器原理？

with 方法常用于打开文件，使用 with 方法打开文件后可以自动关闭文件，即使打开或者使用文件时出现了错误，文件也可以正常关闭。

什么是上下文（context）？

- context 其实说白了，和一篇文章中的上下文是一个意思，在通俗一点，我觉得叫环境更好。
- 上下文虽然叫上下文，但是程序里面一般都只有上文而已，只是为了叫的好听叫上下文。
- 进程中断在操作系统中是有上有下的，不过不这个高深的问题就不要深究了
- 任何实现了 **enter**() 和 **exit**() 方法的对象都可以称之为上下文管理器，上下文管理器对象可以使用with关键字。显然，文件（file）对象也实现了上下文管理器。

那么文件对象是如何实现这两个方法的呢？我们可以模拟实现一个自己的文件类，让该类实现 **enter**() 和 **exit**() 方法。

```
'''
Author: Felix
WX: AXiaShuBai
Email: xiashubai@gmail.com
Blog: https://blog.csdn.net/u011318077
Desc:
'''
# 文件管理器
class File():

    # 初始化方法
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode

    # 打开文件的方法，返回我们打开的对象，也就是传入的文件
    def __enter__(self):
        print("entering")
        self.f = open(self.filename, self.mode)
        return self.f
```

```
    # 关闭文件的方法，不用我们自己手动去关闭文件对象了
    def __exit__(self, *args):
        print("will exit")
        self.f.close()

with File('test.txt', 'w') as f:
    print("writing")
    f.write('hello, world')
```

1.2.5 什么是全缓冲、行缓冲和无缓冲?

- 全缓冲：Python 中默认的缓冲区是 4096 字节，当缓冲区里面的空间占满后，就将数据写入到磁盘中。
- 行缓冲：当写入的数据，每遇到换行符，就将缓冲中的数据写入到磁盘中。
- 无缓冲：无缓冲，顾名思义就是一写入数据，就将数据写入到磁盘中。
- 三种缓冲，可以通过 buffering 参数进行设置。

```
# -*- coding:utf-8 -*-

# python中默认的缓冲区(全缓冲)是4096字节
# buffering可以设置缓冲字节大小，当写入的数据超过设置值，才会写入到文件中
f = open('001_测试缓冲案例文件.txt', 'w', buffering=2048)
# 写入3个字节，打开txt文件为空
f.write('abc')
# 写入2045个字节，总共2048个，还是为空
f.write('*' * 2045)
# 此时我们在写入一个字节，就由缓冲存储到磁盘了，此时打开txt文件就可以看见数据了


# 行缓冲： buffering=1
# f = open('001_测试缓冲案例文件.txt', 'w', buffering=1)
# f.write('abc')
# 只要遇到换行符，就将缓存存到磁盘
# f.write('\n')


# 无缓冲：buffering=0
# 写入数据就直接存储到磁盘
```

1.2.6 什么是序列化和反序列化？JSON 序列化时常用的四个函数是什么?

什么是序列化：我们程序中的变量和对象（比如文字、图片等内容），在传输的时候需要使用二进制数据，将这些变量或对象转换为二进制数据的过程，就是序列化。

什么是反序列化：反序列化就是序列化的逆过程，把获取的二进制数据重建为变量或对象。实际序列化和反序列化就是二进制数据和原始数据格式之间的一个转换过程。

JSON 中常用的四个函数：

- json.dump：将数据序列化到文件中
- json.load：将文件中的内容反序列化读取出来
- json.dumps：将 Python 格式转化为 JSON 的字符串形式（序列化）
- json.loads：将 JSON 的字符串格式转换为 Python 的数据格式（反序列化）

- 上面容易混淆，记住 load 是下载的意思，就是将 JSON 的读取出来或者转换为 Python 的数据格式；dump 是倾倒，意思就是把数据放进去，正好和 load 相反。

注意：标准 JSON 格式数据要使用双引号。

1.2.7 JSON 中 dumps 转换数据时候如何保持中文编码?

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

```
# file = open('papers.json', 'w', encoding='utf-8')
# 将item字典类型的数据转换成json格式的字符串，
# 注意json.dumps序列化时对中文默认使用的ascii编码，要想写入中文，加上ensure_ascii=False
# line = json.dumps(dict(item), ensure_ascii=False) + "\n"
# file.write(line)

import json
a=json.dumps({"python": "你好"},ensure_ascii=False)
print(a)

# {"python": "你好"}
```

1.3 数据类型

1.3.1 Python 中的可变和不可变数据类型是什么?

我们可以判断对象的内存地址是否变化类确定数据的类型：

- 可变数据类型：数据的值改变，对象的 id 值不变；
- 不可变数据类型：数据的值改变，对象的 id 值也随之改变。

如何确定一种数据类型是可变的还是不可变的，在改变 value 值的同时，使用 id() 函数查看变量 id 值是否变化即可。

注意：字典是可变类型，但是字典中的 key 是不可变类型。

数据类型	可变/不可变
整型	不可变
字符串	不可变
元组	不可变
列表	可变
集合	可变
字典	可变

1.3.2 is 和 == 有什么区别?

- is：把两个的对象的 id 值进行比较，看它们是否相等，是否指向同一个内存地址，如果 id 值相等，就是同一个实例对象。

- `==`：把两个对象的内容或者值拿出来比较，调用的是 eq() 方法，值如果相等就是 true，但是它们的 id 不一定相同。

1.3.3 Python 中的单词大小写转换和字母统计？

- 每个单词首字母大写：title()
- 句子第一个字母大写：capitalize()
- 所有字母大写小写：upper()、lower()
- 统计某个字母出现的次数：count('h')

1.3.4 字符串，列表，元组如何反转？反转函数 reverse 和 reversed 的区别？

- reverse：是列表反转的一个方法，用于整个列表元素顺序反转排列。
- reversed：也是一个反转的方法，但是返回的不是一个列表，而是一个迭代器，只要是可迭代对象，都可以使用该方法进行反转。反转后的迭代器可以使用 for 循环或者 next 方法取出元素。

```
# reverse列表反转
ls1 = [1, 2, 3]
ls1.reverse()
print(ls1)

# reversed: reversed()的作用之后，返回的是一个把序列值经过反转之后的迭代器，
ls2 = [4, 5, 6]
print(reversed(ls2)) # 迭代器对象的内存地址
print(list(reversed(ls2)))

# 字符串反转方法1
str1 = 'abced12345'
str2 = ''.join(reversed(str1)) # 使用join拼接反转后的字符
print(str2)

# 字符串反转方法2
str3 = str1[::-1]
print(str3)

# 输出结果：
# [3, 2, 1]
# <list_reverseiterator object at 0x0003545427A5454682E8>
# [6, 5, 4]
# 54321decba
# 54321decba
```

1.3.5 Python 中的字符串格式化的方法有哪些？f-string 格式化知道吗？

- %s：s 代表字符串
- %d：d 代表数字
- %.2f：数字保留 2 位小数
- {}.format()
- f-string：Python 3.6 才开始支持，更加简洁

具体使用参考下面代码：

```
# %s
message = 'I love %s' % (Chengdu)
```

```
print(message) # I love Chengdu

# format
message = 'I love {}'.format(Chengdu)
print(message) # I love Chengdu

# f-string
res= f"{2 * 3}"
print(res)  # 6

student = {'name': 'Felix', 'age': 20}
res1 = f"The student is {student['name']}, aged {student['age']}."
print(res1)
# 'The student is Felix, aged 20.'
```

1.3.6 含有多种符号的字符串分割方法?

- 方法 1：使用 `[]+`，中括号里面写上所有的符号，后面的 `+` 表示可以出现多次，类似于正则表达式
- 方法 2：使用竖线 `|` 表示或

```
# 含有多种分隔符分割
s = 'ab;cd%e\tfg,,jklioha;hp,vrww\tyz'

# 方法1，使用中括号：
# 正则表达式分割,推荐使用
import re
# 将多个分隔符直接写在正则表达式中，使用中括号，后面的+号表示前面的符号可以出现多次
t = re.split(r'[;%,\t]+', s)
print(t)
# ['ab', 'cd', 'e', 'fg', 'jklioha', 'hp', 'vrww', 'yz']

# 方法2：使用|表示或
s="info: xiaoZhang 33 shandong"
res = re.split(r': | ', s)
print(res)
# ['info', 'xiaoZhang', '33', 'shandong']
```

1.3.7 嵌套列表转换为列表，字符串转换为列表的方法

嵌套列表到列表，双 for 循环：

```
# 嵌套列表到单层列表
l1 = [[1,2],[3,4],[5,6]]
# 先从列表l1中取出每个列表i，然后从每个列表i中取出元素j，然后由j生成一个新的列表
l2 = [j for i in l1 for j in i]
print(l2)

# 字符串到列表，使用逗号分隔，如果只是单纯的字符串，直接使用list就可以生成列表
s1 = 'a,b,c,1,2,3'
l3 = s1.split(',')
print(l3)

# [1, 2, 3, 4, 5, 6]
# ['a', 'b', 'c', '1', '2', '3']
```

1.3.8 列表合并的常用方法?

- 使用 extend 方法扩展列表
- 使用加号，两个列表相加，会合并为一个列表
- for 循环取出，然后使用 append 方法，将一个列表元素添加到另外一个列表中

1.3.9 列表如何去除重复的元素，还是保持之前的排序?

使用 set 集合和 sorted 排序，参考代码：

```
l = ['1', '5', '1', '3', '3',]
def list_new(l):
    # 先使用set生成一个集合，会自动删除掉重复的元素
    # 使用sorted排序，排序还是按照以前的索引值
    list_new = sorted(set(l), key=l.index)
    print(list_new)

list_new(l)

# ['1', '5', '3']
```

1.3.10 列表数据如何筛选，筛选出符合要求的数据?

- for 循环，筛选出符合条件的数据，添加到一个空列表中
- 过滤函数 filter 配合匿名函数 lambda 表达式
- 列表解析（推荐使用，效率最高）

```
'''
Author: Felix
WX: AXiaShuBai
Email: xiashubai@gmail.com
Blog: https://blog.csdn.net/u011318077
Desc:
'''

# 列表筛选：
# 方法1：使用过滤函数filter
from random import randint
import time
data = [randint(-10, 10) for _ in range(10)]
```

```
print(data)
print("*" * 100)
# 过滤函数：符合条件的就留下，使用list生成一个列表
res = list(filter(lambda x: x >= 0, data))
print(res)
print("*" * 100)


# 方法2：列表解析,速度更快，推荐使用
start =time.time()
res = [x for x in data if x >= 0]
end = time.time()
print('Running time: %s Seconds'%(end-start))
print(res)

# [5, -6, -10, 3, -4, -6, -9, 10, -3, -1]
#
****************************************************************************************
******************
# [5, 3, 10]
#
****************************************************************************************
******************
# Running time: 0.0 Seconds
# [5, 3, 10]
```

1.3.11 字典中元素的如何排序？sorted 排序函数的使用详解？

- 使用 sorted 排序函数：第一个参数是一个可迭代对象，第二个参数 key 传入排序的规则，使用 lambda 表达式，第三个参数 reverse，默认是 False，可以传入 True 代表倒序排列
- lambda 表达式传入的规则可以传入一个元组包含多个规则，规则中进行判断，不满足的就是 False，排在在后

```
'''
Author: Felix
WX: AXiaShuBai
Email: xiashubai@gmail.com
Blog: https://blog.csdn.net/u011318077
Desc:
'''

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

# 排列列表中字典，直接排列字典也是相同的方法
# 默认reverse=False
d2 = sorted(d1, key=lambda x: x['age'])
print(d2)

# 倒序，传入reverse=True
d3 = sorted(d1, key=lambda x: x['age'], reverse=True)
```

```
print(d3)


# 也可以传入索引值，用于排序
students = [('john', 'A', 15), ('jane', 'B', 12), ('dave','B', 10)]
d4 = sorted(students,key=lambda x: x[2])
print(d4)

# [{'name': 'bob', 'age': 18}, {'name': 'Carl', 'age': 28}, {'name': 'alice',
'age': 38}]
# [{'name': 'alice', 'age': 38}, {'name': 'Carl', 'age': 28}, {'name': 'bob',
'age': 18}]
# [('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
# 先看一下Boolean value 的排序：
# Boolean 的排序会将 False 排在前，True排在后
print(sorted([True,False]))

s = 'abC234568XYZdsf23'
# 排序规则：小写<大写<奇数<偶数
# 原理：先比较元组的第一个值，FALSE<TRUE，如果相等就比较元组的下一个值，以此类推。
print("".join(sorted(s, key=lambda x: (x.isdigit(), x.isupper(), x.isdigit() and
int(x) % 2 == 0, x))))

# False排在后面
# 1.x.isdigit()的作用是把字母放在前边,数字放在后边.
# 2.x.isdigit() and int(x) % 2 == 0的作用是保证奇数在前，偶数在后。
# 3.x.isupper()的作用是在前面基础上,保证字母小写在前大写在后.
# 4.最后的x表示在前面基础上,对所有类别数字或字母排序。
# 同时满足上面的规则


list1=[7, -8, 5, 4, 0, -2, -5]
# 要求1.正数在前负数在后 2.整数从小到大 3.负数从大到小
# 先按照正负排先后，再按照大小排先后
print(sorted(list1,key=lambda x:(x<0, abs(x))))

# x<0，表示负数在后面，正数在前面
# abs(x)表示按绝对值，小的前面，大的在后面

# [False, True]
# abdfsCXYZ33522468
# [0, 4, 5, 7, -2, -5, -8]
```

> 代码中部分案例引用来源（CSDN 博客）：https://blog.csdn.net/u013759354/article/details/80243705

1.3.12 字典如何合并？字典解包是什么？

- 字典合并：a.update(b)
- 字典解包 {**a,**b}

```
# 方法1：
# 使用update方法，将b字典添加到a中，类似于列表的append方法
a = {"A": 1,"B": 2}
b = {"C": 3,"D": 4}
a.update(b)
```

```
print(a)

# 方法2:
# 使用字典解包
a = {"A": 1,"B": 2}
b = {"C": 3,"D": 4}
print({**a})
print({**a,**b})

# {'A': 1, 'B': 2, 'C': 3, 'D': 4}
# {'A': 1, 'B': 2}
# {'A': 1, 'B': 2, 'C': 3, 'D': 4}
```

1.3.13 字典推导式使用方法？字典推导式如何格式化 cookie 值？

- 字典生成式和列表生成式一样，都是使用 for 循环，可以多层 for 循环嵌套
- 字典的键值交换：`{v: k for k,v in d.items()}`
- 字典生成式常用于 cookie 值转换为字典格式

```
# 字典键值交换
d = {'a': '1', 'b': '2', 'c': '3'}
print({v: k for k,v in d.items()})

# 字典推导式的常用于格式化cookie值:
# 字符串分割后得到字典
cookies = "anonymid=k06r6sdauyh36v; depovince=ZGQT; _r01_=1;
JSESSIONID=abcOraT1E7z0JhHDATb0w; ick_login=8f53ebf1-b972-4572-8f77-810953dcfdfe;
first_login_flag=1; ln_uact=55555835@qq.com;loginfrom=null; wp_fold=0"
# 使用字典推导式将上述字符串转化为一个字典，先使用;分割得到一个列表，
# 列表中每一个元素再用=进行分割，列表第一个值为键，第二个值为值
cookies ={i.split("=")[0]: i.split("=")[1] for i in cookies.split(";")}
print(cookies)

# {'1': 'a', '2': 'b', '3': 'c'}
# {'anonymid': 'k06r6sdauyh36v', ' depovince': 'ZGQT', ' _r01_': '1', '
JSESSIONID': 'abcOraT1E7z0JhHDATb0w', ' ick_login': '8f53ebf1-b972-4572-8f77-
810953dcfdfe', ' first_login_flag': '1', ' ln_uact': '55555835@qq.com',
'loginfrom': 'null', ' wp_fold': '0'}
```

1.3.14 zip 打包函数的使用？元组或者列表中元素生成字典？

- zip() 函数，传入可迭代对象作为参数，然后元素一一对应组成一个元组，然后所有元组打包为一个列表。
- zip 方法在 Python 2 和 Python 3 中的不同：在 Python 3 中为了减少内存，zip() 返回的是一个对象。如需展示列表，需手动 list() 转换。

```
# zip函数用于打包元素
# zip() 函数用于将可迭代的对象作为参数，
# 将对象中对应的元素打包成一个个元组，然后返回由这些元组组成的列表
# >>>a = [1,2,3]
# >>> b = [4,5,6]
# >>> zipped = zip(a,b)  打包为元组的列表
# [(1, 4), (2, 5), (3, 6)]
```

```python
# 生成前后元素生成一个字典
l = ['a', 'b', 'c', 'd', 'e','f']
l1 = list(zip(l[:-1],l[1:]))
print(l1)
print(dict(l1))
# [('a', 'b'), ('b', 'c'), ('c', 'd'), ('d', 'e'), ('e', 'f')]
# {'a': 'b', 'b': 'c', 'c': 'd', 'd': 'e', 'e': 'f'}
```

1.3.15 字典的键可以是哪些类型的数据?

- 字典的键是可以被 hash 的，可以被 hash 的对象就是不可变数据，不可变数据可以作为字典的键，比如数字、字符串、元组都可以
- 列表、集合、字典是可变类型不是可 hash 对象，所以不能用列表做为字典的键

1.3.16 变量的作用域是怎么决定的?

- 在 Python 中，定义一个变量，该变量的作用域是该变量被赋值时候所处的位置决定的
- 当一个变量在函数内部被定义和赋值，该变量就是一个局部变量，在函数外部使用就会出现错误
- 函数变量作用域的搜索顺序：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）

1.4 常用内置函数

1.4.1 如何统计一篇文章中出现频率最高的 5 个单词?

统计单词出现的个数，使用 collections 中的 Counter 统计类：

```python
'''
Author: Felix
WX: AXiaShuBai
Email: xiashubai@gmail.com
Blog: https://blog.csdn.net/u011318077
Desc:
'''

import re
from collections import Counter

# 英文文档案例.txt我们可以自己本地创建一个文档，或者网上下载一篇txt的英文文档都可以
# 读取整个文件为一个字符串，英文文档可以自己随便网上下载一个txt的文件
txt = open('英文文档案例.txt').read()

# 使用非字母对上面的字符串进行分割
res = re.split(r'\W+', txt)
# print(res)
# 统计每个单词出现的次数
c = Counter(res)
print(c)
print("*" * 100)

# 统计出现最多的5个单词
print('出现频率最高的5个单词: ')
print(c.most_common(5))
print("*" * 100)
```

```
# Counter({'the': 12, 'and': 10, 'can': 10, 'to': 9, 'in': 8, 'mobile': 6, 'we':
6, 'is': 6, 'us': 6, 'more': 5, 'a': 5, 'of': 5, 'it': 5, 'phone': 4, 'for': 4,
'some': 4, 'We': 4, 'information': 3, 'make': 3, 'use': 3, 'games': 3, 'playing':
3, 'our': 3, 'I': 3, 'online': 3, 'computer': 3, 'all': 3, 'phones': 2, 'school':
2, 'students': 2, 'In': 2, 'my': 2, 'opinion': 2, 'As': 2, 'The': 2, 's': 2,
'help': 2, 're': 2, 'when': 2, 'how': 2, 'going': 2, 'Internet': 2, 'much': 2,
'learn': 2, 'from': 2, 'It': 2, 'time': 2, 'they': 2, 'do': 2, 'worse': 2, 't': 2,
'study': 2, 'Nowadays': 1, 'are': 1, 'becoming': 1, 'popular': 1, 'among': 1,
'middle': 1, 'bring': 1, 'know': 1, '21st': 1, 'century': 1, 'modern': 1, 'age':
1, 'full': 1, 'A': 1, 'one': 1, 'quickest': 1, 'tools': 1, 'exchange': 1,
'fashionable': 1, 'useful': 1, 'invention': 1, 'so': 1, 'ought': 1, 'best': 1,
'Suppose': 1, 'there': 1, 'sudden': 1, 'accident': 1, 'convenient': 1, 'dial': 1,
'immediately': 1, 'There': 1, 'also': 1, 'relax': 1, 'ourselves': 1, 'by': 1,
'them': 1, 'tired': 1, 'studies': 1, 'not': 1, 'wrong': 1, 'follow': 1, 'fashion':
1, 'but': 1, 'most': 1, 'important': 1, 'thing': 1, 'right': 1, 'way': 1, 'Today':
1, 'll': 1, 'talk': 1, 'about': 1, 'lives': 1, 'interesting': 1, 'enjoyable': 1,
'Many': 1, 'like': 1, 'very': 1, 'because': 1, 'get': 1, 'English': 1, 'read': 1,
'good': 1, 'newspapers': 1, 'magazines': 1, 'clever': 1, 'send': 1, 'e': 1,
'mails': 1, 'friends': 1, 'quickly': 1, 'keep': 1, 'touch': 1, 'with': 1,
'people': 1, 'over': 1, 'world': 1, 'But': 1, 'spend': 1, 'too': 1, 'stay': 1,
'net': 1, 'bars': 1, 'day': 1, 'night': 1, 'result': 1, 'their': 1, 'lessons': 1,
'don': 1, 'well': 1, 'any': 1, 'think': 1, 'mustn': 1, 'go': 1, 'summer': 1, 'or':
1, 'winter': 1, 'holidays': 1, '': 1})
#
**********************************************************************************
******************
# 出现频率最高的5个单词:
# [('the', 12), ('and', 10), ('can', 10), ('to', 9), ('in', 8)]
#
**********************************************************************************
******************
```

1.4.2 map 映射函数按规律生成列表或集合?

map 函数接收两个参数，一个函数名，一个可迭代对象，一般传入的一个列表对象，列表中的每个元素都按照传入函数的规则生成一个新的列表。最后的返回值是一个 map 对象。map 对象是一个可迭代对象，可以使用 for 循环取出元素。

```python
# 先看一个列表[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]，以该列表为基础每个数字乘以10
# 生成一个新的列表[0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
# 代码如下:
l1 = [i for i in range(10)]
l2 = []
for i in l1:
    l2.append(i * 10)
print(l2)

# map函数实现上面的功能，代码变的更简单
l3 = [i for i in range(10)]
def mulTen(n):
    return n * 10
l4 = map(mulTen, l3)
print(type(l4))
print(l4)
```

```
# [0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
# <class 'map'>
# <map object at 0x000001F560A7B630>
```

1.4.3 filter 过滤函数如何使用？如何过滤奇数偶数平方根数？

- filter() 函数按照函数的规则过滤掉符合的元素，返回过滤后的新对象。
- 函数接收两个参数，第一个为函数，第二个为序列。
- 序列的每个元素作为参数传递给函数进行判断，然后返回 True 或 False，最后将返回 True 的元素放到新列表中。

```
# 实例1：过滤出列表中的所有奇数
# 定义得到奇数的方法，满足条件的就返回会布尔值True
# filter中满足判断条件为True的就加入到新列表中
def is_odd(n):
    return n % 2 == 1
newlist = filter(is_odd, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
print(newlist)
odd = []
for i in newlist:
    odd.append(i)
print(odd)

# 实例2：过滤出列表中的所有偶数
def is_even(n):
    return n % 2 == 0
newlist = filter(is_odd, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
even = []
for i in newlist:
    even.append(i)
print(even)

# 实例3：过滤出1~100中平方根是整数的数:1,4,9......81,100
import math
def is_sqr(x):
    return math.sqrt(x) % 1 == 0
newlist = filter(is_sqr, range(1, 101))
l = []
for i in newlist:
    l.append(i)
print(l)

# <filter object at 0x000001124675F320>
# [1, 3, 5, 7, 9]
# [1, 3, 5, 7, 9]
# [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

> 案例中部分代码引用来源：https://www.py.cn/jishu/jichu/14894.html

1.4.4 sort 和 sorted 排序函数的用法区别？

- sort 函数是用在 list 列表上的方法，sorted 可以对可迭代对象进行排序。
- list 的 sort 方法返回的是对已经存在的列表进行操作，该方法无返回值。

- sorted 方法有返回值，返回值的是一个新的 list 列表
- reverse 参数：指定排序的规则，reverse = True 降序，reverse = False 升序（默认）

```
# sorted案例
l1 = ['a', 'd', 'c', 'b']

# 按字母升序排列
str2 = sorted(l1, key=str.lower)
print(str2)

# 按字母降序排列
str2 = sorted(l1, key=str.lower, reverse=True)
print(str2)

# ['a', 'b', 'c', 'd']
# ['d', 'd', 'b', 'a']
```

1.4.5 enumerate 为元素添加下标索引?

参考代码：

```
# 索引和内容构成一个tuple类型，然后所有tuple数据生成一个元组列表
# 索引位于元组第一个元素位置，内容位于第二个元素位置

numbers = ['One', 'Two', 'Three', 'Four']
# 默认从0开始添加索引（下标）
print(enumerate(numbers))
print(list(enumerate(numbers)))
for i in enumerate(numbers):
    print(i)

# 添加起始下标位置
new_numbers = list(enumerate(numbers, start=100))
print(new_numbers)

# <enumerate object at 0x000002A833B689D8>
# [(0, 'One'), (1, 'Two'), (2, 'Three'), (3, 'Four')]
# (0, 'One')
# (1, 'Two')
# (2, 'Three')
# (3, 'Four')
# [(100, 'One'), (101, 'Two'), (102, 'Three'), (103, 'Four')]
```

1.4.6 lambda 匿名函数如何使用?

- lambda表达式用于定义一个匿名函数，一般用于定义一个小型函数
- lambda 是一个单一的参数表达式，内部没有类似 def 函数的语句

```python
# 下面就是一个lambda匿名函数，传入参数，返回值就是表达式的计算结果
func = lambda x: x * x - x
res = func(5)
print(res)

# 20
```

1.4.7 type 和 help 函数有什么作用?

- type() 函数实际上是一个类，而不是一个函数。在较早的 Python 中，type 用于创建一个动态类。但是如今 type 函数使用最多的功能，用来查看一个变量属于什么类型。
- help() 函数就是名称一样，它是是一个帮助函数，使用该函数可以查看一个函数有什么功能，该函数有的属性，以及该函数的各种信息。

2. Python 高级语法

2.1 类和元类

2.1.1 类 class 和元类 metaclass 的有什么区别?

类和元类的区别：

- 类的目的是用来来创建实例对象
- 元类的目的是用来创建类（元类是最底层的东西，是所有类的父类）

类：由于类是由元类创建得到的，因此我们可以认为类的本质是一个对象，既然类是一个对象，我们就可以像操作对象一样操作一个类：可以将类赋值给一个变量、可以复制这个类、可以为其添加属性等等。

元类我们实际开发使用并不需要考虑和使用它。

2.1.2 类实例化时候，init 和 new 方法有什么作用?

- **new** 函数：的作用就是创建一个实例对象，给它分配一个内存空间
- **init** 函数：类对象实例化的时候，里面的 self 指向刚刚创建的实例对象

参考代码：

```python
# 类实例化代码的执行过程，比如：obj1 = Province('SiChuan')

# 创建一个类对象
class Province():
    # 类属性，所有实例省份都属于中国，共有的属性
    country = 'China'
    # 初始化属性，用来接收实例中的参数
    def __init__(self, name):
        # 实例属性，实例特有，每个实例的name不相同
        self.name = name

# 创建第一个实例对象
obj1 = Province('SiChuan')

# obj1 = Province('SiChuan')该代码执行过程
# 1. 上面创建了一个类对象，已经有了一个内存空间
```

```
# 2. 实例化第一步执行类中的：__new__函数(该函数默认不用写)
# __new__函数的作用就是创建一个实例对象，给它分配一个内存空间
# 3. 接下来调用__init__函数，里面的self指向刚刚创建的实例对象
# (类对象内部的self指向obj1，self.name=name=obj1.name)
# name参数就接收传递进来的SiChuan
# 如果类中方法调用类中的其它对象属性，调用时候加上self即可
```

2.1.3 实例方法、类方法和静态方法有什么不同?

实例方法、类方法和静态方法，都是属于类的方法，不同点就是传入的参数和调用的方式。

三种方法传入的参数分别是：self（实例化对象）、cls（类本身）、无参数

三种方法的调用方式：

- 实例方法：由对象调用；至少一个 self 参数；执行实例方法时，自动将调用该方法的实例化对象赋值给 self；实例化方法中可以修改实例的属性
- 类方法：由类调用；至少一个 cls 参数；执行类方法时，自动将调用该方法的类赋值给 cls；类方法中可以修改类的属性
- 静态方法：由类调用；无默认参数；内部使用；
- 静态方法特殊性：静态方法不需要传入实例或者类，它是用来被类内部普通方法或类方法使用的一种方法

参考代码：

```
class Foo(object):
    def __init__(self, name):
        self.name = name

    def ord_func(self):
        """ 定义实例方法，至少有一个self参数，self指向实例化的对象 """
        print('实例方法')
        # 类调用自己的静态方法
        print("类调用自己的静态方法")
        self.static_func()

    @classmethod
    def class_func(cls):
        """ 定义类方法，至少有一个cls参数，cls指向类对象 """
        print('类方法')

    @staticmethod
    def static_func():
        """ 定义静态方法 ，无默认参数"""
        print('静态方法')

# 创建一个实例
f = Foo("中国")

print("实例对象可以调用实例方法，类方法，静态方法：")
# 调用实例方法，类方法，静态方法
f.ord_func()
f.class_func()
f.static_func()
```

```
print("\n类只能调用类方法和静态方法:")
# 调用类方法
Foo.class_func()
# 调用静态方法
Foo.static_func()

# Foo.ord_func()
# 因为实例方法必须传入self参数，self参数是一个实例化对象

# 输出结果:
# 实例对象可以调用实例方法，类方法，静态方法:
# 实例方法
# 类调用自己的静态方法
# 静态方法
# 类方法
# 静态方法

# 类只能调用类方法和静态方法:
# 类方法
# 静态方法
```

> 部分文字和代码引用来源（CSDN博客）：https://blog.csdn.net/weixin_42250835/article/details/89943985

2.1.4 类有哪些常用的魔法属性以及它们的作用是什么?

魔法属性，前后双下划线:

- **module**：表示当前的操作在那个模块
- **class**：表示当前操作的是那个类
- **init**：初始化方法，类实例化时，该方法自动执行
- **doc**：类的描述和说明文档信息
- **dict**：查看类所有属性（所有方法），也可以查看模块的所有属性方法，查看类实例对象的所有属性
- **call**：实例化对象后面加括号
- **str**：直接打印实例化的对象时调用方法

参考代码:

```
class Person(object):

    """__doc__用于查看类的描述信息。我是一个person的类"""

    def __init__(self, name, age):
        print("初始化方法创建类对象时自动执行")
        self.name = name
        self.__age = age

a = Person('Felix', 18)

# 查看类所有属性(所有方法),也可以查看模块的所有属性方法，查看类实例对象的所有属性
print(Person.__dict__) # 查看类的属性和方法
print('*' * 50)
print("当前操作的模块: " + a.__module__) # __module__表示当前的操作在那个模块
print('*' * 50)
print(a.__class__)  # __class__表示当前操作的是那个类
```

```
print('*' * 50)
print(a.__doc__) # 查看类实例对象的描述信息,类定义时候进行了描述
print('*' * 50)
print(a.__dict__) # 查看类实例对象的属性

# 输出结果
# 初始化方法创建类对象时自动执行
# {'__module__': '__main__', '__doc__': '__doc__用于查看类的描述信息。我是一个person的
类', '__init__': <function Person.__init__ at 0x0000022B09628E18>, '__dict__':
<attribute '__dict__' of 'Person' objects>, '__weakref__': <attribute
'__weakref__' of 'Person' objects>}
# **************************************************
# 当前操作的模块：__main__
# **************************************************
# <class '__main__.Person'>
# **************************************************
# __doc__用于查看类的描述信息。我是一个person的类
# **************************************************
# {'name': 'Felix', '_Person__age': 18}
```

2.1.5 类中的 property 属性有什么作用?

- property 属性的定义有两种方式：装饰器和类属性
- 通过调用 property 属性，可以简化获取数据的流程
- property 属性目的：优化代码，简化代码，调用 @property 装饰的方法可以获得一个属性，调用该方法写法和调用实例属性一样

参考代码：

```
class Person():

    def __init__(self, name):
        self.name = name

    # 定义一个属性的特殊方法,并且参数只有一个self，有一个返回值
    @property
    def age(self):
        return 28

# 实例化对象
obj = Person("Felix")
# 调用实例属性
name = obj.name
print(name)

# 调用property属性，方法后面的括号不能写
# 这样就类似于调用实例中的一个属性
ret = obj.age # 结果就是方法的返回值
print(ret)

# Felix
# 28
```

2.1.6 描述一下抽象类和接口类的区别和联系?

- **抽象类：**抽象类具有很强的约束性，它规定了很多方法，这些方法必须是继承类实现 的方法。抽象类一大特点就是不能实例化。
- **接口类：**接口列也定义了很多方法，但是它们的方法需要使用引用类实现。主要用于定义不同对象沟通交流的规则。

区别和联系：

1. 接口类是抽象类的一个变体，接口中所有的方法都是抽象的
2. 接口类是可以继承的，但是抽象类不能
3. 接口定义方法，没有实现的代码，而抽象类可以实现部分方法
4. 接口中基本数据类型为 static 而抽象类不是

2.1.7 类中的私有化属性如何访问?

- 私有属性在类中会进行名字重整，直接访问私有属性没有对象，需要使用重整后的名字进行私有属性访问
- 可以使用 **dict** 方法查看类和对象的所有属性和方法

参考代码：

```
class Test(object):

    def __init__(self, name, age):
        self.name = name # 实例属性，外界可以访问
        self.__age = age # 私有属性，外界不能直接访问，内部可以访问

a = Test('Felix', 18)

# 访问实例属性
print(a.name)
# 访问私有属性
# print(a.__age) # 会提示无该对象，因为属性名字已经修改了
# __dict__可以用来查看类和实例化对象的所有属性，
# 查看a对象的所有属性,发现__age已经被修改为单下划线加类的名字，_Test__age
print(a.__dict__)
# 访问私有属性
print(a._Test__age)

# Felix
# {'name': 'Felix', '_Test__age': 18}
# 18
```

2.1.8 类如何才能支持比较操作?

- 使用运算符重载
- @total_ordering 类装饰器可以根据类中定义的，小于方法和等于方法自己进行推测

参考代码：

```
class Rectangle(object):

    def __init__(self, w, h):
        self.w = w
        self.h = h
    def area(self):
```

```
        return self.w * self.h
    # < 运算符重载
    def __lt__(self, other):
        print('in__lt__')
        return self.area() < other.area()
    # <= 运算符重载
    def __le__(self, other):
        print('in__le__')
        return self.area() <= other.area()

# 创建两个矩形实例
r1 = Rectangle(2, 8)
r2 = Rectangle(4, 4)
print(r1 < r2)
print('*' * 50)
in__lt__
False
**************************************************
in__le__
True
```

> 引用来源：https://www.jianshu.com/p/1a15dfff8adc

2.1.9 hasattr()、getattr()、setattr()、delattr()分别有什么作用？

- hasattr(object,name)：判断一个函数是否有name属性或者方法，返回布尔值
- getattr(object, name)：获取对象的属性
- setattr(object, name, values)：赋值给对象的属性，如果该属性不存在，则先创建该对象，然后赋值
- delattr(object, name, values)：删除对象的属性

```
class func(object):
    name = 'Felix'
    def run(self):
        return "hello, python"

# 判断属性或方法是否存在
func = func()
res = hasattr(func, "name") # 判断对象是否有name属性，True
res = hasattr(func, "run") # 判断对象是否有run方法，True
res = hasattr(func, "age") # 判断对象是否有age属性，False

# 获取属性
func = func()
getattr(func, "name") # 获取name属性，Felix
getattr(func, "age")  # 获取不存在的属性，出现错误

# 设置属性
func= func()
res = hasattr(func, "age") # 判断age属性是否存在，False
print(res)
setattr(func, "age", 18) # 对age属性进行赋值，无返回值
res1 = hasattr(func, "age") # 再次判断属性是否存在，True
```

```
# 删除属性
delattr(func, 'name') # 返回值None
```

2.2 高级用法（装饰器、闭包、迭代器、生成器）

2.2.1 编写函数的四个原则是什么?

1. 函数的设计要简洁易懂
2. 函数的声明要简明合理
3. 函数的参数要考虑兼容
4. 函数的功能要专一，一个函数值只实现一个功能

2.2.2 函数调用参数的传递方式是值传递还是引用传递?

Python 的参数传递有：位置参数、默认参数、可变参数、关键字参数。

函数调用参数的传递方式是值传递还是引用传递、需要看具体的情况：

- 不可变数据类型使用值传递：像整数和字符串这样的不可变对象，是通过拷贝进行传递的，因为你无论如何都不可能在原处改变不可变对象。
- 可变数据类型使用引用传递：比如像列表，字典这样的对象是通过引用传递、和C语言里面的用指针传递数组很相似，可变对象能在函数内部改变。

2.2.3 Python 中 pass 语句的作用是什么?

- pass 语句就是跳过，函数编写时候，有时候只有大致的思路，具体的功能如何实现还未考虑。此时我们就可以使用 pass 语句，占用具体功能实现代码的位置，后面细节化的时候，将 pass 语句替换为具体的功能代码
- 使用 pass 语句，程序运行到该处时，不会执行任何的操作，直接跳过该语句

2.2.4 闭包函数的用途和注意事项?

闭包函数：闭包是由函数及其相关的引用环境组合而成的实体（即：闭包 = 函数 + 引用环境（数据），闭包=函数块+定义函数时的环境）

闭包函数用途：

- 用途 1，当闭包执行完后，仍然能够保持住当前的运行环境；
- 用途 2，闭包可以根据外部作用域的局部变量来得到不同的结果。这有点像一种类似配置功能的作用，我们可以修改外部的变量，闭包可以配置一些自定义化的功能，对其他整体不会产生影响，内部函数定义的变量只能局部使用，闭包根据这个变量展现出不同的功能。

闭包函数注意：闭包中是不能修改外部作用域的局部变量的；但是可以在内部函数指定变量 m 不是闭包的局部变量，inner 中的 m 就可以找到外部的 m，加上这句代码即可：`nonlocal m`。

```python
def outer(num):
    def inner(val):
        return num * val
    return inner

# 闭包函数，外部函数返回值是内部函数
m = outer(8)
print(m(5))
# 传入8返回内部函数，内部函数传入参数5，外部的参数8乘以内部的参数5就是最终的值

# 40
```

> 引用来源：https://www.cnblogs.com/huangjiangyong/p/10835768.html

2.2.5 *args 和 **kwargs 的区别?

- *args：不定个数的非关键字参数，允许函数传入多个不定量个数的非关键字参数，传入方式元组，传进的所有参数都会被 args 变量收集，它会根据传进参数的位置合并为一个元组（tuple），args 是元组类型，这就是包裹位置参数传递。
- **kwargs：不定个数的关键字参数，允许函数传入多个不定量个数的关键字参数，传入方式字典，kwargs 是一个字典（dict），收集所有关键字参数，包裹关键字参数传递。

```python
# 传入位置参数和关键字参数
# 一个关键字直接使用等号，多个关键字参数加使用字典，前面加**
def may_function(*args, **kwargs):
    print(args)
    print(kwargs)
    pass
may_function(1, 2, 3, a=3)
print('*' * 50)
may_function(1, 2, 3, **{'a': 3, 'b': 4, 'c': 5})
print('*' * 50)

# (1, 2, 3)
# {'a': 3}
# **************************************************
# (1, 2, 3)
# {'a': 3, 'b': 4, 'c': 5}
# **************************************************
```

2.2.6 位置参数、关键字参数、包裹位置参数、包裹关键字参数执行顺序及使用注意?

- 混合使用时候，要注意先后顺序，最先是位置参数，然后默认参数，其次是包裹位置参数，最后是包裹关键字参数。
- 位置参数：定义的位置和调用函数时候传入参数的位置必须一致。
- 关键字参数：关键字参数位置可以不用固定。
- *args 作为位置参数，只能是最后一个位置参数，后面的参数都是关键字参数。

2.2.7 如何进行参数拆包?

参考代码：

```python
# 参数拆包
```

```python
def funb(name, age, *args, **kwargs):
    print(name)
    print(age)
    print(args)
    print(kwargs)

# 先默认传入位置参数name和age，然后再传入包裹位置，最后传入包裹关键字
# 多个关键字，要使用字典,并且前面加**
funb('Felix', 25, 'music', 'sport', grade=2)
print('*' * 50)
funb('Felix', 25, 'music', 'sport', **{'grade':2, 'class': 3})

# Felix
# 25
# ('music', 'sport')
# {'grade': 2}
# **************************************************
# Felix
# 25
# ('music', 'sport')
# {'grade': 2, 'class': 3}
```

2.2.8 装饰器函数有什么作用？装饰器函数和普通函数有什么区别？

装饰器函数和普通函数的区别：

- 参数是一个函数（函数的参数是一个函数）；
- 返回值是一个函数（内部定义一个函数，返回值就是返回这个函数的执行结果）

装饰器好处：使用 Python 装饰器的可以在不用更改原函数的代码前提下给函数增加新的功能。

2.2.9 带固定参数和不定参数的装饰器有什么区别？

- 带固定参数的装饰器：装饰器内部函数参数和函数的参数一致
- 带不定参数的装饰器：装饰器内部函数使用 *args、**kwargs 接收函数的参数

参考代码：

```python
# 带固定参数的装饰器
import time

# 定义一个装饰函数，函数的参数是一个函数
def deco(func):
    # 定义一个内部函数，实现具体的功能，
    # 原始函数带有参数，该处传入参数到该内部函数
    def wrapper(a, b):
        startTime = time.time()
        func(a, b)
        endTime = time.time()
        msecs = (endTime - startTime) * 1000
        print("原函数获得的拓展功能，原始函数func_a运行耗时：%d ms" % msecs)
    # 装饰函数的返回值是内部函数的执行结果
    return wrapper

# 使用@符号拓展函数功能，func_a就具有了deco函数的功能
```

```python
@deco
def func_a(a, b):
    print("带有固定参数的装饰器演示：")
    time.sleep(2)
    print("传入的参数求和：%d" % (a + b))

if __name__ == '__main__':
    func_a(1, 2)

# 带有固定参数的装饰器演示：
# 传入的参数求和：3
# 原函数获得的拓展功能，原始函数func_a运行耗时：2000 ms
# 带不定参数的装饰器
import time

# 定义一个装饰函数，函数的参数是一个函数
def deco(func):
    # 定义一个内部函数，实现具体的功能，
    # 原始函数带有不定参数，该处传入不定参数到该内部函数
    def wrapper(*args, **kwargs):
        startTime = time.time()
        func(*args, **kwargs)
        endTime = time.time()
        msecs = (endTime - startTime) * 1000
        print("原函数获得的拓展功能，原始函数func_a运行耗时：%d ms" % msecs)
    # 装饰函数的返回值是内部函数的执行结果
    return wrapper

# 传入3个参数
@deco
def func_b(a, b, c):
    print("带有不定参数3个的装饰器演示：")
    time.sleep(2)
    print("传入的不定参数求和：%d" % (a + b + c))

if __name__ == '__main__':
    func_b(1, 2, 3)

# 带有不定参数3个的装饰器演示：
# 传入的不定参数求和：6
# 原函数获得的拓展功能，原始函数func_a运行耗时：2000 ms
```

> 以上该代码引用自作者本人博客：https://blog.csdn.net/u011318077/article/details/89422237

2.2.10 描述一下一个装饰器的函数和多个装饰器的函数的执行步骤？

- 一个装饰器的函数：先执行装饰器，装饰函数，然后执行函数本身，查看一个装饰器函数代码执行结果。
- 两个或多个装饰器的函数：
  - 装饰器装饰顺序：从最靠近原始函数开始进行装饰，逐级向外进行（外函数）。
  - 函数执行顺序：所有装饰函数装饰结束后，再执行原始函数，此时原始函数指向的是最后一个装饰器内函数的地址，内函数又从最外面的装饰器开始执行，逐级向里面执行（内函数），所有内函数开始执行完了，最后开始执行原始函数下的代码。
- 原则：先里后外装饰，再外后里执行，最后执行原始函数。

```python
#  该代码引用自作者本人博客
# 一个装饰器函数执行顺序
def deco(func):
    print("装饰器开始对原始函数进行装饰(我是外函数): ")
    def wrapper():
        print("...开始验证权限(我是内函数)...")
        func()
    return wrapper

@deco
def func_a():
    print("...权限验证通过...")

func_a()

# 装饰器开始对原始函数进行装饰(我是外函数):
# ...开始验证权限(我是内函数)...
# ...权限验证通过...
# 两个装饰器函数执行顺序
def deco1(func):
    print("001装饰器开始对函数进行装饰")
    def wrapper():
        print("001装饰器内函数开始执行")
        return " (001) " + func() + " (001) "
    return wrapper

def deco2(func):
    print("002装饰器开始对函数进行装饰")
    def wrapper():
        print("002装饰器内函数开始执行")
        return " [002] " + func() + " [002] "
    return wrapper

@deco2
@deco1
def func_a():
    print("...003原始函数终于执行了...")
    return "Hello Python Decorator"

ret = func_a()
print(ret)

# 001装饰器开始对函数进行装饰
# 002装饰器开始对函数进行装饰
# 002装饰器内函数开始执行
# 001装饰器内函数开始执行
# ...003原始函数终于执行了...
# [002]  (001) Hello Python Decorator (001)  [002]
```

2.2.11 知道通用装饰器和类装饰器吗?

通用装饰器:

- 参数采用不定参数 args 和 kwargs

- 返回函数返回值的通用装饰器，装饰器最后返回的是原始函数的的返回值

类装饰器：

- 类中有个魔法方法 **call** 方法，实例化对象当做一个函数执行时，就是调用的该方法
- 函数经类装饰器装饰后，函数执行时候，调用的是类中的 **call** 方法

```
# 通用装饰器
def w_test(func):
    def inner(*args, **kwargs):
        ret = func(*args, **kwargs)
        return ret
    return inner

@w_test
def test1():
    print('test1 called')
    return 'python'

# test1 called
# python
# 使用类装饰函数
class Test(object):

    def __init__(self, func):
        print('test init')
        print('func name is %s ' % func.__name__)
        self.__func = func

    def __call__(self, *args, **kwargs):
        print('装饰器中的功能')
        self.__func()

@Test
def test():
    print('this is test func')
test()

# test init
# func name is test
# 装饰器中的功能
# this is test func
```

2.2.12 浅拷贝和深拷贝的区别?

浅拷贝：copy.copy，浅拷贝只是一个顶层的拷贝

- 只拷贝了引用，并没有拷贝内容，只是将内存地址指向了被复制的对象，两个对象指向的内存地址相同
- 列表浅拷贝，列表内的对象可修改，因此只拷贝了列表中每个元素指向的内存的地址
- 字典浅拷贝，也是拷贝的每个键每个值的地址引用

深拷贝：copy.deepcopy 深拷贝是对于一个对象所有层次的拷贝（递归）

- 深拷贝后，指向了一个新的内存地址，表面内容一样，其实地址已经变化

- 列表深拷贝，里面每个元素的地址也都变了

应用：如果想对数据进行修改，不破坏原始数据，就用深拷贝，完全复制一份出来，与原数据无关。

2.2.13 元组的拷贝要注意什么?

- 注意 1：元组是不可修改对象，如果元组里面元素也是不可修改对象，使用浅拷贝和深拷贝，都只是复制引用，类似于赋值
- 注意 2：但是，元组里面的对象是可修改对象，比如列表、字典，则深拷贝则是深拷贝，地址会发生变化，浅拷贝还是复制引用

参考代码：

```
import copy

# 1. 元组内部为不可修改对象
a = (1, 2)
b = copy.copy(a)
c = copy.deepcopy(a)
print(id(b))
print(id(c))

# 2592934205320
# 2592934205320

# 2. 元组内部为可修改对象
a = [1, 2]
b = [3, 4]
c = (a, b)
d = copy.copy(c)
e = copy.deepcopy(c)
print(id(c))
print(id(d))
print(id(e))
a.append(5)
print(d)
print(e)
# 输出结果不同，深拷贝是所有拷贝，地址已发生变化，但是浅拷贝还是仅复制了引用

# 2592934180040
# 2592934180040
# 2592934205512
# ([1, 2, 5], [3, 4])
# ([1, 2], [3, 4])
```

2.2.14 全局变量是否一定要使用 global 进行声明?

在一个函数中对全局变量进行修改的时候，是否需要使用 global 进行声明：要看是否对全局变量的执行时候指向内存地址进行了修改。

- 如果修改了指向地址，即让全局变量指向了一个新的地方（内存地址），那么必须使用 global，类似于深拷贝
- 如果仅仅修改了指向空间中的数据，指向的内存地址不变，此时不用必须使用 global，类似于浅拷贝
- `+=` 操作修改了指向地址，列表内置函数（append.、insert.）只修改数据，指向地址不变

2.2.15 可迭代对象和迭代器对象有什么区别?

- 可迭代对象：可迭代对象可以实现 **iter** 方法，该方法的作用就是返回一个迭代器对象。常见的可迭代对象、字符串、列表、元组。
- 迭代器：迭代器实现了 **iter** 和 **next**() 方法。通过不断调用 next 方法可以，返回迭代器容器中的值，直到所有的值都被取出，最后抛出 StopIteration 异常。

2.2.16 描述一下 for 循环执行的步骤?

for 循环执行过程：

- 第一步：调用 **iter** 方法返回一个迭代器对象
- 第二步：调用 **next** 方法取出迭代器对象中的元素
- 第三步：所有元素取出后，处理 StopIteration 异常

2.2.17 迭代器就是生成器，生成器一定是迭代器，这句话对吗?

- 不对
- 生成器一定是迭代器是对的，但是迭代器不一定是生成器
- 生成器其实是一种特殊的迭代器，不过这种迭代器更加优雅。但是，调用的时候，它不需要像迭代器对象一样写 **iter**() 和 **next**() 方法了

2.2.18 yield 关键字有什么好处?

yield 是一个关键字，如果函数中含有 yield 关键字，该函数就变成了生成器函数。

yield 关键的好处：节省内存空间和可以暂停函数。

补充节约内存的原理：

- 每次执行到 yield，因为底层的实现就是中断的原理，保存栈帧，加载栈帧。
- 每次执行结束内存释放，执行的时候占用一点内存，消耗的内存资源就很少。

2.2.19 yield 和 return 关键字的关系和区别?

带 yield 关键字的函数是一个生成器函数，已经不是一个普通函数了，没有运行到 yield，函数后面的语句可以暂停运行。相当于保存了一个函数的运行状态。

return 关键字，运行到该处后返回一个值，但是还会继续执行后面的代码，但是 yield 会停止之后的代码继续执行，注意，只是停止生成器函数内部的代码，生成器函数外部代码不受影响。

2.2.20 简单描述一下 yield 生成器函数的执行步骤?

- 函数每次执行到 yield 暂停运行（生成器函数外部代码不受影响，正常运行）
- 下次调用时候，从 yield 之后开始运行

```
# 包含yield关键字，就变成了生成器函数
def func():
    print('开始执行.....')
    while True:
        res = yield 5
        print("res:", res)

# 下面调用函数并没有执行，可以先将后面的语句注释掉
# 下面调用func时，并不是调用执行函数，而是创建了一个生成器对象
# 生成器对象访问通过next方法
# 逐行运行代码观察效果
```

```
g = func()
print("第一次调用执行结果: ")
print(next(g))
print("*" * 100)

print("第二次调用执行结果: ")
print(next(g))
print("*" * 100)

print("第三次调用执行结果(已经开始了While循环，继续调用，输出结果都一样): ")
print(next(g))
print("*" * 100)
```

- 执行结果

```
第一次调用执行结果:
开始执行.....
5
****************************************************************************************************
第二次调用执行结果:
res: None
5
****************************************************************************************************
第三次调用执行结果(已经开始了While循环，继续调用，输出结果都一样):
res: None
5
****************************************************************************************************
```

2.2.21 生成器函数访问方式有哪几种？生成器函数中的 send() 有什么作用？

yield 生成器访问的两种方法：

- 对象调用 next() 或者 send() 函数
- 使用 for 循环访问对象

send() 函数可以传递一个参数，传递的参数进行赋值

2.2.22 Python 中递归的最大次数？

- Python 中默认的设置是 1000 次。
- 我们可以在代码中加入以下语句进行设置，改变最大次数。

```
import sys
sys.setrecursionlimit(1500) # set the maximum depth as 2000
```

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。

2.2.23 递归函数停止的条件是什么？

- 递归函数结束的条件要自己进行定义，并且定义在函数的内部

- 在要停止之前进行一个判断，判断递归的次数是否达到某一限定值

参考代码：

```
# 定义一个递归函数
def number(num):
    print(num)
    # 递归结束的条件
    if num == 5:
        return
    # 自己调用自己，每次减去1
    number(num - 1)

number(8)
8
7
6
```

2.4 模块

2.4.1 如何查看模块所在的位置?

模块所在的位置使用 **file** 方法查看。参考代码：

```
# 这两个库是直接调用的Anaconda环境中
import scrapy
import random
# 这个库是后期Pycharm安装的，安装在项目文件下的venv中
import Asterisk

# 显示模块搜索所在的路径
print(random.__file__)
print(scrapy.__file__)
print(Asterisk.__file__)

# H:\ProgramDevelop\Anaconda\lib\random.py
# H:\ProgramDevelop\Anaconda\lib\site-packages\scrapy\__init__.py
# D:\Hello World\python_work\Python_advanced_learning\venv\lib\site-
packages\Asterisk\__init__.py
```

2.4.2 import 导入模块时候，搜索文件的路径顺序?

- 最先搜索当前路径
- 然后搜索上级目录
- 再搜索当前项目虚拟环境
- 再搜索本地的 Python 环境，比如使用的 Anaconda 环境

2.4.3 多模块导入共享变量的问题?

多个 py 文件在 import 的时候，会先去 sys.module 里面检查是否已经 import 了，如果已经 import 了，就不再重复 import，否则就 import 进来。

`from aaa.yyy import x` 则不一样，test.py 中这样 from import，此时 x 就是 test 自己命名空间中的变量。所以 x 只在test.py 中有效，无论如何对 x 修改，都无法影响 yyy 中的 x。

如果需要共享变量，就不要使用 `from yyy import x` 这种形式，而是使用 `import file` ，然后就可以通过 yyy.x 来使用，然后 `yyy.x='abc'` 可以进行修改。这样处理全局性的变量就可以共享的。也就是保持一个独立的 namespace，这样 Python 不会再次导入，从而实现共享。

> 引用来源（CSDN博客）：https://blog.csdn.net/lovedingd/article/details/99556325

2.4.4 Python 常用内置模块有哪些?

- time：时间模块，获取时间相关
- random：随机生成内容,随机生成整数，浮点数等
- logging：日志模块
- os：提供系统交互功能
- 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

2.4.5 Python 中常见的异常有哪些?

参考下表：

错误名称	含义
BaseException	所有异常的基类
Exception	常规错误的基类
AttributeError	试图访问一个对象没有的树形，比如foo.x，但是foo没有属性x
IOError	输入/输出异常；基本上是无法打开文件
ImportError	无法引入模块或包；基本上是路径问题或名称错误
IndentationError	语法错误（的子类）；代码没有正确对齐
IndexError	下标索引超出序列边界，比如当x只有三个元素，却试图访问x[5]
KeyError	试图访问字典里不存在的键
KeyboardInterrupt	Ctrl+C被按下
NameError	使用一个还未被赋予对象的变量
SyntaxError	Python代码非法，代码不能编译(个人认为这是语法错误，写错了）
TypeError	传入对象类型与要求的不符合
UnboundLocalError	试图访问一个还未被设置的局部变量，基本上是由于另有一个同名的全局变量，导致你以为正在访问它
ValueError	传入一个调用者不期望的值，即使值的类型是正确的肥vgh看不放歌

2.4.6 如何捕获异常？万能异常捕获是什么？

- Python 的异常捕获常用 try...except... 结构
- 万能异常捕获： `except Exception as e`

2.4.7 Python 异常相关的关键字主要有哪些？

关键字	关键字说明
try/except	捕获异常并处理
pass	忽略异常
as	定义异常实例（except MyError as e）
else	如果try中的语句没有引发异常，则执行else中的语句
finally	无论是否出现异常，都执行的代码
raise	抛出/引发异常

2.4.8 异常的完整写法是什么？

参考代码：

```
try:
    # 输入一个整数
    num = int(input("输入一个整数: "))
    # 使用 100 除以输入的整数
    result = 100/num
    print(result)

except ValueError:
    print("请输入整数")、

except Exception as result:
    print("错误 %s" % result)

else:
    print("成功")

finally:
    print("最终执行的代码，无论上面代码执行结果如何")
```

2.4.9 包中的 init.py 文件有什么作用？

- 包中一般会包含多个 py 文件，多个文件夹
- 包下面每个文件夹中都需要创建一个 **init**.py 文件，该文件可以是一个空文件
- 该文件告诉机器这个目录属于一个包

2.4.10 模块内部的 name 有什么作用？

导入模块时候，模块内部本来会自动自动执行的代码不会被执行。参考以下代码：

```
def say_hello():
    print("你好，我是hello")

if __name__ == "__main__":
    # 文件被导入时，能够直接执行的代码不需要被执行！我们只是需要使用模块中的方法
    # 意思就是该模块导入到别的脚本中，以下代码不会被执行

    # 如果if __name__ == "__main__":不加该语句
    # 下面的代码，被导入到一个新的脚本时候，新的脚本执行时候，
    # 该代码中下面的三行代码会自动执行，这并不是我们所希望的
    # 因为我们导入模块，一般只是需要调用模块中的方法

    print(__name__)
    print("Felix开发的模块")
    say_hello()
__main__
Felix开发的模块
你好，我是hello
```

2.5 面向对象

2.5.1 面向过程和面向对象编程的区别？各自的优缺点和应用场景？

面向过程：程序设计的核心是过程（流水线式思维），过程即解决问题的步骤。

- 面向过程的设计就好比精心设计好一条流水线，考虑周全什么时候处理什么东西。
- 优点：极大的降低了写程序的复杂度，只需要顺着要执行的步骤，堆叠代码即可。
- 缺点：一套流水线或者流程就是用来解决一个问题，代码牵一发而动全身。
- 应用场景：一旦完成基本很少改变的场景，著名的例子有 Linux 内核、Git，以及 Apache HTTP Server
- 面向过程的程序设计把计算机程序视为一系列的命令集合，即一组函数的顺序执行。为了简化程序设计，面向过程把函数继续切分为子函数，即把大块函数通过切割成小块函数来降低系统的复杂度。

面向对象：程序设计的核心是对象，所有数据类型都可以视为对象，当然也可以自定义对象。

- 自定义的对象数据类型就是面向对象中的类（Class）的概念
- 优点：解决了程序的扩展性。对某一个对象单独修改，会立刻反映到整个体系中，如对游戏中一个人物参数的特征和技能修改都很容易。
- 缺点：可控性差，无法向面向过程的程序设计流水线式的可以很精准的预测问题的处理流程与结果，面向对象的程序一旦开始就由对象之间的交互解决问题，即便是上帝也无法预测最终结果。
- 应用场景：需求经常变化的软件，一般需求的变化都集中在用户层，互联网应用、企业内部软件、游戏等都是面向对象的程序设计大显身手的好地方。

面向过程编程：

```
全局变量1
全局变量2
全局变量3
...
def 函数1():
    pass
def 函数2():
    pass
```

```
def 函数3():
    pass
def 函数4():
    pass
def 函数5():
    pass
```

面向对象编程：

```
class 类(object):
    属性1
    属性2
    def 方法1(self):
        pass
    def 方法2(self):
        pass

class 类2(object):
    属性3
    def 方法3(self):
        pass
    def 方法4(self):
        pass
    def 方法5(self):
        pass
```

> 部分内容引用来源（CSDN博客）：https://blog.csdn.net/aaronthon/article/details/81714411

2.5.2 面向对象设计的三大特征是什么？

封装：

- 类将一系列的属性和方法进行了封装，在外部是不可见的，只有通过类提供的接口才能与属于类的实例对象进行信息交换
- 多个函数封装为一个 py 文件，就是函数封装为模块
- 多个属性和方法也可以封装为一个类
- 封装就是面向对象编程
- 封装是面向对象编程的第一大特性

继承：

- 类可以由自己的类派生，派生类称为子类。类派生出的类拥有父类的属性和方法
- 继承能够提升代码的重用率，即开发一个类，可以在多个子功能中直接使用
- 继承能够有效的进行代码的管理，当某个类有问题只要修改这个类就行，而其继承这个类的子类往往就会自动修改了，不用一个个需修改每个子类
- 一个子类可以继承多个父类

多态：

- 类可以根据不同的参数类型调用不同的方法，同一个方法也可以处理不同类型的参数
- 同时，子类也可以重写父类的方法，此时，父类子类的同一个方法具有了多种形态
- 调用该方法时候，子类已经重写了这个方法，就调用子类中的方法，如果子类中没有方法，直接调用父类中的方法
- 以上都是多态的具体体现

2.5.3 面向对象中有哪些常用概念?

类：类是具有共同特性的一类事物或者对象。他们一般都具有相同的特征，比如动物、城市、水果，抽象的一类事物

对象/实例：具体的一个对象，比如苹果，成都，具体的一个对象。

实例化：类到对象一个过程，即抽象化事物到具体事物的一个过程。

类和对象的关系：

- 一个具体，代表一类事物的某一个对象
- 一个抽象，代表的某一类事物

类中的内容，应该具有两个内容

- 表明事物的特征，叫做属性（变量）
- 表明事物功能或者动作，称为成员方法（函数）

2.5.4 多继承函数有那几种书写方式?

多继承的写法有以下三种方式：

- 方式 1：父类名.方法名（父类方法中的所有参数），比如：Parent.**init**(self, name)
- 方式 2：使用 super() 继承，super().方法名，比如：super().**init**(name, *args, **kwargs)，为避免多继承报错，使用不定长参数，接受参数
- 方式 3：使用 super() 继承，super(父类名, self).方法名，比如：super(Son1, self).**init**(name, age, *args, **kwargs)

2.5.5 多继承函数执行的顺序（MRO）?

- MRO：Python 3 默认里面有一个 C3 算法，C3 算法用来处理多继承的调用顺序（MRO），用来保证每个类只被调用一次的算法
- 可以使用类名.**mro** 显示出该类中的 super() 的调用顺序，结果是一个元组

比如：

```
- Grandson.__mro__
- (<class '__main__.Grandson'>, <class '__main__.Son1'>, <class '__main__.Son2'>,
<class '__main__.Parent'>, <class 'object'>)
- 执行到super()处的时候的调用顺序：先找该类Grandson的下一个，执行Son1，然后再执行Son1里面的super(),然后根据Son1里面的MRO继续执行
```

2.5.6 面向对象的接口如何实现?

接口：接口的功能就是对类中的方法做一定的约束。Python 并没有像 Java 语言一样有规定的接口。但是，可以自己实现接口功能。

自己构造类实现接口或者使用模块 abc，提供抽象类和抽象方法的功能。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。

```python
# 定义接口类的名称，以大写的I开头
class I接口:

    def func_1(self):
        # 获取一条数据的方法
        raise Exception('子类中必须实现该方法')

class 类名(I接口):
    def func_2(self):
        print(123)
```

2.6 设计模式

2.6.1 什么是设计模式?

程序中设计模式：大量的开发人员在长期的开发过程中，针对某一类问题，总结出来的通用解决方案。程序中使用设计模式，可以大大提高开发的效率，减少犯错的几率，是他人可以更好的理解代码，代码更加可靠和高效。

设计模式是一种解决问题的方法模式，常见的是工厂模式和单例模式。

2.6.2 面向对象中设计模式的六大原则是什么?

面向对象设计模式中有以下主要原则：

- 开闭原则 Open-Close Principle（OCP）：一个软件实体如类、模块和函数应该对扩展开放，对修改关闭。目的就是保证程序的扩展性好，易于维护和升级。开闭原则被称为面向对象设计的基石，实际上，其他原则都可以看作是实现开闭原则的工具和手段。
- 单一职责原则 Single-Responsibilitiy Principle（SRP）：对一个类而言，应该仅有一个引起它变化的原因。如果存在多于一个动机去改变一个类，那么这个类就具有多于一个的职责，就应该把多余的职责分离出去，再去创建一些类来完成每一个职责。
- 里氏替换原则 Liskov Substitution Principle：子类可以扩展父类的功能，但是不能改变父类原有的功能。
- 依赖倒置原则 Dependence Inversion Principle（DIP）：是一个类与类之间的调用规则。这里的依赖就是代码中的耦合。高层模块不应该依赖底层模块，二者都应该依赖其抽象了；抽象不依赖细节；细节应该依赖抽象。
- 接口隔离原则（Interface Segregation Principle）：用于恰当的划分角色和接口，具有两种含义：1、用户不应该依赖它不需要的借口；2、类间的依赖关系应该建立在最小的的接口上。
- 迪米特原则Law of Demeter（最小知识原则）：一个对象应该对其他对象有最少的了解。通俗来说就是，一个类对自己需要耦合或者调用的类知道的最少，你类内部怎么复杂，我不管，那是你的事，我只知道你有那么多公用的方法，我能调用。

> 引用来源：https://www.jianshu.com/p/59955e6c71b5

2.6.3 列举几个常见的设计模式?

- 单例模式
- 工厂模式
- 策略模式

2.6.4 Mixin 设计模式是什么？它的特点和优点?

Mixin 设计模式：可以将一些类中的功能动态的组合使用。

Mixin 的特点：

- Mixin 类具有一种具体的功能，功能一般都是单一，如果需要多个功能，则使用多个 Mixin 类。
- Mixin 类并不依赖子类的实现，子类没有继承这个 Mixin 类，同样可以正常运行，只不过是缺少了某个功能而已。

Minxin 的优点：

- 在不修改类的情况下，可以方便扩充类的功能
- 可以方便的调节组合各种功能
- 可以根据不同的功能划分各种组件
- 可以避免创建很多新的类，导致类的继承混乱

2.6.5 什么是单例模式？单例模式的作用？

单例模式是为了类实例化的时候，只有一个实例对象。目的：保证只有一个实例对象。

参考代码：

```
class City(object):
    instance = None

    def __new__(cls, *args, **kwargs):
        if not cls.instance:
            cls.instance = super().__new__(cls)
            return cls.instance
        else:
            return cls.instance

c1 = City()
c2 = City()
# 重写new方法之前，id是不同，重新之后，实例对象的id是相同的
print(id(c1))  # 1232423432429
print(id(c2))  # 1232423432429
```

2.6.7 单例模式的应用场景有那些？

单例模式常见应用场景：

- 多线程池
- 应用配置
- 数据库配置
- 数据库连接池
- 网站的计数器
- 应用程序的日志

2.7 内存管理

2.7.1 Python 的内存管理机制是什么？

内存管理机制：引用计数、垃圾回收、内存池。

引用计数：引用计数是对变量引用次数的一种标记手段，对象被引用时，我们对其计数增加 1，当该对象不被引用时，我们对其计数减去 1，如果计数变成了 0，说明该对象没有被引用，此时我们就可以删除该对象。

垃圾回收：Python 中有三种垃圾回收机制，其中引用计数为主，还有标记清除和分代回收。

内存池：Python 的内存分为大内存和小内存，分界点是 256K

- malloc 分配大内存，内存池分配小内存
- Python 的内存池是一个金字塔模型：最顶层是用户对对象的直接操作，中间是分配小内存，最底层是大内存。

2.7.2 Python 的内寸管理的优化方法？

- 手动垃圾回收机制
- 调高垃圾回收阈值
- 避免各种循环引用

2.7.3 Python 中内存泄漏有哪几种？

- 常发性内存泄漏：某段代码会被多次执行，每当执行到内存泄露的代码时，就会导致一次内存泄漏。
- 偶发性内存泄漏：某段代码只有偶尔执行，当该段代码被执行时候，发生内存泄露。
- 一次性内存泄漏：某段代码只会被执行一次，内存泄露也只会发生一次。
- 隐式内存泄漏：某段代码或者程序会一致运行，运行过程中会不断的占用内存，但是内存释放只有代码或者程序结束时候才会执行。程序如此运行下去，可能会耗光所有的内存，导致内存泄露就是隐式内存泄漏。

2.7.4 Python 中如何避免内存泄漏？

- 内存泄漏：由于代码的编写设计错误，导致内存不在使用，但是该内存并未释放，造成了内存的浪费。
- 对象间的循环引用，**del**() 函数会引起内存泄漏。不使用一个对象时使用 `del object` 来删除一个对象的引用计数就可以有效防止内存泄露问题。
- 模块 gc 可以查看不能回收的对象的相关信息，我们可以做相应处理。
- 使用 sys.getrefcount(obj) 方法获取引用计数，返回值判断是否有内存泄露。

2.7.5 内存溢出的原因有哪些？

内存溢出的主要原因：

1. 一次性读取的数量量过大；
2. 对象的引用，使用完成后并没有清空；
3. 代码编写时候使用了太多的循环，可能存在死循环等，检查代码中的循环，是否会产生无用的对象；
4. 内存值设定太小，可以将内存值改大。

2.7.6 Python 退出时是否释放所有内存分配？

- 不是
- 代码中存在循环引用其他对象，或者有全局命名空间的对象的模块，退出时候也不会完全是否内存
- C 库保留的内存部分，Python 退出时候并不会马上释放

3. 系统编程

3.1 多进程、多线程、协程、并行、并发、锁

3.1.1 并发与并行的区别和联系？

- 并发：当任务数大余 CPU 核数，通过各种任务调度算法，实现用多个任务表面看起来是同时执行，实际上是快速切换交替执行，某一时刻，有些任务并没有执行。

- 并行：当任务数小于等于 CPU 核数，每个 CPU 核都可以单独执行一个独立的任务，同一时刻，所有任务同时执行。当多进程的任务数小于 CPU 的核数，多个任务同时执行，可以实现物理意义上的并行。

3.1.2 程序中的同步和异步与现实中一样吗?

现实生活中，同步指的是同时做几件事情，异步指的是做完一件事后再做另外一件事，程序中的同步和异步是把现实生活中的概念对调：

- 程序中的异步指的是现实生活中的同步，程序中的同步指的是现实生活中的异步
- 所有程序中异步请求好于同步请求，异步请求才是一次请求多个，同步请求一次只请求一个

3.1.3 进程、线程、协程的区别和联系?

进程：进程是一个正在运行的代码或者程序，进程拥有自己的内存空间，它是资源分配的单位。特点是数据进程之间数据不共享，进程占用资源多。

线程：线程是依赖进程存在的，一个进程可以只有一个线程或者多个线程。多线程之间可以共享数据。

协程：是一种用户态的轻量级线程，协程的执行和调度由用户自主控制。

联系：

- 进程下面至少有一个线程
- 线程不能独立存在，依赖于进程

3.1.4 多进程和多线程的区别?

多进程：多个代码片段在独立同时进行，可以同时利用多个内核，真正的多并发，同一时刻占用内存为多个进程之和。

- 当多进程的任务数小于 CPU 的核数，多个任务同时执行，可以实现物理意义上的并行
- 实际开发中，多进程任务数远远大于 CPU 核数，就只能通过调度轮流执行，实现的是多并发

多线程：多个代码片段同时执行，由于存在 GIL，本质上，同时只有一个线程再执行，只有一个进程占用内存。这个不执行，下一个才执行，内部先获取锁，释放锁，并不能发挥多核性能

- 多线程是伪多并发，同一时刻占用内存还是一个进程的
- Python 使用多进程才能利用 CPU 的多核资源

3.1.5 协程的优势是什么?

- 在协程之中，调用一个任务就类似于 Python 中调用一个函数一样，消耗的资源最小
- 协程的相互切换只是单纯的操作 CPU 的上下文，因此一秒钟可以切换个上百万次
- 协程最大优势就是消耗的资源少，大量切换系统也不会产生问题。

3.1.6 多线程和多进程分别用于哪些场景?

- 计算密集型（多进程）：代码没有等待时间，使用多进程，发挥多核的性能
- IO 密集型（多线程，协程）：（input,output)，比如网络收发、文件读写，具有等待时间，优先使用协程，再考虑多线程

比如：

- 请求网络，发出请求，假设需要等待时间 1 分钟，然后返回内容，这个等待时间就可以先解锁当前线程，因为还在等待返回内容，开始下个线程。等待的 1 分钟内可以发出多个请求，返回内容回来时候，再次请求锁，不断上锁解锁。
- 网页爬取：多线程比单线程爬取性能有提升，因为遇到 iOS 阻塞会自动释放 GIL 锁。

3.1.7 全局解释器锁（GIL）是什么？如何解决 GIL 问题？

全局解释器锁（GIL）

- Python 代码的执行是由 Python 虚拟机进行控制，Python 语言本身并没有 GIL 问题，是 Python 虚拟机（Python 解释器）的问题
- 在主循环中只能有一个线程在执行，每个线程执行的时候都需要先获取 GIL，GIL 锁的作用是保证同一时刻只有一个线程在执行代码

线程释放 GIL 锁的情况：

- 在 IO 操作等可能会引起阻塞的 system call 之前，可以暂时释放 GIL，但在执行完毕后，必须重新获取 GIL

解决多线程的 GIL 问题：

- 方法 1：更换 Python 解释器（默认解释器是 C 语言编写的），换成 Java 写的解释器 jypython
- 方法 2：用其它语言替代多线程代码，Python 胶水语言，可以直接调用 C 语言，Java 语言等写的代码

3.1.8 Python 中有哪些锁（LOCK）？它们分别有什么作用？

- 全局解释器锁（GIL）：一个线程获取了锁后，必须等待该锁被释放后，其它线程才能获取该锁。全局解释器锁可以避免数据共享产生冲突。
- 同步锁：同一时刻的一个进程下的一个线程只能使用一个 CPU，要确保这个线程下的程序在一段时间内被 CPU 执行，那么就要用到同步锁。作用：因为有可能当一个线程在使用 CPU 时，该线程下的程序可能会遇到 IO 操作，那么 CPU 就会切到别的线程上去，这样就有可能会影响到该程序结果的完整性。
- 死锁：指两个或两个以上的线程或进程在执行程序的过程中，因争夺资源而相互等待的一个现象。
- 递归锁（可重入锁）：在 Python 中为了支持同一个线程中多次请求同一资源，Python 提供了可重入锁。这个 RLock 内部维护着一个 Lock 和一个 counter 变量，counter 记录了 acquire 的次数，从而使得资源可以被多次 require。直到一个线程所有的 acquire 都被 release，其他的线程才能获得资源。

3.1.9 Python 中如何实现多线程和多进程？

- 多线程使用 threading.Thread 或_thread（不推荐）
- 多进程使用 multiprocessing.Process

参考代码：

```
# 多线程演示
#coding=utf-8

import threading
import time

def multi_thread():
    print("我是多线程演示")
    time.sleep(3)

# if开始是主线程
# t.start()开始的就是子线程
if __name__ == "__main__":
```

```
    # 循环启动多个线程
    for i in range(5):
        # 传入函数名，函数不需要括号，有括号就是调用函数了
        t = threading.Thread(target=multi_thread)
        t.start() #启动线程，即让线程开始执行
# 多进程演示
# 一边吃饭一边喝酒
# -*- coding:utf-8 -*-

import multiprocessing
import os
from time import sleep, ctime

def eating():
    for i in range(3):
        print("一边吃饭......%d" % i)
        sleep(1)
        print("process1 id: ", os.getpid())

def drinking():
    for i in range(3):
        print("一边喝酒......%d" % i)
        sleep(1)
        print("process2 id: ", os.getpid())

def main():
    # 写函数名，没有括号，告诉Process要执行的函数在哪里
    # 写上函数名后面的括号就是马上执行该函数了，我们下面使用start执行函数
    p1 = multiprocessing.Process(target=eating)
    p2 = multiprocessing.Process(target=drinking)

    # 调用start后，进程才正式开始执行
    p1.start()
    p2.start()

if __name__ == '__main__':
    # 获取父进程ID,即整个if __name__ == '__main__':下面的代码
    print("parent process: ", os.getppid())
    main()
```

3.1.10 守护线程和非守护线程是什么?

- 非守护线程：Python 中默认多线程是非守护，main() 函数中的主代码执行完毕了，子线程还在继续执行
- 守护线程：setDaemon(True)，主代码结束执行结束后，子线程直接结束，即使还没有执行完成
- 为了使非守护线程，子线程执行结束后再执行主线程剩下的代码，可以使用 join 函数等待子线程执行结束

3.1.11 多线程的执行顺序是什么样的?

多线程的执行是无序的，如果要有序执行，线程启动后可以加入延时时间。

参考代码：

```
import threading
```

```
import time

class MyThread(threading.Thread):
    def run(self):
        for i in range(2):
            # 等待1秒后执行
            time.sleep(3)
            # name属性中保存的是当前线程的名字
            msg = "我是线程 "+self.name+' @ '+str(i)
            print(msg)

def test():
    for i in range(3):
        t = MyThread()
        t.start()

if __name__ == '__main__':
    test()
我是线程 Thread-2 @ 0
我是线程 Thread-1 @ 0
我是线程 Thread-3 @ 0

我是线程 Thread-3 @ 1
我是线程 Thread-2 @ 1
我是线程 Thread-1 @ 1
```

运行结果可以看出，多线程执行是无序的随机的。

3.1.12 多线程非安全是什么意思?

- 在一个进程内的所有线程共享全局变量，很方便在多个线程间共享数据
- 缺点就是，线程是对全局变量随意遂改可能造成多线程之间对全局变量的混乱（即线程非安全）
- 如果多个线程同时对同一个全局变量操作，会出现资源竞争问题，从而数据结果会不正确

参考代码:

```
import threading

sum = 0
loopSum = 100000

def myAdd():
    global sum, loopSum
    # sum初始值是0，循环第一次0+1=1，循环第二次1+1=2，循环第99次等于98+1=99
    for i in range(1, loopSum):
        sum += 1

def myMinu():
    global sum, loopSum
    for i in range(1, loopSum):
        sum -= 1

# 改写为多线程执行
if __name__ == '__main__':
    print("Starting...{0}".format(sum))
```

```
    # 开始多线程实例，多次执行，看看执行结果是否一样
    # 由于t1和t2同时开始，互相调用他们之间的运行结果sum值，
    # 最终完成后的sum值就会一直变化
    t1 = threading.Thread(target=myAdd, args=())
    t2 = threading.Thread(target=myMinu, args=())

    t1.start()
    t2.start()

    t1.join()
    t2.join()

    # 加减同时执行，最后发现每次的运行结果都不同
    # 多线程共享变量出现测冲突问题
    print("Done...{0}".format(sum))
Starting...0
Done...-30729
```

> 代码引用来源（CSDN 博客）：https://blog.csdn.net/qq_42633819/article/details/85693031

3.1.13 互斥锁是什么？有什么好处和坏处？

- 互斥锁用来解决变量冲突的问题
- 上锁解锁的具体过程：线程要使用一个变量时候，先调用 acquire() 函数获取锁，使用完成后，调用 release() 函数释放锁，同一时刻变量只能被一个线程使用。
- 锁的好处：确保了某个线程运行时候不会和其它线程产生冲突
- 锁的坏处：降低了代码执行的效率，如果某段代码出现可能会出现死锁问题。

3.1.14 什么是僵尸进程和孤儿进程？

- 孤儿进程： 当父进程已经停止运行后，有些子进程可能还在运行，并没有结束，这些进程就是孤儿进程。
- 僵尸进程：一般使用 fork 方法创建了子进程，子进程运行停止运行后，父进程并哪里不知道子进程已经结束了，以为子进程还在运行，任然占用进程描述符，子进程就变成了僵尸进程。僵尸进程产生一般是父进程中没有调用 wait 函数。

3.1.15 多线程和多进程如何实现通信？

- 多线程通信：queue.Queue()
- 多进程通信：multiprocessing.Queue
- 进程池通信：multiprocessing.Manager().Queue()

3.1.16 Python 3 中 multiprocessing.Queue() 和 queue.Queue() 的区别？

- multiprocess.Queue：不同进程之间的通信队列，主要解决多进程之间的通信问题。
- queue.Queue：同一个进程内的非阻塞队列，实际是一个普通的队列模式，先进先出。

3.1.17 如何使用多协程并发请求网页？

- 使用协程池 gevent 模块，遇到阻塞，自动切换协程
- 由于 IO 操作非常耗时，程序经常会处于等待状态，比如请求多个网页有时候需要等待，gevent 可以自动切换协程
- 遇到阻塞自动切换协程，程序启动时执行 monkey.patch_all() 解决，多个协程，可以采用 pool 管理并发数，限制并发任务数

参考代码：

```
from gevent import monkey; monkey.patch_all()
from urllib import request
from gevent.pool import Pool

def run_task(url):
    print("Visit --> %s" % url)
    try:
        response = request.urlopen(url)
        data = response.read()
        print("%d bytes received from %s." %(len(data), url))
    except Exception:
        print("error")
    return 'url:%s -->finish' % url

if __name__ == '__main__':
    # 指定协程同时执行数量
    pool = Pool(2)
    urls = ['https://github.com/', 'https://blog.csdn.net/',
'https://www.hao123.com/']
    # 协程池执行任务
    results = pool.map(run_task, urls)
    # 打出结果，打出上面返回的结果
    # map最终返回的值是一个列表，就是每个任务执行完成之后的返回值
    print(results)

# 查看运行结果可以发现，先访问了两个网址
# 第一个协程执行完毕后，开始第三个协程
# 最终的结果为每次任务执行完成后返回值，组成的一个列表
Visit --> https://github.com/
Visit --> https://blog.csdn.net/
133390 bytes received from https://github.com/.
Visit --> https://www.hao123.com/
322244 bytes received from https://blog.csdn.net/.
321144 bytes received from https://www.hao123.com/.
['url:https://github.com/ -->finish', 'url:https://blog.csdn.net/ -->finish',
'url:https://www.hao123.com/ -->finish']
```

3.1.18 简单描述一下 asyncio 模块实现异步的原理?

- asyncio 异步模块底层调用还是使用的 yield 函数，利用的 yield 函数可以中断函数运行的上下文机制，模块内已经封装 selector 方法，避免了复杂的回调关系。
- asyncio 主要功能是有一个任务调度器 event loop，使用 async def 来定义异步函数，作为任务执行的逻辑，然后使用 create_task 接口把任务放入调度器 event loop 上。event loop 的运行过程是个不停循环的过程，不停查看等待类别有没有可以执行的任务，如果有的话执行任务，直到碰到 await 之类的主动让出 event loop 的函数，如此反复，实现异步。

4. 网络编程

4.1 TCP UDP HTTP SEO WSGI 等

4.1.1 UDP 和 TCP 有什么区别以及各自的优缺点?

- UDP：Internet 协议集支持一个无连接的传输协议，该协议称为用户数据报协议（UDP，User Datagram Protocol）。UDP 为应用程序提供了一种无需建立连接就可以发送封装的 IP 数据报的方法。
- TCP：传输控制协议（TCP，Transmission Control Protocol） 是一种面向连接的、可靠的、基于字节流的传输层通信协议。TCP 通信需要经过创建连接、数据传送、终止连接三个步骤。TCP 通信模型中，在通信开始之前，一定要先建立相关的链接，才能发送数据，类似于生活中打电话。

TCP 四大特点：

- 面向连接
- 可靠传输
- 错误校验
- 流量控制和阻塞管理

TCP 和 UDP 的区别

- TCP 面向连接（read/write/send/recv），而 UDP 无连接 （sendto、recvfrom）
- TCP 是可靠传输（超时重传 + 数据应答），UDP 不可靠
- TCP 是点对点连接，UDP 可以一对多
- TCP 面向字节流，UDP 面向数据报
- TCP 给 HTTP、HTTPS、FTP、TELNET、SMTP 等使用，UDP 给 DNS、DHCP、NFS、IGMP、TFTP 等

> 以上部分解释引用来源 (百度百科)：https://baike.baidu.com/item/UDP/571511?fr=aladdin

4.1.2 IP 地址是什么？有哪几类？

IP 地址版本：IPv4（正在使用）、IPv5（未正式使用）、IPv6（正在使用）

IPv4 为例：xxx.xxx.xxx.xxx，四组数字，但是包括两部分：网络地址（网络号）和主机地址（主机号）。

比如：

- 192.168.43. 同一个局域网，前三组当做网络号
- 143 最后一组当做主机号，同一个局域网，不同电脑网络号不同

IPv4 地址分类：A/B/C/D/E 五类，我们局域网一般用的是 C 类。

注意：在这么多网络 IP 中，国际规定有一部分 IP 地址是用于我们的局域网使用，也就是属于私网 IP，不在公网中使用的，所以家里网络地址都是 192.168.xxx.xxx, 它们的范围是：10.0.0.0～10.255.255.255、172.16.0.0～172.31.255.255、192.168.0.0～192.168.255.255。

IP 地址 127.0.0.1~127.255.255.255 用于回路测试、127.0.0.1 可以代表本机 IP 地址。

4.1.3 举例描述一下端口有什作用？

- 定位到一台主机是使用的 IP 地址，但是我们电脑上有很多网络软件，比如浏览器、QQ 软件
- 此时定位到一个软件，就需要使用端口，使用 IP 地址和端口号就可以定位到我们电脑上某个软件了，但是要注意，端口不是一成不变的

4.1.4 不同电脑上的进程如何实现通信的？

- 不同的电脑需要使用网络进行通信，网络通信就需要使用 TCP/IP 协议
- 然后通过 IP 地址和端口后，每台电脑的程序就有唯一的身份标识符，此时不同电脑之间就可以通过网络进行通信

4.1.5 列举一下常用网络通信名词?

- MAC 地址：在设备与设备之间数据通信时用来标记收发双方（网卡的序列号）
- IP 地址：在逻辑上标记一台电脑，用来指引数据包的收发方向（相当于电脑的序列号）
- 网络掩码：用来区分 IP 地址的网络号和主机号
- 默认网关：当需要发送的数据包的目的 IP 不在本网段内时，就会发送给默认的一台电脑，成为网关
- 集线器：已过时，用来连接多态电脑，缺点：每次收发数据都进行广播，网络会变的拥堵
- 交换机：集线器的升级版，有学习功能知道需要发送给哪台设备，根据需要进行单播、广播
- 路由器：连接多个不同的网段，让他们之间可以进行收发数据，每次收到数据后，IP 不变，但是 MAC 地址会变化
- 光猫：光猫是一种类似于基带 modem（数字调制解调器）的设备，和基带 modem 不同的是接入的是光纤专线，是光信号。 光电收发器是用局域网中光电信号的转换，而仅仅是信号转换，没有接口协议的转换。
- DNS：用来解析出 IP（类似电话簿），通过网址解析出对应的 IP
- HTTP 服务器：提供浏览器能够访问到的数据

> 以上名词解释引用自百度百科

4.1.6 描述一下请求一个网页的步骤（浏览器访问服务器的步骤）?

具体步骤：

1. 解析域名：浏览器输入网址，通过 DNS 服务器解析域名，浏览器得到 Web 服务器的 IP 地址
2. 客户端连接到 Web 服务器：客户端，我们通常指的是浏览器，与 Web 服务器的 HTTP 端口（默认情况下 80）建立一个 TCP 套接字的连接。
3. 发送 HTTP 请求：通过 TCP 套接字，浏览器向 Web 服务器发送一个请求报文，一个请求报文包括请求行、请求头、空行、请求体四部分。
4. 服务器端解析请求并返回 HTTP 响应：Web 服务器解析请求，定位资源，返回响应报文。
5. 释放 TCP 连接：Web 浏览器主动关闭 TCP 套接字，关闭 TCP 连接。
6. 客户端浏览器解析响应报文，解析成 HTML 内容，经过浏览器的渲染之后就能展示给人们了。

4.1.7 HTTP 与 HTTPS 协议有什么区别?

HTTP 和 HTTPS 的基本概念：

- HTTP：超文本传输协议 HyperText Transfer Protocol 的缩写是 Http，它是目前使用最广泛的一种网络协议。发送信息是以明文方式进行的，存在安全隐患。
- HTTPS：HTTPS 是 HTTP 的安全升级版，是为了安全而设计的。传输数据基于 SSL 协议，SSL 协议可以提供数据加密，保证了数据传输时候的安全性。
- HTTP 与 HTTPS 的区别：
- HTTPS 协议需要申请 CA 证书，证书使用需要一定的费用。
- HTTP 是明文传输，HTTPS 是加密传输。
- HTTP 和 HTTPS 使用的端口不同，前者是 80，后者是 443。
- HTTP 无状态连接，HTTPS 是 SSL+HTTP 的安全连接。

4.1.8 TCP 中的三次握手和四次挥手是什么?

TCP 数据传输之前，服务器端和客户端需要建立连接，建立和断开连接的过程就是三次握手和四次挥手。

对象：TCP 客户端与服务器端

目的（礼尚往来）：

- 3 次握手：建立连接（占用资源）
- 4 次挥手：断开连接（释放资源）

具体步骤：

3 次握手：

- 客户端的 connect 方法默认是阻塞的，3 次握手成功，连接建立后才会解除阻塞
- 第 1 次：客户端发送数据给服务器端，客户端要连接服务器，让服务器准备好资源
- 第 2 次：服务器发送数据给客户端，告诉客户端我已经准备好了资源，同时让客户端也准备好资源
- 第 3 次：客户端确认也准备好了，双方都已经准备好资源，创建连接

4 次挥手：

- 第 1 次：客户端关闭发送数据，客户端发送消息给服务器，我要关闭发送了
- 第 2 次：服务器发送数据给客户端，我已收到数据，我要关闭接收数据了
- 第 3 次：服务器发送数据给客户端，我要关闭发送数据了
- 第 4 次：客户端发送数据给服务器，我已接收到数据，我也关闭接收数据

4.1.9 TCP 短连接和长连接的优缺点？各自的应用场景？

TCP 短连接和长连接的优缺点：

- 对于频繁的请求操作，保持长连接，可以减少资源的浪费，但是如果一直保持连接，有大量的客户端都保持长连接，最终会造成服务器压力过大。
- 短连接建立和断开连接简单，但是客户端频繁的请求连接，也会占用时间、宽带，造成资源的浪费。
- 具体使用长连接还是短连接，要看具体的场景。

TCP 长/短连接的应用场景：

- 对于数据库的操作，一般采用长连接，因为需要频繁多次的请求数据
- 对于网站服务器，一般都采用短连接，大量用户访问，长连接相对于短连接会耗费大量的资源。所以并发量大，但每个用户无需频繁操作情况下需用短链接好。

4.1.10 TCP 第四次挥手为什么要等待 2MSL？

MSL 是 Maximum Segment Lifetime 英文的缩写，中文可以译为“报文最大生存时间”，他是任何报文在网络上存在的最长时间，超过这个时间报文将被丢弃。

TCP 第四次挥手为什么要等待 2MSL，这最主要是因为两个理由：

- 为了保证客户端发送的最后一个 ACK 报文段能够到达服务器。因为这个 ACK 有可能丢失，从而导致处在 LAST-ACK 状态的服务器收不到对 FIN-ACK 的确认报文。服务器会超时重传这个 FIN-ACK，接着客户端再重传一次确认，重新启动时间等待计时器。最后客户端和服务器都能正常的关闭。假设客户端不等待 2MSL，而是在发送完 ACK 之后直接释放关闭，一但这个 ACK 丢失的话，服务器就无法正常的进入关闭连接状态。
- 它还可以防止已失效的报文段。客户端在发送最后一个 ACK 之后，再经过经过 2MSL，就可以使本链接持续时间内所产生的所有报文段都从网络中消失。从保证在关闭连接后不会有还在网络中滞留的报文段去骚扰服务器。

注意：在服务器发送了 FIN-ACK 之后，会立即启动超时重传计时器。客户端在发送最后一个 ACK 之后会立即启动时间等待计时器。

> 引用来源（CSDN 博客）：https://blog.csdn.net/xiaofei0859/article/details/6044694/

4.1.11 HTTP 最常见的请求方法有哪些？

- GET：GET 请求的参数都会放在 URL 地址里面，然后发送给服务器，服务器返回信息。根据浏览器的不同，发送的参数大小有限制，一般大小是在 2~8K 之间。对于一些隐私安全信息，使用 GET 请求就会不安全。
- POST：POST 请求一般用于向服务器端提交数据，请求的信息放在 FromData 里面。传输的数据大小取决于服务器端的设置。POST 请求的安全性高于 GET 请求。
- PUT：PUT 请求主要用来传输数据给服务器端，传输的数据用来替代指定的文档内容。PUT 请求的本质是 idempotent 的方法，一般情况下，PUT 和 POST 请求没有特别的区分，根据语义使用即可。
- DELETE：DELETE 请求是用来让服务器删除指定的页面，DELETE 请求一般会返回 3 种状态码：
  - 200（OK）：删除成功，返回已经删除的资源
  - 202（Accepted）：删除请求已经接受，但没有被立即执行
  - 204（No Content）：删除请求已经被执行，但是没有返回资源

4.1.12 GET 请求和 POST 请求有什么区别?

- GET 请求的请求参数一般放在 URL 地址中，POST 请求的参数放在 formdata
- GET 请求的安全性低于 POST 请求
- GET 请求在执行后退操作时候没有影响，但是 POST 执行后退操作会再次发出请求
- GET 请求生成的 URL 地址可以被标记，而 POST 请求的 URL 地址一般是固定不变的，不可以被标记
- GET 请求参数会保留在浏览器的历史记录里面，可以方便的查看请求信息，而 POST 请求不会保留参数
- GET 请求使用的是 URL 编码，POST 请求可以采用多种编码格式

4.1.13 cookie 和 session 的有什么区别?

- 数据存放位置不同：cookie 的数据是存储在客户端的浏览器里面；session 的数据存储在服务器端。
- 安全性不同：本地的 cookie 信息可以分析后进行修改，然后使用修改后的 cookie 发送请求，但是 session 是存储在服务器端，安全性高于 cookie。对于重要的安全信息使用 session 存储。
- 通常单个 cookie 的最大容量是 4K，并且 cookie 的存储个数有一定的限制，目前浏览器的上限一般是 20 个 cookie 值。
- 推荐使用：将登陆信息等重要信息存放为 SESSION；其他信息如果需要保留，可以放在 COOKIE 中

4.1.14 七层模型和五层模型是什么?

OSI（OSI（Open System Interconnect），即开放式系统互联。 一般都叫 OSI 参考模型，是 ISO（国际标准化组织）组织在 1985 年研究的网络互连模型 ），定义了网络互连的七层框架（物理层、数据链路层、网络层、传输层、会话层、表示层、应用层）。

TCP/IP 五层模型：

- 物理层：主要作用是定义物理设备如何传输数据（光缆，网线）
- 数据链路层：为通讯实体间建立数据链路连接
- 网络层：为数据在节点之间传输创建逻辑链路
- 传输层：数据的传输都是在这层定义的，数据过大分包，分片。
- 应用层 (相当于七层模型中的会话层、表示层、应用层)：为应用软件提供了很多服务，构建于协议之上。

> 引用来源（CSDN 博客）：https://blog.csdn.net/mestryas/article/details/81629218

4.1.15 HTTP 协议常见状态码及其含义?

状态码	含义
200	OK 客户端请求成功
301	Moved Permanently 请求永久重定向
200	OK 客户端请求成功
301	Moved Permanently 请求永久重定向
302	Moved Temporarily 请求临时重定向
304	Not Modified 文件未修改，可以直接使用缓存的文件。
400	Bad Request 客户端请求有语法错误，不能被服务器所理解
401	Unauthorized 请求未经授权，这个状态代码必须和 WWW-Authenticate 报头域一起使用
403	Forbidden 服务器收到请求，但是拒绝提供服务
404	Not Found 请求资源不存在，eg：输入了错误的 URL
500	Internal Server Error 服务器发生不可预期的错误
503	Server Unavailable 服务器当前不能处理客户端的请求，一段时间后可能恢复正常

4.1.16 HTTP 报文基本结构？列举常用的头部信息？

报文里面有四部分组成：状态行、头部信息（首部行）、空行、体（正文信息）

- 头（Head）和正文（Body）以一个空行隔开
- 头信息里面看似类似字典的键值，实际是一连串的字符串，以空格和换行隔开

常用的头部信息：

- Accept：接收的媒体类型。
- Accept-Charset：接收受的字符集。
- Accept-Encoding：接受的编码。
- Accept-Language：接收的语言类型。
- Connection：连接方式 。
- Host：请求的主机地址。
- User-Agent：用户代理 。
- Content-Encoding：返回内容的编码，如 gzip。
- Content-Language：返回内容的语言。
- Content-Length：返回内容的字节长度。
- Content-Type：返回内容的媒体类型，如 text/html。
- Expires：设置缓存过期时间，Cache-Control 也会相应变化。
- Last-Modified：最近修改时间，用于客户端缓存，与 If-Modified-Since 配合使用。
- Server：服务器端的信息。

4.1.17 SEO 是什么？

- SEO（Search Engine Optimization）：翻译过来就是搜索引擎优化。
- SEO 的目的是为了提高网站在搜索引擎结果中的排名，比如你百度搜索一个内容，一般只会看第一页，调到第二页的几率都会很小，因此网站设计时候，就需要考虑怎么提高自己网站内容的排名，就需要根据搜索引擎的规则进行相关的优化。
- SEO 的原理是不断的分析各种搜索引擎的排名规律，然后根据这个规律对自己的网站进行优化，从而不断提高网站的排名，最终目的是为了提高网站的访问量，吸引更多的用户。

4.1.18 伪静态 URL、静态 URL 和动态 URL 的区别?

- 静态 URL：网站访问速度非常快，因为资源已经准备好了，只等待用户去读取即可；网站名称具有一定的规律，方便用户记忆。对于 SEO 搜索有利。
- 动态 URL：用户访问网址后，服务器需要根据用户的请求返回数据，速度相对于静态请求会稍慢，但是随着现在服务器能力不断的强大，速度方面的差异已经明显的差异，对于用户体验没有区别。
- 伪静态 URL：实际上 URL 地址看似是静态 URL，本质上还是一个动态 URL 地址，经过伪装后，网址方便记忆，但是需要额外的设置，同时访问速度并没有加快，对于 SEO 搜索也没有明显影响。网站开发者可以根据需要自行选择使用哪种 URL。

4.1.19 浏览器镜头请求和动态请求过程的区别?

- 静态请求：浏览器发送请求，服务器从存储硬盘里面读取已有数据，返回数据给浏览器
- 静态请求：浏览器发送请求，服务器调用 Web 框架，获取得到数据，然后返回数据给浏览器

4.1.20 WSGI 接口有什么好处?

- Web Server Gateway Interface：简称 WSGI
- 使用 WSGI 接口，我们可以将 Web 服务器端和处理请求的 Web 框架进行分开，针对不同的需求我们可以开发不同的 Web 服务器和 Web 框架，然后自己进行组合使用。

4.1.21 简单描述浏览器通过 WSGI 接口请求动态资源的过程?

- 服务器端的 Nginx 一直处于监听状态，当客户端的浏览器发来请求后，Nginx 根据请求信息进行分析，对于静态请求分发到静态资源返回数据，对于动态的请求分发到相应的处理端口，计算分析后返回数据。
- 程序的 WSGI 会一直监听某个端口，当监听到数据后，然后请求的数据封装成一个字典对象，然后同时提供一个 start_response 方法用于处理浏览器请求，返回相应的数据。
- 字典对象和 start_response 方法会作为参数，传递给一个实例，该实例一般有 wsgi_app(environ, start*response) 或者实现了* ***call****(self, environ, start*response) 方法。
- 上述的实例通过调用 start_response 方法返回 WSGI 的中间件，中间件再返回给 Nginx，数据最后返回给客户端的浏览器。

5. 数据库

5.1 MySQL

5.1.1 NoSQL 和 SQL 数据库的比较?

- NoSQL：无关系型的数据库（not only sql），NoSQL 数据库不支持 SQL 语法，存储的数据都是以键值对的形式。不同的 NoSQL 数据库，使用的语言类似，但是并不相同。每种 NoSQL 数据库都有自己的优势和应用场景。
- NoSQL 常用的数据库种类：Mongodb、Redis、HBase Hadoop、Cassandra Hadoop

NoSQL 和 SQL 数据库的比较：

- 应用场景不同：SQL 数据库用于关系特别复杂的数据库，NoSQL 一般用于没有明显关系的数据存储

- “事务”特性的支持：SQL 对事务的支持非常完善，而 NoSQL 基本不支持事务

两者在不断地取长补短，呈现融合趋势。

5.1.2 了解 MySQL 的事物吗？事物的四大特性是什么？

事务是一个操作队列，队列中所有命令要不全部执行成功，要不一个也不执行，它是一个整体，要么执行要么不执行，不能只执行其中一个命令。事物我们可以对比生活中的银行转账，一个银行账号转账到另外一个账号，中间会有很多操作，但是最终的结果要么是钱转到对方账号，要么是钱未转出，只可能是这两种结果。

事务四大特性（ACID）

- 原子性（Atomicity）：要么全部被执行成功，要么就全部不被执行
- 一致性（Consistency）：事务执行以后，数据库从一种状态转换为另一种状态。
- 隔离性（Isolation）：事务在成功之前，处于隔离状态，事务不会受到外界的影响。
- 持久性（Durability）：事务成功之后，状态就永久保存，数据的状态不会发生改变，事务提交后出现其它错误，事务的处理结果也会得到保存。

5.1.3 关系型数据库的三范式是什么？

在开发人员长期的使用和总结下，积累了大量的经验规范，这些规范用于数据库设计，这些总结的规范被称为范式，一般使用的是三范式：

- 第一范式（1NF）：列具有原子性，一列不能再分成几列
- 第二范式（2NF）：表必须具有主键，如果有列没有包含在主键之中，那么这些列必须依赖于主键，并且不能只依赖于主键的一部分。
- 第三范式（3NF）：非主键列必须直接依赖于主键，不能使用传递依赖。比如以下间接依赖不被允许：非主键列 A 依赖于非主键列 B，非主键列 B 依赖于主键。

5.1.4 关系型数据库的核心元素是什么？

- 数据行（记录具体的信息）
- 数据列（字段，类似于表格的表头）
- 数据表（数据行的集合）
- 数据库（数据表的集合）

5.1.5 简单描述一下 Python 访问 MySQL 的步骤？

- 创建 connection 连接，连接数据库
- 获取 cursor 游标对象，该对象用于执行 SQL 语句
- 查询操作数据库
- 关闭 cursor 游标对
- 关闭 connection 连接

5.1.6 写一个 Python 连接操作 MySQL 数据库实例？

```
'''
Author: Felix
WX: AXiaShuBai
Email: xiashubai@gmail.com
Blog: https://blog.csdn.net/u011318077
Date: 2019/12/9 11:17
Desc:
'''
```

```python
# -*- coding:utf-8 -*-
# date: 2019-05-30

# 使用pymysql连接数据库
# 对数据库进行增删改

from pymysql import *

def main():
    # 创建Connection连接
    conn = connect(
        host='localhost', port=3306, database='tao_bao',
        user='root', password='xxxxxxxx', charset='utf8'
    )
    # 获得Cursor游标对象，该对象用于执行sql语句
    cs1 = conn.cursor()

    # 执行sql语句，读取出sql语句选择的table中的数据
    # 返回结果是数据的行数
    print(cs1.execute('select * from goods;'))
    print('*' * 100)

    # 取出的数据是以元组的形式
    # 取出1行数据,取过的就不会再取
    print(cs1.fetchone())
    print('*' * 100)
    print(cs1.fetchone())
    print('*' * 100)

    # 取出多行数据,传入取出的行数
    print(cs1.fetchmany(3))
    print('*' * 100)

    # 取出所有的数据
    print(cs1.fetchall())
    print('*' * 100)

    # 关闭Cursor对象
    cs1.close()
    # 关闭Connection对象
    conn.close()
    print("MySQL数据库连接已关闭")

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

5.1.7 SQL 语句主要有哪些？分别有什么作用？

- DQL：数据查询语言，查询数据，如 select
- DML：数据操作语言，增加、修改、删除数据，如 insert、udpate、delete
- TPL：事务处理语言，处理事务，包括 begin transaction、commit、rollback
- DCL：数据控制语言，进行授权与权限回收，如 grant、revoke
- DDL：数据定义语言，管理数据库和数据表，如 create、drop
- CCL：指针控制语言，通过指针对表进行操作，如 declare cursor

5.1.8 MySQL 有哪些常用的字段约束？

参考下表：

约束	名称	作用
primary key	主键	表的主键，具有唯一性
unique key	唯一约束	值唯一，不能出现重复
not null	非空约束	该字段必须有值，不能为空
default	默认值	字段默认值，字段没有填写时候直接使用默认值
auto_increment	自增长约束	主键编号的自动增长
comment	注释	表、列都可以设置注释 (comment)
foreign key	外键约束	字段进行外键约束

补充：unsigned 是 MYSQL 自定义的类型，非标准 SQL。用途是起到约束数值的作用，可以增加数值范围（相当于把负数那部分加到正数上）。

5.1.9 什么是视图？视图有什么作用？

- 视图是根据查询结果返回的一张虚拟的表，用于数据查询对比。
- 视图是一条 SELECT 语句被执行后返回的结果，返回的结果就是一张表，该表就是视图。
- 视图并会存储具体的数据，当数据库里面的数据发生变化以后，视图同时也会发生变化。

视图的作用：

- 视图可以简化用户的查询操作
- 视图可以方便的对比数据
- 视图为数据库提供了一定程度的逻辑独立性
- 视图能够对机密数据提供安全保护

5.1.10 什么是索引？索引的优缺点是什么？

- 索引：索引是数据表的引用指针，它是表空间的一个组成部分。类似于一本书的目录。
- 索引目的：增加数据库的查询速度，索引类似一个目录，查询数据时候直接查询索引，大大提高查询的效率。
- 索引的优缺点：提高数据库查询速度，创建数据时候，也需要创建对应的索引，降低了数据写入的速度，此外建立大量的索引，也会占用大量的磁盘空间。

5.1.11 NULL 是什么意思？它和空字符串一样吗？

NULL 这个值表示 UNKNOWN（未知）：它不表示“”（空字符串）。对 NULL 这个值的任何比较都会生产一个 NULL 值。您不能把任何值与一个 NULL 值进行比较，并在逻辑上希望获得一个答案。可以使用 IS NULL 来进行 NULL 判断。

5.1.12 主键、外键和索引的区别？

定义：

- 主键：具有唯一性，不能为空值
- 外键：该表的外键，一般是另外一个表的主键，可以重复和空值
- 索引：该字段值唯一，不能重复，可以为空值

作用：

- 主键：确保数据的完整性
- 外键：用来建立表与表之间的联系
- 索引：是提高数据查询的速度

个数：

- 主键：主键只能有一个
- 外键：一个表可以有多个外键
- 索引：一个表可以有多个唯一索引

5.1.13 char 和 varchar 的区别?

char 固定长度的数据类型，varchar 可变长度的数据类型，它们的区别是：

- char(M) 类型，M 是指定的数据的字节大小，每个值都占用 M 字节，当值小于 M 字节，右边使用空格补全，在 varchar(M) 类型的数据列里，每个值只占用刚好够用的字节再加上一个用来记录其长度的字节（即总长度为 L+1 字节）。
- varchar 使用场景：字符串长度不确定，一般长度大于平均长度，数据更新不频繁，使用多字节字符集存储字符串的地方。
- char 使用场景：一般用于存储固定长度的数据，比如身份证和手机号，一般用于存储长度较短的数据，避免出现存储碎片，获得更好的 IO 性能。

5.1.14 SQL 注入是什么？如何避免 SQL 注入?

SQL 注入：通过欺骗伪装的手段，把恶意的 SQL 命令插入到 Web 表单提交给服务器，作为请求的查询字符串，最终恶意请求获取到数据库的大量数据。

SQL 注入产生的原因：数据库设计时候，对于一些不规范的 SQL 语句未进行排除，同时未对一些特殊字符进行过滤，导致客户端可以通过全局变量 POST 和 GET 提交一些 SQL 语句正常执行。

参考以下代码，以下语法就会引起 SQL 注入：

```
def get_computer_by_ID(self):
    '''查询商品的信息'''
    find_ID = input("输入查询电脑的编号: ")
    sql = """select * from coumputers where ID='%s';""" % find_ID
    print("SQL语句: %s" % sql)
    self.execute_sql(sql)

# sql注入问题:
# 上面查询商品的sql语句如下:
# sql = """select * from coumputers where ID='%s';""" % find_ID
# ID后面是一个双单引号''
# 如果我们输入的电脑的ID是, 请输入要查询的商品的ID: ' or 1=1 or '
# 结果就是: select * from coumputers where ID='' or 1=1 or '';
# 由于使用的是or只要满足一个即可, 1=1满足, 这样就会查询出所有的商品
# 因此通过sql注入就可以获取到所有的数据了
```

SQL 注入的防止措施：

- 配置文件 magic_quotes_gpc 和 magic_quotes_runtime 进行相应的设置
- SQL 语句书写规范，不要省略双引号和单引号
- 可以使用 addslashes 对 SQL 语句进行转换
- SQL 语句中的不要出现数据库操作关键词：update、delete、select
- SQL 语句中不要使用 `*` 等特殊符号
- 优化数据库各种字段命名，不容易被猜到进行攻击
- 在浏览器上不要显示错误信息，将错误信息写到服务器的日志文件

5.1.15 存储引擎 MyISAM 和 InnoDB 有什么区别?

- InnoDB 支持事务，MyISAM 不支持。
- MyISAM 适合查询以及插入为主的应用，InnoDB 适合频繁修改以及涉及到安全性较高的应用。
- InnoDB 支持外键，MyISAM 不支持。
- 从 MySQL 5.5.5 以后，InnoDB 是默认引擎。
- MyISAM 支持全文类型索引，而 InnoDB 不支持全文索引。
- InnoDB 中不保存表的总行数， `select count(*) from table` 时，InnoDB 需要扫描整个表计算有多少行，但 MyISAM 只需简单读出保存好的总行数即可。注：当 count(*) 语句包含 where 条件时 MyISAM 也需扫描整个表。
- 对于自增长的字段，InnoDB 中必须包含只有该字段的索引，但是在 MyISAM 表中可以和其他字段一起建立联合索引。
- 清空整个表时，InnoDB 是一行一行的删除，效率非常慢。MyISAM 则会重建表。MyISAM 使用 delete 语句删除后并不会立刻清理磁盘空间，需要定时清理，命令： `OPTIMIZE table dept;`
- InnoDB 支持行锁（某些情况下还是锁整表，如 `update table set a=1 where user like '%lee%'` ）
- MyISAM 创建表生成三个文件：.frm 数据表结构、.myd 数据文件、.myi 索引文件，Innodb 只生成一个 .frm 文件，数据存放在 ibdata1.log

应用场景：

- 现在一般都选用 InnoDB，主要是 MyISAM 的全表锁，读写串行问题，并发效率锁表，效率低，MyISAM 对于读写密集型应用一般是不会去选用的。
- MyISAM 不支持事务处理等高级功能，但它提供高速存储和检索，以及全文搜索能力。如果应用中需要执行大量的 SELECT 查询，那么 MyISAM 是更好的选择。
- InnoDB 用于需要事务处理的应用程序，包括 ACID 事务支持。如果应用中需要执行大量的 INSERT 或 UPDATE 操作，则应该使用 InnoDB，这样可以提高多用户并发操作的性能。

> 引用来源（CSDN 博客）：https://blog.csdn.net/qq_35642036/article/details/82820178

5.1.16 MySQL 中有哪些锁?

MyISAM 支持表锁，InnoDB 支持表锁和行锁，默认为行锁。

- 表级锁：锁占用资源少，加锁速度快，不会出现死锁。锁力度较大，冲突发生的概率最高，并发了较低
- 行级锁：锁占用资源大，加锁速度慢，会出现死锁。锁力度小，冲突发生的概率小，并发度较高

5.1.17 三种删除操作 drop、truncate、delete 的区别?

- drop（删除表）：删除内容和定义，释放空间。简单来说就是把整个表去掉。以后要新增数据是不可能的，除非新增一个表。drop 语句将删除表的结构被依赖的约束（constrain），触发器（trigger）索引（index）；依赖于该表的存储过程/函数将被保留，但其状态会变为：invalid。
- truncate（清空表中的数据）：删除内容、释放空间但不删除定义（保留表的数据结构）。与 drop 不同的是，只是清空表数据而已。truncate table 删除表中的所有行，但表结构及其列、约束、索引等保持不变。新行标识所用的计数值重置为该列的种子。如果想保留标识计数值，请改用

delete。注意：truncate 不能删除行数据，要删就要把表清空。
- delete（删除表中的数据）：delete 语句用于删除表中的行。delete 语句执行删除的过程是每次从表中删除一行，并且同时将该行的删除操作作为事务记录在日志中保存。

执行速度，一般来说：drop> truncate > delete。

补充：delete 语句是数据库操作语言（dml），这个操作会放到 rollback segement 中，事务提交之后才生效；如果有相应的 trigger，执行的时候将被触发。truncate、drop 是数据库定义语言（ddl），操作立即生效，原数据不放到 rollback segment 中，不能回滚，操作不触发 trigger。

应用场景：

- 当你不再需要该表时， 用 drop
- 当你仍要保留该表，但要删除所有记录时， 用 truncate
- 当你要删除部分记录时（always with a where clause），用 delete

> 引用来源：https://www.cnblogs.com/fjl0418/p/7929420.html

5.1.18 MySQL 中的存储过程是什么？有什么优点？

MySQL 存储过程就是执行一定功能的 SQL 语句的集合。通常我们执行 SQL 语句时候，都需要先写 SQL 语句，然后经过编译，最后才执行。存储过程就是将一系列的具有特定功能的 SQL 语句编译后存储在数据库，我们要使用该语句时候，直接调用存储的名称即可。

MySQL 存储过程类似面向对象编程，将 SQL 语句转换成了一个对象一样。可以提高数据库操作的便捷性和提高 SQL 语句的执行速度，MySQL 也是一种安全机制，可以提高数据库的安全性。

5.1.19 MySQL 数据库的有哪些种类的索引？

- index：普通索引，数据没有任何限制，并且可以重复
- unique：数据唯一，允许有空值
- primary key：主键索引，具有唯一性，一个表只有一个主键，主键不能是空值
- 组合索引：使用多个字段创建索引
- fulltext：全文索引，fulltext 类似一个搜索引擎，配合 match against 语句使用。

5.1.20 MySQL 的事务隔离级别？

- 未提交读（Read Uncommitted）：数据修改了，还未提交，但是也可以读取到数据。
- 提交读（Read Committed）：数据修改后，成功提交以后才能读取数据。
- 可重复读（Repeated Read）：事物中读取到数据不受其它事物的影响，就算其它事物已经修改并成功提交，也不会影响当前事物的读取。
- 串行读（Serializable）：读取数据时候先要获取到表级共享锁，读写操作会相互阻塞
- MySQL 数据库（InnoDB 引擎）：默认使用可重复读

5.1.21 MySQL 中的锁如何进行优化？

- 读写分离
- 分段加锁
- 减少锁持有的时间
- 多个线程尽量以相同的顺序去获取资源

以上都不是绝对原则，都要根据情况，比如不能将锁的粒度过于细化，不然可能会出现线程的加锁和释放次数过多，反而效率不如一次加一把大锁。

5.1.22 解释 MySQL 外连接、内连接与自连接的区别？

- 交叉连接: 交叉连接即笛卡尔积，不需要使用任何条件，一个表和另一个表的所有记录一一匹配比

对。类似于集合取交集。
- 内连接：内连接即取交集，指定特定的条件，查询返回符合条件的结果，不符合条件的直接被排除掉。内连接只连接匹配的行。
- 外连接：查询返回的结果，包含符合条件的数据，也包含不符合条件的数据。但是有一个表是主表，左外连接，左边的表是主表，所有表数据都会显示，左边表有的数据右表没有的，则会显示空值。同理，右外连接，以右边的表作为主表，左表没有数据的，显示为空值。

5.1.23 如何进行 SQL 优化?

选择正确的存储引擎：

- 以 MySQL 为例，包括有两个存储引擎 MyISAM 和 InnoDB，每个引擎都有利有弊。
- MyISAM 适合于一些需要大量查询的应用，但其对于有大量写操作并不是很好。甚至你只是需要 update 一个字段，整个表都会被锁起来，而别的进程，就算是读进程都无法操作直到读操作完成。另外，MyISAM 对于 SELECT COUNT(*) 这类的计算是超快无比的。

优化字段的数据类型：

- 记住一个原则，越小的列会越快。如果一个表只会有几列罢了（比如说字典表，配置表），那么，我们就没有理由使用 INT 来做主键，使用 MEDIUMINT, SMALLINT 或是更小的 TINYINT 会更经济一些。如果你不需要记录时间，使用 DATE 要比 DATETIME 好得多。当然，你也需要留够足够的扩展空间。

为搜索字段添加索引：

- 索引并不一定就是给主键或是唯一的字段。如果在你的表中，有某个字段你总要会经常用来做搜索，那么最好是为其建立索引，除非你要搜索的字段是大的文本字段，那应该建立全文索引。

避免使用 Select * 读取数据：

- 从数据库里读出越多的数据，那么查询就会变得越慢。并且，如果你的数据库服务器和 Web 服务器是两台独立的服务器的话，这还会增加网络传输的负载。即使你要查询数据表的所有字段，也尽量不要用 `*` 通配符，善用内置提供的字段排除定义也许能给带来更多的便利。

使用 ENUM 而不是 VARCHAR：

- ENUM 类型是非常快和紧凑的。在实际上，其保存的是 TINYINT，但其外表上显示为字符串。这样一来，用这个字段来做一些选项列表变得相当的完美。例如，性别、民族、部门和状态之类的这些字段的取值是有限而且固定的，那么，你应该使用 ENUM 而不是 VARCHAR。

尽可能的使用 NOT NULL：

- 除非你有一个很特别的原因去使用 NULL 值，你应该总是让你的字段保持 NOT NULL。 NULL 其实需要额外的空间，并且，在你进行比较的时候，你的程序会更复杂。 当然，这里并不是说你就不能使用 NULL 了，现实情况是很复杂的，依然会有些情况下，你需要使用 NULL 值。

固定长度的表会更快：

- 如果表中的所有字段都是“固定长度”的，整个表会被认为是“static”或“fixed-length”。 例如，表中没有如下类型的字段： VARCHAR，TEXT，BLOB。只要你包括了其中一个这些字段，那么这个表就不是“固定长度静态表”了，这样，MySQL 引擎会用另一种方法来处理。
- 固定长度的表会提高性能，因为 MySQL 搜寻得会更快一些，因为这些固定的长度是很容易计算下一个数据的偏移量的，所以读取的自然也会很快。而如果字段不是定长的，那么，每一次要找下一条的话，需要程序找到主键。
- 并且，固定长度的表也更容易被缓存和重建。不过，唯一的副作用是，固定长度的字段会浪费一些空间，因为定长的字段无论你用不用，他都是要分配那么多的空间。

> 部分内容引用来源（CSDN 博客）：https://blog.csdn.net/fromatozhappy/article/details/86515671

5.1.24 什么是 MySQL 主从？主从同步有什么好处？

MySQL 主从同步：MySQL 有多个服务器去，一个服务器作为主服务器，其它服务器作为从服务器，通过配置我们可以将主服务器的所有数据或者部分数据复制到从服务器上。复制的过程是异步进行的。

主从同步的好处：（读写分离，数据备份，负载均衡）我们可以设置主服务器写入数据，读取数据使用从服务器。保证数据库负载均衡。数据随时都在进行同步备份，保证了数据的安全性。

5.1.25 MySQL 主从与 MongoDB 副本集有什么区别？

- MongoDB 是在同一个服务器（一台机器），同时复制备份多个数据库
- MySQL 复制备份数据是在多个服务器（多台机器），复制备份多个数据库
- MongoDB 复制备份是同步的，MySQL 复制备份可以是异步的，也可以主从同步
- MongoDB 读写在指定的某一个数据库作为主数据库
- MySQL 一般主服务器写，从服务器读，读写分离

5.1.26 MySQL 账户权限怎么分类的？

MySQL 账户体系：根据账户所具有的权限的不同，MySQL 的账户可以分为以下几种

- 服务实例级账号：启动了一个 mysqld，即为一个数据库实例；如果某用户如 root，拥有服务实例级分配的权限，那么该账号就可以删除所有的数据库、连同这些库中的表
- 数据库级别账号：对特定数据库执行增删改查的所有操作
- 数据表级别账号：对特定表执行增删改查等所有操作
- 字段级别的权限：对某些表的特定字段进行操作
- 存储程序级别的账号：对存储程序进行增删改查的操作

注意：进行账户操作时，需要使用 root 账户登录，这个账户拥有最高的实例级权限，账户的操作主要包括创建账户、删除账户、修改密码、授权权限等。

> 引用来源：https://www.cnblogs.com/denix-32/p/10066036.html

5.1.27 如何使用 Python 面向对象操作 MySQL 数据库？

面向对象操作 MySQL 数据库，就是将 MySQL 的增删改查方法封装成一个个函数，操作数据库直接调用函数即可，参考下面查询数据库案例。

```
'''
Author: Felix
WX: AXiaShuBai
Email: xiashubai@gmail.com
Blog: https://blog.csdn.net/u011318077
Date: 2019/12/9 11:17
Desc:
'''

# 使用pymysql连接数据库
# 条件查询

from pymysql import *

class JD(object):
    def __init__(self):
        # 创建Connection连接
```

```
        self.conn = connect(
            host='localhost', port=3306, database='jing_dong',
            user='root', password='xxxxxxxx', charset='utf8'
        )
        # 获得Cursor对象，该对象用于执行sql语句
        self.cursor = self.conn.cursor()

    def __del__(self):
        # 关闭Cursor对象
        self.cursor.close()
        # 关闭Connection对象
        self.conn.close()
        print("-----查询完成,数据库连接已关闭-----")

    def execute_sql(self, sql):
        # 执行sql语句的方法
        # 执行sql语句,读取出sql语句选择的table中的数据
        self.cursor.execute(sql)
        # 取出上面已经查询到的所有数据，因为结果一个元组嵌套的元组，我们逐行打印显示出来
        for temp in self.cursor.fetchall():
            print(temp)

    def show_all_items(self):
        '''显示所有的商品'''
        sql = "select * from goods;"
        self.execute_sql(sql)

    def show_cates(self):
        '''显示所有商品分类'''
        sql = "select name from goods_cates;"
        self.execute_sql(sql)

    def show_brands(self):
        '''显示所有商品分类'''
        sql = "select name from goods_brands;"
        self.execute_sql(sql)

    # 静态方法，类内部调用，不需要传入参数
    @staticmethod
    def print_menu():
        print("--------京东--------")
        print("1: 所有的商品")
        print("2: 所有的商品分类")
        print("3: 所有的商品品牌分类")
        return input("请输入功能所对应的序号(1或着2或者3): ")

    def run(self):
        while True:
            num = self.print_menu()
            if num == "1":
                # 查询所有的商品
                self.show_all_items()
            elif num == "2":
                # 查询所有的商品分类
                self.show_cates()
```

```
            elif num == "3":
                # 查询所有的商品品牌分类
                self.show_brands()
            else:
                print("输入序号有误，请重新输入")

def main():
    # 1.创建一个京东商城对象
    jd = JD()

    # 2.调用这个对象的run方法，让其运行
    jd.run()

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

> 代码引用来源（CSDN 博客）：https://blog.csdn.net/qq_40667484/article/details/87903707

5.2 Redis

5.2.1 Redis 是什么？常见的应用场景？

Redis 是一个非关系的数据库，存储数据的方式是键值对，所有数据存储在内存之中，因此存写速度非常快，缓存最常用的就是 Redis 数据库。

Redis 应用场景：

- Session 共享（单点登录）
- 页面缓存
- 队列
- 排行榜/计数器
- 发布/订阅

5.2.2 Redis 常见数据类型有哪些？各自有什么应用场景？

- 常见的数据类型一共 5 种：String、Hash、List、Set、Sorted set
- String：最常用的数据类型，用于网站计数，排名，粉丝数量等
- Hash：field 和 value 的一个映射表，一般用于存储对象，常用于存储各种信息
- List：类似于列表，和列表一样支持查找和遍历，常用于消息列表，网站的分页也可以使用
- Set：set 功能和 List 类似，但是它可以排除重复的数据，并且和集合一样，可以进行交集、并集、差集的操作。比如可以用于 QQ 好友的 DNA 分析，是否有共同好友等等。
- Sorted set：有序集合，常用于网站的排行榜。

5.2.3 非关系型数据库 Redis 和 MongoDB 数据库的结构有什么区别？

Redis 数据库最基本单元就是键值对的字典，一个数据库中可以独立存在多个字典，打开 Redis 数据库，默认有 16 个数据库，从 0 开始编号，默认连接的也是 0 号数据库：

- Redis 服务—>数据库（默认 16 个）—> 字典
- 创建键值数据（字典形式），字典都存放在数据库中
- Redis 默认支持 16 个数据库，设置配置文件可以支持更多，并且目前没有上限限制

MongoDB 数据库最基本的单元是文档 document，一个文档就类似于一个字典，一组文档组成一个集合 collection，一系列集合组成一个数据库，多个数据库就是一个 MongoDB 实例（MongoDB 实例下面可以创建多个数据库）。

数据库结构：MongoDB 实例—> 数据库 database（可创建多个）—> 集合 collection–>- 文档 document

5.2.4 Redis 和 MongoDB 数据库的键（key）和值（value）的区别?

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

MongoDB 文档（字典格式）的值可以是字符串、整数、数组，以及文档（字典）等类型。

文档的特性（键和值的特性）：

1. 文档的键值对是有序的
2. 文档的值可以是字符串，整数，数组，以及文档等类型
3. 文档的值区分大小写及值的类型
4. 文档中的值不仅可以是在双引号里面的字符串，还可以是其他几种数据类型（甚至可以是整个嵌入的文档）
5. 文档不能有重复的键
6. 文档的键是字符串

5.2.5 Redis 持久化机制是什么？有哪几种方式?

Redis 是一个支持持久化的内存数据库，通过持久化机制把内存中的数据同步到硬盘文件来保证数据持久化。当 Redis 重启后通过把硬盘文件重新加载到内存，就能达到恢复数据的目的。

实现过程：单独创建 fork() 一个子进程，将当前父进程的数据库数据复制到子进程的内存中，然后由子进程写入到临时文件中，持久化的过程结束了，再用这个临时文件替换上次的快照文件，然后子进程退出，内存释放。

Reids 支持两种不同的持久化操作。Redis 的一种持久化方式叫快照（snapshotting，RDB），另一种方式是只追加文件（append-only file，AOF）。

快照（snapshotting）持久化（RDB）：Redis 可以通过创建快照来获得存储在内存里面的数据在某个时间点上的副本。Redis 创建快照之后，可以对快照进行备份，可以将快照复制到其他服务器从而创建具有相同数据的服务器副本（Redis 主从结构，主要用来提高 Redis 性能），还可以将快照留在原地以便重启服务器的时候使用。快照持久化是 Redis 默认采用的持久化方式，在 redis.conf 配置文件中。

```
save 900 1           # 在900秒(15分钟)之后，如果至少有1个key发生变化，Redis就会自动触发
BGSAVE命令创建快照。
save 300 10          # 在300秒(5分钟)之后，如果至少有10个key发生变化，Redis就会自动触发
BGSAVE命令创建快照。
save 60 10000        # 在60秒(1分钟)之后，如果至少有10000个key发生变化，Redis就会自动触发
BGSAVE命令创建快照。
```

AOF（append-only file）持久化：与快照持久化相比，AOF 持久化 的实时性更好，因此已成为主流的持久化方案。默认情况下 Redis 没有开启 AOF（append only file）方式的持久化，可以通过 appendonly 参数开启. 开启 AOF 持久化后每执行一条会更改 Redis 中的数据的命令，Redis 就会将该命令写入硬盘中的 AOF 文件。AOF 文件的保存位置和 RDB 文件的位置相同，都是通过 dir 参数设置的，默认的文件名是 appendonly.aof。

在 Redis 的配置文件中存在三种不同的 AOF 持久化方式：

```
appendonly yes

appendfsync always    # 每次有数据修改发生时都会写入AOF文件,这样会严重降低Redis的速度
appendfsync everysec  # 每秒钟同步一次，显示地将多个写命令同步到硬盘
appendfsync no        # 让操作系统决定何时进行同步
```

> 部分内容引用来源（CSDN 博客）： https://blog.csdn.net/hguisu/article/details/90748916

5.2.6 Redis 的事务是什么?

NoSQL 数据库对事务支持性能不太完善，但是 Redis 支持原子操作和隔离操作。

Redis 通过 MULTI、EXEC、WATCH 等一系列的命令来实现事务（transaction）功能。

Redis 事务是一些列 Redis 命令的集合：它是一个单独的隔离操作，事物中的命令队列都会按顺序全部执行，不会被外来事物干扰；事物的执行具有原子性，事务中的命令要么全部执行成功，要么全部都不执行。

Redis 事物执行过程：开始事务、命令入队、执行事务。

5.2.7 为什么要使用 Redis 作为缓存?

Redis 数据都是存储在内存中，性能强大，它支持高性能和高并发。

- 高性能：传统数据库都是从硬盘读取数据，Redis 直接从缓存读取。
- 高并发：缓存的抗压能力是大于数据库的，请求直接从缓存进行而不用去数据库，大大提高的并发性。

5.2.8 Redis 和 Memcached 的区别?

Redis 支持的数据类型种类更多，支持 string、list、set、zset、hash 等数据类型。Memcached 只支持简单的数据类型 string。

Redis 支持数据持久化，内存中的数据可以存储到硬盘之中，但是 memecache 数据只支持存储在内存之中。

Redis 支持集群模式，Memcached 不支持。

5.2.9 Redis 如何设置过期时间和删除过期数据?

Redis 数据库里面存储数据的时候，就可以设置过期时间，对于一些用户数据，利用过期时间进行清理，防止数据越积越多，占用大量的空间。设置 key 的时候，可以通过 expire time 设定过期的时间，到期后就自动删除数据。

- 定期删除数据：Redis 里面有大量的数据，Redis 默认 100ms 随机检查数据过期时间是否已经过期，过期就将其删除。但是数据抽取是随机，如果数据库又大量的数据，有些数据过期并不会删除。此时需要使用惰性删除。
- 惰性删除数据 ：定期删除没有删除过期的数据，当我们查询这个数据时候，数据才会被删除，就是惰性删除。但是定期删除和惰性删除可能还是会遗漏数据，此时就要使用 Redis 的内存淘汰机制。

5.2.10 Redis 有哪几种数据淘汰策略?

Redisn 内存淘汰，我们首先需要设置 server.maxmemory，该参数表示可以使用最大的内存。当内存达到设置值后就开始淘汰数据。

内存淘汰策略种类：

1. volatile-lru: 从设置了过期时间的数据中，选择近期最少使用数据进行删除
2. volatile-ttr: 从设置了过期时间的数据中，选择即将过期的数据进行删除
3. volatile-random: 从设置了过期时间的数据中，中随机选择数据进行删除
4. allkeys-lru: 从所有数据中，选择最少使用的数据进行删除，
5. allkeys-random: 从所有数据中，随机选择数据删除
6. no-eviction：严禁删除数据，新的数据写入时候没有空间，就会报错

5.2.11 Redis 为什么是单线程的?

Redis 是基于内存的操作，CPU 不是 Redis 的瓶颈，Redis 的瓶颈最有可能是机器内存的大小或者网络带宽。

单线程容易实现，而且 CPU 不会成为瓶颈，就直接采用单线程的方案。

5.2.12 单线程的 Redis 为什么这么快?

- 纯内存操作
- 单线程操作，避免了频繁的上下文切换
- 采用了非阻塞 I/O 多路复用机制

5.2.13 缓存雪崩和缓存穿透是什么？如何预防解决?

缓存雪崩：可以理解为由于原有缓存失效，新缓存未到期间（例如：我们设置缓存时采用了相同的过期时间，在同一时刻出现大面积的缓存过期），所有原本应该访问缓存的请求都去查询数据库了，而对数据库 CPU 和内存造成巨大压力，严重的会造成数据库宕机。从而形成一系列连锁反应，造成整个系统崩溃。

缓存雪崩解决方法：

- 事前：尽量保证整个 Redis 集群的高可用性，发现机器宕机尽快补上。选择合适的内存淘汰策略。大多数系统设计者考虑用加锁（ 最多的解决方案）或者队列的方式保证来保证不会有大量的线程对数据库一次性进行读写，从而避免失效时大量的并发请求落到底层存储系统上。
- 事中：本地 ehcache 缓存 + Hystrix 限流 & 降级，避免 MySQL 崩掉。
- 事后：利用 Redis 持久化机制保存的数据尽快恢复缓存。

缓存穿透：缓存穿透是指用户查询数据，在数据库没有，自然在缓存中也不会有。这样就导致用户查询的时候，在缓存中找不到，每次都要去数据库再查询一遍，然后返回空（相当于进行了两次无用的查询）。这样请求就绕过缓存直接查数据库，这也是经常提的缓存命中率问题。一般是黑客故意去请求缓存中不存在的数据，导致所有的请求都落到数据库上，造成数据库短时间内承受大量请求而崩掉。

缓存穿透解决方法：

- 方法一：采用布隆过滤器，将所有可能存在的数据哈希到一个足够大的 bitmap 中，一个一定不存在的数据会被这个 bitmap 拦截掉，从而避免了对底层存储系统的查询压力。
- 方法二：简单粗暴的方法，如果一个查询返回的数据为空（不管是数据不存在，还是系统故障），我们仍然把这个空结果进行缓存，但它的过期时间会很短，最长不超过五分钟。通过这个直接设置的默认值存放到缓存，这样第二次到缓冲中获取就有值了，而不会继续访问数据库。

> 引用来源（CSDN 博客）：https://blog.csdn.net/xlgen157387/article/details/79530877

5.2.14 布隆过滤器是什么?

布隆过滤器通过引入 K 个相互独立的哈希函数，将元素进行哈希后，然后进行元素判重的过程。

相对于一般算法，具有极高的空间效率和极快的查询效率，缺点是有一定的误判率（极低）和删除困难。

Bloom-Filter 核心思想是多个哈希函数，降低误判率，只要有一个 hash 值不在集合之中，就可以判断元素步骤集合之中。布隆过滤器一般用于大量数据去重判断，比如爬虫去重。

5.2.15 简单描述一下什么是缓存预热、缓存更新和缓存降级？

- **缓存预热**：缓存预热一般针对新系统而言，比如上线某个网站，我们先将一些数据提前加载到缓存系统中，避免系统上线时候，大量用户到数据库请求数据，导致负载过高。
- **缓存更新**： 除了数据库自带的数据淘汰机制，开发人员也可以自定义淘汰策略，对数据库的缓存数据进行更新。最常用的定时去清理过期的缓存；有用户请求时候，对数据进行更新。
- **缓存降级**：当出现大量访问或者恶意访问时候，为了保证服务器核心功能有效，需要主动对一些缓存数据进行降级处理，从而保证核心数据不受影响。缓存降级是为了防止数据库雪崩。

5.2.16 如何解决 Redis 的并发竞争 Key 的问题？

- 大量请求同时取请求一个 key，导致最终的结果和我们预想的结果不同，就是并发竞争 Key。
- 针对并发竞争 key 的问题，实际开发使用分布式锁应对。推荐使用 ZooKeeper 和 Redis 来实现分布式锁。但是使用锁的缺点会降低 Redis 的性能。如果不存在竞争，最好不要使用。

5.2.17 写一个 Python 连接操作 Redis 数据库实例？

Redis 数据库操作可以使用 redis.Redis 或 redis.StrictRedis（推荐使用）。

```
'''
Author: Felix
WX: AXiaShuBai
Email: xiashubai@gmail.com
Blog: https://blog.csdn.net/u011318077
Date: 2019/12/9 11:17
Desc:
'''

# -*- coding: utf-8 -*-

import redis

# 使用redis.StrictRedis类操作redis数据库
# 使用命令与redis客户端的命令一致

def python_redis():
    # 创建一个StrictRedis对象，连接操作redis数据库
    # 创建连接池管理redis连接，可以避免每次建立和释放连接的开销
    # 指定主机和端口建立redis连接,默认使用的0数据库，可以不写
    try:
        pool = redis.ConnectionPool(host='127.0.0.1', port=6379, db=0)
        sr = redis.StrictRedis(connection_pool=pool)
        # 添加一个键值对，输出返回结果，添加成功True，添加失败False
        res = sr.set('name', 'python')
        print(res)
        # 取出键的值,如果不存在，返回None
        res = sr.get('name')
        print(res)
        # 修改键的值,键存在就直接修改其值
        res = sr.set('name', 'php')
        print(res)
        # 再次取出键的值
```

```
            res = sr.get('name')
            print(res)
            # 删除键值，成功则返回删除的个数，否则返回0
            res = sr.delete('name')
            print(res)
            # 获取所有的键,输出结果是一个所有键的列表，否则返回空lieb
            res = sr.keys()
            print(res)
        except Exception as e:
            print(e)

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

5.2.18 什么是分布式锁?

分布式锁是解决并发访问共享资源的一种方法。分布式锁具有互斥性，同一时间只能有一个访问客户拥有锁，锁超时以后，会自动释放锁，防止出现死锁。锁设计时候要支持非阻塞和阻塞。

分布式锁的实现方案

- 数据库实现（乐观锁）
- 基于 ZooKeeper 的实现
- 基于 Redis 的实现（推荐）

5.2.19 Python 如何实现一个 Redis 分布式锁?

Redis 分布式锁应该具备哪些条件：

- 在分布式系统环境下，一个方法在同一时间只能被一个机器的一个线程执行
- 高可用的获取锁与释放锁
- 高性能的获取锁与释放锁
- 具备可重入特性
- 具备锁失效机制，防止死锁
- 具备非阻塞锁特性，即没有获取到锁将直接返回获取锁失败

选用 Redis 实现分布式锁原因：

- Redis 有很高的性能
- Redis 命令对此支持较好，实现起来比较方便

使用命令介绍：

SETNX

- SETNX key val：当且仅当 key 不存在时，set 一个 key 为 val 的字符串，返回 1；若 key 存在，则什么都不做，返回 0。

expire

- expire key timeout：为 key 设置一个超时时间，单位为 second，超过这个时间锁会自动释放，避免死锁。

delete

- delete key：删除 key, 在使用 Redis 实现分布式锁的时候，主要就会使用到这三个命令。

Redis 分布式锁实现思想（Redis 实现分布式锁主要使用 Redis 单线程的对 key 原子操作特性）：

- 获取锁的时候，使用 setnx 加锁，并使用 expire 命令为锁添加一个超时时间，超过该时间则自动释放锁，锁的 value 值为一个随机生成的 UUID（Universally Unique Identifier，翻译为中文是通用唯一识别码，UUID 的目的是让分布式系统中的所有元素都能有唯一的识别信息），通过此值在释放锁的时候进行判断。
- 获取锁的时候还设置一个获取的超时时间，若超过这个时间则放弃获取锁。
- 释放锁的时候，通过 UUID 判断是不是该锁，若是该锁，则执行 delete 进行锁释放。

Redis 分布式锁实例：

```
# -*- coding:utf-8 -*-
# project_xxx\venv\Scripts python

'''
Author: Felix
Email: xiashubai@gmail.com
Blog: https://blog.csdn.net/u011318077
Date: 2019/12/4 20:58
Desc:
'''
import redis
import uuid
import time

class RedisLock():

    def __init__(self, lock_name, time_out, acquire_time):
        # 连接redis数据库
        self.redis_client = redis.StrictRedis(host='127.0.0.1', port=6379, db=0)
        self.lock_name = lock_name    # 锁的名称(锁的唯一ID)
        self.time_out = time_out  # 锁的超时时间(超过时间，自动释放锁)
        self.acquire_time = acquire_time  # 获取锁的时间(超过时间，放弃获取锁)

    def acquire_lock(self):
        '''获取一个分布式锁'''
        # 生成一个唯一的uuid值作为锁(键)的值(补充：UUID，Universally Unique
Identifier，翻译为中文是通用唯一识别码，UUID 的目的是让分布式系统中的所有元素都能有唯一的识别信息)
        identifier = str(uuid.uuid4())
        end = time.time() + self.acquire_time
        lock = 'string:lock:' + self.lock_name
        while time.time() < end:
            # 设置一个锁，设置锁的名称和唯一的UUID值
            if self.redis_client.setnx(lock, identifier):
                # 给锁设置超时时间，防止进程崩溃导致其它进程无法获取锁
                self.redis_client.expire(lock, self.time_out)
                return identifier
            elif not self.redis_client.ttl(lock): # ttl获取锁的生存时间
                self.redis_client.expire(lock, self.time_out)
            time.sleep(0.001)
        return False

    def release_lock(self, identifier):
        """通用的锁释放函数"""
```

```
        lock = "string:lock:" + self.lock_name
        pip = self.redis_client.pipeline(True)
        while True:
            try:
                pip.watch(lock)
                # 获取锁的值，即设置锁时的UUID值
                lock_value = self.redis_client.get(lock)
                if not lock_value:
                    return True

                if lock_value.decode() == identifier:
                    pip.multi()
                    pip.delete(lock)
                    pip.execute()
                    return True
                pip.unwatch()
                break
            except redis.exceptions.WatchError:
                pass
        return False

if __name__ == '__main__':
    redis_lock = RedisLock(lock_name='lock001', time_out=10, acquire_time=10)
```

> 部分内容引用来源（CSDN 博客）：https://blog.csdn.net/tuesdayma/article/details/82751790

5.2.20 如何保证缓存与数据库双写时的数据一致性?

- 可以给缓存设置有效时间
- 删除数据时候采用延时测量，确保请求完成后，再删除数据

5.2.21 集群是什么？Redis 有哪些集群方案?

Redis 集群是多个服务器组成的一个组。通常会把数据存在一个 master 节点，然后在这个 master 节点和对应的 salve 节点 之间进行数据同步。读取数据时候都是从 master 节点读取，salve 只用来同步数据，但这个 master 节点失效以后，会启动一个 slave 节点作为新的 master 节点。

Redis 集群方案：

- twemproxy：类似于 Redis 的一个代理方，本来需要连接 Redis 的地方连接 twemproxy
- codis：开发中使用最多的集群方案，和 twemproxy 类似。但是当节点数量改变后，旧节点数据可以自动恢复到新的节点上去。
- Redis cluster：Redis 数据库默认的集群，分布式算法使用的是 hash 槽，支持节点设置从节点。

5.2.22 Redis 常见性能问题和解决方案?

- 重要的数据，slave 节点开启 AOF 备份，备份时间可以设置每秒一次
- 主从节点最好设置在同一个局域网络中，可以保持网络连接稳定
- 主从节点数据复制时候推荐使用链表结构
- 主节点不要做持久化操作
- 具有较大压力的主节点上面就继续增加从节点

5.2.23 了解 Redis 的同步机制么?

- 主从同步，第一次同步时候，master 节点执行一次 bgsave，后续得到操作记录都采用内存 buffer 进行记录，操作完成后的 rdb 文件全部同步复制到 slave 节点

- slave 节点完成复制后，将 rdb 文件加载到内存，然后通知 master 节点，将复制期间的操作同步到 slave 节点，此时就完成了一次同步过程

5.2.24 如果有大量的 key 需要设置同一时间过期，一般需要注意什么？

- 大量的 key 的过期时间相同，可能导致刚刚过期的那个时间点，Redis 数据库操作没有反应
- 通常情况下，不建议设置大量相同的过期时间，可以通过设置一个随机值，对过期时间进行随机分散处理

5.2.25 如何使用 Redis 实现异步队列？

- 使用 list 队列，rpush 放入消息，lpop 取出消息，lpop 没有消息取出的时候，设置一个 sleep 时间，然后过一段时间再次重试
- list 有一个 blpop 指令，会一直等待消息到来，消息到来之前一直处于阻塞状态。
- Redis 延时队列 sortedset 实现，拿时间戳作为 score，用户使用 zrangebyscore 指令获取 n 秒之前的数据轮询进行处理。

5.2.26 列举一些常用的数据库可视化工具？

- MySQL：Navicat Premium、DBeaver
- Redis：Redis Desktop Manager
- MongoDB：Robot 3T

5.3 MongoDB

5.3.1 NoSQL 数据库主要分为哪几种？分别是什么？

- NoSQL 主要有四种模型 ： 键值对、文档、列族、图。
- NoSQL 数据分布：数据一般存储在不同的服务器上，不同的数据不同的服务器都有备份，每个节点都可以进行查找操作。
- NoSQL 数据复制: 主从复制模式，主节点负责写入数据，从节点随时从主节点复制数据，各个节点保持数据同步；对等复制模式，任何节点都可以进行写入操作，节点间相互同步其数据。

5.3.2 MongoDB 的主要特点及适用于哪些场合？

- 特点：极高的性能，方便部署和使用，存储数据简洁方便。支持的数据类型丰富，可以动态查询，可以存储大型数据。
- 适用场合：网站数据存储、缓存、大型但是价值较低的数据对象、用于对象和 JSON 格式数据的存储、也适用于高伸缩性的应用场景（MongoDB 包含对 MapReduce 引擎的内置支持）。

5.3.3 MongoDB 中的文档有哪些特性？

文档 document：文档是 MongoDB 中的基本单元即 BSON，类似关系数据库中的行，文档有唯一的标识 _id，文档都是以键值方式。文档的键值类似字典形式，键使用双引号（可省略），值字符串使用双引号（不能省略），整数不用双引号。

文档的特性：

- 文档的键值对是有序的
- 文档的值可以是字符串，整数，列表，字典，数组，以及文档等类型
- 文档的值区分大小写及值的类型
- 文档中的值不仅可以是在双引号里面的字符串，还可以是其他几种数据类型（甚至可以是整个嵌入的文档）。
- 文档不能有重复的键。
- 文档的键是字符串。除了少数例外情况，键可以使用任意 UTF-8 字符。

5.3.4 MongoDB 中的 key 命名要注意什么?

- `\0`（空字符）不能使用
- 带有 `.` 号，`_` 号和 `$` 号前缀的 Key 被保留
- 大小写有区别，Age 不同于 age
- 同一个文档不能有相同的 Key
- 除了上面几条规则外，其他所有 UTF-8 字符都可以使用

5.3.5 MongoDB 数据库使用时要注意的问题?

- MongoDB 没有数据恢复机制，需要主动做好数据备份
- MongoDB 默认全索引，索引是放在内存之中，默认支持 2.5G 的数据，具体大小与服务器配置有关，可以自行设置
- MongoDB 默认地址是 127.0.0.1，但是存在不安全性，使用是需要配置 localhost 主机名
- MongoDB 使用 GetLastError 确保变更

5.3.6 常用的查询条件操作符有哪些?

如下:

```
$gt ---- > 大于
$lt ---- < 小于
$gte ---- >=  大于等于
$lte ---- <=  小于等于
$ne ---- != 、<>  不等于
$in ---- in  包含
$nin ---- not in  不包含
$all ---- all 所有的
$or ---- or  或者
$not ---- 否定匹配
```

5.3.7 MongoDB 常用的管理命令有哪些?

如下:

```
# 使用管理员进入操作数据库
use admin
# 增加用户
db.addUser('felix','12345678')
db.addUser('felix','12345678',true) true表示只读
# 查看所有的用户列表
db.system.users.find()
# 删除用户
db.removeUser('felix')
# 查看数据库所有用户
show users
# 查看所有数据库
show dbs
# 查看所有的集合
show collections
# 查看所有集合的状态
db.printCollectionStats()
# 修复数据库
db.repairDatabase()
```

```
# 查看profiling信息
show profiling
# 复制数据库
db.copyDatabase('test1','test2')
# 删除集合
db.students.drop()
# 删除当前数据库
db.dropDatabase()
```

5.3.8 MongoDB 为何使用 GridFS 来存储文件?

- GridFS 是一种存储规范，主要用来存储大文件
- GridFS 可以将大文件转换为小文件，然后进行存放，解决了 BSON 对象限制大小的问题

5.3.9 如果一个分片（Shard）停止或很慢的时候，发起一个查询会怎样?

- 分片停止了，查询数据时候会返回一个错误
- 分片响应速度很慢，MongoDB 会等待它的响应

5.3.10 分析器在 MongoDB 中的作用是什么?

- MongoDB 中的数据库分析器用来显示数据库中每个操作性能特点。
- MongoDB 分析器可以分析查询操作的速度，方便我们对数据库进行优化，比如对一些操作慢的查询，可以添加索引提高查询的效率。

5.3.11 MongoDB 中的名字空间（namespace）是什么?

- MongoDB 存储 BSON 对象在集合（collection）中。
- 数据库名字和集合名字采用句点进行连接，这种连接就叫做名字空间（namespace）。
- MongoDB 每个集合和每个索引都对应一个命名空间，这些命名空间的元数据集中在 16M 的 *.ns 文件中，平均每个命名占用约 628 字节，也即整个数据库的命名空间的上限约为 24000。

5.3.12 更新操作会立刻 fsync 到磁盘吗?

- 不会立刻同步，因为磁盘的写操作默认是延时执行。一般默认时间是 2~3 秒。
- 比如一秒内数据库一个对象进行大量操作，但是磁盘只会刷新一次，写入磁盘就是有延时的。

5.3.13 什么是 master 或 primary? 什么是 secondary 或 slave?

- master 或者 primary 是集群（replica set）主节点，用于写入数据操作。当一个主节点失效后，一个 slave 从节点会升级为主节点。
- secondary 或者 slave 从当前的 master 主节点进行复制同步，主要用于数据的备份。
- 主从节点主要目的是提高数据库的性能和数据库的安全性。

5.3.14 必须调用 getLastError 来确保写操作生效了么?

- 并不需要，操作数据库时，有没有调用 getLastError 方法，数据库做的操作其实都是一样的。
- 我们调用 getLastError 方法，只是为了确认数据库写入操作是否成功，目的只是为了进行主观确认，但是写入是否成功并不是由 getLastError 决定的。

5.3.15 MongoDB 副本集原理及同步过程?

副本集同步过程：Primary 主节点负责写入数据，Secondary 节点会读取 Primary 的 oplog 信息得到复制信息，然后开始复制数据，同时将复制信息也写入到自己的 oplog。

执行具体步骤：检查自己 oplog 信息中最近的时间戳，然后 Primary 节点检查自己的时间戳，将大于该时间戳的操作记录写入到 oplog.rs 集合中。通过比较时间戳确定数据是否需要同步。

注意：在副本集中，如果所有的 Secondary 失效了，仅剩下 Primary 节点，此时 Primary 会自动变成 Secondary，不能对外提供服务。

5.3.16 MongoDB 中的分片是什么意思?

- 分片是为了应对大量的数据存储操作，将大数据进行分割，然后存储在不同的物理节点上
- 当单台机器性能不能满足时，就需要使用分片技术，利用大量的机器来应对数据量的增加和大量的读写操作

5.3.17 “ObjectID”有哪些部分组成?

- ObjectID 由四部分组成：时间戳、客户端的 ID、客户进程的 ID、增量计数器。
- _id 保证了文档的唯一性，插入文档时，我们可以指定 _id，如果用户没有指定，MongoDB 会自动设置一个 _id。
- _id 的头 4 个字节代表的是当前的时间戳，接着的后 3 个字节表示的是机器 id 号，接着的 2 个字节表示 MongoDB 服务器进程 id，最后的 3 个字节代表递增值。

5.3.18 在 MongoDB 中什么是索引?

MongoDB 索引是一种特殊的数据类型，它将数据转换为特定的形式，当查找数据时候，可以通过对象的索引，迅速查找到对象。

- 索引一般选取特定字段进行存储，并且有自己的排序规律，对索引进行排序。
- 如果不采用索引，MongoDB 查询文档时候，就会像遍历列表一样，去查询所有数据，大大降低查询的效率。

5.3.19 什么是聚合?

- 聚合操作就是根据一定的操作，返回特定的结果。 MongoDB 中的聚合操作，应该使用 aggregate() 方法。
- 聚合操作类似 SQL 数据库中的分组操作组合 group by。

5.3.20 写一个 Python 连接操作 MongoDB 数据库实例?

```
'''
Author: Felix
WX: AXiaShuBai
Email: xiashubai@gmail.com
Blog: https://blog.csdn.net/u011318077
Date: 2019/12/9 11:17
Desc:
'''
# 案例引用自作者本人博客

# -*- coding: utf-8 -*-
# 注意，先要启动本地的MongoDB服务器

from pymongo import MongoClient

class TestMongo():

    # 初始化时自动创建连接
    def __init__(self):

        # 创建MongoDB客户端
        client = MongoClient(host='127.0.0.1', port=27017)
```

```
        # 使用方括号选择test数据库里面的t251集合
        self.collection = client["test"]["t251"]
        # 也可以使用.的方式
        # self.collection = client.test.t251

# 实例化
test_mongo = TestMongo()
# test_mongo.xxx具体操作
```

6. 数据解析提取

6.1 正则表达式

6.1.1 match、search 和 findall 有什么区别?

- match 从字符串的开始进行匹配，如果字符串第一个字符不符合匹配规则，则匹配失败，函数返回 None 值;
- search 从字符串左侧开始，然后向右匹配字符串，当找到第一个匹配，匹配结束;
- findall 查找整个字符串，返回所有的匹配结果，匹配结果是一个列表。

6.1.2 正则表达式的 ()、[]、{} 分别代表什么意思?

- `()`：匹配的的字符串进行分组，目的是为了提取匹配的字符串。表达式中有几个 `()` 就有几个相应的匹配字符串, 一个 `()` 代表一个组。
- `[]`：定义匹配的字符范围，匹配多个数字多个字母等, 匹配中括号任何一个都可以，类似或或或... 比如 `[a-zA-Z0-9]` 表示匹配字母和数字。`[\s*]` 表示空格或者 `*` 号。（注意：中括号里面的所有表达式只匹配一个就结束，中括号只代表一个字符。要匹配多个后面使用 `+` 号）
- `{}`：一般用来定义匹配的长度，只限制 `{}` 它前面的一个字符，比如 `\d{3}` 表示匹配三个数字，`\d{1,3}` 表示匹配一到三个数字，包括 1 和 3。

6.1.3 正则表达式中的 `.\*`、`.+`、`.\*?`、`.+?` 有什么区别?

- `*` 匹配 0 个或多个的表达式，贪婪模式
- `+` 匹配 1 个或多个的表达式，非贪婪
- `?` 匹配 0 个或 1 个由问好前面字符或者表达式，非贪婪方式
- `.*`：贪婪匹配，找到满足条件的最大匹配
- `.+`：贪婪匹配，找到满足条件的最大匹配
- `.*?`：非贪婪，找到满足条件的最小匹配
- `.+?`：非贪婪，找到满足条件的最小匹配

参考代码:

```
'''
Author: Felix
WX: AXiaShuBai
Email: xiashubai@gmail.com
Blog: https://blog.csdn.net/u011318077
Date: 2019/12/9 11:17
Desc:
'''

import re
```

```
html = '<H1>Chapter<H1><H1>Chapter<H1>END'

res = re.match(r'<.*>', html)
print(res)
print(res.group())
print('*' * 50)

res = re.match(r'<.+>', html)
print(res)
print(res.group())
print('*' * 50)

# 非贪婪，?后面有一个>，会找到第一个满足表达式的匹配就结束匹配
res = re.match(r'<.+?>', html)
print(res)
print(res.group())
print('*' * 50)

# 非贪婪，?后面有一个>，会找到第一个满足表达式的匹配就结束匹配
res = re.match(r'<.*?>', html)
print(res)
print(res.group())
print('*' * 50)
<re.Match object; span=(0, 30), match='<H1>Chapter<H1><H1>Chapter<H1>'>
<H1>Chapter<H1><H1>Chapter<H1>
**************************************************
<re.Match object; span=(0, 30), match='<H1>Chapter<H1><H1>Chapter<H1>'>
<H1>Chapter<H1><H1>Chapter<H1>
**************************************************
<re.Match object; span=(0, 4), match='<H1>'>
<H1>
**************************************************
<re.Match object; span=(0, 4), match='<H1>'>
<H1>
**************************************************
```

6.1.4 `.\*?` 贪婪匹配的一种特殊情况？当 `\*` 和 `?` 中间有一个字符会怎么样？

`?` 本来是匹配 0 次或 1 次，但是 `?` 前面字符的前面是 `.*` 会出现意向不到的结果。

参考代码：

```
import re

# 特殊情况，表达式中含有.*?，但是?和前面的*间隔一个字符在后面不在一起
# 如果结尾是?号，直接匹配到末尾
# 如果?号后面还有字符，直接匹配到最后一个符合条件的字符
print('特殊情况：')

html1 = '<H1>Chapter<H1><H1>Chapter<H1>555END'

res = re.match(r'<.*>5', html1) # <H1>Chapter<H1><H1>Chapter<H1>5
print(res.group())

res = re.match(r'<.*>?5', html1) # <H1>Chapter<H1><H1>Chapter<H1>555
```

```
print(res.group())

res = re.match(r'<.*>?N', html1) # <H1>Chapter<H1><H1>Chapter<H1>555EN
print(res.group())

res = re.match(r'<.*>?', html1) # <H1>Chapter<H1><H1>Chapter<H1>555END
print(res.group())

res = re.match(r'<.*>?.*', html1) # <H1>Chapter<H1><H1>Chapter<H1>555END
print(res.group())

res = re.match(r'<.*>5?', html1) # <H1>Chapter<H1><H1>Chapter<H1>5
print(res.group())

res = re.match(r'<.*5?', html1) # <H1>Chapter<H1><H1>Chapter<H1>555END
print(res.group())
```

6.1.5 `\s` 和 `\S` 是什么意思？re.S 是什么意思？

- `\s`：空白字符都可以匹配，比如空格符、制表符、换页符等，等价于 `[\f\n\r\t\v]`。
- `\S`：非空白字符都可以匹，等价于 `[^\f\n\r\t\v]`，`^` 表示除了什么以外。
- re.S：re.S 参数代表前面的字符串是一个整体，里面的特殊字符都是普通字符，比如特殊符号 `\n`、`\t` 都只是普通字符，可以去掉 `\` 的转义含义。

6.1.6 写一个表达式匹配座机或者手机号码？

参考代码：

```
import re

# 全国座机电话判断,区号有些三位有些四位，电话号码有些8位或7位
# 手机号都是11位，1开头，r'1\d{10}?'
res = re.match(r'\d{3,4}-?\d{7,8}', '0717-7640999')
print(res)
print(res.group())
<re.Match object; span=(0, 12), match='0717-7640999'>
0717-7640999
```

6.1.7 正则表达式检查 Python 中使用的变量名是否合法？

参考代码：

```
'''
Author: Felix
WX: AXiaShuBai
Email: xiashubai@gmail.com
Blog: https://blog.csdn.net/u011318077
Desc:
'''

import re

names = ['name1', '_name', 'name_1_1', '2_name', '_name_', '#_name_','name?_',
'name!']
```

```
for name in names:
    # 注意：\w等价于[A-Za-z0-9_]，但是可以匹配中文字符
    # 变量名字母或者下划线开头(不能数字开头)，字母数字下划线结尾
    # [a-zA-Z_]:判断第一位一定只能是字母或者下划线
    # [a-zA-Z0-9_]*:第二位开始只能是字母数字下划线，可以出现0次或多次
    # $：检测到字符串的结尾,最后一个符合的字符也是字符串的结尾
    # ^：判断开头，match默认从开头开始判断，所以可以不写
    res = re.match(r'^[a-zA-Z_][a-zA-Z0-9_]*$', name)
    if res:
        print("变量名 %s 符合要求" % res.group())
    else:
        print("变量名 %s 非法" % name)
变量名 name1 符合要求
变量名 _name 符合要求
变量名 name_1_1 符合要求
变量名 2_name 非法
变量名 _name_ 符合要求
变量名 #_name_ 非法
变量名 name?_ 非法
变量名 name! 非法
```

6.1.8 正则表达式检查邮箱地址是否符合要求?

邮箱名称规则自己定义，参考代码：

```
import re

def check_email_address():

    email = input('请输入邮箱地址:')
    # 如果正则表达式需要使用特殊字符：. ? + 等符号需要前面添加\反斜杠进行转义
    # 转义后代表一个普通字符
    ret = re.match(r'[a-zA-Z0-9]{6,12}@(163|126|qq)\.com$', email)
    if ret:
        print("%s 符合要求" % email)
    else:
        print("%s 不符合要求" % email)

if __name__ == '__main__':
    check_email_address()
请输入邮箱地址:12345ABC@126.com
12345ABC@126.com 符合要求

请输入邮箱地址:123ddd-12@126.com
123dd-12@126.com 不符合要求
```

6.1.9 如何使用分组匹配 HTML 中的标签元素?

`\w`：匹配包括下划线的任何单词字符。等价于 `'[A-Za-z0-9_]'`。

标签元素匹配：

```
# 为了限制前后匹配的标签一致，就可以使用小括号()分组和\num匹配前面的分组
```

```
# \1就代表匹配与第一个分组(\w*)匹配到的结果要一模一样
html_str = '<h1>hello, world!</h1>'
ret = re.match(r'<(\w*)>.*</\1>', html_str)
print(ret.group())
print('*' * 50)

# 进行两个分组，使用\1 \2,类似位置参数
# 注意\1 \2 位置不要写反了，写反了就匹配不到了
html_str = '<body><div>hello, world!</div></body>'
ret = re.match(r'<(\w*)><(\w*)>.*</\2></\1>', html_str)
print(ret.group())
print('*' * 50)
<h1>hello, world!</h1>
**************************************************
<body><div>hello, world!</div></body>
**************************************************
```

6.1.10 如何使用 re.sub 去掉“028-00112233 # 这是一个电话号码”# 和后面的注释内容?

re.sub 用于替换字符串中的匹配项，匹配到的内容全部替换。

先进行 findall 查找匹配，然后进行替换，最后返回结果。

```
re.sub(pattern, repl, string, count=0)
import re

# sub替换案例
# 删除注释
phone = "028-00112233 # 这是一个电话号码"
num = re.sub(r'#.*$', "", phone)
print("电话号码 : ", num)
print('*' * 50)

# 移除非数字的内容,\D非数字内容，全部替换
phone = "2004-959-559 # 这是一个电话号码"
num = re.sub(r'\D', "", phone)
print("电话号码 : ", num)
print('*' * 50)
电话号码 :  2004-959-559
**************************************************
电话号码 :  2004959559
**************************************************
```

6.1.11 re.sub 替换如何支持函数调用？举例说明?

参考代码：

```
import re

# 定义一个函数，tem参数就是正则匹配的结果
def add(temp):
    strNum =temp.group()
    num = int(strNum) + 1
    return str(num)

# add自动调用上面的add函数，匹配的值处理后返回一个值
# 返回值替换掉字符串中的值
ret = re.sub(r'\d+', add, 'python = 999')
print(ret) # 1000
```

> 引用来源（CSDN 博客）： https://blog.csdn.net/weixin_38819889/article/details/93845829

6.1.12 如何只匹配中文字符?

参考代码：

```
import re

title = '世界 你好，hello world'

# 匹配中文字符，找出所有的中文字符
p = re.compile(r'[\u4e00-\u9fa5]+')
rst1 = p.findall(title)
print(rst1)
['世界', '你好']
```

6.1.13 如何过滤评论中的表情?

主要是匹配表情包的范围，将表情包的范围用空替换掉。

参考代码：

```
import re
pattern = re.compile(u'[\uD800-\uDBFF][\uDC00-\uDFFF]')
pattern.sub('',text)
```

6.1.14 Python 中的反斜杠 \ 如何使用正则表达式匹配?

- 方式 1：使用四个反斜杠
- 方式 2：正则表达式使用 r 声明，然后使用两个反斜杠

参考代码：

```
# 注意，python中\a和\b都有特殊含义，使用\\去掉转义
lll = "c:\\a\\b\\c"

ret = re.match('c:\\\\a', lll)
print(ret.group())
print('*' * 50)
```

```
ret = re.match(r'c:\\a', lll)
print(ret.group())
print('*' * 50)
c:\a
**************************************************
c:\a
**************************************************
```

6.1.15 如何提取出下列网址中的域名？

提取网址中的域名，参考代码：

```
import re

#提取出域名
s = """
http://www.interoem.com/messageinfo.asp?id=35`
http://3995503.com/class/class09/news_show.asp?id=14
http://lib.wzmc.edu.cn/news/onews.asp?id=769
http://www.zy-ls.com/alfx.asp?newsid=377&id=6
http://www.fincm.com/newslist.asp?id=415
"""

p = re.compile(r"(http://.+?/).+")
res = p.findall(s)
print(res)
['http://www.interoem.com/', 'http://3995503.com/', 'http://lib.wzmc.edu.cn/',
'http://www.zy-ls.com/', 'http://www.fincm.com/']
```

> 代码引用来源（CSDN 博客）：https://blog.csdn.net/qq_34663267/article/details/81810164

6.1.16 去掉 'ab;cd%e\tfg,,jklioha;hp,vrww\tyz' 中的符号，拼接为一个字符串？

使用 split 和 join 函数，参考代码：

```
import re
# 将多个分隔符直接写在正则表达式中，使用中括号，后面的+号表示前面的符号可以出现多次
# 中括号每个字符都是一个匹配
s = 'ab;cd%e\tfg,,jklioha;hp,vrww\tyz'
t = re.split(r'[;%,\t]+', s)
res = ('').join(t)
print(res)
abcdefgjkliohahpvrwwyz
```

6.1.17 str.replace 和 re.sub 替换有什么区别？

- replace 一次只能替换一个
- sub 可以一次传入多个替换对象

参考代码：

```
import re

s = r'\tafa\t13d\t8773\rfafa\r323'
```

```
# replace替换方法，但是每次只能替换一个
# 删除所有\t,替换为空格，也可以替换为其它字符
s1 = s.replace(r'\t', '')
print(s1)
print('*' * 50)

# re.sub正则替换一次可以替换多个
s = '\tafa\t13d\t8773\rfafa\r323'
s2 = re.sub(r'[\t\r]', '', s)
print(s2)
print('*' * 50)
afa13d8773\rfafa\r323
**************************************************
afa13d8773fafa323
**************************************************
```

6.1.18 如何使用重命名分组修改日期格式?

参考代码:

```
import re

s = '2016-05-23 10:52:12 i am ok, you are ok'

# 捕获组进行关键字编号，格式如下：(?P<名称>) \g<名称>
t = re.sub(r'(?P<year>\d{4})-(?P<month>\d{2})-(?P<day>\d{2})',
           r'\g<month>/\g<day>/\g<year>', s)
print(t)
print("*" * 50)
05/23/2016 10:52:12 i am ok, you are ok
**************************************************
```

**6.1.19 (?:x) a(?=x) a(?!=x) (?<=x)a (?

```
'''
Author: Felix
WX: AXiaShuBai
Email: xiashubai@gmail.com
Blog: https://blog.csdn.net/u011318077
Date: 2019/12/9 11:27
Desc:
'''

import re

# 正常捕获匹配
s1 = 'http://google.com/index'
res = re.match('(http|ftp)://([^/\r\n]+)(/[^\r\n]*)?', s1)
print(res)
print(res.groups())
print('*' * 50)

# 非捕获匹配
```

```
s1 = 'http://google.com/index'
res = re.match('(?:http|ftp)://([^/\r\n]+)(/[^\r\n]*)?', s1)
print(res)
print(res.groups())
print('*' * 50)

# 正常捕获，一个正则表达式是正常匹配，第一个括号返回网络协议；
# 后一个正则表达式是非捕获匹配，返回结果中不包括网络协议。
<re.Match object; span=(0, 23), match='http://google.com/index'>
('http', 'google.com', '/index')
**************************************************
<re.Match object; span=(0, 23), match='http://google.com/index'>
('google.com', '/index')
**************************************************
```

先行断言：

```
'''
Author: Felix
WX: AXiaShuBai
Email: xiashubai@gmail.com
Blog: https://blog.csdn.net/u011318077
Date: 2019/12/13 14:06
Desc:
'''

import re

s = '98%1KK58%2AA65%3'

# 先行断言和先行否定断言都是取出来的断言表达式的匹配结果
# 找出以%结尾的所有数字
res = re.findall('\d+(?=%)', s)
print(res)
print('*' * 50)

# 找出所有不是以%结尾的数字
res = re.findall('\d+(?!%)', s)
print(res)
print('*' * 50)
['98', '58', '65']
**************************************************
['9', '1', '5', '2', '6', '3']
**************************************************
```

后瞻断言：

```
'''
Author: Felix
WX: AXiaShuBai
Email: xiashubai@gmail.com
Blog: https://blog.csdn.net/u011318077
Date: 2019/12/13 14:06
Desc:
```

```
'''

import re

s = '98%1KK58%2AA65%3'

# 找出数字前面是%的数字
res = re.findall('(?<=%)\d+', s)
print(res)
print('*' * 50)

# 找出数字前面不是%的数字
res = re.findall('(?<!%)\d+', s)
print(res)
print('*' * 50)
['1', '2', '3']
**************************************************
['98', '58', '65']
**************************************************
```

6.2 XPath

6.2.1 XML 是什么？XML 有什么用途?

XML 是一种可扩展标记语言（EXtensible Markup Language）。XML 是类似 HTML 的标记语言，设计目的是用来传输数据，而不是像 HTML 一样显示数据。

XML 的用途：

- XML 把数据从 HTML 分离
- XML 简化数据共享、传输
- XML 简化平台变更
- XML 使您的数据更有用

6.2.2 XML 和 HTML 之间有什么不同?

设计的目的不同：

- XML 用来传输和存储数据，关心点是数据的内容。
- HTML 用来显示数据，关心点是数据的外观。
- HTML 中使用的标签都是预定义的。HTML 文档只能使用在 HTML 标准中定义过的标签。XML 可以自定义标签和文档结构。

本质区别：HTML 目的是显示信息，而 XML 目的是在传输信息。

6.2.3 描述一下 XML lxml XPath 之间有什么关系?

- XML 是一种类似 HTML 的标记语言
- lxml 是 Python 的一个解析库，支持 HTML 和 XML 的解析
- XPath，全称 XML Path Language，即 XML 路径语言，它是一门在 XML 文档中查找信息的语言，它最初是用来搜寻 XML 文档的，但是它同样适用于 HTML 文档的搜索

使用步骤：请求网页获取的 text 文本内容，使用 lxml 解析为 XML 语言，然后使用 XPath 查找；请求网页获取到的 bytes 内容，可以使用 BeautifulSoup4（bs4 也可以传入字符串，bs4 会根据传入类型自己识别解码），解析模块可以选择 lxml 进行解析，然后查找。

6.2.4 介绍一下 XPath 的节点?

- 有七种类型的节点：元素、属性、文本、命名空间、处理指令、注释以及文档（根）节点。
- XPath 里面总共有七种节点：element、 attribute、text、namespace、processing instruction、comment 和 document noteds。

6.2.5 XPath 中有哪些类型的运算符?

XPath 运算符可以根据其属性分为不同的类别。以下是不同类型的 XPath 运算符：

- 比较运算符
- 布尔运算符
- 数字函数运算符
- 字符串函数
- 节点函数运算符

6.2.6 XPath 中的 `///`、`./`、`../`、`.//` 别有什么区别?

- `/` 引入绝对位置路径，从文档的根开始。比如 `/book`，从文档中的 book 标签节点开始向下查找。
- `//` 从匹配选择的当前节点中选择文档中的节点，而不考虑它们的位置，`//book`，选取文档中所有的 book 标签，不管他们在什么位置
- `.` 从上下文节点开始引入相对位置路径。点称为“上下文项表达式”，找的是相对路径，和 Python 中的文件路径一样，会找父级和子级。如果在表达式的开头加上一个点，它将使其特定于上下文。换句话说，它将 `id="Passwd"` 在您调用“通过 XPath 查找元素”方法的节点的上下文中搜索元素。
- `./` 从选定的当前节点开始向下查找
- `../` 选取当前节点的父节点开始向下查找
- `.//` 从选择的当前节点处开始，向下查找，不管在文中的位置

补充：`.` 主要是为了引入上下文，相对路径，如果使用了 `.` 就表示从当前节点下开始。

实际数据解析中，都是选选定了文档，然后从当前文档的根节点开始，比如：

一级文档：response，提取出二级内容 jobs，然后在每个 jobs 里面继续提取 jobname

```
jobs = response.xpath(".//div[@id='resultList']//div[@class='el']")
jobName = job.xpath("./p/span/a/text()").extract_first().strip()
```

6.2.7 XPath 中如何同时选取多个路径?

使用 `|` 运算符，可以同时选取若干个路径：

```
//article/number | //article/author
选取 article 元素的所有 number 和 author元素。
```

6.2.8 XPath 中的 `\*` 和 `@\*` 分别表示什么含义?

- 一般用于选取未知的节点
- `*` 匹配任何元素节点
- `@*` 匹配任何属性节点

```
未知节点
/article/*  选取 article 元素的所有子元素。
//* 选取文档中的所有元素。
//author[@*]    选取所有带有属性的author元素

已知属性的节点
//author[@id='novelInfo'] 选取所有拥有id为novelInfo的属性的author元素
```

6.2.9 如何使用位置属性选取节点中的元素?

可以使用类似 Python 列表中的位置，元素太多可以使用 last()，或者使用 position。

```
/bookstore//book 选取属于 bookstore 子元素的下的所有book 子元素。
/bookstore/book[3]  选取属于 bookstore 子元素的第3个 book 元素。
/bookstore/book[4]  选取属于 bookstore 子元素的第4个 book 元素。
/bookstore/book[last()] 选取属于 bookstore 子元素的最后一个 book 元素。
/bookstore/book[last()-1]   选取属于 bookstore 子元素的倒数第二个 book 元素。
/bookstore/book[position()<3]   选取最前面的两个属于 bookstore 元素的子元素的 book 元素。
```

> 引用来源：https://www.2cto.com/shouce/w3school/xpath/xpath_syntax.asp.html

6.2.10 XPath 中如何多条件查找?

使用 and 书写多个条件，参考代码：

```
# -*- coding:utf-8 -*-
'''
Author: Felix
Email: xiashubai@gmail.com
Blog: https://blog.csdn.net/u011318077
Date: 2019/12/16 14:49
Desc:
'''
from lxml import etree

xml_test = '''
<Article>
<list><data type="String">Alpha</data></list>
<list><data type="Number1">200</data></list>
<list><data type="Number2">200</data></list>
<list><data type="Number3">200</data></list>
<list><data type="Boolean">true</data></list>
<list><data type="Number2">true</data></list>
</Article>
'''

html = etree.HTML(xml_test, etree.HTMLParser())
print(html.xpath('//list//data[text()=200]/text()'))
print(html.xpath('//list//data[text()=200 and @type="Number2"]/text()'))
['200', '200', '200']
['200']
```

6.2.11 Scrapy 和 lxml 中的 XPath 用法有什么不同?

- Scrapy 里面的 xpath 默认使用的是 Selector 下的 xpath 方法, 返回的是一个 Selector 列表
- lxml 里面的 xpath 默认就是使用的 SelectorList 下的 xpath 方法，返回的是一个内容列表

参考代码:

```
# -*- coding:utf-8 -*-
'''
Author: Felix
WX: AXiaShuBai
Email: xiashubai@gmail.com
Blog: https://blog.csdn.net/u011318077
Date: 2019/12/16 11:52
Desc:
'''

# Selector和SelectorList都是属于selector.py模块，只是两种对象的结果不一样
# scrapy里面的xpath默认就是使用的Selector下的xpath方法,返回的是一个Selector列表
# lxml里面的xpath默认就是使用的SelectorList下的xpath方法，返回的是一个内容列表

from lxml import etree
from scrapy.selector import Selector

body = '<body>hello</body><body>world</body>'

# scrapy下使用的是Selector类(返回结果是selector对象)下,
selectors = Selector(text=body)
print(type(selectors.xpath('//body/text()')))
print(selectors.xpath('//body/text()')) # 结果是一个selector列表
print(selectors.xpath('//body/text()').extract()) # 内容的列表
print(selectors.xpath('//body/text()').extract()[0])
print(selectors.xpath('//body/text()').extract_first()) # 推荐写法
print(selectors.xpath('//body/text()')[0])  # 不推荐写法
print(selectors.xpath('//body/text()')[0].extract())  # 不推荐写法
print('*' * 50)

# lxml下使用的xpath是SelectorList类(返回结果是selector对象的内容对象)下，因此可以直接列表
取出内容
html = etree.HTML(body, etree.HTMLParser())
print(type(html.xpath('//body/text()')))
print(html.xpath('//body/text()'))
print(html.xpath('//body/text()')[0])
print(html.xpath('//body/text()')[-1])
<class 'scrapy.selector.unified.SelectorList'>
[<Selector xpath='//body/text()' data='hello'>, <Selector xpath='//body/text()'
data='world'>]
['hello', 'world']
hello
hello
<Selector xpath='//body/text()' data='hello'>
hello
**************************************************
<class 'list'>
['hello', 'world']
```

```
hello
world
```

6.2.12 用过哪些常用的 XPath 开发者工具?

- 开源的 XPath 表达式工具：XMLQire
- Chrome 插件：XPath Helper
- Firefox 插件：XPath Checker（新版已经没有了该插件）
- Firefox 中有一个 Try Path 插件，用来验证 XPath 路径是否正确
- 浏览器检查元素里面，选定元素后，可以复制 XPath 路径

6.3 BeautifulSoup4

6.3.1 BeautifulSoup4 是什么？有什么特点?

BeautifulSoup4 是 Python 的一个 HTML 或 XML 的解析库，我们可以用它来方便的从网页中提取数据，它拥有强大的 API 和多样的解析方式。

Beautiful Soup4 的三个特点：

- Beautiful Soup 提供一些简单的方法和 Python 式函数，用于浏览，搜索和修改解析树，它是一个工具箱，通过解析文档为用户提供需要抓取的数据
- Beautiful Soup 自动将转入稳定转换为 Unicode 编码，输出文档转换为 UTF-8 编码，不需要考虑编码，除非文档没有指定编码方式，这时只需要指定原始编码即可
- Beautiful Soup 位于流行的 Python 解析器（如 lxml 和 html5lib）之上，允许您尝试不同的解析策略或交易速度以获得灵活性。

> 引用来源（CSDN 博客）：https://blog.csdn.net/huang714/article/details/97367656

6.3.2 三种解析工具：正则表达式 lxml BeautifulSoup4 各自有什么优缺点?

解析工具	解析速度	使用难度
BeautifulSoup4	最慢	最简单
lxml	快	简单
正则	最快	最难

6.3.3 etree.parse()、etree.HTML() 和 etree.tostring() 有什么区别?

- etree.parse() 直接接受一个 HTML 文件，解析为 XML 对象，会自动修复 HTML 文件中缺失的信息，如声明信息、标签缺失等
- etree.HTML() 接受的一个字符串类型的对象，一般先获取 html 文档然后读取后传入，解析成 XML 对象
- etree.tostring() 用来将 etree.HTML() 解析后的对象转换为 bytes 字节对象然后输出，可以使用 decode() 解码为字符串对象

6.3.4 BeautifulSoup4 支持的解析器以及它们的优缺点?

解析器	使用方法	优势	劣势
Python 标准库	BeautifulSoup(markup，“html.parser”)	Python 的内置标准库、执行速度适中、文档容错能力强	Python 2.7.3 之前版本容错较差
lxml HTML 解析器	BeautifulSoup(markup，“lxml”)	速度快、容错能力强	需要 C 语言库支持
lxml XML 解析器	BeautifulSoup(markup,[“lxml”，“xml”]) BeautifulSoup(markup，“xml”)	速度快、支持 XML 的解析	需要 C 语言库的支持
html5lib	BeautifulSoup(markup，“html5lib”)	容错性好、解析结果是 HTML5 格式的文档	解析速度慢

6.3.5 BeautifulSoup4 中的四大对象是什么?

四大对象：Tag 标签、NavigableString 字符串、Beautifulsoup 文档、Comment 注释

Tag（标签对象）

- 对应 HTML 中的标签
- 可以通过 soup.tag_name 查看
- tag 标签有两个重要的属性
  - name
  - attribute（使用时用 .attrs）：attrs 属性是一个字典，修改删除类似于字典

NavigableString（一般为字符串对象）

- 查看标签内容中的值
- 用法 tag_name.string
- 标签中的内容不能编辑，但是可以替换

Beautifulsoup（整个文档对象）

- 表示一个文档的内容，大部分可以当做一个 tag 对象
- Beautifulsoup 实例的对象包含一个特殊的属性
- 对象包含了一个值为“[document]”的特殊属性，使用方法 soup.name
- Comment（注释对象）

通常情况下 Tag、NavigableString、BeautifulSoup 几乎覆盖了所有 HTML 和 XML 中的所有内容, 但是还有一些特殊对象。容易让人担心的内容是文档的注释部分，Comment 对象是一个特殊类型的 NavigableString 对象。

参考代码：

```
from urllib import request
from bs4 import BeautifulSoup

url = 'http://www.baidu.com'

rsp = request.urlopen(url)
```

```
content = rsp.read()

# 创建一个bs的实例
# lxml是指使用的 lxml HTML 解析器
soup = BeautifulSoup(content, 'lxml')
print(type(soup))

print(soup.title) # 获取整个标签内容
print(soup.title.name)  # 获取标签的名称
print(soup.title.attrs) # 属性为空
print(soup.title.string) # 获取标签中的内容
print(soup.name)
<class 'bs4.BeautifulSoup'>
<title>百度一下，你就知道</title>
title
{}
百度一下，你就知道
[document]
```

6.3.6 BeautifulSoup4 中如何格式化 HTML 代码?

- 使用 prettify() 方法，该方法可以把解析的字符串以标准的缩进格式输出。
- 先使用 `soup = BeautifulSoup(content, 'lxml')` 解析得到 soup 对象，然后格式化 soup.prettify()。
- 这里需要注意的是，输出结果里面包含 body 和 html 节点，也就是说对于不标准的 HTML 字符串 BeautifulSoup，可以自动更正格式。这一步不是由 prettify() 方法做的，而是在初始化 BeautifulSoup 时就完成了。

6.3.7 BeautifulSoup4 中 find 和 find_all 方法的区别?

- find 查找满足条件的第一个节点，结果是单个元素，第一个匹配的元素，类型依然是 tag 类型
- find_all 查找满足条件的所有节点，返回的结果是一个列表，可以使用 for 循环取出

6.3.8 string、strings 和 stripped_strings 有什么区别?

- string：获取子节点字符串里面的内容。tag 只有一个 NavigableString 类型的子节点, 那么这个 tag 可以使用 .string 得到子节点内容。
- strings：一次获取子节点中所有的字符串内容，如果有多个字符串内容，需要使用循环获取。
- stripped_strings: 用来格式化字符串，strings 的字符串结果中可能有很多空格或者空行，使用 stripped_strings 可以去除多余空白内容。

6.3.9 BeautifulSoup4 输出文档的编码格式是什么?

- 使用 Beautiful Soup 解析输出文档时，输出时候编码会自动转换，转换后的格式是 UTF-8 编码
- 如果我们知道原始文档的编码，可以指定编码参数，防止自动转换出错

```
soup = BeautifulSoup(markup, from_encoding="GB2312")
```

7. 网络爬虫

7.1 网络爬虫是什么？它有什么特征?

网络爬虫定义：网络爬虫可以按照一定的规则，自动地抓取网络上的信息。有时候又叫做网络蜘蛛。

两大特征：

- 按照开发者的要求下载数据保存数据
- 能自动在网络上进行爬取

三大步骤：

- 请求下载网页
- 提取解析正确的信息然后保存
- 根据爬虫规则，自动跳转到新的网页，执行以上两步内容

爬虫分类：

- 通用爬虫（搜索引擎，类似百度、搜狗）
- 专用爬虫（聚焦爬虫）
- Python 中的网络爬虫都是专用爬虫

7.2 Python 中常用的爬虫模块和框架有哪些？它们有什么优缺点？

Python 自带：urllib、urllib2（Python 2.x）、urllib3、httplib2

第三方：requests、Scrapy、Pyspider

- urllib 是 Python 中请求 url 连接的官方标准库，urllib3 则是增加了连接池等功能，两者互相都有补充的部分。urllib 中，request 这个模块主要负责构造和发起网络请求，并在其中加入 Headers、Proxy 等。需要一步一步操作，代码比较繁琐。
- requests：requests 是 Python 的第三方库, 适合人类使用的库，使用简单，功能集成性强。
- Scrapy 和 Pyspider：第三方爬虫框架，爬取网站数据，提取结构性数据而编写的应用框架。可以应用在包括数据挖掘，信息处理或存储历史数据等一系列的程序中。

7.3 搜索引擎中的 ROBOTS 协议是什么？

搜索引擎的排名规则：PageRank 算法、点击次数、被引用次数。搜索引擎返回的大部分东西都是无用的，对多媒体资源搜索缺乏。通用搜索引擎所返回的网页里 90% 的内容。

ROBOTS 协议：网站开发时候定义的一些规则，目的是告诉搜索引擎哪些网页被爬取，哪些网页不能被爬取。但是这个规定只是网站自己单方面，属于道德层面的约定。只要用户可以看见的内容，实际都是可以被爬取的。

比如淘宝的 ROBOTS 协议：

```
https://www.taobao.com/robots.txt
User-agent:  Baiduspider
Allow:  /article
Allow:  /oshtml
Allow:  /ershou
Allow: /$
Disallow:  /product/
Disallow:  /

User-Agent:  Googlebot
Allow:  /article
Allow:  /oshtml
Allow:  /product
Allow:  /spu
```

```
Allow:  /dianpu
Allow:  /oversea
Allow:  /list
Allow:  /ershou
Allow: /$
Disallow:  /
```

7.4 urlib 和 requests 库请求网页有什么区别?

urlib 请求百度

```
# 导入 urllib 中的 request
from urllib import request

# 使用 urlopen，打开一个网址
urllib_response = request.urlopen("http://www.baidu.com")
# read() 方法用于读取响应， decode() 处理，默认 read() 是字节流
print(urllib_response.read().decode("utf-8"))
print(type(urllib_response.read()))  # <class 'bytes'>
```

requests 请求百度

```
# 导入requests库
import requests

response = requests.get("http://www.baidu.com")
# 请求百度会出现乱码，因为百度默认解析编码为 utf-8
# requests 的 response 默认为 ISO-8859-1
# 通过 response.encoding 直接设置此次响应解析编码
response.encoding = 'utf-8'
# 通过 response.text 即可查看响应的文本 body
print(response.text)
```

7.5 网页中的 ASCII Unicode UTF-8 编码之间的关系?

字符（Character）是各种文字和符号的总称，包括各国家文字、标点符号、图形符号、数字等字符集（Character set）是多个字符的集合字符集包括：ASCII 字符集、GB2312 字符集、GB18030 字符集、Unicode 字符集等。

ASCII 编码是 1 个字节，而 Unicode 编码通常是 2 个字节。

ASCII 只有 127 个字符，表示英文字母的大小写、数字和一些符号，但由于其他语言用 ASCII 编码表示字节不够，例如：常用中文需要两个字节，且不能和 ASCII 冲突，中国定制了 GB2312 编码格式，相同的，其他国家的语言也有属于自己的编码格式。

由于每个国家的语言都有属于自己的编码格式，在多语言编辑文本中会出现乱码，这样 Unicode 应运而生，Unicode 就是将这些语言统一到一套编码格式中，通常两个字节表示一个字符，而 ASCII 是一个字节表示一个字符，这样如果你编译的文本是全英文的，用 Unicode 编码比 ASCII 编码需要多一倍的存储空间，在存储和传输上就十分不划算。

为了解决上述问题，又出现了把 Unicode 编码转化为“可变长编码”UTF-8 编码，UTF-8 编码将 Unicode 字符按数字大小编码为 1-6 个字节，英文字母被编码成一个字节，常用汉字被编码成三个字节，如果你编译的文本是纯英文的，那么用 UTF-8 就会非常节省空间，并且 ASCII 码也是 UTF-8 的一部分。

UTF-8 是 Unicode 的实现方式之一，它可以看做 Unicode 的升级版。

计算机系统通用的字符编码工作方式：

- 在计算机内存中，统一使用 Unicode 编码，当需要保存到硬盘或者需要传输的时候，就转换为 UTF-8 编码。
- 用记事本编辑的时候，从文件读取的 UTF-8 字符被转换为 Unicode 字符到内存里，编辑完成后，保存的时候再把 Unicode 转换为 UTF-8 保存到文件。

> 以上部分内容引用自百度百科：https://baike.baidu.com/item/%E5%AD%97%E7%AC%A6%E9%9B%86/946585?fr=aladdin

7.6 urllib 如何检测网页编码？

使用 chardet 库，参考代码：

```
import chardet
from urllib import request

if __name__ == '__main__':
    url = 'http://china.chinadaily.com.cn/a/201901/15/WS5c54654afdadc3902.html'
    # 打开一个URL然后返回页面的内容
    rsp = request.urlopen(url)
    # 把返回的结果读取出来
    # <class 'http.client.HTTPResponse'>
    print(type(rsp))
    html = rsp.read()
    # 直接读取的html是字节格式 <class 'bytes'>
    print(type(html))
    # 利用chardet检测编码
    # 类型是一个字典，字典中有采用的编码
    cs = chardet.detect(html)
    print(type(cs)) # <class 'dict'>
    print(cs) # {'encoding': 'utf-8',....}

    # 获取网页编码用于解码
    # 返回的字节，需要解码
    # 使用get函数获得不会出错，get函数获取到的编码，就用检测到的编码解码
    # 如果没有检测到，就采用utf-8解码
    html = html.decode(cs.get('encoding', 'utf-8'))
    print(html)
```

7.7 urllib 中如何使用代理访问网页？

参考代码：

```
# 注意，文中使用代理可能已经失效，需要自己更换代理IP

from urllib import request, error
```

```
if __name__ == '__main__':

    url = 'http://www.cnqiang.com/'

    # 使用代理的步骤
    # 1.设置代理IP和端口号,进入代理网站选择一个IP:PORT
    proxy = {'http': '47.97.190.145:9999'}
    # 2.创建ProxyHandler
    proxy_handler = request.ProxyHandler(proxy)
    # 3.创建Opener
    opener = request.build_opener(proxy_handler)
    # 4.安装Opener
    request.install_opener(opener)

    # 现在如果访问url,就会使用代理服务器
    try:
        rsp = request.urlopen(url)
        html = rsp.read().decode()
        print(html)
    except error.URLError as e:
        print(e)
    except error.HTTPError as e:
        print(e)
    except Exception as e:
        print(e)
```

7.8 如果遇到不信任的 SSL 证书，如何继续访问?

比如访问某订票网址:

```
# 遇到不信任的SSL证书
# 如果要继续访问，则需要执行忽略处理
import ssl
from urllib import request

# 我们可以使用非认证的上下文环境替换认证的上下文环境,替换后就可以正常访问了
# 设置以下代码
ssl._create_default_https_context = ssl._create_unverified_context

url = 'https://kyfw.12306.cn/otn/leftTicket/init?linktypeid=dc'
rsp = request.urlopen(url)
html = rsp.read().decode()
print(html)
```

7.9 如何提取和使用本地已有的 cookie 信息?

- 使用 cookiejar
- 参考人人网登陆

```
# 自动使用cookie登陆的流程
# 打开登陆页面后自动通过用户密码登陆
# 自动提取反馈回来的cookie
```

```python
# 利用提取的cookie登陆隐私页面

from urllib import request, parse
from http import cookiejar

# 创建cookiejar的实例
cookie = cookiejar.CookieJar()
# 生成cookie的管理器
cookie_handler = request.HTTPCookieProcessor(cookie)
# 创建http请求管理器
http_handler = request.HTTPHandler()
# 生成https管理器
https_handler = request.HTTPSHandler()
# 创建请求管理器
opener = request.build_opener(http_handler, https_handler, cookie_handler)

# 初次登录,验证后给我们cookie
def login():
    '''
    负责初次登录
    需输入用户名和密码，用来获取cookie凭证
    '''
    # 登录用户地址，进入人人网登录首页，查看网页源码
    # 网页源码中打开查找，查找“下次自动登录”
    # 然后向上找form，里面就有提交表单的地址格式，login-form
    url = 'http://www.renren.com/PLogin.do'
    # 此键值需要从登录form的对应两个input中提取name属性
    data = {'email': '12345678@qq.com', 'password': '12345678'}
    # 把数据进行编码
    data = parse.urlencode(data)
    # 创建一个请求对象
    req = request.Request(url, data=data.encode())
    # 使用opener发起请求,会自动提取我的cookie
    rsp = opener.open(req)

def getHomePage():
    url = 'http://www.renren.com/574862780'
    # 如果已经执行了login,则opener则自动已经包含了相应的cookie值
    rsp = opener.open(url)
    # 读取网页的内容并进行解码
    html = rsp.read().decode()
    # 将打开的网页保存为html文件，然后浏览器打开
    with open('43_13_rsp.html', 'w') as f:
        f.write(html)

if __name__ == '__main__':
    # 初次使用用户名密码登陆后提取得到cookie
    login()
    # 使用获取的额cookie登陆个人主页
    getHomePage()
```

> 引用来源（CSDN 博客）： https://blog.csdn.net/weixin_30731305/article/details/95785019

7.10 requests 请求中出现乱码如何解决?

requests 库一般请求得到的内容都没有问题，但是有时候还是会出现乱码问题。

使用以下几行代码可以轻松解决问题：

- 方法 1：requests 会自动进行解码，并根据请求头部信息进行编码推测，当你访问 r.text 之时，requests 会使用其推测的文本编码。我们也可以使用 r.encoding 属性来指定编码。

```
import re
from bs4 import BeautifulSoup
import requests

url = 'https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E7%88%AC%E8%99%AB/5162711?
fromtitle=%E8%9C%98%E8%9B%9B&fromid=8135707'
user_agent = 'Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/67.0.3396.99 Safari/537.36'
headers = {'User-Agent': user_agent}
# 请求获取网址
r = requests.get(url, headers=headers)
r.encoding = 'utf-8' # 指定编码，解决中文乱码的问题
print(r.text)

soup = BeautifulSoup(r.text, 'html.parser', from_encoding='utf-8')
```

- 方法 2：使用 r.content，首先得到的是 bytes 类型，然后再解码为 str 类型

```
import requests

url='https://baike.baidu.com/item/%E7%BD%91%E7%BB%9C%E7%88%AC%E8%99%AB/5162711?
fromtitle=%E8%9C%98%E8%9B%9B&fromid=8135707'
r = requests.get(url)
html=r.content
html_doc=str(html,'utf-8')
print(html_doc)
```

7.11 requests 库中 response.text 和 response.content 的区别?

- response.text：返回的数据是字符串类型，编码是根据请求头自动推测，可能会出现错误
- response.content：返回的数据是字节类型，没有指定编码类型，需要使用 deocde 方法进行编码

猜测编码方式使用的是 chardet.detect 方法，实际开发更推荐使用 response.content.decode()。

```
# 本代码引用自作者本人博客

'''
Author: Felix
WX: AXiaShuBai
Email: xiashubai@gmail.com
Blog: https://blog.csdn.net/u011318077
Date: 2019/12/18 11:52
Desc:
'''

# -*- coding:utf-8 -*-
```

```
import requests

url = "https://www.baidu.com"
headers = {"User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:68.0)
Gecko/20100101 Firefox/68.0"}
response = requests.get(url=url, headers=headers)

print(response.status_code)

response.encoding = "utf-8"
print(type(response.text))

# - 解码类型:  没有指定,默认utf-8
# - 如何修改编码方式:response.content.decode("utf-8")
print(type(response.content))
print(type(response.content.decode()))
200
<class 'str'>
<class 'bytes'>
<class 'str'>
```

7.12 实际开发中用过哪些框架?

Scrapy 框架:

- 底层是 twisted 异步 IO 框架，Python 实现的爬虫框架
- 可以加入 request 和 beautifulsoup
- 强大的扩展功能
- 内置的 cssselector 和 xpath 解析器
- 默认深度优先

PySpider 框架

- 底层是 PyQuery 实现的
- 可以实现高并发的爬取数据，注意使用代理
- 提供了一个爬虫任务管理界面，可以实现爬虫的停止、启动、调试，支持定时爬取任务
- 扩展性较差
- 不需要太多自定义功能时用 PySpider 即可，一些定制性高的, 需要自定义一些功能时则使用 Scrapy

7.13 Scrapy 和 PySpider 框架主要有哪些区别?

- 原生的 Scrapy 并不支持 JS 渲染，需要单独下载 scrapy-splash 进行配置，而 PySpider 支持 phantomjs 第三方渲染
- PySpider 内置的 pyquery 选择器，Scrapy 支持的是 XPath 和 CSS 选择器
- Scrapy 全部采用命令行操作，PySpider 有较好的 WebUI，更加直观
- PySpider 易于调试，Scrapy 调试方式更加复杂，不直观
- Scrapy 扩展性更强，可以自定义功能，可以引入第三方库，PySpider 这方面稍微不足

7.14 Scrapy 的主要部件及各自有什么功能?

Scrapy 主要有 5 大部件和 2 个中间件。

- **Scrapy Engine（引擎）**：负责 Spider、ItemPipeline、Downloader、Scheduler 中间的通讯，信号、数据传递等
- **Scheduler（调度器）**：它负责接受引擎发送过来的 Request 请求，并按照一定的方式进行整理排列，入队，当引擎需要时，交还给引擎
- **Downloader（下载器）**：负责下载 Scrapy Engine（引擎）发送的所有 Requests 请求，并将其获取到的 Responses 交还给 Scrapy Engine（引擎），由引擎交给 Spider 来处理
- **Spider（爬虫）**：它负责处理所有 Responses，从中分析提取数据，获取 Item 字段需要的数据，并将需要跟进的 URL 提交给引擎，再次进入 Scheduler（调度器）
- **Item Pipeline（管道）**：它负责处理 Spider 中获取到的 Item，并进行进行后期处理（详细分析、过滤、存储等）的地方
- **Downloader Middlewares（下载中间件）**：你可以当作是一个可以自定义扩展下载功能的组件
- **Spider Middlewares（Spider 中间件）**：你可以理解为是一个可以自定扩展和操作引擎和 Spider 中间通信的功能组件（比如进入 Spider 的 Responses，和从 Spider 出去的 Requests）

> 图片及文字引用来源：https://blog.csdn.net/qq_37143745/article/details/80996707

7.15 描述一下 Scrapy 爬取一个网站的工作流程?

Scrapy 爬取一个网站的工作流程如下，结合 7.14 中的图片理解：

1. Scrapy Engine 引擎打开一个网站，爬虫先处理第一个要请求的 URL
2. 引擎从 Spider 拿到第一个要爬取的 URL 地址，并通过调度器以 Request 进行调度
3. 引擎向调度器获取下一个要爬取的 URL 地址
4. 调度器返回下一个要爬取的 URL 地址给引擎，引擎将 URL 地址通过下载中间件转发给下载器
5. 页面下载完毕以后，下载器就生成一个该页面的 Response 响应，通过下载中间件发送给引擎
6. 引擎从下载器中接收到 Response，并通过 Spider 中间件发送给 Spider 处理
7. Spider 处理 Response 并返回爬取到的 Item 及新的 Request 给引擎
8. 引擎将 Spider 返回的，爬取到的 Item 给 Item Pipeline，将 Request 给调度器
9. 从第二步开始重复，直到调度器中没有更多的 Request（等待爬取的 URL），引擎关闭该网站

7.16 Scrapy 中的中间件有什么作用?

- 常用的中间件：爬虫中间件 Spider Middleware 和下载器中间件 Downloader Middleware。
- 爬虫中间件：Spider 中间件是在引擎及 Spider 的特定工具, 处理 spider 的输入（response）和输出（item 及 requests）。
- 下载器中间件：下载器中间件是介于 Scrapy 的 request/response 处理的框架，是一个轻量、底层的系统，主要左右就是更换代理 IP，更换 Cookies，更换 User-Agent，自动重试。

7.17 Scrapy 项目中命名时候要注意什么?

爬虫项目文件夹命名后，里面的 py 文件

- 会自动创建以项目文件夹名称相关的类，名称的首字母会大写，
- 之后的字母会自动变成小写

爬虫文件及处理数据的主 py 文件，要换一个名称，不要和项目文件名相同。

爬虫主 py 文件中的爬虫 name 和爬虫 py 文件命区别开。

7.18 Scrapy 项目中的常用命令有哪些?

```
scrapy startproject（创建一个爬虫项目）
scrapy crawl XX（运行 XXX 爬虫项目）
scrapy shell http://www.hao123.cn（调试网址为 http://www.hao123.cn 的网站）
scrapy version 查看版本信息
scrapy list 查看爬虫信息，显示目录所有的爬虫
```

7.19 scrapy.Request() 中的 meta 参数有什么作用?

meta 参数

- meta 参数对应的值必须是一个字典
- 它的主要作用是用来传递数据的
- meta 是通过 Request 产生时传进去
- 通过 Response 对象中取出来

7.20 Python 中的协程阻塞问题如何解决?

采用协程访问三个网站：

- 由于 IO 操作非常耗时，程序经常会处于等待状态
- 比如请求多个网页有时候需要等待，gevent 可以自动切换协程
- 遇到阻塞自动切换协程，程序启动时执行 monkey.patch_all() 解决

参考案例：

```
from gevent import monkey; monkey.patch_all()
import gevent
from urllib import request
def run_task(url):
    print("Visit --> %s" % url)
    try:
        response = request.urlopen(url)
        data = response.read()
        print("%d bytes received from %s." %(len(data), url))
    except Exception:
        print("error")

if __name__ == '__main__':
    urls = ['https://github.com/', 'https://blog.csdn.net/',
'https://bbs.csdn.net/']
    # 定义协程方法
    greenlets = [gevent.spawn(run_task, url) for url in urls]
    # 添加协程任务，并且启动运行
    gevent.joinall(greenlets)

# 查看运行结果可以发现，三个协程是同时触发的，但是结束顺序不同
# 网页请求的时间不同，故结束顺序不同
# 但是该程序其实只有一个线程
Visit --> https://github.com/
Visit --> https://blog.csdn.net/
Visit --> https://bbs.csdn.net/
34697 bytes received from https://blog.csdn.net/
75062 bytes received from https://bbs.csdn.net/
79347 bytes received from https://github.com/
```

7.21 Scrapy 中常用的数据解析提取工具有哪些?

- 数据解析：lxml pyquery
- 数据提取：CSS、XPath、re 正则、BeautifulSoup4

7.22 描述一下 Scrapy 中数据提取的机制?

- Scrapy 有自己默认的提取机制，Selector 选择器，支持 xpath，css 和 re 正则表达式
- Selector 是建立在 lxml 库之上的，具有较快的解析速度和准确率
- Selector 解析结果有两种类型：SelectorList 和 Selector，前者是后者的集合，前者也是一个列表对象。我们可以使用 extract 和 extract_first 提取数据

7.23 Scrapy 是如何实现去重的？指纹去重是什么?

- Scrapy 配置文件里面有个 dont_filter 参数，设置 False 就是开启去重，默认值就是 False；
- 调度器会根据每一次的 URL 请求，对请求的相关信息进行加密，加密后的信息存储在一个集合之中，如果有新的请求，加密后和集合中已有的信息进行对比，如果已经存在，则改请求就淘汰，以此实现去重。

7.24 Item Pipeline 有哪些应用?

当 Item 在 Spider 被获取以后，数据会传递到 Item Pipeline，自己定义的 Pipeline 就会对所有的 Item 进行处理，提取出有用的数据。

Item Pipeline 的常用应用：

- 整理 HTML 数据
- 检测爬取的数据的合法性
- 检测 Item 是否包含某些字段
- 检测重复的数据并将其丢弃
- 保存需要的数据到文档或者数据库之中

7.25 Scrapy 中常用的调试技术有哪些?

- Parse 命令
- Scrapy shell 命令
- logging 日志

7.26 Scrapy 中有哪些常见异常以及含义?

异常	说明
DropItem	该异常由 item pipeline 抛出，用于停止处理 item
CloseSpider	该异常由 spider 回调函数 callback 抛出，来暂停或停止 spider，支持的参数：reason(str)：关闭的原因
IgnoreRequest	该异常由调度器 scheduler 或其它下载器中间件抛出，申明忽略该 request
NotConfigured	该异常由某些组件抛出，声明其仍然保持关闭，这些组件包括：Extensions、Item pipelines、Downloader middlwares、Spider middlewares
NotSupporter	该异常声明一个不支持的特性

7.27 Spider、CrawlSpider、XMLFeedSpider 和 RedisSpider 有什么区别?

- Spider：scrapy.Spider, 是所有 Spider 的基类，它是最基础的爬虫，所有的 spider 都会继承 scrapy.Spider。它提供了 start_requests() 的默认实现，读取并请求 spider 属性中的 start_urls，并根据返回的 response 调用 spider 中的 parse 方法。
- CrawlSpider：scrapy.spiders.CrawlSpider，规则爬虫，提供了一个新的属性 rules，该属性是一个包含一个或多个 Rule 对象的集合，每个 Rule 对爬取网站的动作定义了特定的规则。
- XMLFeedSpider：scrapy.spiders.XMLFeedSpider 设计用于通过迭代各个节点来分析 XML 源。
- RedisSpider：scrapy_redis.spiders.RedisSpider,scrapy-redis 是 Scrapy 框架基于 Redis 数据库的组件，用于 scrapy 项目的分布式开发和部署，可以方便的进行分布式爬取和数据处理。

7.28 scrapy-redis 是什么？相比 Scrapy 有什么优点?

- scrapy-redis 是 Scrapy 框架是以 Redis 数据库为基础的一个组件，主要用来分布式开发和部署。
- 相对于 Scapy 最大优点就是分布式爬取和分布式数据处理

7.29 使用 scrapy-redis 分布式爬虫，需要修改哪些常用的配置?

- 一般在配置文件中添加如下几个常用配置选项即可
- 1（必须）使用了 scrapy_redis 的去重组件，在 Redis 数据库里做去重

```
DUPEFILTER_CLASS = "scrapy_redis.dupefilter.RFPDupeFilter"
```

- 2（必须）使用了 scrapy_redis 的调度器，在 Redis 里分配请求

```
SCHEDULER = "scrapy_redis.scheduler.Scheduler"
```

- 3（可选）在 Redis 中保持 scrapy-redis 用到的各个队列，从而允许暂停和暂停后恢复，也就是不清理 Redis queues

```
SCHEDULER_PERSIST = True
```

- 4（必须）通过配置 RedisPipeline 将 item 写入 key 为 spider.name : items 的 Redis 的 list 中，供后面的分布式处理 item 这个已经由 scrapy-redis 实现，不需要我们写代码，直接使用即可

```
ITEM_PIPELINES = {
    'scrapy_redis.pipelines.RedisPipeline': 100 ,
}
```

- 5（必须）指定 Redis 数据库的连接参数

```
REDIS_HOST = '127.0.0.1'
REDIS_PORT = 6379
```

> 引用来源：https://www.cnblogs.com/Alexephor/p/11446167.html

7.30 常见的反爬虫措施有哪些？如何应对？

- headers 请求头反爬虫：伪造 headers，比如使用代理
- 基于用户行为反爬虫：使用动态的方法去爬取数据，模拟浏览器用户的行为
- 对于 IP 问题，我们可以自己搭建或者购买代理 IP 池
- 一些网站的动态页面：分析浏览器的 Ajax 请求，模拟 Ajax 请求, 也可以使用 Selenium 模块

7.31 BloomFitler 是什么？它的原理是什么？

- BloomFilter 是一种空间效率很高的随机数据结构，它利用位数组很简洁的表示一个集合，并能判断一个元素是否属于这个集合。
- BloomFilter 使用多个哈希函数来降低错误率。基本原理：通过 k 个 hash 函数将这个元素映射到位数组的 k 个点，将他们设置为 1。检索时，我们查看这 k 个点是否都为 1 就能够判断元素是否在 BloomFilter 中（会有一定的误差率）；如果 k 个点有一个点不为 1，那么这个元素肯定不在 BloomFilter 里面。

Python 中使用 BloomFilter

```
# -*- coding: utf-8 -*-

from pybloom import BloomFilter

# 创建一个一个容量为1000，漏失率为0.0001的布隆过滤器
f = BloomFilter(capacity=1000, error_rate=0.0001)
for x in range(100):
    f.add(x)
print(f)
# 判断数字是否在容器(数据库)中
print(110 in f)
print(24 in f)
<pybloom.pybloom.BloomFilter object at 0x000002AAE1F965F8>
False
True
```

7.32 为什么会用到代理？代码展现如何使用代理？

有些网址的反爬虫措施就是检测用户 IP 地址，如果一个 IP 地址频繁访问，就可能会判断为机器操作，而不是用户操作，IP 地址就会被封。此时我们就需要使用 IP 代理，模仿普通用户访问。

代理使用参考代码：

```
# 注意代码中IP代理可能已经失效，需要自己更换有效的代理地址
# -*- coding: utf-8 -*-
# 单个IP代理设置方式
import requests

proxies = {
    'http': '116.25.170.133:9999',
    'https': '116.25.170.133:9999',
}

url = 'https://www.hao123.com/'
response = requests.get(url, proxies=proxies)  #verify=False是否验证服务器的SSL证书

print(response.status_code)
print(response.headers)
# -*- coding: utf-8 -*-

#导入random，对ip池随机筛选
# 多ip代理模式
import requests
import random

proxy = [
{
    'http': 'http://61.135.217.7:80',
    'https': 'http://61.135.217.7:80',
    },
{
        'http': 'http://118.114.77.47:8080',
        'https': 'http://118.114.77.47:8080',
    },
{
        'http': 'http://112.114.31.177:808',
        'https': 'http://112.114.31.177:808',
    },
{
        'http': 'http://183.159.92.117:18118',
        'https': 'http://183.159.92.117:18118',
    },
{
        'http': 'http://110.73.10.186:8123',
        'https': 'http://110.73.10.186:8123',
    },
]

url = 'https://www.baidu.com/'

response = requests.get(url, proxies=random.choice(proxy))
```

7.33 爬取的淘宝某个人的历史消费信息（登陆需要账号、密码、验证码），你会如何操作?

- 使用内置浏览器，可以直接使用账号密码进行登录，验证码可以保存到本地手动输入验证码，或者可以使用多线程连接打码网站进行解决

- 基本上除了解压缩的参数，其他都要携带上，以防止反爬机制用到其他参数

7.34 网站中的验证码是如何解决的？

具体可以使用以下法方法：

- 使用 session 或者使用 cookie 解决。
- 使用上述方法无法解决的，可以使用 Selenium 模块，手动构造 cookie，该方法无需识别验证码，难点在于确定该网站是用 cookie 中的什么 key 值来表示“用户名”和“密码”的，需要多次请求进行分析，并且有些 cookie 是加密过，破解较困难。
- 最有效的解决办法就是直接输出，手动输入验证码，但是效率太低。
- 此外，现在有专门的打码平台，专门解决各种验证码的问题。

7.35 动态页面如何有效的抓取？

- 直接 URL 法：通过多次请求，分析 HTML 页面结构，使用 Firfox 或者 Chrome 开发者工具，分析每一次 JS 点击动作所发出的请求 URL，然后找出相应的规律。
- Selenium：模拟用户的操作，但是效率较低，大量抓取页面不可行。
- Scrapyjs 工具

7.36 如何使用 MondoDB 和 Flask 实现一个 IP 代理池？

实现逻辑：

- 爬取免费代理 IP 网站（西刺代理、快代理、66 代理等）的高匿代理
- 检测出有效的高匿代理，存储到 MongoDB 数据库中
- 定时检测 MongoDB 数据库已有 IP 代理是否有效，进行更新删除
- 使用 Flask 的 API 接口调用 IP 代理池中的 IP 代理

8. 数据分析及可视化

8.1 Python 数据分析通常使用的环境、工具和库都有哪些？库功能是什么？

- 基础环境一般为 Anaconda
- 编辑器多用 PyCharm、Ipython、JupyterNotebook

常用库有：

- NumPy：数值计算
- Matplotlib：数据可视化
- Pandas：数据预处理和数据分析
- scikit-learn：机器学习算法建模预测

8.2 常用的数据可视化工具有哪些？各自有什么优点？

- Matplotlib：Python 可视化程序库，是最基础的数据可视化库，具有很强的自定义功能。
- pygal：主要用于浏览器交互绘图，输出的图表是 SVG 格式，可以直接插入到网页中，可以动态显示数据。
- PyEcharts：百度公司推出的一个 Py 可视化工具，具有大量的在线模板，使用简单，可以直接复制进去数据，生成精美的图表。
- Plotly：和 PyEcharts 类似，可以快速生成精美的图表，并且生成的图表可以和用户进行交互。

8.3 数据分析的一般流程是什么？

基本流程：

- 提出问题（明确分析目的）
- 准备数据
- 分析数据
- 洞察结论

详细流程：

- 需求层
  - 目标确定
- 数据层
  - 数据获取
  - 数据规整（清洗）
- 分析层
  - 描述性分析
  - 指标计算
  - 数据可视化
  - 探索性分析
  - 建模分析
  - 模型验证
  - 迭代优化
- 输出层
  - 数据分析报告
  - 总结结论
  - 提出建议并实施

引用来源：《概率论与数理统计》

8.4 数据分析中常见的统计学概念有哪些？

- 相对数与绝对数：绝对数描述客观事物总体在一定时间和地点条件下的总规模，总水平的指标。相对数是指两个相关事物的比值。
- 百分数和百分点：百分数表示个体占总体的程度。百分点表示相同事物不同时期的增幅。
- 频数频率：频数是绝对数，频率是相对数。频数指某种事物或现象在其所在总体出里出现的次数。
- 比例和比率：比例指个体数值在总体数值中的占比。
- 倍数与番数：倍数是一个数除以另一个数所得的商。
- 同比与环比：同比是指与历史同时期进行比较得到的数值；环比是指与前一个统计期比较所得到的数值。

引用来源：《概率论与数理统计》

8.5 归一化方法有什么作用？

- 原始数据的量纲和数量级存在差异，直接对原始数据进行综合分析，此时数值较高的指标的作用会被放大，二数值较低的指标就会被弱化。针对该问题，我们就需要使用归一法。通过归一法对原始数据指标进行标准化处理。然后再进行综合分析。
- 通过归一化处理后的数据，应用模型进行处理的时候，可以大大提高模型的收敛速度，提高数据处理的效率。

8.6 常见数据分析方法论？

- PEST：PEST 是 Political Economic Social Technological 的缩写，主要用于研究宏观环境对具体对象的的影响。
- 5W2H：是以下 why、what、who、when、where、how、how much7 个单词的缩写，核心思想是以提问的方式进行数据分析，分别提出以上 7 个问题，然后逐一进行回答，寻找解决问题的方法。
- 逻辑树分析法：核心思想是问题不断分解，大问题分解为小问题，小问题又逐渐分解为小问题，逐层深入，依次解决各个问题，整个结构就像一个树一样。

8.7 如何理解欠拟合和过拟合?

- 欠拟合是数据挖掘的基本概念，但是采集的数据太少，机器学习时候，由于数据量太少，不能通过学习得到一定的规律，相当于机器学习不能形成自我认知的能力，就是欠拟合。
- 过拟合和欠拟合正好相反，由于一般是数据太有规律，通过机器学习训练的太好了，但是实际应用中，可能会出现一些未知的情况，此时就会产生偏差，甚至出现错误。所以机器学习的时候欠拟合和过拟合都是不合适的。因此数据挖掘学习的时候，我们应该适当的删除一些数据，使其更接近实际开发测试的环境。

8.8 为什么说朴素贝叶斯是“朴素”的?

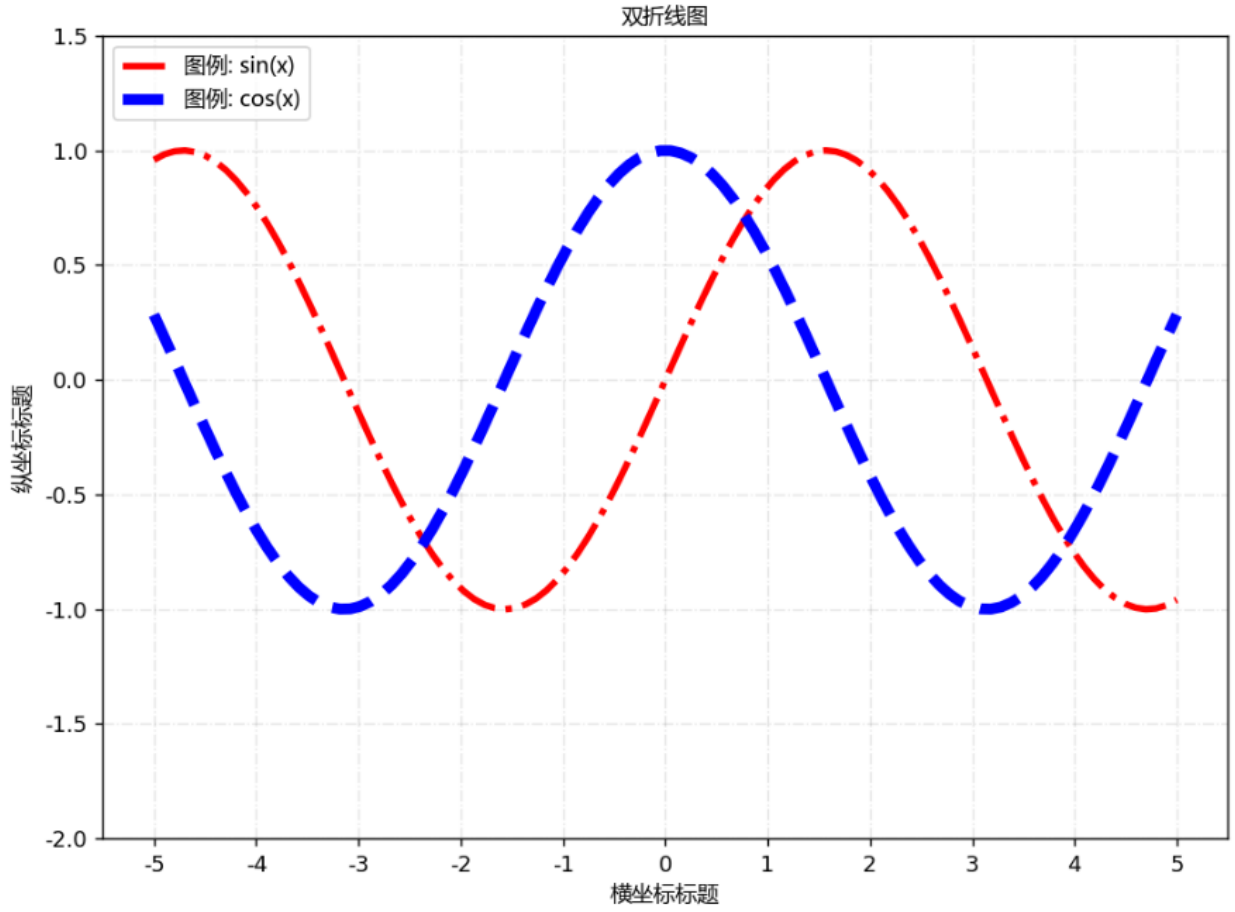
- 朴素贝叶斯是朴素的，主要是因为算法非常简单，但是准确率确非常高。
- 朴素贝叶斯依赖的算法也非常的简单，假设也容易理解，并没有多么高深晦涩的算法，这点来看也是朴素的。
- 此外，朴素贝叶斯算法不仅对简单的问题有效，遇到复杂的问题，一样结果准确有效。

8.9 Matplotlib 绘图中如何显示中文?

显示中文有两种方法，参考代码：

```
'''
Author: Felix
WX: AXiaShuBai
Email: xiashubai@gmail.com
Blog: https://blog.csdn.net/u011318077
Date: 2019/12/22 10:55
Desc:
'''

# -*- coding: utf-8 -*-

# 导入包
import matplotlib.pyplot as plt
import numpy as np
from matplotlib import font_manager

# 显示中文的两种方法:
# 方法1：加上下面两句代码，推荐使用，只需要设置一行即可，后面不需要像方法2一样多次传参数
# 使用以下方式，给键传入值，值为中文字体的英文名称：黑体=SimHei,宋体=SimSun
# plt.rcParams['font.sans-serif'] = ['SimHei']
plt.rcParams['axes.unicode_minus'] = False  # 用来正常显示负号

# 方法2：先实例化一个自定义字体格式，传入的参数为电脑系统中文字体的路径
# windows系统字体都放在Fonts里面，路径可以打开CMD窗口，拖一个中文字体到CMD窗口，
# 路径就出来了，按下Enter，会打开显示当前的字体
```

```python
# 注意，自定义中文字体后，下面需要显示中文的地方还需要传入fontproperties=my_font参数
my_font = font_manager.FontProperties(fname='C:\WINDOWS\Fonts\MSYH.TTC')

# 创建数据
x = np.linspace(-5, 5, 100)
y1 = np.sin(x)
y2 = np.cos(x)

# 创建figure窗口，指定窗口大小和分辨率
plt.figure(dpi=128, figsize=(8, 6))
# 画曲线1,其中color可以简写为c，linestyle可以简写为ls,推荐全写,label是图例
plt.plot(x, y1, label='图例: sin(x)', color='red', linewidth=3.0, linestyle='-.')
# 画曲线2
plt.plot(x, y2, label='图例: cos(x)', color='blue', linewidth=5.0, linestyle='--')

# 设置坐标轴范围,起点和端点
# plt.xlim((-5, 5))
# plt.ylim((-2, 2))

# 设置图表名称和坐标轴名称
plt.title("双折线图", fontsize=24, fontproperties=my_font)
plt.xlabel('横坐标标题', fontsize=14, fontproperties=my_font)
plt.ylabel('纵坐标标题', fontsize=14, fontproperties=my_font)

# 设置坐标轴刻度，刻度间隔
my_x_ticks = np.arange(-5, 5.5, 1)
my_y_ticks = np.arange(-2, 2.0, 0.5)
plt.xticks(my_x_ticks)
plt.yticks(my_y_ticks)

# 显示网格线，显示图例，注意该处显示中文的参数是prop，alpha透明度
plt.grid(axis='both', color='grey', linestyle='-.', alpha=0.2)
plt.legend(prop=my_font, loc='upper left')

# 显示出所有设置
plt.show()
```



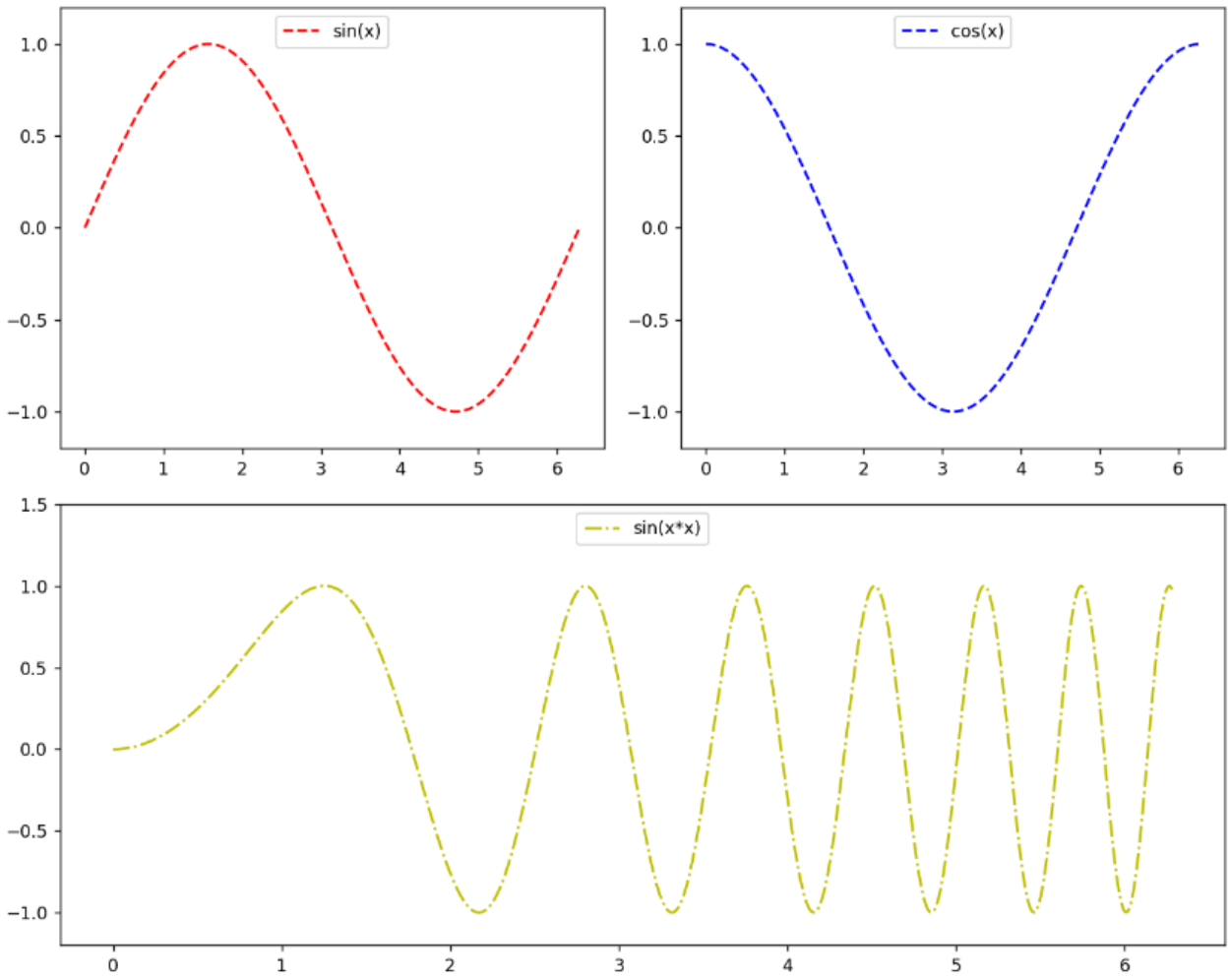
以上绘图案例参考 matplotlib 官方文档中案例。

8.10 Matplotlib 中如何在一张图上面画多张图?

参考代码:

```
'''
Author: Felix
WX: AXiaShuBai
Email: xiashubai@gmail.com
Blog: https://blog.csdn.net/u011318077
Date: 2019/12/25 13:55
Desc:
'''

# -*- coding:utf-8 -*-

import numpy as np
import matplotlib.pyplot as plt
#创建自变量数组
x= np.linspace(0,2*np.pi,500)
#创建函数值数组
y1 = np.sin(x)
y2 = np.cos(x)
y3 = np.sin(x*x)

#创建图形
plt.figure(figsize=(10, 8), dpi=128)
```

```
'''
多图形绘制需要设置图形所在位置：
subplot(numRows, numCols, plotNum)
前两个参数自动将图形分隔为几行几列，
第三个参数为从左上角开始，从左到右，从上到下图形所在的位置
如果有一行只有一列，那么其它行不同列也当做一个整体，就是一列
比如下面：subplot(2,2,1),分成2行2列的第1个元素
subplot(2,2,2),分成2行2列的第2个元素
subplot(2,1,2),分成2行1列的第2个元素,第一行所有列当做一个整体
'''

# 设置绘制图形的位置
#第一行第一列图形
ax1 = plt.subplot(2,2,1)
#第一行第二列图形
ax2 = plt.subplot(2,2,2)
#第二行的一个图形
ax3 = plt.subplot(2,1,2)

# 绘制ax1,先传入图形位置
plt.sca(ax1)
# 绘制红色曲线
plt.plot(x, y1, label='sin(x)', color='red', linestyle='--')
# 限制y坐标轴范围
plt.ylim(-1.2,1.2)
# 显示图例
plt.legend(loc='upper center')

# 绘制ax2
plt.sca(ax2)
# 绘制蓝色曲线，线条颜色和风格可以使用以下缩写方式
plt.plot(x, y2, 'b--', label='cos(x)')
plt.ylim(-1.2,1.2)
# 显示图例
plt.legend(loc='upper center')

# 绘制ax3
plt.sca(ax3)
plt.plot(x, y3, 'y-.', label='sin(x*x)')
plt.ylim(-1.2,1.5)
# 显示图例
plt.legend(loc='upper center')

plt.show()
```



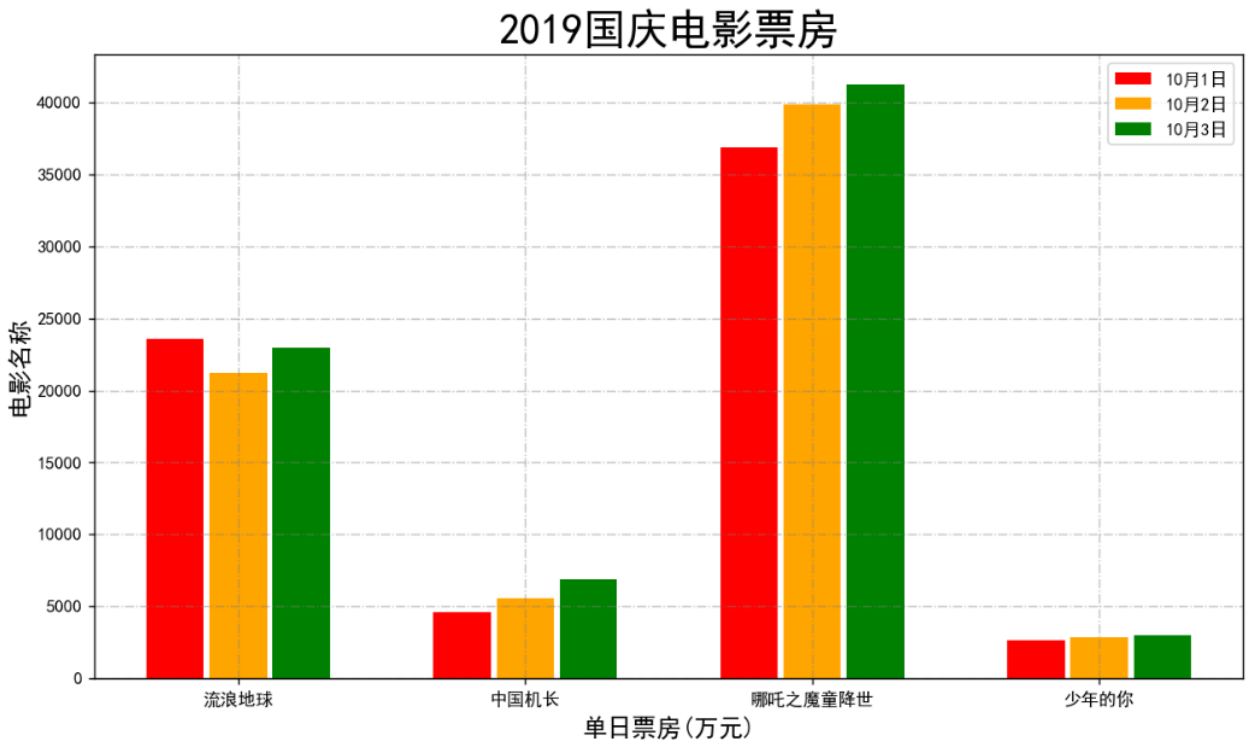
引用来源（CSDN 博客）：https://blog.csdn.net/chengqiuming/article/details/78601382

8.11 使用直方图展示多部电影 3 天的票房情况?

参考代码：

```
# -*- coding: utf-8 -*-

# 导入包
import matplotlib.pyplot as plt

# 显示中文和显示负号
plt.rcParams['font.sans-serif'] = ['SimHei']
plt.rcParams['axes.unicode_minus'] = False

# X轴和Y轴数据,三天电影票房情况，票房单位万元
movie = ["流浪地球","中国机长","哪吒之魔童降世","少年的你"]
y_1 = [23585,4566,36898,2620]
y_2 = [21235,5598,39852,2835]
y_3 = [22986,6895,41250,2985]

# 设置图形的大小
plt.figure(figsize=(10, 6), dpi=128)

# 设置X轴的坐标数据,条形图宽度为0.2,先设置1号的数据，绘制2号，然后3号
# X轴坐标向右移动0.02即可，也可以多移动一点，保持一点间隙
```

```
bar_width = 0.2
x_1 = list(range(len(movie)))
x_2 = [i + 0.02 + bar_width for i in x_1]
x_3 = [i + 0.02 + bar_width for i in x_2]

# 竖直条形图，用的是width设置宽度，x轴参数是一个可迭代对象，一般为列表
# 分别绘制三天的电影票房，绘制三次即可
plt.bar(x_1, y_1, width=bar_width, label='10月1日', color='red')
plt.bar(x_2, y_2, width=bar_width, label='10月2日',color='orange')
plt.bar(x_3, y_3, width=bar_width, label='10月3日',color='green')

# 将X轴转换为对应的字符串，只对应中间的，字就在正中间
plt.xticks(x_2, movie, rotation = 0)

# 设置图片，X轴，Y轴标题
plt.title("2019国庆电影票房", fontsize=24)
plt.xlabel("单日票房(万元)", fontsize=14)
plt.ylabel("电影名称", fontsize=14)

# 设置网格,显示图例
plt.grid(axis='both', color='grey', linestyle='-.', alpha=0.5)
plt.legend(loc='upper right')

# 显示图形
plt.show()
```



8.12 描述一下 NumPy array 对比 Python list 的优势?

- NumPy array 数据存储时候，所需要的空间更小。读取和写入的速度相对于 Python 中的更快。
- NumPy array 中的数据都是同一个类型，list 里面的数据类型可以不同。array 数据在内存中就是连续的，list 不一定是连续的。这就是 array 占用资源更少速度更快的原因之一。
- 在 NumPy 中的 array，可以使用 vector 和 matrix 类型的处理函数，处理数据速度更快更加的方便。

8.13 数据清洗有哪些方法?

- 数据清洗遵循的原则：数据的完整性、数据的全面性、数据的合法性、数据的唯一性。
- 缺失数据的处理：我们首先判断缺失数据是否有价值，如果缺失数据没有价值，直接对其删除；如果缺失的数据有意义，我们可以根据已有的数据进行推导，比如我们可以分析数据到的平均值，最大值，最小值，进行合理的补充。
- 错误值或者异常值的处理：我们可以分析观察整个数据的规律，对于明显的错误值或者异常值，进行清理去除。
- 重复数据的处理：通过分析检测数据，如果有明显重复的数据记录，我们可以对数据进行合并或者去除处理。
- 不一致数据的处理：一般我们对某个字段的数据都会进行约束，可能是类型约束、范围约束等等，如果数据中出现了明显的不一致数据。对于该部分异常的数据就需要进行去除。

注：本文部分文字内容和代码引用于网络公开博文、百度百科和出版图书，已标注引用来源。