

51、正则匹配，匹配日期2018-03-20？

```
url='https://sycm.taobao.com/bda/tradinganaly/overview/get_summary.json?dateRange=2018-03-20%7C2018-03-20&dateType=recent1&device=1&token=ff25b109b&_=1521595613462'
```

仍有同学问正则，其实匹配并不难，提取一段特征语句，用 (.*?) 匹配即可。

```
30 import re
31
32 url='https://sycm.taobao.com/bda/tradinganaly/overview/get_summary.json?dateRange=2018-03-20%7C2018-03-20&dateType=recent1&device=1&token=ff25b109b&_=1521595613462'
33
34 result = re.findall(r'dateRange=(.*?)%7C(.*)&', url)
35 print(result)
[('2018-03-20', '2018-03-20')]
```

52、list=[2,3,5,4,9,6]，从小到大排序，不许用sort，输出[2,3,4,5,6,9]？

利用min()方法求出最小值，原列表删除最小值，新列表加入最小值，递归调用获取最小值的函数，反复操作。

```
list_
1.ht
FOLDE
> 数
< tes
> i
> j
> k
> l
> m
> n
> o
> p
> q
> r
> s
> t
> u
> v
> w
> x
> y
> z
76 list=[2,3,5,4,9,6]
77 new_list=[]
78 def get_min(list):
79     # 获取列表最小值
80     a=min(list)
81     # 删除最小值
82     list.remove(a)
83     # 将最小值加入新列表
84     new_list.append(a)
85     # 保证最后列里面有值，递归调用获取最小值
86     # 直到所有值获取完，并加入新列表返回
87     if len(list)>0:
88         get_min(list)
89     return new_list
90
91 new_list=get_min(list)
92 print(new_list)
[2, 3, 4, 5, 6, 9] ←
```

53、写一个单列模式

因为创建对象时`__new__`方法执行，并且必须`return`返回实例化出来的对象所`cls.__instance`是否存在，不存在的话就创建对象，存在的话就返回该对象，来保证只有一个实例对象存在（单列），打印ID，值一样，说明对象同一个。

```
37 # 实例化一个单例
38 class Singleton(object):
39     __instance = None
40
41     def __new__(cls, age, name):
42         #如果类属性__instance的值为None,
43         #那么就创建一个对象，并且赋值为这个对象的引用，保证下次调用这个方法时
44         #能够知道之前已经创建过对象了，这样就保证了只有1个对象
45         if not cls.__instance:
46             cls.__instance = object.__new__(cls)
47         return cls.__instance
48
49 a = Singleton(18, "dongGe")
50 b = Singleton(8, "dongGe")
51
52 print(id(a))
53 print(id(b))
54
55 a.age = 19 #给a指向的对象添加一个属性
56 print(b.age) #获取b指向的对象的age属性
57
2223334542920
2223334542920
19
```

54、保留两位小数？

题目本身只有`a = "%.03f"%1.3335`,让计算a的结果，为了扩充保留小数的思路，提供`round`方法（数值，保留位数）。

```
59 # 保留2位小数
60 a = "%.03f"%1.3335 ←
61 print(a,type(a)) ←
62 b=round(float(a),1) ←
63 print(b) ←
64 b=round(float(a),2) ←
65 print(b)
66
67 A = zip(("a","b","c","d","e"),(1,2,3,4,5))
68 A0 = dict(A)
69 # print(A0)
1.333 <class 'str'>
1.3
1.33
```

55、求三个方法打印结果？

fn("one",1) 直接将键值对传给字典；

fn("two",2)因为字典在内存中是可变数据类型，所以指向同一个地址，传了新的额参数后，会相当于给字典增加键值对；

fn("three",3,{})因为传了一个新字典，所以不再是原先默认参数的字典。

```
113 def fn(k,v,dic={}):
114     dic[k]=v
115     print(dic)
116 fn("one",1)
117 fn("two",2)
118 fn("three",3,{})
119
{'one': 1}
{'one': 1, 'two': 2}
{'three': 3}
[Finished in 0.3s]
```

56、列出常见的状态码和意义？

200 OK

请求正常处理完毕

204 No Content

请求成功处理，没有实体的主体返回

206 Partial Content

GET范围请求已成功处理

301 Moved Permanently

永久重定向，资源已永久分配新URI

202 Found

302 Found

临时重定向，资源已临时分配新URI

303 See Other

临时重定向，期望使用GET定向获取

304 Not Modified

发送的附带条件请求未满足

307 Temporary Redirect

临时重定向，POST不会变成GET

400 Bad Request

请求报文语法错误或参数错误

401 Unauthorized

需要通过HTTP认证，或认证失败

403 Forbidden

请求资源被拒绝

404 Not Found

无法找到请求资源（服务器无理由拒绝）

500 Internal Server Error

服务器故障或Web应用故障

503 Service Unavailable

服务器超负载或停机维护

57、分别从前端、后端、数据库阐述web项目的性能优化？

前端优化：

1、减少http请求，例如制作精灵图。

2、html和CSS放在页面上部，javascript放在页面下面，因为js加载比HTML和Css加载慢，所以要优先加载html和css，以防页面显示不全，性能差，也影响用户体验差。

后端优化：

1、缓存存储读写次数高，变化少的数据，比如网站首页的信息、商品的信息等。应用程序读取数据时，一般是先从缓存中读取，如果读取不到或数据已失效，再访问磁盘数据库，并将数据再次写入缓存。

2、异步方式，如果有耗时操作，可以采用异步，比如celery。

3、代码优化，避免循环和判断次数太多，如果多个if else判断，优先判断最有可能先发生的情况。

数据库优化：

1、如有条件，数据可以存放于redis，读取速度快。

2、建立索引、外键等。

58、使用pop和del删除字典中的"name"字段，dic={"name":"zs","age":18}？

```
158
159 dic = {"name": "zs", "age": 18}
160 dic.pop("name") ←
161 print(dic)
162 dic = {"name": "zs", "age": 18}
163 del dic["name"] ←
164 print(dic)

{'age': 18}
{'age': 18}
[Finished in 0.3s]
```

59、列出常见MYSQL数据存储引擎？

InnoDB：支持事务处理，支持外键，支持崩溃修复能力和并发控制。如果需要对事务的完整性要求比较高（比如银行），要求实现并发控制（比如售票），那选择InnoDB有很大的优势。如果需要频繁的更新、删除操作的数据，也可以选择InnoDB，因为支持事务的提交（commit）和回滚（rollback）。

MyISAM：插入数据快，空间和内存使用比较低。如果表主要是用于插入新记录和读出记录，那么选择MyISAM能实现处理高效率。如果应用的完整性、并发性要求比较低，也可以使用。

MEMORY：所有的数据都在内存中，数据的处理速度快，但是安全性不高。如果需要很快的读写速度，对数据的安全性要求较低，可以选择MEMOYEY。它对表的大小有要求，不能建立太大的表。所以，这类数据库只使用在相对较小的数据库表。

60、计算代码运行结果，zip函数历史文章已经说了，得出[("a",1),("b",2), ("c",3),("d",4),("e",5)]？

```
72 A = zip(("a","b","c","d","e"),(1,2,3,4,5))
73 A0 = dict(A)
74 A1=range(10)
75 A2=[i for i in A1 if i in A0]
76 A3=[A0[s] for s in A0]
77 print("A0",A0)
78 print(list(zip(("a","b","c","d","e"),(1,2,3,4,5))))
79 print(A2)
80 print(A3)
A0 {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
[('a', 1), ('b', 2), ('c', 3), ('d', 4), ('e', 5)]
[]
[1, 2, 3, 4, 5]
```

dict()创建字典新方法

```
165
166 s = dict([["name","zs"],["age",18]])
167 print(s)
168
```

列表

```
169 s = dict([("name","zs"),("age",18)])
170 print(s)
1
2
3
4
5
A0 {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
[('a', 1), ('b', 2), ('c', 3), ('d', 4), ('e', 5)]
[]
[1, 2, 3, 4, 5]
{'age': 18}
{'age': 18}
{'name': 'zs', 'age': 18} ←
{'name': 'zs', 'age': 18} ←
[Finished in 0.6s]
```

61、简述同源策略？

同源策略需要同时满足以下三点要求：

1) 协议相同

2) 域名相同

3) 端口相同

http:www.test.com与https:www.test.com 不同源——协议不同

http:www.test.com与http:www.admin.com 不同源——域名不同

http:www.test.com与http:www.test.com:8081 不同源——端口不同

只要不满足其中任意一个要求，就不符合同源策略，就会出现“跨域”

62、简述cookie和session的区别？

1 , session 在服务器端 , cookie 在客户端 (浏览器) 。

2 、 session 的运行依赖 session id , 而 session id 是存在 cookie 中的 , 也就是说 , 如果浏览器禁用了 cookie , 同时 session 也会失效 , 存储Session时 , 键与Cookie中的sessionid相同 , 值是开发人员设置的键值对信息 , 进行了base64编码 , 过期时间由开发人员设置。

3 、 cookie安全性比session差。

63、简述多线程、多进程？

进程：

- 1、操作系统进行资源分配和调度的基本单位，多个进程之间相互独立。
- 2、稳定性好，如果一个进程崩溃，不影响其他进程，但是进程消耗资源大，开启的进程数量有限制。

线程：

- 1、CPU进行资源分配和调度的基本单位，线程是进程的一部分，是比进程更小的能独立运行的基本单位，一个进程下的多个线程可以共享该进程的所有资源。
- 2、如果IO操作密集，则可以多线程运行效率高，缺点是如果一个线程崩溃，都会造成进程的崩溃。

应用：

IO密集的用多线程，在用户输入，sleep 时候，可以切换到其他线程执行，减少等待的时间。

CPU密集的用多进程，因为假如IO操作少，用多线程的话，因为线程共享一个全局解释器锁，当前运行的线程会霸占GIL，其他线程没有GIL，就不能充分利用多核CPU的优势。

64、简述any()和all()方法？

any()：只要迭代器中有一个元素为真就为真。

all()：迭代器中所有的判断项返回都是真，结果才为真。

python中什么元素为假？

答案：(0、空字符串、空列表、空字典、空元组、None、 False)

```
In [3]: bool(0)
```

```
In [3]: bool(0)
Out[3]: False

In [4]: bool("")
Out[4]: False

In [5]: bool([])
Out[5]: False

In [6]: bool(())
Out[6]: False

In [7]: bool({})
Out[7]: False

In [8]: bool(None)
Out[8]: False

In [9]:
```

测试all()和any()方法

```
In [9]: a = [True, False]
In [10]: any(a)
Out[10]: True

In [11]: all(a)
Out[11]: False

In [12]: a = ""

In [13]: any(a)
Out[13]: False

In [14]: b=['good', 'good', 'good', 'bad']
In [15]: all(b)
Out[15]: True

In [16]: b=['good', 'good', 'good', '']
In [17]: all("b") ← 这里注意是"b"
Out[17]: True

In [18]: any("b") ←
```

```
In [18]: any(' ')
Out[18]: True

In [19]: b
Out[19]: ['good', 'good', 'good', '']

In [20]: b=['good', 'good', 'good', ""]

In [21]: b
Out[21]: ['good', 'good', 'good', '']

In [22]: all(b)
Out[22]: False
```

65、`IOError`、`AttributeError`、`ImportError`、`IndentationError`、`IndexError`、`KeyError`、`SyntaxError`、`NameError`分别代表什么异常？

`IOError`：输入输出异常

`AttributeError`：试图访问一个对象没有的属性

`ImportError`：无法引入模块或包，基本是路径问题

`IndentationError`：语法错误，代码没有正确的对齐

`IndexError`：下标索引超出序列边界

`KeyError`：试图访问你字典里不存在的键

`SyntaxError`：Python代码逻辑语法出错，不能执行

`NameError`；使用一个还未赋予对象的变量

66、python中`copy`和`deepcopy`区别？

1、复制不可变数据类型

不管`copy`还是`deepcopy`，都是同一个地址当浅复制的值是不可变对象（数值，字符串，元组）时和="赋值"的情况一样，对象的id值与浅复制原来的值相同。

```
1.htm 274 b = a
FOLDE 275 c = copy.copy(a)
> 数 276 d = copy.deepcopy(a)
< tes 277 print(a,id(a))
> i 278 print(b,id(b))
> e 279 print(c,id(c))
> f 280 print(d,id(d))
```

```
哈哈 2348789079624
哈哈 2348789079624
哈哈 2348789079624
哈哈 2348789079624
[Finished in 0.5s]
```

2、复制的值是可变对象（列表和字典）

浅拷贝copy有两种情况：

第一种情况：复制的对象中无复杂子对象，原来值的改变并不会影响浅复制的值，同时浅复制的值改变也并不会影响原来的值。原来值的id值与浅复制原来的值不同。

第二种情况：复制的对象中有复杂子对象（例如列表中的一个子元素是一个列表），改变原来的值中的复杂子对象的值，会影响浅复制的值。

深拷贝deepcopy：完全复制独立，包括内层列表和字典。

```
# copy和deepcopy浅深复制
# list = [1,2,{name:"ss"}]
list = [1,2,[3,4]]
a=copy.copy(list)
b=copy.deepcopy(list)
# 外层列表: [1,2,[3,4]]
# 内层列表: [3,4]
# 复杂子对象: [3,4]，我们认为包含嵌套结构的内层列表(字典)为复杂子对象
# 简单子对象: 1,2
print("测试原始的数据和copy和deepcopy后的结果及ID地址")
print("结果表明对于外层列表来说，三者是独立的对象")
print("原始数据和id",list,id(list))
print("原始数据和id",a,id(a))
print("原始数据和id",b,id(b))
print("*****")
print("测试修改外层列表的简单子对象，也就是修改1或者2")
print("结果表明修改了原始list之后，a和b并没有随之改变，符合我们的正常逻辑，因为是三个不同的对象")
list[0]=10
print("将1改成10结果",list)
print("将1改成10结果",a)
print("将1改成10结果",b)
print("*****")
print("测试内层列表的值的修改，也就是测试复杂子对象的值的修改")
print("结果表明copy浅拷贝并没有真正将内层列表(字典)独立拷贝出来，导致修改了list内层列表(字典)后a的内层列表(字典)值也变了")
print("结果表明deepcopy深拷贝可以将内层列表和(字典)独立拷贝出来，所以b的内层列表(字典)值不变")
list[2][0]=5
print("将3改成5结果",list)
print("将3改成5结果",a)
print("将3改成5结果",b)
```

```
测试原始数据和copy和deepcopy后的结果及ID地址
结果表明对于外层列表来说，三者是独立的对象
原始数据和id [1, 2, [3, 4]] 2465779174664
原始数据和id [1, 2, [3, 4]] 2465784158664
原始数据和id [1, 2, [3, 4]] 2465779174856
*****
测试修改外层列表的简单子对象，也就是修改1或者2
结果表明修改了原始list之后，a和b并没有随之改变，符合我们的正常逻辑，因为是三个不同的对象
将1改成10结果 [10, 2, [3, 4]]
将1改成10结果 [1, 2, [3, 4]]
将1改成10结果 [1, 2, [3, 4]]
*****
测试内层列表的值的修改，也就是测试复杂子对象的值的修改
结果表明copy浅拷贝并没有真正将内层列表(字典)独立拷贝出来，导致修改了list内层列表(字典)后a的内层列表(字典)值也变了
结果表明deepcopy深拷贝可以将内层列表和(字典)独立拷贝出来，所以b的内层列表(字典)值不变
将3改成5结果 [10, 2, [5, 4]]
将3改成5结果 [1, 2, [5, 4]]
将3改成5结果 [1, 2, [3, 4]]
```

67、列出几种魔法方法并简要介绍用途？

`__init__`：对象初始化方法

`__new__`：创建对象时候执行的方法，单列模式会用到

`__str__`；当使用print输出对象的时候，只要自己定义了`__str__(self)`方法，那么就会打印从在这个方法中return的数据

`__del__`：删除对象执行的方法

68、C:\Users\ry-wu.junya\Desktop>python 1.py 22 33命令行启动程序并传参，`print(sys.argv)`会输出什么数据？

文件名和参数构成的列表

```
C:\Users\ry-wu.junya\Desktop>python 1.py 22 33
['1.py', '22', '33']
```

69、请将`[i for i in range(3)]`改成生成器？

生成器是特殊的迭代器

1、列表表达式的【】改为()即可变成生成器；

2、函数在返回值得时候出现yield就变成生成器，而不是函数了；

中括号换成小括号即可，有没有惊呆了

```
In [1]: a = (i for i in range(3))
```

```
In [2]: type(a)
```

```
Out[2]: generator
```

```
In [3]: -
```

70、a = " hehheh ",去除收尾空格？

```
In [3]: a = " hehheh "
```

```
In [4]: a.strip()
```

```
Out[4]: 'hehheh'
```

```
In [5]: -
```

71、举例sort和sorted对列表排序，list=[0,-1,3,-10,5,9]？

```
187  
188 # 列表排序  
189 list = [0, -1, 3, -10, 5, 9]  
190 list.sort(reverse=False)  
191 print("list.sort在list基础上修改, 无返回值", list)  
192  
193 list = [0, -1, 3, -10, 5, 9]  
194 res = sorted(list, reverse=False)  
195 print("sorted有返回值是新的list", res)
```

```
    5 196 print("返回值",res)
数字大于2了
list.sort在list基础上修改，无返回值 [-10, -1, 0, 3, 5, 9]
sorted有返回值是新的list [0, -1, 3, -10, 5, 9]
返回值 [-10, -1, 0, 3, 5, 9]
```

72、对list排序foo = [-5,8,0,4,9,-4,-20,-2,8,2,-4],使用lambda函数从小到大排序？

```
    199
  200 foo = [-5,8,0,4,9,-4,-20,-2,8,2,-4]
  201 a=sorted(foo,key=lambda x:x)
  202 # foo.sort()
  203 print(a)
  204
数字大于2了
[-20, -5, -4, -4, -2, 0, 2, 4, 8, 8, 9]
```

73、使用lambda函数对list排序foo = [-5,8,0,4,9,-4,-20,-2,8,2,-4]，输出结果为

[0,2,4,8,8,9,-2,-4,-4,-5,-20]，正数从小到大，负数从大到小？

(传两个条件，x<0和abs(x))

```
    199
  200 foo = [-5,8,0,4,9,-4,-20,-2,8,2,-4]
  201 a=sorted(foo,key=lambda x:(x<0,abs(x)))
  202 # foo.sort()
  203 print(a)
  204
数字大于2了
[0, 2, 4, 8, 8, 9, -2, -4, -4, -5, -20]
```

74、列表嵌套字典的排序，分别根据年龄和姓名排序？

```
foo = [{"name":"zs","age":19}, {"name":"ll", "age":54},
```

```
{"name":"wa","age":17}, {"name":"df", "age":23}]
```

```

> In [205]: foo = [{"name": "zs", "age": 19}, {"name": "ss", "age": 54}, {"name": "wa", "age": 17}, {"name": "df", "age": 23}]
> In [206]: a=sorted(foo,key=lambda x:x["age"],reverse=True)
> In [207]: print(a)
> In [208]: a=sorted(foo,key=lambda x:x["name"])
> In [209]: print(a)
> In [210]: 
> In [211]: 

数字大于2了
[8, 0, 4, 9, 8, 2, -5, -4, -20, -2, -4]
[{"name": "ll", "age": 54}, {"name": "df", "age": 23}, {"name": "zs", "age": 19}, {"name": "wa", "age": 17}]
[{"name": "df", "age": 23}, {"name": "ll", "age": 54}, {"name": "wa", "age": 17}, {"name": "zs", "age": 19}]
[Finished in 0.1s]

```

75、列表嵌套元组，分别按字母和数字排序？

```

> In [212]: 
> In [213]: foo = [("zs",19),("ll",54),
> In [214]: ("wa",17),("df",23)]
> In [215]: a=sorted(foo,key=lambda x:x[1],reverse=True)
> In [216]: print(a)
> In [217]: a=sorted(foo,key=lambda x:x[0])
> In [218]: print(a)

数字大于2了
[8, 0, 4, 9, 8, 2, -5, -4, -20, -2, -4]
[('ll', 54), ('df', 23), ('zs', 19), ('wa', 17)]
[('df', 23), ('ll', 54), ('wa', 17), ('zs', 19)]
[Finished in 0.2s]

```

76、列表嵌套列表排序，年龄数字相同怎么办？

```

> In [213]: foo = [["zs",19],["ll",54],
> In [214]: ["wa",23],["df",23],["xf",23]]
> In [215]: a=sorted(foo,key=lambda x:(x[1],x[0]))
> In [216]: print(a)
> In [217]: a=sorted(foo,key=lambda x:x[0])
> In [218]: print(a)

数字大于2了
[8, 0, 4, 9, 8, 2, -5, -4, -20, -2, -4]
[[['zs', 19], ['df', 23], ['wa', 23], ['xf', 23], ['ll', 54]]]
[[['df', 23], ['ll', 54], ['wa', 23], ['xf', 23], ['zs', 19]]]

```

77、根据键对字典排序（方法一，zip函数）？

```

> In [219]: 
> In [220]: 
> In [221]: dic = {"name": "zs", "sex": "man", "city": "bj"}
> In [222]: foo=zip(dic.keys(),dic.values())
> In [223]: foo = [i for i in foo]
> In [224]: print("字典转成列表嵌套元组",foo)

字典转成列表嵌套元组

```

```

225 b=sorted(foo,key=lambda x:x[0]) ← 字典嵌套元组排序
> 数
226 print("根据键排序",b)
227 new_dic = {i[0]:i[1] for i in b} ← 排序完构造新字典
228 print("字典推导式构造新字典",new_dic)
229
230
数字大于2了
字典转成列表嵌套元组 [('name', 'zs'), ('sex', 'man'), ('city', 'bj')]
根据键排序 [('city', 'bj'), ('name', 'zs'), ('sex', 'man')]
字典推导式构造新字典 {'city': 'bj', 'name': 'zs', 'sex': 'man'}

```

78、根据键对字典排序（方法二，不用zip）？

有没有发现dic.items()和zip(dic.keys(), dic.values())都是为了构造列表嵌套字典的结构，方便后面用sorted()构造排序规则

```

231 dic = {"name": "zs", "sex": "man", "city": "bj"}
> 数
232 print("字典转成列表嵌套元组",foo)
233 print(dic.items())
234 b=sorted(dic.items(),key=lambda x:x[0]) ←
235 print("根据键排序",b) ←
236 new_dic = {i[0]:i[1] for i in b}
237 print("字典推导式构造新字典",new_dic)

数字大于2了
字典转成列表嵌套元组 [('name', 'zs'), ('sex', 'man'), ('city', 'bj')]
dict_items([('name', 'zs'), ('sex', 'man'), ('city', 'bj')])
根据键排序 [('city', 'bj'), ('name', 'zs'), ('sex', 'man')]
字典推导式构造新字典 {'city': 'bj', 'name': 'zs', 'sex': 'man'}
[Finished in 0.1s]

```

79、列表推导式、字典推导式、生成器？

```

239 import random
> 数
240 td_list=[i for i in range(10)]
241 print("列表推导式",td_list,type(td_list))
242
243 ge_list=(i for i in range(10))
244 print("生成器",ge_list)
245
246 dic={k:random.randint(4,9) for k in ["a","b","c","d"]}
247 print("字典推导式",dic,type(dic))

数字大于2了
列表推导式 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9] <class 'list'>
生成器 <generator object <genexpr> at 0x0000018F61A0AF68>
字典推导式 {'a': 7, 'b': 6, 'c': 9, 'd': 8} <class 'dict'>
[Finished in 0.2s]

```

80、最后出一道检验题目，根据字符串长度排序，看排序是否灵活运用？

```
> 数 249
< tes 250 s = ["ab", "abc", "a", "djkj"]
<      b = sorted(s, key=lambda x: len(x))
<      print(b, s)
<      s.sort(key=len)
<      print(s)
<      255

数字大于2了
['a', 'ab', 'abc', 'djkj'] ['ab', 'abc', 'a', 'djkj']
['a', 'ab', 'abc', 'djkj']
[Finished in 0.1s]
```

81、举例说明SQL注入和解决办法？

当以字符串格式化书写方式的时候，如果用户输入的有;+SQL语句，后面的SQL语句会执行，比如例子中的SQL注入会删除数据库demo

```
255  
256 # SQL注入  
257 # 例如一条SQL语句是:  
258 input_name="zs"  
259 sql='select * from demo where name="%s"' % input_name  
260 print("正常SQL语句",sql)  
261  
262 input_name="zs;drop database demo"  
263 sql='select * from demo where name="%s"' % input_name  
264 print("SQL注入语句",sql)  
265
```

数字大于2了 正常查询

正常SQL语句 select * from demo where name="zs"

SQL注入语句 select * from demo where name="zs;drop database demo"

[Finished in 0.2s] SQL注入

解决方式：通过传参数方式解决SQL注入

```
6 # 通过参数化方式解决
7 params=[input_name]
8 count = cs1.execute('select * from goods where name=%s', params)
```

82. s="info:xiaoZhang 33 shandong",用正则切分字符串输出['info', 'xiaoZhang', '33', 'shandong'] ?

|表示或，根据冒号或者空格切分

```
321
322     s = "info:xiaoZhang 33 shandong"
323
324     res=re.split(r":| ",s)
325
326     print(res)
```

数字大于2了

```
['info', 'xiaoZhang', '33', 'shandong']
[Finished in 0.1s]
```

83、正则匹配以163.com结尾的邮箱？

```
321
322     email_list = ["xiaoWang@163.com", "xiaoWang@163.comheihei", ".com.xiaowang@qq.com"]
323
324     for email in email_list:
325         ret = re.match("[\w]{4,20}@163\.com$", email)
326         if ret:
327             print("%s 是符合规定的邮件地址,匹配后的结果是:%s" % (email, ret.group()))
328         else:
329             print("%s 不符合要求" % email)
数字大于2了
xiaoWang@163.com 是符合规定的邮件地址,匹配后的结果是:xiaoWang@163.com
xiaoWang@163.comheihei 不符合要求
.com.xiaowang@qq.com 不符合要求
```

84、递归求和？

```
321
322     # 递归完成1+2+3+...+10的和
323     def get_sum(num):
324         if num>=1:
325             res=num+get_sum(num-1)
326         else:
327             res = 0
328
329         return res
330
331     res=get_sum(10)
332     print(res)
```

数字大于2了

```
['info', 'xiaoZhang', '33', 'shandong']
55 ←
[Finished in 1.9s]
```

85、Python字典和json字符串相互转化方法？

json.dumps()字典转json字符串，json.loads()json转字典

```
288 import json
289 dic = {"name": "zs"}
290 res = json.dumps(dic)
291 print(res, type(res))
292 ret = json.loads(res)
293 print(ret, type(ret))
294
```

数字大于2了

```
{"name": "zs"} <class 'str'>
{'name': 'zs'} <class 'dict'>
[Finished in 0.2s]
```

86、MyISAM 与 InnoDB 区别？

1、InnoDB 支持事务，MyISAM 不支持，这一点是非常之重要。事务是一种高级的处理方式，如在一些列增删改中只要哪个出错还可以回滚还原，而 MyISAM 就不可以了；

2、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；

3、InnoDB 支持外键，MyISAM 不支持；

4、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；

5、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。

87、统计字符串中某字符出现次数？

```
294
295 str = "张三 美国 张三 哈哈 张 三"
296 res = str.count("张三")
297 print(res)           ↗
数字大于2了
2 ←
[Finished in 0.5s]
```

88、字符串转化大小写？

```
> i 298
ノ 299 str = "HHHuuu"
カ 300 print(str.upper())
タ 301 print(str.lower())
日
```

数字大于2了

2

HHHUUU

hh.....

```
nnnunuu  
[Finished in 1.9s]
```

89、用两种方法去空格？

```
303 str = "hello world ha ha"  
304 res=str.replace(" ","")  
305 print(res)  
306  
307 list=str.split(" ")  
308 res="".join(list)  
309 print(res)
```

数字大于2了

helloworldhaha

helloworldhaha

90、正则匹配不是以4和7结尾的手机号？

```
313 tels = ["13100001234", "18912344321", "10086", "18800007777"]  
314 for tel in tels:  
315     ret = re.match("1\d{9}[0-3,5-6,8-9]", tel)  
316     if ret:  
317         print("想要的结果",ret.group())  
318     else:  
319         print("%s 不是想要的手机号" % tel)
```

数字大于2了

13100001234 不是想要的手机号 ← 4结果

想要的结果 18912344321

10086 不是想要的手机号 ← 不是手机号

18800007777 不是想要的手机号 ← 7结尾

[Finished in 1.9s]

91、简述Python引用计数机制？

python垃圾回收主要以引用计数为主，标记-清除和分代清除为辅的机制，其中标记-清除和分代回收主要是为了处理循环引用的难题。

引用计数算法

当有1个变量保存了对象的引用时，此对象的引用计数就会加1。

当使用del删除变量指向的对象时，如果对象的引用计数不为1，比如3，那么此时只会让这个引用计数减1，即变为2，当再次调用del时，变为1，如果再调用1次del，此时会真的把对象进行删除。

```
331 import time
332 class Animal(object):
333     # 创建完对象后会自动被调用
334     def __init__(self, name):
335         print('__init__方法被调用')
336         self.__name = name
337
338     # 当对象被删除时，会自动被调用
339     def __del__(self):
340         print('__del__方法被调用')
341         print("%s对象马上被干掉了..." % self.__name)
342 cat = Animal("波斯猫")
343 cat2 = cat
344 cat3 = cat
345 print(id(cat), id(cat2), id(cat3))
346 print("---马上删除cat对象")
347 del cat
348 print("---马上删除cat2对象")
349 del cat2
350 print("---马上删除cat3对象")
351 del cat3
352 print("程序2秒钟后结束")
353 time.sleep(2)
```

2416260856184 2416260856184 2416260856184
---马上删除cat对象
---马上删除cat2对象
---马上删除cat3对象
__del__方法被调用
波斯猫对象马上被干掉了...
程序2秒钟后结束

cat和cat2和cat3的内存ID一致
说明是同一个对象

del方法只有在对象真正被删除时候才调用打印（也就是cat3被删除时候）

92、int("1.4"),int(1.4)输出结果？

int("1.4")报错，int(1.4)输出1

93、列举3条以上PEP8编码规范？

1、顶级定义之间空两行，比如函数或者类定义。

2、方法定义、类定义与第一个方法之间，都应该空一行

3、三引号进行注释

4、使用Pycharm、Eclipse一般使用4个空格来缩进代码

94、正则表达式匹配第一个URL？

findall结果无需加group(),search需要加group()提取

```
354
355 s = '
356 res=re.findall(r'https://.*\.jpg',s)[0]
357 print(res)
358 res=re.search(r'https://.*\.jpg',s)
359 print(res.group())
360 [Finished in 0.2s]
```

数字大于2了
https://rpic.douyucdn.cn/appCovers/2016/11/13/1213973_201611131917_small.jpg
https://rpic.douyucdn.cn/appCovers/2016/11/13/1213973_201611131917_small.jpg
[Finished in 0.2s]

95、正则匹配中文？

```
364
365 title = '你好, hello, 世界'
366 pattern = re.compile(r'[\u4e00-\u9fa5]+')
367 result = pattern.findall(title)
368
369 print (result)
370 [Finished in 1.9s]
```

数字大于2了
['你好', '世界']
[Finished in 1.9s]

96、简述乐观锁和悲观锁？

悲观锁，就是很悲观，每次去拿数据的时候都认为别人会修改，所以每次在拿数据的时候都会上锁，这样别人想拿这个数据就会block直到它拿到锁。传统的关系型数据库里边就用到了很多这种锁机制，比如行锁，表锁等，读锁，写锁等，都是在做操作之前先上锁。

乐观锁，就是很乐观，每次去拿数据的时候都认为别人不会修改，所以不会上锁，但是在更新的时候会判断一下在此期间别人有没有去更新这个数据，可以使用版本号等机制，乐观锁适用于多读的应用类型，这样可以提高吞吐量。

97、r、r+、rb、rb+文件打开模式区别？

模式较多，比较下背背记记即可

| 访问模式 | 说明 |
|------|---|
| 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。 |

98、Linux命令重定向 > 和 >> ?

Linux 允许将命令执行结果 重定向到一个文件

将本应显示在终端上的内容输出 / 追加到指定文件中

> 表示输出，会覆盖文件原有的内容

>> 表示追加，会将内容追加到已有文件的末尾

用法示例：

将 echo 输出的信息保存到 1.txt 里echo Hello Python > 1.txt

将 tree 输出的信息追加到 1.txt 文件的末尾tree >> 1.txt

99、正则表达式匹配出

www.itcast.cn

？

前面的<>和后面的<>是对应的，可以用此方法

```
372
373 labels = ["<html><h1>www.itcast.cn</h1></html>", "<html><h1>www.itcast.cn</h2></html>"]
374 for label in labels:
375     ret = re.match(r"<(\w*)><(\w*)>.*?</\2></\1>", label)
376     if ret:
377         print("%s 是符合要求的标签" % ret.group())
378     else:
379         print("%s 不符合要求" % label)

数字大于2了
<html><h1>www.itcast.cn</h1></html> 是符合要求的标签
<html><h1>www.itcast.cn</h2></html> 不符合要求
[Finished in 0.2s]
```

100、Python传参数是传值还是传址？

Python中函数参数是引用传递（注意不是值传递）。对于不可变类型（数值型、字符串、元组），因变量不能修改，所以运算不会影响到变量自身；而对于可变类型（列表字典）来说，函数体运算可能会更改传入的参数变量。

```
381
382 def selfAdd(a):
383     a+=a
```

```
384  
385 a_int = 1  
386 print(a_int)  
387 selfAdd(a_int)  
388 print(a_int)  
389  
390 a_list = [1, 2]  
391 print(a_list)  
392 selfAdd(a_list)  
393 print(a_list)  
394
```

数字大于2了

```
1  
1  
[1, 2]  
[1, 2, 1, 2]  
[Finished in 0.2s]
```

101、求两个列表的交集、差集、并集？

```
# 从网上找的代码，测试了下，非常高效！  
# 定义两个列表  
a = [1, 2, 3, 4]  
b = [4, 3, 5, 6]  
# 定义两个空列表  
jj1 = [i for i in a if i in b] # 在a中的i，并且在b中，就是交集  
jj2 = list(set(a).intersection(set(b)))  
bj1 = list(set(a).union(set(b))) # 用union方法  
# 打印结果  
cj1 = list(set(b).difference(set(a))) # b中有而a没有的  
cj2 = list(set(a).difference(set(b))) # a中有而b没有的  
print("交集", jj1)  
print("交集", jj2)  
print("并集", bj1)  
# print("并集", bj2)  
print("差集", cj1)  
print("差集", cj2)
```

交集 [3, 4]
交集 [3, 4]
并集 [1, 2, 3, 4, 5, 6]
差集 [5, 6]
差集 [1, 2]

102、生成0-100的随机数？

random.random()生成0-1之间的随机小数，所以乘以100。

```
> 18
> 19 import random
> 20 res1=random.random()
> 21 res2=random.choice(range(0,101))
> 22 res3=random.randint(1,100)
> 23 print(res1)
> 24 print(res2)
> 25 print(res3)
20.575902314280647
44
64
```

随机小数

随机整数

103、lambda匿名函数好处？

精简代码，lambda省去了定义函数，map省去了写for循环过程

```
> 29 # lambda好处
> 30 a = ["苏州","中国","哈哈","","","","日本","","","","德国"]
> 31 res=list(map(lambda x:"填充值" if x=="" else x,a))
> 32 print(res)
['苏州', '中国', '哈哈', '填充值', '填充值', '日本', '填充值', '填充值', '德国']
[Finished in 0.2s]
```

104、常见的网络传输协议？

UDP、TCP、FTP、HTTP、SMTP等等

105、单引号、双引号、三引号用法？

1、单引号和双引号没有什么区别，不过单引号不用按shift，打字稍微快一点。表示字符串的时候，单引号里面可以用双引号，而不用转义字符，反之亦然。

```
'She said:"Yes." ' or "She said: 'Yes.' "
```

2、但是如果直接用单引号扩住单引号，则需要转义，像这样：

' She said:\'Yes.\' '

3、三引号可以直接书写多行，通常用于大段，大篇幅的字符串

hello

world

....

106、Python垃圾回收机制？

Python垃圾回收主要以引用计数为主，标记-清除和分代清除为辅的机制，其中标记-清除和分代回收主要是为了处理循环引用的难题。

引用计数算法

当有1个变量保存了对象的引用时，此对象的引用计数就会加1。

当使用del删除变量指向的对象时，如果对象的引用计数不为1，比如3，那么此时只会让这个引用计数减1，即变为2，当再次调用del时，变为1，如果再调用1次del，此时会真的把对象进行删除。

The screenshot shows a Python script running in a terminal or code editor. A red box highlights a portion of the code where three variables (cat, cat2, cat3) all point to the same object. Below the code, a memory dump shows three identical memory addresses (1438429206680) for cat, cat2, and cat3. This indicates that even though multiple variables point to the same object, its reference count remains at 3 until it is explicitly deleted.

```
80     print('__init__方法被调用')
81     self.__name = name
82
83     # 拆构方法
84     # 当对象被删除时，会自动被调用
85     def __del__(self):
86         print('__del__方法被调用')
87         print("%s对象马上被干掉了..." % self.__name)
88
89 cat = Animal("波斯猫")
90 cat2 = cat
91 cat3 = cat
92 print(id(cat), id(cat2), id(cat3))
93 print("---马上删除cat对象")
94 del cat
95
96 print(id(cat2), id(cat3))
97 print("---马上删除cat2对象")
98 del cat2
99
100 print(id(cat3))
101 print("---马上删除cat3对象")
102 del cat3
```

cat: cat2和cat3, 三者ID一样, 说明是一个
对象

_init__方法被调用
1438429206680 1438429206680 1438429206680
...马上删除cat对象
1438429206680 1438429206680 1438429206680
...马上删除cat2对象
1438429206680 1438429206680
...马上删除cat3对象

del方法被调用
删除cat3后，cat，cat2，cat3的引用都被删除，再用del方法，所以打印了del方法带调用这句话。
而上面还有引用的时候，就不会打印这句话。

107、HTTP请求中get和post区别？

- 1、GET请求是通过URL直接请求数据，数据信息可以在URL中直接看到，比如浏览器访问；而POST请求是放在请求头中的，我们是无法直接看到的；
- 2、GET提交有数据大小的限制，一般是不超过1024个字节，而这种说法也不完全准确，HTTP协议并没有设定URL字节长度的上限，而是浏览器做了些处理，所以长度依据浏览器的不同有所不同；POST请求在HTTP协议中也没有做说明，一般来说是没有设置限制的，但是实际上浏览器也有默认值。总体来说，少量的数据使用GET，大量的数据使用POST；
- 3、GET请求因为数据参数是暴露在URL中的，所以安全性比较低，比如密码是不能暴露的，就不能使用GET请求；POST请求中，请求参数信息是放在请求头的，所以安全性较高，可以使用。在实际中，涉及到登录操作的时候，尽量使用HTTPS请求，安全性更好。

108、Python中读取Excel文件的方法？

应用数据分析库pandas

The screenshot shows a terminal window on the left and an Excel spreadsheet on the right. In the terminal, Python code is running:

```
P 50 import pandas as pd
> 51 df=pd.read_excel("333.xlsx")
> 52 print(df)
> 53
```

A red arrow points from the word "read_excel" in the code to the corresponding method name in the Excel ribbon menu bar. The Excel spreadsheet displays a table with columns: 姓名 (Name), 年龄 (Age), and 城市 (City). The data rows are: 张三 (Age 11, City Beijing), 李四 (Age 12, City Shenzhen), 王五 (Age 12, City Shanghai). The status bar at the bottom of the terminal says "[Finished in 0.8s]".

109、简述多线程、多进程？

进程：

- 1、操作系统进行资源分配和调度的基本单位，多个进程之间相互独立。
- 2、稳定性好，如果一个进程崩溃，不影响其他进程，但是进程消耗资源大，开启的进程数量有限制。

线程：

- 1、CPU进行资源分配和调度的基本单位，线程是进程的一部分，是比进程更小的能独立运行的基本单位，一个进程下的多个线程可以共享该进程的所有资源。
- 2、如果IO操作密集，则可以多线程运行效率高，缺点是如果一个线程崩溃，都会造成进程的崩溃。

应用：

IO密集的用多线程，在用户输入，sleep时候，可以切换到其他线程执行，减少等待的时间。

CPU密集的用多进程，因为假如IO操作少，用多线程的话，因为线程共享一个全局解释器锁，当前运行的线程会霸占GIL，其他线程没有GIL，就不能充分利用多核CPU的优势。

110、python正则中search和match？

```
54 import re
55 s = "小明年龄18岁 工资10000"
56 res=re.search(r"\d+",s).group()
57 print("search结果",res)           search匹配到第一个匹配到的数据
58
59 res=re.findall(r"\d+",s)
60 print("findall结果",res)          满足正则，都匹配，不用加group
61
62 res=re.match("小明",s).group()
63 print("match结果",res)           匹配以"小明"开头的字符串，并匹配出小明
64
65 res=re.match(r"\de",s)
66 print("试错，不加group为none,匹配不到",res) 不加group是不对的
67
68 res=re.match("工资",s).group()
69 print("match结果",res)           工资不是字符串开头，匹配不到，报错

search结果 18
.findall结果 ['18', '10000']
.match结果 小明
试错，不加group为none,匹配不到 None
Traceback (most recent call last):
  File "C:\Users\ry-wu.junyu\Desktop\面试题.py", line 68, in <module>
    res=re.match("工资",s).group()
AttributeError: 'NoneType' object has no attribute 'group'
[Finished in 0.2s with exit code 1]
[cmd: 'C:\Python36\python.exe', '-u', 'C:\Users\ry-wu.junyu\Desktop\面试题.py']
```