内置类型

以下部分描述了解释器中内置的标准类型。

主要内置类型有数字、序列、映射、类、实例和异常。

有些多项集类是可变的。 它们用于添加、移除或重排其成员的方法将原地执行，并不返回特定的项，绝对不会返回多项集实例自身而是返回 None。

有些操作受多种对象类型的支持；特别地，实际上所有对象都可以比较是否相等、检测逻辑值，以及转换为字符串（使用 repr() 函数或略有差异的 str() 函数）。 后一个函数是在对象由 print() 函数输出时被隐式地调用的。

逻辑值检测

任何对象都可以进行逻辑值的检测，以便在 if 或 while 作为条件或是作为下文所述布尔运算的操作数来使用。

一个对象在默认情况下均被视为真值，除非当该对象被调用时其所属类定义了 __bool__() 方法且返回 False 或是定义了 __len__() 方法且返回零。 [1] 下面基本完整地列出了会被视为假值的内置对象:

- 被定义为假值的常量: None 和 False。
- 任何数值类型的零: 0, 0.0, 0j, Decimal(0), Fraction(0, 1)
- 空的序列和多项集: '', (), [], {}, set(), range(0)

产生布尔值结果的运算和内置函数总是返回 0 或 False 作为假值，1 或 True 作为真值，除非另行说明。 （重要例外：布尔运算 or 和 and 总是返回其中一个操作数。）

布尔运算 --- and, or, not

这些属于布尔运算，按优先级升序排列:

运算	结果:	注释
x or y	if *x* is false, then *y*, else *x*	(1)
x and y	if *x* is false, then *x*, else *y*	(2)
not x	if *x* is false, then True, else False	(3)

注释:

1. 这是个短路运算符，因此只有在第一个参数为假值时才会对第二个参数求值。
2. 这是个短路运算符，因此只有在第一个参数为真值时才会对第二个参数求值。
3. not 的优先级比非布尔运算符低，因此 not a == b 会被解读为 not (a == b) 而 a == not b 会引发语法错误。

比较运算

在 Python 中有八种比较运算符。 它们的优先级相同（比布尔运算的优先级高）。 比较运算可以任意串连；例如，`x < y <= z` 等价于 `x < y and y <= z`，前者的不同之处在于 *y* 只被求值一次（但在两种情况下当 `x < y` 结果为假值时 *z* 都不会被求值）。

此表格汇总了比较运算:

运算	含义
<	严格小于
<=	小于或等于
>	严格大于
>=	大于或等于
==	等于
!=	不等于
is	对象标识
is not	否定的对象标识

除不同的数字类型外，不同类型的对象不能进行相等比较。== 运算符总有定义，但对于某些对象类型（例如，类对象），它等于 `is` 。其他 <、<=、> 和 >= 运算符仅在有意义的地方定义。例如，当参与比较的参数之一为复数时，它们会抛出 `TypeError` 异常。

具有不同标识的类的实例比较结果通常为不相等，除非类定义了 `__eq__()` 方法。

一个类实例不能与相同类或的其他实例或其他类型的对象进行排序，除非该类定义了足够多的方法，包括 `__lt__()`，`__le__()`，`__gt__()` 以及 `__ge__()`（而如果你想实现常规意义上的比较操作，通常只要有 `__lt__()` 和 `__eq__()` 就可以了）。

`is` 和 `is not` 运算符无法自定义；并且它们可以被应用于任意两个对象而不会引发异常。

还有两种具有相同语法优先级的运算 `in` 和 `not in`，它们被 iterable 或实现了 `__contains__()` 方法的类型所支持。

数字类型 --- `int`, `float`, `complex`

存在三种不同的数字类型: *整数*, *浮点数* 和 *复数*。 此外，布尔值属于整数的子类型。 整数具有无限的精度。 浮点数通常使用 C 中的 `double` 来实现；有关你的程序运行所在机器上浮点数的精度和内部表示法可在 `sys.float_info` 中查看。 复数包含实部和虚部，分别以一个浮点数表示。 要从一个复数 *z* 中提取这两个部分，可使用 `z.real` 和 `z.imag`。 （标准库包含附加的数字类型，如表示有理数的 `fractions.Fraction` 以及以用户定制精度表示浮点数的 `decimal.Decimal`。）

数字是由数字字面值或内置函数与运算符的结果来创建的。 不带修饰的整数字面值（包括十六进制、八进制和二进制数）会生成整数。 包含小数点或幂运算符的数字字面值会生成浮点数。 在数字字面值末尾加上 `'j'` 或 `'J'` 会生成虚数（实部为零的复数），你可以将其与整数或浮点数相加来得到具有实部和虚部的复数。

Python 完全支持混合运算：当一个二元算术运算符的操作数有不同数值类型时，"较窄"类型的操作数会拓宽到另一个操作数的类型，其中整数比浮点数窄，浮点数比复数窄。不同类型的数字之间的比较，同比较这些数字的精确值一样。[2]

构造函数 `int()`、 `float()` 和 `complex()` 可以用来构造特定类型的数字。

所有数字类型（复数除外）都支持下列运算（有关运算优先级，请参阅：运算符优先级）：

运算	结果：	注释	完整文档
`x + y`	*x* 和 *y* 的和		
`x - y`	*x* 和 *y* 的差		
`x * y`	*x* 和 *y* 的乘积		
`x / y`	*x* 和 *y* 的商		
`x // y`	*x* 和 *y* 的商数	(1)	
`x % y`	remainder of `x / y`	(2)	
`-x`	*x* 取反		
`+x`	*x* 不变		
`abs(x)`	*x* 的绝对值或大小		`abs()`
`int(x)`	将 *x* 转换为整数	(3)(6)	`int()`
`float(x)`	将 *x* 转换为浮点数	(4)(6)	`float()`
`complex(re, im)`	一个带有实部 *re* 和虚部 *im* 的复数。*im* 默认为0。	(6)	`complex()`
`c.conjugate()`	复数 *c* 的共轭		
`divmod(x, y)`	`(x // y, x % y)`	(2)	`divmod()`
`pow(x, y)`	*x* 的 *y* 次幂	(5)	`pow()`
`x ** y`	*x* 的 *y* 次幂	(5)	

注释:

1. 也称为整数除法。 结果值是一个整数，但结果的类型不一定是 int。 运算结果总是向负无穷的方向舍入: `1//2` 为 `0`，`(-1)//2` 为 `-1`，`1//(-2)` 为 `-1` 而 `(-1)//(-2)` 为 `0`。
2. 不可用于复数。 而应在适当条件下使用 `abs()` 转换为浮点数。
3. 从浮点数转换为整数会被舍入或是像在 C 语言中一样被截断；请参阅 `math.floor()` 和 `math.ceil()` 函数查看转换的完整定义。
4. float 也接受字符串 "nan" 和附带可选前缀 "+" 或 "-" 的 "inf" 分别表示非数字 (NaN) 以及正或负无穷。

5. Python 将 `pow(0, 0)` 和 `0 ** 0` 定义为 `1`，这是编程语言的普遍做法。
6. 接受的数字字面值包括数码 `0` 到 `9` 或任何等效的 Unicode 字符（具有 `Nd` 特征属性的代码点）。

   请参阅 http://www.unicode.org/Public/12.1.0/ucd/extracted/DerivedNumericType.txt 查看具有 `Nd` 特征属性的代码点的完整列表。

所有 `numbers.Real` 类型 (`int` 和 `float`) 还包括下列运算:

运算	结果:
`math.trunc(x)`	*x* 截断为 `Integral`
`round(x[, n])`	*x* 舍入到 *n* 位小数，半数值会舍入到偶数。 如果省略 *n*，则默认为 0。
`math.floor(x)`	<= *x* 的最大 `Integral`
`math.ceil(x)`	>= *x* 的最小 `Integral`

有关更多的数字运算请参阅 `math` 和 `cmath` 模块。

整数类型的按位运算

按位运算只对整数有意义。 计算按位运算的结果，就相当于使用无穷多个二进制符号位对二的补码执行操作。

二进制按位运算的优先级全都低于数字运算，但又高于比较运算；一元运算 `~` 具有与其他一元算术运算 (`+` and `-`) 相同的优先级。

此表格是以优先级升序排序的按位运算列表:

运算	结果:	注释
`x | y`	*x* 和 *y* 按位 *或*	(4)
`x ^ y`	*x* 和 *y* 按位 *异或*	(4)
`x & y`	*x* 和 *y* 按位 *与*	(4)
`x << n`	*x* 左移 *n* 位	(1)(2)
`x >> n`	*x* 右移 *n* 位	(1)(3)
`~x`	*x* 逐位取反	

注释:

1. 负的移位数是非法的，会导致引发 `ValueError`。
2. 左移 *n* 位等价于不带溢出检测地乘以 `pow(2, n)` 。
3. 右移 *n* 位等价于除以 `pow(2, n)` ，作向下取整除法。

4. 使用带有至少一个额外符号扩展位的有限个二进制补码表示（有效位宽度为 `1 + max(x.bit_length(), y.bit_length())` 或以上）执行这些计算就足以获得相当于有无数个符号位时的同样结果。

整数类型的附加方法

int 类型实现了 `numbers.Integral` abstract base class。 此外，它还提供了其他几个方法:

`int.`**`bit_length`**`()`

返回以二进制表示一个整数所需要的位数，不包括符号位和前面的零:

```
>>> n = -37
>>> bin(n)
'-0b100101'
>>> n.bit_length()
6
```

更准确地说，如果 `x` 非零，则 `x.bit_length()` 是使得 `2**(k-1) <= abs(x) < 2**k` 的唯一正整数 `k`。 同样地，当 `abs(x)` 小到足以具有正确的舍入对数时，则 `k = 1 + int(log(abs(x), 2))`。 如果 `x` 为零，则 `x.bit_length()` 返回 `0`。

等价于:

```
def bit_length(self):
    s = bin(self)       # binary representation:  bin(-37) --> '-0b1
    s = s.lstrip('-0b') # remove leading zeros and minus sign
    return len(s)       # len('100101') --> 6
```

3.1 新版功能.

`int.`**`to_bytes`**`(`*`length`*`, `*`byteorder`*`, *, `*`signed=False`*`)`

返回表示一个整数的字节数组。

```
>>> (1024).to_bytes(2, byteorder='big')
b'\x04\x00'
>>> (1024).to_bytes(10, byteorder='big')
b'\x00\x00\x00\x00\x00\x00\x00\x00\x04\x00'
>>> (-1024).to_bytes(10, byteorder='big', signed=True)
b'\xff\xff\xff\xff\xff\xff\xff\xff\xfc\x00'
>>> x = 1000
>>> x.to_bytes((x.bit_length() + 7) // 8, byteorder='little')
b'\xe8\x03'
```

整数会使用 *length* 个字节来表示。 如果整数不能用给定的字节数来表示则会引发 `OverflowError`。

byteorder 参数确定用于表示整数的字节顺序。 如果 *byteorder* 为 `"big"`，则最高位字节放在字节数组的开头。 如果 *byteorder* 为 `"little"`，则最高位字节放在字节数组的末尾。 要请求主机系统上的原生字节顺序，请使用 `sys.byteorder` 作为字节顺序值。

signed 参数确定是否使用二的补码来表示整数。 如果 *signed* 为 `False` 并且给出的是负整数，则会引发 `OverflowError`。 *signed* 的默认值为 `False`。

3.2 新版功能.

classmethod `int.`**`from_bytes`**(*bytes*, *byteorder*, ***, *signed=False*)

返回由给定字节数组所表示的整数。

```
>>> int.from_bytes(b'\x00\x10', byteorder='big')
16
>>> int.from_bytes(b'\x00\x10', byteorder='little')
4096
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=True)
-1024
>>> int.from_bytes(b'\xfc\x00', byteorder='big', signed=False)
64512
>>> int.from_bytes([255, 0, 0], byteorder='big')
16711680
```

bytes 参数必须为一个 bytes-like object 或是生成字节的可迭代对象。

byteorder 参数确定用于表示整数的字节顺序。 如果 *byteorder* 为 `"big"`，则最高位字节放在字节数组的开头。 如果 *byteorder* 为 `"little"`，则最高位字节放在字节数组的末尾。 要请求主机系统上的原生字节顺序，请使用 `sys.byteorder` 作为字节顺序值。

signed 参数指明是否使用二的补码来表示整数。

3.2 新版功能.

`int.`**`as_integer_ratio`**()

返回一对整数，其比率正好等于原整数并且分母为正数。 整数的比率总是用这个整数本身作为分子，`1` 作为分母。

3.8 新版功能.

浮点类型的附加方法

float 类型实现了 `numbers.Real` abstract base class。 float 还具有以下附加方法。

`float.`**`as_integer_ratio`**()

返回一对整数，其比率正好等于原浮点数并且分母为正数。 无穷大会引发 `OverflowError` 而 NaN 则会引发 `ValueError`。

`float.`**`is_integer`**()

如果 float 实例可用有限位整数表示则返回 `True`，否则返回 `False`:

```
>>> (-2.0).is_integer()
True
>>> (3.2).is_integer()
False
```

两个方法均支持与十六进制数字符串之间的转换。 由于 Python 浮点数在内部存储为二进制数，因此浮点数与 *十进制数* 字符串之间的转换往往会导致微小的舍入错误。 而十六进制数字符串却允许精确地表示和描述浮点数。 这在进行调试和数值工作时非常有用。

`float.`**`hex`**`()`

以十六进制字符串的形式返回一个浮点数表示。 对于有限浮点数，这种表示法将总是包含前导的 `0x` 和尾随的 `p` 加指数。

classmethod `float.`**`fromhex`**`(`*`s`*`)`

返回以十六进制字符串 *s* 表示的浮点数的类方法。 字符串 *s* 可以带有前导和尾随的空格。

请注意 `float.hex()` 是实例方法，而 `float.fromhex()` 是类方法。

十六进制字符串采用的形式为:

```
[sign] ['0x'] integer ['.' fraction] ['p' exponent]
```

可选的 `sign` 可以是 `+` 或 `-`，`integer` 和 `fraction` 是十六进制数码组成的字符串，`exponent` 是带有可选前导符的十进制整数。 大小写没有影响，在 integer 或 fraction 中必须至少有一个十六进制数码。 此语法类似于 C99 标准的 6.4.4.2 小节中所描述的语法，也是 Java 1.5 以上所使用的语法。 特别地，`float.hex()` 的输出可以用作 C 或 Java 代码中的十六进制浮点数字面值，而由 C 的 `%a` 格式字符或 Java 的 `Double.toHexString` 所生成的十六进制数字符串由为 `float.fromhex()` 所接受。

请注意 exponent 是十进制数而非十六进制数，它给出要与系数相乘的 2 的幂次。 例如，十六进制数字符串 `0x3.a7p10` 表示浮点数 `(3 + 10./16 + 7./16**2) * 2.0**10` 即 `3740.0`:

```
>>> float.fromhex('0x3.a7p10')
3740.0
```

对 `3740.0` 应用反向转换会得到另一个代表相同数值的十六进制数字符串:

```
>>> float.hex(3740.0)
'0x1.d380000000000p+11'
```

数字类型的哈希运算

对于可能为不同类型的数字 `x` 和 `y`，要求 `x == y` 时必定 `hash(x) == hash(y)` (详情参见 `__hash__()` 方法的文档)。 为了便于在各种数字类型 (包括 `int`, `float`, `decimal.Decimal` 和 `fractions.Fraction`) 上实现并保证效率，Python 对数字类型的哈希运算是基于为任意有理数定义统一的数学函数，因此该运算对 `int` 和 `fractions.Fraction` 的全部实例，以及 `float` 和 `decimal.Decimal` 的全部有限实例均可用。 从本质上说，此函数是通过以一个固定质数 `P` 进行 `P` 降模给出的。 `P` 的值在 Python 中可以 `sys.hash_info` 的 `modulus` 属性的形式被访问。

CPython implementation detail: 目前所用的质数设定，在 C long 为 32 位的机器上 `P = 2**31 - 1` 而在 C long 为 64 位的机器上 `P = 2**61 - 1`。

详细规则如下所述:

- 如果 `x = m / n` 是一个非负的有理数且 `n` 不可被 `P` 整除，则定义 `hash(x)` 为 `m * invmod(n, P) % P`，其中 `invmod(n, P)` 是对 `n` 模 `P` 取反。
- 如果 `x = m / n` 是一个非负的有理数且 `n` 可被 `P` 整除（但 `m` 不能）则 `n` 不能对 `P` 降模，以上规则不适用；在此情况下则定义 `hash(x)` 为常数值 `sys.hash_info.inf`。
- 如果 `x = m / n` 是一个负的有理数则定义 `hash(x)` 为 `-hash(-x)`。 如果结果哈希值为 `-1` 则将其替换为 `-2`。
- 特定值 `sys.hash_info.inf`，`-sys.hash_info.inf` 和 `sys.hash_info.nan` 被用作正无穷、负无穷和空值（所分别对应的）哈希值。 （所有可哈希的空值都具有相同的哈希值。）
- 对于一个 `complex` 值 `z`，会通过计算 `hash(z.real) + sys.hash_info.imag * hash(z.imag)` 将实部和虚部的哈希值结合起来，并进行降模 `2**sys.hash_info.width` 以使其处于 `range(-2**(sys.hash_info.width - 1), 2**(sys.hash_info.width - 1))` 范围之内。 同样地，如果结果为 `-1` 则将其替换为 `-2`。

为了阐明上述规则，这里有一些等价于内置哈希算法的 Python 代码示例，可用于计算有理数、`float` 或 `complex` 的哈希值:

```
import sys, math

def hash_fraction(m, n):
    """Compute the hash of a rational number m / n.

    Assumes m and n are integers, with n positive.
    Equivalent to hash(fractions.Fraction(m, n)).

    """
    P = sys.hash_info.modulus
    # Remove common factors of P.  (Unnecessary if m and n already copri
    while m % P == n % P == 0:
        m, n = m // P, n // P

    if n % P == 0:
        hash_value = sys.hash_info.inf
    else:
        # Fermat's Little Theorem: pow(n, P-1, P) is 1, so
        # pow(n, P-2, P) gives the inverse of n modulo P.
        hash_value = (abs(m) % P) * pow(n, P - 2, P) % P
    if m < 0:
        hash_value = -hash_value
    if hash_value == -1:
        hash_value = -2
    return hash_value

def hash_float(x):
    """Compute the hash of a float x."""

    if math.isnan(x):
        return sys.hash_info.nan
    elif math.isinf(x):
        return sys.hash_info.inf if x > 0 else -sys.hash_info.inf
```

```
    else:
        return hash_fraction(*x.as_integer_ratio())

def hash_complex(z):
    """Compute the hash of a complex number z."""

    hash_value = hash_float(z.real) + sys.hash_info.imag * hash_float(z.
    # do a signed reduction modulo 2**sys.hash_info.width
    M = 2**(sys.hash_info.width - 1)
    hash_value = (hash_value & (M - 1)) - (hash_value & M)
    if hash_value == -1:
        hash_value = -2
    return hash_value
```

迭代器类型

Python 支持在容器中进行迭代的概念。 这是通过使用两个单独方法来实现的；它们被用于允许用户自定义类对迭代的支持。 将在下文中详细描述的序列总是支持迭代方法。

容器对象要提供迭代支持，必须定义一个方法:

`container.__iter__()`
: 返回一个迭代器对象。 该对象需要支持下文所述的迭代器协议。 如果容器支持不同的迭代类型，则可以提供额外的方法来专门地请求不同迭代类型的迭代器。 （支持多种迭代形式的对象的例子有同时支持广度优先和深度优先遍历的树结构。） 此方法对应于 Python/C API 中 Python 对象类型结构体的 `tp_iter` 槽位。

迭代器对象自身需要支持以下两个方法，它们共同组成了 *迭代器协议*:

`iterator.__iter__()`
: 返回迭代器对象本身。 这是同时允许容器和迭代器配合 `for` 和 `in` 语句使用所必须的。 此方法对应于 Python/C API 中 Python 对象类型结构体的 `tp_iter` 槽位。

`iterator.__next__()`
: 从容器中返回下一项。 如果已经没有项可返回，则会引发 `StopIteration` 异常。 此方法对应于 Python/C API 中 Python 对象类型结构体的 `tp_iternext` 槽位。

Python 定义了几种迭代器对象以支持对一般和特定序列类型、字典和其他更特别的形式进行迭代。 除了迭代器协议的实现，特定类型的其他性质对迭代操作来说都不重要。

一旦迭代器的 `__next__()` 方法引发了 `StopIteration`，它必须一直对后续调用引发同样的异常。 不遵循此行为特性的实现将无法正常使用。

生成器类型

Python 的 generator 提供了一种实现迭代器协议的便捷方式。 如果容器对象 `__iter__()` 方法被实现为一个生成器，它将自动返回一个迭代器对象（从技术上说是一个生成器对象），该对象提供 `__iter__()` 和 `__next__()` 方法。 有关生成器的更多信息可以参阅 yield 表达式的文档。

序列类型 --- `list`, `tuple`, `range`

有三种基本序列类型：list, tuple 和 range 对象。 为处理 二进制数据 和 文本字符串 而特别定制的附加序列类型会在专门的小节中描述。

通用序列操作

大多数序列类型，包括可变类型和不可变类型都支持下表中的操作。 `collections.abc.Sequence` ABC 被提供用来更容易地在自定义序列类型上正确地实现这些操作。

此表按优先级升序列出了序列操作。 在表格中，*s* 和 *t* 是具有相同类型的序列，*n*, *i*, *j* 和 *k* 是整数而 *x* 是任何满足 *s* 所规定的类型和值限制的任意对象。

`in` 和 `not in` 操作具有与比较操作相同的优先级。 `+` (拼接) 和 `*` (重复) 操作具有与对应数值运算相同的优先级。 [3]

运算	结果：	注释
`x in s`	如果 *s* 中的某项等于 *x* 则结果为 `True`，否则为 `False`	(1)
`x not in s`	如果 *s* 中的某项等于 *x* 则结果为 `False`，否则为 `True`	(1)
`s + t`	*s* 与 *t* 相拼接	(6)(7)
`s * n` 或 `n * s`	相当于 *s* 与自身进行 *n* 次拼接	(2)(7)
`s[i]`	*s* 的第 *i* 项，起始为 0	(3)
`s[i:j]`	*s* 从 *i* 到 *j* 的切片	(3)(4)
`s[i:j:k]`	*s* 从 *i* 到 *j* 步长为 *k* 的切片	(3)(5)
`len(s)`	*s* 的长度	
`min(s)`	*s* 的最小项	
`max(s)`	*s* 的最大项	
`s.index(x[, i[, j]])`	*x* 在 *s* 中首次出现项的索引号（索引号在 *i* 或其后且在 *j* 之前）	(8)
`s.count(x)`	*x* 在 *s* 中出现的总次数	

相同类型的序列也支持比较。 特别地，tuple 和 list 的比较是通过比较对应元素的字典顺序。 这意味着想要比较结果相等，则每个元素比较结果都必须相等，并且两个序列长度必须相同。 （完整细节请参阅语言参考的 比较运算 部分。）

注释:

1. 虽然 `in` 和 `not in` 操作在通常情况下仅被用于简单的成员检测，某些专门化序列 (例如 `str`, `bytes` 和 `bytearray`) 也使用它们进行子序列检测:

```
>>> "gg" in "eggs"
True
```

2. 小于 `0` 的 *n* 值会被当作 `0` 来处理 (生成一个与 *s* 同类型的空序列)。 请注意序列 *s* 中的项并不会被拷贝；它们会被多次引用。 这一点经常会令 Python 编程新手感到困扰；例如:

```
>>> lists = [[]] * 3
>>> lists
[[], [], []]
>>> lists[0].append(3)
>>> lists
[[3], [3], [3]]
```

   具体的原因在于 `[[]]` 是一个包含了一个空列表的单元素列表，所以 `[[]] * 3` 结果中的三个元素都是对这一个空列表的引用。 修改 `lists` 中的任何一个元素实际上都是对这一个空列表的修改。 你可以用以下方式创建以不同列表为元素的列表:

```
>>> lists = [[] for i in range(3)]
>>> lists[0].append(3)
>>> lists[1].append(5)
>>> lists[2].append(7)
>>> lists
[[3], [5], [7]]
```

   进一步的解释可以在 FAQ 条目 如何创建多维列表? 中查看。
3. 如果 *i* 或 *j* 为负值，则索引顺序是相对于序列 *s* 的末尾: 索引号会被替换为 `len(s) + i` 或 `len(s) + j`。 但要注意 `-0` 仍然为 `0`。
4. *s* 从 *i* 到 *j* 的切片被定义为所有满足 `i <= k < j` 的索引号 *k* 的项组成的序列。 如果 *i* 或 *j* 大于 `len(s)`，则使用 `len(s)`。 如果 *i* 被省略或为 `None`，则使用 `0`。 如果 *j* 被省略或为 `None`，则使用 `len(s)`。 如果 *i* 大于等于 *j*，则切片为空。
5. *s* 从 *i* 到 *j* 步长为 *k* 的切片被定义为所有满足 `0 <= n < (j-i)/k` 的索引号 `x = i + n*k` 的项组成的序列。 换句话说，索引号为 `i`, `i+k`, `i+2*k`, `i+3*k`，以此类推，当达到 *j* 时停止 (但一定不包括 *j*)。 当 *k* 为正值时，*i* 和 *j* 会被减至不大于 `len(s)`。 当 *k* 为负值时，*i* 和 *j* 会被减至不大于 `len(s) - 1`。 如果 *i* 或 *j* 被省略或为 `None`，它们会成为“终止”值 (是哪一端的终止值则取决于 *k* 的符号)。 请注意，*k* 不可为零。 如果 *k* 为 `None`，则当作 `1` 处理。
6. 拼接不可变序列总是会生成新的对象。 这意味着通过重复拼接来构建序列的运行时开销将会基于序列总长度的乘方。 想要获得线性的运行时开销，你必须改用下列替代方案之一:

   - 如果拼接 `str` 对象，你可以构建一个列表并在最后使用 `str.join()` 或是写入一个 `io.StringIO` 实例并在结束时获取它的值
   - 如果拼接 `bytes` 对象，你可以类似地使用 `bytes.join()` 或 `io.BytesIO`，或者你也可以使用 `bytearray` 对象进行原地拼接。 `bytearray` 对象是可变的，并且具有高效的重分配机制

    - 如果拼接 `tuple` 对象，请改为扩展 `list` 类
    - 对于其它类型，请查看相应的文档
7. 某些序列类型 (例如 `range`) 仅支持遵循特定模式的项序列，因此并不支持序列拼接或重复。
8. 当 *x* 在 *s* 中找不到时 `index` 会引发 `ValueError`。 不是所有实现都支持传入额外参数 *i* 和 *j*。 这两个参数允许高效地搜索序列的子序列。 传入这两个额外参数大致相当于使用 `s[i:j].index(x)`，但是不会复制任何数据，并且返回的索引是相对于序列的开头而非切片的开头。

不可变序列类型

不可变序列类型普遍实现而可变序列类型未实现的唯一操作就是对 `hash()` 内置函数的支持。

这种支持允许不可变类型，例如 `tuple` 实例被用作 `dict` 键，以及存储在 `set` 和 `frozenset` 实例中。

尝试对包含有不可哈希值的不可变序列进行哈希运算将会导致 `TypeError`。

可变序列类型

以下表格中的操作是在可变序列类型上定义的。 `collections.abc.MutableSequence` ABC 被提供用来更容易地在自定义序列类型上正确实现这些操作。

表格中的 *s* 是可变序列类型的实例，*t* 是任意可迭代对象，而 *x* 是符合对 *s* 所规定类型与值限制的任何对象 (例如，`bytearray` 仅接受满足 `0 <= x <= 255` 值限制的整数)。

运算	结果:	注释
`s[i] = x`	将 *s* 的第 *i* 项替换为 *x*	
`s[i:j] = t`	将 *s* 从 *i* 到 *j* 的切片替换为可迭代对象 *t* 的内容	
`del s[i:j]`	等同于 `s[i:j] = []`	
`s[i:j:k] = t`	将 `s[i:j:k]` 的元素替换为 *t* 的元素	(1)
`del s[i:j:k]`	从列表中移除 `s[i:j:k]` 的元素	
`s.append(x)`	将 *x* 添加到序列的末尾 (等同于 `s[len(s):len(s)] = [x]`)	
`s.clear()`	从 *s* 中移除所有项 (等同于 `del s[:]`)	(5)
`s.copy()`	创建 *s* 的浅拷贝 (等同于 `s[:]`)	(5)
`s.extend(t)` 或 `s += t`	用 *t* 的内容扩展 *s* (基本上等同于 `s[len(s):len(s)] = t`)	

运算	结果：	注释
`s *= n`	使用 *s* 的内容重复 *n* 次来对其进行更新	(6)
`s.insert(i, x)`	在由 *i* 给出的索引位置将 *x* 插入 *s* (等同于 `s[i:i] = [x]`)	
`s.pop([i])`	提取在 *i* 位置上的项，并将其从 *s* 中移除	(2)
`s.remove(x)`	删除 *s* 中第一个 `s[i]` 等于 *x* 的项目。	(3)
`s.reverse()`	就地将列表中的元素逆序。	(4)

注释:

1. *t* 必须与它所替换的切片具有相同的长度。
2. 可选参数 *i* 默认为 `-1`，因此在默认情况下会移除并返回最后一项。
3. 当在 *s* 中找不到 *x* 时 `remove()` 操作会引发 `ValueError`。
4. 当反转大尺寸序列时 `reverse()` 方法会原地修改该序列以保证空间经济性。 为提醒用户此操作是通过间接影响进行的，它并不会返回反转后的序列。
5. 包括 `clear()` 和 `copy()` 是为了与不支持切片操作的可变容器 (例如 `dict` 和 `set`) 的接口保持一致。 `copy()` 不是 `collections.abc.MutableSequence` ABC 的一部分，但大多数具体的可变序列类都提供了它。

   3.3 新版功能: `clear()` 和 `copy()` 方法。

6. *n* 值为一个整数，或是一个实现了 `__index__()` 的对象。 *n* 值为零或负数将清空序列。 序列中的项不会被拷贝；它们会被多次引用，正如 通用序列操作 中有关 `s * n` 的说明。

列表

列表是可变序列，通常用于存放同类项目的集合（其中精确的相似程度将根据应用而变化）。

class **list**([*iterable*])

可以用多种方式构建列表：

- 使用一对方括号来表示空列表: `[]`
- 使用方括号，其中的项以逗号分隔: `[a]`, `[a, b, c]`
- 使用列表推导式: `[x for x in iterable]`
- 使用类型的构造器: `list()` 或 `list(iterable)`

构造器将构造一个列表，其中的项与 *iterable* 中的项具有相同的的值与顺序。 *iterable* 可以是序列、支持迭代的容器或其它可迭代对象。 如果 *iterable* 已经是一个列表，将创建并返回其副本，类似于 `iterable[:]`。 例如，`list('abc')` 返回 `['a', 'b', 'c']` 而 `list( (1, 2, 3) )` 返回 `[1, 2, 3]`。 如果没有给出参数，构造器将创建一个空列表 `[]`。

其它许多操作也会产生列表，包括 `sorted()` 内置函数。

列表实现了所有 一般 和 可变 序列的操作。 列表还额外提供了以下方法：

sort(*, *key=None*, *reverse=False*)

此方法会对列表进行原地排序，只使用 < 来进行各项间比较。 异常不会被屏蔽 —— 如果有任何比较操作失败，整个排序操作将失败（而列表可能会处于被部分修改的状态）。

`sort()` 接受两个仅限以关键字形式传入的参数 (仅限关键字参数):

key 指定带有一个参数的函数，用于从每个列表元素中提取比较键（例如 `key=str.lower`）。 对应于列表中每一项的键会被计算一次，然后在整个排序过程中使用。 默认值 `None` 表示直接对列表项排序而不计算一个单独的键值。

可以使用 `functools.cmp_to_key()` 将 2.x 风格的 *cmp* 函数转换为 *key* 函数。

reverse 为一个布尔值。 如果设为 `True`，则每个列表元素将按反向顺序比较进行排序。

当顺序大尺寸序列时此方法会原地修改该序列以保证空间经济性。 为提醒用户此操作是通过间接影响进行的，它并不会返回排序后的序列（请使用 `sorted()` 显示地请求一个新的已排序列表实例）。

`sort()` 方法确保是稳定的。 如果一个排序确保不会改变比较结果相等的元素的相对顺序就称其为稳定的 --- 这有利于进行多重排序（例如先按部门、再接薪级排序）。

有关排序示例和简要排序教程，请参阅 排序指南 。

CPython implementation detail: 在一个列表被排序期间，尝试改变甚至进行检测也会造成未定义的影响。 Python 的 C 实现会在排序期间将列表显示为空，如果发现列表在排序期间被改变将会引发 `ValueError`。

元组

元组是不可变序列，通常用于储存异构数据的多项集（例如由 `enumerate()` 内置函数所产生的二元组）。 元组也被用于需要同构数据的不可变序列的情况（例如允许存储到 `set` 或 `dict` 的实例）。

class **tuple**([*iterable*])

可以用多种方式构建元组：

- 使用一对圆括号来表示空元组: `()`
- 使用一个后缀的逗号来表示单元组: `a,` 或 `(a,)`
- 使用以逗号分隔的多个项: `a, b, c` or `(a, b, c)`
- 使用内置的 `tuple()`: `tuple()` 或 `tuple(iterable)`

构造器将构造一个元组，其中的项与 *iterable* 中的项具有相同的值与顺序。 *iterable* 可以是序列、支持迭代的容器或其他可迭代对象。 如果 *iterable* 已经是一个元组，会不加改变

地将其返回。 例如，`tuple('abc')` 返回 `('a', 'b', 'c')` 而 `tuple( [1, 2, 3] )` 返回 `(1, 2, 3)`。 如果没有给出参数，构造器将创建一个空元组 `()`。

请注意决定生成元组的其实是逗号而不是圆括号。 圆括号只是可选的，生成空元组或需要避免语法歧义的情况除外。 例如，`f(a, b, c)` 是在调用函数时附带三个参数，而 `f((a, b, c))` 则是在调用函数时附带一个三元组。

元组实现了所有 一般 序列的操作。

对于通过名称访问相比通过索引访问更清晰的异构数据多项集，`collections.namedtuple()` 可能是比简单元组对象更为合适的选择。

range 对象

`range` 类型表示不可变的数字序列，通常用于在 `for` 循环中循环指定的次数。

class **range**(*stop*)
class **range**(*start*, *stop*[, *step*])

range 构造器的参数必须为整数（可以是内置的 `int` 或任何实现了 `__index__` 特殊方法的对象）。 如果省略 *step* 参数，其默认值为 `1`。 如果省略 *start* 参数，其默认值为 `0`，如果 *step* 为零则会引发 `ValueError`。

如果 *step* 为正值，确定 range `r` 内容的公式为 `r[i] = start + step*i` 其中 `i >= 0` 且 `r[i] < stop`。

如果 *step* 为负值，确定 range 内容的公式仍然为 `r[i] = start + step*i`，但限制条件改为 `i >= 0` 且 `r[i] > stop`.

如果 `r[0]` 不符合值的限制条件，则该 range 对象为空。 range 对象确实支持负索引，但是会将其解读为从正索引所确定的序列的末尾开始索引。

元素绝对值大于 `sys.maxsize` 的 range 对象是被允许的，但某些特性 (例如 `len()`) 可能引发 `OverflowError`。

一些 range 对象的例子:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1, 11))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
[]
>>> list(range(1, 0))
[]
```

range 对象实现了 一般 序列的所有操作，但拼接和重复除外（这是由于 range 对象只能表示符合严格模式的序列，而重复和拼接通常都会违反这样的模式）。

start

start 形参的值 (如果该形参未提供则为 `0`)

stop

stop 形参的值

step

step 形参的值 (如果该形参未提供则为 `1`)

`range` 类型相比常规 `list` 或 `tuple` 的优势在于一个 `range` 对象总是占用固定数量的（较小）内存，不论其所表示的范围有多大（因为它只保存了 `start`, `stop` 和 `step` 值，并会根据需要计算具体单项或子范围的值）。

range 对象实现了 `collections.abc.Sequence` ABC，提供如包含检测、元素索引查找、切片等特性，并支持负索引 (参见 序列类型 --- list, tuple, range):

```
>>> r = range(0, 20, 2)
>>> r
range(0, 20, 2)
>>> 11 in r
False
>>> 10 in r
True
>>> r.index(10)
5
>>> r[5]
10
>>> r[:5]
range(0, 10, 2)
>>> r[-1]
18
```

使用 `==` 和 `!=` 检测 range 对象是否相等是将其作为序列来比较。 也就是说，如果两个 range 对象表示相同的值序列就认为它们是相等的。 （请注意比较结果相等的两个 range 对象可能会具有不同的 `start`, `stop` 和 `step` 属性，例如 `range(0) == range(2, 1, 3)` 而 `range(0, 3, 2) == range(0, 4, 2)`。）

在 3.2 版更改: 实现 Sequence ABC。 支持切片和负数索引。 使用 `int` 对象在固定时间内进行成员检测，而不是逐一迭代所有项。

在 3.3 版更改: 定义 '==' 和 '!=' 以根据 range 对象所定义的值序列来进行比较（而不是根据对象的标识）。

3.3 新版功能: `start`, `stop` 和 `step` 属性。

参见:

- linspace recipe 演示了如何实现一个延迟求值版本的适合浮点数应用的 range 对象。

文本序列类型 --- str

在 Python 中处理文本数据是使用 str 对象，也称为 *字符串*。 字符串是由 Unicode 码位构成的不可变 序列。 字符串字面值有多种不同的写法：

- 单引号: '允许包含有 "双" 引号'
- 双引号: "允许包含有 '单' 引号"。
- 三重引号: '''三重单引号''', """三重双引号"""

使用三重引号的字符串可以跨越多行 —— 其中所有的空白字符都将包含在该字符串字面值中。

作为单一表达式组成部分，之间只由空格分隔的多个字符串字面值会被隐式地转换为单个字符串字面值。 也就是说，("spam " "eggs") == "spam eggs"。

请参阅 字符串和字节串字面值 有解有关不同字符串字面值的更多信息，包括所支持的转义序列，以及使用 r ("raw") 前缀来禁用大多数转义序列的处理。

字符串也可以通过使用 str 构造器从其他对象创建。

由于不存在单独的“字符”类型，对字符串做索引操作将产生一个长度为 1 的字符串。 也就是说，对于一个非空字符串 *s*, s[0] == s[0:1]。

不存在可变的字符串类型，但是 str.join() 或 io.StringIO 可以被被用来根据多个片段高效率地构建字符串。

在 3.3 版更改: 为了与 Python 2 系列的向下兼容，再次允许字符串字面值使用 u 前缀。 它对字符串字面值的含义没有影响，并且不能与 r 前缀同时出现。

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

返回 *object* 的 字符串 版本。 如果未提供 *object* 则返回空字符串。 在其他情况下 str() 的行为取决于 *encoding* 或 *errors* 是否有给出，具体见下。

如果 *encoding* 或 *errors* 均未给出，str(object) 返回 object.__str__()，这是 *object* 的“非正式”或格式良好的字符串表示。 对于字符串对象，这是该字符串本身。 如果 *object* 没有 __str__() 方法，则 str() 将回退为返回 repr(object)。

如果 *encoding* 或 *errors* 至少给出其中之一，则 *object* 应该是一个 bytes-like object (例如 bytes 或 bytearray)。 在此情况下，如果 *object* 是一个 bytes (或 bytearray) 对象，则 str(bytes, encoding, errors) 等价于 bytes.decode(encoding, errors)。 否则的话，会在调用 bytes.decode() 之前获取缓冲区对象下层的 bytes 对象。 请参阅 二进制序列类型 --- bytes, bytearray, memoryview 与 缓冲协议 了解有关缓冲区对象的信息。

将一个 bytes 对象传入 str() 而不给出 *encoding* 或 *errors* 参数的操作属于第一种情况，将返回非正式的字符串表示（另请参阅 Python 的 -b 命令行选项）。 例如:

```
>>> str(b'Zoot!')
"b'Zoot!'"
```

有关 str 类及其方法的更多信息，请参阅下面的 文本序列类型 --- str 和 字符串的方法 小节。 要输出格式化字符串，请参阅 格式化字符串字面值 和 格式字符串语法 小节。 此外还可以参阅 文本处理服务 小节。

字符串的方法

字符串实现了所有 一般 序列的操作，还额外提供了以下列出的一些附加方法。

字符串还支持两种字符串格式化样式，一种提供了很大程度的灵活性和可定制性（参阅 str.format()，格式字符串语法 和 自定义字符串格式化）而另一种是基于 C printf 样式的格式化，它可处理的类型范围较窄，并且更难以正确使用，但对于它可处理的情况往往会更为快速 (printf 风格的字符串格式化)。

标准库的 文本处理服务 部分涵盖了许多其他模块，提供各种文本相关工具（例如包含于 re 模块中的正则表达式支持）。

str.**capitalize**()

返回原字符串的副本，其首个字符大写，其余为小写。

在 3.8 版更改: 第一个字符现在被放入了 titlecase 而不是 uppercase。 这意味着复合字母类字符将只有首个字母改为大写，而再不是全部字符大写。

str.**casefold**()

返回原字符串消除大小写的副本。 消除大小写的字符串可用于忽略大小写的匹配。

消除大小写类似于转为小写，但是更加彻底一些，因为它会移除字符串中的所有大小写变化形式。 例如，德语小写字母 'ß' 相当于 "ss"。 由于它已经是小写了，lower() 不会对 'ß' 做任何改变；而 casefold() 则会将其转换为 "ss"。

消除大小写算法的描述请参见 Unicode 标准的 3.13 节。

3.3 新版功能.

str.**center**(*width*[, *fillchar*])

返回长度为 *width* 的字符串，原字符串在其正中。 使用指定的 *fillchar* 填充两边的空位（默认使用 ASCII 空格符）。 如果 *width* 小于等于 len(s) 则返回原字符串的副本。

str.**count**(*sub*[, *start*[, *end*]])

反回子字符串 *sub* 在 [*start*, *end*] 范围内非重叠出现的次数。 可选参数 *start* 与 *end* 会被解读为切片表示法。

str.**encode**(*encoding="utf-8"*, *errors="strict"*)

返回原字符串编码为字节串对象的版本。 默认编码为 'utf-8'。 可以给出 *errors* 来设置不同的错误处理方案。 *errors* 的默认值为 'strict'，表示编码错误会引发 UnicodeError。 其他可用的值为 'ignore', 'replace', 'xmlcharrefreplace', 'backslashreplace' 以及任何其他通过 codecs.register_error() 注册的值，请参阅 错误处理方案 小节。 要查看可用的编码列表，请参阅 标准编码 小节。

在 3.1 版更改: 加入了对关键字参数的支持。

`str.`**`endswith`**`(`*`suffix`*`[, `*`start`*`[, `*`end`*`]])`

如果字符串以指定的 *suffix* 结束返回 `True`，否则返回 `False`。 *suffix* 也可以为由多个供查找的后缀构成的元组。 如果有可选项 *start*，将从所指定位置开始检查。 如果有可选项 *end*，将在所指定位置停止比较。

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

返回字符串的副本，其中所有的制表符会由一个或多个空格替换，具体取决于当前列位置和给定的制表符宽度。 每 *tabsize* 个字符设为一个制表位（默认值 8 时设定的制表位在列 0, 8, 16 依次类推）。 要展开字符串，当前列将被设为零并逐一检查字符串中的每个字符。 如果字符为制表符 (`\t`)，则会在结果中插入一个或多个空格符，直到当前列等于下一个制表位。 （制表符本身不会被复制。） 如果字符为换行符 (`\n`) 或回车符 (`\r`)，它会被复制并将当前列重设为零。 任何其他字符会被不加修改地复制并将当前列加一，不论该字符在被打印时会如何显示。

```
>>> '01\t012\t0123\t01234'.expandtabs()
'01      012     0123    01234'
>>> '01\t012\t0123\t01234'.expandtabs(4)
'01  012 0123    01234'
```

`str.`**`find`**`(`*`sub`*`[, `*`start`*`[, `*`end`*`]])`

返回子字符串 *sub* 在 `s[start:end]` 切片内被找到的最小索引。 可选参数 *start* 与 *end* 会被解读为切片表示法。 如果 *sub* 未被找到则返回 `-1`。

> **注解:** `find()` 方法应该只在你需要知道 *sub* 所在位置时使用。 要检查 *sub* 是否为子字符串，请使用 `in` 操作符:
>
> ```
> >>> 'Py' in 'Python'
> True
> ```

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

执行字符串格式化操作。 调用此方法的字符串可以包含字符串字面值或者以花括号 `{}` 括起来的替换域。 每个替换域可以包含一个位置参数的数字索引，或者一个关键字参数的名称。 返回的字符串副本中每个替换域都会被替换为对应参数的字符串值。

```
>>> "The sum of 1 + 2 is {0}".format(1+2)
'The sum of 1 + 2 is 3'
```

请参阅 格式字符串语法 了解有关可以在格式字符串中指定的各种格式选项的说明。

> **注解:** 当使用 `n` 类型 (例如: `'{:n}'.format(1234)`) 来格式化数字 (`int`, `float`, `complex`, `decimal.Decimal` 及其子类) 的时候，该函数会临时性地将 `LC_CTYPE` 区域设置为 `LC_NUMERIC` 区域以解码 `localeconv()` 的 `decimal_point` 和 `thousands_sep` 字段，如果它们是非 ASCII 字符或长度超过 1 字节的话，并且 `LC_NUMERIC` 区域会与 `LC_CTYPE` 区域不一致。 这个临时更改会影响其他线程。

在 3.7 版更改: 当使用 n 类型格式化数字时，该函数在某些情况下会临时性地将 LC_CTYPE 区域设置为 LC_NUMERIC 区域。

str.**format_map**(*mapping*)

类似于 str.format(**mapping)，不同之处在于 mapping 会被直接使用而不是复制到一个 dict。 适宜使用此方法的一个例子是当 mapping 为 dict 的子类的情况:

```
>>> class Default(dict):
...     def __missing__(self, key):
...         return key
...
>>> '{name} was born in {country}'.format_map(Default(name='Guido'))
'Guido was born in country'
```

3.2 新版功能.

str.**index**(*sub*[, *start*[, *end*]])

类似于 find()，但在找不到子类时会引发 ValueError。

str.**isalnum**()

如果字符串中的所有字符都是字母或数字且至少有一个字符，则返回 True ， 否则返回 False 。 如果 c.isalpha() ， c.isdecimal() ， c.isdigit() ， 或 c.isnumeric() 之中有一个返回 True ，则字符``c``是字母或数字。

str.**isalpha**()

如果字符串中的所有字符都是字母，并且至少有一个字符，返回 True ，否则返回 False 。字母字符是指那些在 Unicode 字符数据库中定义为 "Letter" 的字符，即那些具有 "Lm"、"Lt"、"Lu"、"Ll" 或 "Lo" 之一的通用类别属性的字符。 注意，这与 Unicode 标准中定义的"字母"属性不同。

str.**isascii**()

如果字符串为空或字符串中的所有字符都是 ASCII ， 返回 True ， 否则返回 False 。 ASCII 字符的码点范围是 U+0000-U+007F 。

3.7 新版功能.

str.**isdecimal**()

如果字符串中的所有字符都是十进制字符且该字符串至少有一个字符，则返回 True ， 否则返回 False 。十进制字符指那些可以用来组成10进制数字的字符，例如 U+0660 ，即阿拉伯字母数字0 。 严格地讲，十进制字符是 Unicode 通用类别 "Nd" 中的一个字符。

str.**isdigit**()

如果字符串中的所有字符都是数字，并且至少有一个字符，返回 True ，否则返回 False 。 数字包括十进制字符和需要特殊处理的数字，如兼容性上标数字。这包括了不能用来组成十进制数的数字，如 Kharosthi 数。 严格地讲，数字是指属性值为 Numeric_Type=Digit 或 Numeric_Type=Decimal 的字符。

str.**isidentifier**()

如果字符串是有效的标识符，返回 True ，依据语言定义， 标识符和关键字 节。

调用 keyword.iskeyword() 来检测字符串 s 是否为保留标识符，例如 def 和 class。

示例：

```
>>> from keyword import iskeyword

>>> 'hello'.isidentifier(), iskeyword('hello')
True, False
>>> 'def'.isidentifier(), iskeyword('def')
True, True
```

str.**islower**()

如果字符串中至少有一个区分大小写的字符 [4] 且此类字符均为小写则返回 True ，否则返回 False 。

str.**isnumeric**()

如果字符串中至少有一个字符且所有字符均为数值字符则返回 True ，否则返回 False 。数值字符包括数字字符，以及所有在 Unicode 中设置了数值特性属性的字符，例如 U+2155, VULGAR FRACTION ONE FIFTH。 正式的定义为：数值字符就是具有特征属性值 Numeric_Type=Digit, Numeric_Type=Decimal 或 Numeric_Type=Numeric 的字符。

str.**isprintable**()

如果字符串中所有字符均为可打印字符或字符串为空则返回 True ，否则返回 False 。不可打印字符是在 Unicode 字符数据库中被定义为 "Other" 或 "Separator" 的字符，例外情况是 ASCII 空格字符 (0x20) 被视作可打印字符。 （请注意在此语境下可打印字符是指当对一个字符串发起调用 repr() 时不必被转义的字符。 它们与字符串写入 sys.stdout 或 sys.stderr 时所需的处理无关。）

str.**isspace**()

如果字符串中只有空白字符且至少有一个字符则返回 True ，否则返回 False 。

空白 字符是指在 Unicode 字符数据库（参见 unicodedata）中主要类别为 Zs ("Separator, space") 或所属双向类为 WS, B 或 S 的字符。

str.**istitle**()

如果字符串中至少有一个字符且为标题字符串则返回 True ，例如大写字符之后只能带非大写字符而小写字符必须有大写字符打头。 否则返回 False 。

str.**isupper**()

如果字符串中至少有一个区分大小写的字符 [4] 且此类字符均为大写则返回 True ，否则返回 False 。

str.**join**(*iterable*)

返回一个由 *iterable* 中的字符串拼接而成的字符串。 如果 *iterable* 中存在任何非字符串值包括 `bytes` 对象则会引发 `TypeError`。 调用该方法的字符串将作为元素之间的分隔。

`str.`**`ljust`**`(`*`width`*`[, `*`fillchar`*`])`

返回长度为 *width* 的字符串，原字符串在其中靠左对齐。 使用指定的 *fillchar* 填充空位 (默认使用 ASCII 空格符)。 如果 *width* 小于等于 `len(s)` 则返回原字符串的副本。

`str.`**`lower`**`()`

返回原字符串的副本，其所有区分大小写的字符 [4] 均转换为小写。

所用转换小写算法的描述请参见 Unicode 标准的 3.13 节。

`str.`**`lstrip`**`([`*`chars`*`])`

返回原字符串的副本，移除其中的前导字符。 *chars* 参数为指定要移除字符的字符串。 如果省略或为 `None`，则 *chars* 参数默认移除空格符。 实际上 *chars* 参数并非指定单个前缀；而是会移除参数值的所有组合:

```
>>> '   spacious   '.lstrip()
'spacious   '
>>> 'www.example.com'.lstrip('cmowz.')
'example.com'
```

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

此静态方法返回一个可供 `str.translate()` 使用的转换对照表。

如果只有一个参数，则它必须是一个将 Unicode 码位序号（整数）或字符（长度为 1 的字符串）映射到 Unicode 码位序号、（任意长度的）字符串或 `None` 的字典。 字符键将会被转换为码位序号。

如果有两个参数，则它们必须是两个长度相等的字符串，并且在结果字典中，x 中每个字符将被映射到 y 中相同位置的字符。 如果有第三个参数，它必须是一个字符串，其中的字符将在结果中被映射到 `None`。

`str.`**`partition`**`(`*`sep`*`)`

在 *sep* 首次出现的位置拆分字符串，返回一个 3 元组，其中包含分隔符之前的部分、分隔符本身，以及分隔符之后的部分。 如果分隔符未找到，则返回的 3 元组中包含字符本身以及两个空字符串。

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

返回字符串的副本，其中出现的所有子字符串 *old* 都将被替换为 *new*。 如果给出了可选参数 *count*，则只替换前 *count* 次出现。

`str.`**`rfind`**`(`*`sub`*`[, `*`start`*`[, `*`end`*`]])`

返回子字符串 *sub* 在字符串内被找到的最大（最右）索引，这样 *sub* 将包含在 `s[start:end]` 当中。 可选参数 *start* 与 *end* 会被解读为切片表示法。 如果未找到则返回 `-1`。

`str.`**`rindex`**`(`*`sub`*`[, `*`start`*`[, `*`end`*`]])`

类似于 `rfind()`，但在子字符串 *sub* 未找到时会引发 `ValueError`。

`str.`**`rjust`**`(`*`width`*`[, `*`fillchar`*`])`

返回长度为 *width* 的字符串，原字符串在其中靠右对齐。 使用指定的 *fillchar* 填充空位 (默认使用 ASCII 空格符)。 如果 *width* 小于等于 `len(s)` 则返回原字符串的副本。

`str.`**`rpartition`**`(`*`sep`*`)`

在 *sep* 最后一次出现的位置拆分字符串，返回一个 3 元组，其中包含分隔符之前的部分、分隔符本身，以及分隔符之后的部分。 如果分隔符未找到，则返回的 3 元组中包含两个空字符串以及字符串本身。

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

返回一个由字符串内单词组成的列表，使用 *sep* 作为分隔字符串。 如果给出了 *maxsplit*，则最多进行 *maxsplit* 次拆分，从 *最右边* 开始。 如果 *sep* 未指定或为 `None`，任何空白字符串都会被作为分隔符。 除了从右边开始拆分，`rsplit()` 的其他行为都类似于下文所述的 `split()`。

`str.`**`rstrip`**`([`*`chars`*`])`

返回原字符串的副本，移除其中的末尾字符。 *chars* 参数为指定要移除字符的字符串。 如果省略或为 `None`，则 *chars* 参数默认移除空格符。 实际上 *chars* 参数并非指定单个后缀；而是会移除参数值的所有组合:

```
>>> '   spacious   '.rstrip()
'   spacious'
>>> 'mississippi'.rstrip('ipz')
'mississ'
```

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

返回一个由字符串内单词组成的列表，使用 *sep* 作为分隔字符串。 如果给出了 *maxsplit*，则最多进行 *maxsplit* 次拆分（因此，列表最多会有 `maxsplit+1` 个元素）。 如果 *maxsplit* 未指定或为 `-1`，则不限制拆分次数（进行所有可能的拆分）。

如果给出了 *sep*，则连续的分隔符不会被组合在一起而是被视为分隔空字符串 (例如 `'1,,2'.split(',')` 将返回 `['1', '', '2']`)。 *sep* 参数可能由多个字符组成 (例如 `'1<>2<>3'.split('<>')` 将返回 `['1', '2', '3']`)。 使用指定的分隔符拆分空字符串将返回 `['']`。

例如

```
>>> '1,2,3'.split(',')
['1', '2', '3']
>>> '1,2,3'.split(',', maxsplit=1)
['1', '2,3']
>>> '1,2,,3,'.split(',')
['1', '2', '', '3', '']
```

如果 *sep* 未指定或为 `None`，则会应用另一种拆分算法：连续的空格会被视为单个分隔符，其结果将不包含开头或末尾的空字符串，如果字符串包含前缀或后缀空格的话。 因此，使用 `None` 拆分空字符串或仅包含空格的字符串将返回 `[]`。

例如

```
>>> '1 2 3'.split()
['1', '2', '3']
>>> '1 2 3'.split(maxsplit=1)
['1', '2 3']
>>> '   1   2   3   '.split()
['1', '2', '3']
```

`str.`**`splitlines`**`([`*`keepends`*`])`

返回由原字符串中各行组成的列表，在行边界的位置拆分。 结果列表中不包含行边界，除非给出了 *keepends* 且为真值。

此方法会以下列行边界进行拆分。 特别地，行边界是 universal newlines 的一个超集。

表示符	描述
`\n`	换行
`\r`	回车
`\r\n`	回车 + 换行
`\v` 或 `\x0b`	行制表符
`\f` 或 `\x0c`	换表单
`\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

在 3.2 版更改: `\v` 和 `\f` 被添加到行边界列表

例如

```
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines()
['ab c', '', 'de fg', 'kl']
>>> 'ab c\n\nde fg\rkl\r\n'.splitlines(keepends=True)
['ab c\n', '\n', 'de fg\r', 'kl\r\n']
```

不同于 `split()`，当给出了分隔字符串 *sep* 时，对于空字符串此方法将返回一个空列表，而末尾的换行不会令结果中增加额外的行:

```
>>> "".splitlines()
[]
>>> "One line\n".splitlines()
['One line']
```

作为比较，`split('\n')` 的结果为:

```
>>> ''.split('\n')
['']
>>> 'Two lines\n'.split('\n')
['Two lines', '']
```

`str.`**`startswith`**(*prefix*[, *start*[, *end*]])

如果字符串以指定的 *prefix* 开始则返回 `True`，否则返回 `False`。 *prefix* 也可以为由多个供查找的前缀构成的元组。 如果有可选项 *start*，将从所指定位置开始检查。 如果有可选项 *end*，将在所指定位置停止比较。

`str.`**`strip`**([*chars*])

返回原字符串的副本，移除其中的前导和末尾字符。 *chars* 参数为指定要移除字符的字符串。 如果省略或为 `None`，则 *chars* 参数默认移除空格符。 实际上 *chars* 参数并非指定单个前缀或后缀；而是会移除参数值的所有组合:

```
>>> '   spacious   '.strip()
'spacious'
>>> 'www.example.com'.strip('cmowz.')
'example'
```

最外侧的前导和末尾 *chars* 参数值将从字符串中移除。 开头端的字符的移除将在遇到一个未包含于 *chars* 所指定字符集的字符时停止。 类似的操作也将在结尾端发生。 例如:

```
>>> comment_string = '#....... Section 3.2.1 Issue #32 .......'
>>> comment_string.strip('.#! ')
'Section 3.2.1 Issue #32'
```

`str.`**`swapcase`**()

返回原字符串的副本，其中大写字符转换为小写，反之亦然。 请注意 `s.swapcase().swapcase() == s` 并不一定为真值。

`str.`**`title`**()

返回原字符串的标题版本，其中每个单词第一个字母为大写，其余字母为小写。

例如

```
>>> 'Hello world'.title()
'Hello World'
```

该算法使用一种简单的与语言无关的定义，将连续的字母组合视为单词。 该定义在多数情况下都很有效，但它也意味着代表缩写形式与所有格的撇号也会成为单词边界，这可能导致不希望的结果:

```
>>> "they're bill's friends from the UK".title()
"They're Bill'S Friends From The Uk"
```

可以使用正则表达式来构建针对撇号的特别处理:

```
>>> import re
>>> def titlecase(s):
...     return re.sub(r"[A-Za-z]+('[A-Za-z]+)?",
...                   lambda mo: mo.group(0).capitalize(),
...                   s)
...
>>> titlecase("they're bill's friends.")
"They're Bill's Friends."
```

str.**translate**(*table*)

返回原字符串的副本，其中每个字符按给定的转换表进行映射。 转换表必须是一个使用 `__getitem__()` 来实现索引操作的对象，通常为 mapping 或 sequence。 当以 Unicode 码位序号（整数）为索引时，转换表对象可以做以下任何一种操作：返回 Unicode 序号或字符串，将字符映射为一个或多个字符；返回 `None`，将字符从结果字符串中删除；或引发 `LookupError` 异常，将字符映射为其自身。

你可以使用 `str.maketrans()` 基于不同格式的字符到字符映射来创建一个转换映射表。

另请参阅 `codecs` 模块以了解定制字符映射的更灵活方式。

str.**upper**()

返回原字符串的副本，其中所有区分大小写的字符 [4] 均转换为大写。 请注意如果 `s` 包含不区分大小写的字符或者如果结果字符的 Unicode 类别不是 "Lu" (Letter, uppercase) 而是 "Lt" (Letter, titlecase) 则 `s.upper().isupper()` 有可能为 `False`。

所用转换大写算法的描述请参见 Unicode 标准的 3.13 节。

str.**zfill**(*width*)

返回原字符串的副本，在左边填充 ASCII `'0'` 数码使其长度变为 *width*。 正负值前缀 (`'+'`/`'-'`) 的处理方式是在正负符号 *之后* 填充而非在之前。 如果 *width* 小于等于 `len(s)` 则返回原字符串的副本。

例如

```
>>> "42".zfill(5)
'00042'
>>> "-42".zfill(5)
'-0042'
```

printf 风格的字符串格式化

注解: 此处介绍的格式化操作具有多种怪异特性，可能导致许多常见错误（例如无法正确显示元组和字典）。 使用较新的 格式化字符串字面值，`str.format()` 接口或 模板字符串 有助于避免这样的错误。 这些替代方案中的每一种都更好地权衡并提供了简单、灵活以及可扩展性优势。

字符串具有一种特殊的内置操作：使用 `%` (取模) 运算符。 这也被称为字符串的 *格式化* 或 *插值* 运算符。 对于 `format % values` (其中 *format* 为一个字符串)，在 *format* 中的 `%` 转换标记

符将被替换为零个或多个 *values* 条目。 其效果类似于在 C 语言中使用 `sprintf()`。

如果 *format* 要求一个单独参数，则 *values* 可以为一个非元组对象。 [5] 否则的话， *values* 必须或者是一个包含项数与格式字符串中指定的转换符项数相同的元组，或者是一个单独映射对象（例如字典）。

转换标记符包含两个或更多字符并具有以下组成，且必须遵循此处规定的顺序：

1. `'%'` 字符，用于标记转换符的起始。
2. 映射键（可选），由加圆括号的字符序列组成 (例如 `(somename)`)。
3. 转换旗标（可选），用于影响某些转换类型的结果。
4. 最小字段宽度（可选）。 如果指定为 `'*'` (星号)，则实际宽度会从 *values* 元组的下一元素中读取，要转换的对象则为最小字段宽度和可选的精度之后的元素。
5. 精度（可选），以在 `'.'` (点号) 之后加精度值的形式给出。 如果指定为 `'*'` (星号)，则实际精度会从 *values* 元组的下一元素中读取，要转换的对象则为精度之后的元素。
6. 长度修饰符（可选）。
7. 转换类型。

当右边的参数为一个字典（或其他映射类型）时，字符串中的格式 *必须* 包含加圆括号的映射键，对应 `'%'` 字符之后字典中的每一项。 映射键将从映射中选取要格式化的值。 例如：

```
>>> print('%(language)s has %(number)03d quote types.' %
...       {'language': "Python", "number": 2})
Python has 002 quote types.
```

在此情况下格式中不能出现 `*` 标记符（因其需要一个序列类的参数列表）。

转换旗标为：

Flag	含义
`'#'`	值的转换将使用“替代形式”（具体定义见下文）。
`'0'`	转换将为数字值填充零字符。
`'-'`	转换值将靠左对齐（如果同时给出 `'0'` 转换，则会覆盖后者）。
`' '`	(空格) 符号位转换产生的正数（或空字符串）前将留出一个空格。
`'+'`	符号字符 (`'+'` 或 `'-'`) 将显示于转换结果的开头（会覆盖 "空格" 旗标）。

可以给出长度修饰符 (`h`, `l` 或 `L`)，但会被忽略，因为对 Python 来说没有必要 -- 所以 `%ld` 等价于 `%d`。

转换类型为：

转换符	含义	注释
`'d'`	有符号十进制整数。	
`'i'`	有符号十进制整数。	

转换符	含义	注释
'o'	有符号八进制数。	(1)
'u'	过时类型 -- 等价于 'd'。	(6)
'x'	有符号十六进制数（小写）。	(2)
'X'	有符号十六进制数（大写）。	(2)
'e'	浮点指数格式（小写）。	(3)
'E'	浮点指数格式（大写）。	(3)
'f'	浮点十进制格式。	(3)
'F'	浮点十进制格式。	(3)
'g'	浮点格式。 如果指数小于 -4 或不小于精度则使用小写指数格式，否则使用十进制格式。	(4)
'G'	浮点格式。 如果指数小于 -4 或不小于精度则使用大写指数格式，否则使用十进制格式。	(4)
'c'	单个字符（接受整数或单个字符的字符串）。	
'r'	字符串（使用 repr() 转换任何 Python 对象）。	(5)
's'	字符串（使用 str() 转换任何 Python 对象）。	(5)
'a'	字符串（使用 ascii() 转换任何 Python 对象）。	(5)
'%'	不转换参数，在结果中输出一个 '%' 字符。	

注释:

1. 此替代形式会在第一个数码之前插入标示八进制数的前缀 ('0o')。
2. 此替代形式会在第一个数码之前插入 '0x' 或 '0X' 前缀（取决于是使用 'x' 还是 'X' 格式）。
3. 此替代形式总是会在结果中包含一个小数点，即使其后并没有数码。

   小数点后的数码位数由精度决定，默认为 6。
4. 此替代形式总是会在结果中包含一个小数点，末尾各位的零不会如其他情况下那样被移除。

   小数点前后的有效数码位数由精度决定，默认为 6。
5. 如果精度为 N，输出将截短为 N 个字符。
6. 参见 **PEP 237**。

由于 Python 字符串显式指明长度，%s 转换不会将 '\0' 视为字符串的结束。

在 3.1 版更改: 绝对值超过 1e50 的 %f 转换不会再被替换为 %g 转换。

二进制序列类型 --- bytes, bytearray, memoryview

操作二进制数据的核心内置类型是 bytes 和 bytearray。 它们由 memoryview 提供支持，该对象使用 缓冲区协议 来访问其他二进制对象所在内存，不需要创建对象的副本。

array 模块支持高效地存储基本数据类型，例如 32 位整数和 IEEE754 双精度浮点值。

bytes 对象

bytes 对象是由单个字节构成的不可变序列。 由于许多主要二进制协议都基于 ASCII 文本编码，因此 bytes 对象提供了一些仅在处理 ASCII 兼容数据时可用，并且在许多特性上与字符串对象紧密相关的方法。

class **bytes**([*source*[, *encoding*[, *errors*]]])

首先，表示 bytes 字面值的语法与字符串字面值的大致相同，只是添加了一个 b 前缀：

- 单引号: b'同样允许嵌入 "双" 引号'。
- 双引号: b"同样允许嵌入 '单' 引号"。
- 三重引号: b'''三重单引号''', b"""三重双引号"""

bytes 字面值中只允许 ASCII 字符（无论源代码声明的编码为何）。 任何超出 127 的二进制值必须使用相应的转义序列形式加入 bytes 字面值。

像字符串字面值一样，bytes 字面值也可以使用 r 前缀来禁用转义序列处理。 请参阅 字符串和字节串字面值 了解有关各种 bytes 字面值形式的详情，包括所支持的转义序列。

虽然 bytes 字面值和表示法是基于 ASCII 文本的，但 bytes 对象的行为实际上更像是不可变的整数序列，序列中的每个值的大小被限制为 0 <= x < 256 (如果违反此限制将引发 ValueError)。 这种限制是有意设计用以强调以下事实，虽然许多二进制格式都包含基于 ASCII 的元素，可以通过某些面向文本的算法进行有用的操作，但情况对于任意二进制数据来说通常却并非如此（盲目地将文本处理算法应用于不兼容 ASCII 的二进制数据格式往往将导致数据损坏）。

除了字面值形式，bytes 对象还可以通过其他几种方式来创建：

- 指定长度的以零值填充的 bytes 对象: bytes(10)
- 通过由整数组成的可迭代对象: bytes(range(20))
- 通过缓冲区协议复制现有的二进制数据: bytes(obj)

另请参阅 bytes 内置类型。

由于两个十六进制数码精确对应一个字节，因此十六进制数是描述二进制数据的常用格式。 相应地，bytes 类型具有从此种格式读取数据的附加类方法：

classmethod **fromhex**(*string*)

此 bytes 类方法返回一个解码给定字符串的 bytes 对象。 字符串必须由表示每个字节的两个十六进制数码构成，其中的 ASCII 空白符会被忽略。

```
>>> bytes.fromhex('2Ef0 F1f2  ')
b'.\xf0\xf1\xf2'
```

在 3.7 版更改: `bytes.fromhex()` 现在会忽略所有 ASCII 空白符而不只是空格符。

存在一个反向转换函数，可以将 bytes 对象转换为对应的十六进制表示。

hex([*sep*[, *bytes_per_sep*]])

返回一个字符串对象，该对象包含实例中每个字节的两个十六进制数字。

```
>>> b'\xf0\xf1\xf2'.hex()
'f0f1f2'
```

如果你希望令十六进制数字符串更易读，你可以指定单个字符分隔符作为 *sep* 形参包含于输出中。 默认会放在每个字节之间。 第二个可选的 *bytes_per_sep* 形参控制间距。 正值会从右开始计算分隔符的位置，负值则是从左开始。

```
>>> value = b'\xf0\xf1\xf2'
>>> value.hex('-')
'f0-f1-f2'
>>> value.hex('_', 2)
'f0_f1f2'
>>> b'UUDDLRLRAB'.hex(' ', -4)
'55554444 4c524c52 4142'
```

3.5 新版功能.

在 3.8 版更改: `bytes.hex()` 现在支持可选的 *sep* 和 *bytes_per_sep* 形参以在十六进制输出的字节之间插入分隔符。

由于 bytes 对象是由整数构成的序列（类似于元组），因此对于一个 bytes 对象 *b*，`b[0]` 将为一个整数，而 `b[0:1]` 将为一个长度为 1 的 bytes 对象。 （这与文本字符串不同，索引和切片所产生的将都是一个长度为 1 的字符串）。

bytes 对象的表示使用字面值格式 (`b'...'`)，因为它通常都要比像 `bytes([46, 46, 46])` 这样的格式更好用。 你总是可以使用 `list(b)` 将 bytes 对象转换为一个由整数构成的列表。

注解: 针对 Python 2.x 用户的说明：在 Python 2.x 系列中，允许 8 位字符串（ 2.x 所提供的最接近内置二进制数据类型的对象）与 Unicode 字符串进行各种隐式转换。 这是为了实现向下兼容的变通做法，以适应 Python 最初只支持 8 位文本而 Unicode 文本是后来才被加入这一事实。 在 Python 3.x 中，这些隐式转换已被取消 —— 8 位二进制数据与 Unicode 文本间的转换必须显式地进行，bytes 与字符串对象的比较结果将总是不相等。

bytearray 对象

`bytearray` 对象是 `bytes` 对象的可变对应物。

class **bytearray**([*source*[, *encoding*[, *errors*]]])

bytearray 对象没有专属的字面值语法，它们总是通过调用构造器来创建：

- 创建一个空实例: `bytearray()`
- 创建一个指定长度的以零值填充的实例: `bytearray(10)`

- 通过由整数组成的可迭代对象: `bytearray(range(20))`
- 通过缓冲区协议复制现有的二进制数据: `bytearray(b'Hi!')`

由于 bytearray 对象是可变的，该对象除了 bytes 和 bytearray 操作 中所描述的 bytes 和 bytearray 共有操作之外，还支持 可变 序列操作。

另请参见 bytearray 内置类型。

由于两个十六进制数码精确对应一个字节，因此十六进制数是描述二进制数据的常用格式。 相应地，bytearray 类型具有从此种格式读取数据的附加类方法：

classmethod **fromhex**(*string*)

bytearray 类方法返回一个解码给定字符串的 bytearray 对象。 字符串必须由表示每个字节的两个十六进制数码构成，其中的 ASCII 空白符会被忽略。

```
>>> bytearray.fromhex('2Ef0 F1f2  ')
bytearray(b'.\xf0\xf1\xf2')
```

在 3.7 版更改: `bytearray.fromhex()` 现在会忽略所有 ASCII 空白符而不只是空格符。

存在一个反向转换函数，可以将 bytearray 对象转换为对应的十六进制表示。

hex([*sep*[, *bytes_per_sep*]])

返回一个字符串对象，该对象包含实例中每个字节的两个十六进制数字。

```
>>> bytearray(b'\xf0\xf1\xf2').hex()
'f0f1f2'
```

3.5 新版功能.

在 3.8 版更改: 与 `bytes.hex()` 相似， `bytearray.hex()` 现在支持可选的 *sep* 和 *bytes_per_sep* 参数以在十六进制输出的字节之间插入分隔符。

由于 bytearray 对象是由整数构成的序列（类似于列表），因此对于一个 bytearray 对象 *b*，`b[0]` 将为一个整数，而 `b[0:1]` 将为一个长度为 1 的 bytearray 对象。 （这与文本字符串不同，索引和切片所产生的将都是一个长度为 1 的字符串）。

bytearray 对象的表示使用 bytes 对象字面值格式 (`bytearray(b'...')`)，因为它通常都要比 `bytearray([46, 46, 46])` 这样的格式更好用。 你总是可以使用 `list(b)` 将 bytearray 对象转换为一个由整数构成的列表。

bytes 和 bytearray 操作

bytes 和 bytearray 对象都支持 通用 序列操作。 它们不仅能与相同类型的操作数，也能与任何 bytes-like object 进行互操作。 由于这样的灵活性，它们可以在操作中自由地混合而不会导致错误。 但是，操作结果的返回值类型可能取决于操作数的顺序。

注解: bytes 和 bytearray 对象的方法不接受字符串作为其参数，就像字符串的方法不接受 bytes 对象作为其参数一样。 例如，你必须使用以下写法:

```
a = "abc"
b = a.replace("a", "f")
```

和:

```
a = b"abc"
b = a.replace(b"a", b"f")
```

某些 bytes 和 bytearray 操作假定使用兼容 ASCII 的二进制格式，因此在处理任意二进数数据时应当避免使用。 这些限制会在下文中说明。

注解: 使用这些基于 ASCII 的操作来处理未以基于 ASCII 的格式存储的二进制数据可能会导致数据损坏。

bytes 和 bytearray 对象的下列方法可以用于任意二进制数据。

bytes.**count**(*sub*[, *start*[, *end*]])
bytearray.**count**(*sub*[, *start*[, *end*]])

返回子序列 *sub* 在 [*start*, *end*] 范围内非重叠出现的次数。 可选参数 *start* 与 *end* 会被解读为切片表示法。

要搜索的子序列可以是任意 bytes-like object 或是 0 至 255 范围内的整数。

在 3.3 版更改: 也接受 0 至 255 范围内的整数作为子序列。

bytes.**decode**(*encoding="utf-8"*, *errors="strict"*)
bytearray.**decode**(*encoding="utf-8"*, *errors="strict"*)

返回从给定 bytes 解码出来的字符串。 默认编码为 `'utf-8'`。 可以给出 *errors* 来设置不同的错误处理方案。 *errors* 的默认值为 `'strict'`，表示编码错误会引发 `UnicodeError`。 其他可用的值为 `'ignore'`, `'replace'` 以及任何其他通过 `codecs.register_error()` 注册的名称，请参阅 错误处理方案 小节。 要查看可用的编码列表，请参阅 标准编码 小节。

注解: 将 *encoding* 参数传给 `str` 允许直接解码任何 bytes-like object，无须创建临时的 bytes 或 bytearray 对象。

在 3.1 版更改: 加入了对关键字参数的支持。

bytes.**endswith**(*suffix*[, *start*[, *end*]])
bytearray.**endswith**(*suffix*[, *start*[, *end*]])

如果二进制数据以指定的 *suffix* 结束则返回 `True`，否则返回 `False`。 *suffix* 也可以为由多个供查找的后缀构成的元组。 如果有可选项 *start*，将从所指定位置开始检查。 如果有可选项 *end*，将在所指定位置停止比较。

要搜索的后缀可以是任意 bytes-like object。

bytes.**find**(*sub*[, *start*[, *end*]])
bytearray.**find**(*sub*[, *start*[, *end*]])

返回子序列 *sub* 在数据中被找到的最小索引，*sub* 包含于切片 `s[start:end]` 之内。可选参数 *start* 与 *end* 会被解读为切片表示法。如果 *sub* 未被找到则返回 `-1`。

要搜索的子序列可以是任意 bytes-like object 或是 0 至 255 范围内的整数。

> **注解:** `find()` 方法应该只在你需要知道 *sub* 所在位置时使用。要检查 *sub* 是否为子串，请使用 `in` 操作符:
>
> ```
> >>> b'Py' in b'Python'
> True
> ```

在 3.3 版更改: 也接受 0 至 255 范围内的整数作为子序列。

bytes.**index**(*sub*[, *start*[, *end*]])
bytearray.**index**(*sub*[, *start*[, *end*]])

类似于 `find()`，但在找不到子序列时会引发 `ValueError`。

要搜索的子序列可以是任意 bytes-like object 或是 0 至 255 范围内的整数。

在 3.3 版更改: 也接受 0 至 255 范围内的整数作为子序列。

bytes.**join**(*iterable*)
bytearray.**join**(*iterable*)

返回一个由 *iterable* 中的二进制数据序列拼接而成的 bytes 或 bytearray 对象。如果 *iterable* 中存在任何非 字节类对象 包括存在 `str` 对象值则会引发 `TypeError`。提供该方法的 bytes 或 bytearray 对象的内容将作为元素之间的分隔。

static bytes.**maketrans**(*from*, *to*)
static bytearray.**maketrans**(*from*, *to*)

此静态方法返回一个可用于 `bytes.translate()` 的转换对照表，它将把 *from* 中的每个字符映射为 *to* 中相同位置上的字符；*from* 与 *to* 必须都是 字节类对象 并且具有相同的长度。

3.1 新版功能.

bytes.**partition**(*sep*)
bytearray.**partition**(*sep*)

在 *sep* 首次出现的位置拆分序列，返回一个 3 元组，其中包含分隔符之前的部分、分隔符本身或其 bytearray 副本，以及分隔符之后的部分。如果分隔符未找到，则返回的 3 元组中包含原序列以及两个空的 bytes 或 bytearray 对象。

要搜索的分隔符可以是任意 bytes-like object。

bytes.**replace**(*old*, *new*[, *count*])

bytearray.**replace**(*old*, *new*[, *count*])

返回序列的副本，其中出现的所有子序列 *old* 都将被替换为 *new*。 如果给出了可选参数 *count*，则只替换前 *count* 次出现。

要搜索的子序列及其替换序列可以是任意 bytes-like object。

> **注解:** 此方法的 bytearray 版本 *并非* 原地操作 —— 它总是产生一个新对象，即便没有做任何改变。

bytes.**rfind**(*sub*[, *start*[, *end*]])

bytearray.**rfind**(*sub*[, *start*[, *end*]])

返回子序列 *sub* 在序列内被找到的最大（最右）索引，这样 *sub* 将包含在 `s[start:end]` 当中。 可选参数 *start* 与 *end* 会被解读为切片表示法。 如果未找到则返回 `-1`。

要搜索的子序列可以是任意 bytes-like object 或是 0 至 255 范围内的整数。

在 3.3 版更改: 也接受 0 至 255 范围内的整数作为子序列。

bytes.**rindex**(*sub*[, *start*[, *end*]])

bytearray.**rindex**(*sub*[, *start*[, *end*]])

类似于 `rfind()`，但在子序列 *sub* 未找到时会引发 `ValueError`。

要搜索的子序列可以是任意 bytes-like object 或是 0 至 255 范围内的整数。

在 3.3 版更改: 也接受 0 至 255 范围内的整数作为子序列。

bytes.**rpartition**(*sep*)

bytearray.**rpartition**(*sep*)

在 *sep* 最后一次出现的位置拆分序列，返回一个 3 元组，其中包含分隔符之前的部分，分隔符本身或其 bytearray 副本，以及分隔符之后的部分。 如果分隔符未找到，则返回的 3 元组中包含两个空的 bytes 或 bytearray 对象以及原序列的副本。

要搜索的分隔符可以是任意 bytes-like object。

bytes.**startswith**(*prefix*[, *start*[, *end*]])

bytearray.**startswith**(*prefix*[, *start*[, *end*]])

如果二进制数据以指定的 *prefix* 开头则返回 `True`，否则返回 `False`。 *prefix* 也可以为由多个供查找的前缀构成的元组。 如果有可选项 *start*，将从所指定位置开始检查。 如果有可选项 *end*，将在所指定位置停止比较。

要搜索的前缀可以是任意 bytes-like object。

bytes.**translate**(*table*, */*, *delete=b''*)

bytearray.**translate**(*table*, */*, *delete=b''*)

返回原 bytes 或 bytearray 对象的副本，移除其中所有在可选参数 *delete* 中出现的 bytes，其余 bytes 将通过给定的转换表进行映射，该转换表必须是长度为 256 的 bytes 对象。

你可以使用 `bytes.maketrans()` 方法来创建转换表。

对于仅需移除字符的转换，请将 *table* 参数设为 `None`：

```
>>> b'read this short text'.translate(None, b'aeiou')
b'rd ths shrt txt'
```

在 3.6 版更改: 现在支持将 *delete* 作为关键字参数。

以下 bytes 和 bytearray 对象的方法的默认行为会假定使用兼容 ASCII 的二进制格式，但通过传入适当的参数仍然可用于任意二进制数据。 请注意本小节中所有的 bytearray 方法都 *不是* 原地执行操作，而是会产生新的对象。

`bytes.`**`center`**`(`*`width`*`[, `*`fillbyte`*`])`
`bytearray.`**`center`**`(`*`width`*`[, `*`fillbyte`*`])`

返回原对象的副本，在长度为 *width* 的序列内居中，使用指定的 *fillbyte* 填充两边的空位（默认使用 ASCII 空格符）。 对于 `bytes` 对象，如果 *width* 小于等于 `len(s)` 则返回原序列的副本。

> **注解:** 此方法的 bytearray 版本 *并非* 原地操作 —— 它总是产生一个新对象，即便没有做任何改变。

`bytes.`**`ljust`**`(`*`width`*`[, `*`fillbyte`*`])`
`bytearray.`**`ljust`**`(`*`width`*`[, `*`fillbyte`*`])`

反回原对象的副本，在长度为 *width* 的序列中靠左对齐。 使用指定的 *fillbyte* 填充空位（默认使用 ASCII 空格符）。 对于 `bytes` 对象，如果 *width* 小于等于 `len(s)` 则返回原序列的副本。

> **注解:** 此方法的 bytearray 版本 *并非* 原地操作 —— 它总是产生一个新对象，即便没有做任何改变。

`bytes.`**`lstrip`**`([`*`chars`*`])`
`bytearray.`**`lstrip`**`([`*`chars`*`])`

返回原序列的副本，移除指定的前导字节。 *chars* 参数为指定要移除字节值集合的二进制序列 —— 这个名称表明此方法通常是用于 ASCII 字符。 如果省略或为 `None`，则 *chars* 参数默认移除 ASCII 空白符。 *chars* 参数并非指定单个前缀；而是会移除参数值的所有组合：

```
>>> b'   spacious   '.lstrip()
b'spacious   '
>>> b'www.example.com'.lstrip(b'cmowz.')
b'example.com'
```

要移除的字节值二进制序列可以是任意 bytes-like object。

> **注解:** 此方法的 bytearray 版本 *并非* 原地操作 —— 它总是产生一个新对象，即便没有做任何改变。

bytes.**rjust**(*width*[, *fillbyte*])
bytearray.**rjust**(*width*[, *fillbyte*])

返回原对象的副本，在长度为 *width* 的序列中靠右对齐。 使用指定的 *fillbyte* 填充空位（默认使用 ASCII 空格符）。 对于 bytes 对象，如果 *width* 小于等于 len(s) 则返回原序列的副本。

> **注解:** 此方法的 bytearray 版本 *并非* 原地操作 —— 它总是产生一个新对象，即便没有做任何改变。

bytes.**rsplit**(*sep=None*, *maxsplit=-1*)
bytearray.**rsplit**(*sep=None*, *maxsplit=-1*)

将二进制序列拆分为相同类型的子序列，使用 *sep* 作为分隔符。 如果给出了 *maxsplit*，则最多进行 *maxsplit* 次拆分，从 *最右边* 开始。 如果 *sep* 未指定或为 None，任何只包含 ASCII 空白符的子序列都会被作为分隔符。 除了从右边开始拆分，rsplit() 的其他行为都类似于下文所述的 split()。

bytes.**rstrip**([*chars*])
bytearray.**rstrip**([*chars*])

返回原序列的副本，移除指定的末尾字节。 *chars* 参数为指定要移除字节值集合的二进制序列 —— 这个名称表明此方法通常是用于 ASCII 字符。 如果省略或为 None，则 *chars* 参数默认移除 ASCII 空白符。 *chars* 参数并非指定单个后缀；而是会移除参数值的所有组合:

```
>>> b'   spacious   '.rstrip()
b'   spacious'
>>> b'mississippi'.rstrip(b'ipz')
b'mississ'
```

要移除的字节值二进制序列可以是任意 bytes-like object。

> **注解:** 此方法的 bytearray 版本 *并非* 原地操作 —— 它总是产生一个新对象，即便没有做任何改变。

bytes.**split**(*sep=None*, *maxsplit=-1*)
bytearray.**split**(*sep=None*, *maxsplit=-1*)

将二进制序列拆分为相同类型的子序列，使用 *sep* 作为分隔符。 如果给出了 *maxsplit* 且非负值，则最多进行 *maxsplit* 次拆分（因此，列表最多会有 maxsplit+1 个元素）。 如果 *maxsplit* 未指定或为 -1，则不限制拆分次数（进行所有可能的拆分）。

如果给出了 *sep*，则连续的分隔符不会被组合在一起而是被视为分隔空子序列（例如 b'1,,2'.split(b',') 将返回 [b'1', b'', b'2']）。 *sep* 参数可能为一个多字节序列（例如 b'1<>2<>3'.split(b'<>') 将返回 [b'1', b'2', b'3']）。 使用指定的分隔符拆分空序列将返回 [b''] 或 [bytearray(b'')]，具体取决于被拆分对象的类型。 *sep* 参数可以是任意 bytes-like object。

例如

```
>>> b'1,2,3'.split(b',')
[b'1', b'2', b'3']
>>> b'1,2,3'.split(b',', maxsplit=1)
[b'1', b'2,3']
>>> b'1,2,,3,'.split(b',')
[b'1', b'2', b'', b'3', b'']
```

如果 *sep* 未指定或为 `None`，则会应用另一种拆分算法：连续的 ASCII 空白符会被视为单个分隔符，其结果将不包含序列开头或末尾的空白符。 因此，在不指定分隔符的情况下对空序列或仅包含 ASCII 空白符的序列进行拆分将返回 `[]`。

例如

```
>>> b'1 2 3'.split()
[b'1', b'2', b'3']
>>> b'1 2 3'.split(maxsplit=1)
[b'1', b'2 3']
>>> b'   1   2   3   '.split()
[b'1', b'2', b'3']
```

bytes.**strip**([*chars*])
bytearray.**strip**([*chars*])

返回原序列的副本，移除指定的开头和末尾字节。 *chars* 参数为指定要移除字节值集合的二进制序列 —— 这个名称表明此方法通常是用于 ASCII 字符。 如果省略或为 `None`，则 *chars* 参数默认移除 ASCII 空白符。 *chars* 参数并非指定单个前缀或后缀；而是会移除参数值的所有组合:

```
>>> b'   spacious   '.strip()
b'spacious'
>>> b'www.example.com'.strip(b'cmowz.')
b'example'
```

要移除的字节值二进制序列可以是任意 bytes-like object。

> **注解:** 此方法的 bytearray 版本 *并非* 原地操作 —— 它总是产生一个新对象，即便没有做任何改变。

以下 bytes 和 bytearray 对象的方法会假定使用兼容 ASCII 的二进制格式，不应当被应用于任意二进制数据。 请注意本小节中所有的 bytearray 方法都 *不是* 原地执行操作，而是会产生新的对象。

bytes.**capitalize**()
bytearray.**capitalize**()

返回原序列的副本，其中每个字节将都将被解读为一个 ASCII 字符，并且第一个字节的字符大写而其余的小写。 非 ASCII 字节值将保持原样不变。

> **注解:** 此方法的 bytearray 版本 *并非* 原地操作 —— 它总是产生一个新对象，即便没有做任何改变。

bytes.**expandtabs**(*tabsize=8*)
bytearray.**expandtabs**(*tabsize=8*)

返回序列的副本，其中所有的 ASCII 制表符会由一个或多个 ASCII 空格替换，具体取决于当前列位置和给定的制表符宽度。 每 *tabsize* 个字节设为一个制表位（默认值 8 时设定的制表位在列 0, 8, 16 依次类推）。 要展开序列，当前列位置将被设为零并逐一检查序列中的每个字节。 如果字节为 ASCII 制表符 (`b'\t'`)，则并在结果中插入一个或多个空格符，直到当前列等于下一个制表位。 （制表符本身不会被复制。） 如果当前字节为 ASCII 换行符 (`b'\n'`) 或回车符 (`b'\r'`)，它会被复制并将当前列重设为零。 任何其他字节会被不加修改地复制并将当前列加一，不论该字节值在被打印时会如何显示:

```
>>> b'01\t012\t0123\t01234'.expandtabs()
b'01      012     0123    01234'
>>> b'01\t012\t0123\t01234'.expandtabs(4)
b'01  012 0123    01234'
```

注解: 此方法的 bytearray 版本 *并非* 原地操作 —— 它总是产生一个新对象，即便没有做任何改变。

bytes.**isalnum**()
bytearray.**isalnum**()

如果序列中所有字节都是字母类 ASCII 字符或 ASCII 十进制数码并且序列非空则返回 `Ture`，否则返回 `False`。 字母类 ASCII 字符就是字节值包含在序列 `b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'` 中的字符。 ASCII 十进制数码就是字节值包含在序列 `b'0123456789'` 中的字符。

例如

```
>>> b'ABCabc1'.isalnum()
True
>>> b'ABC abc1'.isalnum()
False
```

bytes.**isalpha**()
bytearray.**isalpha**()

如果序列中所有字节都是字母类 ASCII 字符并且序列不非空则返回 `True`，否则返回 `False`。 字母类 ASCII 字符就是字节值包含在序列 `b'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'` 中的字符。

例如

```
>>> b'ABCabc'.isalpha()
True
>>> b'ABCabc1'.isalpha()
False
```

bytes.**isascii**()
bytearray.**isascii**()

如果序列为空或序列中所有字节都是 ASCII 字节则返回 `True` ， 否则返回 `False` 。 ASCII 字节的取值范围是 0-0x7F。

3.7 新版功能.

bytes.**isdigit**()
bytearray.**isdigit**()

如果序列中所有字节都是 ASCII 十进制数码并且序列非空则返回 `True` ， 否则返回 `False` 。 ASCII 十进制数码就是字节值包含在序列 `b'0123456789'` 中的字符。

例如

```
>>> b'1234'.isdigit()
True
>>> b'1.23'.isdigit()
False
```

bytes.**islower**()
bytearray.**islower**()

如果序列中至少有一个小写的 ASCII 字符并且没有大写的 ASCII 字符则返回 `True` ， 否则返回 `False` 。

例如

```
>>> b'hello world'.islower()
True
>>> b'Hello world'.islower()
False
```

小写 ASCII 字符就是字节值包含在序列 `b'abcdefghijklmnopqrstuvwxyz'` 中的字符。 大写 ASCII 字符就是字节值包含在序列 `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'` 中的字符。

bytes.**isspace**()
bytearray.**isspace**()

如果序列中所有字节都是 ASCII 空白符并且序列非空则返回 `True` ， 否则返回 `False` 。 ASCII 空白符就是字节值包含在序列 `b' \t\n\r\x0b\f'` (空格, 制表, 换行, 回车, 垂直制表, 进纸) 中的字符。

bytes.**istitle**()
bytearray.**istitle**()

如果序列为 ASCII 标题大小写形式并且序列非空则返回 `True` ， 否则返回 `False` 。 请参阅 `bytes.title()` 了解有关“标题大小写”的详细定义。

例如

```
>>> b'Hello World'.istitle()
True
>>> b'Hello world'.istitle()
False
```

bytes.**isupper**()
bytearray.**isupper**()

如果序列中至少有一个大写字母 ASCII 字符并且没有小写 ASCII 字符则返回 `True` ，否则返回 `False` 。

例如

```
>>> b'HELLO WORLD'.isupper()
True
>>> b'Hello world'.isupper()
False
```

小写 ASCII 字符就是字节值包含在序列 `b'abcdefghijklmnopqrstuvwxyz'` 中的字符。
大写 ASCII 字符就是字节值包含在序列 `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'` 中的字符。

bytes.**lower**()
bytearray.**lower**()

返回原序列的副本，其所有大写 ASCII 字符均转换为对应的小写形式。

例如

```
>>> b'Hello World'.lower()
b'hello world'
```

小写 ASCII 字符就是字节值包含在序列 `b'abcdefghijklmnopqrstuvwxyz'` 中的字符。
大写 ASCII 字符就是字节值包含在序列 `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'` 中的字符。

> **注解:** 此方法的 bytearray 版本 *并非* 原地操作 —— 它总是产生一个新对象，即便没有做任何改变。

bytes.**splitlines**(*keepends=False*)
bytearray.**splitlines**(*keepends=False*)

返回由原二进制序列中各行组成的列表，在 ASCII 行边界符的位置拆分。 此方法使用 universal newlines 方式来分行。 结果列表中不包含换行符，除非给出了 *keepends* 且为真值。

例如

```
>>> b'ab c\n\nde fg\rkl\r\n'.splitlines()
[b'ab c', b'', b'de fg', b'kl']
>>> b'ab c\n\nde fg\rkl\r\n'.splitlines(keepends=True)
[b'ab c\n', b'\n', b'de fg\r', b'kl\r\n']
```

不同于 `split()`，当给出了分隔符 *sep* 时，对于空字符串此方法将返回一个空列表，而末尾的换行不会令结果中增加额外的行:

```
>>> b"".split(b'\n'), b"Two lines\n".split(b'\n')
([b''], [b'Two lines', b''])
```

```
>>> b"".splitlines(), b"One line\n".splitlines()
([], [b'One line'])
```

bytes.**swapcase**()
bytearray.**swapcase**()

返回原序列的副本，其所有小写 ASCII 字符均转换为对应的大写形式，反之亦反。

例如

```
>>> b'Hello World'.swapcase()
b'hELLO wORLD'
```

小写 ASCII 字符就是字节值包含在序列 `b'abcdefghijklmnopqrstuvwxyz'` 中的字符。大写 ASCII 字符就是字节值包含在序列 `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'` 中的字符。

不同于 `str.swapcase()`，在些二进制版本下 `bin.swapcase().swapcase() == bin` 总是成立。 大小写转换在 ASCII 中是对称的，即使其对于任意 Unicode 码位来说并不总是成立。

注解: 此方法的 bytearray 版本 *并非* 原地操作 —— 它总是产生一个新对象，即便没有做任何改变。

bytes.**title**()
bytearray.**title**()

返回原二进制序列的标题版本，其中每个单词以一个大写 ASCII 字符为开头，其余字母为小写。 不区别大小写的字节值将保持原样不变。

例如

```
>>> b'Hello world'.title()
b'Hello World'
```

小写 ASCII 字符就是字节值包含在序列 `b'abcdefghijklmnopqrstuvwxyz'` 中的字符。大写 ASCII 字符就是字节值包含在序列 `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'` 中的字符。所有其他字节值都不区分大小写。

该算法使用一种简单的与语言无关的定义，将连续的字母组合视为单词。 该定义在多数情况下都很有效，但它也意味着代表缩写形式与所有格的撇号也会成为单词边界，这可能导致不希望的结果:

```
>>> b"they're bill's friends from the UK".title()
b"They'Re Bill'S Friends From The Uk"
```

可以使用正则表达式来构建针对撇号的特别处理:

```
>>> import re
>>> def titlecase(s):
...     return re.sub(rb"[A-Za-z]+('[A-Za-z]+)?",
...                   lambda mo: mo.group(0)[0:1].upper() +
...                              mo.group(0)[1:].lower(),
```

```
...                     s)
...
>>> titlecase(b"they're bill's friends.")
b"They're Bill's Friends."
```

注解: 此方法的 bytearray 版本 *并非* 原地操作 —— 它总是产生一个新对象，即便没有做任何改变。

bytes.**upper**()
bytearray.**upper**()

返回原序列的副本，其所有小写 ASCII 字符均转换为对应的大写形式。

例如

```
>>> b'Hello World'.upper()
b'HELLO WORLD'
```

小写 ASCII 字符就是字节值包含在序列 `b'abcdefghijklmnopqrstuvwxyz'` 中的字符。大写 ASCII 字符就是字节值包含在序列 `b'ABCDEFGHIJKLMNOPQRSTUVWXYZ'` 中的字符。

注解: 此方法的 bytearray 版本 *并非* 原地操作 —— 它总是产生一个新对象，即便没有做任何改变。

bytes.**zfill**(*width*)
bytearray.**zfill**(*width*)

返回原序列的副本，在左边填充 `b'0'` 数码使序列长度为 *width*。 正负值前缀 (`b'+'` / `b'-'`) 的处理方式是在正负符号 *之后* 填充而非在之前。 对于 `bytes` 对象，如果 *width* 小于等于 `len(seq)` 则返回原序列。

例如

```
>>> b"42".zfill(5)
b'00042'
>>> b"-42".zfill(5)
b'-0042'
```

注解: 此方法的 bytearray 版本 *并非* 原地操作 —— 它总是产生一个新对象，即便没有做任何改变。

printf 风格的字节串格式化

注解: 此处介绍的格式化操作具有多种怪异特性，可能导致许多常见错误（例如无法正确显示元组和字典）。 如果要打印的值可能为元组或字典，请将其放入一个元组中。

字节串对象 (`bytes`/`bytearray`) 具有一种特殊的内置操作：使用 `%` (取模) 运算符。 这也被称为字节串的 *格式化* 或 *插值* 运算符。 对于 `format % values` (其中 *format* 为一个字节串对

象)，在 *format* 中的 `%` 转换标记符将被替换为零个或多个 *values* 条目。 其效果类似于在 C 语言中使用 `sprintf()`。

如果 *format* 要求一个单独参数，则 *values* 可以为一个非元组对象。 [5] 否则的话，*values* 必须或是是一个包含项数与格式字节串对象中指定的转换符项数相同的元组，或者是一个单独的映射对象（例如元组）。

转换标记符包含两个或更多字符并具有以下组成，且必须遵循此处规定的顺序：

1. `'%'` 字符，用于标记转换符的起始。
2. 映射键（可选），由加圆括号的字符序列组成 (例如 `(somename)`)。
3. 转换旗标（可选），用于影响某些转换类型的结果。
4. 最小字段宽度（可选）。 如果指定为 `'*'` (星号)，则实际宽度会从 *values* 元组的下一元素中读取，要转换的对象则为最小字段宽度和可选的精度之后的元素。
5. 精度（可选），以在 `'.'` (点号) 之后加精度值的形式给出。 如果指定为 `'*'` (星号)，则实际精度会从 *values* 元组的下一元素中读取，要转换的对象则为精度之后的元素。
6. 长度修饰符（可选）。
7. 转换类型。

当右边的参数为一个字典（或其他映射类型）时，字节串对象中的格式 *必须* 包含加圆括号的映射键，对应 `'%'` 字符之后字典中的每一项。 映射键将从映射中选取要格式化的值。 例如：

```
>>> print(b'%(language)s has %(number)03d quote types.' %
...       {b'language': b"Python", b"number": 2})
b'Python has 002 quote types.'
```

在此情况下格式中不能出现 `*` 标记符（因其需要一个序列类的参数列表）。

转换旗标为：

Flag	含义
`'#'`	值的转换将使用“替代形式”（具体定义见下文）。
`'0'`	转换将为数字值填充零字符。
`'-'`	转换值将靠左对齐（如果同时给出 `'0'` 转换，则会覆盖后者）。
`' '`	(空格) 符号位转换产生的正数（或空字符串）前将留出一个空格。
`'+'`	符号字符 (`'+'` 或 `'-'`) 将显示于转换结果的开头（会覆盖 "空格" 旗标）。

可以给出长度修饰符 (`h`, `l` 或 `L`)，但会被忽略，因为对 Python 来说没有必要 -- 所以 `%ld` 等价于 `%d`。

转换类型为：

转换符	含义	注释
`'d'`	有符号十进制整数。	
`'i'`	有符号十进制整数。	

转换符	含义	注释
'o'	有符号八进制数。	(1)
'u'	过时类型 -- 等价于 'd'。	(8)
'x'	有符号十六进制数（小写）。	(2)
'X'	有符号十六进制数（大写）。	(2)
'e'	浮点指数格式（小写）。	(3)
'E'	浮点指数格式（大写）。	(3)
'f'	浮点十进制格式。	(3)
'F'	浮点十进制格式。	(3)
'g'	浮点格式。 如果指数小于 -4 或不小于精度则使用小写指数格式，否则使用十进制格式。	(4)
'G'	浮点格式。 如果指数小于 -4 或不小于精度则使用大写指数格式，否则使用十进制格式。	(4)
'c'	单个字节（接受整数或单个字节对象）。	
'b'	字节串（任何遵循 缓冲区协议 或是具有 __bytes__() 的对象）。	(5)
's'	's' 是 'b' 的一个别名，只应当在基于 Python2/3 的代码中使用。	(6)
'a'	字节串（使用 repr(obj).encode('ascii','backslashreplace) 转换任何 Python 对象）。	(5)
'r'	'r' 是 'a' 的一个别名，只应当在基于 Python2/3 的代码中使用。	(7)
'%'	不转换参数，在结果中输出一个 '%' 字符。	

注释:

1. 此替代形式会在第一个数码之前插入标示八进制数的前缀 ('0o')。
2. 此替代形式会在第一个数码之前插入 '0x' 或 '0X' 前缀（取决于是使用 'x' 还是 'X' 格式）。
3. 此替代形式总是会在结果中包含一个小数点，即使其后并没有数码。

   小数点后的数码位数由精度决定，默认为 6。
4. 此替代形式总是会在结果中包含一个小数点，末尾各位的零不会如其他情况下那样被移除。

   小数点前后的有效数码位数由精度决定，默认为 6。
5. 如果精度为 N，输出将截短为 N 个字符。
6. b'%s' 已弃用，但在 3.x 系列中将不会被移除。
7. b'%r' 已弃用，但在 3.x 系列中将不会被移除。

8. 参见 **PEP 237**。

注解: 此方法的 bytearray 版本 *并非* 原地操作 —— 它总是产生一个新对象，即便没有做任何改变。

参见: **PEP 461** - 为 bytes 和 bytearray 添加 % 格式化

3.5 新版功能.

内存视图

`memoryview` 对象允许 Python 代码访问一个对象的内部数据，只要该对象支持 缓冲区协议 而无需进行拷贝。

class **memoryview**(*obj*)

创建一个引用 *obj* 的 `memoryview`。 *obj* 必须支持缓冲区协议。 支持缓冲区协议的内置对象包括 `bytes` 和 `bytearray`。

`memoryview` 具有 *元素* 的概念，即由原始对象 *obj* 所处理的基本内存单元。 对于许多简单类型例如 `bytes` 和 `bytearray` 来说，一个元素就是一个字节，但是其他的类型例如 `array.array` 可能有更大的元素。

`len(view)` 与 `tolist` 的长度相等。 如果 `view.ndim = 0`，则其长度为 1。 如果 `view.ndim = 1`，则其长度等于 view 中元素的数量。 对于更高的维度，其长度等于表示 view 的嵌套列表的长度。 `itemsize` 属性可向你给出单个元素所占的字节数。

`memoryview` 支持通过切片和索引访问其元素。 一维切片的结果将是一个子视图:

```
>>> v = memoryview(b'abcefg')
>>> v[1]
98
>>> v[-1]
103
>>> v[1:4]
<memory at 0x7f3ddc9f4350>
>>> bytes(v[1:4])
b'bce'
```

如果 `format` 是一个来自于 `struct` 模块的原生格式说明符，则也支持使用整数或由整数构成的元组进行索引，并返回具有正确类型的单个 *元素*。 一维内存视图可以使用一个整数或由一个整数构成的元组进行索引。 多维内存视图可以使用由恰好 *ndim* 个整数构成的元素进行索引，*ndim* 即其维度。 零维内存视图可以使用空元组进行索引。

这里是一个使用非字节格式的例子:

```
>>> import array
>>> a = array.array('l', [-11111111, 22222222, -33333333, 44444444])
>>> m = memoryview(a)
>>> m[0]
```

```
-11111111
>>> m[-1]
44444444
>>> m[::2].tolist()
[-11111111, -33333333]
```

如果下层对象是可写的，则内存视图支持一维切片赋值。 改变大小则不被允许:

```
>>> data = bytearray(b'abcefg')
>>> v = memoryview(data)
>>> v.readonly
False
>>> v[0] = ord(b'z')
>>> data
bytearray(b'zbcefg')
>>> v[1:4] = b'123'
>>> data
bytearray(b'z123fg')
>>> v[2:3] = b'spam'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: memoryview assignment: lvalue and rvalue have different
>>> v[2:6] = b'spam'
>>> data
bytearray(b'z1spam')
```

由带有格式符号 'B', 'b' 或 'c' 的可哈希（只读）类型构成的一维内存视图同样是可哈希的。 哈希定义为 `hash(m) == hash(m.tobytes())`:

```
>>> v = memoryview(b'abcefg')
>>> hash(v) == hash(b'abcefg')
True
>>> hash(v[2:4]) == hash(b'ce')
True
>>> hash(v[::-2]) == hash(b'abcefg'[::-2])
True
```

在 3.3 版更改: 一维内存视图现在可以被切片。 带有格式符号 'B', 'b' 或 'c' 的一维内存视图现在是可哈希的。

在 3.4 版更改: 内存视图现在会自动注册为 `collections.abc.Sequence`

在 3.5 版更改: 内存视图现在可使用整数元组进行索引。

`memoryview` 具有以下一些方法:

__eq__(*exporter*)

memoryview 与 **PEP 3118** 中的导出器这两者如果形状相同，并且如果当使用 `struct` 语法解读操作数的相应格式代码时所有对应值都相同，则它们就是等价的。

对于 `tolist()` 当前所支持的 `struct` 格式字符串子集，如果 `v.tolist() == w.tolist()` 则 `v` 和 `w` 相等:

```
>>> import array
>>> a = array.array('I', [1, 2, 3, 4, 5])
>>> b = array.array('d', [1.0, 2.0, 3.0, 4.0, 5.0])
>>> c = array.array('b', [5, 3, 1])
>>> x = memoryview(a)
>>> y = memoryview(b)
>>> x == a == y == b
True
>>> x.tolist() == a.tolist() == y.tolist() == b.tolist()
True
>>> z = y[::-2]
>>> z == c
True
>>> z.tolist() == c.tolist()
True
```

如果两边的格式字符串都不被 `struct` 模块所支持，则两对象比较结果总是不相等（即使格式字符串和缓冲区内容相同）：

```
>>> from ctypes import BigEndianStructure, c_long
>>> class BEPoint(BigEndianStructure):
...     _fields_ = [("x", c_long), ("y", c_long)]
...
>>> point = BEPoint(100, 200)
>>> a = memoryview(point)
>>> b = memoryview(point)
>>> a == point
False
>>> a == b
False
```

请注意，与浮点数的情况一样，对于内存视图对象来说，`v is w` 也 *并不* 意味着 `v == w`。

在 3.3 版更改: 之前的版本比较原始内存时会忽略条目的格式与逻辑数组结构。

tobytes(*order=None*)

将缓冲区中的数据作为字节串返回。 这相当于在内存视图上调用 `bytes` 构造器。

```
>>> m = memoryview(b"abc")
>>> m.tobytes()
b'abc'
>>> bytes(m)
b'abc'
```

对于非连续数组，结果等于平面化表示的列表，其中所有元素都转换为字节串。 `tobytes()` 支持所有格式字符串，不符合 `struct` 模块语法的那些也包括在内。

3.8 新版功能: *order* 可以为 {'C', 'F', 'A'}。 当 *order* 为 'C' 或 'F' 时，原始数组的数据会被转换至 C 或 Fortran 顺序。 对于连续视图，'A' 会返回物理内存的精确副本。 特别地，内存中的 Fortran 顺序会被保留。对于非连续视图，数据会先被转换为 C 形式。 *order=None* 与 *order='C'* 是相同的。

hex([*sep*[, *bytes_per_sep*]])

返回一个字符串对象，其中分别以两个十六进制数码表示缓冲区里的每个字节。

```
>>> m = memoryview(b"abc")
>>> m.hex()
'616263'
```

3.5 新版功能.

在 3.8 版更改: 与 `bytes.hex()` 相似， `memoryview.hex()` 现在支持可选的 *sep* 和 *bytes_per_sep* 参数以在十六进制输出的字节之间插入分隔符。

tolist()

将缓冲区内的数据以一个元素列表的形式返回。

```
>>> memoryview(b'abc').tolist()
[97, 98, 99]
>>> import array
>>> a = array.array('d', [1.1, 2.2, 3.3])
>>> m = memoryview(a)
>>> m.tolist()
[1.1, 2.2, 3.3]
```

在 3.3 版更改: `tolist()` 现在支持 `struct` 模块语法中的所有单字符原生格式以及多维表示形式。

toreadonly()

返回 memoryview 对象的只读版本。 原始的 memoryview 对象不会被改变。

```
>>> m = memoryview(bytearray(b'abc'))
>>> mm = m.toreadonly()
>>> mm.tolist()
[89, 98, 99]
>>> mm[0] = 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot modify read-only memory
>>> m[0] = 43
>>> mm.tolist()
[43, 98, 99]
```

3.8 新版功能.

release()

释放由内存视图对象所公开的底层缓冲区。 许多对象在被视图所获取时都会采取特殊动作（例如，`bytearray` 将会暂时禁止调整大小）；因此，调用 release() 可以方便地尽早去除这些限制（并释放任何多余的资源）。

在此方法被调用后，任何对视图的进一步操作将引发 `ValueError` (`release()` 本身除外，它可以被多次调用):

```
>>> m = memoryview(b'abc')
>>> m.release()
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

使用 `with` 语句，可以通过上下文管理协议达到类似的效果:

```
>>> with memoryview(b'abc') as m:
...     m[0]
...
97
>>> m[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operation forbidden on released memoryview object
```

3.2 新版功能.

cast(*format*[, *shape*])

将内存视图转化为新的格式或形状。 *shape* 默认为 `[byte_length//new_itemsize]`，这意味着结果视图将是一维的。 返回值是一个新的内存视图，但缓冲区本身不会被复制。 支持的转化有 1D -> C-contiguous 和 C-contiguous -> 1D。

目标格式仅限于 `struct` 语法中的单一元素原生格式。 其中一种格式必须为字节格式 ('B', 'b' 或 'c')。 结果的字节长度必须与原始长度相同。

将 1D/long 转换为 1D/unsigned bytes:

```
>>> import array
>>> a = array.array('l', [1,2,3])
>>> x = memoryview(a)
>>> x.format
'l'
>>> x.itemsize
8
>>> len(x)
3
>>> x.nbytes
24
>>> y = x.cast('B')
>>> y.format
'B'
>>> y.itemsize
1
>>> len(y)
24
>>> y.nbytes
24
```

将 1D/unsigned bytes 转换为 1D/char:

```
>>> b = bytearray(b'zyz')
>>> x = memoryview(b)
>>> x[0] = b'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: memoryview: invalid value for format "B"
>>> y = x.cast('c')
>>> y[0] = b'a'
>>> b
bytearray(b'ayz')
```

将 1D/bytes 转换为 3D/ints 再转换为 1D/signed char:

```
>>> import struct
>>> buf = struct.pack("i"*12, *list(range(12)))
>>> x = memoryview(buf)
>>> y = x.cast('i', shape=[2,2,3])
>>> y.tolist()
[[[0, 1, 2], [3, 4, 5]], [[6, 7, 8], [9, 10, 11]]]
>>> y.format
'i'
>>> y.itemsize
4
>>> len(y)
2
>>> y.nbytes
48
>>> z = y.cast('b')
>>> z.format
'b'
>>> z.itemsize
1
>>> len(z)
48
>>> z.nbytes
48
```

将 1D/unsigned long 转换为 2D/unsigned long:

```
>>> buf = struct.pack("L"*6, *list(range(6)))
>>> x = memoryview(buf)
>>> y = x.cast('L', shape=[2,3])
>>> len(y)
2
>>> y.nbytes
48
>>> y.tolist()
[[0, 1, 2], [3, 4, 5]]
```

3.3 新版功能.

在 3.5 版更改: 当转换为字节视图时，源格式将不再受限。

还存在一些可用的只读属性：

obj

内存视图的下层对象:

```
>>> b  = bytearray(b'xyz')
>>> m = memoryview(b)
>>> m.obj is b
True
```

3.3 新版功能.

nbytes

`nbytes == product(shape) * itemsize == len(m.tobytes())`。 这是数组在连续表示时将会占用的空间总字节数。 它不一定等于 `len(m)`:

```
>>> import array
>>> a = array.array('i', [1,2,3,4,5])
>>> m = memoryview(a)
>>> len(m)
5
>>> m.nbytes
20
>>> y = m[::2]
>>> len(y)
3
>>> y.nbytes
12
>>> len(y.tobytes())
12
```

多维数组:

```
>>> import struct
>>> buf = struct.pack("d"*12, *[1.5*x for x in range(12)])
>>> x = memoryview(buf)
>>> y = x.cast('d', shape=[3,4])
>>> y.tolist()
[[0.0, 1.5, 3.0, 4.5], [6.0, 7.5, 9.0, 10.5], [12.0, 13.5, 15.0,
>>> len(y)
3
>>> y.nbytes
96
```

3.3 新版功能.

readonly

一个表明内存是否只读的布尔值。

format

一个字符串，包含视图中每个元素的格式（表示为 `struct` 模块样式）。 内存视图可以从具有任意格式字符串的导出器创建，但某些方法 (例如 `tolist()`) 仅限于原生的单元素格式。

在 3.3 版更改: 格式 `'B'` 现在会按照 struct 模块语法来处理。 这意味着 `memoryview(b'abc')[0] == b'abc'[0] == 97`。

itemsize

memoryview 中每个元素以字节表示的大小:

```
>>> import array, struct
>>> m = memoryview(array.array('H', [32000, 32001, 32002]))
>>> m.itemsize
2
>>> m[0]
32000
>>> struct.calcsize('H') == m.itemsize
True
```

ndim

一个整数，表示内存所代表的多维数组具有多少个维度。

shape

一个整数元组，通过 `ndim` 的长度值给出内存所代表的 N 维数组的形状。

在 3.3 版更改: 当 ndim = 0 时值为空元组而不再为 `None`。

strides

一个整数元组，通过 `ndim` 的长度给出以字节表示的大小，以便访问数组中每个维度上的每个元素。

在 3.3 版更改: 当 ndim = 0 时值为空元组而不再为 `None`。

suboffsets

供 PIL 风格的数组内部使用。 该值仅作为参考信息。

c_contiguous

一个表明内存是否为 C-contiguous 的布尔值。

3.3 新版功能.

f_contiguous

一个表明内存是否为 Fortran contiguous 的布尔值。

3.3 新版功能.

contiguous

一个表明内存是否为 contiguous 的布尔值。

3.3 新版功能.

集合类型 --- set, frozenset

set 对象是由具有唯一性的 hashable 对象所组成的无序多项集。 常见的用途包括成员检测、从序列中去除重复项以及数学中的集合类计算，例如交集、并集、差集与对称差集等等。 （关于其他容器对象请参看 `dict`, `list` 与 `tuple` 等内置类，以及 `collections` 模块。）

与其他多项集一样，集合也支持 `x in set`, `len(set)` 和 `for x in set`。 作为一种无序的多项集，集合并不记录元素位置或插入顺序。 相应地，集合不支持索引、切片或其他序列类的操作。

目前有两种内置集合类型，`set` 和 `frozenset`。 `set` 类型是可变的 --- 其内容可以使用 `add()` 和 `remove()` 这样的方法来改变。 由于是可变类型，它没有哈希值，且不能被用作字典的键或其他集合的元素。 `frozenset` 类型是不可变并且为 hashable --- 其内容在被创建后不能再改变；因此它可以被用作字典的键或其他集合的元素。

除了可以使用 `set` 构造器，非空的 set (不是 frozenset) 还可以通过将以逗号分隔的元素列表包含于花括号之内来创建，例如: `{'jack', 'sjoerd'}`。

两个类的构造器具有相同的作用方式：

class **set**([*iterable*])
class **frozenset**([*iterable*])

返回一个新的 set 或 frozenset 对象，其元素来自于 *iterable*。 集合的元素必须为 hashable。 要表示由集合对象构成的集合，所有的内层集合必须为 `frozenset` 对象。 如果未指定 *iterable*，则将返回一个新的空集合。

`set` 和 `frozenset` 的实例提供以下操作：

len(s)
返回集合 *s* 中的元素数量（即 *s* 的基数）。

x in s
检测 *x* 是否为 *s* 中的成员。

x not in s
检测 *x* 是否非 *s* 中的成员。

isdisjoint(*other*)
如果集合中没有与 *other* 共有的元素则返回 `True`。 当且仅当两个集合的交集为空集合时，两者为不相交集合。

issubset(*other*)
set <= other
检测是否集合中的每个元素都在 *other* 之中。

set < other
检测集合是否为 *other* 的真子集，即 `set <= other and set != other`。

issuperset(*other*)
set >= other
检测是否 *other* 中的每个元素都在集合之中。

set > other

检测集合是否为 *other* 的真超集，即 `set >= other and set != other`。

union(**others*)
set | other | ...

返回一个新集合，其中包含来自原集合以及 others 指定的所有集合中的元素。

intersection(**others*)
set & other & ...

返回一个新集合，其中包含原集合以及 others 指定的所有集合中共有的元素。

difference(**others*)
set - other - ...

返回一个新集合，其中包含原集合中在 others 指定的其他集合中不存在的元素。

symmetric_difference(*other*)
set ^ other

返回一个新集合，其中的元素或属于原集合或属于 *other* 指定的其他集合，但不能同时属于两者。

copy()

返回原集合的浅拷贝。

请注意，非运算符版本的 `union()`, `intersection()`, `difference()`，以及 `symmetric_difference()`, `issubset()` 和 `issuperset()` 方法会接受任意可迭代对象作为参数。相比之下，它们所对应的运算符版本则要求其参数为集合。这就排除了容易出错的构造形式例如 `set('abc') & 'cbs'`，而推荐可读性更强的 `set('abc').intersection('cbs')`。

`set` 和 `frozenset` 均支持集合与集合的比较。两个集合当且仅当每个集合中的每个元素均包含于另一个集合之内（即各为对方的子集）时则相等。一个集合当且仅当其为另一个集合的真子集（即为后者的子集但两者不相等）时则小于另一个集合。一个集合当且仅当其为另一个集合的真超集（即为后者的超集但两者不相等）时则大于另一个集合。

`set` 的实例与 `frozenset` 的实例之间基于它们的成员进行比较。例如 `set('abc') == frozenset('abc')` 返回 `True`，`set('abc') in set([frozenset('abc')])` 也一样。

子集与相等比较并不能推广为完全排序函数。例如，任意两个非空且不相交的集合不相等且互不为对方的子集，因此以下 *所有* 比较均返回 `False`: `a<b`, `a==b`, or `a>b`。

由于集合仅定义了部分排序（子集关系），因此由集合构成的列表 `list.sort()` 方法的输出并无定义。

集合的元素，与字典的键类似，必须为 hashable。

混合了 `set` 实例与 `frozenset` 的二进制位运算将返回与第一个操作数相同的类型。例如: `frozenset('ab') | set('bc')` 将返回 `frozenset` 的实例。

下表列出了可用于 `set` 而不能用于不可变的 `frozenset` 实例的操作：

update(**others*)
set |= other | ...
更新集合，添加来自 others 中的所有元素。

intersection_update(**others*)
set &= other & ...
更新集合，只保留其中在所有 others 中也存在的元素。

difference_update(**others*)
set -= other | ...
更新集合，移除其中也存在于 others 中的元素。

symmetric_difference_update(*other*)
set ^= other
更新集合，只保留存在于集合的一方而非共同存在的元素。

add(*elem*)
将元素 *elem* 添加到集合中。

remove(*elem*)
从集合中移除元素 *elem*。 如果 *elem* 不存在于集合中则会引发 `KeyError`。

discard(*elem*)
如果元素 *elem* 存在于集合中则将其移除。

pop()
从集合中移除并返回任意一个元素。 如果集合为空则会引发 `KeyError`。

clear()
从集合中移除所有元素。

请注意，非运算符版本的 `update()`, `intersection_update()`, `difference_update()` 和 `symmetric_difference_update()` 方法将接受任意可迭代对象作为参数。

请注意，`__contains__()`, `remove()` 和 `discard()` 方法的 *elem* 参数可能是一个 set。 为支持对一个等价的 frozenset 进行搜索，会根据 *elem* 临时创建一个该类型对象。

映射类型 --- `dict`

mapping 对象会将 hashable 值映射到任意对象。 映射属于可变对象。 目前仅有一种标准映射类型 *字典*。 （关于其他容器对象请参看 `list`, `set` 与 `tuple` 等内置类，以及 `collections` 模块。）

字典的键 *几乎* 可以是任何值。 非 hashable 的值，即包含列表、字典或其他可变类型的值（此类对象基于值而非对象标识进行比较）不可用作键。 数字类型用作键时遵循数字比较的一

般规则：如果两个数值相等 (例如 `1` 和 `1.0`) 则两者可以被用来索引同一字典条目。 （但是请注意，由于计算机对于浮点数存储的只是近似值，因此将其用作字典键是不明智的。）

字典可以通过将以逗号分隔的 `键: 值` 对列表包含于花括号之内来创建，例如：`{'jack': 4098, 'sjoerd': 4127}` 或 `{4098: 'jack', 4127: 'sjoerd'}`，也可以通过 `dict` 构造器来创建。

class **dict**(***kwarg*)
class **dict**(*mapping*, ***kwarg*)
class **dict**(*iterable*, ***kwarg*)

返回一个新的字典，基于可选的位置参数和可能为空的关键字参数集来初始化。

如果没有给出位置参数，将创建一个空字典。 如果给出一个位置参数并且其属于映射对象，将创建一个具有与映射对象相同键值对的字典。 否则的话，位置参数必须为一个 iterable 对象。 该可迭代对象中的每一项本身必须为一个刚好包含两个元素的可迭代对象。 每一项中的第一个对象将成为新字典的一个键，第二个对象将成为其对应的值。 如果一个键出现一次以上，该键的最后一个值将成为其在新字典中对应的值。

如果给出了关键字参数，则关键字参数及其值会被加入到基于位置参数创建的字典。 如果要加入的键已存在，来自关键字参数的值将替代来自位置参数的值。

作为演示，以下示例返回的字典均等于 `{"one": 1, "two": 2, "three": 3}`：

```
>>> a = dict(one=1, two=2, three=3)
>>> b = {'one': 1, 'two': 2, 'three': 3}
>>> c = dict(zip(['one', 'two', 'three'], [1, 2, 3]))
>>> d = dict([('two', 2), ('one', 1), ('three', 3)])
>>> e = dict({'three': 3, 'one': 1, 'two': 2})
>>> a == b == c == d == e
True
```

像第一个例子那样提供关键字参数的方式只能使用有效的 Python 标识符作为键。 其他方式则可使用任何有效的键。

这些是字典所支持的操作（因而自定义的映射类型也应当支持）：

list(d)
返回字典 *d* 中使用的所有键的列表。

len(d)
返回字典 *d* 中的项数。

d[key]
返回 *d* 中以 *key* 为键的项。 如果映射中不存在 *key* 则会引发 `KeyError`。

如果字典的子类定义了方法 `__missing__()` 并且 *key* 不存在，则 `d[key]` 操作将调用该方法并附带键 *key* 作为参数。 `d[key]` 随后将返回或引发 `__missing__(key)` 调用所返回或引发的任何对象或异常。 没有其他操作或方法会发起调用 `__missing__()`。 如果未定义 `__missing__()`，则会引发 `KeyError`。 `__missing__()` 必须是一个方法；它不能是一个实例变量：

```
>>> class Counter(dict):
...     def __missing__(self, key):
...         return 0
>>> c = Counter()
>>> c['red']
0
>>> c['red'] += 1
>>> c['red']
1
```

上面的例子显示了 `collections.Counter` 实现的部分代码。 还有另一个不同的 `__missing__` 方法是由 `collections.defaultdict` 所使用的。

d[key] = value

将 `d[key]` 设为 *value*。

del d[key]

将 `d[key]` 从 *d* 中移除。 如果映射中不存在 *key* 则会引发 `KeyError`。

key in d

如果 *d* 中存在键 *key* 则返回 `True`，否则返回 `False`。

key not in d

等价于 `not key in d`。

iter(d)

返回以字典的键为元素的迭代器。 这是 `iter(d.keys())` 的快捷方式。

clear()

移除字典中的所有元素。

copy()

返回原字典的浅拷贝。

classmethod **fromkeys**(*iterable*[, *value*])

使用来自 *iterable* 的键创建一个新字典，并将键值设为 *value*。

`fromkeys()` 是一个返回新字典的类方法。 *value* 默认为 `None`。 所有值都只引用一个单独的实例，因此让 *value* 成为一个可变对象例如空列表通常是没有意义的。 要获取不同的值，请改用 字典推导式。

get(*key*[, *default*])

如果 *key* 存在于字典中则返回 *key* 的值，否则返回 *default*。 如果 *default* 未给出则默认为 `None`，因而此方法绝不会引发 `KeyError`。

items()

返回由字典项 (`(键, 值)` 对) 组成的一个新视图。 参见 视图对象文档。

keys()

返回由字典键组成的一个新视图。 参见 视图对象文档。

pop(*key*[, *default*])

如果 *key* 存在于字典中则将其移除并返回其值，否则返回 *default*。 如果 *default* 未给出且 *key* 不存在于字典中，则会引发 `KeyError`。

popitem()

从字典中移除并返回一个 (键，值) 对。 键值对会按 LIFO 的顺序被返回。

`popitem()` 适用于对字典进行消耗性的迭代，这在集合算法中经常被使用。 如果字典为空，调用 `popitem()` 将引发 `KeyError`。

在 3.7 版更改: 现在会确保采用 LIFO 顺序。 在之前的版本中，`popitem()` 会返回一个任意的键/值对。

reversed(d)

返回一个逆序获取字典键的迭代器。 这是 `reversed(d.keys())` 的快捷方式。

3.8 新版功能.

setdefault(*key*[, *default*])

如果字典存在键 *key*，返回它的值。如果不存在，插入值为 *default* 的键 *key*，并返回 *default*。 *default* 默认为 `None`。

update([*other*])

使用来自 *other* 的键/值对更新字典，覆盖原有的键。 返回 `None`。

`update()` 接受另一个字典对象，或者一个包含键/值对（以长度为二的元组或其他可迭代对象表示）的可迭代对象。 如果给出了关键字参数，则会以其所指定的键/值对更新字典: `d.update(red=1, blue=2)`。

values()

返回由字典值组成的一个新视图。 参见 视图对象文档。

两个 `dict.values()` 视图之间的相等性比较将总是返回 `False`。 这在 `dict.values()` 与其自身比较时也同样适用:

```
>>> d = {'a': 1}
>>> d.values() == d.values()
False
```

两个字典的比较当且仅当它们具有相同的 (键，值) 对时才会相等（不考虑顺序）。 排序比较 ('<', '<=', '>=', '>') 会引发 `TypeError`。

字典会保留插入时的顺序。 请注意对键的更新不会影响顺序。 删除并再次添加的键将被插入到末尾。

```
>>> d = {"one": 1, "two": 2, "three": 3, "four": 4}
>>> d
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> list(d)
['one', 'two', 'three', 'four']
```

```
>>> list(d.values())
[1, 2, 3, 4]
>>> d["one"] = 42
>>> d
{'one': 42, 'two': 2, 'three': 3, 'four': 4}
>>> del d["two"]
>>> d["two"] = None
>>> d
{'one': 42, 'three': 3, 'four': 4, 'two': None}
```

在 3.7 版更改: 字典顺序会确保为插入顺序。 此行为是自 3.6 版开始的 CPython 实现细节。

字典和字典视图都是可逆的。

```
>>> d = {"one": 1, "two": 2, "three": 3, "four": 4}
>>> d
{'one': 1, 'two': 2, 'three': 3, 'four': 4}
>>> list(reversed(d))
['four', 'three', 'two', 'one']
>>> list(reversed(d.values()))
[4, 3, 2, 1]
>>> list(reversed(d.items()))
[('four', 4), ('three', 3), ('two', 2), ('one', 1)]
```

在 3.8 版更改: 字典现在是可逆的。

参见: `types.MappingProxyType` 可被用来创建一个 `dict` 的只读视图。

字典视图对象

由 `dict.keys()`, `dict.values()` 和 `dict.items()` 所返回的对象是 *视图对象*。 该对象提供字典条目的一个动态视图，这意味着当字典改变时，视图也会相应改变。

字典视图可以被迭代以产生与其对应的数据，并支持成员检测：

len(dictview)

返回字典中的条目数。

iter(dictview)

返回字典中的键、值或项（以 (键, 值) 为元素的元组表示）的迭代器。

键和值是按插入时的顺序进行迭代的。 这样就允许使用 `zip()` 来创建 (值, 键) 对: `pairs = zip(d.values(), d.keys())`。 另一个创建相同列表的方式是 `pairs = [(v, k) for (k, v) in d.items()]`.

在添加或删除字典中的条目期间对视图进行迭代可能引发 `RuntimeError` 或者无法完全迭代所有条目。

在 3.7 版更改: 字典顺序会确保为插入顺序。

x in dictview

如果 *x* 是对应字典中存在的键、值或项（在最后一种情况下 *x* 应为一个 (键, 值) 元组）则返回 `True`。

reversed(dictview)

返回一个逆序获取字典键、值或项的迭代器。 视图将按与插入时相反的顺序进行迭代。

在 3.8 版更改: 字典视图现在是可逆的。

键视图类似于集合，因为其条目不重复且可哈希。 如果所有值都是可哈希的，即 (键, 值) 对也是不重复且可哈希的，那么条目视图也会类似于集合。 （值视图则不被视为类似于集合，因其条目通常都是有重复的。） 对于类似于集合的视图，为抽象基类 `collections.abc.Set` 所定义的全部操作都是有效的 (例如 `==`, `<` 或 `^`)。

一个使用字典视图的示例:

```
>>> dishes = {'eggs': 2, 'sausage': 1, 'bacon': 1, 'spam': 500}
>>> keys = dishes.keys()
>>> values = dishes.values()

>>> # iteration
>>> n = 0
>>> for val in values:
...     n += val
>>> print(n)
504

>>> # keys and values are iterated over in the same order (insertion orc
>>> list(keys)
['eggs', 'sausage', 'bacon', 'spam']
>>> list(values)
[2, 1, 1, 500]

>>> # view objects are dynamic and reflect dict changes
>>> del dishes['eggs']
>>> del dishes['sausage']
>>> list(keys)
['bacon', 'spam']

>>> # set operations
>>> keys & {'eggs', 'bacon', 'salad'}
{'bacon'}
>>> keys ^ {'sausage', 'juice'}
{'juice', 'sausage', 'bacon', 'spam'}
```

上下文管理器类型

Python 的 `with` 语句支持通过上下文管理器所定义的运行时上下文这一概念。 此对象的实现使用了一对专门方法，允许用户自定义类来定义运行时上下文，在语句体被执行前进入该上下文，并在语句执行完毕时退出该上下文:

`contextmanager.__enter__()`

进入运行时上下文并返回此对象或关联到该运行时上下文的其他对象。 此方法的返回值会绑定到使用此上下文管理器的 `with` 语句的 `as` 子句中的标识符。

一个返回其自身的上下文管理器的例子是 file object。 文件对象会从 __enter__() 返回其自身，以允许 `open()` 被用作 `with` 语句中的上下文表达式。

一个返回关联对象的上下文管理器的例子是 `decimal.localcontext()` 所返回的对象。 此种管理器会将活动的 decimal 上下文设为原始 decimal 上下文的一个副本并返回该副本。 这允许对 `with` 语句的语句体中的当前 decimal 上下文进行更改，而不会影响 `with` 语句以外的代码。

contextmanager.**__exit__**(*exc_type*, *exc_val*, *exc_tb*)

退出运行时上下文并返回一个布尔值旗标来表明所发生的任何异常是否应当被屏蔽。 如果在执行 `with` 语句的语句体期间发生了异常，则参数会包含异常的类型、值以及回溯信息。 在其他情况下三个参数均为 `None`。

自此方法返回一个真值将导致 `with` 语句屏蔽异常并继续执行紧随在 `with` 语句之后的语句。 否则异常将在此方法结束执行后继续传播。 在此方法执行期间发生的异常将会取代 `with` 语句的语句体中发生的任何异常。

传入的异常绝对不应当被显式地重新引发 —— 相反地，此方法应当返回一个假值以表明方法已成功完成并且不希望屏蔽被引发的异常。 这允许上下文管理代码方便地检测 `__exit__()` 方法是否确实已失败。

Python 定义了一些上下文管理器来支持简易的线程同步、文件或其他对象的快速关闭，以及更方便地操作活动的十进制算术上下文。 除了实现上下文管理协议以外，不同类型不会被特殊处理。 请参阅 `contextlib` 模块查看相关的示例。

Python 的 generator 和 `contextlib.contextmanager` 装饰器提供了实现这些协议的便捷方式。 如果使用 `contextlib.contextmanager` 装饰器来装饰一个生成器函数，它将返回一个实现了必要的 `__enter__()` and `__exit__()` 方法的上下文管理器，而不再是由未经装饰的生成器函数所产生的迭代器。

请注意，Python/C API 中 Python 对象的类型结构中并没有针对这些方法的专门槽位。 想要定义这些方法的扩展类型必须将它们作为普通的 Python 可访问方法来提供。 与设置运行时上下文的开销相比，单个类字典查找的开销可以忽略不计。

其他内置类型

解释器支持一些其他种类的对象。 这些对象大都仅支持一两种操作。

模块

模块唯一的特殊操作是属性访问: `m.name`，这里 *m* 为一个模块而 *name* 访问定义在 *m* 的符号表中的一个名称。 模块属性可以被赋值。 （请注意 `import` 语句严格来说也是对模块对象的一种操作；`import foo` 不要求存在一个名为 *foo* 的模块对象，而是要求存在一个对于名为 *foo* 的模块的 (永久性) *定义*。）

每个模块都有一个特殊属性 `__dict__`。 这是包含模块的符号表的字典。 修改此字典将实际改变模块的符号表，但是无法直接对 `__dict__` 赋值 (你可以写 `m.__dict__['a'] = 1`，这会将 `m.a` 定义为 `1`，但是你不能写 `m.__dict__ = {}`)。 不建议直接修改 `__dict__`。

内置于解释器中的模块会写成这样: `<module 'sys' (built-in)>`。 如果是从一个文件加载，则会写成 `<module 'os' from '/usr/local/lib/pythonX.Y/os.pyc'>`。

类与类实例

关于这些类型请参阅 对象、值与类型 和 类定义。

函数

函数对象是通过函数定义创建的。 对函数对象的唯一操作是调用它: `func(argument-list)`。

实际上存在两种不同的函数对象：内置函数和用户自定义函数。 两者支持同样的操作（调用函数），但实现方式不同，因此对象类型也不同。

更多信息请参阅 函数定义。

方法

方法是使用属性表示法来调用的函数。 存在两种形式：内置方法（例如列表的 `append()` 方法）和类实例方法。 内置方法由支持它们的类型来描述。

如果你通过一个实例来访问方法（即定义在类命名空间内的函数），你会得到一个特殊对象: *绑定方法* (或称 *实例方法*) 对象。 当被调用时，它会将 `self` 参数添加到参数列表。 绑定方法具有两个特殊的只读属性: `m.__self__` 操作该方法的对象，而 `m.__func__` 是实现该方法的函数。 调用 `m(arg-1, arg-2, ..., arg-n)` 完全等价于调用 `m.__func__(m.__self__, arg-1, arg-2, ..., arg-n)`。

与函数对象类似，绑定方法对象也支持获取任意属性。 但是，由于方法属性实际上保存于下层的函数对象中 (`meth.__func__`)，因此不允许设置绑定方法的方法属性。 尝试设置方法的属性将会导致引发 `AttributeError`。 想要设置方法属性，你必须在下层的函数对象中显式地对其进行设置:

```
>>> class C:
...     def method(self):
...         pass
...
>>> c = C()
>>> c.method.whoami = 'my name is method'  # can't set on the method
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'method' object has no attribute 'whoami'
>>> c.method.__func__.whoami = 'my name is method'
```

```
>>> c.method.whoami
'my name is method'
```

更多信息请参阅 标准类型层级结构。

代码对象

代码对象被具体实现用来表示“伪编译”的可执行 Python 代码，例如一个函数体。 它们不同于函数对象，因为它们不包含对其全局执行环境的引用。 代码对象由内置的 `compile()` 函数返回，并可通过从函数对象的 `__code__` 属性从中提取。 另请参阅 `code` 模块。

可以通过将代码对象（而非源码字符串）传给 `exec()` 或 `eval()` 内置函数来执行或求值。

更多信息请参阅 标准类型层级结构。

类型对象

类型对象表示各种对象类型。 对象的类型可通过内置函数 `type()` 来获取。 类型没有特殊的操作。 标准库模块 `types` 定义了所有标准内置类型的名称。

类型以这样的写法来表示: `<class 'int'>`。

空对象

此对象会由不显式地返回值的函数所返回。 它不支持任何特殊的操作。 空对象只有一种值 `None` (这是个内置名称)。 `type(None)()` 会生成同一个单例。

该对象的写法为 `None`。

省略符对象

此对象常被用于切片 (参见 切片)。 它不支持任何特殊的操作。 省略符对象只有一种值 `Ellipsis` (这是个内置名称)。 `type(Ellipsis)()` 会生成 `Ellipsis` 单例。

该对象的写法为 `Ellipsis` 或 `...`。

未实现对象

此对象会被作为比较和二元运算被应用于它们所不支持的类型时的返回值。 请参阅 比较运算 了解更多信息。 未实现对象只有一种值 `NotImplemented`。 `type(NotImplemented)()` 会生成这个单例。

该对象的写法为 `NotImplemented`。

布尔值

布尔值是两个常量对象 `False` 和 `True`。 它们被用来表示逻辑上的真假（不过其他值也可被当作真值或假值）。 在数字类的上下文中（例如被用作算术运算符的参数时），它们的行为分别类似于整数 0 和 1。 内置函数 `bool()` 可被用来将任意值转换为布尔值，只要该值可被解析为一个逻辑值（参见之前的 逻辑值检测 部分）。

该对象的写法分别为 `False` 和 `True`。

内部对象

有关此对象的信息请参阅 标准类型层级结构。 其中描述了栈帧对象、回溯对象以及切片对象等等。

特殊属性

语言实现为部分对象类型添加了一些特殊的只读属性，它们具有各自的作用。 其中一些并不会被 `dir()` 内置函数所列出。

`object.`**`__dict__`**
: 一个字典或其他类型的映射对象，用于存储对象的（可写）属性。

`instance.`**`__class__`**
: 类实例所属的类。

`class.`**`__bases__`**
: 由类对象的基类所组成的元组。

`definition.`**`__name__`**
: 类、函数、方法、描述器或生成器实例的名称。

`definition.`**`__qualname__`**
: 类、函数、方法、描述器或生成器实例的 qualified name。

3.3 新版功能.

`class.`**`__mro__`**
: 此属性是由类组成的元组，在方法解析期间会基于它来查找基类。

`class.`**`mro`**()
: 此方法可被一个元类来重载，以为其实例定制方法解析顺序。 它会在类实例化时被调用，其结果存储于 `__mro__` 之中。

`class.`**`__subclasses__`**()
: 每个类会保存由对其直接子类的弱引用组成的列表。 此方法将返回一个由仍然存在的所有此类引用组成的列表。 例如:

```
>>> int.__subclasses__()
[<class 'bool'>]
```

备注

[1] 有关这些特殊方法的额外信息可参看 Python 参考指南 (基本定制)。

[2] 作为结果，列表 `[1, 2]` 与 `[1.0, 2.0]` 是相等的，元组的情况也类似。

[3] 它们必须如此，因为解析器无法区分这些操作数的类型。

4(1,2,3,4)区分大小写的字符是指所属一般类别属性为 "Lu" (Letter, uppercase), "Ll" (Letter, lowercase) 或 "Lt" (Letter, titlecase) 之一的字符。

5(1,2)要格式化单独一个元组，那么你应当提供一个单例元组，其唯一的元素就是要被格式化的元组。