文本处理

王肖燕

内容介绍

一．字符串

- 字符串处理函数
- 字符串应用举例

二．正则表达式

- 正则表达式语法
- re模块
- 子模式
- 常用函数
- 应用举例

第一部分

字符串处理函数

字符串

- 不同编码格式之间相差很大
- 采用不同的编码格式
  - 意味着不同的表示和存储形式
- 把同一字符存入文件时，写入的内容可能会不同
- 在试图理解其内容时必须了解编码规则并进行正确的解码
  - 如果解码方法不正确就无法还原信息
- 从这个角度来讲，字符串编码也具有加密的效果

字符串

- Python 3.x完全支持中文字符，默认使用UTF8编码格式，无论是一个数字、英文字母，还是一个汉字，在统计字符串长度时都按一个字符对待和处理。
- >>> s = ‘中国江苏南京'
- >>> len(s)　　#字符串长度，或者包含的字符个数
- 6
- >>> s = ‘中国江苏南京ABCDE'　#中文与英文字符都算一个字符
- >>> len(s)
- 11
- >>> 姓名 = '张三'　　#使用中文作为变量名
- >>> print(姓名)　　#输出变量的值
- 张三

字符串

- 不可变序列类型
  - 除了支持序列通用方法（包括分片操作）以外，还支持特有的字符串操作方法
- Python字符串驻留机制
  - 对于短字符串，将其赋值给多个不同的对象时，内存中只有一个副本，多个对象共享该副本
  - 长字符串不遵守驻留机制

字符串常用方法

- find()、rfind()、index()、rindex()、count()
- find()和rfind方法
  - 分别用来查找一个字符串在另一个字符串指定范围（默认是整个字符串）中首次和最后一次出现的位置，如果不存在则返回-1
- index()和rindex()方法
  - 用来返回一个字符串在另一个字符串指定范围中首次和最后一次出现的位置，如果不存在则抛出异常
- count()方法
  - 用来返回一个字符串在当前字符串中出现的次数

字符串常用方法

```
In [1]: s="apple,peach,banana,peach,pear"

In [2]: s.find("peach")
Out[2]: 6

In [3]: s.find("peach",7)
Out[3]: 19

In [4]: s.rfind("p")
Out[4]: 25

In [5]: s.index("p")
Out[5]: 1

In [6]: s.rindex("p")
Out[6]: 25

In [7]: s.count("p")
Out[7]: 5
```

字符串常用方法

- split()、rsplit()、partition()、rpartition()
- split()和rsplit()方法
  - 分别用来以指定字符为分隔符，把当前字符串从左往右或从右往左分隔成多个字符串，并返回包含分隔结果的列表
- partition()和rpartition()
  - 用来以指定字符串为分隔符将原字符串分隔为3部分，即分隔符前的字符串、分隔符字符串、分隔符后的字符串，如果指定的分隔符不在原字符串中，则返回原字符串和两个空字符串。

字符串常用方法

```
In [1]: s="apple,peach,banana,peach,pear"

In [2]: s.split(",")
Out[2]: ['apple', 'peach', 'banana', 'peach', 'pear']

In [3]: s.partition(",")
Out[3]: ('apple', ',', 'peach,banana,peach,pear')

In [4]: s.rpartition(",")
Out[4]: ('apple,peach,banana,peach', ',', 'pear')
```

```
In [9]: t="\n\nhello\t\t world \n\n\n My name is bob  "

In [10]: t.split(None,1)
Out[10]: ['hello', 'world \n\n\n My name is bob  ']

In [11]: t.rsplit(None,1)
Out[11]: ['\n\nhello\t\t world \n\n\n My name is', 'bob']

In [12]: t.split(None,2)
Out[12]: ['hello', 'world', 'My name is bob  ']

In [13]: t.rsplit(None,2)
Out[13]: ['\n\nhello\t\t world \n\n\n My name', 'is', 'bob']

In [14]: t.split(maxsplit=100)
Out[14]: ['hello', 'world', 'My', 'name', 'is', 'bob']
```

字符串常用方法

- lower()、upper()、capitalize()、title()、swapcase()

- >>> s = "What is Your Name?"
- >>> s.lower()　　#返回小写字符串
- 'what is your name?'
- >>> s.upper()　　#返回大写字符串
- 'WHAT IS YOUR NAME?'
- >>> s.capitalize()　　#字符串首字符大写
- 'What is your name?'
- >>> s.title()　　#每个单词的首字母大写
- 'What Is Your Name?'
- >>> s.swapcase()　　#大小写互换
- 'wHAT IS yOUR nAME?'

字符串常用方法

- 查找替换replace()，类似于Word中的“全部替换”功能。

- >>> s = "中国，中国"
- >>> print(s)
- 中国，中国
- >>> s2 = s.replace("中国", "中华人民共和国") #两个参数都作为一个整理
- >>> print(s2)
- 中华人民共和国，中华人民共和国

举例

- 应用：测试用户输入中是否有敏感词，如果有的话就把敏感词替换为3个星号***。

```
words=("非法","暴力","测试","话")
text="这句话描述的是测试中使用暴力枚举！"
```

举例

- 应用：测试用户输入中是否有敏感词，如果有的话就把敏感词替换为3个星号***。

```
#方法一
words=("非法","暴力","测试","话")
text="这句话描述的是测试中使用暴力枚举！"
for e in words:
    if e in text:
        text=text.replace(e, "***")
print(text)
#方法二
import re
text="这句话描述的是测试中使用暴力枚举！"
pat=r"非法|暴力|测试|话"
r=re.sub(pat,"***",text)
print(r)
```

字符串常用方法

- strip()、rstrip()、lstrip()
- s.startswith(t)、s.endswith(t)，字符串是否以指定字符串开始或结束

```
>>> s = 'Beautiful is better than ugly.'
>>> s.startswith('Be')          #检测整个字符串
>>> s.startswith('Be', 5)       #指定检测范围起始位置
>>> s.startswith('Be', 0, 5)    #指定检测范围起始和结束位置
```

- zfill()返回指定宽度的字符串，在左侧以字符0进行填充。

```
>>> 'abc'.zfill(5)       #在左侧填充数字字符0
'00abc'
>>> 'abc'.zfill(2)       #指定宽度小于字符串长度时，返回字符串本身
'abc'
```

- center()、ljust()、rjust()，返回指定宽度的新字符串，原字符串居中、左对齐或右对齐出现在新字符串中，如果指定宽度大于字符串长度，则使用指定的字符（默认为空格）进行填充。

```
>>> 'Hello world!'.center(20)       #居中对齐，以空格进行填充
>>> 'Hello world!'.center(20, '=')  #居中对齐，以字符=进行填充
>>> 'Hello world!'.ljust(20, '=')   #左对齐
>>> 'Hello world!'.rjust(20, '=')   #右对齐
```

字符串常用方法

- isalnum()、isalpha()、isdigit()、isdecimal()、isnumeric()、isspace()、isupper()、islower()，用来测试字符串是否为数字或字母、是否为字母、是否为数字字符、是否为空白字符、是否为大写字母以及是否为小写字母。

```
>>> '1234abcd'.isalnum()
True
>>> '1234abcd'.isalpha()        #全部为英文字母时返回True
False
>>> '1234abcd'.isdigit()        #全部为数字时返回True
False
>>> 'abcd'.isalpha()
True
>>> '1234.0'.isdigit()
False
```

字符串常量

- Python标准库string中定义数字字符、标点符号、英文字母、大写字母、小写字母等常量。

```
>>> import string
>>> string.digits
'0123456789'
>>> string.punctuation
'!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~'
>>> string.ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

第二部分

正则表达式

内容介绍

- **正则表达式**
  - ✓ 正则表达式元字符
  - ✓ 正则表达式模式
  - ✓ re模块
  - ✓ 常用函数
  - ✓ 应用举例

正则表达式

- 正则表达式使用某种预定义的模式去匹配一类具有共同特征的字符串
  - 记录文本规则的代码
  - 主要用于处理字符串，可以快速、准确地完成复杂的查找、替换等处理要求
  - 在文本编辑与处理、网页爬虫之类的场合中有重要应用
- re模块
  - 提供了正则表达式操作所需要的功能

举例1

- 行定位符，用于描述子串的边界
  - “^”：行的开始
  - “$”：行的结尾
- 例：正则表达式**^tm**, **tm$**
  - “tm equal Tomorrow”
  - “Tomorrow Moon equal tm”
  - 如果要匹配的子串“tm”可以出现在字符串的任意部分，可以写成**tm**

元字符

- 除了“^”和“$”外，还有很多元字符
- 如：\b, \w
  - 正则表达式\bmr\w*\b
  - 匹配以字母mr开头的单词
  - 从单词开始处(\b)，然后匹配字母mr，接着是任意数量的字母或数字(\w*)，最后是单词结束处(\b)
  - 可匹配“mrsoft”、“mrValue”、“mr123456”，而不能与“amr”匹配

正则表达式——元字符

元字符	功能说明
\b	匹配单词头或单词尾
\d	匹配任何数字，相当于[0-9]
\D	与\d含义相反，等效于[^0-9]
\s	匹配任何空白字符，包括空格、制表符、换页符，与 [\f\n\r\t\v] 等效
\S	与\s含义相反
\w	匹配任何字母、数字以及下划线，相当于[a-zA-Z0-9_]
\W	与\w含义相反\w含义相反，与“[^A-Za-z0-9_]”等效
^	匹配行首，匹配以^后面的字符开头的字符串
$	匹配行尾，匹配以$之前的字符结束的字符串
****	表示位于\之后的为转义字符
.	

正则表达式——限定符

元字符	功能说明
*	匹配位于*之前的字符或子模式的0次或多次出现
+	匹配位于+之前的字符或子模式的1次或多次出现
|	匹配位于|之前或之后的字符
()	将位于()内的内容作为一个整体来对待
?	匹配位于?之前的0个或1个字符。当此字符紧随任何其他限定符（*、+、?{n}、{n,}、{n,m}）之后时，匹配模式是“非贪心的”。“非贪心的”模式匹配搜索到的、尽可能短的字符串，而默认的“贪心的”模式匹配搜索到的、尽可能长的字符串。例如，在字符串“oooo”中，“o+?”只匹配单个“o”，而“o+”匹配所有“o”
{m}	匹配前面的字符m次
{m,}	匹配前面的字符最少m次
{m, n}	匹配前面的字符至少m次，至多n次
[]	表范围，匹配位于[]中的任意一个字符，用于没有预定义元字符的字符集合，匹配任意一个汉字[\u4e00-\u9fa5]
-	在[]之内用来表示范围，[a-z]
[^xyz]	反向字符集，匹配除x、y、z之外的任何字符
[^a-z]	反向范围字符，匹配除小写英文字母之外的任何字符

举例2

- 给出正则表达式
  - 匹配元音字母 [aeiou]
  - 匹配非字母字符 [^a-zA-Z]
  - 匹配身份证号 \d{17}(\d|X) \d{17}[0-9X]
    - 18位，前17位为数字，最后一位是校验位，可以为数字或字符“X”
  - 匹配Ip地址，例如：“127.0.0.1” \d{3}\.\d{1,3}\.\d{1,3}\.\d{1,3}

Python中使用正则表达式

- 正则表达式作为模式字符串使用
  - “[^a-zA-Z]”
  - “\bm\w*\b”
- 模式字符串写为原始字符串
  - r“\bm\w*\b”

使用re模块实现正则表达式操作

- import re
  - 用于实现正则表达式的操作

① re模块提供的方法

- search()、match()、findall()

② re模块的compile()方法

- 将模块字符串转换为正则表达式对象

re模块主要方法

方法	功能说明
compile(pattern[, flags])	创建模式对象
findall(pattern, string[, flags])	返回包含字符串中所有与给定模式匹配的项的列表
match(pattern, string[, flags])	从字符串的开始处匹配模式，返回match对象或None
search(pattern, string[, flags])	在整个字符串中寻找模式，返回match对象或None
split(pattern, string[, maxsplit=0])	根据模式匹配项分隔字符串
sub(pat, repl, string[, count=0])	将字符串中所有与pat匹配的项用repl替换，返回新字符串，repl可以是字符串或返回字符串的可调用对象，作用于每个匹配的match对象

match()方法

- 从字符串的开始处进行匹配，如果在起始位置匹配成功，则返回Match对象，否则返回None
  - **re.match(pattern,string,[flags])**
  - pattern：模式字符串
  - string：要匹配的字符串
  - flags：可选参数，标志位，控制匹配方式
    - 如：I或IGNORECASE，表示执行不区分字母大小写的匹配

match()方法

```
import re
pat=r"tm_\w+"
s="Tm_hello tm_Hello"
m=re.match(pat,s,re.I)
print(m)
print("匹配值的开始位置：",m.start())
print("匹配值的结束位置：",m.end())
print("匹配位置的元组：",m.span())
print("要匹配的字符串：",m.string)
print("匹配数据：",m.group())
```

```
<re.Match object; span=(0, 8), match='Tm_hello'>
匹配值的开始位置：  0
匹配值的结束位置：  8
匹配位置的元组：  (0, 8)
要匹配的字符串：  Tm_hello tm_Hello
匹配数据：  Tm_hello
```

search()方法

- 在整个字符串中搜索第一个匹配的值，如果匹配成功，则返回Match对象，否则返回None
  - **re.search(pattern,string,[flags])**
  - pattern：模式字符串
  - string：要匹配的字符串
  - flags：可选参数，标志位，控制匹配方式
    - 如：I或IGNORECASE，表示执行不区分字母大小写的匹配

search()方法

```
import re
pat=r"tm_\w+"
s1="Tm_hello tm_Hello"
m1=re.search(pat,s1,re.I)
print(m1)
s2="中级培训Tm_hello tm_Hello"
m2=re.search(pat,s2,re.I)
print(m2)
```

```
<re.Match object; span=(0, 8), match='Tm_hello'>
<re.Match object; span=(4, 12), match='Tm_hello'>
```

findall()方法

- 在整个字符串中搜索所有匹配的字符串，如果匹配成功，则返回包含匹配结构的列表，否则返回空列表
  - **re.findall(pattern,string,[flags])**
  - pattern：模式字符串
  - string：要匹配的字符串
  - flags：可选参数，标志位，控制匹配方式
    - 如：I或IGNORECASE，表示执行不区分字母大小写的匹配

findall()方法

```
import re
pat=r"tm_\w+"
s1="Tm_hello tm_Hello"
m1=re.findall(pat,s1,re.I)
print(m1)
s2="中级培训Tm_hello tm_Hello"
m2=re.findall(pat,s2,re.I)
print(m2)
```

```
['Tm_hello', 'tm_Hello']
['Tm_hello', 'tm_Hello']
```

sub()方法

- 用于实现字符串的替换
  - **re.sub(pattern,repl,string[,count][,flags])**
  - pattern：模式字符串
  - repl: 替换的字符串
  - string：要被查找替换的原始字符串
  - count: 匹配后替换的最大次数，默认为0：替换所有的匹配
  - flags：可选参数，标志位，控制匹配方式
    - 如：I或IGNORECASE，表示执行不区分字母大小写的匹配

```
import re
pat=r"1[34578]\d{9}"
s="中奖号码为: 84978981 联系电话: 13612345678"
r=re.sub(pat,"1XXXXXXXXXX",s)
print(r)
```

```
中奖号码为: 84978981 联系电话: 1XXXXXXXXXX
```

split()方法

- 根据正则表达式分割字符串，并以列表的形式返回
  - **re.split(pattern,string[,maxsplit][,flags])**
  - pattern：模式字符串
  - string：要匹配的字符串
  - maxsplit：可选参数，最大的拆分次数
  - flags：可选参数，标志位，控制匹配方式
    - 如：I或IGNORECASE，表示执行不区分字母大小写的匹配

```
import re
pat=r"a\wa"
s="bobacabarryabaaliceadawang"
r=re.split(pat,s)
print(r)
```

```
['bob', 'barry', 'alice', 'wang']
```

应用举例1

① 在‘Beautiful is better than ugly.’中

a) 寻找所有的单词；

b) 寻找以字母b开头的完整单词；

② 分隔字符串‘alpha. beta....gamma delta’，以“.”作为分隔符。

③ 把字符串“Mr. Wang, Mr. Ma, Ms. Zhu, Mr. Liu”中的“Mr.”或“Ms.”都替换成“Prof.”。

④ 匹配文本中所有电话号码，并隐藏部分信息，只显示头3位和后4位。

⑤ 去除文本中重复单词，只保留一个不重复的。

正则表达式对象

- re模块的**compile()方法**将正则表达式编译生成正则表达式对象
  - 可以提高字符串处理速度，更强大的文本处理功能
- **match(string[, pos[, endpos]])**
  - 在字符串开头或指定位置进行搜索
  - 模式必须出现在字符串开头或指定位置
- **search(string[, pos[, endpos]])**
  - 在整个字符串中进行搜索
- **findall(string[, pos[, endpos]])**
  - 在字符串中查找所有符合正则表达式的字符串并返回列表
- **sub(repl, string[, count = 0])**
  - 实现字符串替换功能，其中参数repl可以为字符串或返回字符串的可调用对象

正则表达式对象

```
import re
s="baby boy little girl happy"
pat=re.compile(r"\bb\w+\b")
r1=pat.search(s)
print("r1:",r1)
r2=pat.match(s)
print("r2:",r2)
r3=pat.findall(s)
print("r3:",r3)
```

```
r1: <re.Match object; span=(0, 4), match='baby'>
r2: <re.Match object; span=(0, 4), match='baby'>
r3: ['baby', 'boy']
```

正则表达式对象

```
>>> example = '''Beautiful is better than ugly.
        Explicit is better than implicit.
        Simple is better than complex.
        Complex is better than complicated.
        Flat is better than nested.
        Sparse is better than dense.
        Readability counts.'''
>>> pattern = re.compile(r'\bb\w*\b', re.I) #匹配以b或B开头的单词
>>> print(pattern.sub('*', example))    #将符合条件的单词替换为*
* is * than ugly.
Explicit is * than implicit.
Simple is * than complex.
Complex is * than complicated.
Flat is * than nested.
Sparse is * than dense.
Readability counts.
```

Thank You!