# 第5章 字符串与字符编码

字符串(str)是由字符序列构成的对象,以一对英文的引号(单引号、双引号或三引号)为边界符(定界符)。引号之间的字符序列是字符串的内容。作为一种序列,字符串支持序列的一系列通用操作,如按索引访问元素、切片、成员测试、计算长度等。字符串是一种不可变的可迭代对象,一旦创建完成,该对象就不可修改。

字符串的部分内容已经在第2章和第4章中有所介绍。本章先补充介绍字符串的构造方法,接着介绍字符集与字符编码,然后介绍字符串的格式化方法,最后介绍字符串的一些常用方法。

## 5.1 字符串构造

在Python中,字符串的构造主要通过两种方法来实现,一种是使用str类来构造字符串对象;另一种是以成对的单引号、双引号或三引号为边界符,直接将字符序列括起来。利用英文引号作为定界符构造字符串的方法在2.1节中已经介绍过了,本章不再赘述。

**1. 用str类来构造字符串**

可以用str(object='')或str(bytes_or_buffer[,encoding[,errors]])两种格式来构造字符串。第一种格式是从一个对象object来构造字符串;第二种格式是将一字节串解码为一个字符串。这种格式将在5.2节字符编码中阐述。

用str(object='')生成字符串时,参数object的默认值为空字符串,如果没有传递参数,默认得到一个空字符串。例如:

```
>>> x = str()          #生成一个空字符串
>>> x
''
>>> len(x)             #字符串x中的元素个数
0
```

如果给str(object='')中的参数object传递一个对象,则分为以下两种情况。

(1) 如果该object对象所属的类中定义了__str__()方法,则按照该对象的__str__()方法返回结果(本书不阐述该内容)。

(2) 如果该object对象所属的类没有定义__str__()方法,则在该对象两侧添加字符串边界符来生成字符串对象。例如:

```
>>> str(10)
'10'
>>> y = str([1,2,3])
```

```
>>> y
'[1, 2, 3]'
>>> len(y)              #列表中每个逗号后面自动添加了一个空格,所以长度为9
9
```

### 2. 单、双引号构造字符串的注意事项

如果字符串序列中的元素包含单引号，且不用转义字符，那么整个字符串要用双引号或三引号作为边界符来构造，否则就会出错。例如：

```
>>> "I'm a student."
"I'm a student."
>>> print("I'm a student.")      #print()函数输出字符串时,不显示引号边界符
I'm a student.
>>> '''I'm a student.'''
"I'm a student."
>>> """I'm a student."""
"I'm a student."
>>> 'I'm a student.'             #出错
File "<console>", line 1
'I'm a student.'
               ^
SyntaxError: unterminated string literal (detected at line 1)
>>>
```

如果字符串序列的元素包含双引号，且不用转义字符，那么整个字符串要用单引号或三引号作为边界符来构造，否则就会出错。例如：

```
>>> '"I am a student", he said.'
'"I am a student", he said.'
>>> '''"I am a student", he said.'''
'"I am a student", he said.'
>>> """"I am a student", he said."""
'"I am a student", he said.'
>>> ""I am a student", he said."      #出错
File "<console>", line 1
""I am a student", he said."
  ^
SyntaxError: invalid syntax
>>>
```

上述案例中，当字符串中出现了与字符串边界符相同的字符时，该字符会被 Python 解释器误认为是字符串边界符，从而导致语法错误。为了告诉系统，该字符只是普通字符而不是字符串边界符，需要在该字符前面添加反斜线（“\”）。例如：

```
>>> 'I\'m a student.'
"I'm a student."
>>> "\"I am a student\", he said."
'"I am a student", he said.'
>>>
```

### 3. 转义字符

在单引号或双引号前面加反斜线（“\”）是对反斜线后面的引号进行转义，表示该引号不是字符串的边界符。计算机中的一些不可见字符通常需要使用转义来表达。转义字符以“\”开头，后接某些特定的字符或数字。Python 中常用的转义字符如表 5.1 所示。

表 5.1　Python 中常用转义字符

| 转义字符 | 含　义 | 转义字符 | 含　义 | 转义字符 | 含　义 |
|---|---|---|---|---|---|
| \(位于行尾) | 续行符 | \" | 双引号(") | \t | 横向(水平)制表符 |
| \\ | 一条反斜线(\) | \n | 换行符 | \v | 纵向(垂直)制表符 |
| \' | 单引号(') | \r | 回车 | \f | 换页符 |

以下例子给出了换行符和制表符的转义表示。

```
>>> s = "我们一起学习 Python。\nPython 语言功能强大。"   # \n 表示换行
>>> s
'我们一起学习 Python。\nPython 语言功能强大。'
>>> print(s)                                              # 打印时,\n 产生了换行的效果
我们一起学习 Python。
Python 语言功能强大。
>>> print("我们一起学习 Python.\t\tPython 语言功能强大。")
我们一起学习 Python.      Python 语言功能强大。
>>>
```

字符串中,一个横向制表符(\t)相当于按一次 Tab 键。

**4. 原始字符串**

在字符串中,反斜线("\")和后面字符的组合通常表示一个特殊的转义字符。如果想让反斜线自身表示一个普通的单独字符,避免其与后面的字符组合为转义字符,则需要在字符串左侧边界符前面添加字符 r。

在 Windows 下,文件路径通常用反斜线分隔,如果反斜线后面是 t 或 n 等字符,默认就会构成制表符、换行符等特殊转义字符,在程序中无法表示正确的文件路径。这时可以在表示路径的字符串左侧边界符前面添加字符 r,避免反斜线与后面的字符构成转义字符。例如:

```
>>> print("c:\next\test")          # 不能正确表示路径
c:
ext est
>>> print(r"c:\next\test")         # 可以正确表示路径
c:\next\test
>>>
```

**5. 三引号构造字符串的进一步说明**

三引号构造的字符串可以跨越多行,并保留换行符、引号、制表符等信息。在一对三引号之间可以自由地使用单引号和双引号,这些单引号或双引号都是字符串序列中的元素。例如:

```
>>> s = '''英文缩写"AI"表示人工智能.
... 人工智能可以帮助人们提高生产效率'''
>>> s          # 查看字符串 s,换行符用\n 表示了
'英文缩写"AI"表示人工智能.\n 人工智能可以帮助人们提高生产效率'
>>> print(s)  # 打印时,字符串换行了
英文缩写"AI"表示人工智能。
人工智能可以帮助人们提高生产效率
>>>
```

编写 Python 程序时,可以使用三引号来构造跨越多行的注释语句块。

# 5.2 字符集与字符编码

## 5.2.1 字符集与编码方法

任何字符在计算机中都是以特定的编码来表示、存储和传输的。为了方便相互之间的信息交换和识别，在计算机领域先后制定了多种编码方式。

### 1. ASCII 字符集与编码

ASCII(American Standard Code for Information Interchange，美国信息交换标准代码)是基于拉丁字母的一套计算机编码系统，主要用于表示现代英语和其他西欧语言。ASCII字符集包括英文字母、数字、英文标点符号等常用的字符，如表 5.2 所示。其中，数字 0～127称为字符的码点值(编号)。在计算机中要用特定的二进制编码方式来表示这个码点值，才能进行存储、转发、识别。标准 ASCII(基础 ASCII)将码点值用 7 位二进制数来表示。由于计算机中通常以字节(8 位)为单位来表示信息，因此 ASCII 规定这个字节中剩下的最高位为二进制 0。标准 ASCII 的码点值(编号)与字符的对照关系如表 5.2 所示。

**表 5.2 标准 ASCII 与字符的对照关系**

| 编号 | 字符 | 编号 | 字符 | 编号 | 字符 | 编号 | 字符 | 编号 | 字符 | 编号 | 字符 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | NUL | 22 | SYN | 44 | , | 66 | B | 88 | X | 110 | n |
| 1 | SOH | 23 | ETB | 45 | — | 67 | C | 89 | Y | 111 | o |
| 2 | STX | 24 | CAN | 46 | . | 68 | D | 90 | Z | 112 | p |
| 3 | ETX | 25 | EM | 47 | / | 69 | E | 91 | [ | 113 | q |
| 4 | EOT | 26 | SUB | 48 | 0 | 70 | F | 92 | \ | 114 | r |
| 5 | ENQ | 27 | ESC | 49 | 1 | 71 | G | 93 | ] | 115 | s |
| 6 | ACK | 28 | FS | 50 | 2 | 72 | H | 94 | ^ | 116 | t |
| 7 | BEL | 29 | GS | 51 | 3 | 73 | I | 95 | _ | 117 | u |
| 8 | BS | 30 | RS | 52 | 4 | 74 | J | 96 | ` | 118 | v |
| 9 | HT | 31 | US | 53 | 5 | 75 | K | 97 | a | 119 | w |
| 10 | LF | 32 | (space) | 54 | 6 | 76 | L | 98 | b | 120 | x |
| 11 | VT | 33 | ! | 55 | 7 | 77 | M | 99 | c | 121 | y |
| 12 | FF | 34 | " | 56 | 8 | 78 | N | 100 | d | 122 | z |
| 13 | CR | 35 | # | 57 | 9 | 79 | O | 101 | e | 123 | { |
| 14 | SO | 36 | $ | 58 | : | 80 | P | 102 | f | 124 | \| |
| 15 | SI | 37 | % | 59 | ; | 81 | Q | 103 | g | 125 | } |
| 16 | DLE | 38 | & | 60 | < | 82 | R | 104 | h | 126 | ～ |
| 17 | DC1 | 39 | ' | 61 | = | 83 | S | 105 | i | 127 | DEL |
| 18 | DC2 | 40 | ( | 62 | > | 84 | T | 106 | j | | |
| 19 | DC3 | 41 | ) | 63 | ? | 85 | U | 107 | k | | |
| 20 | DC4 | 42 | * | 64 | @ | 86 | V | 108 | l | | |
| 21 | NAK | 43 | + | 65 | A | 87 | W | 109 | m | | |

### 2. GBK 字符集与编码

当计算机的应用逐步推广时，各种语言的文字字符都需要进行编码，以方便计算机的存储和信息的传输与交换。

GB2312是我国制定的简体中文编码规则，GBK是对GB2312的扩充。GBK编码在Windows内部对应其代码页为cp936。GBK字符集中包含常用中文字符和ASCII字符。GBK字符集也用码点值表示相应的编号，其中，ASCII字符保持其码点值不变。为了兼容ASCII，ASCII字符集中的字符采用一字节的二进制对码点值（编号）进行编码，该字节的最高位为0。使用2字节的二进制编码表示一个中文字符的码点值，其中，高字节的最高位为1。

在GBK字符集中，“我”的码点值（编号）为20178，转换为二进制为1001110 11010010。系统能够检测到需要使用2字节才能表示这个编码，因此认定为中文字符，将高字节的最高位设定为1，得到该字符的GBK二进制编码为11001110 11010010。

在Python中，可以使用字符串的encode()方法查看字符在特定编码体系下的十六进制编码的字节串（bytes类型）。例如：

```
>>> "我".encode('gbk')
b'\xce\xd2'
>>>
```

在GBK字符集中，当中文字符和ASCII字符编码混合出现时，如果第一字节中的最高位为0，则取一字节去掉最高位后即为该字符的二进制码点值。如果第一字节最高位为1，则取连续的两字节，去掉高字节的最高位1，后15位即为相应中文字符的二进制码点值。根据码点值就可以确定对应的字符。

例如，十六进制显示的字节串b'\xce\xd2'，转换成二进制即为11001110 11010010。转换方法如下。

```
>>> int('ced2', 16)      #先将十六进制ced2转换为十进制的整数52946
52946
>>> bin(52946)           #再将十进制的整数52946转换为二进制数
'0b1100111011010010'
>>>
```

去掉高字节的最高位1后，后15位为1001110 11010010。将此二进制数字转换为十进制方法如下。

```
>>> int("100111011010010",2)
20178
>>>
```

在GBK编码表中可以查到，编号为20178的字符是“我”。

### 3. Unicode字符集与编码

不同国家有不同的语言，也就有不同的字符编码规则。不同的编码规则格式之间差别很大，不同编码的长度可能不同，并且同一编码在不同的编码体系中可能表示不同的字符。如果一篇文章既有英文，又有中文，还有日文，那无论采用上述哪种编码规则，都可能会出现乱码。

为了方便信息的交流，Unicode字符集包括全世界所有语言的字符。不同的字符在Unicode字符集中有不同的码点值（编号）。可以使用ord()函数来查询字符对应的Unicode码点值，可以利用chr()函数查询Unicode码点值对应的字符。例如：

```
>>> ord("a")
97
```

```
>>> chr(97)
'a'
>>> ord("我")
25105
>>> chr(25105)
'我'
>>>
```

虽然有了统一的 Unicode 码点值，但还无法确定如何用二进制编码来表示、存储和传递码点值。因此，需要一套编码规则来表示码点值。

UTF-32 编码规则使用 4 字节（32 位）对 Unicode 字符集中的码点值（编号）进行编码，任何一个字符均需要 4 字节的编码表示一个字符的码点值（编号）。这种方式以 4 字节为一组表示一个字符的编号，实现简单。但对一些码点值比较小的字符来说，会造成大量的存储空间浪费和传输时网络资源的浪费。例如，对于英文字母、英文标点符号等字符的码点值（编号）最大不超过 127，只需要 1 字节就可以编码；如果也采用 4 字节的编码，那么每个字符前 3 字节的编码存储空间就会浪费掉。

为了解决 UTF-32 在编码 Unicode 字符集时的资源浪费问题，后来又陆续提出了目前比较常用的 UTF-16 和 UTF-8 编码。目前最常用的是 UTF-8 编码。

UTF-8 编码是“可变长编码”，使用 1～4 字节的编码表示 Unicode 字符集中的一个码点值（对应一个字符）。编码的长度根据不同的字符（不同的码点值）而有所变化。它以 1 字节对英文字符（兼容 ASCII）进行编码，以 3 字节或 4 字节对中文字符进行编码，还有一些语言的字符使用 2 字节、3 字节或 4 字节进行编码。

表 5.3 给出了 Unicode 字符的码点值与 UTF-8 编码格式的对应关系。1～127 的码点值兼容 ASCII，用 1 字节编码，其中最高位为 0，后 7 位是码点值对应的二进制编码。128～2047 的码点值用 2 字节编码，其中高字节的最高三位为 110，低字节的最高两位为 10。这两个字节中其他位上根据码点值的二进制位依次从低到高进行填充，不足部分填 0。2048～65 535 的码点值用 3 字节编码，其中高字节的最高四位为 1110，后面两字节的最高两位均为 10。这三个字节的其他位上根据码点值的二进制位依次从低到高进行填充，不足部分填 0。“我”这个字符的 Unicode 码点值为 25105。可以用 ord()函数获得该字符的 Unicode 码点值（编号），例如：

**表 5.3　Unicode 码点值与 UTF-8 编码格式对应关系**

| Unicode 字符码点值十六进制范围 | Unicode 字符码点值十进制范围 | UTF-8 的二进制编码格式 |
|---|---|---|
| 0x00～0x7F | 0～127 | 0×××××××× |
| 0x80～0x7FF | 128～2047 | 110×××××　10×××××× |
| 0x800～0xFFFF | 2048～65 535 | 1110××××　10××××××　10×××××× |
| 0x10000～0x10FFFF | 65 536～1 114 111 | 11110×××　10××××××　10××××××　10×××××× |

```
>>> ord("我")
25105
>>>
```

码点值 25105 转换为二进制值为 110001000010001。计算如下。

```
>>> bin(25105)
'0b110001000010001'
>>>
```

根据表 5.3 中的规则，UTF-8 对该 Unicode 码点值的编码规则为 1110××××10×××××××10××××××，其中，×表示码点值中的二进制位对应的值。先把 25105 对应的二进制值 110001000010001 后 6 位填充到 UTF-8 编码的第三字节最后 6 位(010001)，加上最后一字节的最高两位 10，得到最后一字节的编码为 10010001；然后将 25105 对应的二进制值 110001000010001 倒数第 7 位至倒数第 12 位(001000)填充到 UTF-8 编码的第二字节后 6 位，加上第二字节的最高两位为 10，得到第二字节的编码为 10001000；最后将 25105 对应的二进制值 110001000010001 中剩余的最高三位值(110)填充到 UTF-8 编码的第一字节最低三位，加上第一字节最高四位的 1110，第五位因为空余补 0，得到第一字节的编码为 11100110。所以中文字符"我"的 UTF-8 编码是 111001101000100010010001，对应的十六进制编码字节串为 b'\xe6\x88\x91'。可以用 encode()方法查看字符在特定编码体系下的十六进制编码字节串。例如：

```
>>> "我".encode("UTF-8")
b'\xe6\x88\x91'
>>>
```

在 Python 中，可以先把十六进制的字符串转换为十进制整数，再将十进制整数转换为二进制来查看二进制编码。例如：

```
>>> int('e68891',16)      # 这是 UTF-8 编码对应的数值，注意与 Unicode 编号的区别
15108241
>>> bin(15108241)
'0b111001101000100010010001'
>>>
```

码点值大于或等于 65 536 的 Unicode 字符采用 4 字节的 UTF-8 编码，最高字节的前 5 位为 11110，后面连续的 3 字节最高两位均为 10。这 4 字节其余位上的编码填充方式与 2 字节或 3 字节编码的填充方式类似，这里不再赘述。

根据 UTF-8 编码寻找对应字符时，如果第一字节中的最高位为 0，则取一字节去掉最高位后剩余的二进制值即为相应字符的二进制码点值。如果第一字节最高三位为 110，则取连续的两字节，去掉高字节的最高三位 110 和低字节的最高两位 10，剩余的二进制位即构成该字符的二进制码点值。如果第一字节最高四位为 1110，则取连续的三字节，去掉高字节的最高四位 1110 和后两字节的各自最高两位 10，剩余的二进制位即构成该字符的二进制码点值。4 字节编码的情况类似。根据码点值就可以确定对应的字符。

## 5.2.2　字符与编码的转换

Python 中一个或多个字符均用字符串表示，类型为 str。字符或字符串对应的 UTF-8、GBK 等编码用字节串表示，类型为 bytes。字符串 str 和字节串 bytes 均为不可变对象类型。

str 类型和 bytes 类型经常需要相互转换。例如，字符串需要转换为相应编码的字节串在网络上进行传输，对方接收到字节串后需要根据相同的编码体系规则转换为字符串呈现出来。

str 类里有一个 encode()方法，根据字符串生成由参数中 encoding 指定的编码方式编码的字节串。bytes 类有一个 decode()方法，将字节串使用参数 encoding 指定的编码方式解码成为字符串 str 类型的数据。例如：

```
>>> s = "学生"
>>> s1 = s.encode("gbk")          #用 GBK 编码
>>> s1
b'\xd1\xa7\xc9\xfa'
>>> s2 = s.encode("utf-8")        #用 UTF-8 编码
>>> s2
b'\xe5\xad\xa6\xe7\x94\x9f'
>>> s1.decode("gbk")              #用与编码相同的规则解码,否则会产生乱码
'学生'
>>> s2.decode("utf-8")            #用与编码相同的规则解码,否则会产生乱码
'学生'
>>> s3 = s.encode("ascii")        #中文字符串不能以 ASCII 编码
Traceback (most recent call last):
File "<console>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-1: ordinal not in range
(128)
```

另外，可以使用 bytes(string,encoding[,errors])根据参数中 encoding 指定的编码规则来生成字符串 string 对应的编码，该编码为 bytes 类型的字节串。可以使用 str(bytes_or_buffer[,encoding[,errors]])根据参数中 encoding 指定的解码规则来生成字节串 bytes_or_buffer 对应的字符串。例如：

```
>>> s = "学生"
>>> b1 = bytes(s,encoding = "gbk")      #用 GBK 编码规则对字符串编码
>>> b1
b'\xd1\xa7\xc9\xfa'
>>> b2 = bytes(s,encoding = "utf-8")    #用 UTF-8 编码规则对字符串编码
>>> b2
b'\xe5\xad\xa6\xe7\x94\x9f'
>>> str(b1,encoding = "gbk")            #用与编码相同的规则解码,否则会产生乱码
'学生'
>>> str(b2,encoding = "utf-8")          #用与编码相同的规则解码,否则会产生乱码
'学生'
```

Python 3 中采用 Unicode 字符，数字字符、英文字母、汉字等都按一个字符来对待和处理。例如：

```
>>> len("大学生")                    #字符串中包含三个字符
3
>>> s = "我今年 18 岁.我叫 Mike Li"  #Mike 与 Li 之间有一个空格,其余无空格
>>> len(s)
16
>>>
```

## 5.3 字符串格式化

用加号拼接字符串可以生成满足某些格式要求的字符串，但通常需要复杂或大量的程序代码。例如，需要先将其他类型的对象转换为字符串对象。字符串格式化的方法使得程

序可以在字符串中嵌入变量并定义变量代入的格式。本节分别介绍利用%运算符、字符串的 format()方法、format_map()方法、f_string()字面量方法进行字符串格式化的过程。

## 5.3.1　用%格式化字符串

用%进行字符串格式化涉及两个概念：格式定义和格式化运算。字符串内部格式的定义以%开头。字符串后面的格式化运算符%表示用其后面的对象代替格式串中的格式，最终得到一个字符串。

字符串格式化的一般形式如图 5.1 所示。这里字符串中只给出了一个格式定义。字符串中，格式定义的两端均可以有普通字符或其他字符串格式的定义。

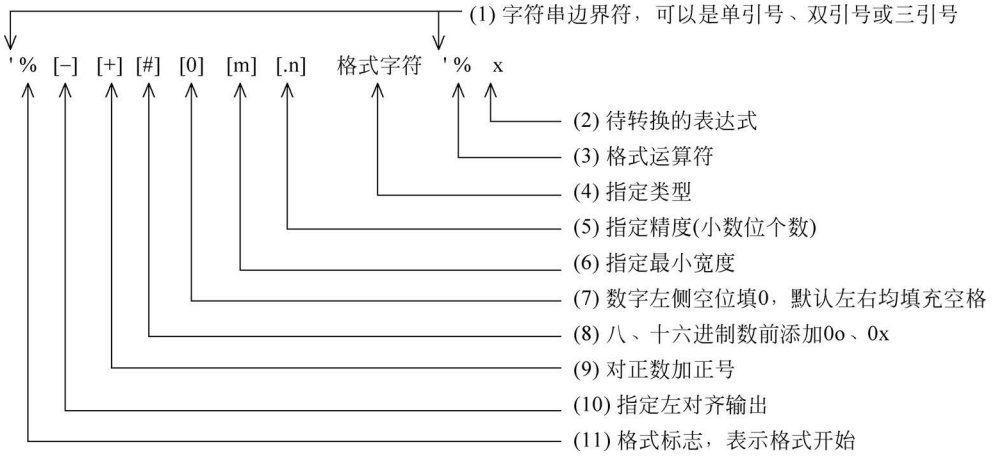


图 5.1　字符串格式化的一般形式

常用的格式字符含义如表 5.4 所示。

**表 5.4　字符串的格式字符**

| 格　　式 | 说　　明 |
|---|---|
| %c | 格式化字符或编码 |
| %s | 格式化字符串等任何类型的对象 |
| %d，%i | 格式化整数 |
| %u | 格式化无符号整数 |
| %% | 百分号(%) |
| %o | 格式化八进制数 |
| %x | 格式化十六进制数 |
| %f | 格式化浮点数，默认保留 6 位小数，可指定小数位数 |
| %F | 同%f；并且将 inf 和 nan 分别转换为 INF 和 NAN |
| %e、%E | 分别用 e 和 E 表示科学记数法格式的浮点数，如 1.2e+03 表示 $1.2\times10^3$ |
| %g、%G | 根据值的大小采用科学记数法或者浮点数形式；采用科学记数法时，分别用 e 和 E 表示；当数值中的数字个数大于 6 时，默认保留 6 个数字，可以自己指定保留的数字个数 |

在如图 5.1 所示的字符串格式定义中，方括号[ ]中的内容可以省略，最简单的格式是%后面加格式字符，如%f、%d、%c 等。例如：

```
>>> "%c" % 90      #字符 Z 的 Unicode 编号为 90
'Z'
```

```
>>> ord("Z")
90
>>>
```

最小宽度是格式化后的值所保留的最小字符个数。如果有小数位，则小数点占一位；如果实际宽度不足，则按照规则填补空格或0；如果实际宽度超过指定的位数，则按照实际宽度存储或显示。精度（对于数字来说）则是结果中应该包含的小数点后面的位数。例如：

```
>>> a = 3.1416
>>> '%6.2f' % a            #总宽度6位,保留2位小数,小数点占1位,不足部分左侧补2个空格
' 3.14'
>>> '%f' % 3.1416          #浮点数默认保留6位小数
'3.141600'
>>> '%.2f' % 3.1416        #指定保留2位小数
'3.14'
>>> '%07.2f' % 3.1416      #宽度7位,保留2位小数,默认右对齐,左侧空位填0
'0003.14'
>>> '%+07.2f' % 3.1416     #正数加正号,正号占1位,空位填0
'+003.14'
>>> '%-7.2f' % -3.1416     #符号"-"表示左对齐,右侧2个空位填空格
'-3.14 '
>>> '%2.1f' % 3.1416       #指定宽度小于实际宽度时,按照实际宽度
'3.1'
>>>
```

最小宽度和精度之间不能有空格，格式字符和其他选项之间也不能有空格，如%8.2f。

格式代码中的格式字符要与格式运算符后面的对象类型匹配。如果两者类型不匹配，一般会引发程序异常。当格式字符为s时，格式运算符后面对应的对象可以是任何类型。例如：

```
>>> '%s' % 5
'5'
>>> "%s" % 1.1
'1.1'
>>> "%s" % [1,2]
'[1, 2]'
>>>
```

可以一次格式化多个对象。此时需要将这些对象表示成一个元组的形式。这些对象在元组中的位置与格式化字符的位置要一一对应。例如：

```
>>> '%.2f# %4d, %s' % (3.456727,89,'Lily')
'3.46# 89,Lily'
>>>
```

上述代码中，%.2f表示3.456727的格式形式，%4d表示89的格式形式，%s表示'Lily'的格式形式，字符串里面的井号（#）和逗号（,）均按原样、原位置输出。

**【例5-1】** 利用字符串格式化方式输出如图5.2所示的“九九乘法表”。

程序源代码如下。

```
#example5_1.py
#coding = utf-8
for i in range(1,10):
    for j in range(1,i+1):
```

```
1*1=1
1*2=2   2*2=4
1*3=3   2*3=6   3*3=9
1*4=4   2*4=8   3*4=12  4*4=16
1*5=5   2*5=10  3*5=15  4*5=20  5*5=25
1*6=6   2*6=12  3*6=18  4*6=24  5*6=30  6*6=36
1*7=7   2*7=14  3*7=21  4*7=28  5*7=35  6*7=42  7*7=49
1*8=8   2*8=16  3*8=24  4*8=32  5*8=40  6*8=48  7*8=56  8*8=64
1*9=9   2*9=18  3*9=27  4*9=36  5*9=45  6*9=54  7*9=63  8*9=72  9*9=81
```

图 5.2　九九乘法表

```
        print("%d*%d=%-4d"%(j,i,i*j),end="")
    print()
```

在例 5-1 中，乘数和被乘数均占一个字符宽度输出；乘积占 4 个字符宽度输出且左对齐。

## 5.3.2　用 format()方法格式化字符串

字符串的 format()方法用花括号和冒号来代替传统的%方式表示字符串格式的定义，一般形式如图 5.3 所示。

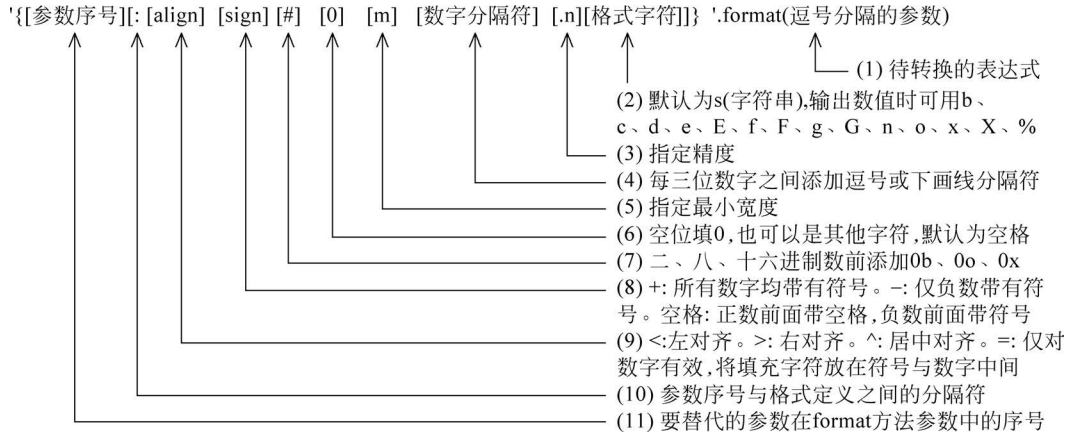


图 5.3　format()方法的一般形式

在一个字符串中可以有多个花括号{和}括起来的格式定义与占位符。format()方法中的大部分格式字符与传统的利用%进行格式化的格式字符相同。格式符 n 与 g 的功能相同，插入随区域而异的数字分隔符。格式符%表示将数字表示为百分数，也就是将参数值乘以 100，然后在后面加上百分号。

### 1. format()方法中的参数传递方式

1）根据位置顺序传递参数

字符串使用 format()方法格式化时，可以根据位置顺序来传递参数。例如：

```
>>> "我叫{},今年{}岁;他叫{},今年{}岁".format("张三", 18, "李四", 20)
'我叫张三,今年 18 岁;他叫李四,今年 20 岁'
>>>
```

上述例子中，分别将 format()参数中的"张三"、18、"李四"和 20 这 4 个值按照位置顺序依次传递到字符串的 4 对花括号中。

2）根据位置索引值传递参数

字符串使用 format()方法格式化时，也可以通过索引值来引用 format()参数中对应位置上的值，只要 format()方法相应位置上有参数值即可，参数索引从 0 开始。例如：

```
>>> "我叫{0},今年{1}岁;他叫{2},今年{3}岁".format("张三", 18, "李四", 20)
'我叫张三,今年 18 岁;他叫李四,今年 20 岁'
>>> "我叫{2},今年{3}岁;他叫{0},今年{1}岁".format("张三", 18, "李四", 20)
'我叫李四,今年 20 岁;他叫张三,今年 18 岁'
>>>
```

上述例子中，花括号内的数字 0～3 分别表示 format()中相应位置上的参数。

3）根据位置引用序列

字符串 format()方法中的参数也可以是序列，通过参数的位置索引表示相应的序列，并用序列中的元素索引来引用相应的元素值。例如：

```
>>> zhang = ("张三", 18)
>>> li = ("李四", 20)
>>> "我叫{0[0]},今年{0[1]}岁;他叫{1[0]},今年{1[1]}岁".format(zhang,li)
'我叫张三,今年 18 岁;他叫李四,今年 20 岁'
>>>
```

上述代码中，0[0]的第一个 0 表示 format(zhang,li)中的第 0 个参数（也就是 zhang），0[0]的方括号内的 0 表示 zhang 中的第 0 个元素（也就是字符串'张三'）。0[1]的 0 表示 format(zhang,li)中的第 0 个参数（也就是 zhang），方括号内的 1 表示 zhang 中的第 1 个元素（也就是 18）。1[0]中的 1 表示 format(zhang,li)中的第 1 个参数（也就是 li），方括号中的 0 表示 li 中的第 0 个元素（也就是"李四"）。1[1]中的第一个 1 表示 format(zhang,li)中的第 1 个参数（也就是 li），方括号中的 1 表示 li 中的第一个元素（也就是 20）。

4）展开序列后按位置或索引传递参数

字符串使用 format()方法格式化时，也可以用“*序列名称”的形式作为 format()方法的实际参数。作为实际参数的序列名称前面加星号（*）表示将序列展开，然后通过位置依次将展开后的元素传递到目标字符串中。例如：

```
>>> zhang = ("张三", 18)
>>> li = ("李四", 20)
>>> "我叫{},今年{}岁;他叫{},今年{}岁".format( * zhang, * li)
'我叫张三,今年 18 岁;他叫李四,今年 20 岁'
>>>
```

上述例子从本质上来说，先把 format(*zhang，*li)中的 *zhang 和 *li 分别展开为"张三",18 和"李四",20，然后执行"我叫{},今年{}岁；他叫{},今年{}岁".format("张三",18,"李四",20)。该例子也可以改写如下。

```
>>> "我叫{0},今年{1}岁;他叫{2},今年{3}岁".format( * zhang, * li)
'我叫张三,今年 18 岁;他叫李四,今年 20 岁'
>>>
```

5）以关键参数形式传递参数

字符串使用 format()方法格式化时，也可以使用关键参数的形式（变量名＝值）为 format()方法传递实际参数。例如：

```
>>> "我叫{zname},今年{zage}岁".format(zname = "张三", zage = 18)
```

```
'我叫张三,今年 18 岁'
>>> zhang = ("张三", 18)
>>> "我叫{z[0]},今年{z[1]}岁".format(z = zhang)
'我叫张三,今年 18 岁'
>>> name = '张三'
>>> age = 18
>>> '我叫{n},今年{a}岁'.format(n = name,a = age)
'我叫张三,今年 18 岁'
>>>
```

但不可以采用以下方式。

```
>>> '我叫{name},今年{age}岁'.format(name,age)      #不能这样写
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    '我叫{name},今年{age}岁'.format(name,age)
KeyError: 'name'
>>>
```

6）展开字典中的元素按关键参数形式传递

字符串使用 format()方法格式化时，也可用“** 字典名”的形式为 format()方法传递实际参数。作为实际参数的字典名称前面加两个星号（**）将字典中的元素依次展开为“键＝值”的关键参数形式。例如：

```
>>> zhang = {"name":"张三", "age":18}
>>> "我叫{name},今年{age}岁".format( ** zhang)
'我叫张三,今年 18 岁'
>>>
```

上述例子中，先将 format( ** zhang)中的字典参数“** zhang”展开为：name＝"张三"，age＝18，得到 format(name＝"张三"，age＝18)，然后按照关键参数分别将 name 和 age 的值传递到相应的位置上。

位置参数与关键参数传递方式、序列前面加一个星号的参数传递方式、字典前面加两个星号的参数传递方式将在第 6 章函数的设计部分详细介绍。

**2. 格式的设置**

用字符串 format()方法格式化字符串时，如果需要设置格式，则按照图 5.3 中的模式在冒号后面设置格式符。其中，冒号之前的部分表示 format()中的参数名称或位置索引，冒号之后的部分表示格式符。例如：

```
>>> "我的成绩为{0:>3}分,超过了全班{1:.2%}的同学".format(95,2/3)
'我的成绩为 95 分,超过了全班 66.67% 的同学'
>>>
```

上述案例中，{0：＞3}中的 0 表示 format()中的第 0 个参数，大于号表示右对齐，3 表示占 3 个字符的位置，不足部分左侧补空格；{1：.2%}表示取 format()中的第 1 个参数，以百分数形式呈现，保留 2 位小数。

如果数字较长，为了方便识别，可以在数字中每隔 3 位插入一个逗号或下画线作为分隔符。例如：

```
>>> "对数字每 3 位加一个逗号{:,}".format(1234567890)
'对数字每 3 位加一个逗号 1,234,567,890'
>>> "也可以用下画线分隔数字{0:_}".format(1234567890)
```

```
'也可以用下画线分隔数字 1_234_567_890'
>>>
```

**3. 日期和时间的格式化**

字符串的 format()方法还可以用来格式化日期和时间。例如：

```
>>> import datetime
>>> date_now = datetime.datetime.now()
>>> date_now
datetime.datetime(2023, 9, 20, 14, 28, 43, 235265)
>>> "当前日期为{:%Y-%m-%d}".format(date_now)
'当前日期为 2023-09-20'
>>> "当前时间为{:%Y-%m-%d %H:%M:%S}".format(date_now)
'当前时间为 2023-09-20 14:28:43'
>>>
```

## 5.3.3 用 format_map()方法格式化字符串

使用 format_map(mapping)方法格式化字符串与使用 format()方法格式化字符串的用法类似，都使用一对花括号来表示占位符的边界。不同之处在于 format_map(mapping)方法使用 mapping 对象对字符串进行格式化。该 mapping 参数对象是一个包含要替换的字符串中占位符键值对的字典对象。例如：

```
>>> zhang = {"name":"张三", "age":18}
>>> "我叫{name},今年{age}岁".format_map(zhang)
'我叫张三,今年 18 岁'
>>>
```

参数字典中，键值对的数量可以多于格式化占位符中需要的键值对数量。例如：

```
>>> zhang = {'age': 18, 'isStu': True, 'name': '张三'}
>>> "我叫{name},今年{age}岁".format_map(zhang)
'我叫张三,今年 18 岁'
>>>
```

一个类的对象通过__dict__属性可以获取以该对象中的实例属性和对应的值构成的键值对为元素的字典，然后利用字符串的 format_map()方法将该字典映射到字符串格式化的占位符上。例如：

```
>>> class Person:                          # 自定义 Person 类
...     def __init__(self, name, age, isStu):
...         self.name = name
...         self.age = age
...         self.isStu = isStu
...
>>> zhang = Person("张三", 18, True)       # 创建 Person 类的一个对象 zhang
>>> zhang.__dict__                         # 查看 zhang 的__dict__属性
{'age': 18, 'isStu': True, 'name': '张三'}
>>> "我叫{name},今年{age}岁".format_map(zhang.__dict__)
'我叫张三,今年 18 岁'
>>>
```

类与对象及属性等概念将在第 7 章进行介绍，读者可以学完第 7 章后再阅读自定义类的相关案例。

### 5.3.4　用 f-strings 字面量方法格式化字符串

如果一个字符串前面带有 f 或 F 字符，则字符串中可以含有表达式，该表达式需要用花括号括起来。将计算完的花括号内的表达式结果转换为字符串，替换到该花括号及其内部表达式所在的位置，生成一个格式化的字符串对象。这种方法称为字符串格式化的字面量方法(f-strings)。字面量格式化方式类似于字符串的 format()方法，使用起来更加灵活、方便。推荐使用此方式进行字符串的格式化。

f-strings 采用 {content:format} 设置字符串格式。content 表示要替换并填入字符串的内容，可以是变量、表达式、函数、lambda 表达式等。format 是格式描述符，与字符串 format()方法中的格式描述符相同。采用默认格式时不必指定格式描述符，只需要{content}即可。例如：

```
>>> weight = 60
>>> height = 1.72
>>> f"我的体重{weight}千克, 身高{height:> 5.2f}米"
'我的体重 60 千克, 身高 1.72 米'
>>> s = f"我的 BMI 值为{weight/(height ** 2):> 6.2f}"
>>> s
'我的 BMI 值为 20.28'
>>>
```

格式描述符的详细用法可以参考 Python 官方文档(进入网址为 https://docs.python.org 的 Documentation 页面→Library Reference→Text Processing Services→Format String Syntax)。

花括号中可以放入任何表达式。例如：

```
>>> s = f"姓名:{input('请输入姓名:')} 学号:{int(input('请输入学号:'))}"
请输入姓名:杨
请输入学号:2020055001
>>> s
'姓名:杨 学号:2020055001'
>>>
```

**【例 5-2】** 编写程序，输入一个字符串，分别统计大写字母、小写字母、数字以及其他字符的个数，并分别以百分号占位符、format()方法和 f-strings 方法格式化字符串的方式打印输出各种字符个数。数字仅包括阿拉伯数字。

程序源代码如下。

```
#example5_2.py
#coding = utf - 8
s = input('请输入一个字符串:')
c1,c2,c3,c4 = 0,0,0,0
for i in s:
    if "A"<= i <= "Z":
        c1 += 1
    elif "a"<= i <= "z":
        c2 += 1
    elif "0"<= i <= "9":
        c3 += 1
    else:
```

```
            c4 += 1
print("大写字母 %d 个;小写字母 %d 个;数字 %d 个;其他字符 %d 个。" %(c1,c2,c3,c4))
print("大写字母{0}个;小写字母{1}个;数字{2}个;其他字符{3}个。".format(c1,c2,c3,c4))
print(f"大写字母{c1}个;小写字母{c2}个;数字{c3}个;其他字符{c4}个。")
```

程序 example5_2.py 的一次运行结果：

```
请输入一个字符串:学生 I am a student since 2000
大写字母 1 个;小写字母 15 个;数字 4 个;其他字符 8 个。
大写字母 1 个;小写字母 15 个;数字 4 个;其他字符 8 个。
大写字母 1 个;小写字母 15 个;数字 4 个;其他字符 8 个。
```

思考题 5-1：程序 example5_2.py 如何改用字典来累计并保存各类字符的统计值，并分别用 format()和 format_map()方法格式化字符串？

## 5.4 字符串常用方法

本节介绍字符串对象的一些常用方法。由于字符串属于不可变序列类型，常用方法中涉及返回字符串的都是新字符串，原有字符串对象保持不变。

### 5.4.1 英文字母大小写转换

字符串对象中包含进行英文字母大小写转换的方法。表 5.5 给出了各种应用场景下的大小写转换方法。

**表 5.5 字符串的大小写转换方法**

| 方 法 | 功 能 |
|---|---|
| lower() | 将字符串中大写字母转换为小写字母，其他字符不变，并返回新字符串 |
| upper() | 将字符串中小写字母转换为大写字母，其他字符不变，并返回新字符串 |
| swapcase() | 将字符串中字符的大小写互换 |
| capitalize() | 将字符串首字母转换为大写形式，其他字母转换为小写形式 |
| title() | 将字符串中每个单词的首字母转换为大写形式，其他字母转换为小写形式 |

以下给出了字符串中英文字母大小写转换的应用案例。

```
>>> s = "Python 程序设计语言。"
>>> s1 = s.lower()
>>> s1
'python 程序设计语言。'
>>> s          #原字符串保持不变
'Python 程序设计语言。'
>>> s.upper()
'PYTHON 程序设计语言。'
>>> s.swapcase()
'pYTHON 程序设计语言。'
>>>
>>> s = "python programming language."
>>> s.capitalize()
'Python programming language.'
>>> s.title()
'Python Programming Language.'
>>>
```

在编写程序时，经常用字符串的 lower()或 upper()方法比较不区分大小写的字符串。

【例 5-3】　用户从键盘依次输入若干个字符串组成一个列表 list1。每输完一个字符串加入列表后，询问是否结束输入；如果此时输入'y'或者'yes'(不区分大小写)，则结束输入。打印输出 list1 的内容。

程序源代码如下。

```
#example5_3.py
#coding = utf - 8
print("输入若干个字符串组成列表 list1。")
ans = 'n'
i = 1
list1 = []                                    #初始化一个空列表
while ans.upper() not in ['Y','YES'] :        #判断是否结束
    x = input("请输入第" + str(i) + "个字符串:")
    list1.append(x)
    i += 1
    ans = input("结束输入吗?(不区分大小写的 y 或 yes 表示结束,其他表示继续):")
print("列表 list1:",list1)
```

程序 example5_3.py 可能的一次运行结果如下。

```
输入若干个字符串组成列表 list1.
请输入第 1 个字符串:学习
结束输入吗?(不区分大小写的 y 或 yes 表示结束,其他表示继续):n
请输入第 2 个字符串:Python
结束输入吗?(不区分大小写的 y 或 yes 表示结束,其他表示继续):no
请输入第 3 个字符串:程序设计
结束输入吗?(不区分大小写的 y 或 yes 表示结束,其他表示继续):Y
列表 list1: ['学习', 'Python', '程序设计']
```

## 5.4.2　判断字符串中的字符元素特点

表 5.6 列出了判断一个字符串中字符元素特点的常用方法。

**表 5.6　判断字符串中字符元素特点的方法**

| 方　法 | 功　能 |
|---|---|
| islower() | 如果是小写字符串则返回 True，否则返回 False |
| isupper() | 如果是大写字符串则返回 True，否则返回 False |
| istitle() | 如果是标题大小写字符串则返回 True，否则返回 False |
| isprintable() | 如果字符串是可打印的返回 True，否则返回 False |
| isspace() | 如果是空白字符串则返回 True，否则返回 False |
| isascii() | 如果字符串中的所有字符都是 ASCII 字符则返回 True，否则返回 False。空字符串也是 ASCII 字符串 |
| isalnum() | 如果字符串仅由字母或数字字符构成则返回 True，否则返回 False |
| isalpha() | 如果字符串仅由字母构成则返回 True，否则返回 False |
| isdecimal() | 如果字符串是十进制字符串则返回 True，否则返回 False。如果字符串中的所有字符都是十进制，并且字符串中至少有一个字符，则该字符串是十进制字符串。不包括罗马数字、汉字数字等 |
| isdigit() | 如果字符串是数字字符串则返回 True，否则返回 False。如果字符串中的所有字符都是数字，并且至少包含一个字符，则该字符串为数字字符串。不包括罗马数字、汉字数字等 |

续表

| 方　法 | 功　能 |
| --- | --- |
| isnumeric() | 如果字符串是数字字符串则返回 True，否则返回 False。如果字符串中的所有字符都是数字，并且至少包含一个字符，则该字符串为数字字符串。包括罗马数字、汉字数字等 |
| isidentifier() | 如果字符串是有效的 Python 标识符则返回 True，否则返回 False |

**【例 5-4】** 输入一个字符串，利用 isupper()、islower()、isdigit()分别统计英文大写字母、英文小写字母、数字及其他字符的个数。用字符串格式化方式分别显示各类字符的个数。

程序源代码如下。

```
#example5_4.py
#coding = utf - 8
s = input('请输入一个字符串:')
c1,c2,c3,c4 = 0,0,0,0
for i in s:
    if i.isupper():
        c1 += 1
    elif i.islower():
        c2 += 1
    elif i.isdigit():
        c3 += 1
    else:
        c4 += 1
print(f"英文大写字母{c1}个;英文小写字母{c2}个;" +
      f"数字{c3}个;其他字符{c4}个。")
```

程序 example5_4.py 的一次运行结果如下。

```
请输入一个字符串:Since 2000, I am a student.
英文大写字母 2 个;英文小写字母 14 个;数字 4 个;其他字符 7 个。
```

### 5.4.3 子串的查找与统计

在字符串中可以查找并定位某子串出现的位置、统计某子串出现的次数。表 5.7 给出了子串查找与统计的常用方法。

**表 5.7　子串的查找与统计方法**

| 方　法 | 功　能 |
| --- | --- |
| find()与 rfind() | s.find(sub[,start[,end]])和 s.rfind(sub[,start[,end]])在一个较长的字符串 s 中，在[start,end)范围内查找并返回子串 sub 首次出现的位置索引，如果没有找到则返回－1。查找范围包括开始值 start 的位置，不包括结束值 end 的位置。默认范围是整个字符串。find()方法从左往右查找，rfind()方法从右往左查找 |
| index()与 rindex() | s.index(sub[,start[,end]])和 s.rindex(sub[,start[,end]])方法在一个较长的字符串 s 中，查找并返回在[start,end)范围内子串 sub 首次出现的位置索引，如果不存在则抛出异常。默认范围是整个字符串。其中，index()方法从左往右查找，rindex()方法从右往左查找 |
| count() | s.count(sub[,start[,end]])方法在一个较长的字符串 s 中，查找并返回[start,end)范围内子串 sub 出现的次数，如果不存在则返回 0。默认范围是整个字符串 |

字符串的 find()方法与 index()方法类似，均可以从前往后寻找子串首次出现的位置。

区别是 find()方法没有找到子串时返回－1，而 index()方法没有找到子串时将返回程序的异常信息。字符串的 rfind()和 rindex()方法都是从后往前查找子串首次出现的位置。区别是 rfind()方法没有找到子串时返回－1，rindex()方法没有找到子串时返回程序的异常信息。例如：

```
>>> s = "learn programming for deep learning"
>>> s.find("learn")          #在整个字符串范围内查找
0
>>> s.index("learn")
0
>>>
>>> s.find("Learn")          #不存在该子串,返回 - 1
 - 1
>>> s.index("Learn")
Traceback (most recent call last):
  File "< interactive input >", line 1, in < module >
ValueError: substring not found
>>>
>>> s.find("learn",6)        #从 index 为 6 的位置开始查找
27
>>> s.index("learn",6)
27
>>>
>>> s.find("learn",6,15)     #在区间[6, 15)范围内查找
 - 1
>>> s.index("learn",6,15)
Traceback (most recent call last):
  File "< interactive input >", line 1, in < module >
ValueError: substring not found
>>>
>>> s.rfind("learn")         #从右侧开始查找
27
>>> s.rindex("learn")
27
>>>
>>> s.rfind("learn",0,10)    #在区间[0, 10)范围内,从右侧开始查找
0
>>> s.rindex("learn",0,10)
0
>>>
```

字符串的 count()方法统计子串出现的次数。例如：

```
>>> s = "learn programming for deep learning"
>>> s.count("learn")
2
>>> s.count("Learn")
0
>>>
```

### 5.4.4　分割字符串

分割字符串的常用方法如表 5.8 所示。

表 5.8 分割字符串的方法

| 方　法 | 功　能 |
| --- | --- |
| split() | split(sep=None,maxsplit=-1)以 sep 指定字符为分隔符,从左往右将字符串分割开来,并将分割后的子串组成列表返回。当 sep 设置为 None(默认值)时,将以任何空白字符(包括\n \r \t \f 和空格)为分隔符进行分割,并在结果中丢弃空字符串。参数 maxsplit 表示最大分割次数;默认为-1,表示分割次数没有限制,只要出现分隔符 sep 就进行分割 |
| rsplit() | rsplit(sep=None,maxsplit=-1) 以 sep 指定字符为分隔符,从右往左将字符串分割开来,并将分割后的子串组成列表返回。当 sep 设置为 None(默认值)时,将以任何空白字符(包括\n \r \t \f 和空格)为分隔符进行分割,并在结果中丢弃空字符串。参数 maxsplit 表示最大分割次数;默认为-1,表示分割次数没有限制,只要出现分隔符 sep 就进行分割 |
| splitlines() | splitlines(keepends=False)方法按照行分隔符(也就是换行符'\r'、'\r\n'或\n')来分割字符串,返回以子串为元素的列表。参数 keepends 默认为 False,表示每行构成的字符串不保留换行符;如果设置为 True,结果中将保留换行符 |
| partition() | 使用给定的分隔符将字符串划分为三部分。如果在字符串中找到分隔符,则返回一个 3 元组,包含分隔符之前的部分、分隔符本身和它之后的部分。如果在字符串中未找到分隔符,则返回包含原始字符串和两个空字符串的 3 元组 |
| rpartition() | 使用给定的分隔符将字符串划分为三部分。从字符串的末尾开始搜索分隔符。如果找到分隔符,则返回一个 3 元组,其中包含分隔符之前的部分、分隔符本身以及它后面的部分。如果没有找到分隔符,则返回一个包含两个空字符串和原始字符串的 3 元组 |

字符串的 split()方法和 rsplit()方法用法类似。下面以 split()为例简要列举使用方法。

```
>>> s = "Python programming language"
>>> s.split()              #默认通过空白符分割,空格是一种空白符
['Python', 'programming', 'language']
>>> s.split(" ")           #以空格分割
['Python', 'programming', 'language']
>>> s = "Python,programming,language"
>>> s.split(",")           #通过逗号分割
['Python', 'programming', 'language']
>>> s.split(",",1)         #最多分割 1 次
['Python', 'programming,language']
>>> s.split(";")           #通过分号";"分割;s 中没出现分号,s 整体作为列表的单一元素
['Python,programming,language']
>>>
```

对于 split(),如果不指定分隔符,实际上表示以任何空白字符(包括连续出现的)作为分隔符。空白字符包括空格、换行符、制表符等。可以通过 string 模块中的 whitespace 常量来查看所有表示空白的字符,例如:

```
>>> import string
>>> string.whitespace
'\t\n\r\x0b\x0c'
>>>
```

**【例 5-5】** 编写程序,从键盘输入一周的产品销售额(单位:元),销售金额之间用逗号分割,输出每天的销售金额和平均销售金额,保留 2 位小数。

第 1 种方法的程序代码:

```
#example5_5_1.py
#coding = utf - 8
s = input("请输入最近一周的销售金额,用逗号分隔:")
sale_list = s.split(",")
ss = 0
print("各天的销售金额为:",end = "")
for i in range(len(sale_list)):
    if i < len(sale_list) - 1:
        print(sale_list[i],end = "元,")
    else:
        print(sale_list[i],end = "元\n")

    ss += float(sale_list[i])

avg = ss/len(sale_list)
print("平均销售额:{:.2f}".format(avg))
```

程序 example5_5_1.py 的一次运行结果：

```
请输入最近一周的销售金额,用逗号分割:80,68,75,90,70,86,95
各天的销售金额为:80 元,68 元,75 元,90 元,70 元,86 元,95 元
平均销售额:80.57
```

第 2 种方法的程序代码：

```
#example5_5_2.py
#coding = utf - 8
s = input("请输入最近一周的销售金额,用逗号分隔:")
sale_list = s.split(",")

print("各天的销售金额为:",end = "")
for i in range(len(sale_list)):
    if i < len(sale_list) - 1:
        print(sale_list[i],end = "元,")
    else:
        print(sale_list[i],end = "元\n")

sale_list = list(map(float,sale_list))      #构建元素为 float 类型的列表
avg = sum(sale_list)/len(sale_list)
print(f"平均销售额:{avg:.2f}")
```

读取一个文本文件后，通常使用 splitlines(keepends＝False)方法，按照行分隔符(也就是换行符'\r'、'\r\n'或'\n')，将每行的内容切分成单独的字符串，返回以子串为元素的列表。参数 keepends 默认为 False，表示每行构成的字符串不保留换行符；如果设置为 True，结果中将保留换行符。例如：

```
>>> s = "Python\r 程序\r\n 设计\n 语 言"
>>> s.splitlines()
['Python', '程序', '设计', '语 言']
>>> s.splitlines(keepends = True)
['Python\r', '程序\r\n', '设计\n', '语 言']
>>>
```

## 5.4.5　用 join()连接可迭代对象中的元素

字符串的 join()方法可用来连接可迭代对象中的元素，并在两个元素之间插入调用

join()方法的字符串，返回一个字符串。调用格式为：间隔字符串对象.join(可迭代对象)。例如：

```
>>> s = "programming language"                    #字符串序列中,每个字符为一个元素
>>> "#".join(s)
'p#r#o#g#r#a#m#m#i#n#g# #l#a#n#g#u#a#g#e'
>>> s = ("Python","programming","language")       #每个字符串为一个元素
>>> "-".join(s)
'Python-programming-language'
>>> "###".join(s)                                 #多个字符构成的字符串作为间隔
'Python###programming###language'
>>>
```

join()方法是 split()方法的逆方法。例如：

```
>>> s = "Python programming language"
>>> split_list = s.split()
>>> split_list
['Python', 'programming', 'language']
>>> " ".join(split_list)
'Python programming language'
>>>
```

### 5.4.6 子串与字符替换

#### 1. replace()

replace(old,new,count=-1)方法用于查找字符串中的 old 子串并用 new 子串来替换。参数 count 默认值为-1，表示替换所有匹配项，否则最多替换 count 次。返回替换后的新字符串，原字符串保持不变。例如：

```
>>> s = "学习 Python,learning Python programming language."
>>> s.replace("Python","Java")          #替换所有匹配项
'学习 Java,learning Java programming language.'
>>> s.replace("Python","Java",1)        #最多替换 1 次
'学习 Java,learning Python programming language.'
>>> s.replace("python","Java")          #找不到匹配项,不做替换
'学习 Python,learning Python programming language.'
>>>
```

#### 2. translate()与 maketrans()

字符串的 translate(table)方法使用给定的转换表 table 替换字符串中的每个字符。转换表可以自己定义或使用 str 中的静态方法 maketrans()来定义。

1）自定义转换表

自定义转换表必须是从 Unicode 序号到 Unicode 序号、字符串或 None 映射的字典。使用 translate()方法时，如果字符串中字符的 Unicode 序号出现在 key 中，则该字符用字典中对应 key 的相应字符来进行替换。例如：

```
>>> s = "python programming"
>>> table1 = {112:80, 111:79}
>>> s.translate(table1)
'PythOn PrOgramming'
>>>
>>> table2 = {112:"P", 111:"O"}
```

```
>>> s.translate(table2)
'PythOn PrOgramming'
>>>
```

注意，上述例子中转换表 table1 和 table2 定义时必须以字符的 Unicode 编号作为字典的 key。

2）用 str.maketrans()来创建转换表

字符串的静态方法 maketrans()创建可用于 translate()的转换表。如果方法 maketrans()只有一个参数，它必须是一个将 Unicode 序号（整数）或字符映射为 Unicode 序号、字符串或 None 的字典。如果字典中的键是字符，该方法自动将字符键转换为序号。如果 maketrans()有两个参数，这两个参数必须是长度相等的字符串，并且在结果字典中，前一个参数字符串中的每个字符将被映射到后一个参数字符串中相同位置的字符。如果 maketrans()有第三个参数，它必须是一个字符串，这个字符串中的字符将在结果中映射为 None。例如：

```
>>> s = "python programming"
>>> table3 = str.maketrans({"p":"P", "o":"O"})
>>> table3
{111: 'O', 112: 'P'}
>>> s.translate(table3)
'PythOn PrOgramming'
>>>
>>> table4 = str.maketrans({112:80, 111:79})
>>> table4
{111: 79, 112: 80}
>>> s.translate(table4)
'PythOn PrOgramming'
>>>
>>> table5 = str.maketrans({111: 'O', 112: 'P'})
>>> table5
{111: 'O', 112: 'P'}
>>> s.translate(table5)
'PythOn PrOgramming'
>>>
>>> table6 = str.maketrans("opq","OPQ","!")
>>> table6
{33: None, 111: 79, 112: 80, 113: 81}
>>> "python programming!".translate(table6)
'PythOn PrOgramming'
>>>
```

## 5.4.7　去除首尾子串

表 5.9 给出了字符串对象中去除首尾特定子串或首尾空白字符的方法。

**表 5.9　去除字符串中首位特定字串或空白符的方法**

| 方　法 | 功　能 |
|---|---|
| removeprefix() | 如果参数中给定的前缀字符串存在，则删除该前缀，返回一个新字符串 |
| removesuffix() | 如果参数中给定的后缀字符串存在，则删除该后缀，返回一个新字符串 |
| strip() | 去除字符串两侧的空白字符或指定字符序列中的字符，返回一个新字符串 |
| lstrip() | 去除字符串左侧的空白字符或指定字符序列中的字符，返回一个新字符串 |
| rstrip() | 去除字符串右侧的空白字符或指定字符序列中的字符，返回一个新字符串 |

字符串的 strip()、lstrip()和 rstrip()分别用于去除字符串中两端、左侧和右侧的空白字符，并返回一个新字符串。例如：

```
>>> s = " \n Python and Java, Java and Python \t \n"
>>> s.strip()      # 没有指定字符参数,默认去除 s 两端的空白字符
'Python and Java, Java and Python'
>>>
```

可以从两端逐一去除指定字符序列中的字符，直到不是该序列中的字符为止。例如：

```
>>> s = "Pythonon and Java, Python, Java and Python"
>>> s.strip("Python")
' and Java, Python, Java and '
>>>
```

字符串的方法 lstrip()和 rstrip()分别从左侧和右侧去除相应的字符。例如：

```
>>> s.lstrip("Python")
' and Java, Python, Java and Python'
>>> s.rstrip("Python")
'Pythonon and Java, Python, Java and '
>>>
```

如何去除字符串中所有的空格？可以使用字符串的 replace()方法。例如：

```
>>> s = " Python programming language "
>>> s.replace(" ", "")
'Pythonprogramminglanguage'
>>>
```

空白字符的概念和内容请参看 5.4.4 节。

### 5.4.8 判断是否以特定子串开始或结束

字符串的 startswith()方法用于判断某位置上的子串是否以特定的前缀开始。调用格式为 S.startswith(prefix[,start[,end]])，如果字符串 S 以指定的前缀 prefix 开始，则返回 True，否则返回 False。默认整个字符串 S 参与比较。如果带有可选参数 start，表示从字符串 S 的 start 位置开始比较。如果带有可选参数 end，则在字符串 S 的 end 位置停止比较。start 位置上的元素参与比较，但 end 位置上的元素不参与比较。例如：

```
>>> s = "python programming"
>>> s.startswith("py")
True
>>> s.startswith("pro")
False
>>> s.startswith("pro",7)
True
>>> s.startswith("pro",7,9)
False
>>> s.startswith("pro",7,10)
True
>>>
```

前缀也可以是一个以字符串为元素的元组。例如：

```
>>> s.startswith(("py","pro"))
True
```

```
>>> s.startswith(("py","pro"),7,10)
True
>>>
```

字符串的 endswith()用于判断某位置上的子串是否以特定的后缀结束。其调用格式为 S.endswith(suffix[,start[,end]]),参数 suffix 表示待搜索的后缀,其余参数的含义与 startswith()方法中的同名参数含义相同。后缀也可以是一个以字符串为元素的元组。这里不对 endswith()方法展开阐述。

## 习题

1. 从键盘输入圆的半径,计算圆的面积和周长,分别用本章介绍的 4 种字符串格式化方法在屏幕上输出圆的面积和周长,各保留 2 位小数,圆周率用 math 模块中定义的 pi 值。程序保存为 exercise5_1.py。

2. 从键盘输入一个英文句子,根据空白符将该字符串分割为多个子串构成的列表。去除列表中各子串中的英文标点符号。统计输入的句子中各单词出现的次数。程序保存为 exercise5_2.py。提示:string 模块中的 punctuation 表示英文标点符号构成的字符串。

3. 生成一个包括 10 个字符的随机密码,密码中的字符只能是英文大小写字母、数字、“@”或“_”。程序保存为 exercise5_3.py。