Table of Contents

Python_book

第一章

第1节：概述

版本：Python3.x 跟着官方趋势走，与2不兼容（差别特别大）

python特点

1、Simple：
Python语法简单、格式优美，写程序让人有种写作文的感觉，所以显得简单
2、Easy to learn：
容易学习，通常解析型语言都比编译型语言容易学习。所谓静态语言和动态语言很重要的一点就是，声明变量是否需要指定其数据类型。
3、Free and Open Source:
免费和开源，一旦你要发布你的程序，那么你的源代码就会随着一起发布
4、High-level Language：
高层语言也就意味着你能够更简单和轻松地实现一些功能，而不需要考虑程序本身底层的一些细节（比如内存操作等）
5、Portable：
可移植性，由于其开源在很多平台上都得到了支持，比如Linux，mac os 都默认安装了python解释器
6、Interpreted：
解析型语言，如C/C++语言的执行时要经过编译成机器码才能执行，而Python等动态语言是直接由解释器所解释执行的。通常编译型语言执行速度非常快，解释型语言相对比较慢（开发效率刚好相反）
7、Object Oriented：
面向对象特性也是Python的一大特点，当然python也可以面向过程使用
8、Extensible：
可扩展性，如果你需要你的一段关键字代码运行得更快或者希望某些算法不公开，你可以把你的部分程序用C或C++编写，然后再你的python程序中使用它们
9、Embeddable：
你可以把python嵌入你的C/C++程序，从而向你的程序用户提供脚本功能

第**2**节：定位应用

1、**web**开发（**flask/Django/Tornade**）

2、科学计算**/**数据分析**/**算法学习（**Numpy/Scipy**）

3、机器学习（**Scikit-Learn**）

4、网络爬虫（**Scrapy/BeautifulSoup**）

5、图片处理**/**游戏开发（**Pillow**）

6、运维**/**测试自动化开发（**saltstack**）

第3节：开启虚拟环境

开启虚拟环境（局部环境）（会污染全局）

```
python -m venv tutorial-env （创建虚拟环境，多了一个文件夹）

tutorial-env\Scripts\activate.bat（运行虚拟环境，在命令行也可以）

import sys   # 文件系统

print(sys.path)  # 保留python的路径    执行以上两行看到已经把虚拟环境加进来了，可以开始正式地编程了！每一个项目都需要开启自己的一个虚拟环境。
```

第4节：pip（包管理工具操作）

- pip install novas（下载包裹）机器学习、
- pip install requests == 2.6.0（下载版本）
- pip uninstall novas
- pip show flask（显示包裹信息）
- pip list（显示所有包裹）
- pip freeze > requirements.txt 将安装的包裹列表输出到requirements.txt中（生成了一个提示文件）,以便执行pip install -r requirements.txt（虚拟环境被删掉，只下载了项目代码，自己创建回原虚拟环境（tutorial-env\Scripts\activate.bat）。执行pip install -r requirements.txt（下载回之前输到txt文件的包裹），然后pip list 查看时就有了之前被删掉的安装的包裹。）

> 上传到github时，不上传虚拟环境：删掉这个虚拟文件，在项目文件上右击新建一个文件名为.gitignore，在新建的文件里面写上tutorial-env（忽略的文件名，以便后续安装回该虚拟环境文件夹），VCS里的git里的add直接发布，或者在命令行直接写命令一直到git push推上去，进github主页里就没有那个文件了。

第5节：输入输出测试工具

```
输入工具  编辑器注释ctrl+/
内建函数 input获取的数据都是字符串类型
string=input("请输入你的名字")
print(type(string))

输出工具
print("123")
print("123","456",end="")
print("123",456,1+2)

格式化打印  字符串用法  %s占位
print("我叫%s,我的年龄是%d,我的成绩%0.2f"%('wss',20,60.09))
print("我叫%s"%'wss')
```

第6节：变量

变量的定义

可变的数据

变量的语法

```
python赋值变量

1 普通方式
a = 10

2 链式方式
b=c=d=20

3 多元方式
e,f=10,20

交换方式，变量直接绑定下来，然后依次赋值
e,f = f,e
print(a,b,c,d,e,f)
```

变量命名的特点

1. 不可以是关键字和保留字
2. 数字、字母、下划线、汉字，数字不能开头
3. 尽量语义化
4. 严格区分大小写
5. 不能使用特殊符号

变量的特性

6. 不支持自增自减
7. 变量可以存储不同的数据类型

第7节：数据类型

js与python数据类型的对比

```
js：数字、字符串、boolean、null、undefined（声明未定义）、object
python：数字、字符串、布尔、null、列表（数组）元组、字典（object\json）集合

js分类：基本数据类型和引用数据类型
python分类：可变数据类型和不可变数据类型（全部是引用的）

可变数据类型：列表、字典、集合  其余都是不可变的
```

布尔型boolean

```
非空非0为真，0或None为假
Ture   False
```

数字型number

```
1.整数int
整数的最大值取决于电脑内存的大小，没有上限

2.浮点数float
只要有小数点就是浮点数，无穷小数会做精度的处理（除运算也会做精度的处理）
    0.1
    5.0

数字精度：内建函数、模板字符串（控制小数精度）

3.负数

4.分数
```

字符串 str

语法

```
1、单引号
2、双引号
    换行的时候加 \
```

3、三引号
    三引号''''''(支持换行、块级注释) """""""(注释)

注意

只要是用引号括起来的就是字符串
python不区分字符和字符串
字符串是不可变对象

延展

转义字符 “\”
使用特殊字符时就需要转义字符
常用的转移字符：\n \t \r * \\
字符编码
    ASCII
        起源于美国，主要对英文进行编码
    GBK：中国的字符编码
    Unicode：万国码
    utf-8：更精确的万国码
前缀
    r“str”r后的字符串为普通字符串
    b“str”b后的字符串为bytes格式
    u“str”u后的字符串为Unicode格式

重要用法

切片(截取)
切片功能

```
a = "山西优逸客科技有限公司"
print(a[2:5])
```

切片语法 string[start:end:step] [初始值：结束值：步进值] 所有的值都可以是负值或空
默认:start == 0（前开始、后开始） end == 结尾（前结尾、后结尾） step == 1 如果为-1,就是改变方向

```
print(a[-1])
print(a[-2:])
print(dir(a))
```

模仿方法（面向对象） 字符串内建函数（方法）自己练习

字符串高级

字符串内键函数 print（dir（aa）） 会打印出一个列表（数组），里面有双前后下划线的方法称为模板方法（面向对象），把其余的字符串内建函数（字符串的方法）练习一下。

string.capitalize() 把字符串的第一个字符大写
string.center(width) 返回一个原字符串居中,并使用空格填充至长度 width 的新字符串
string.count(str, beg=0, end=len(string)) 返回 str 在 string 里面出现的次数，如果 beg 或者 end 指定则返回指定范围内 str 出现的次数
string.decode(encoding='UTF-8', errors='strict') 以 encoding 指定的编码格式解码 string，如果出错默认报一个 ValueError 的异常，除非 errors 指定的是 'ignore' 或者'replace'
string.encode(encoding='UTF-8', errors='strict') 以 encoding 指定的编码格式编码 string，如果出错默认报一个ValueError 的异常，除非 errors 指定的是'ignore'或者'replace'
string.endswith(obj, beg=0, end=len(string)) 检查字符串是否以 obj 结束，如果beg 或者 end 指定则检查指定的范围内是否以 obj 结束，如果是，返回 True,否则返回 False.
string.expandtabs(tabsize=8) 把字符串 string 中的 tab 符号转为空格，tab 符号默认的空格数是 8。
string.find(str, beg=0, end=len(string)) 检测 str 是否包含在 string 中，如果 beg 和 end 指定范围，则检查是否包含在指定范围内，如果是返回开始的索引值，否则返回-1
string.format() 格式化字符串
string.index(str, beg=0, end=len(string)) 跟find()方法一样，只不过如果str不在 string中会报一个异常.
string.isalnum() 如果 string 至少有一个字符并且所有字符都是字母或数字则返回 True,否则返回 False
string.isalpha() 如果 string 至少有一个字符并且所有字符都是字母则返回 True,否则返回 False
string.isdecimal() 如果 string 只包含十进制数字则返回 True 否则返回 False.
string.isdigit() 如果 string 只包含数字则返回 True 否则返回 False.
string.islower() 如果 string 中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是小写，则返回 True，否则返回 False
string.isnumeric() 如果 string 中只包含数字字符，则返回 True，否则返回 False
string.isspace() 如果 string 中只包含空格，则返回 True，否则返回 False.
string.istitle() 如果 string 是标题化的(见 title())则返回 True，否则返回 False
string.isupper() 如果 string 中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是大写，则返回 True，否则返回 False
string.join(seq) 以 string 作为分隔符，将 seq 中所有的元素(的字符串表示)合并为一个新的字符串
string.ljust(width) 返回一个原字符串左对齐,并使用空格填充至长度 width 的新字符串
string.lower() 转换 string 中所有大写字符为小写.
string.lstrip() 截掉 string 左边的空格
string.maketrans(intab, outtab)
maketrans() 方法用于创建字符映射的转换表，对于接受两个参数的最简单的调用方式，第一个参数是字符串，表示需要转换的字符，第二个参数也是字符串表示转换的目标。
max(str) 返回字符串 str 中最大的字母。
min(str) 返回字符串 str 中最小的字母。
string.partition(str) 有点像 find()和 split()的结合体,从 str 出现的第一个位置起,把 字 符 串 string 分 成 一 个 3 元 素 的 元 组 (string_pre_str,str,string_post_str),如果 string 中不包含str 则 string_pre_str == string.
string.replace(str1, str2, num=string.count(str1)) 把 string 中的 str1 替换成 str2,如果 num 指定，则替换不超过 num 次.
string.rfind(str, beg=0,end=len(string)) 类似于 find()函数，不过是从右边开始查找.
string.rindex(str, beg=0,end=len(string)) 类似于 index()，不过是从右边开始.
string.rjust(width) 返回一个原字符串右对齐,并使用空格填充至长度 width 的新字符串
string.rpartition(str) 类似于 partition()函数,不过是从右边开始查找
string.rstrip() 删除 string 字符串末尾的空格.

```
string.split(str="", num=string.count(str)) 以 str 为分隔符切片 string，如果 num 有指定值，则仅分隔 num+ 个子字符串
string.splitlines([keepends]) 按照行('\r', '\r\n', \n')分隔，返回一个包含各行作为元素的列表，如果参数 keepends 为 False，不包含换行符，如果为 True，则保留换行符。
string.startswith(obj, beg=0,end=len(string)) 检查字符串是否是以 obj 开头，是则返回 True，否则返回 False。如果beg 和 end 指定值，则在指定范围内检查.
string.strip([obj]) 在 string 上执行 lstrip()和 rstrip()
string.swapcase() 翻转 string 中的大小写
string.title() 返回"标题化"的 string,就是说所有单词都是以大写开始，其余字母均为小写(见 istitle())
string.translate(str, del="") 根据 str 给出的表(包含 256 个字符)转换 string 的字符,要过滤掉的字符放到 del 参数中
string.upper() 转换 string 中的小写字母为大写
string.zfill(width) 返回长度为 width 的字符串，原字符串 string 右对齐，前面填充0
```

None

列表List

定义

保存一系列相关数据的集合

语法

```
arr = [1,2,3,4,5]
访问列表
print(arr[0])
```

访问列表

```
arr[index]
```

常用的遍历列表的方式

```
 方法1

    arr = [[1,2],[3,4]]
    for i in arr:
        print(i)
        print(arr.index(i),i)
```

```
range()内建函数：
1个参数：结束的值  会返回包含0-n之间的数的序列
print(list(range(10)))    #强制转换成列表
2个参数：开始的位置，结束的位置。返回包含开始-结束之间的序列
print(list(range(5,10)))
3个参数：开始的位置，结束的位置，步进值。
```

方法2

```
arr = [[1,2],[3,4]]

for i in range(len(arr)):
    print(i,arr[i])
```

方法3

```
arr = [[1,2],[3,4]]

enumerate() 返回列表元素的下标和值
print(list(enumerate(arr)))
for i,v in enumerate(arr):
    print(i,v)
```

方法4

```
赋值 枚举类型去遍历
a,b,c,d,e = (1,2,3,4,5)
```

列表的长度

```
获取列表长度
print(len(arr[-1]))
```

列表的深拷贝和浅拷贝

```
深拷贝浅拷贝
arr1 = arr       #传址
print(id(arr1),id(arr))
a = 10
b = a
print(id(a),id(b))
```

```
arr1[0] = 'a'
print(arr1)
print(arr)

import copy      #导入内置包    （ctrl+点击，进去包里看包里的源码）
arr1 = copy.copy(arr)    #copy浅拷贝，把arr传过来   （值相等，地址不相等）
arr1 = arr.copy()   #浅拷贝   （值相等，地址也相等）
print('arr1',id(arr1),arr1)
print('arr',id(arr),arr)

import copy
arr = [[1,2],[3,4]]
arr1 = copy.deepcopy(arr)   #深拷贝（多维数组）   （值相等，地址相等）
arr1[0][0] = 'a'   #原来的arr不变    （地址不相等）
print('arr1',id(arr1),arr1)
print('arr',id(arr),arr)
```

列表高级

推导式

```
列表推导式：快速生成新的列表
推导式的分类：
1 列表推导式   2 字典推导式   3 集合推导式
1 列表推导式
arr = [1,2,3,4,5]
语法：
1 元素幂运算  in后是要遍历的内容,for后是要的东西,最前面的要的结果,
arr2 = [ item*item for item in arr ]
print(arr2)
2 输出偶数,并幂运算
arr2 = [ item*item for item in arr if item%2==0]
print(arr2)
3 输出1-100的偶数
arr2 = list(range(1,101))
arr3 = [ item for item in arr2 if item%2==0]
print(arr3)
```

列表的内建函数

```
List内建函数
1、L.append(var)   #追加元素
2、L.insert(index,var)
3、L.pop(var)      #返回最后一个元素，并从list中删除之
4、L.remove(var)   #删除第一次出现的该元素
5、L.count(var)    #该元素在列表中出现的个数
```

```
6、L.index(var)    #该元素的位置,无则抛异常
7、L.extend(list)  #追加list，即合并list到L上
8、L.sort()        #排序
9、L.reverse()     #倒序
list 操作符:,+,*，关键字del
a[1:]       #片段操作符，用于子list的提取
[1,2]+[3,4] #为[1,2,3,4]。同extend()
[2]*4       #为[2,2,2,2]
del L[1]    #删除指定下标的元素
del L[1:3]  #删除指定下标范围的元素
```

元组 **tuple**

定义

不可改变的列表，不能修改

语法

```
语法：

t1 = ()
print(type(t1))
t2 = (1,)
print(type(t2))
面试题  数组的id地址不变就是一个数组
t1 = ([1,2],[3,4])
t1[0][0] = 2
print(t1)
```

注意

定义一个元组得加逗号，不然定义的就是int型

```
1、元组和列表类似，区别就是元组的元素不可以改变
2、元组通过圆括号定义
3、可以使用切片
4、元组可以定义只有一个元素的元组
    t2 = (1,)
    print(type(t2))
5、空元组   mytuple = ()   #tuple类型
```

访问元组：同列表

便利元组：同列表

元组高级

元组的内建函数

```
1.cmp(tuple1, tuple2)  比较两个元组元素。
2.len(tuple)  计算元组元素个数。
3.max(tuple)  返回元组中元素最大值。
4.min(tuple)   返回元组中元素最小值。
5.tuple(seq)  将列表转换为元组。

tuple.index(obj,end=len(tuple))
tuple.count(obj)   #返回obj出现
```

字典 **dict**（对象，**json**）

语法

创建方式

```
（1）json方式
dict1 = {1:123,'name':'wss',(1,2,3):'abc'}
访问
print(dict1[1])
print(dict1['name'])
print(dict1[(1,2,3)])
（2）通过内建函数dict()创建
dict2 = dict()    #创建了空字典
设置键值
dict2[1] = 123
dict2['name'] = 'wss'
dict2[(1,2,3)] = 'abc'
访问
print(dict2)
（3）批量创建键值
dict3 = dict.fromkeys([1,2,3,4,5],'a')
print(dict3)
```

注意:

1、字典以键值对形式存在：**{key:value}**

2、查找和插入的速度极快，不会随着**key**的增加而增加

3、一些可变的数据类型不可做**key**值，而**value**可以是任意数据

4、字典中数据元素是无序的，没有办法进行向索引和切片操作

5、字典是可变数据类型

> 字典的键需要进行哈希运算，所以键是不可变数据类型（只要键能进行哈希运算，所有内容都可以作为键【字符串、数字型、变量（存的值是不可变的数据类型能够进行哈希运算）】

字典访问方式

- mylist[键]

添加属性和方法

- mylist['newKey'] = value

删除字典属性和方法

- del

```
删除字典的数据以及字典  通过del
del dict3[2]
print(dict3)
del dict3
print(dict3)
```

字典的遍历

- for in ``` 遍历字典 python里只有for in 1 for k in dict3: print(k) print(dict3[k])

2 print(dict3.items()) for k,v in dict3.items(): print(k,v)

3 arr = zip([1,2,3,4],['a','b','c','d'],[11,22,33,44]) # 将传入的列表一一对应 print(list(arr)) dict3.keys() # 返回键 dict3.values() #返回值 for k,v in zip(dict3.keys(),dict3.values()): print(k,v)

```
### 字典高级
#### 字典的内建函数
```

字典dictionary内建函数 1、D.get(key, 0) #同dict[key]，多了个没有则返回缺省值，0。[]没有则抛异常 2、D.has_key(key) #有该键返回TRUE，否则FALSE 3、D.keys() #返回字典键的列表 4、D.values() #以列表的形式返回字典中的值，返回值的列表中可包含重复元素 5、D.items() #将所有的字典项以列表方式返回，这些列表中的每一项都来自于(键,值),但是项在返回时并没有特殊的顺序

6、D.update(dict2) #增加合并字典 7、D.popitem() #得到一个pair，并从字典中删除它。已空则抛异常 8、D.clear() #清空字典，同del dict 9、D.copy() #拷贝字典 10、D.cmp(dict1,dict2) #比较字典，(优先级为元素个数、键大小、键值大小)

```
                    #第一个大返回1，小返回-1，一样返回0
```

11、D.pop(key[,default]) #删除字典给定键 key 所对应的值，返回值为被删除的值。key值必须给出。 否则，返回default值。 12、dict.fromkeys(seq[, val]) #创建一个新字典，以序列 seq 中元素做字典的键，val 为字典所有键对应的初始值

dictionary的复制 dict1 = dict #别名 dict2=dict.copy() #克隆，即另一个拷贝。

```
#### 字典推导式
```

字典的推导式 d = {'a':1,'A':10,'b':2,'B':20} 1 d2 = { k:str(v)+'m' for k,v in d.items()} print(d2) 2 d3 = { k:str(v + d[k.upper()])+'px' for k,v in d.items() if k.islower()} print(d3)

```
#### 深拷贝、浅拷贝
#### collection  一个更强大的dict包
## 集合 set（结合了字典和列表）
### 定义
    set()    #把其他内容转化为集合
### 集合高级
#### 集合推导式
```

集合推导式 法1 s1 = {1,2,3,4,5,6} s2 = { item+1 for item in s1} print(s2) 法2 s2 = { item for item in s1 if item%2==0} print(s2)

```
#### 集合内建函数
```

add() 为集合添加元素 clear() 移除集合中的所有元素 copy() 拷贝一个集合 difference() 返回多个集合的差集 difference_update() 移除集合中的元素，该元素在指定的集合也存在。 discard() 删除集合中指定的元素 intersection() 返回集合的交集 intersection_update() 删除集合中的元素，该元素在指定的集合中不存在。 isdisjoint() 判断两个集合是否包含相同的元素，如果没有返回 True，否则返回 False。 issubset() 判断指定集合是否为该方法参数集合的子集。 issuperset() 判断该方法的参数集合是否为指定集合的子集 pop() 随机移除元素 remove() 移除指定元素 symmetric_difference() 返回两个集合中不重复的元素集合。 symmetric_difference_update() 移除当前集合中在另外一个指定集合相同的元素，并将另外一个指定集合中不同的元素插入到当前集合中。 union() 返回两个集合的并集 update() 给集合添加元素

```
```
通过set集合元素互异性进行列表的去重
arr1 = [1,1,2,2]
```
```

```
print(list(set(arr1)))
```

测试数据类型

type

第8节：运算符

算数运算符

```
+ - * /   %   //地板除    **幂运算

(1)+  加法运算：算数运算，字符串拼接作用    str  list  tuple
print("str1" + "str2")
print([1,2,3] + [4,5,6])
print((1,2,3) + (4,5,6))
(2)*  乘法运算：用法同加法
print('srt1'*2)
print([1,2]*2)
print((1,2)*2)
(3)/  除法的运算结果都是浮点型
print(10/5)
(4)//  地板除:向下取整
print(14//5)
(5)**  幂运算
print(2**3)
```

逻辑运算符

```
and  or  not  is  isnot（判断是否为同一个对象）

区分is 、= =：is 判断是否为同一个对象（就一个）

面试题（is和==；可变、不可变存储方式的理解）
arr1 = [1,2,3]
arr2 = arr1
print(arr1==arr2)
print(arr1 is arr2)    # 是同一个（传址，谁可以用这个数组）

arr1 = [1,2,3]
arr2 = [1,2,3]
print(arr1==arr2)
print(arr1 is arr2)    # 不是同一个

a = 10
b = a
print(a==b)
print(a is b)
```

```
a = 10
b = 10
print(a==b)
print(a is b)
```

关系运算符

```
<  <= >  >=  !=   = =
```

位运算符

```
~a 按位取反   a<<n:左移n位   a>>n:右移n位   &：按位与   |: 或运算   ^:异或运算
二进制、八进制、十进制、十六进制

print(0b10)   #输出二进制
FFFFFF    每两位表示一个数
FF
F*16^1+F*16^0 = 255
rgb(255,255,255)

~: 按位取反
print(~0b10)
print(~10)
原码   反码   补码

a<<n:左移n位      每移一位乘以2（扩大2倍）
print(2<<1)
10 ->  100

&：按位与    （同真为真，有假为假）
print(2&3)
10
11

|: 或运算    （有真为真）
print(2|3)

^:异或运算  (相同为0，不同为1)
print(2^3)
```

赋值运算符

```
=  +=  -=  *=  /=  //=   %=  **=
```

```
a = 10
a/=5   #a = a/5
print(a)

a = 10
a //=3   #a = a//3
print(a)

a = 2
a**= 3   #a = a**3
print(a)
```

一元运算符 （**del** 删除字典）

```
一元运算符   del

arr = [1,2,3]
del arr
print(arr)
```

三元运算符

```
三元运算符
a,b = 10,20
print("a>b")if a>b else print("b>a")
```

第9节：流程控制

分支（必须写内容）

```
 if
 a,b=10,20
 if a>b:
     print("a>b")
 elif a==b:
     print("a=b")
 else:
     print("a<b")

 pass   占位
"""
if(a>b){

}
"""
```

循环

```
（1）for
range()序列
for i in range(1,11):
    print(i)

（2）while
a1,b1=10,20
while a1<b1:
    print(a1)
    a1+=1

for 与 while
是否明确循环次数

(3)干预循环
        # 猜数字
# import  random
# num1 = random.randint(1,100)
# while True:
#     num2 = int(input("请输入一个0~100之间的数：\n"))
#     if num2>num1:
```

```
#         print("数字过大")
#     elif num2==num1:
#         print("恭喜你猜对了")
#         break
#     else:
#         print("数字过小")
```

其他

列表推导式

```
列表推导式
"""
js中:
let arr = []
for(let i=0;i<11;i++){
    arr.push('a')
}
"""
 arr = [i for i in range(1,10)]
 arr = ['a' for i in range(1,10)]
```

列表转字符串

```
列表转字符串    jion把列表里的所有字符串进行拼接
print(",".join(['1','2','3','4']))

99乘法表
arr = "\n".join(["".join(['%s*%s=%s '%(j,i,j*i)for j in range(1,i+1)]) for i in range(1
,10)])
print(arr)
```

第10节：函数

定义

将特定功能的代码进行封装，以便重复利用

优点

- 使程序更加简单
- 逻辑性更强
- 调用更方便
- 维护起来更加容易

语法

```
"""
def 函数名（形参）：
  函数体
  return [value]
"""
```

语法注意: 函数相对于del关键字必须保持一定的空格缩进

调用

```
"""
函数名（实参）
"""
```

参数形式

- 必选参数（不写就错）

```
def fn(name):
print(name)
fn()       # TypeError
```

- 缺省参数 (必选参数和默认参数的顺序不可调整，默认第一个形参)

```
def fn(name,sex='男'):
```

```
print(name,sex)
fn('wss')     # 换位置会出现SyntaxError
```

- 可变参数

```
def fn(name,sex='男',*cj):
print(name,sex,cj,type(cj))
fn('wss','男',12,34,45)
```

- 关键字参数

```
def fn(name,sex='男',*cj,**attr,):
print(name,sex,cj,attr,type(attr))
fn('wss','男',12,14,34,56,height=180,weight=100)
```

- 传参的顺序：必选参数、默认参数（缺省参数）、可变参数、关键字参数
- 特殊用法：万能参数

```
def fn(*args,**kwargs):
print(args,kwargs)
fn(12,3445,2244,age='18',name='wss')
```

- 解构用法
  加一个，解构出来 加一个*，解构出来

```
def fn(a,b,c,d):
print(a,b,c,d)
arr = [1,2,3,4]
fn(*arr)

def fn(name,age,sex):
print(name,age,sex)
dict1 = {'name':'wss','age':18,'sex':'男'}
fn(**dict1)     #  fn(name='wss',age=18,sex='男')  也可
fn(name='wss',sex='男',age=18)   使用键值形式可以不遵循顺序
```

返回值

```
return
def fn():
return 1,2,3
print(fn())     #返回值不可以是多个，如果有多个返回值，默认为元组
return作用:返回并终止函数
return注意:
    1、可以写多个return语句，最终只有一个被执行
```

```
    2、不能在return语句之后书写代码
    3、return只能返回一个数据，如果想返回多个数据必须为list/tuple/dict
 a,b,c=fn()
 print(fn(1,2,3))   #1,2,3
```

例子（ 斐波那契数列）

```
（1）输出第几个斐波那契数列的值
def f(n):
 a,b=1,1
 if n==1 or n ==2:
     return 1
 else:
     i=3
     while i<=n:
         a,b=b,a+b
         i+=1
     return b
print(f(int(input("输入一个数："))))
```

```
（2）输出斐波那契数列  （yield）
def fab(max):
 n, a, b = 0, 0, 1
 while n < max:
     yield b
     a, b = b, a + b
     n+=1
for i in fab(10):
 print(i)
```

高阶函数

实参高阶函数

- 把函数名当作参数传递给另一个函数

返回值高阶函数

- 返回值中包含函数名

```
1 普通写法
加法运算
def fn(a,b,c):
  return a+b+c
print(fn(1,2,3))
```

```
2 柯里化写法
def fn(a):
  def fn1(b):
      def fn2(c):
          return a+b+c
      return fn2
  return fn1
print(fn(10)(20)(30))
```

匿名函数 **lambda**

语法

```
lamdba   参数1，参数2，..:函数体
```

调用

- 自调用

```
print((lambda a,b:a+b)(10,20))
```

- 作为回调函数进行使用 高阶函数中作为参数

```
def fn1(a):
  return lambda  b: lambda c:a+b+c
print(fn1(10)(20)(30))
```

- 赋值

```
fn = lambda a,b:a+b
print(fn(10,20))
```

局部变量与全局变量

global: 顶层全局变量

```
a = 123
def fn():
  global a     # 声明成全局变量
  a = 456
  print(a)
fn()
print(a)
```

nonlocal：不是局部也不是全局，必须在嵌套函数中使用 （嵌套函数中的上层变量）

```
a = 123
def fn1():
  a = 456
  def fn2():
      nonlocal a     # 不是局部
      a = 789
      print(a)
  fn2()
  print(a)
fn1()
print(a)
```

作用域

- 命名空间
- python的四个作用域LEGB
- 作用域搜索顺序:python作用域搜索顺序遵循LEGB规则

闭包函数

在一些语言中，在函数中可以（嵌套）定义另一个函数时，如果内部的函数引用了外部的函数的变量，则可能产生闭包。闭包可以用来在一个函数与一组“私有”变量之间创建关联关系。在给定函数被多次调用的过程中，这些私有变量能够保持其持久性。

递归函数

如果一个函数在内部不调用其它的函数，而是自己本身的话，这个函数就是递归函数。

装饰器

定义

- 装饰器实际上就是为了给某个程序增加功能

使用场景

``` 定义公式：<函数+实参高阶函数+返回值高阶函数+闭包函数+语法糖=装饰器>

比如该程序已经上线或者已经被使用，那么久不能大批量地修改源代码，这样是不科学也是不现实的。
```

```
### 装饰器三原则
1．不能修改被装饰函数的源代码
2．不能改变被装饰函数的调用方式
3．满足1、2的情况下给程序增加功能
### 添加新功能：输出总时间
```

import time

def fn(): #所有的函数必须放在前面；定义好了写到内存中，没执行

time.sleep(1)

print("end..")

fn()

方式一

def fn2(fn1): # 局部的**fn1** 调用完**fn2**应该注释掉内部的变量

def newFn():

start = time.time()

fn1() # 函数的闭包，内层函数使用外层变量**fn1**

end = time.time()

print("总时间：**%s"%(end-start))**

return newFn # 返回**newFn**给**fn2**，相当于此时的**fn2**就是**newFn**了；注意这里不是返回**newFn()**

fn = fn2(fn) # 开始的**fn**是全局的**fn**作为实参给**fn2**，执行将**newFn**给**fn2**，把原来上边的**fn**覆盖了而存储成了**newFn**，然后再把**fn2**赋给此刻的**fn**

fn()

```
```
方式二：装饰器语法
def fn2(fn1):
def newFn():
start = time.time()
fn1()
end = time.time()
print("总时间：%s"%(end-start))
return newFn

@fn2
def fn():
time.sleep(1)
print("end..")

@fn2
def fn1():
print("abc")

fn()
fn1()
```

装饰器的传参

原函数传参

```
def fn2(fn1):
    def newFn(*args,**kwargs):
        start = time.time()
        fn1(*args,**kwargs)   # 解构一下
        end = time.time()
        print("总时间：%s"%(end-start))
    return newFn

@fn2
def fn(name):
    time.sleep(1)
    print("my name is %s"%name)
```
```

```
fn("wss"
```

装饰器传参

```
 def fn2(funName):
    def argsfn(fn):
        def newFn(*args,**kwargs):
            print("这是装饰(%s)函数"%funName)
            start = time.time()
            fn(*args,**kwargs)   # 接收参数
            end = time.time()
            print("总时间：%s"%(end-start))
        return newFn
    return argsfn

@fn2('fn函数')
def fn(name):
    time.sleep(1)
    print("my name is %s"%name)

fn("wss")
```

装饰器嵌套

```
"""
@fn2()
@fn1()
def fn():
    pass
"""

# 定义@fn1装饰器：输出装饰器名字
def fn1(funName1):
    print("%s在执行装饰器"%funName1)
    def newFun(fn):
        def newFun2(*args,**kwargs):
            print("%s在调用"%funName1)
            fn(*args,**kwargs)
        return newFun2
    return newFun

# @fn2装饰器  输出装饰器的名字
def fn2(funName2):
    print("%s在执行装饰器" % funName2)
    def newFun(fn):
```

```
        def newFun2(*args,**kwargs):
            print("%s在调用"%funName2)
            fn(*args,**kwargs)
        return newFun2
    return newFun

@fn2("fun2")
@fn1("fun1")
def fn(name):
    print(name)

fn("wss")
```

- 了解装饰器运行流程: 装饰器执行是按照从上到下执行

装饰器高级

- 类装饰器

第11节：常用系统内建函数

常用系统内建函数

```
            内置函数
abs()    dict()    help()    min()    setattr()
all()    dir()    hex()    next()    slice()
any()    divmod()    id()    object()    sorted()
ascii()    enumerate()    input()    oct()    staticmethod()
bin()    eval()    int()    open()    str()
bool()    exec()    isinstance()    ord()    sum()
bytearray()    filter()    issubclass()    pow()    super()
bytes()    float()    iter()    print()    tuple()
callable()    format()    len()    property()    type()
chr()    frozenset()    list()    range()    vars()
classmethod()    getattr()    locals()    repr()    zip()
compile()    globals()    map()    reversed()    __import__()
complex()    hasattr()    max()    round()
delattr()    hash()    memoryview()    set()
```

第12节：面向对象OOP

面向对象OOP （Object Oriented Programming）

类与对象概述

```
类：类是对象的抽象，比如飞机图纸，月饼模具
对象：对象是类的实例，比如飞机、月饼
面向对象的三大特点：封装、继承（默认继承object）、多态
关系：类是对象的模型，对象是类的具体实例化
```

语法

```
"""
class  类名(基类、父类)
    # 实例属性
    def __int__(self):   # self => this 实例 ;  __int__ => constructor (构造函数)
        self.name='wss'
        self.age = 18
    # 实例方法
    def say(self):
        print(self.name)
"""
```

```
# class Person:
#     def __init__(self,name='wss',age=18):
#         self.name= name
#         self.age = age
#     def say(self):
#         print("my name is %s"%self.name)
#     def a(self):
#         print("I am %s years old"%self.age)
# # 实例化，创建对象
# xm = Person('小明',17)
# xm.say()
# xm.a()
```

案例

案例一：时钟

```
## 创建一个时钟类
# 属性  小时:分钟:秒钟
# 方法
# run  表跑起来
# set  设置时间
# print（obj）  输出现在的时间
# class Clock:
#     def __init__(self,hou=0,min=0,sec=0):     # 魔法方法
#         self.hou = hou
#         self.min = min
#         self.sec = sec
#     def __str__(self):    # 对实例的描述  print(obj) 打印 __str__()魔法方法的返回值
#         return '%0.2d:%0.2d:%0.2d'%(self.hou,self.min,self.sec)
#     def run(self):
#         while True:
#             print(self)
#             time.sleep(1)   # 阻塞代码
#             self.sec += 1
#             if self.sec>59:
#                 self.min+=1
#                 self.sec=0
#                 if self.min>59:
#                     self.hou+=1
#                     self.min=0
#                     if self.hou>23:
#                         self.hou=0
#
#
# c1 = Clock(0,0,0)
# c1.run()
```

案例二：烤地瓜

```
## 创建一个烤地瓜类
# 属性：
# cookedLevel：这是数字；0~3表示还是生的，超过3表示半生不熟，超过5 表示已经烤好了，超过8表示已经烤成了木炭！地瓜刚开始的时候是生的。
# cookedString：这是字符串；描述地瓜的生熟程度
# condiments:这是地瓜的配料表，比如番茄酱、芥末酱
# 方法：
# cook() 把地瓜烤一段时间
# addCondiments() 给地瓜添加配料
# __init__() 设置默认属性
# __str__() 让print的结果看起来更好一些

class Potato:
```

```
    def __init__(self,cookedLevel=0,cookedString='生',condiments='番茄酱'):
        self.cookedLevel = cookedLevel
        self.cookedString = cookedString
        self.condiments = condiments
        self.condimentsList = ['芥末酱','番茄酱','甘梅酱']
    def __str__(self):
        return '您好，这是您的%s口味的%s的地瓜'%(self.condiments,self.cookedString)
    def cook(self,min):
        # while True:                    # 没有这行代码，结果打印一行
        #     time.sleep(1)                      # 阻塞代码
            self.cookedLevel += min              # 每秒加一
            if self.cookedLevel>=0 and self.cookedLevel<4:
                self.cookedString = '生'
            elif self.cookedLevel<6:
                self.cookedString = '半生不熟'
            elif self.cookedLevel<8:
                self.cookedString = '可以享用了'
            elif self.cookedLevel>=8:
                self.cookedString = '糊巴了'
    def addCondiments(self,num):
        self.condiments = self.condimentsList[num]


# 客户来了
p1 = Potato()
p2 = Potato()
p3 = Potato()
p4 = Potato()

p1.cook(2)
p2.cook(5)
p3.cook(7)
p4.cook(9)

p1.addCondiments(0)
p2.addCondiments(2)
p3.addCondiments(1)
p4.addCondiments(2)

print(p1)
print(p2)
print(p3)
print(p4)
```

案例三：绝地求生

链接1

类变量（类可以使用的属性）与实例变量（实例可以使用的属性）

- 类变量：定义在类中，可以被类方法（修改），可以被实例访问
- 实例变量：实例变量只能实例调用，且权重大（优先级大），它是后写的，可以把之前的内容覆盖掉

案例

```

**class P:**

**sex = '男' #** 类变量，能够被实例继承（图纸）

#

**def init(self):**

**self.name = 'xm' #** 实例属性，实例变量

**self.sex = '男' #** 实例变量只能实例调用，且权重大（优先级大），它是后写的，可以把之前的内容覆盖掉

#

**@staticmethod #** 静态类方法，什么都不能调用，就是一个普通的公共函数功能， 连指针都没有

**def say2():**

**print("this is static")**

#
```

@classmethod # 类方法，可调用类变量

def setSex(cls,sex):

print(cls.name)

cls.sex = sex

#

def say1(self): # 实例方法，可调用实例的方法和属性，还可以调用继承过来的类变量

print(self.name)

print(self.sex)

xm = P()

 # xm.sex = '女**' #** 创建了实例属性（实例对象），给实例重新赋予属性

P.sex = '女**' #** 所有的实例用的都是同一个类变量

print(xm.sex) # 类变量可以被实例调用，实例变量与类变量重名时，优先调用实例变量

print(P.sex) # 类变量也可以被类调用

xh = P()

xh.say1()

xh.say2()

xm = P()

print(xm.name)

print(xm.sex) # 小明的**sex**继承的类变量

P.setSex('女**') #** 调用类方法，修改类变量

print(xm.sex)

xm.setSex('男**') # xm**这个实例调用了继承的类方法，修改了类变量

print(xm.sex)

xm.sex = '女**' #** 添加了实例属性**sex**，覆盖了继承来的类变量**sex**，并没有修改了类变量

print(xm.sex)

print(P.sex)

```
## 静态类方法、类方法、实例方法
### 静态类方法
@staticmethod
### 类方法
@classmethod
```

""" 静态方法：通过@staticmethod 装饰，不依赖于类变量、类方法，实例变量、实例方法 类方法：通过@classmethod 装饰，修改类变量 实例方法：可调用实例的方法和属性，还可以调用继承过来的类变量 """

```
## 私有属性和私有方法
```

```
__spam
```

所谓私有，就是在外部不能调用，方法之间可以使用的 私有方法以及属性需要在方法名或属性名前加"__",但是python中没有真正的私有方法、私有属性

class P():

def init(self,name,age):

self.__name = name

self.__age = age

def say(self):

print("my name is %s"%self.name) #内部可以使用**name**

#

p = P('小红**',19)**

print(dir(p))

p.name = "肖红**" #** 外部不可使用**name**

p._P__name = "肖红**" #** 君子协议，但是可以强制改

p.say()

```
## 装饰器
```

@property 私有：将方法改成一个同名的属性，这个属性不能修改（因为就没有这个属性，只能看）

class P():

def init(self,name,age):

self.__name = name

self.__age = age

def say(self):

print("my name is %s"%self.__name)

#

@property # 装饰器

def name(self):

return self.__name

#

p = P('小红**',19)**

print(dir(p))

p.name = "1234" # 只能让外部看，不能修改（比如，血量，枪，金币）

print(p.name)

```
### 类装饰器
```

def MyClass(obj): obj.age = 19 # 类变量 obj.say = lambda cls:print(cls.age) # 类方法 return obj

@MyClass class P: def **init**(self,name): self.name = name class A:

p = P("小红") print(dir(p))

```
### 实例方法装饰器
```

import time

def sumtime(fn): def newFn(self,*args,**kwargs): start = time.time() fn(self,*args,**kwargs) end = time.time() print("%s你好帅"%self.name) print("总时间：%s"%(end-start)) return newFn

实例方法装饰器

class P: def **init**(self,name): self.name = name @sumtime def run(self): time.sleep(1) print(self.name)

p = P("wss") p.run()

```
## super()
* super 相当于this指针，用来引用父类而不必显式地指定它们的名称
## 魔法方法
* __slots__  限制Class属性和方法
* __name__   类名字
```

class P:

"""

Person类 人类

"""

def init(self):

pass

slots = ['name','sex']

name = "Person"

def run(self):

print(self.doc)

#

p = P()

P.name = "wss"

p.sex = "男"

 # p.age = 18

print(dir(p))

print(p.doc)

```
* print(self.__doc__)  类文档字符串、开头注释
```

拿到函数的注释

def fn():

"""

this is fn

:return:

"""

pass

print(fn.doc)

```
* __new__  与  __init__  关系
```

new ：第一步、创建一个对象，必须有返回值。把实例创建出来，可能任何属性都没有

init ：第二步、给实例初始化，赋予属性

new 与 **init** 关系

class Car:

def init(self):

self.color = ""

#

class BMW(Car):

def new(cls, *args, *kwargs): # 第一步、创建一个对象，必须有返回值。把实例创建出来，可能任何属性都没有

print("new")

return super().new(BMW)

#

def init(self): # 第二步、给实例初始化，赋予属性

print("init")

self.pingpai = ''

#

def del(self):

print('del')

#

b = BMW() # b运行完被回收了

print(dir(b))

del b

print(b)

```
* __str__    给实例一个描述
* __del__    实例注销时调用
* [常用魔法方法](https://blog.csdn.net/yunyi4367/article/details/79052970)
## 验证
```

ifinstance（）来检查一个实例的类型 print(isinstance(obj，class)) 验证obj是否为class的实例

Issubclass（）来检查类的继承关系 print(issubclass(P,A))

type（）查看类型

dir（）获取对象的属性和方法 ```

元类

第13节：面向对象单例设计模式

```
# 设计模式之单例模式
# 单例设计模式是怎么来的？
# 在面向对象的程序设计中，当业务并发量非常大时，那么就会出现重复创建相同的对象，每创建一个对象就会开辟一块内存空间，而这些对象其实是一模一样的，那么有没有办法使用得内存对象只创建一次，然后再随处使用呢？单例模式就是为了解决这个问题而产生的。
#
# 实现方式：
# 1、创建一个类静态字段（类变量）__instance
# 2、创建一个静态函数，通过函数的逻辑判断 __instance 是否已存在，如不存在就将对象值赋于__instance，即__instance = 类(),否则直接返回__instance，也即创建的对象都是一样的
# 3、使用单例模式创建对象时直接通过类调用静态函数创建即可
#普通模式
class A(object):
    def __init__(self,name,male):
     self.name = name
     self.male = male
#实例化多个对象
obj1 = A('ben','boy')
obj2 = A('min','girl')
obj3 = A('miao','boy')
##打印内存地址，可以看到内存地址都是不一样的
print (id(obj1),id(obj2),id(obj3))

#单例模式
class A(object):
     __instance = None
     def __init__(self,name,male):
         self.name = name
         self.name = male
     @staticmethod
     def create_obj():
         if not A.__instance:
             A.__instance = A('ben','boy')
             return A.__instance
         else:
             return A.__instance
# 29 >>> #单例模式实例化多个对象
obj1 = A.create_obj()
obj2 = A.create_obj()
obj3 = A.create_obj()
# 33 >>> ##打印内存地址，可以看到内存地址都是一样的
print (id(obj1),id(obj2),id(obj3))
```

第14节：文件操作

操作文件的步骤

```
1、打开文件
2、操作文件（读写）
3、关闭文件
```

open（）函数的用法

```
open(文件路径，打开模式（读、写、读写、二进制、普通内容）)  打开文件
"""
file 文件路径
mode 读取模式：
     r：读取文件内容
     r+：读写，从头覆盖旧内容，没有文件则报错
     w：每次都重新写入，路径不对会创建新文件
     w+：读写，路径不对会创建新文件
     a：追加，路径不对会创建新文件
     a+：读，追加
     rb、rb+、wb、wb+、ab、ab+    以二进制的方法处理
encoding 字符编码

errors(出错处理机制)
strict(默认)
ignore(忽略)

"""
```

读文件

```
f.read([num])
num代表字符的数量，默认为全部

f.readline([num])
文件读取每一个行，通过\r \n EOF (文件结束标识) 来区分每一行,num代表输出的字符

f.readlines()
读取每行内容，以列表的形式输出
```

```
# f = open("./day04.txt",'a',encoding="utf-8")
# f = open("./day04.txt",'rb')

# f1 = open("./day04.txt",'r',encoding="utf-8")
# f2 = open("./day0401.txt",'w',encoding="utf-8")

# 读
# con = f.read()         # 读取全部内容
# con = f.read(5)     # 读取5个字符
# con = f.readline()       # 读取一行
# con = f.readline(2)       # 读取一行的前2个字符
# con = f.readlines()     # 读取每行内容，以列表的形式输出
```

写文件

```
f.write(str)
每次都重新写入，路径不对会创建新文件

f.writelines()
读写，路径不对会创建新文件
```

```
# 写
# f.write("你好优逸客")
# lines = f1.readlines()
# # print(lines)
# f2.writelines(lines)
# f1.close()
# f2.close()
```

操作指针

```
f1.seek(p,0)
每个uincode码，3个字节。移动文件指针到第p个字节

f1.seek(p,1)
相对于当前位置往后移p（文件以二进制方式打开）

f1.seek(-2,2)
文件尾之后的p个字节(负值就是倒着往回)  （文件以二进制方式打开）

f.tell()
返回文件读取指针的位置
```

```
## 操作指针
# f1 = open("./day04.txt",'rb',encoding="utf-8")
# f1 = open("./day0401.txt",'rb')

# f1.seek(2,0)        # 每个uincode码，3个字节。移动文件指针到第p个字节
# f1.seek(2,1)        # 相对于当前位置往后移p
# f1.seek(-2,2)      # 文件尾之后的p个字节(负值就是倒着往回)
# print(f1.read(1))
# print(f1.tell())        # 返回文件读取指针的位置
# f1.close()
```

关闭文件

```
f.close()
1、把缓存区的内容写到硬盘   2、关闭文件

f.flush()
把缓冲区的内容写到硬盘
```

pickle持久化（写到硬盘中）

```
import pickle
 # pickle.dump(obj,file)       # 把obj存储到file文件中
 # pickle.load(obj,file)     # 把file文件中的数据进行读取
 # f = open("./day04.txt",'rb+')
 # obj = [1,2,3,4,5,6,7,8,9]
 # pickle.dump(obj,f)    # 把obj存储到file文件中
 # obj = pickle.load(f)        # 把file文件中的数据进行读取
 # print(obj)
 # f.close()
```

csv数据

第15节：异常

程序为了避免更大的错误，会出现异常。

```
### 文件打开或读取时，可能会发生IORrror异常，可能导致文件无法关闭。出现异常
# try:        # 可能会错误的代码
#     # print(1/0)      # 程序员逻辑错误，水平不够高
#     f = open("day04.txt",'r',encoding="utf-8")
#     con = f.read()
#     print(con)
# except:     # 出错处理程序
#     print("文件出错了")
# finally:        # 不管正确还是错误，总会执行
#     # print("finally")
#     f.close()
```

概述

错误

```
1.语法错误
    推导式、for循环
2.逻辑错误
    1/0、 思路上的错误
```

异常

阶段

```
1.发生异常
2.异常处理
```

定义

因为程序出现错误而在正常流程控制以外采取措施的行为。 当Python检测到一个错误时，解释器就无法继续执行了，反而出现了一些错误提示，这就是所谓的异常。

```
异常与捕获异常
##捕获异常
"""
try:
```

```
    # 可能会发生异常的代码
except  异常类 as e:
    # 出现异常执行的代码
except 异常类2 as e:
    pass
else:
    # 没有发生异常执行的代码
finally:
    # 无论是否发生异常，都要执行的代码
"""
    """
# 自定义异常
class UserError(Exception):
    def __init__(self,num,info):
        self.num = num
        super().__init__(self)      # super传到基类
        self.info = info
    def __str__(self):      # 打印
        return self.info
"""
        """
# 抛出异常
raise UserError(num)
"""
```

捕获异常原理

Try的工作原理，当开始一个try语句后，python就在当前程序的上下文做标记，这样异常出现时就可以回到这里，try子句限制性，接下来发生什么依赖于发生了什么异常。

1. 如果当try后语句执行时发生异常，python就跳回到try并执行第一个匹配该异常的except子句，异常处理完毕，控制流就痛殴整个try语句

2、如果try后语句发生异常，却没有匹配except子句，异常将递交上层try，或者到程序的最上层，终止程序，并输出错误信息。

3、如果try子句执行时没有发生异常，python将这些else语句后的程序，然后控制流通过整个try

异常传递

如果try嵌套，那么如果里的try没有捕获到这个异常，那么外面的try会接收到这个异常，然后进行处理，如果外面的try没有接收到这个异常，那就再向外传递；

函数的嵌套也类似，例如函数A-->函数B-->函数C ,如果异常是在函数C总产生的，那么如果函数C没有对这个异常进行处理，那么这个异常就会传入到函数B中，如果函数B有异常处理，就会按照函数B的异常处理方式进行执行，如果函数B没有异常处理，那么这个异常会被继续传递，以此类推。如果所有的函数都没有异常处理，那么此时就会进行异常的默认处理，即通常见到的那样。

异常类

```
BaseException
 +-- SystemExit  解释器请求退出
 +-- KeyboardInterrupt  用户中断执行（ctr + c）
 +-- GeneratorExit   生成器发生异常来通知退出
 +-- Exception   常规错误基类
      +-- StopIteration     迭代器没有更多的值
      +-- StopAsyncIteration
      +-- ArithmeticError   数值计算错误基类
      |    +-- FloatingPointError   浮点计算错误
      |    +-- OverflowError     数值运算超出最大限制
      |    +-- ZeroDivisionError     除(或取模)零 (所有数据类型)
      +-- AssertionError    断言语句失败
      +-- AttributeError    对象没有这个属性
      +-- BufferError
      +-- EOFError  没有内建输入，到达EOF标记
      +-- ImportError   导入模块失败
      |    +-- ModuleNotFoundError
      +-- LookupError   无效数据查询基类
      |    +-- IndexError 序列中没有此索引(index)
      |    +-- KeyError 映射中没有这个键
      +-- MemoryError   内存溢出错误
      +-- NameError 未声明未初始化的本地变量
      |    +-- UnboundLocalError      访问未初始化的本地变量
      +-- OSError   操作系统错误
      |    +-- BlockingIOError 操作阻塞设置为非阻塞操作的对象（例如套接字）时引发
      |    +-- ChildProcessError 子进程上的操作失败时引发
      |    +-- ConnectionError 连接相关的问题的基类
      |    |    +-- BrokenPipeError
      |    |    +-- ConnectionAbortedError
      |    |    +-- ConnectionRefusedError
      |    |    +-- ConnectionResetError
      |    +-- FileExistsError 尝试创建已存在的文件或目录时引发
      |    +-- FileNotFoundError     在请求文件或目录但不存在时引发
      |    +-- InterruptedError 系统调用被输入信号中断时触发
      |    +-- IsADirectoryError 在目录上请求文件操作时引发
      |    +-- NotADirectoryError 在对非目录的os.listdir()事物请求目录操作（例如）时引发
      |    +-- PermissionError 尝试在没有足够访问权限的情况下运行操作时引发 - 例如文件系统权
限。
      |    +-- ProcessLookupError 当给定进程不存在时引发
      |    +-- TimeoutError 系统功能在系统级别超时时触发。
      +-- ReferenceError    弱引用(Weak reference)试图访问已经垃圾回收了的对象
      +-- RuntimeError  一般的运行时错误
      |    +-- NotImplementedError       尚未实现的方法
      |    +-- RecursionError
      +-- SyntaxError 一般的解释器系统错误
      |    +-- IndentationError 缩进错误
      |         +-- TabError    Tab 和空格混用
      +-- SystemError Python 语法错误
```

```
      +-- TypeError      对类型无效的操作
      +-- ValueError     传入无效的参数
      |    +-- UnicodeError  Unicode 相关的错误
      |         +-- UnicodeDecodeError      Unicode 解码时的错误
      |         +-- UnicodeEncodeError  Unicode 编码时错误
      |         +-- UnicodeTranslateError   Unicode 转换时错误
      +-- Warning    警告的基类
           +-- DeprecationWarning    关于被弃用的特征的警告
           +-- PendingDeprecationWarning     关于特性将会被废弃的警告
           +-- RuntimeWarning    可疑的运行时行为(runtime behavior)的警告
           +-- SyntaxWarning     可疑的语法的警告
           +-- UserWarning  用户代码生成的警告
           +-- FutureWarning     关于构造将来语义会有改变的警告
           +-- ImportWarning
           +-- UnicodeWarning
           +-- BytesWarning
           +-- ResourceWarning
```

第16节：with 上下文管理器

概述

上下文管理器是一个对象，它定义了在执行with语句时要建立的运行时上下文。上下文管理器处理执行代码块所需的运行是上下文的入口和出口。上下文管理器通常使用with语句调用，但是也可以通过直接调用它们的方法来使用。

作用

with还可以处理下文的异常（对里面的错误进行捕获）

用途

1. 保存和恢复各种全局状态
2. 锁定和解锁资源
3. 关闭打开的文件

重点

```

### 重点来了

### **with** 上下文管理器

### 语法

### **with open('day04.txt','rb') as f: # open('day04.txt','rb')** 作为上下文管理器赋给 **f,**它会加上 **enter**、**exit**

### **f.read()**

## 模拟上下文管理器
```

class Simple: # 外边的是全局的，是上文 def **enter**(self): # 入口 print("enter") return self def **exit**(self, exc_type, exc_val, exc_tb): # 终究会退出 不管正确与否都要一个出口，可以处理下文的异常（对里面的错误进行捕获） print('exit')

```
    # print("exc_type",exc_type)
    # print("exc_val",exc_val)
    # print("exc_tb", exc_tb)
    # return True     # 不报错了，对错误进行改造
# def run(self):
#     print(1/0)
```

#

def simple():

print("simple")

return Simple()

with Simple() as f: print("f")

```
# f.run()     # 下文
```

```
```

第17节：迭代器和生成器

迭代器（iterator）

概述

```
迭代对象
__next__()
    通过next()函数可以手动调用迭代器对象__next__()方法，返回下一个值,直到函数出现StopIteration异常，停下来.
__iter__()
    返回迭代器对象本身
```

生成器（generator）

生成器，带有yield的函数，通过yield返回值

yield函数作用

把函数变成了生成器,从当前的位置继续在下一次的执行.

创建生成器

1.通过函数创建

```
    # def fib(num):
#     # i,a,b = 0,1,0
#     i, a, b = 0, 0, 1
#     while i < num:
#         yield b
#         i,a,b = i+1,b,b+a
#
# genObj = fib(10000)     # 生成器，因为fib有yield，会输出时间。不占用很多内存。计算、保存、提取。
# print(next(genObj))     # 调用一次计算一次，计算的时间节省下来。结果也只给一次。
# print(next(genObj))
# print(next(genObj))
# print(next(genObj))
# print(next(genObj))
# # print(genObj)       # 输出genObj是一个生成器
```

2.通过（推导式语法）创建

```
# arr = [1,2,3,4,5,6,7]
# arr2 = [i**2 for i in arr]
# arr3 = (i**2 for i in arr)
# print(arr3)     # arr3 现在是一个生成器了
```

第18节：模块与包（modules）

模块

模块与包是任何大型程序的核心，就连python安装程序本身也是一个包

定义：是一个包含python定义和语句的脚本文件，模块相当于文件

包

定义：是模块的集合，包类似于文件夹

分类

- 正规包 文件夹中包含 **init**.py 文件
- 命名空间包 不包含**init**.py 文件 （）、 不受物理位置限制

常用操作

导入模块的方式

``` 爬虫可以协同开发 （爬虫、 pip、 登录问题，实现不同功能的代码放到一起，就需要模块。每一个模块里都有命名的空间，变量函数都在这个空间里。）

from mymodules import aa import mymodules.aa import mymodules.aa as aa # 起别名 aa.aa1() from mymodules import aa as a a.aa1() from .mymodules import aa # 不会报红线错，但是主文件不行，不能这样用 aa.aa1() from mymodules.sonModules import bb bb.bbfun() from mymodules.sonModules import bb bb.bbfun() from mymodules.sonModules.bb import * # 把bb里的命名空间导入到了主运行函数里； from 后跟包或者由包组成的路径；import 后跟 模块； as 后跟别名 bbfun() aa1()

```
模块的导入顺序

re.py   先从当前目录寻找包，然后从系统路径中寻找
sys.path 列表数据 中保存的寻址路径   cmd 中执行 import sys -> sys.path   可以添加自己的路径
```

```
相对路径导入包中子模块

在非主运行包中 可以使用 相对路径的方式导入
. 当前
```
```

```
.. 上一级
```

运行包

```
__main__    可以使包直接运行
cd desktop -> python demo
```

通过字符串导入包

```
# 通过字符串导入包 ： 有一堆包，可以遍历导入。scrappy里的setting里的中间件，遍历那个对象（字符串），然后根据键的排序调用：collections。
import time
import importlib
time = importlib.import_module("time")
time.sleep(1)
print(time)
```

相关属性

```
print(__file__)     # 保存的文件的路径
print(__name__)     # __main__   包的注释，主运行的文件，给赋上main
print(__package__)      # None   默认情况下是None，作为模块
print(__path__)    #包命名空间
```

第二章

第1节：概述

定义

网络爬虫（又称为网页蜘蛛，网络机器人，在FOAF社区中间，更经常的称为网页追逐者），是一种按照一定的规则，自动地抓取万维网信息的程序或者脚本，另外一些不常使用的名字还有蚂蚁、自动索引、模拟程序或者蠕虫。

分类

- 通用爬虫

百度搜入：爬取，搜索排名，为了更好检索内容、更好地获取关机键 搜索引擎优化

- 聚焦爬虫

猎头：爬取简历进行推荐，把用户的有价值的数据爬取下来了；正常公开免费的数据可以获取

工作流程

1.设置爬取地址，设置请求头（导航，嵌套爬取）

2.解析页面，获取需要信息（分析页面）

3.数据存储（mysql、表格、其他数据库比如基于内存存储的数据库）

Robots协议

Robots协议（也称为爬虫协议、机器人协议等）的全称是"网络爬虫排除标准"（Robots Exclusion Protocol），网站通过Robots协议告诉搜索引擎哪些页面可以抓取，哪些页面不能抓取。

Robots协议

常用工具

- Builtwith 识别网站所用技术

  import builtwith print(builtwith.builtwith('https://www.wordpress.com'))

- whois 查询网站所有者的工具
- uriilb.robotparser 解析robots.txt工具

第2节：HTTP概述

http协议

HyperText Transfer Protocol, 超文本传输协议

https

Hypertext Transfer Protocol over Secure Socket Layer HTTP下加入SSL层，HTTPS的安全基础是SSL，HTTPS的安全基础是SSL，因此加密的详细内容就需要SSL。

区别

- HTTPS存在不同于HTTP的默认端口及一个加密/身份验证层（在HTTP与TCP之间）
- HTTPS广泛用于万维网上安全敏感的通讯，例如交易支付方面
- HTTP端口号：80；HTTPS端口号：443

客户端HTTP请求（带上请求头，证明）

先请求DNS服务器，把ip地址请求到，才能知道请求的是哪台电脑 还要带上一些请求头

- 请求报头(客户端带一些信息给服务器)
- Host（主机和端口号） http://118.190.150.35:9000/login
- Connect（链接类型）：keep alive 长连接，不需要每次都去请求
- User-Agent（浏览器名称） 使用fake_useragent包

```
from fake_useragent import UserAgent

ua = UserAgent()      # 实例化出来
headers={
"User-Agent" :ua.random
}
```

- Accept（接收传输文件类型）

```
*/* 什么都可以接收
image/gif 希望接收GIF图像格式的资源
text/html 表明客户端希望接收html
text/html,application/xhtml+xml;q=9,image/*;q=0.8
支持html文本、xhtml和xml文档、所有的图像格式资源，q为权重(0<=q<=1)默认为1（规定分号前面的内容），越靠近1表示越希望";"之前的内容，为0 不希望的类型
```

- Referer（页面跳转处） 表明产生请求的网页来自于哪个URL
- Accept-Encoding（浏览器接受文件编码格式）
- Accept-Language（浏览器接受语言种类）
- Accept-Charset（浏览器可接受字符编码）
- Cookie（浏览器本地Application中）
- Content-Type（POST请求中数据的数据类型）
- 响应报头
- Cache-Control：must-revalidate，no-cache，private 服务器不希望客户端缓存资源，下次必须重新请求 Cache-Control：max-age=86400 在86400秒的时间内，客户端可以从缓存中读取资源
- Connection：keep-alive 告诉客户端服务器的tcp连接也是一个长连接，客户端可以继续使用这个tcp链接发送http请求
- Content-Encodeing：gzip 告诉客户端服务端发送的资源时采用gzip编码的
- Content-Type：text/html;charset=UTF-8 告诉客户端资源文件类型还有字符编码
- Date：Sun，21 Sep 服务器发送资源时间
- Expires:Sun... 告诉客户端在这个时间内，可以直接访问缓存副本
- Pragma：no-cache与Cache-Control
- Server 服务器信息
- Transfer-Encoding:chunked 告诉客户端，服务器发送的资源的方式是分块发送的
- Vary：Accept--Encoding 告诉缓存服务器，缓存压缩文件还是非压缩文件
- 响应状态码
    - 100~199 服务器成功接收部分请求，需要客户端继续提交其余请求
    - 200~299 服务器成功接收请求并已完成整个处理过程
    - 200 请求成功
    - 300~399 为完成请求，客户需要进一步细化请求
    - 307 304 使用缓存资源
    - 400~499 客户端请求有错误
    - 404 无法找到页面
    - 403 服务器拒绝访问
    - 500~599 服务器端出现错误
    - 500 请求未完成

第3节：数据获取

数据采集

1.urllib包 （包里有丰富的API）

概述

uillib是一个收集了多个使用URL的模块的软件包

组件

- request：打开和阅读url地址的包

  urllib.request

- error：包含 urllib.request 抛出的异常

  urllib.error

- parse：用于处理 URL urllib.parse

```
编码转换
    parse.urlencode({}）
    quote("太原"）:把汉字转换为编码
    urlparse(url):解析url各个部分
```

- robotparser：用于解析 robots.txt 文件

  urllib.robotparser

响应类型为**json**

- json.loads(str) 将 str 转换为 dict
- json.dumps(dict) 将 dict 转换为 str

2.requests包

通过urllib底层实现（在urllib的基础上进行更好的封装）

GET请求

get(URL,params,headers,timeout)

res = requests.get('www.baidu.com') 响应一个地址

res.text 查看响应内容 res.content 字节流 res.url 完整的地址 res.encoding 响应字符编码 res.status_code 响应码

POST请求

post(URL,data,header={})

data={}

res.text res.json() 显示json

代理

proxies = {"http":"http：//12.34.56.79:9527"} response = response.get("http://www.baidu.com",proxies = proxies)

proxie = {

```
    'http':'http://%s'%ip,
    'https': 'https://%s'%ip,
```

}

requests.get('http://wwww.baidu.com',proxies=proxie,timeout=10) # 请求百度，测试ip地址是否可用 本地配置代理

export HTTP_PROXY="http://12.34.56.79.9527" export HTTPS_PROXY="https://12.34.56.79.9527"

cookie和session（保存在浏览器的数据）

- 包中利用cookie以及session来实现登录
- 日报实例:日报系统中找到学生id 可以在布局里找，也可以在updatecontent里找，就是提交两次可以出现。
- 获取cookie

处理HTTPS请求SSL证书验证

response = requests.get("https://www.baidu.com/",verify=True)

跳过验证 verify=False （默认）

3.fake-useragent 工具

- 安装 pip install fake-useragent import User Agent
- 使用

```
ua=User Agent（）
ua.ie
ua.firefox
ua.chrome
ua.random
```

4.Selenium（浏览器测试）

一、概述

Selenium是一个web的自动化测试工具，最初是为网站自动化测试而开发的，类型像我们玩游戏时用的按键精灵，可以按指定的命令自动操作，不同的是Selenium可以直接运行在浏览器上，它支持所有主流的浏览器（包括PhantomJS这些无界面的浏览器）

中文文档

安装:pip install selenium

安装 ChromeDriver Mirror

二、 常用组件

webdriver

操作步骤

1. 导入 from selenium import webdriver
2. driver.get("http://118.190.150.35:9000/login") 打开网址，打开需要时间，建议time.sleep()
3. 操作数据
4. drive.quit() 关闭浏览器

详细操作

① 获取页面元素

- driver.find_element_by_id('loginForm') #通过ID获取元素
- driver.find_element_by_name('continue') #通过name属性获取元素
- driver.find_element_by_xpath("//form[@id='loginForm']/input[4]") #通过XPath查找元素
- driver.find_element_by_link_text('Continue') #通过链接文本获取超链接
- driver.find_element_by_partial_link_text('Conti')
- driver.find_element_by_tag_name('h1') #通过标签名查找元素
- driver.find_element_by_class_name('content') #通过Class name 定位元素
- driver.find_element_by_css_selector('p.content') #通过CSS选择器查找元素

② 事件

- 步骤

- from selenium.webdriver import ActionChains 导入 ActionChains 类
- loginbtn = driver.find_element_by_partial_link_text("登") 获取元素
- action = ActionChains(driver) 创建对象
- action.click(loginbtn) 填写事件
- action.click(loginbtn).perform()
  执行
- 事件

```
click()  单击
double_click()  双击
context_click(ac)  右击
click_and_hold()  按下
drag_and_drop(e1,e2)  将元素e1移动到e2
drag_and_drop_by_offset(e1,x,y)  将元素移动位置
release(e)  释放鼠标按钮
pause(num)  停几秒
send_keys(con)  将con发送到当前聚焦元素
send_keys_to_element(ele,con)  将con发送到元素
```

③ 下拉框处理

步骤

```
1. from selenium.webdriver.support.ui import Select
    导入select类
2. select = Select（ele）
    创建选择对象
3.  select_by_index(1)
    select_by_value("0")
    select_by_visible_rext("选择的内容")
    deselect_all()  取消选择
```

④ 弹窗处理

```
alert = driver.switch to alert()  获取弹框内容
```

⑤ 页面切换

```
driver.switch_to_alert()  获取弹窗内容
for handle in driver.window_handles:
driver.switch_to_window(handle)
```

⑥ 页面前进和后退

```
driver.forward()  #前进
```

```
driver.back()  #后退
```

⑦ Cookies

```
driver.get_cookie()  # 获取cookies
driver.delete_cookie("key")  # 删除对应cookie
driver.delete_all_coolies()  # 删除全部cookie
```

⑧ 页面等待

- 显式等待
- 隐式等待

⑨ 补充(瀑布流布局)

```
dirver.execute_script()
执行JavaScript 脚本

driver.quit()
关闭浏览器

dirver.save_screenshot('abc.png')
页面快照

dirver.page_source
页面源码
```

```
案例:瀑布流布局
### 瀑布流布局
url = "https://unsplash.com/t/wallpapers"
driver.get(url)
driver.execute_script("window.scrollTo(0,2000)")        # 执行JavaScript脚本

time.sleep(5)
with open("img.html",'w',encoding='utf-8') as f:
    f.write(driver.page_source)     # 页面源码

driver.save_screenshot("img.png")       # 页面快照
driver.close()      # 关闭浏览器
```

三、 处理：通过js写的，页面加载；滚动

5.验证码处理

页面数据解析

lxml

Xpath常用规则

- 安装

```
pip install lxml
from lxml import etree
```

- // 从当前节点选取后代节点

```
# root = html.xpath("//div")      # 从当前节点找，任意位置，后加上标签名div
```

- NodeName 选取此节点的所有子节点
- / 从当前节点选取子节点

```
  # root = html.xpath("/div")      # div标签名，从根开始找(找不见)
```

- . 当前节点

```
  # links = html.xpath("//div/ul/li/a/./../div")
```

- .. 当前节点的父节点

```
  link = html.xpath("//div/ul/li/a/../div")
```

- @ 选择所有属性

```
  /@herf  获取herf属性
  /@*  获取元素全部属性
```

- [tag] 选取所有具有指定元素的直接子节点

```
  # links = html.xpath("//div/ul/li[a]")      # [tag]里是条件，获取拥有特点元素的节点
```

- [tag = 'text']

```
  # links = html.xpath("//div/ul/li[a='优逸客']")      # [tag = text]里是条件，获取拥有特点元素值的节点
```

- 节点轴 ```

XPath 轴

links = html.xpath("//div/ul/li/child::div") # li 下面所有的 **div**

print(links)

links = html.xpath("//div/ul/li/attribute::class") # li 下面所有的 **class**类名

print(links)

links = html.xpath("//div/ul/li/child::text()") # li 下面所有的子类的内容

print(links)

links = html.xpath("//div/ul/li[last()]/child::*") # li 下面所有的子元素

print(links)

links = html.xpath("//div/ul/li/child::node()") # li 下面所有的子元素

print(links)

links = html.xpath("//div/ul/li/descendant::*") # li 下面所有的后代元素

print(links)

links = html.xpath("//div/ul/parent::*") # ul 父元素

print(links)

links = html.xpath("//div/ul/ancestor::*") # ul 所有的祖先元素

print(links)

links = html.xpath("//div/ul/li[last()]/a/preceding::*") # 当前元素之前的所有节点

print(links)

```
* 常用函数
contains（）
```

links = html.xpath("//div/ul/li/a[@class='link1 link3']") # 直接用类名获取元素，两个类名都得写

print(links)

links = html.xpath("//div/ul/li/a[contains(@class,'link1')]") # 用**contains**获取元素，只要写一个类名就可以

print(links)

```
last（）
```

links = html.xpath("//div/ul/li[position()<last()]/a/text()") # [position()]里是条件，获取拥有特点元素值的节点 print(links)

```
#### 组件
```

etree（文档树）

HTML初始化生成一个XPath解析
tostring（html，encoding=''）
xpath() 通过获取路径来获取资源

```
为什么请求一个地址可以打开？
浏览器是个什么样的存在可以把网页打开？网页又是一个什么样的存在（html）？
网页在别的电脑上，怎样通过浏览器把别人电脑上的网页拿回来？
获取数据的软件：通过你输入的请求，把地址转换成哪个电脑上的哪个软件的哪个应用。
找见对应的服务器对应的软件的内容，把它的内容发送给你，通过网络传输回来，传回到本地。本地就是把传输回来的数据呈现出来。传输的过程全部是二进制的方式，解析完之后依旧是一个字符串，不是一个对象节点。
是谁帮你把它做成了对象节点？就是js引擎，浏览器内核帮你解析的。从字符串到具体呈现出一个对象，是浏览器帮你做的。
我们通过程序请求，没有浏览器帮我们处理，通过python请求的，python没有浏览器内核，没有办法帮你解析字符串。但是它能拿到文本，不会给你画出来。把字符串变成一个文档树。
```

re 正则

概述

用来描述或者匹配一系列符合某种规则得字符串的表达式

（输入用户名密码，长度不够，特殊符号，怎样知道输入的格式对不对）

正则表达式

- 元字符 ``` 对一类字符进行描述

\w 字符

res = re.findall(r"\w","123abcdef_$^<>") # \w只匹配一个字符,命名的时候可以命名什么

print(res)

\W 非字符

res = re.findall(r"\W","123abcdef_$^<>")

print(res)

\d 数字

res = re.findall(r"\d","123abcdef_$^<>")

print(res)

\D 非数字

res = re.findall(r"\D","123abcdef_$^<>")

print(res)

\b 单词边界**(**单词边界就是空**)**

res = re.findall(r"\bi","this is div")

print(res)

\B 非单词边界

res = re.findall(r"\Bi","this is div")

 # print(res)

\s 空

res = re.findall(r"\s","this is div")

print(res)

\S 非空

res = re.findall(r"\S","this is div")

print(res)

. 除了**\n** 以外的所有字符

res = re.findall(r".","this is div")

print(res)

```
* 原子表
```

原子表 也是匹配一位字符,可以连一起找 [] 类似一个列表

[abc]

[a-z]

[A-Z]

[a-zA-Z1-9]

… 取反

res = re.findall(r"[hs]","this is div") # 匹配**h**和**s**

res = re.findall(r"[a-z]","this is DIV") # 匹配小写字母

res = re.findall(r"[A-Z]**","this is DIV") #** 取反

print(res)

```
* 数量
```

- 0个或多个字符
- 1个或多个 ？ 0个或1个 {n} n个 {n,} n个或多个 {n,m} n个到m个 res = re.findall(r"\w*","this is div") res = re.findall(r"\w+","this is div") res = re.findall(r"\w?","this is div") res = re.findall(r"an?","this is a an div") # 后边的n可有可无 res = re.findall(r"</?div>?","
  this is div
  ") # / 可有可无 res = re.findall(r"\w{2}","abcdefg") res = re.findall(r"\w{2,3}","this is div") # 贪婪特点：尽可能多 res = re.findall(r"\w{2,}","this is div") # 贪婪： 选择全部 print(res) ```
- 其他

  ```
  ^  以什么开始
  $   以什么结束
  a|b  a或b
  ()   表示一个组
  ```

  ``` () 组 (?P=name) 命名组、定义组 正则表达式中调用 (?P=name) \1 \2.. 正则对象中 res.group(n) res.groupdict()

res4 = re.search(r'<?P\w+>(?P.)</(?P=tag)>','

*this is div*
*') res4 = re.search(r'<?P\w+>(?P.*)</(\1)>','*
this is div
') print(res4)

```
re 模块
* re常用函数

1.re.match(pattern,string,flags=0)
```

pattern 正则表达式 steing 要匹配的字符串 flag 模式

尝试从字符串的起始位置匹配一个模式，如果不是起始位置匹配成功的话

成功返回正则对象 group(num=0) groups()

```
2.re.search(pattern,string,flags=0)
```

pattern 正则表达式 steing 要匹配的字符串 flag 模式 re.search() 扫描整个字符串并返回第一个成功的匹配 成功返回正则对象 group(num=0) groups()
```

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

pattern 正则中的模式字符串 repl 替换的字符串也可为一个函数 string 要被查找替换的原始字符串 count 模式匹配后替换的最大次数，默认0，表示替换所有的匹配

```
```
sub 替换函数
res = re.sub("山西","陕西","山西优逸客","陕西",1)
def fn(a):
res = str(int(a.group())*2)
return res
#
res = re.sub("\d+",fn,"山西优逸客 就业人数1000 高薪就业900")
print(res)
```

4.re.compile(pattern[,flags])

```
pattern  一个字符串形式的正则表达式
flags  可选，表示匹配模式
```

5.re.findall(string[,pos[,endpos]])

```
在字符串中找到正则表达式所匹配的所有子串，并且返回一个列表，如果没有找到匹配的，则返回空列表。

string  待匹配的字符串
pos  可选参数，指定字符串的起始位置，默认为0
endpos  可选参数，指定字符串的结束位置，默认为字符串的长度
```

6.re.finditer()

```
返回一个迭代器。可以循环的、一次性输出
```

7.re.split(pattern,string[,maxsplit=0,flags=0])

```
pattern  匹配的正则表达式
string  要匹配的字符串
maxsplit  分割次数
flags  标志位

## split 拆分函数
# rege = re.compile('[;,.]')     # 正则表达式,变成原子表
# res5 = rege.split('a;b,c.d')
# res5 = re.split('[;,.]','a;b,c.d')
# print(res5)
```
```

- 匹配模式

```
re.I  忽略大小写
re.L  表示特殊字符集 \w , \W,  \b ,\B, \s ,\S 依赖于当前环境
re.M  多行模式
re.S  使特殊字符匹配任何字符，包括换行，如果没有此标志，将匹配任何内容除换行符
re.X  此标志允许编写正则表达式，看起来更好。在模式中的空白将被忽略，除非当在字符类或者前面非转义反斜杠和当一条线包含一个# ，既不在字符类中或由非转义反斜杠，从最左侧的所有字符之前，这种# 通过行末尾将被忽略。
```

```

匹配模式

**rege = re.compile('[a-z]',re.I) # re.I忽略大小写**

**res = rege.findall("absdhoashODISAHNO")**

**print(res)**

**rege = re.compile("""[a-z] # 字母""",re.X)**

**res = rege.findall("""**

**absdhoashODISAHNO**

**hfadshfi**

**""")**

**print(res)**

```

bs4

PyQuery

第4节：数据存储

Mysql

redis

文件存储

第5节：项目实战

凤凰网

```
scrapy框架:

ifengspider
```

百度图片

```
# 案例一、爬取图片
# 导入
# from urllib.request import *        # urllib 是一个文件夹，request是一个模块
# from lxml import etree
# import time

# url = "http://unsplash.lofter.com/"
#
# with urlopen(url) as html:
#     # print(html.geturl())        # 没有重定向
#     # print(html.info())
#     # print(html.getcode())
#
#     text = html.read().decode("utf-8")
#
# doc = etree.HTML(text)      # 模拟的
# links = doc.xpath("//div[contains(@class,'m-post-img')]/a/span/img/@src")       # 12
个链接
# # print(links)
#
# # 下载资源
# for i in range(len(links)):
#     time.sleep(1)
#     print("正在下载第%s个"%i)
#     urlretrieve(links[i],'img/%s.jpg'%i)        # 下载资源的函数
```

网易云音乐

```
# 案例二、爬取音乐
# from urllib.request import *        # urllib 是一个文件夹，request是一个模块
# from lxml import etree
# import time
```

```
# # from fake_useragent import UserAgent        # 插件，获取报头的一个库：'User-Agent': u
a.random  'User-Agent': 'ua.chrome'
# # ua = UserAgent()
#
# url = "https://music.163.com/artist?id=2116"
# URL2 = "https://music.163.com/song/media/outer/url?id="
#
# headers = {  # 请求报头
#     'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTM
L, like Gecko) Chrome/73.0.3683.86 Safari/537.36'
#     # 'User-Agent': ua.random
#     # 'User-Agent': 'ua.chrome'
# }
#
# req = Request(url,headers=headers)       # urllib的请求对象  urlopen（url|RequestObjec
t）
#
# with urlopen(req) as html:
#     text = html.read().decode('utf-8')
#
# # 获取
# doc = etree.HTML(text)
# # print(text)     # 反爬机制，没有给服务器发送东西，就会被当作爬虫
#
# songs = doc.xpath("//ul[@class='f-hide']/li/a/text()")
# links = doc.xpath("//ul[@class='f-hide']/li/a/@href")
# ids = [ link[9::] for link in links]
# # print(list(zip(songs,ids)))
#
#
# for sid,title in zip(ids,songs):
#     time.sleep(1)
#     req2 = Request(URL2 + str(sid),headers=headers)
#     with urlopen(req2) as html:
#         urlretrieve(html.geturl(),'songs/%s.mp3'%title)     # 下载资源的函数
#         print("songs/%s.mp3 下载完成"%title)
```

豆瓣电影

```
# 例四 电影的名字和链接
from lxml import etree
from urllib.request import *
import pickle     # 持久化
import time
arr = []

# url = "https://movie.douban.com/top250?start="
```

```
# urls = [url+str(i) for i in range(0,250,25)]
#
# def spider(link):
#     time.sleep(1)
#     print("正在爬取:%s"%link)     # 每一次执行的时候给一个输出：正在爬取哪个页面（0,25,50,75...）第一个link是0，第二个link是25...
#     with urlopen(link) as html:
#         text = html.read().decode("utf-8")        # 文本格式的html
#     doc = etree.HTML(text)        # 转换成dom对象
#
#     titles = doc.xpath("//ol[@class='grid_view']/li/div/div[@class='info']/div[@class='hd']/a/span[1]/text()")
#     links = doc.xpath("//ol[@class='grid_view']/li/div/div[@class='info']/div[@class='hd']/a/@href")
#
#     arr.append(list(zip(titles,links)))       # zip就是一一对应，然后转换成列表，然后存储下来，需要一个全局的arr放入内容，调用arr.append推送
#
# for link in urls:
#     spider(link)      # 循环执行link
#
# with open("./top250.txt",'wb') as f:        # 在当前的文件中打开一个top.250文件，以二进制写入的方式
#     pickle.dump(arr,f)        # 把arr存储到f中，内容就被写过来了


# with open("./top250.txt",'rb') as f:        # 以二进制读的形式打开
#     obj = pickle.load(f)      # 把f文件中的内容加载出来
#
# print(len(obj))     # 一共10页
#
# for item in obj:
#     print(item)     # 一个大列表，里边10个小列表
```

```
### 爬取豆瓣电影详情
from urllib.request import *
import pickle,fake_useragent,xlwt
from lxml.etree import *
import time

with open('top250.txt','rb') as f:
    arr = pickle.load(f)

lists = []
for arr1 in arr:
    for title,url in arr1:
        lists.append(url)
```

```
ua = fake_useragent.UserAgent()
header = {
    'User-Agent':ua.random
}

def spider(url):
    time.sleep(1)
    req = Request(url,headers=header)
    with urlopen(req) as html:
        text = html.read().decode()
    doc = HTML(text)
    # 导演
    pl1 = "/".join(doc.xpath("//div[@id='info']/span[1]/span[@class='attrs']/a[1]/text(
)"))
    # 编剧
    pl2 = "/".join(doc.xpath("//div[@id='info']/span[2]/span[@class='attrs']/a/text()")
)
    # 主演
    pl3 = "/".join(doc.xpath("//div[@id='info']/span[3]/span[@class='attrs']/a/text()")
)
    # 类型
    pl4 = "/".join(doc.xpath("//div[@id='info']/span[@property='v:genre']/text()"))
    # 剧情简介
    pl5 = doc.xpath("//span[@property='v:summary']/text()")[0].strip()
    print("正在爬取:%s" % pl1)
    return {
        # 'dir':pl1,
        # 'edit':pl2,
        # 'actor':pl3,
        # 'type':pl4,
        # 'dis':pl5
         pl1,
         pl2,
         pl3,
         pl4,
         pl5
    }

for url in lists:
    res = list(spider(url))
    print(res)
```

第6节：Scrapy框架

Scrapy框架

安装

```
scrapy
安装
1.pywin32   http://sourceforge.net/projects/pywin32/    #选择对应版本     (用Anaconda拷过来的安装包安的pywin32)
2.Twisted   http://www.lfd.uci.edu/~gohlke/pythonlibs/#twisted   #选择对应版本
把下载文件放在项目中，pip install **.whl （对应的pip install Twisted-18.9.0-cp36-cp36m-win32.whl）
3.pip install Scrapy
```

概述

四个文件

```
items.py
数据：先创建数据（模型），以一个对象的形式存储
可以在里边写两个方法

middlewares.py
中间件：添加额外功能 (钩子函数)

pipelines.py
管道：数据会输出，输出到piplines.py 管道里，爬一个输出一个，输出一个存一个

settings.py
对爬虫进行设置
```

```
先运行 spiders 里的爬虫文件；然后保存数据类型在items.py；然后数据会输出，输出到piplines.py管道里，爬一个输出一个，输出一个存一个，还可以继续创建管道。
每个文件都有不同的功能。

## 步骤
# 1.创建项目    scrapy startproject tutorial(项目名称)
# 2.增加items.py   创建数据模型
```

```
# 3.创建爬虫文件       scrapy genspider dmoz(爬虫名称)   站点(域名)  “爬虫的主力”
# 4.运行       scrapy crawl dmoz(爬虫名称)  在项目里的任何位置都可以运行【有时会出现问题：处理方法：pip install pypiwin32或pip3 install pypiwin32 或 python -m pip install pypiwin32 }】
# scrapt.cfg   配置项
# spiders 文件里边放的是爬虫的所有内容，只要有__init__.py  就证明是一个包，可以被调用。
# douban  也是一个包
```

爬取的循环

运行流程、架构概览

第7节：常见反爬策略

构造合理请求头

```
Accept
User-Agent:第三方库
Referer
Accept-Encoding
Accept-Langukage
```

检查网站生成的Cookie

```
editthiiscookie
```

动态内容获取

限制爬取速度

处理验证码

```
超级鹰/云大码
```

隐藏

```
代理服务
```

第8节：抓包工具

抓包工具不仅可以抓电脑上的，浏览器控制台的debug，就可以看到浏览器的一些请求信息。
整个电脑上，QQ发了一条信息，QQ是通过哪个服务器发的，发给哪个服务器。
手机连上电脑wifi以后，手机的请求发给哪个服务器，服务器给你发送了一个什么请求。就叫做抓包。
爬取手机上的，只要知道手机上的地址就可以爬取。

第9节：一些tip

接下来全栈开发、web开发、网站开发
问题：装饰器是什么？面向对象编程是什么？什么是类变量、静态方法？
多关注面试题、完成项目（项目得拿出手）

第三章

第**1**节：多进程多线程编程

第**2**节：线程