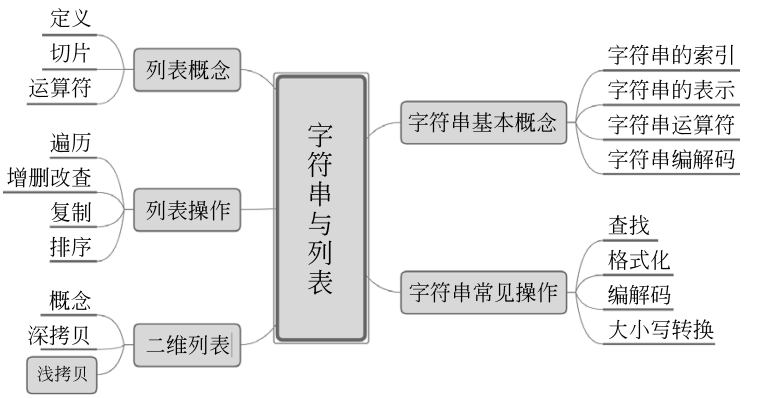
第3章

字符串与列表

3.1 导　　学



学习目标：

- 理解字符串在程序设计中的作用。
- 掌握字符串的常见操作。
- 熟悉使用字符串要注意的常见问题。
- 掌握输入和输出语句的基本用法。
- 掌握列表的创建方式。
- 掌握列表的各种常见操作。
- 理解字符串和列表之间的关系。
- 重点掌握列表的查找、排序等算法。
- 了解二维列表的作用和定义方式。
- 了解深拷贝和浅拷贝的区别。

文本是程序需要处理的最常见的数据形式。第 2 章已经简单介绍过字符串和列表，实际上字符串和列表能做的事情很多。使用 Python 可以从字符串中提取部分字符串，添加或删除空白字符，将字母转换成小写或大写，检查字符串的格式是否正确。甚至可以

编写 Python 代码访问剪贴板，复制或粘贴文本。

除了字符串之外，还有一个非常重要的数据结构类型是必须要熟练掌握的，那就是列表数据类型。列表可以包含多个值，这样编写程序来处理大量数据就变得更容易。而且，由于列表本身又可以包含其他列表，所以可以用它们将数据安排成层次结构。

实际上，字符串本质上就是一个元素是字符的列表，列表的多数操作可以同时使用在字符串上。本章将探讨字符串和列表的相关知识，讨论字符串和列表中的各种常见操作，了解常见的字符串和列表常见处理函数，并在最后的实例中使用这些知识。

3.2 字符串概述

3.2.1 字符串的表示

在 Python 中表示一个字符串值是相当简单的：它们以单引号开始和结束。示例如下：

```
s='Hello'
```

但是如果字符串内包含单引号呢？示例如下：

```
s='That is Tom's cat'
```

程序会给出如下错误信息：

```
s='That is Tom's cat'
                    ^
SyntaxError: invalid syntax
```

因为 Python 认为这个字符串在 Tom 之后就结束了，剩下的(s cat')是无效的 Python 代码。好在，还有其他办法来输入字符串。

另一种方式是使用双引号。使用双引号的一个好处，就是字符串中可以使用单引号字符。示例如下：

```
s="That is Tom's cat"
print(s)
```

因为字符串以双引号开始，所以 Python 知道单引号是字符串的一部分，而不是表示字符串的结束。但是，如果在字符串中既需要使用单引号又需要使用双引号，那就要使用转义字符。

注意，虽然有多种字符串的表示方法，但不可混用，如不能用单引号作为字符串开始，用双引号作为结束。

3.2.2 字符串的索引

可以使用下标来访问字符串中的每个字符，示例如下：

```
s="Hello world!"
print(len(s))
print(s[1])
```

运行结果：

```
12
e
```

从结果可知，字符计数包含了空格和感叹号，所以'Hello world! '有 12 个字符，H 的下标是 0，!的下标是 11。

3.2.3 转义字符

程序设计语言中的一些符号承担了专用功能，例如引号用于标识字符串，如果字符串中需要出现引号并把这些功能符号当作普通符号使用时，则应该利用转义符号消除功能符号的特殊功能。转义字符包含一个倒斜杠(\)，紧跟着是想要添加到字符串中的字符。尽管它包含两个字符，但公认它是一个转义字符。例如，单引号的转义字符是\'。可以在单引号开始和结束的字符串中使用。为了看看转义字符的效果，在交互式环境中输入以下代码：

```
s="That is Tom\'s cat"
print(s)
```

结果也能正常输出：

```
That is Tom's cat
```

Python 知道，因为 Tom\'s 中的单引号有一个倒斜杠，所以它不是表示字符串结束的单引号。转义字符\'和\"让编程者在字符串中加入单引号和双引号。表 3-1 列出了常见的转义字符。

表 3-1 常见的转移字符

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	响铃
\b	退格(Backspace)
\e	转义
\000	空

续表

转义字符	描　述
\n	换行
\v	纵向制表符
\t	横向制表符
\r	回车
\f	换页
\oyy	八进制数，yy 代表的字符，例如：\o12 代表换行
\xyy	十六进制数，yy 代表的字符，例如：\x0a 代表换行
\other	其他的字符以普通格式输出

来看下面的程序：

```
s='That is \nTom\'s cat'
print(s)
s='That is \rTom\'s cat'
print(s)
```

运行结果：

```
That is
Tom's cat
Tom's cat
```

从结果来看，回车和换行效果差不多。实际上，它们还是有区别的。回车\r 本义是光标重新回到本行开头，r 的英文 return，控制字符可以写成 CR，即 Carriage Return。换行\n 本义是光标往下一行(不一定到下一行行首)，n 的英文是 newline，控制字符可以写成 LF，即 Line Feed。

3.2.4　字符串类型

有几种特殊的字符串。第一种是在字符串开始的引号之前加上 r，使它成为原始字符串。原始字符串会完全忽略所有的转义字符，打印出字符串中所有的倒斜杠。例如，以下代码：

```
s=r'That is\\\nTom\'s\rcat'
print(s)
```

运行结果：

```
That is\\\nTom\'s\rcat
```

可以看出，现在字符串里面所有的转义字符都失效了。因为这是原始字符串，

Python 认为倒斜杠是字符串的一部分，而不是转义字符的开始。如果输入的字符串包含许多倒斜杠，如后面要介绍的正则表达式字符串，那么原始字符串就很有用。

如果在字符串开头添加'u'，那么此字符串成为 Unicode 字符串。一般用在中文字符串前面，防止因为源码储存格式问题，导致再次使用时出现乱码。示例如下：

```
s1=u"北京"
print(s1)
```

运行结果：

```
北京
```

如果在字符串开头添加'b'，那么此字符串成为二进制字符串。二进制字符串是一个 bytes 对象，如网络编程中，服务器和浏览器只认 bytes 类型数据。Bytes 类型字符串和普通字符串可以相互转换，示例如下：

```
s1="北京"
s2=s1.encode("utf-8")
print(s2)
print(s2.decode(encoding="utf-8"))
```

运行结果：

```
b'\xe5\x8c\x97\xe4\xba\xac'
北京
```

此时变量 s2 的二进制字符串里面的值，实际上是“北京”这两个汉字的 Unicode 码，以二进制的形式表达出来。

3.2.5 多行字符串

如果一个字符串里面的字符太长，虽然可以用\n 转义字符将换行放入一个字符串，但使用多行字符串通常更容易。在 Python 中，多行字符串的起止是 3 个单引号或 3 个双引号。“三重引号”之间的所有引号、制表符或换行，都被认为是字符串的一部分。Python 的代码块缩进规则不适用于多行字符串。示例如下：

```
s="""Hello
            Tom's brother,
        How are you?"""
print(s)
```

运行结果：

```
Hello
            Tom's brother,
        How are you?
```

输出结果里面，保留了字符串里面的空格。同时，里面的单引号字符也不需要转义。

之前介绍过 Python 是用＃字符来开始一段注释。但＃号注释只能支持一行注释，如果要使用多行注释一般使用多行字符串，示例如下：

```
"""
This program tries to print all the lists.
Author:Tom
Date:2019-02-19
Version:1.0
"""
```

3.2.6 字符串运算符

1. 切片运算符

如果指定一个下标，将得到字符串在该处的字符。如果用一个下标和另一个下标指定一个范围，开始下标将被包含，结束下标则不包含。因此，如果字符串 s 是'Hello world!'，s［0:5］就是'Hello'。通过 s［0:5］得到的子字符串，将包含 s［0］到 s［4］的全部内容，而不包括下标 5 处的空格。下面给出各种切片所得的结果示例：

```
s="Hello world!"
print(s[0:5])
print(s[0:])
print(s[:5])
print(s[:])
print(s[-1:20])
```

运行结果：

```
Hello
Hello world!
Hello
Hello world!
!
```

请注意，字符串切片并没有修改原来的字符串。可以从一个变量中获取切片，记录在另一个变量中。示例如下：

```
s="Hello world!"
print(id(s))
new_s=s[4:6]
print(id(new_s))
```

运行结果：

```
1940063662640
```

```
1940034647296
```

可以看出，新的字符串和旧的字符串存储在两个不同的位置。

2. in 和 not in 运算符

和列表一样，in 和 not in 运算符可以用来判断一个字符串是否在另一个字符串中。示例如下：

```
s="Hello world!"
print('Hello' in s)
print('hello' not in s)
```

运行结果：

```
True
True
```

3. 连接运算符

连接运算符用来连接两个字符串，变成一个更大的字符串。示例如下：

```
s1='Hello'
s2='world!'
s3=s1+' '+s2
print(s3)
```

运行结果：

```
Hello world!
```

4. 复制运算符

复制运算符可以重复拼接字符串多次，能够非常方便地构造一个新字符串。示例如下：

```
s1='Hello '*3
print(s1)
```

运行结果：

```
Hello Hello Hello
```

3.2.7 字符串编码

关于字符串编码，一直是一个困扰语言学习者和编程工作者的难题。这里解释一下相关概念，并介绍两个字符串编码的相关方法。

1. 字符与字节

一个字符不等价于一个字节，字符是人类能够识别的符号，而这些符号要保存到计算

的存储中就需要用计算机能够识别的字节来表示。一个字符往往有多种表示方法，不同的表示方法会使用不同的字节数。这里所说的不同的表示方法就是指字符编码，如字母A～Z都可以用ASCII码表示(占用1字节)，也可以用Unicode表示(占用2字节)，还可以用UTF-8表示(占用1字节)。字符编码的作用就是将人类可识别的字符转换为机器可识别的字节码，以及反向过程。

Unicode才是真正的字符串，而用ASCII、UTF-8、GBK等字符编码表示的是字节串。关于这点，在Python的官方文档中经常可以看到这样的描述"Unicode string"，"translating a Unicode string into a sequence of bytes"。

代码写在文件中，而字符是以字节形式保存在文件中的，因此在文件中定义字符串时被当作字节串也是可以理解的。但是，有时需要的是字符串，而不是字节串。

对字符串取长度，结果应该是所有字符串的个数，无论中文还是英文。

对字符串对应的字节串取长度，就跟编码(encode)过程使用的字符编码有关(如UTF-8编码，一个中文字符需要用3字节来表示；GBK编码，一个中文字符需要用2字节来表示)

2. 编码与解码

Unicode字符编码，也是字符与数字的映射，但是这里的数字被称为代码点(code point)，实际上就是十六进制的数字。

Python官方文档中对Unicode字符串、字节串与编码之间的关系有如下描述。

Unicode字符串是一个代码点序列，代码点取值范围为0到0x10FFFF(对应的十进制为1114111)。这个代码点序列在存储(包括内存和物理磁盘)中需要被表示为一组字节(0～255的值)，而将Unicode字符串转换为字节序列的规则称为编码。

这里说的编码不是指字符编码，而是指编码的过程以及这个过程中所使用到的Unicode字符的代码点与字节的映射规则。这个映射不必是简单的一对一映射，因此编码过程也不必处理每个可能的Unicode字符。

将Unicode字符串转换为ASCII编码的规则很简单。对于每个代码点：

- 如果代码点数值＜128，则每字节与代码点的值相同。
- 如果代码点数值＞＝128，则Unicode字符串无法在此编码中进行表示(这种情况下，Python会引发一个UnicodeEncodeError异常)。

将Unicode字符串转换为UTF-8编码使用以下规则：

- 如果代码点数值＜128，则由相应的字节值表示(与Unicode转ASCII字节一样)。
- 如果代码点数值＞＝128，则将其转换为一个2字节，3字节或4字节的序列，该序列中的每个字节都在128到255之间。

简单总结如下：

- 编码：将Unicode字符串中的代码点转换为特定字符编码对应的字节串的过程和规则。
- 解码(decode)：将特定字符编码的字节串转换为对应的Unicode字符串中的代码点的过程和规则。

可见，无论是编码还是解码，都需要一个重要因素，就是特定的字符编码。因为一个字符用不同的字符编码进行编码后的字节值以及字节数大部分情况下是不同的，反之亦然。

3. Python 文件执行过程

磁盘上的文件都是以二进制格式存放的，其中文本文件都是以某种特定编码的字节形式存放的。对于程序源代码文件的字符编码是由编辑器指定的，如使用 Pycharm 来编写 Python 程序时会指定工程编码和文件编码为 UTF-8，那么 Python 代码被保存到磁盘时就会被转换为 UTF-8 编码对应的字节(encode 过程)后写入磁盘。当执行 Python 代码文件中的代码时，Python 解释器在读取 Python 代码文件中的字节串之后，需要将其转换为 Unicode 字符串(decode 过程)之后才执行后续操作。

这个转换过程(decode，解码)需要指定文件中保存的字节使用的字符编码是什么，才能知道这些字节在 Unicode 这张万国码和统一码中找到其对应的代码点是什么。这里指定字符编码的方式如图 3-1 所示。

```
#-*-coding:utf-8-*-
```

图 3-1 给出了 Python 文件的执行过程。

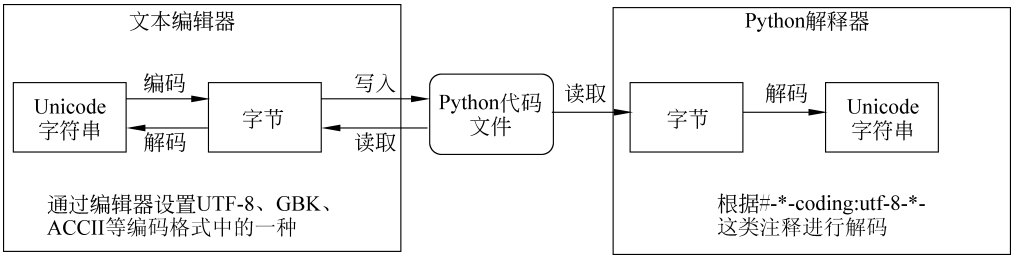


图 3-1 Python 文件的执行过程

3.3 字符串基本操作

3.3.1 大小写转换

字符串的一个重要问题是关于大小写方面的。如有些系统在登录时需要输入验证码，而验证码是不管大小写的，所以此时要把验证码统一转换为大写或者小写然后进行比对。和大小写相关的方法如下。

- upper()：转换 string 中的小写字母为大写。
- lower()：转换 string 中的大写字母为小写。
- swapcase()：翻转 string 中的大小写。
- istitle()：如果 string 是标题化的(见 title())则返回 True，否则返回 False。

- isupper()：如果 string 中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是大写，则返回 True，否则返回 False。
- islower()：如果 string 中包含至少一个区分大小写的字符，并且所有这些(区分大小写的)字符都是小写，则返回 True，否则返回 False。
- title()：返回“标题化”的 string，就是说所有单词都是以大写开始，其余字母均为小写。
- capitalize()：把字符串的第一个字符大写。

例 3-1 展示了各种常见大小写转换函数。

【例 3-1】 大小写转换示例

```
s1='Hello world! '
print(s1.istitle())
print(s1.title())
print(s1.swapcase())
print(s1.capitalize())
print(s1.isupper())
print(s1.islower())
s1=s1.upper()
print(s1)
print(s1.isupper())
s1=s1.lower()
print(s1)
print(s1.islower())
```

运行结果：

```
False
Hello World!
hELLO WORLD!
Hello world!
False
False
HELLO WORLD!
True
hello world!
True
```

从结果可以看出，swapcase()函数可以实现大小写互换，capitalize()函数将首字母变成大写；isupper()是用来判断一个字符串是不是大写；相应地，islower()函数是用来判断字符串是否为小写的。要注意，必须每个字符都是小写或者大写才会返回 True，如上例中因为一开始'H'为大写而其他字符为小写，所以两个函数都返回 False。

upper()函数可以将一个字符串所有字符转为大写，lower()函数则将所有字符转为小写。因为 upper()和 lower()字符串方法本身返回字符串，所以也可以在返回的字符串

上继续调用字符串方法。这样的方式称为方法调用链。示例如下：

```
s1='Hello world!'
print(s1.upper().lower().upper().islower())
```

3.3.2 字符类型判断

除了 islower()和 isupper()，还有几种字符串方法，它们的名字以 is 开始。这些方法返回一个布尔值，描述了字符串的特点。下面是一些常用的 isX 字符串方法。

- isalpha()：如果字符串只包含字母，并且非空，返回 True，否则返回 False。
- isalnum()：如果字符串只包含字母和数字，并且非空，返回 True，否则返回 False。
- isdecimal()：如果字符串只包含数字字符，并且非空，返回 True，否则返回 False。
- isnumeric()：如果字符串只包含数字字符，并且非空，返回 True，否则返回 False。
- isdigit()：如果字符串只包含数字字符，并且非空，返回 True，否则返回 False。
- isspace()：如果字符串只包含空格、制表符和换行，并且非空，返回 True，否则返回 False。
- istitle()：如果字符串仅包含以大写字母开头、后面都是小写字母的单词，返回 True，否则返回 False。

通过例 3-2 来了解这些方法的作用。

【例 3-2】 字符类型判断

```
s1='Hello'
print(s1.isalpha())
s2=" "
print(s2.isspace())
s3="2"
print(s3.isnumeric())
print(s3.isdigit())
print(s3.isdecimal())
s4="2a"
print(s3.isalnum())
```

运行结果：

```
True
True
True
True
True
True
```

从上例可以看到，isdigit()、isdecimal()和 isnumeric()三个函数看起来非常相似。实际上是有区别的，如表 3-2 所示。

表 3-2　函数对比表

函数	Unicode 数字	Byte 数字（单字节）	全角数字（双字节）	汉字数字	罗马数字	小数
isdigit()	True	True	True	False	False	False
isdecimal()	True	Error	True	False	False	False
isnumeric()	True	Error	True	True	True	False

3.3.3　字符串检查

在对字符串进行操作之前，往往还需要判断一下字符串的其他相关属性。这类方法具体介绍如下。

- count(sub,start= 0,end=len(string))：sub 表示搜索的子字符串，start 表示字符串开始搜索的位置。默认为第一个字符，第一个字符索引值为 0。end 表示字符串中结束搜索的位置。字符中第一个字符的索引为 0。默认为字符串的最后一个位置。该方法返回 str 在 string 里面出现的次数，如果 beg 或者 end 指定则返回指定范围内 str 出现的次数。
- startswith(str,beg=0,end=len(string))：str 表示检测的字符串，beg 用于设置字符串检测的起始位置。默认为第一个字符，第一个字符索引值为 0。end 表示用于设置字符串检测的结束位置。字符中第一个字符的索引为 0。默认为字符串的最后一个位置。该方法用于检查字符串是否是以指定子字符串开头，如果是则返回 True，否则返回 False。如果参数 beg 和 end 指定值，则在指定范围内检查。
- endswith(str,beg=0,end=len(string))：str 表示检测的字符串，beg 用于设置字符串检测的起始位置。默认为第一个字符，第一个字符索引值为 0。end 表示用于设置字符串检测的结束位置。字符中第一个字符的索引为 0。默认为字符串的最后一个位置。该方法用于检查字符串是否是以指定子字符串结尾，如果是则返回 True，否则返回 False。如果参数 beg 和 end 指定值，则在指定范围内检查。

下面通过例 3-3 来了解这些函数的用法。

【例 3-3】　字符串检查

```
s1='Hello world!'
print(s1.count('l',0,5))
print(s1.startswith('He',2,5))
print(s1.endswith('orld'))
```

运行结果：

```
2
False
False
```

3.3.4 字符串格式化

在输出字符串时，往往需要将该字符串进行格式化，以便增加可读性。支持格式化字符串的相关函数如下。

- center(width,[fillchar])：返回一个原字符串居中，并使用 fillchar 字符填充至长度 width 的新字符串，默认使用空格。
- ljust(width,[fillchar])：返回一个原字符串左对齐，并使用 fillchar 字符填充至长度 width 的新字符串，默认使用空格。
- rjust(width,[fillchar])：返回一个原字符串右对齐，并使用 fillchar 字符填充至长度 width 的新字符串，默认使用空格。
- lstrip([chars])：截掉 string 左边的字符串，默认使用空格。
- rstrip([chars])：截掉 string 末尾的空格，默认使用空格。
- strip([chars])：在 string 上执行 lstrip()和 rstrip()，默认使用空格。
- zfill(width)：返回长度为 width 的字符串，原字符串 string 右对齐，前面填充 0。

下面通过例 3-4 来了解这些函数的用法。

【例 3-4】 字符串格式化

```
s1='12'
print(s1.center(10, '*'))
print(s1.ljust(10, '*'))
print(s1.rjust(10, '*'))
print(s1.zfill(10))
s2='001200'
print(s2.lstrip('00'))
print(s2.rstrip('0'))
print(s2.strip('0'))
```

运行结果：

```
****12****
12********
********12
0000000012
1200
0012
12
```

下面单独介绍一个非常强大的字符串方法，format()。该函数能够对字符串做出各种格式化操作，参数可以不限个数，位置可以不按顺序。示例如下：

```
#通过位置
print('{0},{1}'.format('Tom',20))
```

```
print('{},{}'.format('Tom',20))
print('{1},{0},{1}'.format('Tom',20))
```

运行结果：

```
Tom,20
Tom,20
20,Tom,20
```

也可以通过设置关键字参数，来指定位置：

```
#通过关键字参数
print('{name},{age}'.format(age=18,name='Tom'))
#通过列表索引设置参数
my_list=['Tom',18]
print("姓名：{0[0]}，年龄：{0[1]}".format(my_list))  #"0" 是必须的
```

运行结果：

```
Tom,18
姓名：Tom，年龄：18
```

也可以用来实现填充和对齐，示例如下：

```
#填充与对齐
print('{:>8}'.format('189'))          #189 右对齐
print('{:<8}'.format('189'))          #189 左对齐
print('{:^8}'.format('189'))          #189 中间对齐
print('{:0>8}'.format('189'))         #00000189 右对齐并补 0
print('{:a>8}'.format('189'))         #aaaaa189 右对齐补 a
```

输出结果：

```
     189
189
  189
00000189
aaaaa189
```

也可以实现不同精度和类型的设置，示例如下：

```
#精度与类型
print('{:.2f}'.format(321.33345))      #321.33 2 表示小数位数
print('{:,}'.format(1234567890))       #1,234,567,890 用来做金额的千位分隔符
print('{:.3%}'.format(0.25))           #25.000%3 表示百分比中小数点位数
print('{:.3e}'.format(2500000000))     #2.500e+09 指数输出，3 表示小数点位数
```

运行结果：

```
321.33
```

```
1,234,567,890
25.000%
2.500e+09
```

最后还可以用来实现不同进制的输出：

```
#b、d、o、x 分别是二进制、十进制、八进制、十六进制。#x 小写输出,#X 大写输出
print('{:b}'.format(10))                    #二进制 10010
print('{:d}'.format(10))                    #十进制 18
print('{:o}'.format(10))                    #八进制 22
print('{:x}'.format(10))                    #十六进制 12
print('{:#x}'.format(10))                   #十六进制 0xa
print('{:#X}'.format(10))                   #十六进制 0XA
```

运行结果：

```
1010
10
12
a
0xa
0XA
```

3.3.5 字符串查找

字符串类型支持查找的方法有如下几种。

- string.find(str,beg=0,end=len(string))：检测 str 是否包含在 string 中，如果 beg 和 end 指定范围，则检查是否包含在指定范围内，如果是返回开始的索引值，否则返回-1。
- string.index(str,beg=0,end=len(string))：跟 find()方法一样，只不过如果 str 不在 string 中会报一个异常。
- max(str)：返回字符串 str 中最大的字母，根据 ASCII 码来比较；这是内置函数，不是字符串类型提供的方法。
- min(str)：返回字符串 str 中最小的字母，根据 ASCII 码来比较；这是内置函数，不是字符串类型提供的方法。
- string.rfind(str,beg=0,end=len(string))：类似于 find()函数，不过是从右边开始查找。
- string.rindex(str,beg=0,end=len(string))：类似于 index()，不过是从右边开始。

下面通过例 3-5 了解上面函数的使用方式。

【例 3-5】 字符串查找

```
s='Hello'
```

```
print(max(s))
print(min(s))
print(s.find('l'))
print(s.index('l'))
print(s.rfind('l'))
print(s.rindex('l'))
print(s.find('m'))
print(s.index('m'))
```

运行结果：

```
o
H
2
2
3
3
-1
Traceback (most recent call last):
  File "D:\Python32\chapter2\src\data.py", line 9, in <module>
    print(s.index('m'))
ValueError: substring not found
```

3.3.6 字符串修改

字符串修改涉及一些重要的字符串函数，具体介绍如下。

- join(seq)：以 string 作为分隔符，将 seq 中所有的元素(的字符串表示)合并为一个新的字符串。
- split(str="",num=string.count(str))：以 str 为分隔符切片 string，如果 num 有指定值，则仅分隔 num+1 个子字符串。Str 指分隔符，默认为所有的空字符，包括空格、换行(\n)、制表符(\t)等。Num 指定分隔次数，默认为-1，即分隔所有；若指定，则返回列表的元素个数，限定为 Num+1 个。
- splitlines([keepends])：按照行('\r','\r\n',\n')分隔，返回一个包含各行作为元素的列表，如果参数 keepends 为 False，不包含换行符，如果为 True，则保留换行符。
- partition(str)：str 为分隔符，该方法返回一个 3 元的元组，第一个为分隔符左边的子串，如果此字符串没有此分隔符，此子串为第一个字符串；第二个为分隔符本身；第三个为分隔符右边的子串。
- rpartition(str)：类似于 partition()函数，不过是从右边开始查找。
- replace(str1,str2,num=string.count(str1))：把 string 中的 str1 替换成 str2，如果 num 指定，则替换不超过 num 次。
- maketrans(intab,outtab])：maketrans()方法用于创建字符映射的转换表，对于接受两个参数的最简单的调用方式，第一个参数是字符串，表示需要转换的字符，

第二个参数也是字符串表示转换的目标。

- translate(str)：根据 str 给出的表(包含 256 个字符)转换 string 的字符。

下面通过例 3-6 来阐述上述函数。

【例 3-6】 字符串修改

```
s='Hello world\tHello'
print('-'.join(s))
print(s.split())
print(s.split("l"))
print(s.split("l",1))
print(s.splitlines())
print(s.partition('l'))
print(s.partition('m'))
print(s.rpartition('l'))
print(s.replace('l','m',2))
trantab=s.maketrans('oled', '1234')
print(s.translate(trantab))
```

运行结果：

```
H-e-l-l-o--w-o-r-l-d-  -H-e-l-l-o
['Hello', 'world', 'Hello']
['He', '', 'o wor', 'd\tHe', '', 'o']
['He', 'lo world\tHello']
['Hello world\tHello']
('He', 'l', 'lo world\tHello')
('Hello world\tHello', '', '')
('Hello world\tHel', 'l', 'o')
Hemmo world Hello
H3221 w1r24 H3221
```

3.3.7 字符串编解码

字符串编码的知识 3.2.7 节已经介绍过，本节介绍字符串支持编解码的相关方法。

- encode(encoding='UTF-8',errors='strict')：以 encoding 指定的编码格式编码 string，得到一个 bytes 类型对象，如果出错默认报一个 ValueError 的异常，除非 errors 指定的是'ignore'或者'replace'。encoding 参数可选，即要使用的编码，默认编码为 'utf-8'。字符串编码常用类型有 utf-8、gb2312、cp936、gbk 等。errors 参数也可选，用来设置不同错误的处理方案。默认为 'strict'，意为编码错误引起一个 UnicodeEncodeError。其他可能值有 'ignore'、'replace'、'xmlcharrefreplace'等。
- decode(encoding='UTF-8',errors='strict')：以 encoding 指定的编码格式解码 string，errors 参数和 encode()函数一样。

下面通过例 3-7 了解编解码函数的使用方式。

【例 3-7】 字符串编解码

```
str1="你好,世界"
str2="Hello world"
print("utf8 编码: ",str1.encode(encoding="utf8",errors="strict")) #等价于
print("utf8 编码: ",str1.encode("utf8"))
print("utf8 编码: ",str2.encode(encoding="utf8"))
print("gbk 编码: ",str1.encode(encoding="gbk"))
print("gbk 编码: ",str2.encode(encoding="gbk"))
```

运行结果:

```
utf8 编码: b'\xe4\xbd\xa0\xe5\xa5\xbd\xef\xbc\x8c\xe4\xb8\x96\xe7 \x95\x8c'
utf8 编码: b'Hello world'
gbk 编码: b'\xc4\xe3\xba\xc3\xa3\xac\xca\xc0\xbd\xe7'
gbk 编码: b'Hello world'
```

3.4 列表概述

3.4.1 列表的定义

列表由一系列按特定顺序排列的元素组成,是一个用 list 类定义的序列,包括了创建、操作和处理列表的方法。可以创建包含字母表中所有字母、数字 0～9 或所有家庭成员姓名的列表;也可以将任何东西加入列表中,其中的元素之间可以没有任何关系。

鉴于列表通常包含多个元素,给列表指定一个表示复数的名称(如 letters、digits 或 names)是个不错的主意。

在 Python 中,用方括号([])来表示列表,并用逗号来分隔其中的元素。列表的定义有两种常见格式:使用方括号和使用 list 类的构造方法。

1. 使用方括号

常见的列表定义方式是使用下面这种格式:

```
列表名=[列表元素 1,列表元素 2, ...]
```

下面是一个简单的列表示例,这个列表包含一个星期的每天:

```
week_day=['星期一','星期二','星期三','星期四',
        '星期五','星期六','星期日']
print(week_day)
```

如果让 Python 将此列表打印出来,Python 将打印列表的内部表示,包括方括号:

```
['星期一', '星期二', '星期三', '星期四', '星期五', '星期六', '星期日']
```

也可以定义为

```
week_day=[]
```

这表示是一个空列表，里面什么元素也没有。但可以使用列表的一些操作添加新元素。

方括号内甚至可以放任何支持迭代的容器，或者迭代对象，如果该对象已经是一个list，那么定义新 list 实际上是一个复制过程。

```
week_day1=['星期一','星期二']
week_day2=[x for x in week_day1]
```

此时 week_day2 列表中的元素和 week_day1 中的一模一样。

2. 使用 list 类的构造方法

通过 list 类的构造方法，定义语法格式为

```
列表名=list(支持迭代的对象)
```

支持迭代的对象包括各种序列类型，如字符串、列表、元组、集合等。示例如下：

```
list1=list()                    #空列表
list2=list([1,2,3])             #包含三个元素 1,2,3 的列表
list3=list(range(2,6))          #包含四个元素 2,3,4,5 的列表
list4=list('abd')               #包含三个元素'a','b','d'的列表
```

列表是一种序列，看起来和其他语言中的数组差不多。实际上还是有区别的，因为一个列表是任何元素的序列，就是说列表中的元素的类型可以是各种类型，可以这样定义：

```
list5=[1, 2.9, 'name', [1,2]]
```

3.4.2 列表元素

列表是一个有序序列。刚才的例子中只能一次输出所有元素，如果想要访问列表中个别元素，需要通过元素的下标(也称位置或者索引)来得到。访问语法如下：

```
列表名[下标]
```

想要访问列表 week_day 中的“星期三”，可以这样：

```
week_day=['星期一','星期二','星期三','星期四',
        '星期五','星期六','星期日']
print(week_day[2])
```

当请求获取列表元素时，Python 只返回该元素，而不包括方括号和引号：

```
星期三
```

这正是要让用户看到的结果——整洁、干净的输出。

还可以对任何列表元素调用第 2 章介绍的字符串方法。例如，可使用字符串类的一个函数 title()让输出元素的格式更整洁：

```
print(week_day[2].title())
```

特别要注意的是，第一个元素的下标是从 0 开始的，而不是 1。一般情况下，一个 n 个元素的列表，其下标的范围是：0～n－1。如果不小心访问了超出下标范围的列表元素，例如：

```
week_day=['星期一','星期二','星期三','星期四', '星期五','星期六','星期日']
print(week_day[7])
```

就会得到这样的异常信息：

```
IndexError: list index out of range
```

图 3-2 给出了列表 week_day 中的 7 个元素。

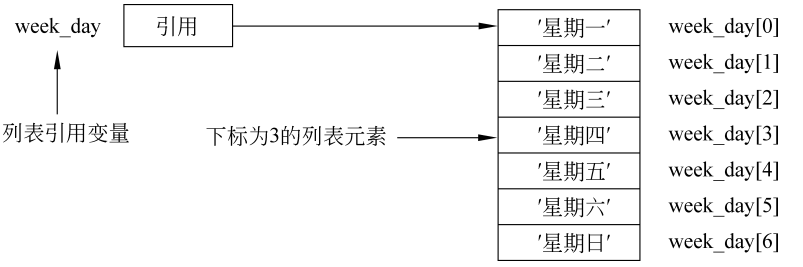


图 3-2　列表 week_day 中的 7 个元素

Python 为访问最后一个列表元素提供了一种特殊语法。通过将索引指定为－1，可以让 Python 返回最后一个列表元素：

```
week_day=['星期一','星期二','星期三','星期四',
        '星期五','星期六','星期日']
print(week_day[-1])
print(week_day[-7])
```

运行结果：

```
星期日
星期一
```

当下标为－1 时，访问的是倒数第一个元素，也就是最后一个元素。而下标为－7 时，访问的是倒数第 7 个元素，也就是第一个元素。实际上，如果下标值为负数，可以用列表元素的长度和下标进行计算，从而得到下标位置。在本例中，元素个数是 7，下标为－1 时，使用 7＋(－1)＝6，实际上访问的就是下标为 6 的元素。

下标为负数时，也是有范围的，如果这样访问也会有下标越界的错误：

```
print(week_day[-8])
```