第9章. 数字

截止上一章，我们已经讨论完了Python程序的框架和内在逻辑。从本章开始直到第13章，我们将注意力放在Python语法的细节上，且会介绍更多的编程技巧。

第3章已经说明Python中的数字也是对象。在前面的例子中我们已经多次与数字打交道，但到目前为止并没有系统地讨论代表数字的类型，以及能够通过它们的属性对数字施加的操作。本章将弥补这些缺失的知识点。

9-1. 数字的表示

（教程：3.1.1、15）
（语言参考手册：2.4.4～2.4.7）
（标准库：内置函数、内置类型、sys）

Python提供了内置类型int、float和complex，分别用来表示整数、浮点数和复数。第3章已经提到了，整数可以通过内置函数int()创建，浮点数可以通过内置函数float()创建，复数可以通过内置函数complex()创建，下面会详细讨论这三个内置函数。此外，第3章还说明了内置类型bool是int的子类，布尔值可通过内置函数bool()创建，而bool()本质上是对传入的对象进行逻辑值检测。

---

int()的第一种语法与bool()类似，为：

```
class int(object=0)
```

其功能相当于将通过object参数指定的对象“强制类型转换（type cast）”为整数，具体规则为：

- 如果该对象具有__int__魔术属性，则调用它。
- 否则，如果该对象具有__index__魔术属性，则调用它。
- 判断是否可以将其理解为int()的第二种语法，如果是则转向它。
- 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，则得到整数0。

int()的第二种语法将整数的指定进制字符串表示法转换为整数，为：

```
class int(string, base=10)
```

其中string参数只能是一个字符串、字节串或字节数组；而base参数应是一个整数或一个并非整数但具有__index__魔术属性的对象，不论哪种情况最后都只能在2～36这个范围内，或者是特殊值0。整数的指定进制字符串表示法能够以“+”或“-”开头。当base参数是2～36范围内的整数时，后续的字符只能是该进制下的数字：n进制的数字为0～n-1，当n大于10时用a～z或A～Z表示数字10～35。当base参数是0时，后续的字符将被解析为一个“整数字面值（integer literals）”。

第2章曾经提到过数字字面值，而整数字面值是数字字面值的一种，本质上就是一个具有下列格式的令牌：

```
[prefix[_]]digit([_]digit)*
```

其中prefix代表表明进制的前缀，可以是：

- “0b”或“0B”：表明采用二进制。
- “0o”或“0O”：表明采用八进制。
- “0x”或“0X”：表明采用十六进制。

如果省略了前缀，则表明采用十进制。digit则代表该进制下的某个数字，二进制下只能是0或1，八进制下可以是0～7，十进制下可以是0～9，十六进制下则是0～f（或0～F）。而可选的下划线是为了提高可读性而被插入的，用于对数字分组，遵循如下惯例：

- 二进制：以4个数字为一组，从右向左分组，前缀后也添加“_”。
- 八进制：以2个数字为一组，从右向左分组，前缀后也添加“_”。
- 十进制：以3个数字为一组，从右向左分组。
- 十六进制：以2个数字为一组，从右向左分组，前缀后也添加“_”。

此外需要注意，十进制整数字面值不能以“0”开头。

下面是一些二进制整数字面值的例子：

```
0b0         0B1         0b10110           0B_101_1010_0101
```

下面是一些八进制整数字面值的例子：

```
0o0         0O7         0o26              0O_6_25_30_47
```

下面是一些十进制整数字面值的例子：

```
0           9           206               9_839_647_243_766
```

下面是一些十六进制整数字面值的例子：

```
0x0         0Xf         0xf09a8           0X_b_87_cf_2a_c5
```

下面给出一些通过int()创建整数的例子。在所有内置类型中，只有bool、int、和float实现了魔术属性__int__（此外，int还实现了__index__，而float的__int__其实就等价于__trunc__），因此int()的第一种语法默认仅能作用于它们。请通过如下命令行和语句验证：

```
$ python3

>>> int(False)
0
>>> int(True)
1
>>> int(0b1010)
10
>>> int(0O76)
62
>>> int(1000)
1000
>>> int(0xffff)
65535
>>> int(0.41)
0
>>> int(783.8)
783
>>> int()
0
>>>
```

注意在这些例子中被传入的布尔值、整数和浮点数都是字面值形式，且后两者都不是负数，因为数字字面值不能有正负号，给它们添加了“+”或“-”后其实得到了算术表达式（会在下一节讨论）。

下面是int()的第二种语法的一些例子：

```
>>> int("-1345", 6)
-353
>>> int(b"ZZZ", 36)
46655
>>> int("-0x9f8", 0)
-2552
>>> int("0b101", 0)
```

```
    5
    >>> int("10_045_287")
    10045287
    >>>
```

内置函数bin()、oct()和hex()可以看成int()的第二种语法的逆操作，即分别获得整数的二进制、八进制和十六进制字符串表示。它们的语法为：

```
bin(x)
oct(x)
hex(x)
```

注意如果x参数传入的不是一个整数，则会尝试调用该对象的__index__魔术属性。下面是一些例子：

```
>>> bin(205)
'0b11001101'
>>> oct(205)
'0o315'
>>> hex(205)
'0xcd'
>>> bin(-33)
'-0b100001'
>>> oct(-33)
'-0o41'
>>> hex(-33)
'-0x21'
>>>
```

与其他很多编程语言（例如C）中的整数类型仅能表示有限范围的整数不同，Python 3中的整数类型没有范围限制。（但Python 2中的整数类型与C中的long类型相同。）这意味着，一个整数字面值理论上可以大到耗尽整个内存空间。在其内部实现中，Python 3将整数视为一个由抽象数字形成的序列。通过表9-1列出的sys属性可以了解到一个这样的抽象数字能表示的范围和占用的内存空间。当然，对于绝大部分Python程序来说，没有必要了解整数在Python解释器内部是如何实现的。

表9-1. 整数实现相关sys属性

属性	说明
sys.int_info	一个具名元组，提供整数在Python解释器内部的表示形式相关信息。

具体来说，sys.int_info引用的具名元组包含如下两个元素：

- bits_per_digit：抽象数字的有效位数，因此其能表示的范围是0～2^bits_per_digit-1。
- sizeof_digit：存储一个抽象数字需要的字节数。

请执行如下语句：

```
>>> import sys
>>> sys.int_info
sys.int_info(bits_per_digit=30, sizeof_digit=4)
>>>
```

这说明在Python 3中一个抽象数字占用4字节，表示范围是0～2^30-1。

与int()类似，float()的第一种语法为：

```
class float(object=0.0)
```

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

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

如果省略了object参数，则得到浮点数0.0。

float()的第二种语法将实数的字符串表示法转换为浮点数，为：

```
class float(string)
```

其中string参数同样只能是字符串、字节串或字节数组。实数的字符串表示法能够以“+”或“-”开头，后续的字符被理解为一个“浮点数字面值（floating point literals）”。

作为数字字面值的一种，像整数字面值一样，浮点数字面值不包括正负号。但浮点数字面值有两种格式。

第一种格式符合人们日常生活中对实数的表述：

```
[digit([_]digit)*].[digit([_]digit)*]
```

该格式中必然有小数点“.”，其左侧是整数部分，右侧是小数部分，两者可能同时存在，也可能仅有一方存在。

第二种格式符合实数的科学计数法的表述：

```
[digit([_]digit)*][.digit([_]digit)*](e|E)exponent
```

该格式中必然有“e”或“E”，其左侧可以是一个十进制整数字面值，也可以是浮点数字面值的第一种格式，右侧则是一个十进制整数（可以包含正负号）。

下面是浮点数字面值第一种格式的一些例子：

```
0.0         1.          .0001       3.14_15_93
```

而下面则是浮点数字面值第二种格式的一些例子：

```
9e6         0.09E-7     123.e-2     0E0
```

注意浮点数字面值只能采用十进制，因此字面值中的所有数字都只能在0～9这个范围内。

在所有内置类型中，只有bool、int和float实现了魔术属性__float__，因此float()的第一种语法默认仅能作用于它们。请通过如下命令行和语句验证：

```
>>> float(True)
1.0
>>> float(False)
0.0
>>> float(0)
0.0
>>> float(1)
1.0
>>> float(3.14_15_93)
3.141593
>>> float(0.09E-7)
9e-09
>>>
```

下面则是float()的第二种语法的一些例子：

```
>>> float('0.0')
0.0
>>> float(b'1.')
1.0
>>> float('123.e-2')
1.23
>>>
```

需要特别强调的是，与整数没有范围限制且永远精确不同，浮点数有范围限制且是不精确的。请先看下面这个例子：

```
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
>>> 0.1 * 3 == 0.3
False
>>>
```

该结果说明涉及浮点数的运算并不严格与数学中的实数运算相一致。

出现上面这种现象的原因是，整数只是向+∞和-∞这两个方向无限延伸，因此只要使用逻辑上可以无限扩展的抽象数字构成的序列，就可以精确表示每一个整数；而实数除了向+∞和-∞这两个方向无限延伸外，还向+0和-0这两个方向无限延伸，因为小数点后可以有无穷多位。因此，在计算机中是无法精确表示每一个实数的，目前被广泛采用的实数编码方式是IEEE 754标准，该编码只能精确表示实数中的很小一部分，故被称为“浮点数”以示区分。

IEEE754标准定义了3种浮点数精度：“单精度（single precision）”对应Fortran中的REAL*4类型和C中的float类型；“双精度（double precision）”对应Fortran中的REAL*8类型和C中的double类型；“扩展双精度（double-extended precision）”对应Fortran中的REAL*10类型和C中的long double类型。而Python中的浮点数则总是采用双精度。

具体来说，一个双精度浮点数用8字节表示，也就是64位，其中第一位是用于区分正实数和负实数的“符号位（sign bit）”，后续的11位是“指数（exponent）”部分，最后的52位代表“尾数（mantissa）”部分。指数部分和尾数部分都被视为一个二进制整数字面值。如果用s代表符号位（取值+1和-1），用e代表指数，m代表尾数，则一个浮点数所精确表示的实数可以通过如下公式计算：

$$s \times m \times 2^e$$

但e并不等于指数部分的字面值，m并不等于尾数部分的字面值，s也并非总是有效，而是需要根据表9-2所示的解读方式进行解读。

表9-2. IEEE 754双精度浮点数编码的解读

类别	指数部分	尾数部分	说明
规格化数	$1\sim2^{11}-2$	$0\sim2^{52}-1$	e的值等于指数字面值减去$2^{10}-1$。m的值等于尾数字面值除以2^{52}再加上1。s有效。
非规格化数	0	$1\sim2^{52}-1$	e的值等于$2-2^{10}$。m的值等于尾数字面值除以2^{52}。s有效。
零		0	e的值等于0。m的值等于0。s有效。
无穷	$2^{11}-1$	0	e和m无意义。s有效。
非数		$1\sim2^{52}-1$	e、m和s都无意义。

从表9-2可以看出，IEEE 754标准规定的浮点数编码解读起来比较复杂。但我们可以简单记住一个常识，即Python中的浮点数分为五类：

➤ “规格化数（normals）”：指数的范围是-1022~1023，尾数的范围是[1, 2)（以1/252为间隔的长度），支持正负号。
➤ “非规格化数（subnormals）”：指数是-1022，尾数的范围是(0, 1)（以1/252为间隔的长度），支持正负号。
➤ “零（zeros）”：指数是0，尾数也是0，支持正负号。因此需要区分+0和-0。
➤ “无穷（infinites）”：指数是和尾数无意义，但支持正负号。因此需要区分+∞和-∞，在计算机科学中通常用+INF和-INF表示。
➤ “非数（not a number）”：指数、尾数和符号都无意义。在计算机科学中通常用NaN表示。

综上所述，浮点数只能精确表示整个实数轴中的很小一部分，这些点是离散分布的，越靠近0越密集，越远离0越稀疏。在$(-2^{-1022}, 2^{-1022})$这个范围内，具有最高精度$2^{-1022}/2^{52} = 2^{-1074} \approx 4.94\times10^{-324}$，这被称为“机器精度”。而浮点数能够精确表示的绝对值最大的两个实数（排除特殊值+∞和-∞）是$\pm(2-2^{-52})\times2^{1023}) \approx \pm1.80\times10^{308}$；规格化数能够精确表示的绝对值最小的两个实数是$\pm2^{-1022} \approx 1.11\times10^{-307}$。当一个实数不能用浮点数精确表示时，就只能用最接近的浮点数代替，而这就带来了误差。在计算机科学中这被称为“表示性错误（representation errors）”，例如前面给出的例子的内在原理是0.1无法用浮点数精确表示，用来近似表示0.1的浮点数实际上是：

$$3602879701896397 \times 2^{-55}$$

它等于0.1000000000000000055511151231257827021181583404541015625。

由于浮点数是不精确的，所以float类型的__repr__魔术属性被重写，以避免显示过多的数字。Python 2在显示浮点数时只是简单的保留17位有效数字，因此会将0.1显示为“0.10000000000000001”。Python 3则更加智能，显示一个浮点数时更符合人们的日常习惯，例如会将0.1直接显示为“0.1”。

我们可以通过表9-3列出的sys属性了解到Python解释器实现浮点数的细节。

表9-3. 浮点数实现相关sys属性

属性	说明
sys.float_info	一个具名元组，提供浮点数在Python解释器内部的表示形式相关信息。
sys.float_repr_style	一个字符串，反映float类型的__repr__的行为。

sys.float_info引用的具名元组包含如下元素：

- max：规格化数能够精确表示的最大正实数。
- max_exp：规格化数以2为底的指数最大值加1。
- max_10_exp：规格化数以10为底的指数最大值。
- min：规格化数能够精确表示的最小正实数。
- min_exp：规格化数以2为底的指数最小值加1。
- min_10_exp：规格化数以10为底的指数最小值。
- dig：尾数的字面值的有效位数。
- mant_dig：尾数的位数。
- epsilon：大于1.0的最小规格化数与1.0之间的差值。
- radix：指数相对的底数。
- rounds：舍入方式。

sys.float_repr_style则可以引用如下字符串：

- “short”：以Python 3的方式显示浮点数。
- “legacy”：以Python 2的方式显示浮点数。

请通过下面的语句来验证上述两个sys属性：

```
      >>> sys.float_info
      sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308,
min=2.2250738585072014e-308, min_exp=-1021, min_10_exp=-307, dig=15, mant_dig=53,
epsilon=2.220446049250313e-16, radix=2, rounds=1)
      >>> sys.float_repr_style
      'short'
      >>>
```

表示性错误给涉及浮点数的程序设计带来了很多麻烦，例如进行浮点数比较时不应使用==，而应判断两个浮点数之差是否小于一个非常小的实数（如sys.float_info.epsilon）。请看下面的例子：

```
>>> 0.1 * 3 - 0.3 < sys.float_info.epsilon
True
>>>
```

此外，由于采用2作为底数，所以浮点数的舍入规则与数学中的不一致。值得一提的是，标准库中的decimal模块提供了对以10为底的实数运算的支持，可以保证指定精度的运算结果精确，相应类型又被称为“定点数（fixed-point numbers）”。标准库中的fractions模块则提供了对有理数范围内运算的支持，它也是精确的。关于这两个模块的详细讨论超出了本书的范围。

---

complex()同样有两种语法，第一种语法为：

```
class complex(real=0, imag=0)
```

其功能相当于先将通过real参数和image参数指定的对象强制类型转换为复数，然后根据如下公式计算出最终得到的复数：

$$real + imag \times 1j$$

而将对象强制类型转换为复数的具体规则为：

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

当然，如果被传入对象本身就是复数，则没有必要进行强制类型转换，直接根据上述公式计算即可。特别的，如果调用complex()时没有给出参数，则得到复数0j。

complex()的第二种语法将复数的字符串表示法转换为复数，为：

```
class complex(string)
```

其string参数只能是字符串。复数的字符串表示法必须满足“±a±bj”或“±a±bJ”的形式，其中a是一个整数字面值或浮点数字面值，而bj或bJ则构成了一个“虚数字面值

（imaginary literals）”。特别的，a为0.0或0时，可以被省略；b为0.0或0时，可以省略整个虚数字面值部分；当它们同时为0.0或0时，记为“0j”或“0J”。虚数字面值的格式很简单：“j”或“J”为虚数单位（等价于数学中的i），而b则是一个整数字面值或浮点数字面值。下面是一些虚数字面值的例子：

```
0.0j        1J          2.7e-8j           3.14_15_93J
```

没有任何内置类型实现了魔术属性__complex__，但bool、int和float都实现了魔术属性__float__，因此complex()的第一种语法默认仅能作用于它们和complex自身。请通过如下语句验证：

```
>>> complex(True)
(1+0j)
>>> complex(False)
0j
>>> complex(0)
0j
>>> complex(1)
(1+0j)
>>> complex(0.0)
0j
>>> complex(3.14)
(3.14+0j)
>>> complex(2-3j)
(2-3j)
>>> complex(-1+0j)
(-1+0j)
>>> complex(5j)
5j
>>> complex(True, False)
(1+0j)
>>> complex(False, True)
1j
>>> complex(1, 1)
(1+1j)
>>> complex(0, 1.0)
1j
>>> complex(3.14, 2.7)
(3.14+2.7j)
>>> complex(5j, 3-2j)
(2+8j)
>>> complex(16j, 0.5)
16.5j
>>>
```

下面则是complex()的第二种语法的一些例子：

```
>>> complex('0')
0j
>>> complex('0J')
0j
>>> complex('1-2.5J')
(1-2.5j)
>>> complex('-13+6j')
```

```
(-13+6j)
>>>
```

需要强调的是，“a+bj”不能被写成“bj+a”，也不能被写成“a + bj”。

9-2. 算术运算

（语言参考手册：6.1、6.5～6.7）
（标准库：内置函数、内置类型）

整数、浮点数和复数都支持算术运算，否则就没有意义。本节将大体上按照优先级从高到低的顺序（参考表2-2）介绍Python中的算术运算。在下面关于算术运算的语法中，以“expr”开头的令牌代表一个表达式，被称为“操作数（operands）”，对其求值得到的对象必须是整数、浮点数或复数，否则将导致抛出TypeError异常。而在涉及多个操作数的运算中，将按照如下规则进行“隐式类型转换（implicit type cast）”：

- 只要有一个操作数是复数，其他操作数都会被转换为复数。
- 否则，只要有一个操作数是浮点数，则其他操作数都会被转换为浮点数。
- 否则，所有操作数都保持为整数。

首先是通过正号“+”和负号“-”定义的一元运算，其语法为：

```
{+|-}expr
```

“+”会导致调用expr.__pos__()，“-”会导致调用expr.__neg__()。正负号几乎是所有算术运算中优先级最高的，除了幂运算这个特例。

幂运算是通过“**”定义二元运算，其语法为：

```
expr1 ** expr2
```

当expr1具有魔术属性__pow__时，会调用expr1.__pow__(expr2)；否则会调用expr2.__rpow__(expr1)。

需要特别强调的是，“**”运算符是从右往左运算的，而其运算的优先级高于正负号。所以“a**-b”会被理解为“a**(-b)”，但“-a**b”会被理解为“-(a**b)”。请通过如下语句验证：

```
>>> 3 ** -2
0.1111111111111111
>>> -3 ** 2
-9
>>>
```

如果对零（0、0.0或0j，下同）进行负数次幂运算，会抛出ZeroDivisionError异常，例如：

```
>>> 0 ** -2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: 0.0 cannot be raised to a negative power
>>> 0j ** -7j
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: 0.0 to a negative or complex power
>>>
```

内置函数pow()的功能是“**”的功能的超集，其语法为：

```
pow(base, exp[, mod])
```

当省略了mod参数时，pow()就等价于base**exp；但当给出了mod参数时，pow()在逻辑上等价于base**exp%mod，且执行效率速度更快。对于后一种情况，当base具有__pow__时会调用base.__pow__(exp, mod)；否则会调用exp.__rpow__(base, mod)。“%”代表取余运算，将在下面讨论。

通过“*”定义的乘法，通过“/”定义的除法，通过“//”定义的整除，以及通过“%”定义的取余，都是优先级仅次于正负号的二元运算。它们的语法分别为：

```
expr1 * expr2
expr1 / expr2
expr1 // expr2
expr1 % expr2
```

在进行乘法时，如果expr1具有魔术属性__mul__，则会调用expr1.__mul__(expr2)；否则会调用expr2.__rmul__(expr1)。

在进行除法时，如果expr1具有魔术属性__truediv__，则会调用expr1.__truediv__(expr2)；否则会调用expr2.__rtruediv__(expr1)。

整除仅适用于整数和浮点数，等价于先进行除法，再以向下取整的方式对商保留整数部分。如果expr1具有魔术属性__floordiv__，则会调用expr1.__floordiv__(expr2)；否则会调用expr2.__rfloordiv__(expr1)。请看下面的例子：

```
>>> -3 // 2
-2
>>> 23.7 // 3.3
7.0
>>>
```

取余同样仅适用于整数和浮点数，用于获得整除中的余数。x % y在逻辑上等价于x - (x // y) * y，但涉及浮点数时两种算法的精度可能不同。如果expr1具有魔术属性__mod__，则会调用expr1.__mod__(expr2)；否则会调用expr2.__rmod__(expr1)。请看下面的例子：

```
>>> -3 % 2
1
>>> 23.7 % 3.3
0.6000000000000005
>>>
```

内置函数divmod()的功能是同时进行整除和取余，其语法为：

divmod(*x*, *y*)

该函数返回一个二元组，等价于(x//y, x%y)，但执行速度更快。如果x具有魔术属性__divmod__，则会调用x.__divmod__(y)；否则会调用y.__rdivmod__(x)。请看下面的例子：

```
>>> divmod(-3, 2)
(-2, 1)
>>> divmod(23.7, 3.3)
(7.0, 0.6000000000000005)
>>>
```

对于除法、整除和取余来说，如果对expr2求值的结果是零，则会抛出ZeroDivisionError异常，例如：

```
>>> (1+2.3j) / 0j
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: complex division by zero
>>> 42 // 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>> -3.58 % 0.0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: float modulo
```

```
>>>
```

值得一提，通过“@”定义的矩阵乘法与乘法、除法、整除和取余具有相同的优先级。但矩阵不是内置类型，也没有被标准库实现。“@”必须与第三方模块（主要是numpy）联合使用，故本书不对其进行讨论。

除了作为正负号外，还可以通过“+”和“-”定义二元运算，分别代表加法和减法，其语法为：

```
expr1 + expr2
expr1 - expr2
```

对于加法来说，当expr1具有魔术属性__add__时会调用expr1.__add__(expr2)；否则会调用expr2.__radd__(expr1)。对于减法来说，当expr1具有魔术属性__sub__时会调用expr1.__sub__(expr2)；否则会调用expr2.__rsub__(expr1)。加减法的优先级位于乘除法之后，与数学上保持一致。

内置函数abs()的功能是返回整数或浮点数的绝对值，以及复数的模，其语法为：

```
abs(x)
```

该函数会调用x.__abs__()，因此要求x实现魔术属性__abs__。下面是一些例子：

```
>>> abs(-2)
2
>>> abs(3.78)
3.78
>>> abs(3+4j)
5.0
>>> abs(3-4j)
5.0
>>>
```

int和float都实现了如下魔术属性以支持各种方式的取整：

- __trunc__：截断，只保留整数部分。
- __floor__：向下取整，即返回比当前数字小的最大整数。
- __ceil__：向上取整，即返回比当前数字大的最小整数。
- __round__：四舍五入，可以指定舍入到小数点后哪一位。

其中前三个分别被math.trunc()、math.floor()和math.ceil()在内部调用，但本书不详细讨论math模块。

内置函数round()的语法为：

```
round(x[, ndigits=None])
```

这会导致调用x.__round__(x, ndigits)，如果给出了ndigits参数（必须是一个非负整数），则舍入到小数点后第ndigits位，否则舍入到整数（即小数点后第0位）。注意四舍五入需要遵守“奇进偶不进”原则，请看下面例子：

```
>>> round(0.5)
0
>>> round(1.5)
2
>>> round(2.5)
2
>>> round(3.5)
4
>>> round(4.5)
4
>>> round(5.5)
6
>>> round(6.5)
6
>>> round(7.5)
8
>>> round(8.5)
8
>>> round(9.5)
10
>>>
```

最后，PEP 8建议，书写多种算术运算符混合在一起构成的表达式时，仅在优先级最低的运算符两侧添加空格，当优先级多于2时则使用括号，而复数总是被用括号括起来，例如：

```
1*2 + 3/4 - 5%6
3.2**2 / 5.7
(7.4**-2 * 9.8) - 16
(1+2j)**(3-4j)
```

而书写很长的算术表达式时，应利用括号将其分行书写，且后续行应以运算符开头，例如：

```
(1 + 2 + 3
        + 4 + 5 + 6
        + 7 + 8 + 9)
```

9-3. 位运算

（语言参考手册：6.6、6.8、6.9）
（标准库：内置类型）

只有整数支持位运算。当Python中的整数进行位运算时，我们应忽略整数在内存空间中的真实表示，而将每个整数都想象成一个无限长度的“补码（complement code）”。解读一个有符号整数的长度为n的补码时，先检查b_n是0还是1：当b_n是0时，意味着该补码对应0或一个正整数，其值通过如下公式计算：

$$\sum_{i=1}^{n} b_i \times 2^{i-1}$$

当b_n是1时，意味着该补码对应一个负整数，其值通过如下公式计算：

$$1 + \sum_{i=1}^{n} \sim b_i \times 2^{i-1}$$

其中b_i代表第i位的值（0或者1），而~代表按位取反运算，会在下面讨论。解读一个无符号整数（必然非负）的长度为n的补码时，先在补码的最高位前添加一个0，然后按照上述规则解读这个有符号整数的长度为n+1的补码。

只有一个一元位运算，即通过“~”定义的按位取反，语法为：

```
~expr
```

这会调用expr.__invert__()，作用就是将整数的补码的所有位都颠倒，即将0替换成1，1替换成0。根据上面介绍的补码解读方式可以推知，按位取反在数学上的意义是：

```
~expr == -expr-1
```

按位取反的优先级与正负号相同。请看下面的例子：

```
>>> ~2
-3
>>> ~-3
2
>>> ~0
-1
>>> ~-1
```

```
0
>>>
```

通过“<<”定义的左移和通过“>>”定义的右移是优先级仅次于加减法的二元位运算，其语法为：

```
expr1 << expr2
expr1 >> expr2
```

其中expr1和expr2都必须是非负整数。

如果expr1具有魔术属性__lshift__，则左移通过调用expr1.__lshift__(expr2)实现，否则会调用expr2.__rlshift__(expr1)。左移的本质是将expr1的补码向左移动expr2位，并在右侧添加expr2个0。由于expr1被限制为非负整数，所以左移在数学上的意义是：

```
expr1 << expr2 == expr1 * 2**expr2
```

请看下面的例子：

```
>>> 2 << 1
4
>>> 3 << 2
12
>>>
```

如果expr1具有魔术属性__rshift__，则右移通过调用expr1.__rshift__(expr2)实现，否则会调用expr2.__rrshift__(expr1)。右移的本质是将expr1的编码向右移动expr2位，并丢弃最右侧的expr2位数据。由于expr1被限制为非负整数，所以右移在数学上的意义是：

```
expr1 >> expr2 == expr1 // 2**expr2
```

请看下面的例子：

```
>>> 2 >> 1
1
>>> 3 >> 2
0
>>>
```

分别通过“&”、“^”和“|”定义的二元位运算按位与、按位异或和按位或，优先级依次降低，且按位与的优先级低于左移和右移。它们的语法为：

```
expr1 & expr2
expr1 ^ expr2
expr1 | expr2
```

注意这三种位运算都是将两个操作数的每组对应位分别进行逻辑中的与、异或和或运算（0代表False，1代表True），得到的结果重组为一个整数的补码，因此需要两个操作数的补码位数相同。当expr1和expr2的补码长度不相同时，通过如下规则将较短的补码填补到与较长的补码等长：

- 对于非负整数的补码，在最高位补0。
- 对于负整数的补码，在最高位补1。

如果expr1具有魔术属性__and__，则按位与通过调用expr1.__and__(expr2)实现，否则会调用expr2.__rand__(expr1)。按位与将expr1与expr2的每组对应位分别进行逻辑与运算。注意如下特殊规律：

- 0与任何整数进行按位与结果都是0。
- -1与任何整数进行按位与结果都是该整数本身。

请看下面的例子：

```
>>> -18 & 7
6
>>> 0 & -18
0
>>> 7 & 0
0
>>> -18 & -1
-18
>>> -1 & 7
7
>>> 0 & -1
0
>>>
```

如果expr1具有魔术属性__xor__，则按位异或通过调用expr1.__xor__(expr2)实现，否则会调用expr2.__rxor__(expr1)。按位异或是将expr1与expr2的每组对应位分别进行逻辑异或运算。注意如下特殊规律：

- 0与任何整数进行按位异或结果都是该整数本身。
- -1与任何整数进行按位异或结果都是该整数的相反数减去1。
- 任何整数与其自身进行按位异或结果都是0。
- 任何整数与其自身的按位取反进行按位异或结果都是-1。

请看下面的例子：

```
>>> 183 ^ -96
-233
>>> -96 ^ 0
-96
>>> 0 ^ 183
183
>>> -1 ^ -96
95
>>> 183 ^ -1
-184
>>> 183 ^ 183
0
>>> 183 ^ ~183
-1
>>> -96 ^ -96
0
>>> -96 ^ ~-96
-1
>>>
```

如果expr1具有魔术属性__or__，则按位或通过调用expr1.__or__(expr2)实现，否则会调用expr2.__ror__(expr1)。按位或是将expr1与expr2的每组对应位分别进行逻辑或运算。注意如下特殊规律：

- 0与任何整数进行按位或结果都是该整数本身。
- -1与任何整数进行按位或结果都是-1。

请看下面的例子：

```
>>> -18 | 7
-17
>>> 0 | -18
-18
>>> 7 | 0
7
>>> -18 | -1
-1
>>> -1 | 7
-1
>>> 0 | -1
-1
>>>
```

9-4. 数字的比较

（语言参考手册：6.10.1）
（标准库：内置类型）

int、float和complex都重写了魔术属性__lt__、__le__、__eq__、__ne__、__gt__和__ge__，使得整数、浮点数和复数相互间可以比较，且完全符合数学中的比较规则。当然，如果涉及复数，则只能进行==和!=比较，不支持<、>、<=和>=比较。

下面的例子说明整数、浮点数和复数之间可以相互比较：

```
>>> 0 == 0.0
True
>>> 0 == 0j
True
>>> 1 < 3.5
True
>>> 12.6 == 12.6+0j
True
>>>
```

9-5. 增强赋值语句

（语言参考手册：7.2.1）

所谓的“增强赋值语句（augmented assignment statements）”，指的是能够同时实现某种二元算术运算或位运算，以及赋值的语句。增强赋值语句依靠所谓的“增强操作符（augmented operators）”实现，这类操作符不能形成表达式，只能形成增强赋值语句。换句话说，增强赋值语句虽然具有类似于表达式的形式，但只能被作为一条单独的语句使用。表9-4列出了所有增强操作符。

表9-4. 增强操作符

增强操作符	语法	逻辑上等价于	优先尝试调用
`+=`	***obj += expr***	*obj = obj + expr*	*obj.__iadd__(expr)*
`-=`	***obj -= expr***	*obj = obj - expr*	*obj.__isub__(expr)*
`*=`	***obj *= expr***	*obj = obj * expr*	*obj.__imul__(expr)*
`@=`	***obj @= expr***	*obj = obj @ expr*	*obj.__imatmul__(expr)*
`/=`	***obj /= expr***	*obj = obj / expr*	*obj.__itruediv__(expr)*
`//=`	***obj //= expr***	*obj = obj // expr*	*obj.__ifloordiv__(expr)*
`%=`	***obj %= expr***	*obj = obj % expr*	*obj.__imod__(expr)*
`**=`	***obj **= expr***	*obj = obj ** expr*	*obj.__ipow__(expr)*
`<<=`	***obj <<= expr***	*obj = obj << expr*	*obj.__ilshift__(expr)*

增强操作符	语法	逻辑上等价于	优先尝试调用
>>=	***obj >>= expr***	*obj = obj >> expr*	*obj.__irshift__(expr)*
&=	***obj &= expr***	*obj = obj & expr*	*obj.__iand__(expr)*
^=	***obj ^= expr***	*obj = obj ^ expr*	*obj.__ixor__(expr)*
|=	***obj |= expr***	*obj = obj | expr*	*obj.__ior__(expr)*

表9-4给出了每条增强赋值语句在逻辑上等价的赋值语句，但前者比后者的执行速度快，因为对obj只求值一次，进行原地赋值，而不是创建一个新对象再赋值。这意味着在执行增强赋值语句之前，obj必须已经引用了某个对象。

而表9-4中的“优先尝试调用”列的含义是：当obj具有该列中给出的魔术属性的情况下，会调用该魔术属性；否则，obj会调用相应二元算术运算或二元位运算对应的魔术属性。举个例子，如果对象a具有__iadd__魔术属性，则a += b将导致执行a.__iadd__(b)，然后再用返回值对a赋值；否则，a += b将导致执行a.__add__(b)或b.__radd__(a)，然后再用返回值对a赋值。没有任何内置类型实现了表9-4中列出的魔术属性，因此这些魔术属性是由自定义数字类型（在第15章讨论）使用，以使得增强赋值语句的行为可以与相应二元算术运算或二元位运算存在差别。

下面是一些增强赋值语句的例子：

```
$ python3

>>> a = 12
>>> a += 3
>>> a
15
>>> a *= 2.4
>>> a
36.0
>>> a /= -3.2j
>>> a
(-0+11.25j)
>>> b = 1
>>> b <<= 3
>>> b
8
>>> b &= 2
>>> b
0
>>> class C:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...
>>> obj = C(1, -1)
>>> obj.x %= 5
>>> obj.x
1
>>> obj.y *= 4
>>> obj.y
-4
>>>
```

9-6. 数字的附加属性

（标准库：内置类型）

最后，数字类型还各自实现了一些附加属性，下面分别讨论。

int、float和complex都提供了实例属性real和imag，分别对应数字的实部和虚部。复数a+bj的实部是a，虚部是b。对于整数和浮点数来说，实部就是其自身，而虚部总是0。它们也都提供了类属性conjugate()以获得其共轭复数。a+bj的共轭复数是a-bj，因此整数和浮点数的共轭复数就是其自身。下面是一些列子：

```
>>> (5).real
5
>>> (5).imag
0
>>> (5).conjugate()
5
>>> (-0.3).real
-0.3
>>> (-0.3).imag
0.0
>>> (-0.3).conjugate()
-0.3
>>> (-4.5+8j).real
-4.5
>>> (-4.5+8j).imag
8.0
>>> (-4.5+8j).conjugate()
```

请注意，符合IEEE 754标准的浮点数能精确表示的数字都是有理数，而有理数总是能表示成一个分数的形式。因此float提供了类属性as_integer_ratio()，以将一个浮点数转换成一个二元组，其中第一个元素为分子，第二个元素为分母。由于整数也可被视为分母为1分子为其自身的分数，所以int也提供了as_integer_ratio()。请看下面这些例子：

```
>>> (5).as_integer_ratio()
(5, 1)
>>> (0).as_integer_ratio()
(0, 1)
>>> (-0.3).as_integer_ratio()
(-5404319552844595, 18014398509481984)
>>> (0.1).as_integer_ratio()
(3602879701896397, 36028797018963968)
>>>
```

注意上面例子中对应-0.3的分数不是-3/10，对应0.1的分数也不是1/10，这是因为分数总是精确的，而-0.3和0.1却是近似值。

int还提供了如下属性以在整数和其补码之间进行转换：to_bytes()将一个整数转换为指定长度的补码；from_bytes()将补码转换为一个整数。需要注意的是，to_bytes()是方法，而from_bytes()是类方法。

to_bytes()的语法为：

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

其中length参数用于指定补码的字节数，需传入一个正整数；byteorder参数用于指定补码的字节序，可传入“big”（大端法）和“little”（小端法）；signed参数用于说明该补码是否对应有符号整数，默认为否。正常情况该函数会返回一个二进制串，但如果指定长度的补码无法表示该整数则会抛出OverflowError异常。请看下面的例子：

```
>>> (1).to_bytes(1, 'little')
b'\x01'
>>> (1000).to_bytes(1, 'little')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: int too big to convert
>>> (1).to_bytes(2, 'little')
b'\x01\x00'
>>> (1).to_bytes(2, 'big')
b'\x00\x01'
>>> (-2).to_bytes(1, 'little', signed=True)
b'\xfe'
>>> (-1000).to_bytes(1, 'little', signed=True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
OverflowError: int too big to convert
>>> (-2).to_bytes(2, 'little', signed=True)
b'\xfe\xff'
>>> (-2).to_bytes(2, 'big', signed=True)
b'\xff\xfe'
>>>
```

from_bytes()的语法为：

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

其中bytes参数用于传入代表补码的二进制序列，byteorder参数和signed参数的含义与在to_bytes()中相同。请看下面的例子：

```
>>> int.from_bytes(b'\x01', 'little')
1
>>> int.from_bytes(b'\x01\x00', 'little')
1
>>> int.from_bytes(b'\x00\x01', 'little')
256
>>> int.from_bytes(b'\x00\x01', 'big')
1
>>> int.from_bytes(b'\x01\x00', 'big')
256
```

```
>>> int.from_bytes(b'\xfe', 'little', signed=True)
-2
>>> int.from_bytes(b'\xfe\xff', 'little', signed=True)
-2
>>> int.from_bytes(b'\xff\xfe', 'little', signed=True)
-257
>>> int.from_bytes(b'\xff\xfe', 'big', signed=True)
-2
>>> int.from_bytes(b'\xfe\xff', 'big', signed=True)
-257
>>>
```

任何一个整数都有与其等价的二进制字符串表示法，例如25对应“0b11001”，-25对应“-0b11001”。这与整数的补码表示不同，第一位总是1，符号位也被单独列出。int提供的类属性bit_length()和bit_count()从这种二进制字符串表示法中提取有用信息：它们都忽略了符号位，前者提取出这种表示法的位数，后者提取出这种表示法中有多少个1。请看下面的例子：

```
>>> (25).bit_length()
5
>>> (25).bit_count()
3
>>> (-25).bit_length()
5
>>> (-25).bit_count()
3
>>>
```

float提供了类属性is_integer()，以判断一个浮点数是否可以无精度损失地被转换成一个整数。注意该函数的判断标准是考察该浮点数的小数部分是否为0。请看下面的例子：

```
>>> (0.1*9 + 0.1).is_integer()
True
>>> (0.1*9 + 0.1)
1.0
>>> (0.1*3*3 + 0.1).is_integer()
False
>>> (0.1*3*3 + 0.1)
1.0000000000000002
>>>
```

一个整数不论采用哪种进制都可以精确表示，但浮点数不同，当采用的进制不是2的几次方时，将无法精确表示。浮点数被显示时默认采用十进制，而这种表示法导致表示性错误。浮点数的十六进制字符串表示的格式为：

```
'[sign]0x1.fractionpexponent'
```

其中各部分的含义是：

- sign：正负号标志，取“+”或“-”，“+”可以被省略。
- 0x：十六进制标志，表明后续的1.fraction构成了一个十六进制浮点数。
- fraction：一个13位十六进制非负整数，代表小数部分。
- p：指数标志，代表2。
- exponent：一个有符号整数，代表指数部分。

假设fraction的13个数字从高到低用h_1～h_{13}表示，则整个字符串被解读为：

$$sign \times \left(1 + \sum_{i=1}^{13} h_i \times 16^{-i}\right) \times 2^{exponent}$$

浮点数的这种表示法总是精确的。

float提供了属性hex()和fromhex()以支持在浮点数和它的十六进制字符串表示之间进行转换。与int的to_bytes()和from_bytes()类似，hex()是方法而fromhex()是类方法。

hex()的语法为：

```
float.hex()
```

请看下面的例子：

```
>>> (0.1*3*3 + 0.1).hex()
'0x1.0000000000001p+0'
>>> (0.1*9 + 0.1).hex()
'0x1.0000000000000p+0'
>>> (9.2).hex()
'0x1.2666666666666p+3'
>>> (-9.2).hex()
'-0x1.2666666666666p+3'
```

而fromhex()的语法为：

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

请看下面的例子：

```
>>> float.fromhex('0x1.0000000000001p+0')
1.0000000000000002
>>> float.fromhex('0x1.0000000000000p+0')
1.0
>>> float.fromhex('0x1.2666666666666p+3')
9.2
>>> float.fromhex('-0x1.2666666666666p+3')
-9.2
>>>
```