第 5 章 字符串和正则表达式

5.1 文本序列类型——字符串

在程序中经常需要处理文本内容，例如在控制台打印程序信息、把文本写入文件等。在 Python 中文本数据由 str 对象或 strings 进行处理。

5.1.1 字符串的创建

字符串就是一串按顺序排列的字符。汉字、字母、数字、空格等都是字符。例如，"hello" 就是一个字符串，它的长度是 5——h、e、l、l、o。字符串中也可以有空格，如"hello world" 包含 11 个字符，其中有一个字符是 "hello" 和 "world" 之间的空格。字符串的长度没有上限，如果字符串的长度是 0，称它为“空字符串”。

字符串的值可以由以下 3 种方式创建：单引号（如'str'）、双引号（如"str"）、三引号（如'''str'''）或者六引号（如"""str"""）。其中，由单引号创建的字符串可以不使用转义字符即可表示双引号；同样，由双引号创建的字符串可以不使用转义字符即可表示单引号。三引号创建的字符串，如果太长而不便于查看代码，可以用反斜杠\来代表代码跨行，但不会输出反斜杠\本身。

【例 5.1】 字符串操作。

```
>>> '"hello",world'
'"hello",world'
>>> print('"hello",world')
"hello",world
>>> print("hello,'world'")
hello,'world'
>>> print("""hello\
 world""")
hello world
```

5.1.2 字符串的转义与连接

字符串的转义符号以反斜杠\开头，和大多数语言一样，\n 代表换行。如果不想让\表示转义，那就要在字符串前面加入符号 r 使用原始字符串。详见例 5.2。

【例 5.2】 字符串转义。

```
>>> print('c:\windows\newfolder')
c:\windows
newfolder
```

```
>>> print(r'c:\windows\newfolder')
c:\windows\newfolder
```

字符串可以用+号进行连接，如果想多次连接同一个字符串，则可以使用*号。

【例 5.3】 字符串连接。

```
>>> print('hello' + 'world')
helloworld
>>> print(2 * 'hello' + 'world')
hellohelloworld
```

表 5.1 是字符转义表，可供参考。

表 5.1 字符转义表

转 义 字 符	描 述
\(在行尾时)	续行符
\\	反斜杠符号
\'	单引号
\"	双引号
\a	警告声
\b	退格
\e	转义
\000	空
\n	换行
\v	纵向制表符
\t	横向制表符
\r	回车
\t	水平制表符
\f	换页
\oyy	八进制数 yy 代表的字符，例如 \o12 代表换行
\xyy	十进制数 yy 代表的字符，例如 \x0a 代表换行
\other	其他字符以普通格式输出

5.1.3 数字字符串与时间的格式化

Python 使用一个字符串作为模板。模板中有格式符，这些格式符为真实值预留位置，并说明真实数值应该呈现的格式。Python 用一个 tuple 将多个值传递给模板，每个值对应一个格式符。其中，模板格式为：

```
[(name)][flags][width].[precision]typecode%(value1,value2,…)
```

其中，数字字符和作用如表 5.2 所示。

其中比较有用的是 m.n，它们可以控制输出浮点数和整数的总宽度以及浮点数的小数精度，数字字符转换方式如表 5.3 所示。

表 5.2　数字字符和作用

符　号	作　用
*	定义宽度或者小数点后数据的精度
–	用作左对齐
+	在正数前面显示加号（+）
（空格键）	在正数前面显示空格
#	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'%'
(var)	映射变量（字典参数）
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）

表 5.3　数字字符转换方式

格式化字符	转 换 方 式
%c	转换为字符（ASCII 码值，或者长度为一的字符串）
%r	优先用 repr()函数进行字符串转换
%s	优先用 str()函数进行字符串转换
%d / %i	转换为有符号十进制数
%u	转换为无符号十进制数
%o	转换为无符号八进制数
%x/%X	（Unsigned）转换为无符号十六进制数（x/X 代表转换后的十六进制字符的大小写）
%e/%E	转换为科学记数法（e/E 控制输出 e/E）
%f/%F	转换为浮点数（小数部分自然截断）
%g/%G	转换为浮点数，根据值的大小采用%e 或 %f 格式
%%	输出%

【例 5.4】 数字字符转换。

```
>>> "%x" % 123
'7b'
>>> "%X" % 123
'7B'
>>> "%#X" % 123
'0X7B'
>>> "%#x" % 123
'0x7b'
>>> '%f' % 1234.567890
'1234.567890'
>>> '%.2f' % 1234.567890
'1234.57'
>>> '%E' % 1234.567890
'1.234568E+03'
>>> '%e' % 1234.567890
'1.234568e+03'
>>> '%g' % 1234.567890
'1234.57'
>>> '%G' % 1234.567890
'1234.57'
```

```
>>> "%e" % (111111111111111111111L)
'1.111111e+21'
>>> print("%22.10e" % (111111111111111111111))
             1.1111111111e+21
```


日期和时间的格式化参数如表 5.4 所示。

表 5.4　日期和时间的格式化参数

格式化字符	转 换 方 式
%a	星期几的简写
%A	星期几的全称
%b	月份的简写
%B	月份的全称
%c	标准的日期的时间串
%C	年份的后两位数字
%d	十进制表示的每月的第几天
%D	月/天/年
%e	在两字符域中，十进制表示的每月的第几天
%F	年-月-日
%g	年份的后两位数字，使用基于周的年
%G	年份，使用基于周的年
%h	简写的月份名
%H	24 小时制的小时
%I	12 小时制的小时
%j	十进制表示的每年的第几天
%m	十进制表示的月份
%M	十时制表示的分钟数
%n	新行符
%p	本地的 AM 或 PM 的等价显示
%r	12 小时的时间
%R	显示小时和分钟，即 hh:mm
%S	十进制的秒数
%t	水平制表符
%T	显示时、分、秒，即 hh:mm:ss
%u	每周的第几天，星期一为第一天（值从 0 到 6，星期一为 0）
%U	每年的第几周，把星期日作为第一天（值从 0 到 53）
%V	每年的第几周，使用基于周的年
%w	十进制表示的星期几（值从 0 到 6，星期天为 0）
%W	每年的第几周，把星期一作为第一天（值从 0 到 53）
%x	标准的日期串
%X	标准的时间串
%y	不带世纪的十进制年份（值从 0 到 99）
%Y	带世纪部分的十进制年份
%z	%Z 时区名称，如果不能得到时区名称则返回空字符
%%	百分号

5.1.4 字符串的索引与切片

与 C 语言相似，Python 中字符串可以通过下标访问某个字符；不同之处在于，Python 中字符串的索引更加灵活，还具备了非常有用的切片操作。字符串索引和切片的用法与列表相同，此处不再赘述。

需要注意的是，字符串一旦创建就无法更改。所以无法通过字符串的索引进行单个字符的修改或重新赋值。可以利用索引及切片操作重新创建一个新的字符串。

5.1.5 常见的字符串操作

内置的字符串函数主要有 len()、endswith()、startswith()、lower()、upper()、capitalize()、find()、index()、isalpha()、isdigit()、join()、replace()、split()等。

【例 5.5】 常见的字符串操作。

```
>>> word = 'hello,world'
>>> len(word)
11
>>> word.endswith('world')
True
>>> word.endswith('world.')
False
>>> word.startswith('hello')
True
>>> word.startswith('Hello')
False
>>> word.capitalize()
'Hello,world'
>>> word.find('ello')
1
>>> word.find('abcd')
-1
>>> word.find('hello')
0
>>> 'ello' in word
True
>>> 'abcd' in word
False
>>> word.index("hw")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
>>> word = 'hello,world'
>>> word.isalpha()
False
>>> word = 'helloworld'
>>> word.isalpha()
True
>>> number = '1234.5678'
>>> number.isdigit()
False
>>> number = '12345678'
>>> number.isdigit()
```

```
True
>>> l = [1, 2, 3]
>>> print(l)
[1, 2, 3]
>>> ','.join(l)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sequence item 0: expected str instance, int found
>>> l = ['1', '2', '3']
>>> print(l)
['1', '2', '3']
>>> ','.join(l)
'1,2,3'
>>> word = 'hello'
>>> word.replace('llo', 'llo world')
'hello world'
>>> l = '1, 2, 3'
>>> l.split(',')
['1', ' 2', ' 3']
>>> word = '   hello world   '
>>> word
'   hello world   '
>>> word.strip()
'hello world'
>>> word.strip(' hd')
'ello worl'
```

关键字 in 可以判断一个字符串是否出现在另一个字符串中。

```
>>> word = 'hello,world'
>>> 'hello' in word
True
>>> 'helloworld' in word
False
```

5.2 正则表达式

正则表达式是对字符串操作的一种逻辑公式,就是用事先定义好的一些特定字符及这些特定字符的组合，组成一个“规则字符串”，这个“规则字符串”用来表达对字符串的一种过滤逻辑。正则表达式是一种文本模式，该模式描述在搜索文本时要匹配的一个或多个字符串。在很多文本编辑器里，正则表达式通常被用来检索、替换那些匹配某个模式的文本。它在程序处理文字的时候非常有用。Python 加入了 re 模块，提供与 Perl 类似的字符串正则操作。在使用正则表达式前，需要用语句 import　re 导入 re 模块。

5.2.1 正则表达式的语法

正则表达式可以包含普通字符和特殊（转义）字符。只使用普通字符的正则表达式是最简单的正则表达式，因为它只和自己匹配，如 A 匹配 A，b 匹配 b。然而实际运用中往往需要更高级的正则表达式。例如在一段文本中找到某个人的邮箱，假设这个人的邮箱是由一段数字+@符号+email.com 组成，当不知道那段数字时，需要使用特殊（转义）字符模糊匹配那段数

字，相关示例如下所示：

```
>>> import re
>>> text="hello's email:123456789@email.com"
>>> re.findall(r'\d+@email.com', text)
['123456789@email.com']
```

表 5.5 列出了正则表达式模式语法中的特殊元素。如果使用模式的同时提供了可选的标志参数，某些模式元素的含义会改变。

表 5.5　正则表达式模式语法中的特殊元素

模　式	描　述
^	匹配字符串的开头
$	匹配字符串的末尾
.	匹配任意字符，除了换行符。当 re.DOTALL 标记被指定时，则可以匹配包括换行符的任意字符
[⋯]	用来表示一组字符，单独列出，如[amk] 匹配 'a'、'm'或'k'
[^⋯]	不在[]中的字符，如[^abc] 匹配除了 a、b、c 之外的字符
re*	匹配 0 个或多个表达式
re+	匹配 1 个或多个表达式
re?	匹配 0 个或 1 个由前面的正则表达式定义的片段，非贪婪方式
re{ n}	匹配 n 个前面表达式。例如，"o{2}"不能匹配"Bob"中的"o"，但是能匹配"food"中的两个 o
re{ n,}	精确匹配 n 个前面表达式。例如，"o{2,}"不能匹配"Bob"中的"o"，但能匹配"foooood"中的所有 o。"o{1,}"等价于"o+"，"o{0,}"则等价于"o*"
re{ n, m}	匹配 n～m 次由前面的正则表达式定义的片段，贪婪方式
a| b	匹配 a 或 b
(re)	匹配括号内的表达式，也表示一个组
(?imx)	正则表达式包含 3 种可选标志：i、m 或 x。只影响括号中的区域
(?-imx)	正则表达式关闭 i、m 或 x 可选标志。只影响括号中的区域
(?: re)	类似 (...)，但是不表示一个组
(?imx: re)	在括号中使用 i、m 或 x 可选标志
(?-imx: re)	在括号中不使用 i、m 或 x 可选标志
(?#⋯)	注释
(?= re)	前向肯定界定符。如果所含正则表达式，以⋯表示，在当前位置成功匹配时则成功，否则失败。但一旦所含表达式已经尝试，匹配引擎根本没有提高；模式的剩余部分还要尝试界定符的右边
(?! re)	前向否定界定符。与肯定界定符相反。当所含表达式不能在字符串当前位置匹配时成功
(?> re)	匹配的独立模式，省去回溯
\w	匹配字母数字
\W	匹配非字母数字
\s	匹配任意空白字符，等价于 [\t\n\r\f]

续表

模　式	描　述
\S	匹配任意非空字符
\d	匹配任意数字，等价于 [0-9]
\D	匹配任意非数字
\A	匹配字符串开始
\Z	匹配字符串结束，如果存在换行，则只匹配到换行前的结束字符串
\z	匹配字符串结束
\b	匹配一个单词边界，也就是单词和空格间的位置。例如，'er\b' 可以匹配"never" 中的 'er'，但不能匹配 "verb" 中的 'er'
\B	匹配非单词边界。例如，'er\B' 能匹配 "verb" 中的 'er'，但不能匹配 "never" 中的 'er'
\n, \t, 等	匹配一个换行符，匹配一个制表符，等
\1…\9	匹配第 n 个分组的内容
(?:…)	匹配一个不用保存的分组
贪婪模式*?、+?、??	使正则表达式尽可能匹配多次
(?P=name)	匹配任何命名为 name 的文本
(?P<name>…)	这个正则表达式匹配到的子字符串只能由 name 命名访问到

与此同时，表 5.6 列举出了一些常用的转义序列，供读者参考。

表 5.6　常用的转换序列

元 字 符	功 能 说 明
\n	匹配换行符
\f	匹配换页符
\A	匹配字符串的开头
\b	匹配单词的开头和结尾
\d	匹配任意 Unicode 数字。如果只想匹配 ASCII 数字，推荐使用[0-9]匹配
\D	匹配任意不是 Unicode 数字的字符
\s	匹配 Unicode 空格字符或 ASCII 空格字符，取决于匹配模式
\r	匹配一个回车符
\w	匹配任何字母、数字以及下画线
[a-z]	匹配 a ~ z 的任意字符
[^a-z]	匹配除 a ~ z 的任意字符

正则表达式匹配时可以包含一些可选的特殊参数来控制匹配的模式，如表 5.7 所示。多个参数可以通过按位 OR(|)来指定，如 re.I | re.M 被设置成 I 和 M 标志。

表 5.7　正则表达式匹配模式

模　式	描　述
re.I	使匹配对大小写不敏感
re.L	做本地化识别匹配
re.M	多行匹配，可能会对^和$符号产生影响
re.S	使 . 匹配包括换行在内的所有字符
re.U	根据 Unicode 字符集解析字符。这个标志影响 \w, \W, \b, \B
re.X	该标志通过给予更灵活的格式以便正则表达式写得更易于理解

5.2.2　正则表达式与 Python 语言

re.compile(pattern, flags=0)：可以将正则表达式模式编译成一个正则表达式对象。如果这个正则表达式在程序中需要多次使用，那么最好先行编译一下以提高程序效率。

Pattern：一个字符串形式的正则表达式。

Flags：可选，表示匹配模式，如忽略大小写、多行模式等。具体可用值为：

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：即为' . '并且包括换行符在内的任意字符（' . '不包括换行符）；

re.U：表示特殊字符集 \w, \W, \b, \B, \d, \D, \s, \S，依赖于 Unicode 字符属性数据库；

re.X：为了增加可读性，忽略空格和' # '后面的注释。

```
>>> prog = re.compile(pattern)
>>> result = prog.match(string)
```

re.match(pattern, string, flags=0)：如果字符串开头 0 个或多个字符与 pattern 匹配，则返回相应的匹配对象，否则返回 None。各参数的含义如下：

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。

可以使用 group(num) 或 groups() 匹配对象函数来获取匹配表达式。

group(num=0)匹配的整个表达式的字符串。group() 可以一次输入多个组号，在这种情况下它将返回一个包含那些组所对应值的元组；groups()返回一个包含所有小组字符串的元组、从 1 到所含的小组号。

【例 5.6】 字符串匹配。

```
>>> import re
>>> string = "123456789@email.com"
>>> pattern = r'\d+@email.com'
>>> prog = re.compile(pattern)
>>> result = prog.match(string)
>>> print(result)
<_sre.SRE_Match object; span=(0, 19), match='123456789@email.com'>
```

re.search(pattern, string, flags=0)搜索 string 中可以匹配到的第一个位置，并返回相应的匹

配对象。如果字符串中没有位置与 pattern 匹配，则返回 None，参数的含义同 re.match 参数的含义。

【例 5.7】 字符串查找。

```
>>> import re
>>> string = "hello's email:123456789@email.com"
>>> pattern = r'\d+@email.com'
>>> prog = re.compile(pattern)
>>> result = prog.search(string)
>>> print(result)
<_sre.SRE_Match object; span=(14, 33), match='123456789@email.com'>
```

【**re.match 与 re.search 的区别**】 re.match 只匹配字符串的开始，如果字符串开始不符合正则表达式，则匹配失败，函数返回 None；而 re.search 匹配整个字符串，直到找到一个匹配。

re.split(pattern, string[, maxsplit=0, flags=0]) 按照能够匹配的子串将字符串分割后返回列表。各参数的含义如下：

pattern：匹配的正则表达式。

string：要匹配的字符串。

maxsplit：分割次数，maxsplit=1 分割一次，默认为 0，不限制次数。

re.findall(pattern, string, flags=0)搜索 string 所有可以匹配到 pattern 的非重复子字符串，并返回列表。各参数的含义如下：

pattern：匹配的正则表达式。

string：待查找的字符串。

pos：可选参数，指定字符串的起始位置，默认为 0。

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。

【例 5.8】 字符串查找。

```
>>> import re
>>> string = """
... hello's email:123456789@email.com
... world's email:987654321@email.com
... """
>>> pattern = r'\d+@email.com'
>>> prog = re.compile(pattern)
>>> result = prog.findall(string)
>>> print(result)
['123456789@email.com', '987654321@email.com']
```

re.finditer(pattern, string, flags=0)与 findall()函数相同，但返回的不是一个列表，而是一个迭代器。

【例 5.9】 查找字符串返回迭代器。

```
>>> import re
>>> string = """
... hello's email:123456789@email.com
... world's email:987654321@email.com
... """
>>> pattern = r'\d+@email.com'
```