第10章. 字符串和二进制序列

10-1. 字符串的表示

（教程：3.1.2）
（语言参考手册：2.4.1、2.4.2、3.3.1）
（标准库：内置函数、内置类型）

在前面的例子中我们已经多次与字符串打交道。Python中的字符串在内部总是采用某种Unicode编码方式，第7章已经说明可以通过sys.getdefaultencoding()查看具体编码方式，默认为UTF-8。

内置类型str表示字符串。可以通过内置函数str()来创建字符串。str()的第一种语法为：

```
class str(object='')
```

其功能相当于将通过object参数指定的对象强制类型转换为字符串，具体规则为：

- 判断是否可以将其理解为str()的第二种语法，如果是则转向它。
- 如果该对象具有__str__魔术属性，则调用它。
- 否则，如果该对象具有__repr__魔术属性，则调用它。
- 如果上述条件都不能满足，则抛出TypeError异常。

特别的，若省略了object参数，则得到空串。

在第7章介绍过，内置函数repr()也可以将传入的对象转换成一个字符串，那么它与str()有什么区别呢？上面的规则1和2的含义是，当某个类型没有实现__str__，只实现了__repr__时，str()和repr()是等价的。然而第3章说明，object同时实现了__str__和__repr__，所以str()其实总是调用传入对象的__str__魔术属性。下面的例子说明object的__str__和__repr__是等价的：

```
>>> obj = object()
>>> obj.__repr__()
'<object object at 0x1011bc840>'
>>> obj.__str__()
'<object object at 0x1011bc840>'
>>>
```

仅当一个类型同时实现了__str__和__repr__，且两者不同时，str()和repr()的行为才会有差异。在这种情况下，通常__str__提供适合人类阅读的字符串，而__repr__提供适合机器分析的字符串。

事实上还有一个与repr()类似的内置函数ascii()，语法为：

```
ascii(object)
```

它与repr()的区别在于返回的字符串中仅包含ASCII字符，非ASCII字符会用转义序列表示。下面是一个例子：

```
>>> repr('123你好abc')
"'123你好abc'"
>>> ascii('123你好abc')
"'123\\u4f60\\u597dabc'"
>>>
```

str()的第二种语法为：

```
class str(object=b'', encoding='utf-8', errors='strict')
```

当通过object参数指定的对象是二进制序列时，就意味着使用str()的这种语法。encoding参数指定用于解读该二进制序列的编码方式；errors参数指定当遇到指定编码方式无法解读的编码时如何处理。（这两个参数与io.TextIOWrapper中的encoding和errors参数完全相同。）

虽然str()并不涉及字符串字面值，但后者却是比str()使用更频繁的获得字符串的手段。

字符串字面值有四种格式：

```
[prefix]'shortstring'

[prefix]"shortstring"

[prefix]'''longstring'''

[prefix]"""longstring"""
```

其中shortstring代表能写在一行内的短字符串，longstring代表必须写成多行的长字符串。而prefix代表的字符串前缀被总结在表10-1中，注意r/R可以和f/F联合使用，但u/U不能和r/R以及f/F联合使用。

表10-1. 字符串前缀

前缀	说明
u、U	Unicode字符串字面值。表示该字符串使用某种Unicode编码，不论脚本的编码声明是什么。
r、R	原始字符串字面值。不支持转义序列，即将“\”视为普通字符，而非转义标志。
f、F	格式化字符串字面值。可包含通过大括号标明的表达式。

当字符串的前缀中没有r/R时，字符串内可以包含表2-1列出的转义序列。此外，当用'或'''作为字符串字面值分界标志时，字符串中可以直接包含"，不用将其写为转义序列\"；当用"或"""作为字符串字面值分界标志时，字符串中可以直接包含'，不用将其写为转义序列\'。下面是一些例子：

```
>>> 'How are you?'
'How are you?'
>>> 'I\'m fine.'
"I'm fine."
>>> "How are you?"
'How are you?'
>>> "I'm fine."
"I'm fine."
>>> s = '''\
... James: "How are you?"
... Lucy: "I\'m fine."\
... '''
>>> s
'James: "How are you?"\nLucy: "I\'m fine."'
>>> print(s)
James: "How are you?"
Lucy: "I'm fine."
>>> s = """\
... James: \"How are you?\"
... Lucy: \"I'm fine.\"\
... """
>>> s
'James: "How are you?"\nLucy: "I\'m fine."'
>>> print(s)
James: "How are you?"
Lucy: "I'm fine."
>>>
```

从这些例子可以看出如下规律：

➤ 长字符串中的EOL会被原封不动地保留，但通过在行末尾添加“\”可以去掉不需要的EOL。
➤ 当通过sys.displayhook()将一个字符串写入标准输出时，会自动将其转化为字符串字面值的第一种格式，但会保留转义序列；而当通过print()将一个字符串写入标准输出时，会去掉字符串字面值两侧的引号，并将所有转义序列替换为相应字符。

此外，PEP 8建议长字符串字面值总是使用"""作为分界标志，而不要使用'''。

字符串前缀u/U仅当Python脚本本身不采用某种Unicode编码时才有必要使用；r/R则是当字符串中包含很多“\”，而又不想逐个写为转义序列“\\”时才有必要使用。值得详细说明的是f/F，它实现了“格式化字符串字面值（formatted string literals）”，这会在后面被详细讨论。

最后，短字符串字面值写在一起时，会被自动合并。请看下面的例子：

```
>>> 'H' 'e' 'll' 'o'
'Hello'
>>> "Hello" ' world!'
'Hello world!'
>>>
```

这一机制是在词法分析阶段实现的，即从行中提取令牌时会自动合并相邻的字符串令牌。而由于这些令牌不可能跨行，该机制只能应用于短字符串字面值。

在实际编写Python脚本时，可以利用上述机制将一个长字符串写成大量相邻的短字符串，这样就可以在维持代码可读性的前提下省掉转义不需要的换行符的麻烦。请看下面的例子：

```
>>> ('a, b, c, d, '
...          'e, f, g, h, '
...          'i, j, k, l, '
...          'm, n, o, p, '
...          'q, r, s, t, '
...          'u, v, w, x, '
...          'y, z')
'a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y,
z'
>>>
```

10-2. 字符串的操作

（教程：3.1.2）
（语言参考手册：3.2、6.10.1）
（标准库：内置函数、内置类型）

内置函数chr()和ord()提供了在字符和它的Unicode码位之间转换的手段。chr()的语法为：

```
chr(codepoint)
```

其codepoint参数需传入0x0～0x10FFFF范围内的整数，返回的对象是仅包含一个字符的字符串，该字符的Unicode码位恰好等于该整数。

而ord()的语法为：

```
ord(character)
```

其character参数需传入仅包含一个字符的字符串，返回的对象是代表该字符的Unicode码位的整数。

显然，chr()和ord()互为逆操作。这里需要强调的是，Unicode码位是Unicode技术分配给字符的固定数字，具有固定不变的映射关系，与具体采用的编码方式无关。下面是一个例子：

```
>>> ord('€')
8364
>>> chr(8364)
'€'
>>> bytearray('€', 'utf8')
bytearray(b'\xe2\x82\xac')
>>> bytearray('€', 'utf16')
bytearray(b'\xff\xfe\xac ')
>>>
```

该例子说明欧元符号“€”的Unicode码位是8364，但采用UTF-8编码时“€”的编码为3个字节，而采用UTF-16编码时“€”的编码为4个字节。（bytearray()将在本章后面讨论。）

字符串可以通过“+”相互拼接，例如：

```
>>> 'a' + 'b' + 'c' + 'd'
'abcd'
>>> """a
... b
... c""" + '\nd'
'a\nb\nc\nd'
>>>
```

与相邻短字符串字面值的拼接不同，通过“+”实现的字符串拼接是在脚本执行时进行的。值得说明的是，CPython为了优化脚本执行速度引入了“常数折叠（constant folding）”功能，通过“+”拼接不超过20个字符串字面值时，会在编译时就将它们合并为一个字符串字面值，这就与相邻短字符串字面值的拼接相同了。

此外，通过“+”还可以拼接引用字符串的变量，例如：

```
>>> str1 = 'abc'
>>> str2 = 'xyz'
```

```
>>> str1 + 'def'
'abcdef'
>>> str1 + str2
'abcxyz'
>>>
```

这是无法通过相邻短字符串字面值的拼接实现的。

拼接多个相同的字符串时，还可以通过让该字符串与一个整数做乘法实现。注意当该整数不是正数时，将得到空串。请看下面的例子：

```
>>> 3 * 'abc'
'abcabcabc'
>>> "xy" * 2
'xyxy'
>>> 0 * 'hello'
''
>>> 'open' * -1
''
>>>
```

拼接字符串的“+”和“*”也可以混合使用，例如：

```
>>> 2*'ab' + 'xyz' + 'ab'*3
'ababxyzababab'
>>>
```

从字符串支持“+”和“*”，可以推知字符串也支持“+=”和“*=”，例如：

```
>>> s = 'a'
>>> s += 'b'
>>> s
'ab'
>>> s *= 3
>>> s
'ababab'
>>>
```

除了字符串拼接之外，str还提供了表10-2列出的函数属性来实现字符串操作。下面将依次讨论它们，除了str.format()和str.format_map()会放在下一节讨论。

表10-2. str属性

范畴	属性	说明
编码	**str.encode()**	返回该字符串在指定编码方式下的字节串。

范畴	属性	说明
大小写	str.upper()	返回该字符串的所有字符均大写的副本。
	str.lower()	返回该字符串的所有字符均小写的副本。
	str.swapcase()	返回该字符串的所有字符均反转大小写的副本。
	str.capitalize()	返回该字符串的首字符大写、其余字符均小写的副本。
	str.title()	返回该字符串的每个单词首字符大写、其余字符均小写的副本。
	str.casefold()	返回该字符串的消除大小写的副本。
替换	str.replace()	返回将该字符串的指定子串替换为另一个子串的副本。
	str.maketrans()	一个静态方法，返回一个供str.translate()使用的转换对照表。
	str.translate()	返回将该字符串按照转换对照表进行一对一替换的副本。
	str.expandtabs()	返回将该字符串中的水平制表符替换为空格的副本。
去掉前后缀	str.strip()	返回去掉了字符串开头和末尾的指定字符的副本。
	str.lstrip()	返回去掉了字符串开头的指定字符的副本。
	str.rstrip()	返回去掉了字符串末尾的指定字符的副本。
	str.removeprefix()	返回去掉了字符串开头的指定子串的副本。
	str.removesuffix()	返回去掉了字符串末尾的指定子串的副本。
填充	str.ljust()	返回字符串在末尾用指定字符填充到指定长度的副本。
	str.rjust()	返回字符串在开头用指定字符填充到指定长度的副本。
	str.center()	返回字符串在开头和末尾用指定字符填充到指定长度的副本。
	str.zfill()	返回代表数字的字符串在开头用0填充到指定长度的副本。
合并与拆分	str.join()	以字符串为分隔符将多个其他字符串合并。
	str.split()	以指定子串作为分隔符将字符串拆分，正向搜索。
	str.rsplit()	以指定子串作为分隔符将字符串拆分，反向搜索。
	str.partition()	以指定子串作为分隔符将字符串拆分为3部分，正向搜索。
	str.rpartition()	以指定子串作为分隔符将字符串拆分为3部分，反向搜索。
	str.splitlines()	将字符串按行拆分。
子串	str.count()	返回指定子串在字符串指定区域内的非重叠出现次数。
	str.find()	返回指定子串在字符串指定区域内首次出现的位置，正向搜索。
	str.rfind()	返回指定子串在字符串指定区域内首次出现的位置，反向搜索。
	str.index()	与str.find()的区别是未找到指定子串时抛出ValueError异常。
	str.rindex()	与str.rfind()的区别是未找到指定子串时抛出ValueError异常。
	str.startswith()	判断字符串是否以指定子串开头。
	str.endswith()	判断字符串是否以指定子串结尾。
	str.isupper()	判断字符串是否所有字符均大写。
	str.islower()	判断字符串是否所有字符均小写。

范畴	属性	说明
判断	**str.istitle()**	判断字符串是否每个单词首字符大写、其余字符均小写。
	str.isidentifier()	判断字符串是否是有效的标识符。
	str.isprintable()	判断字符串是否只包含可打印字符。
	str.isascii()	判断字符串是否只包含ASCII字符。
	str.isspace()	判断字符串是否只包含空白符。
	str.isalnum()	判断字符串是否只包含字母和数字。
	str.isalpha()	判断字符串是否只包含字母。
	str.isnumeric()	判断字符串是否只包含用于表示数字的字符。
	str.isdigit()	判断该字符串是否只包含数字。
	str.isdecimal()	判断该字符串是否只包含十进制数字。
格式化	**str.format()**	执行字符串格式化操作。
	str.format_map()	str.format()的一种特殊形式。

---

str.encode()的语法为：

```
str.encode(encoding='utf-8', errors='strict')
```

该函数返回一个二进制序列，为该字符串在通过encoding参数指定的编码方式下的编码。errors参数用于指定遇到指定的编码方式不支持的字符时如何处理。下面是一个例子：

```
>>> s = 'décevoir'
>>> s.encode()
b'd\xc3\xa9cevoir'
>>> s.encode('latin-1')
b'd\xe9cevoir'
>>>
```

使用大小写相关的函数属性时需要注意以下几点：

➤ 只有区分大小写的字符才会受它们影响。
➤ 有些语言的字符大小写不是一一对应的，导致s.swapcase().swapcase() == s不一定成立（但对英语来说这是成立的）。
➤ 在将整个字符串的首字符大写时，该字符会被放入Unicode的Lt类而非Lu类。这导致s.upper().isupper()有可能为Fasle（但对英语来说这总是得到True）。
➤ str.casefold()是str.lower()的进阶版，例如会将德语中的小写字母“ß”进一步替换为“ss”。

下面的例子说明了大小写相关的函数属性的作用：

```
>>> s = 'eine Straße mit vier Bahnen'
>>> s.upper()
'EINE STRASSE MIT VIER BAHNEN'
>>> s.lower()
'eine straße mit vier bahnen'
>>> s.swapcase()
'EINE sTRASSE MIT VIER bAHNEN'
>>> s.capitalize()
'Eine straße mit vier bahnen'
>>> s.title()
'Eine Straße Mit Vier Bahnen'
>>> s.casefold()
'eine strasse mit vier bahnen'
>>>
```

---

str.replace()实现了基于子串的替换，其语法为：

```
str.replace(old, new[, count])
```

其中old参数是需要被替换的子串，new参数是被替换成的子串。替换时将从字符串的左侧开始搜索，每遇到一处就替换一处，至多count次（如果省略了count参数则替换所有子串）。

下面的例子说明了该函数属性的用法：

```
>>> s = 'To be or not to be.'
>>> s.replace('o', 'a')
'Ta be ar nat ta be.'
>>> s.replace('o', 'a', 2)
'Ta be ar not to be.'
>>> s.replace('be', 'do')
'To do or not to do.'
>>> s.replace('be', 'doo', 1)
'To doo or not to be.'
>>>
```

str.maketrans()和str.translate()总是联合使用以实现字符的映射。str.maketrans()是一个静态方法，功能是生成一个“转换对照表”。而str.translate()则基于该转换对照表进行字符替换。转换对照表是一个提供了__getitem__函数属性的对象（通常是序列或映射类型的对象），其作用是以一个字符为索引，得到另一个字符、一个字符串或None。这样就可以将该字符映射到另一个字符或字符串，或者将其删除（None的含义）。

str.maketrans()的第一种语法是：

```
staticmethod str.maketrans(table)
```

其table参数必须被传入一个字典：该字典的键可以是仅包含单个字符的字符串，也可以是一个非负整数（被解读为某个字符的Unicode码位）；而相应的值则可以是任意长度的字符串或None。可以认为，该字典本身就是转换对照表。

str.maketrans()的第二种语法为：

```
staticmethod str.maketrans(x, y[, z])
```

其中x参数和y参数必须被传入长度相等的两个字符串，表示转换对照表需将x中的字符映射到y中相同位置的字符；如果给出了z参数，则也必须被传入一个字符串，表示转换对照表需将z中的每个字符都映射到None。

str.translate()的语法则是：

```
str.translate(table)
```

其中table参数需被传入代表转换对照表的对象。

下面的例子说明了上述两个函数属性的用法：

```
>>> s = 'To be or not to be.'
>>> table1 = str.maketrans({'e':'x', 'o':'yz', 'b':None})
>>> table2 = str.maketrans('oe', 'ai', 'tn')
>>> s.translate(table1)
'Tyz x yzr nyzt tyz x.'
>>> s.translate(table2)
'Ta bi ar a a bi.'
>>>
```

str.expandtabs()用于将字符串中的水平制表符替换为空格，其语法为：

```
str.expandtabs(tabsize=8)
```

其中tabsize参数用于指定将一个水平制表符替换为几个空格，如果忽略则默认替换为8个空格。下面的例子说明了该函数属性的用法：

```
>>> s = 'a\tb\tc'
>>> print(s)
a     b     c
>>> s.expandtabs()
'a       b       c'
>>> s.expandtabs(4)
'a   b   c'
>>> s.expandtabs(1)
'a b c'
>>>
```

---

str.strip()、str.lstrip()和str.rstrip()的功能是基于指定字符识别字符串的前缀和/或后缀，并将其清除。它们的语法为：

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

其中chars参数需要被传入一个字符串，而识别前缀/后缀的方法是：从字符串首/尾字符开始向后/前依次检查，直到找到第一个不在通过chars参数指定的字符串中的字符为止，该字符之前/后的部分被视为前缀/后缀。如果chars参数被省略，则默认取空白符（空格、制表符和分页符）。下面的例子说明了上述函数属性的用法：

```
>>> s = 'abc789cba'
>>> s.strip('a')
'bc789cb'
>>> s.strip('ac')
'bc789cb'
>>> s.strip('abc')
'789'
>>>
>>> s = 'www.example.com'
>>> s.strip('cmow.')
'example'
>>> s.lstrip('cmow.')
'example.com'
>>> s.rstrip('cmow.')
'www.example'
>>>
```

str.removeprefix()和str.removesuffix()的功能则是基于指定子串识别字符串的前缀或后缀，并将其清除。它们的语法为：

```
str.removeprefix(prefix)
str.removesuffix(suffix)
```

当调用str.removeprefix()时，字符串的开头必须严格匹配prefix参数传入的字符串才会被当成前缀清除，否则什么也不做；当调用str.removesuffix()时，字符串的末尾必须严格匹配suffix参数传入的字符串才会被当成后缀清除，否则什么也不做。下面的例子说明了它们的作用：

```
>>> s = 'www.example.com'
>>> s.removeprefix('www.')
'example.com'
>>> s.removesuffix('www.')
'www.example.com'
>>> s.removeprefix('.com')
'www.example.com'
>>> s.removesuffix('.com')
'www.example'
>>>
```

---

str.ljust()、str.rjust()和str.center()使我们可以用指定字符将字符串填充到指定长度。它们的语法分别是：

```
str.ljust(width[, fillchar])
str.rjust(width[, fillchar])
str.center(width[, fillchar])
```

其中width参数需被传入一个整数，用于指定长度；fillchar参数需被传入仅包含一个字符的字符串，表示用该字符填充，如果省略则默认使用空格。如果指定的长度小于或等于原字符串的长度，则返回原字符串本身。此外，str.ljust()会导致原字符串在填充后字符串的开头，str.rjust()会导致原字符串在填充后字符串的末尾，而str.center()会尽可能使原字符串位于填充后字符串的中间。下面的例子说明了它们的作用：

```
>>> s = 'aaa'
>>> s.ljust(7, 'b')
'aaabbbb'
>>> s.ljust(7)
'aaa    '
>>> s.ljust(2, 'b')
'aaa'
>>> s.rjust(5, 'b')
'bbaaa'
>>> s.rjust(5)
'  aaa'
>>> s.rjust(3)
'aaa'
>>> s.center(9, 'b')
'bbbaaabbb'
>>> s.center(9)
'   aaa   '
>>> s.center(8, 'b')
'bbaaabbb'
>>> s.center(8)
'  aaa   '
```

```
>>> s.center(0)
'aaa'
>>> s.center(-1, 'b')
'aaa'
>>>
```

str.zfill()是用于填充代表数字的字符串的，它总是用0在数字的左侧填充，但如果数字有正负号，则会在“+”或“-”之后填充。该函数的语法为：

```
str.zfill(width)
```

width参数同样用于指定填充后字符串的长度，而当该长度小于或等于原字符串的长度时会返回原字符串本身。下面的例子说明了该函数的作用：

```
>>> '1'.zfill(3)
'001'
>>> '+1'.zfill(3)
'+01'
>>> '-1'.zfill(3)
'-01'
>>> '10.9'.zfill(8)
'000010.9'
>>> '+10.9'.zfill(8)
'+00010.9'
>>> '-10.9'.zfill(8)
'-00010.9'
>>> '18+9j'.zfill(10)
'0000018+9j'
>>> '-18-9j'.zfill(10)
'-000018-9j'
>>>
```

---

str.join()提供了另一种拼接字符串的方法，其语法为：

```
str.join(iterable)
```

其iterable参数必须被传入一个元素都是字符串的可迭代对象。调用该函数属性的字符串本身将被作为分隔符，以拼接被传入可迭代对象中的字符串。下面的例子说明了其用法：

```
>>> '.'.join(['www', 'example', 'com'])
'www.example.com'
>>> '@'.join('abc')
'a@b@c'
>>> ''.join(('x', 'y', 'z'))
'xyz'
>>> 'xor'.join(['1011', '1010', '0011'])
'1011xor1010xor0011'
```

```
>>>
```

str.split()和str.rsplit()则提供了与str.join()相反的操作，即基于给定分隔符将字符串拆分成一个列表。它们的语法为：

```
str.split(sep=None, maxsplit=-1)
str.rsplit(sep=None, maxsplit=-1)
```

其中sep参数应被传入一个作为分隔符的字符串，如果传入None则以连续的空格作为分隔符（不论连续的空格有几个）；而maxsplit参数用于限制最大拆分次数，应被传入一个非负整数，取默认实参值-1时表示不限制。仅当拆分次数存在最大值时，str.split()和str.rsplit()才有区别，前者从字符串的开头向末尾搜索分隔符，而后者从字符串的末尾向开头搜索分隔符。请看下面的例子：

```
>>> 'www.example.com'.split('.')
['www', 'example', 'com']
>>> 'www.example.com'.split('.', 1)
['www', 'example.com']
>>> 'www.example.com'.rsplit('.', 1)
['www.example', 'com']
>>> '1011xor1010xor0011'.split('xor')
['1011', '1010', '0011']
>>> 'a b  c    d'.split()
['a', 'b', 'c', 'd']
>>> ''.split('abc')
['']
>>> ''.split()
[]
>>>
```

注意上面的例子说明空串被拆分时，如果指定了分隔符则会得到包含一个空串的列表，否则得到一个空列表。

str.partition()和str.rpartition()基于给定的分隔符将字符串划分成三部分，且在返回的列表中包含分隔符本身。它们的语法为：

```
str.partition(sep)
str.rpartition(sep)
```

注意这两个函数总是返回三元组。如果分隔符位于字符串的开头/末尾，则三元组的第一个/第三个元素是空串。如果字符串中不包含该分隔符，则三元组的第一个元素是该字符串本身，后两个元素都是空串。这两个函数的区别是，str.partition()将从字符串的开头向末尾搜索分隔符，而str.rpartition()将从字符串的末尾向开头搜索分隔符。下面的例子说明了它们的作用：

```
>>> 'www.example.com'.partition('.')
('www', '.', 'example.com')
>>> 'www.example.com'.rpartition('.')
('www.example', '.', 'com')
>>> 'www.example.com'.partition('com')
('www.example.', 'com', '')
>>> 'www.example.com'.partition('www')
('', 'www', '.example.com')
>>> 'www.example.com'.partition('net')
('www.example.com', '', '')
>>>
```

str.splitlines()则将包含EOL的字符串分解为多行，其语法为：

```
str.splitlines(keepends=False)
```

该函数返回一个列表，列表中的每个元素对应一行，而keepends参数用于控制是否保留EOL。注意str.splitlines()会将表10-3中列出的字符（组合）都视为EOL，这是通用换行的超集。

表10-3. str.splitlines()视为EOL的字符（组合）

字符（组合）	说明
\n	换行。
\r	回车。
\r\n	回车+换行。
\v	垂直制表符。
\f	分页。
\x1c	文件分隔符。
\x1d	组分隔符。
\x1e	记录分隔符。
\x85	下一行（C1控制码）
\u2028	行分隔符。
\u2029	段分隔符。

下面的例子说明了该函数属性的用法：

```
>>> 'abc\ndef\nghi\n'.splitlines(True)
['abc\n', 'def\n', 'ghi\n']
>>> ''.splitlines()
[]
```

```
>>> '\n\n'.splitlines()
['', '']
>>>
```

---

str.count()的语法为：

```
str.count(sub, start=0, end=None)
```

它会从字符串的第start个字符开始（包括该字符）至第end个字符截止（不包括该字符），搜索通过sub参数传入的字符串，并对匹配的次数计数（各匹配处不可重叠）。特别的，当end取默认实参值None时，表示从第start个字符开始搜索直至最后一个字符（包括该字符）。该函数返回一个整数。下面的例子说明了该函数属性的作用：

```
>>> 'To be or not to be'.count('e')
2
>>> 'To be or not to be'.count('e', 0, None)
2
>>> 'To be or not to be'.count('e', 0, 3)
0
>>> 'To be or not to be'.count('e', 0, -1)
1
>>> 'To be or not to be'.count('be')
2
>>> 'To be or not to be'.count('be', 6)
1
>>> 'To be or not to be'.count('be', 0, 5)
1
>>> 'aaa'.count('aa')
1
>>>
```

str.find()和str.rfind()的语法为：

```
str.find(sub, start=0, end=None)
str.rfind(sub, start=0, end=None)
```

它们的执行方式与str.count()基本一致，但会在找到第一处匹配时停止，然后返回匹配处首字符的索引。如果没有找到匹配处，则返回-1。两者的区别是，str.find()从字符串开头向结尾查找，str.rfind()从字符串结尾向开头查找。下面的例子说明了它们的作用：

```
>>> 'To be or not to be'.find('e')
4
>>> 'To be or not to be'.rfind('e')
17
>>> 'To be or not to be'.find('e', 5, None)
17
```

```
>>> 'To be or not to be'.rfind('e', 0, -1)
4
>>> 'To be or not to be'.find('e', 5, -1)
-1
>>> 'To be or not to be'.rfind('e', 5, -1)
-1
>>>
```

str.index()和str.rindex()的语法为：

```
str.index(sub, start=0, end=None)
str.rindex(sub, start=0, end=None)
```

它们与str.find()和str.rfind()的区别仅在于当没有找到匹配处时会抛出ValueError异常。下面的例子说明了它们的作用：

```
>>> 'To be or not to be'.index('e')
4
>>> 'To be or not to be'.rindex('e')
17
>>> 'To be or not to be'.index('e', 5, None)
17
>>> 'To be or not to be'.rindex('e', 0, -1)
4
>>> 'To be or not to be'.index('e', 5, -1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
>>> 'To be or not to be'.rindex('e', 5, -1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
>>>
```

str.startswith()和str.endswith()用于判断一个字符串是否以某个子串开头/结尾，其语法为：

```
str.startswith(prefix, start=0, end=None)
str.endswith(suffix, start=0, end=None)
```

其中prefix参数和suffix参数可以被传入一个字符串，也可以被传入一个字符串元组（表示取其中任意一个字符串）。如果给出了start参数，则从该位置开始比较（包括该字符）；如果给出了end参数，则在该位置停止比较（不包括该字符）。这两个函数总是返回布尔值。下面的例子说明了它们的用法：

```
>>> 'www.example.com'.startswith('www')
True
>>> 'www.example.com'.startswith('com')
```

```
False
>>> 'www.example.com'.startswith(('com','www'))
True
>>> 'www.example.com'.endswith('www')
False
>>> 'www.example.com'.endswith('com')
True
>>> 'www.example.com'.endswith(('com', 'www'))
True
>>> 'www.example.com'.startswith('example', 3)
False
>>> 'www.example.com'.startswith('example', 4)
True
>>> 'www.example.com'.startswith('example', 4, 11)
True
>>> 'www.example.com'.startswith('example', 4, 10)
False
>>> 'www.example.com'.endswith('example', 4)
False
>>> 'www.example.com'.endswith('example', 4, 11)
True
>>>
```

---

str.isupper()、str.islower()和str.istitle()对字符串的大小写状态进行判断，规则是：

➤ 仅当字符串中至少包含一个区分大小写的字符，且此类字符均大写时，str.isupper()才返回True。
➤ 仅当字符串中至少包含一个区分大小写的字符，且此类字符均小写时，str.islower()才返回True。
➤ 仅当字符串中的每个单词的首字符均大写，其余字符均小写时，str.istitle()才返回True。

请看下面的例子：

```
>>> 'EINE STRASSE MIT VIER BAHNEN'.isupper()
True
>>> 'eine straße mit vier bahnen'.islower()
True
>>> 'Eine Straße Mit Vier Bahnen'.istitle()
True
>>> 'Eine Strasse Mit Vier Bahnen'.istitle()
True
>>>
```

str.isidentifier()用于判断一个字符串是否可作为标识符。请看下面的例子：

```
>>> 'a'.isidentifier()
True
>>> '_a'.isidentifier()
```

```
True
>>> '_a8'.isidentifier()
True
>>> '8a_'.isidentifier()
False
>>> '%'.isidentifier()
False
>>> 'class'.isidentifier()
True
>>> 'int'.isidentifier()
True
>>> 'or'.isidentifier()
True
>>>
```

该例子说明str.identifier()只判断该字符串是否满足标识符的语法要求，无法识别关键字和运算符。

下列函数属性以字符串中包含的字符的类型作为判断依据：

➤ str.isprintable()：字符串为空串，或所有字符都不属于Unicode的Other类和Separator类时返回True。

➤ str.isascii()：字符串为空串，或所有字符都是ASCII字符时返回True。

➤ str.isspace()：字符串非空，且所有字符都属于Unicode的Zs类、WS类、B类或S类时返回True。

➤ str.isalnum()：字符串非空，且所有字符要么是字母，要么是数字时返回True。它等价于表达式“str.isalpha() or str.isnumeric()”。

➤ str.isalpha()：字符串非空，且所有字符都属于Unicode的Letter类（具有通用类别属性“Lm”、“Lt”、“Lu”、“Ll”或“Lo”之一）时返回True。

➤ str.isnumeric()：字符串非空，且所有字符都具有Unicode特征属性值Numeric_Type = Numeric、Numeric_Type = Digit或Numeric_Type = Decimal时返回True。

➤ str.isdigit()：字符串非空，且所有字符都具有Unicode特征属性值Numeric_Type = Digit或Numeric_Type = Decimal时返回True。

➤ str.isdecimal()：字符串非空，且所有字符都具有Unicode特征属性值Numeric_Type = Decimal时返回True。

值得特别说明的是，str.isnumeric()、str.isdigit()和str.isdecimal()指定的字符范围依次变小。下面的例子说明了这点：

```
>>> '0'.isnumeric()
True
>>> '0'.isdigit()
True
>>> '0'.isdecimal()
True
>>> '2⁴'.isnumeric()
True
>>> '2⁴'.isdigit()
True
>>> '2⁴'.isdecimal()
False
>>> '½'.isnumeric()
True
>>> '½'.isdigit()
False
>>> '½'.isdecimal()
False
>>> '一二三'.isnumeric()
True
>>> '一二三'.isdigit()
False
>>> '一二三'.isdecimal()
False
>>>
```

最后，字符串之间相互比较时，不论这些字符串采用的编码方式是什么，都会根据每个字符的Unicode码位按照字典顺序进行比较。下面是一些例子：

```
>>> 'a' < 'abc'
True
>>> 'a' < 'bcd'
True
>>> 'b' < 'abc'
False
>>> 'b' < 'bcd'
True
>>> 'abcd' > 'xy'
False
>>>
```

10-3. 格式化字符串

（教程：7.1）
（语言参考手册：2.4.3、3.3.1）
（标准库：内置函数、内置类型）

在Python中，“格式化字符串（formatted strings）”是呈现信息的重要技巧。概括地说，格式化字符串使我们可以将变量嵌入字符串，进而能够靠给变量赋不同值而生成不同的字符串；而每个变量被赋予的值在显示时的格式也可以通过格式化字符串控制。

格式化字符串的基础是__format__魔术属性。第3章已经说明object实现了该魔术属性，因此所有对象都具有__format__。

__format__的作用是返回一个描述该对象的字符串，与__repr__和__str__的区别在于，调用__format__时必须传入一个描述该字符串格式的字符串，被称为“格式规格（format specification）”。很多内置类型都重写了__format__，但它们以及object.__format__的格式规格都使用“格式规格迷你语言（format specification mini-language）”。

内置函数format()是使用__format__魔术属性的基本方式，其语法为：

```
format(object, format_spec='')
```

这等价于调用***object.__format__(format_spec)***。format_spec参数用于传入格式规格，默认传入空串，而在格式规格迷你语言中空串的含义是通过调用__str__来显示该对象。下面的例子说明这点：

```
>>> str(True)
'True'
>>> format(True)
'True'
>>>
>>> str(0x2a)
'42'
>>> format(0x2a)
'42'
>>>
>>> str(16.3)
'16.3'
>>> format(16.3)
'16.3'
>>>
>>> str(9.2+6.0j)
'(9.2+6j)'
>>> format(9.2+6.0j)
'(9.2+6j)'
>>>
>>> str('abc')
'abc'
>>> format('abc')
'abc'
>>>
```

事实上，对于布尔值、数字和字符串之外的对象，可用的格式规格通常只有空串，例如：

```
>>> str([1, 2, 'a'])
"[1, 2, 'a']"
>>> format([1, 2, 'a'])
"[1, 2, 'a']"
>>>
```

空串之外的格式规格必须具有如下格式：

```
   [[fill]align][sign][#][0][width][grouping_option][.precision]
[type]
```

下面逐步介绍该格式中各字段的含义。

虽然格式规格中的各字段都是可选的，但type字段是最基础的，表明了该对象应以什么方式呈现。当该对象是字符串时，type字段只能取值“s”，即以字符串格式显示，因此可以省略。请看下面的例子：

```
>>> str('abc')
'abc'
>>> format('abc', 's')
'abc'
>>> format('abc')
'abc'
>>>
```

当对象是整数时，type字段可以取表10-4中列出的字符，“d”可以被省略。

表10-4. type字段对于整数可取的字符

字符	含义
'd'	以十进制显示。
'b'	以二进制显示，默认不添加“0b”前缀。
'o'	以八进制显示，默认不添加“0o”前缀。
'x'	以十六进制显示，用a~f表示10以上数字，默认不添加“0x”前缀。
'X'	以十六进制显示，用A~F表示10以上数字，默认不添加“0X”前缀。
'n'	以十进制显示，但根据当前区域设置插入适当的数字分隔符。
'c'	显示以该整数作为Unicode码位的字符，整数范围是0~0x110000。

请看下面的例子：

```
>>> format(365)
'365'
>>> format(-365)
'-365'
>>>
>>> format(365, 'd')
'365'
>>> format(-365, 'd')
'-365'
>>>
>>> format(365, 'b')
'101101101'
>>> format(-365, 'b')
'-101101101'
>>>
>>> format(365, 'o')
'555'
>>> format(-365, 'o')
'-555'
>>>
>>> format(365, 'x')
'16d'
>>> format(-365, 'x')
'-16d'
>>>
>>> format(365, 'X')
'16D'
>>> format(-365, 'X')
'-16D'
>>>
>>> format(365, 'n')
'365'
>>> format(-365, 'n')
'-365'
>>>
>>> format(365, 'c')
'ŭ'
>>>
```

当对象是浮点数和复数时，type字段可以取表10-5中列出的字符。如果此时省略了type字段，大体相当于默认取值“g”，但当采用定点表示法时小数点后至少显示一位，默认精度也不同。这些字符同样适用于Decimal类型，但效果存在细微差别。如果将这些字符（除了“n”）应用于整数，则会先自动调用float()将该整数转化为浮点数。

表10-5. type字段对于浮点数和复数的可取字符

字符	含义
'g'	将整个数字舍入到通过precision字段指定的有效位，然后再根据结果决定采用定点表示法还是科学计数法描述。定点表示法中的无穷和非数用inf和nan表示。科学计数法中以e代表指数。自动去掉小数点后多余的0。
'G'	与'g'的区别仅在于定点表示法中的无穷和非数用INF和NAN表示。科学计数法中以E代表指数。
'f'	与'g'的区别是总是采用定点表示法，且不会去掉小数点后多余的0。
'F'	与'G'的区别是总是采用定点表示法，且不会去掉小数点后多余的0。
'e'	与'g'的区别是总是采用科学计数法。

字符	含义
'E'	与'G'的区别是总是采用科学计数法。
'n'	与'g'的区别是根据当前区域设置插入适当的数字分隔符。
'%'	与'f'的区别是将最后结果改为百分数。不支持复数。

请看下面的例子：

```
>>> format(987654321.01234)
'987654321.01234'
>>> format(76.8-13.45926j)
'(76.8-13.45926j)'
>>>
>>> format(987654321.01234, 'g')
'9.87654e+08'
>>> format(76.8-13.45926j, 'g')
'76.8-13.4593j'
>>>
>>> format(987654321.01234, 'G')
'9.87654E+08'
>>> format(76.8-13.45926j, 'G')
'76.8-13.4593j'
>>>
>>> format(987654321.01234, 'f')
'987654321.012340'
>>> format(76.8-13.45926j, 'f')
'76.800000-13.459260j'
>>>
>>> format(987654321.01234, 'F')
'987654321.012340'
>>> format(76.8-13.45926j, 'F')
'76.800000-13.459260j'
>>>
>>> format(987654321.01234, 'e')
'9.876543e+08'
>>> format(76.8-13.45926j, 'e')
'7.680000e+01-1.345926e+01j'
>>>
>>> format(987654321.01234, 'E')
'9.876543E+08'
>>> format(76.8-13.45926j, 'E')
'7.680000E+01-1.345926E+01j'
>>>
>>> format(987654321.01234, 'n')
'9.87654e+08'
>>> format(76.8-13.45926j, 'n')
'76.8-13.4593j'
>>>
>>> format(987654321.01234, '%')
'98765432101.233994%'
>>>
```

width字段为一个正整数，表示该字符串的最小长度。如果省略了width字段，则字符串的长度将根据其内容自动计算。

fill字段和align字段用于控制当内容不足以填满width字段指定的最小长度时的处理方式，因此仅当存在width字段时才有意义。align字段指定内容的对齐方式，可以取表10-6列出的某个字符。fill字段则可以是任何一个字符，用来将字符串填充到最小长度。如果省略了fill字段，则默认用空格填充。如果省略了align字段，则当对象是数字时默认右对齐，其他情况下都默认左对齐。

表10-6. align字段可取字符

字符	含义
'<'	内容在可用空间内左对齐。
'>'	内容在可用空间内右对齐。
'^'	内容在可用空间内居中。
'='	填充字符在正负号和数字之间插入。

下面的例子说明了width、fill和align的作用：

```
>>> format('abc', '5')
'abc  '
>>> format('abc', '<5')
'abc  '
>>> format('abc', '>5')
'  abc'
>>> format('abc', '^5')
' abc '
>>>
>>> format('abc', '@<5')
'abc@@'
>>> format('abc', '@>5')
'@@abc'
>>> format('abc', '@^5')
'@abc@'
>>>
>>> format(-12.2, '9')
'    -12.2'
>>> format(-12.2, '<9')
'-12.2    '
>>> format(-12.2, '>9')
'    -12.2'
>>> format(-12.2, '^9')
'  -12.2  '
>>> format(-12.2, '=9')
'-    12.2'
>>>
>>> format(-12.2, '0<9')
'-12.20000'
>>> format(-12.2, '0>9')
'0000-12.2'
>>> format(-12.2, '0^9')
'00-12.200'
>>> format(-12.2, '0=9')
'-000012.2'
>>>
```

precision字段取值一个非负整数。当对象是字符串时，precision字段表示该字符串被显示时的最大长度，超过该限制时会从尾部截断，例如：

```
>>> format('abcde', '.3')
'abc'
>>>
```

当对象是浮点数或复数时，precision字段用于控制精度，如果type字段取值“g”或“G”则解读为整个数字的有效数字个数，其他情况下解读为小数点后的有效数字个数，例如：

```
>>> format(1234.0, '.3g')
'1.23e+03'
>>> format(1234.0, '.3f')
'1234.000'
>>> format(1234.0, '.3e')
'1.234e+03'
>>> format(1234.0, '.3%')
'123400.000%'
>>>
```

剩下的字段都仅当对象是数字时才有意义。

sign字段用于控制正负号的显示方式，可取值被总结在表10-7中，如果被省略则该字段默认取“-”。

表10-7. sign字段可取字符

字符	含义
'+'	正数和零前添加“+”，负数前添加“-”。
'-'	仅负数前添加“-”。
空格	正数和零前添加空格，负数前添加“-”。

请看下面的例子：

```
>>> format(125, '+')
'+125'
>>> format(-125, '+')
'-125'
>>> format(125, '-')
'125'
>>> format(-125, '-')
'-125'
>>> format(125, ' ')
' 125'
>>> format(-125, ' ')
```

```
'-125'
>>>
```

“#”表示开启“替换形式（alternate form）”，仅适用于整数、浮点数和复数，如果被添加则遵循如下规则：

➤ 整数：以二进制、八进制和十六进制显示时，分别添加前缀“0b”、“0o”、“0x”或“0X”。
➤ 浮点数和复数：总是包含小数点（即使小数部分为0）；如果type取值“g”或“G”，则小数点后还可能保留若干个0。

请看下面的例子：

```
>>> format(3487, '#b')
'0b110110011111'
>>> format(3487, '#o')
'0o6637'
>>> format(3487, '#x')
'0xd9f'
>>> format(3487, '#X')
'0XD9F'
>>> format(3487, '#g')
'3487.00'
>>>
```

“0”其实是一个快捷方式，等价于align字段取“=”且fill字段取“0”，例如：

```
>>> format(-12.2, '09')
'-000012.2'
>>> format(-12.2, '0=9')
'-000012.2'
>>>
```

grouping_option字段仅当type字段取值“d”时才有意义。该字段的可取值是“,”或“_”，代表的是以十进制显示整数时被插入的数字分隔符，注意这与区域设置没有关系。下面的例子说明了其用法：

```
>>> format(3487, ',d')
'3,487'
>>> format(9237, '_d')
'9_237'
>>> format(-23189.23899, ',')
'-23,189.23899'
>>> format(-46978+12345789.0j, '_')
'(-46_978+12_345_789j)'
>>>
```

至此我们讨论完了格式规格迷你语言。然而当我们自定义一个类时，可以通过重写魔术属性__format__来自定义我们自己的格式规格描述语言。下面给出一个极简单的例子：

```
>>> class C:
...     def __init__(self, s):
...         self.s = s
...     def __format__(self, format_spec=''):
...         return self.s.join([format_spec, format_spec])
...
>>> obj = C('abc')
>>> format(obj)
'abc'
>>> format(obj, 'x')
'xabcx'
>>> format(obj, '@')
'@abc@'
>>>
```

在该例子中，自定义类C通过s属性储存一个字符串，而调用它的__format__时，传入的字符串会被作为s属性储存的字符串的前缀和后缀以完成格式化。显然，我们可以通过增加__format__的复杂性来丰富我们自定义的格式规格描述语言的语法，但本书不对此进一步讨论。

在明确了__format__魔术属性的作用后，我们可以开始讨论格式化字符串本身了。我们有两种使用格式化字符串的方式，一种是调用str.format()或str.format_map()，另一种是使用格式化字符串字面值。

不论哪种方法，都允许在字符串中通过如下格式嵌入一个变量：

```
{[field_name][!conversion][:format_spec]}
```

这种格式被称为“替换字段（replacement fields）”，以大括号作为分隔符。一个字符串可以包含任意多个替换字段。如果需要在字符串内表示大括号本身，则需要用“{{”来代表“{”，用“}}”来代表“}”。

替换字段的内容可以通过“!”和“:”被分解为如下三部分：

➤ field_name：用于定位到该变量。当取值一个非负整数时，表示基于位置匹配变量；当取值一个字符串时，表示基于关键字匹配变量。在整数/关键字之后还可跟“.attribute”以指定该变量的某个属性，或者跟“[index]”以指定容器变量的某个元素。如果被省略，则默认取值一个非负整数，为该替换字段在所有省略了field_name的替换字段形成的序列中的索引。

➤ conversion：用于指定如何将赋值给该变量的对象强制转换成字符串。可以取值“s”（使用str()）、“r”（使用repr()）和“a”（使用ascii()）。如果被省略，则默认调用被传入对象的__format__魔术属性。
➤ format_spec：仅当conversion被省略时才有意义，用于提供格式规格。

str.format()的语法为：

```
str.format(*args, **kwargs)
```

其args参数用于给field_name取值非负整数的替换字段赋值，而kwargs参数用于给格式化字符串中field_name取值字符串的替换字段赋值。

下面是一个只基于位置匹配变量的例子：

```
>>> s = '{:0>8b} xor {:0>8b} equals {:0>8b}'
>>> s.format(5, 3, 6)
'00000101 xor 00000011 equals 00000110'
>>> s.format(17, 64, 81)
'00010001 xor 01000000 equals 01010001'
>>>
```

下面是一个只基于关键字匹配变量的例子：

```
>>> s = "{product:<8}{price:>8,.2f}$ per pound"
>>> s.format(product='Apple', price=4.02)
'Apple       4.02$ per pound'
>>> s.format(product='Pear', price=3.951)
'Pear        3.95$ per pound'
>>> s.format(product='Ham', price=11.7)
'Ham        11.70$ per pound'
>>> s.format(product='Watermelon', price=2.0)
'Watermelon    2.00$ per pound'
>>>
```

而下面则是一个同时基于位置匹配变量和基于关键字匹配变量的例子：

```
>>> s = "{character:^5}{:^7d}{:^#7o}{:^#7x}"
>>> s.format(65, 65, 65, character='A')
'  A    65    0o101  0x41  '
>>> s.format(66, 66, 66, character='B')
'  B    66    0o102  0x42  '
>>> s.format(67, 67, 67, character='C')
'  C    67    0o103  0x43  '
>>>
```

从上面的例子可知，结合使用str.format()时，包含替换字段的字符串就相当于一个函数，其内的每个替换字段都会被调用str.format()时的某个实际参数赋值。我们完全可以通过编写普通函数实现同样的功能，但显然不如这种方法简洁。

str.format_map()的语法为：

```
str.format_map(mapping)
```

其中mapping参数必须被传入一个映射（例如字典）。它大体上等价于：

```
str.format(**mapping)
```

因此字符串中的所有替换字段的field_name都必须取值字符串。下面的例子说明了这一点：

```
>>> s = "{product:<8}{price:>8,.2f}$ per pound"
>>> s.format_map({'product':'Apple', 'price':4.02})
'Apple       4.02$ per pound'
>>> s.format_map({'product':'Pear', 'price':3.951})
'Pear        3.95$ per pound'
>>> s.format_map({'product':'Ham', 'price':11.7})
'Ham        11.70$ per pound'
>>> s.format_map({'product':'Watermelon', 'price':2.0})
'Watermelon    2.00$ per pound'
>>>
```

前面已经说明格式化字符串字面值就是具有f/F前缀的字符串字面值。格式化字符串字面值中的所有替换字段的field_name都必须取值一个字符串，且这些字符串都必须是已经被定义的标识符。请看下面的例子：

```
>>> who = 'James'
>>> what = 'football'
>>> where = 'in the garden'
>>> s = f"{who} is playing {what} {where}."
>>> s
'James is playing football in the garden.'
>>> who = 'Lucy'
>>> what = 'golf'
>>> where = 'on the lawn'
>>> s
'James is playing football in the garden.'
>>>
```

注意该例子说明，当一个格式化字符串字面值被创建后，对替换字段的赋值和格式化就已经完成，此后即便相应变量引用的对象发生改变，也不会影响到该格式化字符串字面值。

格式化字符串字面值看起来似乎缺乏灵活性，但考虑到函数的形式参数也是已经被定义的标识符，因此我们可以在函数体中使用格式化字符串。请看下面的例子：

```
>>> def make_sentence(who, what, where):
...     return f"{who} is playing {what} {where}."
...
>>> make_sentence('James', 'football', 'in the garden')
'James is playing football in the garden.'
>>> make_sentence('Lucy', 'golf', 'on the lawn')
'Lucy is playing golf on the lawn.'
>>>
```

最后值得一提的是，作为一种遗留特性，Python 3依然支持“printf风格的字符串格式化”，即以“format % values”的格式定义一个字符串，其中format为包含若干以“%”开头的“转换标记（conversion specifiers）”的字符串，而values则是一个元组，其内的每个元素将基于位置被赋值给转换标记。这种技巧类似于C中的printf()，因此而得名。然而目前这一技巧具有很多怪异特性，可能导致许多常见错误，因此不推荐使用。

10-4. 字节串和字节数组

（语言参考手册：2.4.1、3.3.1、6.10.1）
（标准库：内置函数、内置类型）

第3章已经说明，二进制序列包括字节串、字节数组和内存视图。内存视图是更加特殊的类型，字节串和字节数组则与字符串具有相似性。我们先讨论字节串和字节数组，在本章的最后再讨论内存视图。

字节串可以被视为一种特殊的字符串——它没有采用任何字符编码方式，因此字节序列无法被解读为字符序列。然而，从逻辑上讲，没有字符编码方式本身也可以被视为一种字符编码方式，它编码出的“字符”就是字节（因此是一种一对一映射）。从这个角度，字节串很容易理解，且能对字符串施加的操作大部分也能施加于字节串也就不令人惊讶了。

内置类型bytes表示字节串。可以通过内置函数bytes()来创建字节串。bytes()的第一种语法为：

```
class bytes(object=b'')
```

其功能相当于将通过object参数传入的对象强制类型转换为字节串，具体规则为：

- 判断是否可以将其理解为bytes()的第二种语法，如果是则转向它。
- 如果该对象具有__bytes__魔术属性，则调用它。
- 否则，如果该对象支持“缓冲区协议”（例如字节串和字节数组），则复制该对象。

➤ 否则，如果该对象是个非负整数，则创建一个长度为该非负整数的字节串，每个字节都用\0填充。
➤ 否则，如果该对象是个由0~255范围内的整数形成的可迭代对象，则创建一个长度等于该对象中元素个数的字节串，每个字节的编码依次等于该对象中的元素。
➤ 如果上述条件都不能满足，则抛出TypeError或ValueError异常。

特别的，若省略了object参数，则得到空字节串。

bytes()的第二种语法为：

```
class bytes(string, encoding, errors='strict')
```

string参数必须是一个字符串。encoding参数指定该字符串的编码方式；errors参数用于指定当遇到指定编码方式无法解读的编码时如何处理。后两个参数共同确定了该字符串对应哪个字节串，而这就是被返回的字节串。

没有任何内置类型实现了__bytes__，因此我们必须自己定义。下面的例子同时说明了__bytes__的作用和bytes()的第二种语法：

```
>>> class C:
...     def __init__(self, pre, suf):
...         self.prefix = pre
...         self.suffix = suf
...     def __bytes__(self):
...         return bytes(self.prefix + '***' + self.suffix, 'utf8')
...
>>> obj1 = C('abc', 'xyz')
>>> obj2 = C('I87', '3V')
>>> bytes(obj1)
b'abc***xyz'
>>> bytes(obj2)
b'I87***3V'
>>>
```

而下面的这些例子则说明了通过bytes()创建字节串的其他情况：

```
>>> b1 = bytes(8)
>>> b1
b'\x00\x00\x00\x00\x00\x00\x00\x00'
>>> b2 = bytes([1,0,34,252])
>>> b2
b'\x01\x00"\xfc'
>>> b3 = bytes(b1)
>>> b3
b'\x00\x00\x00\x00\x00\x00\x00\x00'
>>> b4 = bytes(b2)
>>> b4
b'\x01\x00"\xfc'
```

```
>>>
```

类似于字符串字面值，也存在字节串字面值。字节串字面值与字符串字面值的区别有如下三点：

➤ 字节串字面值必须具有前缀b/B。
➤ 字节串字面值的可选前缀是r/R，不支持u/U和f/F。
➤ 不论是短字节串字面值还是长字节串字面值，引号内只能出现ASCII字符，且换行、回车、双引号和单引号都必须用转义序列表示（即\n、\r、\"和\'），不可打印的ASCII字符也必须用转义序列表示（即\xhh）。显然，码位超出ASCII范围（即128~255）的字节只能用转义序列表示（即\xhh）。

注意在前面的例子中已经多次出现过了字节串字面值。

相邻短字节串字面值也会被自动合并。请看下面的例子：

```
>>> b'H' B'e' b'll' B'o'
b'Hello'
>>> b"Hello" b' world!'
b'Hello world!'
>>> b'\n' b'\x00'
b'\n\x00'
>>>
```

这一机制同样是在词法分析阶段实现的。

就像字符串那样，字节串是不可变的。而字节数组与字节串的唯一区别就是——字节数组是可变的。换句话说，我们可以改变字节数组中某个字节的值并保持其他字节不变，而这不会导致创建新的对象。这一特性与C中的数组相同，因此这种类型得名字节数组。

内置类型bytearray表示字节数组。字节数组只能通过内置函数bytearray()创建，不存在对应的字面值。bytearray()的第一种语法为：

```
class bytearray(object=b'')
```

除了返回的是字节数组，且与__bytes__魔术属性无关外，bytearray()的这种语法与bytes()的第一种语法没有区别。

bytearray()的第二种语法为：

```
class bytearray(string, encoding, errors='strict')
```

除了返回的是字节串数组外，bytearray()的这种语法与bytes()的第二种语法没有区别。

下面是一个字节数组的例子：

```
>>> b1 = bytearray([1, 2, 3, 4])
>>> b1
bytearray(b'\x01\x02\x03\x04')
>>> b1[2] = 5
>>> b1
bytearray(b'\x01\x02\x05\x04')
>>>
```

该例子通过下标修改了字节数组中的第3个字节，以说明它是可变的。这其实利用了字节数组是一种可变序列的事实，会在第11章进一步讨论。

除了需要利用可变性的操作之外，字节串和字节数组支持的操作是相同的，且它们可以混合。

例如，我们可以像拼接字符串那样利用“+”和“*”拼接字节串和/或字节数组。请看下面的例子：

```
>>> b'o' + b'p' + b'en'
b'open'
>>> 3 * b'\n'
b'\n\n\n'
>>> bytearray([1, 2]) + bytearray([3, 4])
bytearray(b'\x01\x02\x03\x04')
>>> bytearray(b'0') * 8
bytearray(b'00000000')
>>> b'34' + bytearray(b'abc')
b'34abc'
>>>
```

注意该例子说明，当拼接字节串和字节数组时，最终得到的是字节数组。

显然，字节串和字节数组也支持“+=”和“*=”，例如：

```
>>> b = b'16'
>>> ba = bytearray(b'09@')
>>> b += b'8'
```

```
>>> b
b'168'
>>> ba += bytearray(b'#')
>>> ba
bytearray(b'09@#')
>>> b *= 2
>>> b
b'168168'
>>> ba *= 3
>>> ba
bytearray(b'09@#09@#09@#')
>>>
```

字节串和/或字节数组相互比较时，会根据每个字节的值按字典排序算法进行比较。下面是一些例子：

```
>>> b'abc' > b'cd'
False
>>> b'x' > b'cd'
True
>>> b'x' > bytearray(b'A')
True
>>> bytearray([1, 2, 3]) < bytearray([4, 0])
True
>>> bytearray(b'word') == b'word'
True
>>>
```

表10-8列出了bytes和bytearray共同支持的函数属性。将它与表10-2进行比较，就会发现这些属性与str的函数属性大体相同。事实上，当调用这些属性时，是将字节串和字节数组视为采用ASCII编码的字符串来处理的（超出ASCII编码范围的字节则保留不变）。下面只讨论bytes和bytearray不同于str的地方。

表10-8. bytes和bytearray属性

范畴	属性	说明
编码	**bytes.decode()** **bytearray.decode()**	返回字节串/字节数组在指定编码方式下对应的字符串。
	bytes.hex() **bytearray.hex()**	将字节串/字节数组转换为相应十六进制字符串表示法。
	bytes.fromhex() **bytearray.hex()**	将十六进制字符串表示法转换为相应字节串/字节数组。
大小写	**bytes.upper()** **bytearray.upper()**	返回字节串/字节数组的所有对应ASCII字符均大写的副本。
	bytes.lower() **bytearray.lower()**	返回字节串/字节数组的所有对应ASCII字符均小写的副本。
	bytes.swapcase() **bytearray.swapcase()**	返回字节串/字节数组的所有对应ASCII字符均反转大小写的副本。
	bytes.capitalize() **bytearray.capitalize()**	返回字节串/字节数组的首字节对应ASCII字符大写、其余对应ASCII字符均小写的副本。

范畴	属性	说明
	bytes.title() **bytearray.title()**	返回字节串/字节数组的每个位于单词词首的对应ASCII字符大写、其余对应ASCII字符均小写的副本。
替换	**bytes.replace()** **bytearray.replace()**	返回将字节串/字节数组的指定子字节串/字节数组替换为另一个子字节串/字节数组的副本。
	bytes.maketrans() **bytearray.maketrans()**	静态方法，返回一个供bytes.translate()或bytearray.translate()使用的转换对照表。
	bytes.translate() **bytearray.translate()**	返回将字节串/字节数组按照转换对照表进行一对一替换的副本。
	bytes.expandtabs() **bytearray.expandtabs()**	返回将字节串/字节数组中的对应水平制表符的字节替换为对应空格的字节的副本。
去掉前后缀	**bytes.strip()** **bytearray.strip()**	返回去掉了字节串/字节数组开头和末尾的对应指定ASCII字符的字节的副本。
	bytes.lstrip() **bytearray.lstrip()**	返回去掉了字节串/字节数组开头的对应指定ASCII字符的字节的副本。
	bytes.rstrip() **bytearray.rstrip()**	返回去掉了字节串/字节数组末尾的对应指定ASCII字符的字节的副本。
	bytes.removeprefix() **bytearray.removeprefix()**	返回去掉了字节串/字节数组开头的指定子字节串/字节数组的副本。
	bytes.removesuffix() **bytearray.removesuffix()**	返回去掉了字节串/字节数组末尾的指定子字节串/字节数组的副本。
填充	**bytes.ljust()** **bytearray.ljust()**	返回字节串/字节数组在末尾用对应指定ASCII字符的字节填充到指定长度的副本。
	bytes.rjust() **bytearray.rjust()**	返回字节串/字节数组在开头用对应指定ASCII字符的字节填充到指定长度的副本。
	bytes.center() **bytearray.center()**	返回字节串/字节数组在开头和末尾用对应指定ASCII字符的字节填充到指定长度的副本。
	bytes.zfill() **bytearray.zfill()**	返回代表数字的字节串/字节数组在开头用0填充到指定长度的副本。
合并与拆分	**bytes.join()** **bytearray.join()**	以字节串/字节数组为分隔符将多个其他字节串/字节数组拼接起来。
	bytes.split() **bytearray.split()**	以指定子字节串/字节数组作为分隔符将字节串/字节数组分解，正向搜索。
	bytes.rsplit() **bytearray.rsplit()**	以指定子字节串/字节数组作为分隔符将字节串/字节数组分解，反向搜索。
	bytes.partition() **bytearray.partition()**	以指定子字节串/字节数组作为分隔符将字节串/字节数组分为3部分，正向搜索。
	bytes.rpartition() **bytearray.rpartition()**	以指定子字节串/字节数组作为分隔符将字节串/字节数组分为3部分，反向搜索。
	bytes.splitlines() **bytearray.splitlines()**	将字节串/字节数组按行分解。
子串	**bytes.count()** **bytearray.count()**	返回指定子字节串/字节数组在字节串/字节数组指定区域内的非重叠出现次数。
	bytes.find() **bytearray.find()**	返回指定子字节串/字节数组在字节串/字节数组指定区域内首次出现的位置，正向搜索。
	bytes.rfind() **bytearray.rfind()**	返回指定子字节串/字节数组在字节串/字节数组指定区域内首次出现的位置，反向搜索。
	bytes.index() **bytearray.index()**	与bytes.find()和bytearray.find()的区别仅在于未找到子字节串/字节数组时抛出ValueError异常。

范畴	属性	说明
	bytes.rindex() **bytearray.rindex()**	与bytes.rfind()和bytearray.rfind()的区别仅在于未找到子字节串/字节数组时抛出ValueError异常。
	bytes.startswith() **bytearray.startswith()**	判断字节串/字节数组是否以指定子字节串/字节数组开头。
	bytes.endswith() **bytearray.endswith()**	判断字节串/字节数组是否以指定子字节串/字节数组结尾。
判断	**bytes.isupper()** **bytearray.isupper()**	判断字节串/字节数组是否所有对应ASCII字符均大写。
	bytes.islower() **bytearray.islower()**	判断字节串/字节数组是否所有对应ASCII字符均小写。
	bytes.istitle() **bytearray.istitle()**	判断字节串/字节数组是否每个位于单词词首的对应ASCII字符大写、其余对应ASCII字符均小写。
	bytes.isascii() **bytearray.isascii()**	判断字节串/字节数组是否每个字节都对应ASCII字符。
	bytes.isspace() **bytearray.isspace()**	判断字节串/字节数组是否每个字节都对应ASCII空白符（即空格、\t、\n、\r、\f、\x0b）。
	bytes.isalnum() **bytearray.isalnum()**	判断字节串/字节数组是否每个字节都对应ASCII字母或ASCII数字。
	bytes.isalpha() **bytearray.isalpha()**	判断字节串/字节数组是否每个字节都对应ASCII字母（即a~z和A~Z）。
	bytes.isdigit() **bytearray.isdigit()**	判断字节串/字节数组是否每个字节都对应ASCII数字（即0~9）。

首先，由于不涉及Unicode编码，所以bytes和bytearray没有对应str.casefold()、str.isprintable()、str.isnumeric()和str.isdecimal()的函数属性。（判断字节串/字节数组中是否只包含数字只有bytes.isdigit()/bytearray.isdigit()一种方法。）

第二，由于只有字符串才能作为标识符，所以bytes和bytearray没有对应str.isidentifier()的函数属性。

第三，字节串/字节数组不支持格式化，所以bytes和bytearray没有对应str.format()和str.format_map()的函数属性。（其实字节串也支持“printf风格的字节串格式化”，但该技巧不推荐使用。）

第四，字节串/字节数组的函数属性bytes.decode()/bytearray.decode()与str.encode()正好形成逆操作，其语法为：

```
bytes.decode(encoding='utf-8', errors='strict')
bytearray.decode(encoding='utf-8', errors='strict')
```

请看下面的例子：

```
>>> b = '再见'.encode('gb2312')
>>> b
b'\xd4\xd9\xbc\xfb'
>>> ba = bytearray(b)
>>> ba
bytearray(b'\xd4\xd9\xbc\xfb')
>>> b.decode('ascii', 'backslashreplace')
'\\xd4\\xd9\\xbc\\xfb'
>>> ba.decode('ascii', 'backslashreplace')
'\\xd4\\xd9\\xbc\\xfb'
>>> b.decode('utf-8', 'backslashreplace')
'\\xd4ّ\\xfb'
>>> ba.decode('utf-8', 'backslashreplace')
'\\xd4ّ\\xfb'
>>> b.decode('gb2312', 'backslashreplace')
'再见'
>>> ba.decode('gb2312', 'backslashreplace')
'再见'
>>>
```

第五，由于ASCII字符无法覆盖编码128~255的字节，所以用字节串字面值的方式显示一个字节串时大概率会包含\xhh形式的转义序列，导致可读性较低。注意到一个非负整数的十六进制字符串表示法能够更直观地反映一个二进制序列的结构，所以bytes和bytearray都提供了如下函数属性来将一个二进制序列转化为对应非负整数的十六进制字符串表示法：

```
bytes.hex(sep='', bytes_per_sep=0)
bytearray.hex(sep='', bytes_per_sep=0)
```

而sep参数和bytes_per_sep参数都是为了进一步提高可读性的，前者需被传入一个至多包含一个字符的字符串以指定分隔符，后者需被传入一个整数以指定两个分隔符之间的部分对应几个字节（正整数表示从右向左开始分隔，负整数表示从左向右开始分隔，0等价于1）。请看下面的例子：

```
>>> b'abcde'.hex()
'6162636465'
>>> b'abcde'.hex('-')
'61-62-63-64-65'
>>> b'abcde'.hex('_', 2)
'61_6263_6465'
>>> b'abcde'.hex(' ', -3)
'616263 6465'
>>>
```

注意这些函数返回的字符串并没有0x或0X前缀。

而bytes和bytearray都提供了如下类方法来作为bytes.hex()/bytearray.hex()的逆操作：

```
classmethod bytes.fromhex(string)
classmethod bytearray.fromhex(string)
```

string参数需被传入作为某二进制序列的十六进制字符串表示法的字符串，且如果包含分隔符的话只能是空格。请看下面的例子：

```
>>> bytes.fromhex('6162636465')
b'abcde'
>>> bytearray.fromhex('6162636465')
bytearray(b'abcde')
>>> bytes.fromhex('61 62 63 64 65')
b'abcde'
>>> bytearray.fromhex('616263 6465')
bytearray(b'abcde')
>>>
```

10-5. 内存视图

（标准库：内置函数、内置类型、io）

为了讨论内存视图，我们必须先明确什么是第7章中提到过的类字节对象。类字节对象其实就是支持缓冲区协议的对象，例如字节串和字节数组，以及通过标准库中的array模块实现的“数组（arrays）”（本书不详细讨论array模块）。

缓冲区协议的本质是允许通过内存视图来访问一个对象的底层内存结构，也就是所谓的“缓冲区”。缓冲区本质上就是一块内存空间，其内可以储存一个C或Fortran中的数组，也可以储存通过I/O系统调用操作的数据块，不论哪种情况都属于底层数据结构。

内存视图是通过内置函数memoryview()创建的，其语法为：

```
class memoryview(object)
```

其中object参数必须是一个类字节对象，或者另一个内存视图。memoryview具有表10-9列出的非函数属性和表10-10列出的函数属性。

表10-9. memoryview的非函数属性

属性	说明
obj	引用通过该内存视图访问的类字节对象。
nbytes	引用一个整数，为缓冲区对应数组的大小，单位是字节。
readonly	引用一个布尔值，表明该缓冲区是否是只读的。
format	引用一个字符串，为缓冲区中元素的类型。

属性	说明
itemsize	引用一个整数，为缓冲区中元素的大小，单位是字节。
ndim	引用一个整数，为缓冲区对应数组的维度。
shape	引用一个整数元组，以反映缓冲区对应数组的形状。
strides	引用一个整数元组，以反映缓冲区对应数组按前n个维度分解出的子数组的大小，单位是字节。
suboffsets	引用一个整数元组，供PIL风格的数组在内部使用。
c_contiguous	引用一个布尔值，以表明缓冲区是否是C连续的。
f_contiguous	引用一个布尔值，以表明缓冲区是否是Fortran连续的。
contiguous	引用一个布尔值，以表明缓冲区是否是C连续或Fortran连续的。

表10-10. memoryview的函数属性

属性	说明
__eq__()	重写了__eq__，以支持判断两个内存视图是否相等。
tobytes()	将缓冲区中的数据转换为一个字节串返回。
hex()	将缓冲区中的数据转换为一个字节串的十六进制字符串表示法返回。
tolist()	将缓冲区中的数据转换为一个列表返回。
cast()	改变内存视图的格式。
toreadonly()	返回该内存视图的一个只读拷贝。
release()	释放该内存视图。

从表10-9可以看出，内存视图将缓冲区统一抽象为一个数组，不论该缓冲区绑定的对象是哪种类型。除了可以是多维的之外，该数组与通过array模块实现的数组完全相同，其内的每个元素都必须具有相同的类型，也就具有相同的长度。

memoryview的obj、nbytes和readonly这三个属性，回答了关于该内存视图的基本问题：绑定的缓冲区是哪个对象，该缓冲区大小是多少，该缓冲区是否只读（只读意味着不可变）。下面是一个例子：

```
$ python3

>>> m1 = memoryview(b'abc')
>>> m1.obj
b'abc'
>>> m1.nbytes
3
>>> m1.readonly
True
>>> m2 = memoryview(bytearray([1, 2, 3, 4]))
>>> m2.obj
bytearray(b'\x01\x02\x03\x04')
```

```
>>> m2.nbytes
4
>>> m2.readonly
False
>>> m2.obj[0] = 0
>>> m2.obj
bytearray(b'\x00\x02\x03\x04')
>>>
```

该例子创建了两个内存视图，m1用于访问一个字节串，因此相应缓冲区是只读的；m2用于访问一个字节数组，因此相应缓冲区是可读可写的。

memoryview的format属性说明了被抽象出的数组中元素的类型，可以引用表10-11列出的类型码（这与array模块使用的类型码相同）；itemsize属性则说明了该数组中元素的长度，与format属性具有对应关系。

表10-11. memoryview.format可引用的类型码

类型码	对应的C类型	元素长度（单位：字节）	
		32位机器	64位机器
'b'	signed char	1	1
'B'	unsigned char	1	1
'u'	wchar_t	2	2
'h'	signed short	2	2
'H'	unsigned short	2	2
'i'	signed int	2	4
'I'	unsigned int	2	4
'l'	signed long	4	8
'L'	unsigned long	4	8
'q'	signed long long	8	8
'Q'	unsigned long long	8	8
'f'	float	4	4
'd'	double	8	8

继续上面的例子（注意这里用到了array模块定义的array类型）：

```
>>> m1.format
'B'
>>> m1.itemsize
1
>>> m2.format
'B'
>>> m2.itemsize
```

```
1
>>> import array
>>> m3 = memoryview(array.array('h', [-107, 83]))
>>> m3.format
'h'
>>> m3.itemsize
2
>>> m4 = memoryview(array.array('I', (65535,)))
>>> m4.format
'I'
>>> m4.itemsize
4
>>> m5 = memoryview(array.array('l', [4396, -20003231, 576]))
>>> m5.format
'l'
>>> m5.itemsize
8
>>> m6 = memoryview(array.array('f', [0.25]))
>>> m6.format
'f'
>>> m6.itemsize
4
>>> m7 = memoryview(array.array('d', (-0.001, 0, 1e-12)))
>>> m7.format
'd'
>>> m7.itemsize
8
>>>
```

memoryview的ndim、shape和strides属性描述了被抽象出的数组的形状，特别当该数组是多维时更加有效。然而访问字节串、字节数组和array模块实现数组的内存视图只能抽象出一维数组，这意味着ndim引用1，shape引用仅1个元素的元组（该元素代表数组第1个维度的索引数），strides引用元组(1,)（因为该数组只能基于一个维度分解，分解后的子数组都只包含一个元素）。请继续上面的例子：

```
>>> m1.ndim
1
>>> m1.shape
(3,)
>>> m1.strides
(1,)
>>> m2.ndim
1
>>> m2.shape
(4,)
>>> m2.strides
(1,)
>>>
```

虽然内存视图总是将缓冲区抽象为一个数组，但该缓冲区实际上可能分布于内存空间中的多处。memoryview的c_contiguous、f_contiguous和contiguous属性用于说明该缓冲区是否连续。c_contiguous引用True表明该缓冲区符合C数组的格式，f_contiguous引用True表明该缓冲区符合Fortran数组的格式，两者仅在储存多维数组时有差别。contiguous等价于“c_contiguous or f_contiguous”。请继续上面的例子：

```
>>> m1.c_contiguous
True
>>> m1.f_contiguous
True
>>> m1.contiguous
True
>>>
```

接下来讨论memoryview的函数属性。首先，它重写了魔术属性__eq__，使得两个内存视图相等当且仅当它们同时满足如下两个条件：

- 抽象出的数组结构相同。
- 抽象出的数组的每对元素按照其类型转化为值后，两个值按照C中==运算判断为相等。

此外，一个内存视图还可以与一个类字节对象比较，两个类字节对象之间也可以比较，此时会自动在内部为类字节对象创建临时的内存视图，然后基于上述规则进行比较。请继续上面的例子：

```
>>> a = array.array('d', [1.0, 2.0, 3.0, 4.0, 5.0])
>>> m8 = memoryview(a)
>>> b = array.array('I', [1, 2, 3, 4, 5])
>>> m9 = memoryview(b)
>>> c = array.array('l', [1, 2, 3, 4, 5])
>>> m10 = memoryview(c)
>>> a == b == c == m8 == m9 == m10
True
>>>
```

memoryview的tobytes()、hex()和tolist()都用于查看缓冲区的内容。

memoryview.tobytes()的语法是：

```
memoryview.tobytes(order=None)
```

其order参数可传入如下字符串：

- “C”：将缓冲区转化为一个C数组，然后再转化为字节串。
- “F”：将缓冲区抽转化为一个Fortran数组，然后再转化为字节串。
- “A”：如果缓冲区是连续的，则直接将其转化为字节串；否则同“C”。

而当给order参数传入None时，等价于传入“C”。显然，仅当内存视图抽象出的数组是多维数组时order参数才有意义。调用memoryview.tobytes()时，等价于以该内存视图为参数调用bytes()。

memoryview.hex()的语法为：

```
memoryview.hex(sep='', bytes_per_sep=0)
```

它的效果等价于于先以该内存视图为参数调用bytes()，然后再调用返回的字节串的hex()属性。

memoryview.tolist()的语法为：

```
memoryview.tolist()
```

它将抽象出的数组中的元素依次提取出，然后转换为一个列表。这是查看缓冲区内容最直观的方式。

请继续上面的例子：

```
>>> bytes(m1)
b'abc'
>>> m1.tobytes()
b'abc'
>>> m1.hex()
'616263'
>>> m1.tolist()
[97, 98, 99]
>>>
```

memoryview.cast()的语法为：

```
memoryview.cast(format[, shape])
```

它将返回绑定到同一缓冲区的另一个内存视图，其抽象出的数组中的元素类型由format参数决定，该参数只能被传入“B”、“b”或“c”；而数组的结构则由shape参数决定，该参数需被传入一个整数构成的序列，分别描述该数组的每个维度可被分解出几个子数组。如果shape参数被省略，则保持原结构不变。请继续上面的例子：

```
>>> m11 = memoryview(array.array('d', [13.5, 6.0, -1.2]))
>>> m12 = m11.cast('c', [4, 6])
>>> m11.tobytes()
b'\x00\x00\x00\x00\x00\x00+@\x00\x00\x00\x00\x00\x00\x18@333333\xf3\xbf'
```

```
>>> m12.tobytes()
b'\x00\x00\x00\x00\x00\x00+@\x00\x00\x00\x00\x00\x00\x18@333333\xf3\xbf'
>>> m11.tolist()
[13.5, 6.0, -1.2]
>>> m12.tolist()
[[b'\x00', b'\x00', b'\x00', b'\x00', b'\x00', b'\x00'], [b'+', b'@',
b'\x00', b'\x00', b'\x00', b'\x00'], [b'\x00', b'\x00', b'\x18', b'@', b'3',
b'3'], [b'3', b'3', b'3', b'3', b'\xf3', b'\xbf']]
>>> m12.ndim
2
>>> m12.shape
(4, 6)
>>> m12.strides
(6, 1)
>>>
```

在该例子中内存视图m11和m12都被绑定到同一缓冲区，但m11将该缓冲区视为一个一维数组，每个数组元素都是一个双精度浮点数；而m12将该缓冲区视为一个二维数组，每个数组元素都是一个字符。

memoryview.toreadonly()的语法为：

```
memoryview.toreadonly()
```

它返回绑定到同一缓存区的一个只读内存视图。请继续上面的例子：

```
>>> m13 = m2.toreadonly()
>>> m2.readonly
False
>>> m13.readonly
True
>>>
```

最后，内存视图将缓冲区抽象成数组的意义就在于使我们可以按照数组的方式读写该缓冲区（写入要求内存视图的readonly属性引用False）。但要注意，该技巧只能用于将缓冲区抽象成一维数组的内存视图。而memoryview.release()则解除一个内存视图到缓冲区的绑定，使得这种读写不再能进行，其语法为：

```
memoryview.release()
```

请继续上面的例子：

```
>>> m1.tolist()
[97, 98, 99]
>>> m1[0]
97
```

```
>>> m1[1]
98
>>> m1[2]
99
>>> m1.release()
>>> m1[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
>>> m2.tolist()
[0, 2, 3, 4]
>>> m2[0] = 5
>>> m2.tolist()
[5, 2, 3, 4]
>>> m2.release()
>>> m2[0] = 10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
>>>
```

至此我们讨论完了内存视图的用法。这里值得一提的是，在第7章讨论Python的I/O模型时，提到了BytesIO.getbuffer()将返回一个绑定到该缓冲区的可读可写内存视图。下面给出使用该技巧的例子：

```
$ python3

>>> import io
>>> bytes_stream = io.BytesIO(b'veni,vidi,vici')
>>> m = bytes_stream.getbuffer()
>>> m.obj
<_io._BytesIOBuffer object at 0x102bf73a0>
>>> m.nbytes
14
>>> m.readonly
False
>>> m.format
'B'
>>> m.itemsize
1
>>> m.ndim
1
>>> m.shape
(14,)
>>> m.strides
(1,)
>>> m.contiguous
True
>>> m.tobytes()
b'veni,vidi,vici'
>>> m.hex()
'76656e692c766964692c76696369'
>>> m.tolist()
[118, 101, 110, 105, 44, 118, 105, 100, 105, 44, 118, 105, 99, 105]
>>> m[0] = 117
>>> bytes_stream.getvalue()
b'ueni,vidi,vici'
>>>
```