

第五章 字符串与正则表达式

5.1 字符串基础

5.2 字符串方法

5.3 正则表达式

5.4 实验

5.5 小结

习题

字符串常用的表示方式

- 1、字符串中的字符可以是ASCII字符也可以是其他各种符号。
- 2、它常用英文状态下的单引号（' '）、双引号（" "）或者三单引号（''' '''）、三双引号（""" """）进行表示。

转义字符

字符串中还有一种特殊的字符叫做转义字符，转义字符通常用于不能够直接输入的各种特殊字符。Python常用转义字符：

转义字符	说明
\\	反斜线
\'	单引号
\"	双引号
\a	响铃符
\b	退格符
\f	换页符
\n	换行符
\r	回车符
\t	水平制表符
\v	垂直制表符
\0	Null，空字符串
\000	以八进制表示的ASCII码对应符
\xhh	以十六进制表示的ASCII码对应符

字符串的基础操作

- 求字符串的长度
- 字符串的连接
- 字符串的遍历
- 字符串的包含判断
- 字符串的索引和切片等

1、求字符串的长度

字符串的长度是指字符数组的长度，又可以理解为字符串中的字符个数（空格也算字符），可以用**len()函数**查看字符串的长度。如：

```
>>> sample_str1 = 'Jack loves Python'
```

```
>>> print(len(sample_str1))          #查看字符串长度
```

运行结果：17

2、字符串的连接

字符串的连接是指将多个字符串连接在一起组成一个新的字符串。例如：

```
>>> sample_str2 = 'Jack', 'is', 'a', 'Python', 'fan' #字符串用逗号隔开，组成元组
```

```
>>> print('sample_str2:', sample_str2, type(sample_str2))
```

运行结果如下：

```
sample_str2: ('Jack', 'is', 'a', 'Python', 'fan') <class 'tuple'>
```

当字符串之间没有任何连接符时，这些字符串会直接连接在一起，组成新的字符串。

```
>>> sample_str3 = 'Jack"is"a"Python"fan' #字符串间无连接符，默认合并
```

```
>>> print('sample_str3: ', sample_str3)
```

运行结果如下：

```
sample_str3: JackisaPythonfan
```

字符串之间用 '+' 号连接时，也会出现同样的效果，这些字符串将连接在一起，组成一个新的字符串。

```
>>> sample_str4 = 'Jack' + 'is' + 'a' + 'Python' + 'fan'
```

#字符串 '+' 连接，默认合并

```
>>> print('sample_str4: ', sample_str4)
```

运行结果如下：

```
sample_str4: JackisaPythonfan
```

用字符串与正整数进行乘法运算时，相当于创建对应次数的字符串，最后组成一个新的字符串。

```
>>> sample_str5 = 'Jack'*3
```

#重复创建相应的字符串

```
>>> print('sample_str5: ', sample_str5)
```

运行结果如下：

```
sample_str5: JackJackJack
```

3、字符串的遍历

通常使用**for循环**对字符串进行遍历。例如：

```
>>> sample_str6 = 'Python'
```

```
>>> for a in sample_str6:           #遍历字符串
```

```
    print(a)
```

运行结果如下：

P

y

t

h

o

n

其中变量a，每次循环按顺序代指字符串里面的一个字符。

4、字符串的包含判断

字符串是字符的有序集合，因此用**in操作**来判断指定的字符是否存在包含关系。如：

```
>>> sample_str7 = 'Python'
```

```
>>> print('a' in sample_str7)
```

#字符串中不存在包含关系

```
>>> print('Py' in sample_str7)
```

#字符串中存在包含关系

运行结果如下：

False

True

5、索引和切片

字符串是一个有序集合，因此可以通过偏移量实现索引和切片的操作。在字符串中字符**从左到右**的字符索引依次为**0, 1, 2, 3, ..., len()-1**，字符**从右到左**的索引依次为**-1, -2, -3, ..., -len()**。索引其实简单来说是指字符串的排列顺序，可以通过索引来查找该顺序上的字符。例如：

```
>>> sample_str8 = 'Python'
>>> print(sample_str8[0])
>>> print(sample_str8[1])
>>> print(sample_str8[-1])
>>> print(sample_str8[-2])
```

运行结果如下：

```
P
y
n
o
```

#字符串对应的第一个字符
#字符串对应的第二个字符
#字符串对应的最后一个字符
#字符串对应的倒数第二个字符



注意：虽然索引可以获得该顺序上的字符，但是不能够通过该索引去修改对应的字符。

```
>>> sample_str8[0] = 'b'           #试图修改字符串的第一个字符
```

```
Traceback (most recent call last):  #系统正常报错
```

```
File "<pyshell#4>", line 9, in <module>
```

```
sample_str8[0] = 'b'
```

```
TypeError: 'str' object does not support item assignment
```

切片，也叫分片，和元组与列表相似，是指从某一个索引范围中获取连续的多个字符（又称为子字符）。常用格式如下：

stringname[start:end]

stringname是指被切片的字符串，start和end分别指开始和结束时字符的索引，其中切片的最后一个字符的索引是**end-1**，这里有一个诀窍叫：**包左不包右**。例如：

```
>>> sample_str9 = 'abcdefghijkl'
```

```
>>> print(sample_str9[0:4])  #获取索引为0-4之间的字符串，从索引0开始到3为止，  
不包括索引为4的字符
```

运行结果：abcd

若不指定起始切片的索引位置，**默认是从0开始**；若不指定结束切片的顺序，默认是**字符串的长度-1**。例如：

```
>>> sample_str10 = 'abcdefg'
>>> print("起始不指定", sample_str10[:3])
#获取索引为0-3之间的字符串，不包括3
>>> print("结束不指定", sample_str10[3:])
# 从索引3到最后一个字符，不包括len
```

运行结果如下：

起始不指定 abc

结束不指定 defg

默认切片的字符串是连续的，但是也可以通过指定步进数（step）来跳过中间的字符，其中默认的step是1。例如指定步进数为2：

```
>>> sample_str11 = '012345678'
>>> print('跳2个字符', sample_str11[1:7:2]) #索引1~7，每2个字符截取
```

运行结果：跳2个字符 135

字符串格式化方法

想要进行字符串格式化可以使用**format()方法**。例如：

```
>>> 'My name is {0}, and I am {1}'.format('Jack', 9) #函数格式化
```

```
My name is Jack, and I am 9
```

```
>>> 'a={0:d}, b={1:.3f}'.format(5, 3.1415926) #指定类型与精度
```

```
a=5, b=3.142
```

```
>>> 'a={0:d}, b={1:15.3f}'.format(5, 3.1415926) #指定宽度
```

```
a=5, b=          3.142
```

```
>>> coord = (3, 5)
```

```
>>> 'X: {0[0]}; Y: {0[1]}'.format(coord) #指定变量元素
```

```
'X: 3; Y: 5'
```

```
>>> '{0}{1}{0}'.format('abra', 'cad') # arguments' indices can be repeated
```

```
'abracadabra'
```

```
>>> '{2}, {1}, {0}'.format(*'abc') # unpacking argument sequence
```

```
'c, b, a'
```

字符串格式化方法

通过名字访问变量

```
>>> 'Coordinates: {latitude}, {longitude}'.format(latitude='37.24N', longitude='-115.81W')
'Coordinates: 37.24N, -115.81W'

>>> coord = {'latitude': '37.24N', 'longitude': '-115.81W'}

>>> 'Coordinates: {latitude}, {longitude}'.format(**coord) # 访问字典
'Coordinates: 37.24N, -115.81W'
```

文本对齐与指定宽度及填充字符

```
>>> '{:<30}'.format('left aligned')
'left aligned'

>>> '{:>30}'.format('right aligned')
'right aligned'

>>> '{:^30}'.format('centered')
'centered'

>>> '{:*^30}'.format('centered') # use '*' as a fill char
```

```
!*****centered*****!
```

```
>>> a=25
>>> b='Tom'
>>> f'{b}---{a}' 结果是什么?
```

字符串格式化方法

格式化百分数

```
>>> points = 19
>>> total = 22
>>> 'Correct answers: {:.2%}'.format(points/total)
'Correct answers: 86.36%'
```

格式化时间

```
>>> import datetime
>>> d = datetime.datetime(2010, 7, 4, 12, 15, 58)
>>> '{:%Y-%m-%d}'.format(d)
'2010-07-04'
>>> '{:%H:%M:%S}'.format(d)
'12:15:58'
>>> '{:%Y-%m-%d %H:%M:%S}'.format(d)
'2010-07-04 12:15:58'
```

字符串格式化方法

嵌套变量与更复杂的例子

```
>>> for align, text in zip('<^>', ['left', 'center', 'right']):
...     '{0:{fill}{align}16}'.format(text, fill=align, align=align)
...
```

```
'left<<<<<<<<<<<<<<<'
'~~~~~center~~~~~'
'>>>>>>>>>>>>>>>right'
```

```
>>> octets = [192, 168, 0, 1]
>>> '{:02X}{:02X}{:02X}{:02X}'.format(*octets)
'C0A80001'
```

```
>>> width = 5
>>> for num in range(5,12):
...     for base in 'dXob':
...         print('{0:{width}{base}}'.format(num, base=base, width=width), end=' ')
...     print()
...
```

5	5	5	101
6	6	6	110
7	7	7	111
8	8	10	1000
9	9	11	1001
10	A	12	1010
11	B	13	1011

字符串的格式化通常有两种方式，除了之前提到的用函数的形式进行格式化以外，还可以用字符串格式化表达式来进行格式，常用%进行表示，其中%前面是需要格式化的字符串，而%后面就是需要填充的实际参数，这个实际参数其本质就是元组。%也可以理解为占位符。

```
>>> print('My name is %s, and I am %d'%(Jack, 9)) #表达式格式化
```

```
My name is Jack, and I am 9
```

注意：如果想要将后面填充的浮点数保留两位小数，可以用%.2f表示，同时会对第三位小数进行四舍五入。例如：

```
>>> print('你花了%.2f元钱'%(20.45978)) #浮点数保留两个小数
```

```
你花了20.46元钱
```


字符串常见的格式化符号

格式控制符	说明
%s	字符串（采用str()的显示）或其他任何对象
%r	与%s相似（采用repr()的显示）
%c	单个字符及其ASCII码
%b	参数转换成二进制整数
%d	参数转换成十进制整数
%i	参数转换成十进制整数
%o	参数转换成八进制整数
%u	参数转换成十进制整数
%x	参数转换成十六进制整数，字母小写
%X	参数转换成十六进制整数，字母大写
%e. E	按科学计数法格式转换成浮点数
%f. F	按定点小数格式转换成浮点数
%g. G	按定点小数格式转换成浮点数，与%f. F不同
%p	用十六进制数格式化变量的地址

```
>>> print('%i'%5.6)
5
>>> print('%d'%5.6)
5
>>> print('%e'%5.6)
5.600000e+00
>>> print('%f'%5.6)
5.600000
>>> print('%g'%5.6)
5.6
```

格式化操作符辅助指令

符号	功能
+	在正数前面显示加号(+)
#	在八进制数前面显示零('0'), 在十六进制前面显示'0x'或者'0X'(取决于用的是'x'还是'X')
0	显示的数字前面填充'0'而不是默认的空格
%	'%%'输出一个单一的'%'
m.n	m 是显示的最小总宽度,n 是小数点后的位数(如果可用的话)

```
>>> print('%.2f,%+15.2f'%(3.1415926,3.1415926))
3.14,      +3.14
>>> print('%#x'%20)
0x14
```

第五章 字符串与正则表达式

5.1 字符串基础

5.2 字符串方法

5.3 正则表达式

5.4 实验

5.5 小结

习题

字符串是str类型对象，所以Python内置了一系列操作字符串的方法。其中常用的方法如下：

1. `str.strip([chars])`、`str.lstrip([chars])`、`str.rstrip([chars])`

若方法里面的chars不指定，默认去掉字符串的首、尾空格或者换行符，但是如果指定了chars，那么会删除首尾的chars。例如：

```
>>> sample_fun1 = ' Hello world^#'
```

```
>>> print(sample_fun1.strip())
```

默认去掉首尾空格

```
>>> print(sample_fun1.strip('#'))
```

指定首尾需要删除的字符

```
>>> print(sample_fun1.strip('^#'))
```

运行结果如下：

```
Hello world^#
```

```
Hello world^
```

```
Hello world
```

2. str.count('chars', start, end)

统计chars字符串或者字符在str中出现的次数，从start顺序开始查找一直到end顺序范围结束，默认是从顺序0开始。例如：

```
>>> sample_fun2 = 'abredcdblueabbluebcd'
```

```
>>> print(sample_fun2.count('ab',2,9))      #统计字符串出现的次数
```

运行结果如下：

2

3. str.capitalize()

将字符串的首字母大写。

```
>>> sample_fun3 = 'We should thank UPC.'
```

```
>>> print(sample_fun3.capitalize())          # 首字母大写
```

运行结果如下：

```
We should thank upc.
```

4. str.replace(oldstr, newstr, count)

用旧的子字符串替换新的子字符串，若不指定替换次数count，则默认全部替换。

```
>>> sample_fun4 = 'ab12cd3412cd'
```

```
>>> print(sample_fun4.replace('12','21'))      #不指定替换次数count
```

```
>>> print(sample_fun4.replace('12','21',1))    #指定替换次数count
```

运行结果如下：

```
ab21cd3421cd
```

```
ab21cd3412cd
```

5. `str.find('str',start,end)`、`str.rfind('str',start,end)`---从右边开始查找

查找并返回子字符串在start到end范围内的顺序，默认范围是从父字符串的头开始到尾结束。

```
>>> sample_fun5 = '0123156'
```

```
>>> print(sample_fun5.find('5'))    #查看子字符串的顺序
```

```
>>> print(sample_fun5.find('5',1,4)) #指定范围内没有该字符串默认返回-1
```

```
>>> print(sample_fun5.find('1'))    #多个字符串返回第一次出现时候的顺序
```

运行结果如下：

```
5
```

```
-1
```

```
1
```

6. `str.index('str',start,end)`、`str.rindex('str',start,end)`

该函数与`find`函数一样，但是如果在某一个范围内没有找到该字符串的时候，不再返回-1而是直接报错。

```
>>> sample_fun6 = '0123156'
```

```
>>> print(sample_fun6.index(7))  #指定范围内没有找到该字符串会  
报错
```

运行结果如下：

Traceback (most recent call last):

File "D:/python/space/demo05-02-03.py", line 2, in <module>

```
    print(sample_fun6.index(7))  #指定范围内没有找到该字符串会报  
错
```

TypeError: must be str, not int

7. str.isalnum()

字符串是**由字母或数字**组成则返回true，否则返回false。

```
>>> sample_fun7 = 'abc123'      #字符串由字母和数字组成
>>> sample_fun8 = 'abc'         #字符串由字母组成
>>> sample_fun9 = '123'         #字符串由数字组成
>>> sample_fun10 = 'abc12%'     #字符串由除了数字字母以为的字符组成
print(sample_fun7.isalnum())
print(sample_fun8.isalnum())
print(sample_fun9.isalnum())
print(sample_fun10.isalnum())
```

运行结果如下：

```
True
True
True
False
```

8. str.isalpha()

字符串是否**全是由字母**组成的，是返回true，否则返回false。

```
>>> sample_fun11 = 'abc123'           #字符串中不只有字母
```

```
>>> sample_fun12 = 'abc'             #字符串中只是有字母
```

```
print(sample_fun11.isalpha())
```

```
print(sample_fun12.isalpha())
```

运行结果如下：

False

True

9. str.isdigit()、str.decimal()、str.isnumeric()

字符串是否**全是由数字**组成，是则返回true，否则返回false。

```
>>> sample_fun13 = 'abc12'           #字符串中不只有数字
```

```
>>> sample_fun14 = '12'             #字符串中只是有数字
```

```
print(sample_fun13.isdigit()) # False
```

```
print(sample_fun14.isdigit()) # True
```

str.isdigit()、str.decimal()、str.isnumeric()的区别

```
num = "1"    #unicode数字
num.isdigit()    # True
num.isdecimal() # True
num.isnumeric() # True
```

```
num = " 1 "   # 全角数字
num.isdigit()    # True
num.isdecimal() # True
num.isnumeric() # True
```

```
num = "IV"    # 罗马数字
num.isdigit()    # True
num.isdecimal() # False
num.isnumeric() # True
```

```
num = "四"    # 汉字数字
num.isdigit()    # False
num.isdecimal() # False
num.isnumeric() # True
```

10. str.isspace()

字符串是否**全是由空格**组成的，是则返回true，否则返回false。

```
>>> sample_fun15 = ' abc'                #字符串中不只有空格
```

```
>>> sample_fun16 = ' '                  #字符串中只有空格
```

```
>>> print(sample_fun15.isspace())
```

```
>>> print(sample_fun16.isspace())
```

运行结果如下：

False

True

11. str.islower()

字符串是否**全是小写**，是则返回true，否则返回false。

```
>>> sample_fun17 = 'abc'           #字符串中的字母全是小写
>>> sample_fun18 = 'Abcd'          #字符串中的字母不只有小写
>>> print(sample_fun17.islower())   # True
>>> print(sample_fun18.islower())   # False
```

12. str.isupper()

字符串是否**全是大写**，是则返回true，否则返回false。

```
>>> sample_fun19 = 'abCa'          #字符串中的字母不全是大写字母
>>> sample_fun20 = 'ABCA'          #字符串中的字母全是大写字母
>>> print(sample_fun19.isupper())   # False
>>> print(sample_fun20.isupper())   # True
```

13. str.istitle()

字符串**首字母是否是大写**，是则返回true，否则返回false。

```
>>> sample_fun21 = 'Abc'                #字符串首字母大写
>>> sample_fun22 = 'aAbc'               #字符串首字母不是大写
>>> print(sample_fun21s.istitle())      # True
>>> print(sample_fun22.istitle())      # False
```

14. str.lower()、str.upper()

将字符串中的字母全部**转换成小写或大写字母**。

```
>>> sample_fun23 = 'aAbB'      #将字符串中的字母全部转为小写字母
```

```
>>> print(sample_fun23.lower())
```

```
aabb
```

```
>>> sample_fun23 = 'abcD'      #将字符串中的字母全部转为大写字母
```

```
>>> print(sample_fun24.upper())
```

```
ABCD
```

15. str.swapcase()

将字符串中的字母**大小写交换**。

```
>>> sample_fun24 = 'aAbB'      #将字符串中的字母大小写交换
```

```
>>> print(sample_fun24.swapcase())
```

```
AaBb
```

16. `str.split(sep,maxsplit)`、`str.rsplit(sep,maxsplit)`

将字符串按照指定的sep字符进行**分割**，返回一个字符串列表。maxsplit是指定需要分割的次数，若不指定sep，则默认是分割空白（一个或多个空格、Tab符\t、回车符\r、换行符\n）。

```
>>> sample_fun25 = 'abacdaef'
```

```
>>> print(sample_fun25.split('a'))
```

指定分割字符串

```
>>> print(sample_fun25.split())
```

不指定分割字符串

```
>>> print(sample_fun25.split('a',1))
```

指定分割次数

运行结果如下：

```
['', 'b', 'cd', 'ef']
```

```
['abacdaef']
```

```
['', 'bacdaef']
```

```
>>> sample_fun25 = 'I love python'
```

```
>>> sample_fun25.split() # 默认按空白分割字符串
```

```
['I', 'love', 'python']
```


17. str.startswith(sub[,start[,end]])

判断字符串在指定范围内是否以sub开头，默认范围是整个字符串。

```
>>> sample_fun26 = '12abcdef'
```

```
>>> print(sample_fun26.startswith('12',0,5)) #范围内是否是以该字符开头
```

运行结果如下：

```
True
```

```
commit 758ca05e2eb04532b5d78331ba87c291038e2c61
Author: Garrett-R <xxxx@xxxxx.com>
Date: Sat Dun 27 15:11:12 2015 -0700
```

18. str.endswith(sub[,start[,end]])

判断字符串在指定范围内是否是以sub结尾，默认范围是整个字符串。

```
>>> sample_fun27 = 'abcdef12'
```

```
>>> print(sample_fun27.endswith('12')) #指定范围内是否是以该字符结尾
```

运行结果如下：

```
True
```

19. str.partition(sep)

将字符串从sep第一次出现的位置开始分隔成三部分：sep顺序前、sep、sep顺序后。最后会**返回一个三元组**，如果没有找到sep的时候，返回字符本身和两个空格组成的三元组。

```
>>> sample_fun28 = '123456'
```

```
>>> print(sample_fun28.partition('34')) #指定字符分割，能够找到该字符
```

```
>>> print(sample_fun28.partition('78')) #指定字符分割，不能够找到该字符
```

运行结果如下：

```
('12', '34', '56')
```

```
('123456', '', '')
```

20. str.rpartition(sep)

该函数与partition(sep)函数一致，但是sep不再是第一次出现的顺序，而是最后一次出现的顺序。例如：

```
>>> sample_fun29 = '12345634'
```

```
>>> print(sample_fun29.rpartition('34')) #指定字符最后一次的位置进行分割
```

运行结果如下：

```
('123456', '34', '')
```

21. str.join(iterable)

将可迭代序列iterable（字符串元组、字符串列表、字符串集合、字符串等）中的元素以指定的字符串str作为分隔符，连接生成一个新的字符串。

```
>>> sep = "-"
```

```
>>> seq = ("a", "b", "c"); # 字符串元组
```

```
>>> print(sep.join( seq )) # a-b-c
```

```
c=list("abc")
```

```
print(c)
```

```
# 字符列表转换为字符串,使用join方法
```

```
d=".join(c)
```

```
print(d)
```

```
>>> sep="::"
```

```
>>> sep.join("语数外") #可迭代序列iterable为字符串
```

```
'语::数::外'
```

```
>>> BRICS=["巴西(Brazil)","俄罗斯(Russia)","印度(India)","中国(China)","南非(South Africa)"] #可迭代序列iterable为列表
```

```
>>> sep.join(BRICS)
```

```
'巴西(Brazil)::俄罗斯(Russia)::印度(India)::中国(China)::南非(South Africa)'
```

22. `str.ljust(width, fillchar=' ')`、`str.rjust(width, fillchar=' ')`

返回一个原字符串**左/右对齐**, 并使用指定字符填充至指定长度的新字符串。如果指定的长度小于原字符串的长度则返回原字符串。参数width -- 指定字符串长度, fillchar -- 填充字符, 默认为空格。

```
>>> line = "this is string example....wow!!!"
```

```
>>> print(line.ljust(50, '0'))
```

```
this is string example....wow!!!0000000000000000000000
```

```
>>> print(line.rjust(50, '0'))
```

```
000000000000000000000000this is string example....wow!!!
```

23. `str.center(width, fillchar=' ')`

返回一个原字符串**居中**, 并使用指定字符填充至指定长度的新字符串。

```
>>> print(line.center(50, '0'))
```

```
0000000000this is string example....wow!!!0000000000
```

第五章 字符串与正则表达式

5.1 字符串基础

5.2 字符串方法

5.3 正则表达式

5.4 实验

5.5 小结

习题

5.3.1 认识正则表达式

正则表达式 (Regular Expression)，此处的“Regular”即是“规则”、“规律”的意思，Regular Expression即“描述某种规则的表达式”，因此它又可称为正规表示式、正规表示法、正规表达式、规则表达式、常规表示法等，在代码中常常被简写为regex、regexp或RE。

正则表达式使用某些单个字符串，来描述或匹配某个句法规则的字符串。在很多文本编辑器里，正则表达式通常被用来检索或替换那些符合某个模式的文本，如下面的表5.3、5.4、5.5、5.6所示。

5.3.1 认识正则表达式

表5.3 单个字符匹配

规则	说明
.	匹配任意1个字符（除了\n）
[...]	用来表示一组字符,单独列出。例如, [amk] 匹配 'a', 'm'或'k'; [Pp]ython 匹配 "Python" 或 "python"; rub[ye] 匹配 "ruby" 或 "rube"
[^ ...]	不在[]中的字符。例如, [^abc] 匹配除了a,b,c之外的字符, [^0-9] 匹配除了数字外的字符
\d	匹配数字, 即0-9, 等价于 [0-9]
\D	匹配非数字, 即不是数字, 等价于 [^0-9]
\s	匹配任何空白字符, 包括空格、制表符、换页符等等。等价于 [\f\n\r\t\v]
\S	匹配非空白, 等价于 [^ \f\n\r\t\v]
\w	匹配包括下划线的任何单词字符。等价于'[A-Za-z0-9_]'
\W	匹配任何非单词字符。等价于 '[^A-Za-z0-9_]'
\n, \t, 等.	匹配一个换行符、匹配一个制表符等

5.3.1 认识正则表达式

实例

实例	描述
python	匹配 "python".

实例	描述
[Pp]ython	匹配 "Python" 或 "python"
rub[ye]	匹配 "ruby" 或 "rube"
[aeiou]	匹配中括号内的任意一个字母
[0-9]	匹配任何数字。类似于 [0123456789]
[a-z]	匹配任何小写字母
[A-Z]	匹配任何大写字母
[a-zA-Z0-9]	匹配任何字母及数字
[^aeiou]	除了aeiou字母以外的所有字符
[^0-9]	匹配除了数字外的字符

5.3.1 认识正则表达式

表5.4 表示数量（出现次数）的匹配

规则	说明
<code>re*</code>	匹配前一个表达式出现0次或者无限次，即可有可无
<code>re+</code>	匹配前一个表达式出现1次或者无限次，即至少有1次
<code>re?</code>	匹配前一个表达式出现1次或者0次，即要么有1次，要么没有
<code>re{m}</code>	匹配前一个表达式出现m次。例如， <code>o{2}</code> 不能匹配 "Bob" 中的 "o"，但是能匹配 "food" 中的两个 o。
<code>re{m,}</code>	匹配前一个表达式至少出现m次。例如， <code>o{2,}</code> 不能匹配"Bob"中的"o"，但能匹配 "fooooood"中的所有 o。" <code>o{1,}</code> " 等价于 " <code>o+</code> "。" <code>o{0,}</code> " 则等价于 " <code>o*</code> "。
<code>re{m,n}</code>	匹配前一个表达式出现从m到n次， m省略则默认为0

5.3.1 认识正则表达式

表5.5 表示边界的匹配

规则	说明
<code>^</code> 或 <code>\A</code>	匹配字符串开头。例如, ' <code>^th.*</code> ' 表示被匹配的整个字符串必须以th开头, 可以匹配 'the other', 但不能比配 'in the hospital'。
<code>\$</code> 或 <code>\Z</code>	匹配字符串结尾。例如, ' <code>.*ou\$</code> ' 表示被匹配的整个字符串必须以ou结尾, 可以匹配 'thank you', 但不能比配 'thank you very much'。
<code>\b</code>	匹配一个单词的边界。例如, ' <code>.*er\b</code> ' 能匹配 "is better than" 中的 'is better', 但不匹配 "is betters than" 中的 'is better'。
<code>\B</code>	匹配非单词边界。 ' <code>.*er\B</code> ' 能匹配 is betters than" 中的 'is better', 但不能匹配 "is better than" 中的 'is better'。

例如：

```
>>> re.match(r'th.*','thank you very much') # 匹配th开头的字符串
<re.Match object; span=(0, 19), match='thank you very much'>
>>> re.match(r'^th.*','hi,thank you very much') # 匹配th开头的字符串
None
>>> re.match(r'th.*','hi, thank you very much') # 匹配th开头的字符串
None
>>> re.search(r'th.*','hi, thank you very much') # 查找th开头的子串
<re.Match object; span=(4, 23), match='thank you very much'>
>>> re.search(r'th.*','thank you very much') # 查找th开头的子串
<re.Match object; span=(0, 19), match='thank you very much'>
>>> re.search(r'^th.*ou','thank you very much') # 查找th开头的字符串, ou结
尾的子串
<re.Match object; span=(0, 9), match='thank you'>
>>> re.search(r'^th.*ou','hi, thank you very much') # 查找th开头的字符串,
ou结尾的子串
None
```

例如：

```
>>> re.match(r'.*ou','thank you') # 匹配以ou结尾的子串  
<re.Match object; span=(0, 9), match='thank you'>
```

```
>>> re.match(r'.*ou','thank you very much') # 匹配以ou结尾的  
子串  
<re.Match object; span=(0, 9), match='thank you'>
```

```
>>> re.match(r'.*ou\Z','thank you') # 匹配以ou结尾的字符串  
<re.Match object; span=(0, 9), match='thank you'>
```

```
>>> re.match(r'.*ou\Z','thank you very much') # 字符串不是  
以ou结尾，不匹配！
```

```
None
```

5.3.1 认识正则表达式

表5.6 匹配分组

规则	说明
<code>a b</code>	匹配左右任意一个表达式a或b
<code>(re)</code>	对正则表达式分组并记住匹配的文本
<code>\num</code>	引用分组num匹配到的字符串
<code>(?P<name>)</code>	分组起别名，以便从匹配结果导出分组字典
<code>(?P=name)</code>	引用别名为name分组匹配到的字符串

```
>>> re.match('(PY){0,3}N','PYN')
<re.Match object; span=(0, 3), match='PYN'>
>>> re.match('(PY){0,3}N','PAYN')
>>> re.match('(PY){0,3}N','PYPYN')
<re.Match object; span=(0, 5), match='PYPYN'>
>>> re.match('(PY){0,3}N','N')
<re.Match object; span=(0, 1), match='N'>
```

5.3.1 认识正则表达式

匹配两种子串

```
>>> re.match(r'/player/default/(index|view)', '/player/default/index/1')  
<re.Match object; span=(0, 21), match='/player/default/index'>
```

```
>>> re.match(r'/player/default/(index|view)', '/player/default/view/1')  
<re.Match object; span=(0, 20), match='/player/default/view'>
```

```
>>> re.match(r'/player/default/(index|view)', '/player/default/search')  
None
```

5.3.2 re模块

在Python中需要通过正则表达式对字符串进行匹配的时候，可以导入一个库（模块），名字为re，它提供了对正则表达式操作所需的方法，如表5.7。

表5.7 re模块常见的方法

方法	说明
<code>match(pattern, string, flags=0)</code>	从字符串的开始匹配一个匹配对象(即正则表达式)。flag为标志位，用于控制正则表达式的匹配方式，如是否区分大小写，多行匹配等。
<code>search(pattern, string, flags=0)</code>	在字符串中查找匹配的对象，找到第一个后就返回；如果没有找到就返回None
<code>sub(pattern, repl, string, count=0, flags=0)</code>	替换掉字符串中的匹配项
<code>split(pattern, string, maxsplit=0, flags=0)</code>	分割字符串
<code>findall(pattern, string, flags=0)</code>	获取字符串中所有匹配的对象
<code>compile(pattern, flags=0)</code>	创建模式(Pattern)对象，亦即正则表达式对象

5.3.3 re.match()方法

re.match()是用来进行正则匹配检查的方法，若字符串匹配正则表达式，则match()方法返回**匹配 (Match) 对象**，否则返回None（注意不是空字符串""）。

函数语法：

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

参数	描述
pattern	匹配的正则表达式
string	要匹配的字符串。
flags	标志位，用于控制正则表达式的匹配方式，如：是否区分大小写，多行匹配等。

正则表达式修饰符 - 可选标志

正则表达式可以包含一些可选标志修饰符来控制匹配的模式。修饰符被指定为一个可选的标志。多个标志可以通过按位 OR(|) 它们来指定。如 `re.I | re.M` 被设置成 I 和 M 标志：

修饰符	描述
<code>re.I</code>	使匹配对大小写不敏感
<code>re.L</code>	做本地化识别 (locale-aware) 匹配，即特殊字符集 <code>\w</code> , <code>\W</code> , <code>\b</code> , <code>\B</code> , <code>\s</code> , <code>\S</code> 依赖于当前环境
<code>re.M</code>	多行匹配，影响 <code>^</code> 和 <code>\$</code>
<code>re.S</code>	使 <code>.</code> 匹配包括换行在内的所有字符，否则遇到换行符就重新开始匹配。
<code>re.U</code>	根据Unicode字符集解析字符。这个标志影响 <code>\w</code> , <code>\W</code> , <code>\b</code> , <code>\B</code> 。
<code>re.X</code>	该标志通过给予你更灵活的格式以便你将正则表达式写得更易于理解，这样我们可以在正则表达式中添加一些空白和#注释，编译时却被忽略。

几个简单示例

```
import re # 导入正则包
```

```
# 找出所有1970-1979出生的ID
```

```
ids = ['3504241978xxxxxxxx', '3705021992xxxxxxxx', '3705021975xxxxxxxx']
```

```
id197x = [aID for aID in ids if re.match(r'\d{6}197.{9}', aID)]
```

```
print(id197x)
```

```
# 找出所有以pr打头的单词
```

```
words = ['no', 'print', 'Program', 'plug', 'prepare', 'at', 'present', 'practice']
```

```
word_pr = [word for word in words if re.match(r'pr', word)] # 匹配区分大小写
```

```
print(word_pr)
```

```
word_pr = [word for word in words if re.match(r'pr', word, re.I)] # 不区分大小写
```

```
print(word_pr)
```

```
# 找出所有以pr打头且以t结尾的单词
```

```
word_pr_t = [word for word in words if re.match(r'pr.*t\b', word)]
```

```
print(word_pr_t)
```

```
# 找出所有以pr或a打头的单词
```

```
word_pr_a = [word for word in words if re.match(r'(pr|a)', word)]
```

```
print(word_pr_a)
```

```
['3504241978xxxxxxxx', '3705021975xxxxxxxx']  
['print', 'prepare', 'present', 'preview']  
['print', 'Program', 'prepare', 'present', 'preview']  
['print', 'present']  
['print', 'prepare', 'at', 'present', 'practice']
```

re.MatchObject 对象的方法

方法	描述
group(num=0)	num默认为0，返回被 RE 匹配的整个表达式的字符串。group() 可以一次输入多个组号num，在这种情况下它将返回一个包含那些组所对应值的元组。组号从1开始。
start()	返回匹配开始的位置
end()	返回匹配结束的位置
span()	返回一个元组，包含匹配 (开始,结束) 的位置
groupdict(default=None)	将命名分组以字典形式返回，default表示不参与匹配的分组

```
>>> import re
>>> mo1=re.match('www', 'www.runoob.com') # 在起始位置匹配
>>> print(mo1.group()) # www
>>> print(mo1.span()) # (0, 3)
>>> mo2=re.match('com', 'www.runoob.com') # 不在起始位置匹配
>>> print(mo2) # None
```

re.MatchObject 对象的方法

例如:

```
import re # 导入正则包
```

注意空格

```
line = "Cats are smarter than dogs"
matchObj = re.match(r'(.*)are(.*?)', line, re.M | re.I)
if matchObj:
    print("matchObj.group() : ", matchObj.group())
    print("matchObj.group(1) : ", matchObj.group(1))
    print("matchObj.group(2) : ", matchObj.group(2))
else:
    print("No match!!")
```

运行结果:

```
matchObj.group() : Cats are smarter than dogs
matchObj.group(1) : Cats
matchObj.group(2) : smarter
```

```
line = "Cats are smarter than dogs"
matchObj = re.match(r'(.*)are(.*?)', line, re.M | re.I)
```

正则表达式: `r'(.*)are(.*?)'`

解析:

首先, 这是一个字符串, 前面的一个 `r` 表示字符串为非转义的原始字符串, 让编译器忽略反斜杠, 也就是忽略转义字符。但是这个字符串里没有反斜杠, 所以这个 `r` 可有可无。

- ❑ `(.*)` 第一个匹配分组, `.` 代表匹配除换行符之外的所有字符。
 - ❑ `(.*?)` 第二个匹配分组, `.` 后面多个问号?, 代表**非贪婪**模式, 即只匹配符合条件的最少字符
 - ❑ 后面的一个 `.` 没有括号包围, 所以不是分组, 匹配效果和第一个一样, 但是不计入匹配结果中。
-
- `matchObj.group()` 等同于 `matchObj.group(0)`, 表示匹配到的完整文本字符
 - `matchObj.group(1)` 得到第一组匹配结果, 也就是 `(.*)` 匹配到的
 - `matchObj.group(2)` 得到第二组匹配结果, 也就是 `(.*?)` 匹配到的
 - 因为只有匹配结果中只有两组, 所以如果填 3 则会报错。

贪婪（最长）匹配模式与非贪婪（最短）匹配模式

- **贪婪（最长）匹配模式 `.*`**：找到第一个匹配的位置，然后向后继续匹配其他的表达式符号，直到字符串末尾。
- **非贪婪（最短）匹配模式 `.*?`**：找到第一个匹配的位置后便结束匹配。

<re.Match object; span=(0, 10), match='PYANBNCNDN'>

比较如下两行代码：

```
match = re.match(r'PY.*N', 'PYANBNCNDN')
```

```
match = re.match(r'PY.*?N', 'PYANBNCNDN')
```

<re.Match object; span=(0, 4), match='PYAN'>

又如：

```
>>> import re                                #导入re包
>>> sample_result1 = re.match('Python','Python12') #从头查找匹配字符串
>>> print(sample_result1.group())                #输出匹配的字符串
```

运行结果如下：

Python

又如：匹配以PY开头，后面包括1个或多个N的字符串。

```
>>> re.match('PYN{1,}','PYNN')
<re.Match object; span=(0, 4), match='PYNN'>
>>> re.match('PYN+','PYNN')
<re.Match object; span=(0, 4), match='PYNN'>
>>> re.match('PYN+','PYN')
<re.Match object; span=(0, 3), match='PYN'>
>>> re.match('PYN+','PY') # 匹配不成功
>>>
```


5.3.4 re.search()方法

re.search()方法和re.match()方法相似，也是用来对正则匹配检查的方法但不同的是search()方法是在字符串的头开始一直到尾进行查找，若正则表达式与字符串匹配成功，那么就返回匹配对象，否则返回None。例如：

```
>>> import re
>>> sample_result2 = re.search('Python','354Python12') #依次匹配字符串
>>> print(sample_result2.group())
```

运行结果如下：

Python

re.search()方法与re.match()方法的区别

虽然re.match()和re.search()方法都是指定的正则表达式与字符串进行匹配，但是 re.match()是从字符串的开始位置进行匹配，若匹配成功，则返回匹配对象，否则返回None。而re.search()方法却是从字符串的全局进行扫描，若匹配成功就返回匹配对象，否则返回None。例如：

```
>>> import re
>>> sample_result3 = re.match('abc','abcdef1234') #match只能够匹配头
>>> sample_result4 = re.match('1234','abcdef1234')
>>> print(sample_result3.group())
>>> print(sample_result4)
>>> sample_result5 = re.search('abc','abcdef1234') #search匹配全体字符
>>> sample_result6 = re.search('1234','abcdef1234')
>>> print(sample_result5.group())
>>> print(sample_result6.group())
```

运行结果如下：

abc

None

abc

1234

'(?P<name>...)' 命名分组匹配实例

从身份证 1102231990xxxxxxxxx提取各部分信息

```
s = '1102231990xxxxxxxxx'
res = re.search(r'(?P<province>\d{3})(?P<city>\d{3})(?P<born_year>\d{4})', s)
print(res.groupdict())
print(res.group(1), res.group(2), res.group(3))
```

运行结果:

```
{'province': '110', 'city': '223', 'born_year': '1990'}
110 223 1990
```

提取名字的各组成部分

```
m = re.match(r'(?P<first_name>\w+) (?P<last_name>\w+)", "Malcolm Reynolds")
print(m.groupdict())
```

运行结果:

```
{'first_name': 'Malcolm', 'last_name': 'Reynolds'}
```

5.3.5 re.split()方法

split()方法按照能够匹配的子串将字符串分割后返回列表，它的使用形式如下：

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

参数	描述
pattern	匹配的正则表达式
string	要匹配的字符串。
maxsplit	分隔次数，maxsplit=1 分隔一次，默认为 0，不限制次数。
flags	标志位，用于控制正则表达式的匹配方式，如是否区分大小写，多行匹配等。

```
>>> re.split('\W+', 'runoob, runoob, runoob.')
['runoob', 'runoob', 'runoob', '']
>>> re.split('\W+', ' runoob, runoob, runoob.', 1)
 ['', 'runoob, runoob, runoob.']
>>> re.split('a*', 'hello world') # 对于找不到匹配的字符串，
split 不对其分割
['hello world']
```

5.3.6 re.sub()方法

sub()方法用于替换字符串中的匹配项，它的使用形式如下：

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

参数	描述
pattern	匹配的正则表达式
repl	替换的字符串（空串则仅删除而不替换），也可为一个函数。
string	要匹配的字符串。
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。
flags	标志位，用于控制正则表达式的匹配方式，如是否区分大小写，多行匹配等。

提取电话号码

```
phone = "2004-959-559 # 这是一个国外电话号码"
num = re.sub(r'#. *', "", phone) # 删除字符串中的 Python注释
print("电话号码是: ", num)      # 电话号码是: 2004-959-559 (末尾有1个空格)
num = re.sub(r'\D', "", phone) # 删除非数字字符（这里为短线和空格）
print("电话号码是: ", num)      # 电话号码是: 2004959559
```

5.3.6 re.compile()方法

compile()方法用于编译一个正则表达式，生成一个模式(Pattern)对象，然后可以多次调用 match() 和 search() 函数来匹配或查找字符串。

re.compile(pattern, flags=0)

```
>>> pattern = re.compile(r'\d+') # 用于匹配至少一个数字
>>> m = pattern.match('one12three34four') # 查找头部
>>> print(m) #None -----没有匹配
>>> m = pattern.match('one12three34four', 2, 10) #
从'e'的位置开始匹配>>> print(m) # None -----没有匹配
>>> m = pattern.match('one12three34four', 3, 10) #
从'1'的位置开始匹配
>>> print(m) # 正好匹配 , 返回一个 Match 对象
<_sre.SRE_Match object at 0x10a42aac0>
>>> m.group() # '12'
>>> m.start() # 3
>>> m.end() # 5
>>> m.span() # (3, 5)
```

re.compile()方法另一个例子

```
>>> pattern = re.compile(r'([a-z]+) ([a-z]+)', re.I)
>>> m = pattern.match('Hello World Wide Web')
>>> print(m) # 匹配成功, 返回一个 Match 对象
<_sre.SRE_Match object at 0x10bea83e8>
>>> m.group(0) # 返回匹配成功的整个子串
'Hello World'
>>> m.span(0) # 返回匹配成功的整个子串的索引
(0, 11)
>>> m.group(1) # 返回第一个分组匹配成功的子串
'Hello'
>>> m.span(1) # 返回第一个分组匹配成功的子串的索引
(0, 5)
>>> m.group(2) # 返回第二个分组匹配成功的子串
'World'
>>> m.span(2) # 返回第二个分组匹配成功的子串
(6, 11)
>>> m.groups() # 等价于 (m.group(1), m.group(2), ...)
('Hello', 'World')
>>> m.group(3) # 不存在第三个分组
Traceback (most recent call last): File "<stdin>", line 1, in <module>
IndexError: no such group
```

5.3.7 re.findall()方法

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

注意：match 和 search 是匹配一次 findall 匹配所有。

它的使用形式如下：**re.findall(pattern, string, flags=0)**

参数	描述
pattern	匹配的正则表达式
string	要匹配的字符串。
flags	标志位，用于控制正则表达式的匹配方式，如是否区分大小写，多行匹配等

查找字符串中的所有数字

```
>>>result = re.findall(r'\d+','runoob123google 456')
>>>print(result)
['123', '456']
```


5.3.7 re.findall()方法

如何从字符串中查找以字母b开头、er结尾的所有单词？

```
>>> msg = 'is better bill player than bigger how berry'

>>> re.findall(r'b.*er', msg)
['better bill player than bigger how ber']

>>> re.findall(r'b.*er\b', msg)
['better bet ter than bigger']

>>> re.findall(r'b.*?er', msg)
['better', 'bill player', 'bigger', 'ber']

>>> re.findall(r'b.*?er\b', msg)
['better', 'bill player', 'bigger']

>>> re.findall(r'b\w*?er', msg)
['better', 'bigger', 'ber']

>>> re.findall(r'b\w*?er\b', msg)
['better', 'bigger']
```

Pattern对象的findall()方法

作用于re.findall()方法相同，但可以指定查找开始和结束位置。
它的使用形式如下：

pattern.findall(string, pos=0, endpos=len(string))

参数	描述
string	要匹配的字符串。
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。

查找字符串中的数字

```
>>>pattern = re.compile(r'\d+') # 查找数字
>>>result1 = pattern.findall('runoob 123 google 456')
>>>result2 = pattern.findall('run88oob123google456', 0, 10) # 指定查找范围
>>>print(result1)
['123', '456']
>>>print(result2)
['88', '12']
```

第五章 字符串与正则表达式

5.1 字符串基础

5.2 字符串方法

5.3 正则表达式

5.4 实验

5.5 小结

习题

习题

5.4.1 使用字符串处理函数

1. 我们常看到自己电脑上的文件路径如' C:\Windows\Logs\dosvc', 请将
该路径分割为不同的文件夹。

```
>>> sample_str1 = 'C:\Windows\Logs\dosvc'
>>> sample_slipstr = sample_str1.split('\\') #\转义字符要转一次才是本意
>>> print(sample_slipstr)
```

运行结果如下：

```
['C:', 'Windows', 'Logs', 'dosvc']
```

2. Python的官网是<https://www.python.org>判断该网址是否是以org结尾。

```
>>> sample_str2 = 'https://www.python.org'
>>> print(sample_str2.endswith('org'))      #从字符串末尾开始查找
```

运行结果如下：

```
True
```

5.4.2 正则表达式的使用

写出一个正则表达式来匹配是否是手机号。

```
phone_rule = re.compile(r'1\d{10}$') # 定义一个正则表达式
phone_num = input('请输入一个手机号') # 通过规则去匹配字符串
result = phone_rule.search(phone_num)
if result is not None:
    print('这是一个手机号')
else:
    print('这不是一个手机号')
```

两次运行结果如下：

请输入一个手机号12312345678

这是一个手机号

请输入一个手机号24781131451

这不是一个手机号

5.4.3 使用re模块

用两种方式写出一个正则表达式匹配字符'Python123'中的'Python'并输出字符串'Python'。

```
>>> import re                                #导入re包
>>> pattern = re.compile('Python') #定义正则表达式规则
>>> result1 = pattern.match('Python123') #用match方式匹配字符串
>>> print(result1.group())
>>> result2 = pattern.search('Python123') #用search方式匹配字符串
>>> print(result2.group())
```

5.4.3使用re模块

读取文本文件中的中英文数据并使用正则表达式筛选所需数据。要求：每一行中找到第一个最短的文本串，以'l'开始,'e'结束。

```
import re # 导入正则包
new_lines = [] # 声明new_lines列表

def text_filter(text_lines):# 定义函数：对读取到的数据进行筛选，并将结果存入列表new_lines
    regex_str = r".*?(l.*?e).*" # 正则表达式，非贪婪方式，只找最短的匹配串，
    # regex_str = r".*?(l.*e).*" # 贪婪方式，找到最长的匹配串
    for x in text_lines:
        new_x = x.splitlines() # 注意：splitlines是将传入的字符串去除'\n',之后以列表的形式返回
        match_obj = re.match(regex_str, new_x[0]) # 匹配
        if match_obj: # <=> if match_obj is not None
            new_lines.append(match_obj.group(1))
        else:
            new_lines.append('no')
    return new_lines

file1 = open(r'C:\softwares\readme.txt', 'r') # 打开文件,只读
lines = file1.readlines() # 将文本中的内容以列表的形式（每行为一个元素）赋给lines
file1.close() # 关闭文件
final = text_filter(lines) # 使用正则表达式过滤字符串
print(final)
```

第五章 字符串与正则表达式

5.1 字符串基础

5.2 字符串方法

5.3 正则表达式

5.4 实验

5.5 小结

习题

本章首先讲解了Python字符串概念，字符串的基本操作；其次是字符串的格式化，主要的格式化符号、格式化元组；还有操作字符串的基本方法，这些符号和方法在Python的开发中会被经常使用到。之后，我们学习了正则表达式，re模块和正则表达式的基本表示符号，这些符号可以帮助简化正则表达式。

正则表达式的用途非常广泛，**几乎任何编程语言**都可以使用到它，所以学好正则表达式，对于提高自己的编程能力有非常重要的作用。

第五章 字符串与正则表达式

5.1 字符串基础

5.2 字符串方法

5.3 正则表达式

5.4 实验

5.5 小结

习题

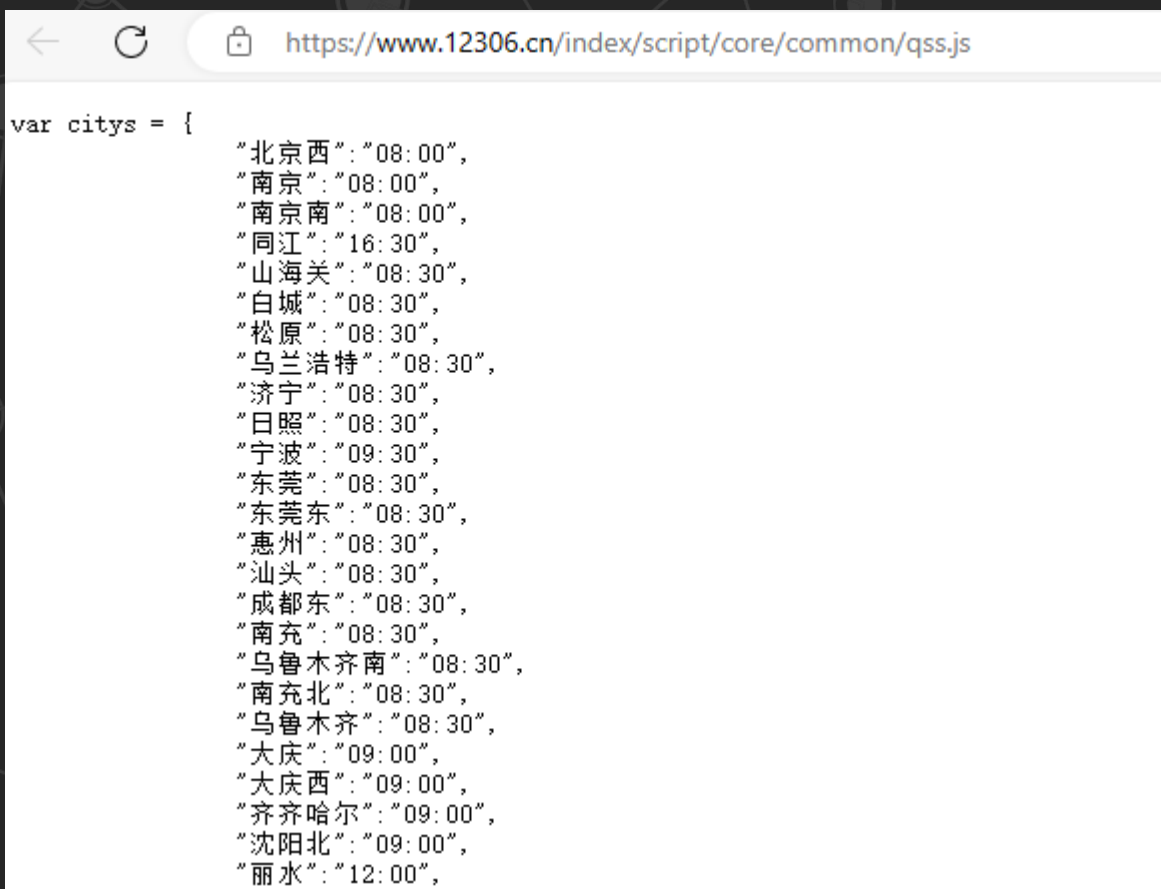
习题：

1. 将字符串'abcdefg' 倒序输出。
2. 写出表示中国境内邮政编码的正则表达式，例如：可匹配100081。（首数字不为0，共6位）
3. 写出能够匹配163邮箱（@163.com）的正则表达式。
4. 简述re模块中re.match()与re.search()的区别。
5. 使用re模块正则提取字符串（例如'abe(ac)ad'）中括号内的内容。
6. 读取文本文件中的中英文数据并使用正则表达式筛选所需数据。要求：从每一行中找到所有的最短子串，它们以'!'开始，以'e'结束。
7. 文本文件depth.xyz中存储了用Tab分割的3列数据，分别表示某区域的经度、纬度和水深值。尝试使用正则表达式提取其中的水深为负值的各行数据，并将其中的负号（-）去除。
8. 分割文本文件2008.txt，并将分割后的各子串保存至文本文件，分别命名为1.txt,2.txt,3.txt...。其中，分隔串是由等号'='构成的字符串，即'==...==', 文本中各分割串长度不等。

习题:

9. 已知中国铁路12306网的各车站的车票起售时间存储在qss.js文件中，试用re模块提取等号右侧的内容，即{}之间（包括{}在内）的内容。

```
var citys = {  
  "北京西":"08:00",  
  "南京":"08:00",  
  "南京南":"08:00",  
  "同江":"16:30",  
  .....  
  "阜新":"15:30",  
  "无为南":"12:30"  
}
```



```
var citys = {  
  "北京西": "08:00",  
  "南京": "08:00",  
  "南京南": "08:00",  
  "同江": "16:30",  
  "山海关": "08:30",  
  "白城": "08:30",  
  "松原": "08:30",  
  "乌兰浩特": "08:30",  
  "济宁": "08:30",  
  "日照": "08:30",  
  "宁波": "09:30",  
  "东莞": "08:30",  
  "东莞东": "08:30",  
  "惠州": "08:30",  
  "汕头": "08:30",  
  "成都东": "08:30",  
  "南充": "08:30",  
  "乌鲁木齐南": "08:30",  
  "南充北": "08:30",  
  "乌鲁木齐": "08:30",  
  "大庆": "09:00",  
  "大庆西": "09:00",  
  "齐齐哈尔": "09:00",  
  "沈阳北": "09:00",  
  "丽水": "12:00",  
}
```

感谢聆听

