第 3 章　数值类型

数值类型用于存储数值，它是 Python 中最基础的一种变量类型，也是 Python 程序实现计算功能的基础。Python 中可支持的常用数值类型有整型（int）、浮点型（float）、复数类型（complex）和布尔类型（bool）。

3.1 整型

3.1.1 早期的整型

类似-2、-1、0、1、2 这样的数被称为整型数据，简称整型。

早期 Python 中普通整型是有限制的，它的取值范围与 C 语言中长整型的取值范围相同，即-2^{31}~2^{31}-1（32 位机中），换言之，Python 中的普通整型最长只能为 32 位，若值超出此范围便会报错。

后来 Python 放宽了对整型的限制，在 2.3 版的 Python 中，若在终端输入一个较长的整数，解释器不会报错，而是在整数后添加一个“L”，表示长整型。如下所示：

```
>>> 10000000000000000000
10000000000000000000L
```

用户也可显式地定义长整型，只需在普通整数之后添加字符“L”或“l”即可（推荐使用“L”，可避免与数字 1 混淆）。示例如下：

```
>>> a = 1000L
```

type()函数可查看对象的类型。使用 type()函数查看以上定义的对象 a 的类型，可以看到其长整型为“long”，如下所示：

```
>>> type(a)
<type 'long'>
```

3.1.2 Python3 中的整型

再后来，Python 有意地消除对整型数据的长度限制。Python 3 版本中取消了对长整型的定义，整数类型不再受到内部规定位数的影响，其取值范围只与计算机的内存有关，换言之，只要计算机的内存足够大，Python 的整型数据就能足够长。因此用户几乎无需再考虑溢出问题。

3.1.3 整型的表示方法

在 Python 中可以使用四种不同的进制表示整型，分别为：二进制（以“0B”或“0b”开头）、八进制（以“0o”或“0O”开头）、十进制和十六进制（以“0x”或“0X”开头）。示例如下：

```
>>> a = 0b01111              #二进制
```

```
>>> type(a)
<class 'int'>
>>> b = 0o71                    #八进制
>>> type(b)
<class 'int'>
>>> c = 0x3A                    #十六进制
>>> type(c)
<class 'int'>
```

3.1.4 进制转换

Python 内置了一系列进制转换函数，包括 bin()、oct()、hex()和 int()，下面将对这些函数的功能逐一进行讲解。

1、bin()

bin()函数用于将十进制整数转换为一个符合二进制整型规范的字符串，其中参数为十进制整数，其用法如下所示：

```
>>> bin(23)                     #将十进制转换为二进制
'0b10111'                       #获得二进制 23 的字符串
```

2、oct()

oct()函数用于将十进制整数转换为一个符合八进制整型规范的字符串，其中参数为十进制整数，其用法如下所示：

```
>>> oct(23)                     #将十进制转换为八进制
'0o27'                          #获得八进制 23 的字符串
```

3、hex()

hex()函数用于将十进制整数转换为一个符合十六进制规范的字符串，其中参数为十进制整数，其用法如下所示：

```
>>> hex(23)                     #将十进制转换为十六进制
'0x17'                          #获得十六进制 23 的字符串
```

4、int()

int()函数可接收一个符合整型规范的字符串，并将该字符串转换为整型。实际上，除符合十进制整型的字符串外，int()函数还可接收符合二进制、八进制和十六进制整型规范的字符串，只是在转换非十进制规范的字符串时，int()函数还需传入该字符串中包含的整型的进制，其格式如下：

```
int(str,radix)
```

下面通过示例来展示将符合不同进制整型规范的字符串转换为十进制的方法。

```
>>> int('0b0111001',2)          #二进制
57
>>> int('0o75',8)               #八进制
61
>>> int('0xAF',16)              #十六进制
175
```

结合以上四个进制转换函数，可在 Python 中实现各种类型之间的互转。二进制、八进制、十进制和十六进制之间互相转换的方式如表 3-1 所示。

表3-1 进制转换表

	二进制	八进制	十进制	十六进制
二进制	—	bin(int(x,8))	bin(int(x,10))	bin(int(x,16))
八进制	oct(int(x,2))	—	oct(int(x,10))	oct(int(x,16))
十进制	int(x,2)	int(x,8)	—	int(x,16)
十六进制	hex(int(x,2))	hex(int(x,8))	hex(int(x,10))	—

3.2 位运算

程序中的所有数据在计算机内存中都以二进制形式存储，位运算即以二进制位为单位进行的操作。Python 中的位运算主要有 6 种，具体如下：

- << 按位左移
- >> 按位右移
- & 按位与
- | 按位或
- ^ 按位异或
- ~ 按位取反

3.2.1 整型存储方式

Python 中的位运算是针对整型数据进行运算，在讲解位运算之前，我们先来学习一下整数在计算机内存中的存储方式。

计算机只能识别高低电平（高为 1，低为 0），因此计算机内的数据都以二进制形式存储，但为了方便计算，计算机在内存中存储的不是整型数据本身，而是整数的补码。

整数的二进制表示形式有 3 种，即原码、反码和补码，这三种表示方法都分为符号位和数值两部分，正数的符号位统一为“0”，负数的符号位统一为“1”。

- 原码：整数原码的数值部分由其它进制的数值直接转换而来，此外最高位添加符号位 0 或 1。如 9 的原码为 01001，-9 的原码为 11001 等。
- 反码：正数的反码与其原码相同，负数的反码为将其原码的数值部分按位取反（即 0 变 1，1 变 0），符号位保持不变。如 9 的反码为 01001，-9 的反码为 10110 等。
- 补码：正数的补码仍与原码相同，负数的补码为保留符号位，使数值部分在反码的基础上加 1。如 9 的补码为 01001，-9 的补码为 10111 等。

3.2.2 按位取反

按位取反需要使用符号“~”来操作，示例代码如下：

```
~obj
```

以上语句表示将操作数 obj 按位取反。按位取反遵循的规则为：位为 1 则取其反 0，位为 0 则取其反 1（包含符号位）。

以十进制 9（二进制值为 01001）的按位取反操作为例，实现过程如图 3-2 所示。

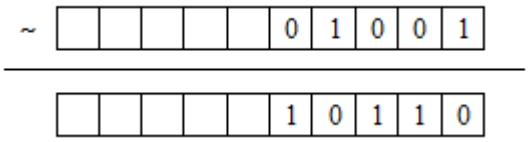

图3-1　按位取反操作示意图

由于计算机以补码为基础进行运算，因此需经过转码才能获取操作结果的原码，转码过程如图 3-2 所示。

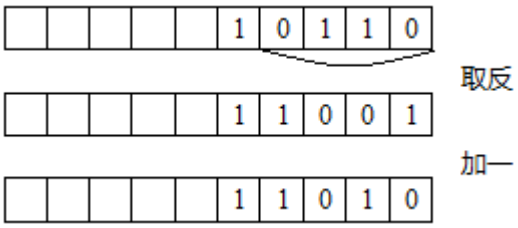

图3-2　转码操作示意图

由图 3-2 可知，执行“~9”操作后获得的二进制整数原码为 11010，即十进制的-10。在 Python 解释器中验证以上操作：

```
>>> bin(~9)
'-0b1010'
>>> ~9
-10
```

观察表达式执行结果，与图 3-2 中操作结果相同。

按位取反是掌握负数在计算机中的存储方式，即掌握负数原码转反码和补码操作的基础，读者应先掌握按位取反运算符的用法，理解计算机中数字存储的原理，再继续其它运算符的学习。

3.2.3 按位左移

按位左移需要使用符号“<<”来操作，示例代码如下：

```
obj << n
```

以上语句表示将操作数 obj 的二进制位向左平移 n 位，低位补零。在其它语言中，因整型位数有限，所以高位溢出时会将其丢弃，但 Python3 中整型数据的取值范围极大，因此 Python 中的位移不存在溢出问题。

以十进制的 9（二进制值为 01001）为例，将其左移 4 位，则其实现过程如图 3-4 所示。

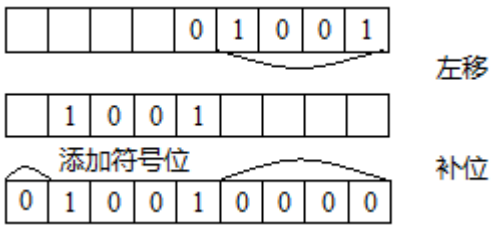

图3-3　向左位移操作示意图

由图 3-4 可知，执行“9<<4”操作后得到的二进制整数应为 010010000，即十进制的 144。下面在 Python 解释器中验证以上操作：

```
>>> bin(9<<4)
'0b10010000'
```

```
>>> 9<<4
144
```

观察表达式执行结果，与图 3-3 中操作结果相同。

值得一提的是，左移操作相当于“obj*(2**n)”，即操作数乘以 2 的 n 次方，用户可藉此原理自行实现按位左移功能。

3.2.4 按位右移

按位与需要使用符号“>>”来操作，示例代码如下：

```
obj >> n
```

以上语句表示将操作数 obj 的二进制位向右平移 n 位，移出的低位被舍弃，高位补一位符号位。

以十进制的 9（二进制值为 01001）为例，将其右移 2 位，则实现过程如图 3-5 所示。

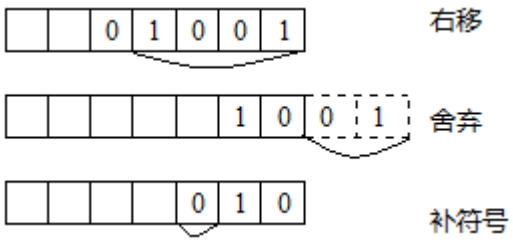

图3-4　向右位移操作示意图

由图 3-5 可知，执行“9>>2”操作后得到的二进制整数应为 010，即十进制的 2。在 Python 解释器中验证以上操作：

```
>>> bin(9>>2)
'0b11'
>>> 9>>2
-3
```

观察表达式执行结果，与图 3-5 中操作结果相同。

在计算机内部，由于负数以补码形式存储，因此实际上计算机在对负数进行位移操作后，需先经过转码才会将结果返回到终端。以十进制的-9（二进制原码为 11001，补码为 10111）为例，将其右移 2 位，则实现过程如图 3-6 所示。

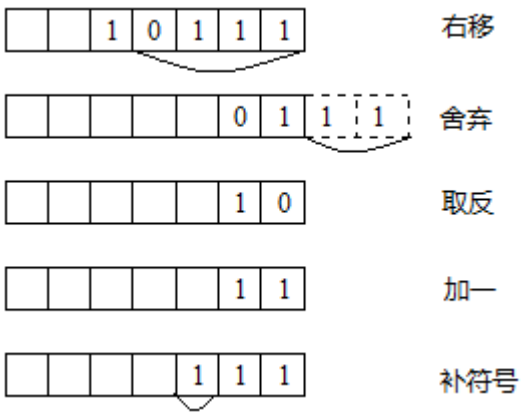

图3-5　负数右位移操作示意图

由图 3-6 可知，执行“-9>>2”操作后得到的二进制整数应为 111，即十进制的-3。在

Python 解释器中验证以上操作：

```
>>> bin(-9>>2)
'-0b11'
>>> -9>>2
-3
```

观察表达式执行结果，与图 3-6 中操作结果相同。

向左位移可通过获取操作数 obj 与 2^n 的乘积实现，理论上向右位移可通过获取操作数 obj 与 2^n 的商实现，但并非所有操作数都能被整除，因此在模拟实现右位移操作时，使用的表达式应为“obj//(2**n)”，而非“obj/(2**n)”。

3.2.5 按位与

按位与需要使用符号“&”来操作，示例代码如下：

```
obj1 & obj2
```

以上语句表示将操作数 obj1 与操作数 obj2 按位相与，并返回操作结果。位与操作遵循的规律为：若相与的两个二进制位都为 1，结果则为 1，否则为 0。

以十进制 9（二进制值为 01001）和 3（二进制值为 011）的位与操作为例，实现过程如图 3-6 所示。

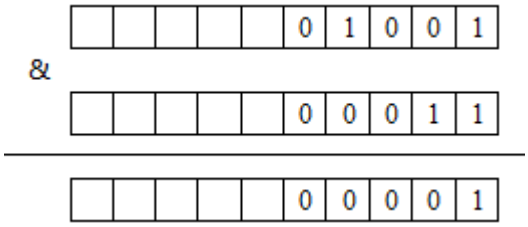

图3-6　位与操作示意图

由图 3-6 可知，执行“9&3”操作后得到的二进制整数为 00001，即十进制的 1。在 Python 解释器中验证以上操作：

```
>>> bin(9&3)
'0b1'
>>> 9&3
1
```

观察表达式执行结果，与图 3-7 中操作结果相同。

由于正数的符号位为 0，负数的符号位为 1，因此两个负数进行位与操作的结果必定为一个负数；由于负数的补码与原码不同，因此计算机内部在进行位与操作后，需经过转码才会返回结果的原码。以十进制数-9（补码为 10111）与-6（10010）的位与操作为例，其实现过程如图 3-8 所示。

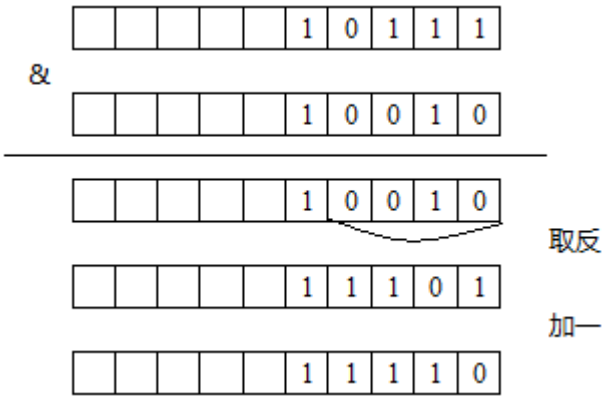

图3-7　负数位与操作示意图

由图 3-8 可知，执行“-9&-6”操作后得到的二进制整数为 11110，即十进制的-14。在 Python 解释器中验证以上操作：

```
>>> bin(-9&-6)
'-0b1110'
>>> -9&-6
-14
```

观察表达式执行结果，与图 3-8 中操作结果相同。

3.2.6 按位或

按位或需要使用符号“|”来操作，示例代码如下：

```
obj1 | obj2
```

以上语句表示将操作数 obj1 与操作数 obj2 按位相或，并返回操作结果。位或操作遵循的规律为：若相与的两个二进制位中有 1，结果为 1，否则为 0。

以十进制 9（二进制值为 01001）和 3（二进制值为 011）的位或操作为例，实现过程如图 3-8 所示。

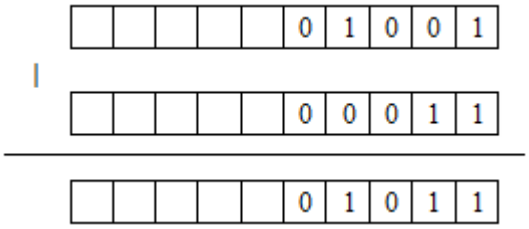

图3-8　按位或操作示意图

由图 3-9 可知，执行“9|3”操作后得到的二进制整数为 01011，即十进制的 11。在 Python 解释器中验证以上操作：

```
>>> bin(9|3)
'0b1011'
>>> 9|3
11
```

观察表达式执行结果，与图 3-8 中操作结果相同。

3.2.7 按位异或

按位异或需要使用符号“^”来操作，示例代码如下：

```
obj1 ^ obj2
```

以上语句表示将操作数 obj1 与操作数 obj2 按位异或，并返回操作结果。按位异或操作遵循的规律为：若进行异或的两个二进制位不同，结果为 1，否则为 0。

以十进制 9（二进制值为 01001）和 3（二进制值为 011）的按位异或操作为例，实现过程如图 3-9 所示。

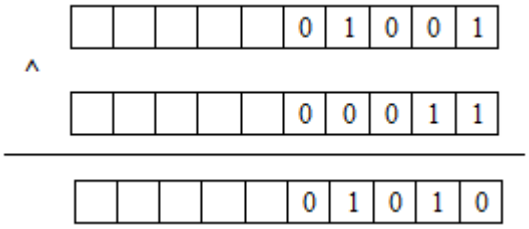

图3-9　按位异或操作示意图

由图 3-9 可知，执行“9^3”操作后得到的二进制整数为 01010，即十进制的 10。在 Python 解释器中验证以上操作：

```
>>> bin(9^3)
'0b1010'
>>> 9^3
10
```

观察表达式执行结果，与图 3-9 中操作结果相同。

3.3 浮点型

3.3.1 浮点型的表示方式

浮点型用于表示实数，如 3.14、5.28、6.0 等。在 Python 中，浮点数一般以十进制或科学计数法表示，示例如下：

```
>>> a = 314.15
>>> b = 3.14e2
```

以上定义的两个浮点型数据大小相同，其中 b 使用科学计数法定义，Python 中科学计数法定义浮点数的格式如下：

```
尾数 e 阶码
尾数 E 阶码
```

以上格式中 e 和 E 含义相同，都表示阶码的基——10；尾数和阶码可正可负，分别使用“+”或“-”表示，若为正数，则“+”可省略（如以上示例中定义的对象 b 便省略了“+”）。

3.3.2 浮点型的取值范围

Python 中的浮点型是双精度的，每个浮点型数据占 8 个字节（即 64 位），且遵守 IEEE 标准，其中 52 位用于存储尾数，11 位用于存储阶码，剩余 1 位则存储符号。Python 中浮点型的取值范围大约为-1.8e308~1.8e308，若超出这个范围，Python 则会将值视为无穷大或无穷小。示例如下：

```
>>>3.14e5
314000.0
>>> 1/3
0.3333333333333333
>>> 100/3
33.333333333333336
```

3.3.3 浮点型的存储

使用 Python2.7 之前版本的读者，在定义浮点数时可能会遇到如下问题：

```
>>> a = 3.13
>>> a
3.1299999999999999
```

以上代码中，定义了一个值为 3.13 的浮点型数字 a，但解释器打印的结果却只是一个与 3.13 接近的包含 17 位有效数字的浮点数。之所以出现这种情况，是因为 Python 中浮点数是用二进制小数（进制的知识将在后续小节讲解）存储的，而用户在定义浮点数时使用十进制小数表示，Python 中浮点数的位数有限，因此在十进制小数转换为二进制小数时会发生截断。

大多数十进制小数在转换为二进制小数时都会出现上述问题，如此一来，当多个不够精确的浮点数进行运算后，所得结果的偏差可能更加明显。但是，实际情况下，一般用户并不需要担心这一现象， 因为 Python 浮点数的精度可以满足大多数计算需求，且 Python2.7 之后的版本中也对上述问题进行了一定的处理。

3.3.4 高精度浮点数

Python 中浮点型的有效位只有 17 位，若有效位超过 17 位，Python2.7 之后的版本同样会对其进行截断，如在 3.5.2 版本中输入一个超长的浮点数。

```
>>> 3.1234597134717356198491
3.1234597134717355
```

我们发现实际存储的仍为只有 17 为有效数字的浮点数，此时若想要使用更高精度的浮点数，则需通过 decimal 模块进行定义。

decimal 模块实现的十进制运算可满足对精度要求较高的场合，其用法如下：

```
>>> import decimal
>>> a = decimal.Decimal('2.1231314143235443566')
>>> print(a)
2.1231314143235443566
```

该方法在 Python2.4 及更高版本中都可使用。

3.4 复数类型

3.4.1 复数类型定义

复数和虚数最早出现在数学领域，“虚数”由 17 世纪的著名数学家笛卡尔提出，用于解决负数的平方根问题，通常记为 i 或 j（Python 中一般记为 j），数学中认为 $i^2=-1$；复数在 18 世纪末逐渐为大多数人接受。

自 1.4 版本起，Python 中加入了复数类型（Complex Type），简称复数。形似 3+2j、3.1+4.9j、-2.3-1.9j 这样的数，在 Python 中都被称为复数。复数由“实部”和“虚部”两部分组成，其中实部是一个实数，虚部则是一个实数与 j（例如 2j、4.9j、-1.9j 等）的组合。

3.4.2 复数的特点

Python 中的复数类型有如下几个特点：

- 复数由实部和虚部构成，其一般形式为：real+imagj；
- 实部 real 和虚部的 imag 都是浮点型；
- 虚部必须有后缀 j 或 J。

3.4.3 创建复数

在 Python 中可以直接将一个复数赋给一个变量，如：

```
>>> a = 3+2j
>>> print(a)
(3+2j)
```

使用 type()函数查看变量 a 的类型，将会获取其类型“complex”，如下所示：

```
>>> type(a)
<class 'complex'>
```

也可以使用内建函数 complex(real,imag)，通过传入实部和虚部的方式定义复数，如：

```
>>> a = complex(3,2)
>>> print(a)
(3+2j)
```

若 imag 缺省，则 imag 使用默认值 0。示例如下：

```
>>> complex(3)
(3+0j)
```

3.4.4 获取复数的实部和虚部

复数的实部和虚部可以通过符号“.”获取，如获取以上定义的复数 a 的实部和虚部，可以使用如下方式：

```
>>> a.real
3.0
```

```
>>> a.imag
2.0
```

3.5 布尔类型是一种特殊整型

Python 中的布尔类型（Bool Type）只有两个取值：True（真）和 False（假）。实际上布尔类型也是整型，其值 True 对应整数 1，False 对应整数 0。布尔类型一般用于布尔测试，一般情况下布尔测试的结果只有 True 和 False（但也有例外）。

3.6 数字运算

相比其它编程语言，Python 中的运算符更为丰富，且功能更为强大，因此 Python 中的数据可以以相对简单的方式，实现丰富的运算功能。Python 中的运算符可分为如下几类：算术运算符、赋值运算符、比较运算符、逻辑运算符等。

3.6.1 算术运算符

Python 中的算术运算符有：+、-、*、/、//、%和**，这些运算符都是双目运算符，在终端输入由两个操作数和一个运算符组成的表达式，Python 解释器就会解析表达式，并打印计算结果。

以操作数 a = 3，b = 5 为例，Python 中各个运算符的功能及示例如表 3-2 所示。

表3-2 算术运算符

运算符	说明	示例
+	加：使两个操作数相加，获取操作数的和	a + b，结果为 8
-	减：使两个操作数相减，获取操作数的差	a –b ，结果为-2
*	乘：使两个操作数相乘，获取操作数的积	a * b，结果为 15
/	除：使两个操作数相除，获取操作数的商	a / b，结果为 0.6
//	整除：使两个操作数相除，获取商的整数部分	a // b，结果为 0
%	取余：使两个操作数相除，获取余数	a % b，结果为 3
**	幂：使两个操作数进行幂运算，获取 obj1 的 obj2 次幂	a ** b，结果为 243

在终端中直接输入示例中的表达式，结果将会直接在终端输出，但在此过程中，运算符两侧的操作数 a 和 b 并没有被修改，这是因为运算的结果并未被保存到变量 a 或 b 之中。以加法为例，示例如下：

```
>>> a = 3
>>> b = 5
>>> a + b
8
>>> print(a)
3
>>> print(b)
5
```

Python 中的算术运算符支持对相同或不同类型的数字进行各种运算（需注意：对复数

进行除+、-、*之外的运算时可能会产生警告），且无需进行转换，例如进行如下的混合运算：

```
>>> 3 + (3+2j)          #整型 + 复数
(6+2j)
>>> 3 * 4.5             #整型 * 浮点型
13.5
>>> 5.5 - (2+3j)        #浮点型 - 复数
(3.5-3j)
>>> True + (1+2j)       #布尔类型 + 复数
(2+2j)
```

解释器都会给出正确结果。这是由于 Python 在对不同类型的对象进行运算时，会强制将对象的值进行临时类型转换，这些转换遵循如下规律：

- 布尔类型在进行算术运算时，将其分别视为数值 0 和 1；
- 整型与浮点型运算时，将整型转化为浮点型；
- 其它类型与复数运算时，将其它类型转换为复数类型。

简单来说，混合运算中类型相对简单的操作数会被转换为与复杂类型操作数相同的类型。

多学一招：+和*的特殊用法

（1）+的特殊用法

其它语言中，“+”运算符操作的对象只能是数值，但 Python 中，“+”还能对字符串进行拼接操作，如对字符串“hello ”和字符串“world”进行“+”操作，示例如下：

```
>>> 'hello ' + 'world'
'hello world'
```

由表达式的执行结果可知，以上两个字符串被连接成了一个长字符串“hello world”。

（2）*的特殊用法

若要在 C 语言中打印一个由 30 个“-”组成的分割线，需要在 printf()函数中完整地传入 30 个“-”，示例如下：

```
printf("------------------------------\n")
```

但在 Python 中，使用“*”便能简单地实现以上功能，具体示例如下：

```
>>> print("-"*30)
------------------------------
```

由以上示例可知，在 Python 中，字符串与数值由乘号连接时，Python 解释器会将字符串重复打印多次（由数值决定）。

3.6.2 赋值运算符

赋值运算符的功能是：将一个表达式或对象赋给一个左值，其中左值必须是一个可修改的值，不能为一个常量。“=”是基本的赋值运算符，此外“=”可与算术运算符组合成复合赋值运算符。Python 中的复合赋值运算符有：+=、-=、*=、/=、//=、%=、**=，它们的功能相似，例如“a+=b”等价于“a=a+b”，“a-=b”等价于“a=a-b”，诸如此类。

赋值运算符也是双目运算符，以 a = 3，b = 5 为例，Python 中各个赋值运算符的功能及示例如表 3-3 所示。

表3-3 赋值运算符

运算符	说明	示例
=	等：将右值赋给左值	a = b，a 为 5
+=	加等：使右值与左值相加，将和赋给左值	a += b，a 为 8
-=	减等：使右值与左值相减，将差赋给左值	a –= b ，a 为-2
*=	乘等：使右值与左值相乘，将积赋给左值	a *= b，a 为 15
/=	除等：使右值与左值相除，将商赋给左值	a /= b，a 为 0.6
//=	整除等：使右值与左值相除，将商的整数部分赋给左值	a //= b，a 为 0
%=	取余等：使右值与左值相除，将余数赋给左值	a %= b，a 为 3
**=	幂等：获取左值的右值次方，将结果赋给左值	a **= b，a 为 243

经以上操作后，左值 a 发生了改变，但右侧操作数 b 并没有被修改。以“+=”为例，示例如下：

```
>>> a = 3
>>> b = 5
>>> a += b
>>> print(a)
8
>>> print(b)
5
```

需要说明的是，与 C 语言不同，Python 中在进行赋值运算时，即便两侧操作数的类型不同也不会报错，且左值可正确地获取右操作数的值（不会发生截断等现象），这与 Python 中变量定义与赋值的方式有关，这点将在后续小节（3.7 对象与引用）中着重讲解。

3.6.3 比较运算符

比较运算符同样是双目运算符，它与两个操作数构成一个表达式，这种表达式通常用于布尔测试，测试的结果只能是 True 或 False。比较运算符的操作数可以是表达式或对象，Python 中的比较运算符有：==、!=、>、<、>=、<=，以 a = 3,b = 5 为例，其功能与相关示例分别如表 3-4 所示。

表3-4 比较运算符

运算符	说明	示例
==	比较左值和右值，若相同则为真，否则为假	a = 3，b = 5 a == b 不成立，结果为 False
!=	比较左值和右值，若不相同则为真，否则为假	a = 3，b = 5 a != b 成立，结果为 True
>	比较左值和右值，若左值大于右值则为真，否则为假	a = 3，b = 5 a > b 不成立，结果为 False
<	比较左值和右值，若左值小于右值则为真，否则为假	a = 3，b = 5 a < b 成立，结果为 True
>=	比较左值和右值，若左值大于或等于右值则为真，否则为假	a = 3，b = 5，a >= b 不成立，结果为 False a = 3，b = 3，a >= b 成立，结果为 True
<=	比较左值和右值，若左值小于或等于右值则为真，否则为假	a = 3，b = 5，a <= b 成立，结果为 True a = 3，b = 3，a<= b 成立，结果为 True

比较运算符只对操作数进行比较，不会对操作数自身造成影响，即经过比较运算符运算

后的操作数不会被修改。

3.6.4 逻辑运算符

Python 中也支持逻辑运算，但 Python 中逻辑运算符的功能与其它语言有所不同。Python 中分别使用“or”、“and”、“not”这三个单词作为逻辑运算“或”、“与”、“非”的运算符，其中 or 与 and 为双目运算符，not 为单目运算符。

逻辑运算符的操作数可以为表达式或对象，下面将对它们的功能分别进行说明。

（1）or

若 or 运算符左操作数的布尔值为 True，则返回左操作数，否则返回右操作数或其计算结果（若为表达式），示例如下：

```
>>> 0 or 3+5              #左操作数布尔值为 False
8
>>> 3 or []               #左操作数布尔值为 True
3
```

（2）and

若左操作数的布尔值为 False，则返回左操作数或其计算结果（若为表达式），否则返回右操作数的执行结果，示例如下：

```
>>> 3-3 and 5
0
>>> 3-4 and 5
5
```

（3）not

若操作数的布尔值为 False 则返回 True，否则返回 False，示例如下：

```
>>> not(3-5)
False
>>> not(False)
True
```

3.7 运算符优先级

Python 中支持使用多个不同的运算符连接简单表达式，实现相对复杂的功能，为了避免含有多个运算符的表达式出现歧义，Python 为每种运算符都设定了优先级。Python 中各种运算符的优先级由低到高依次如表 3-5 所示。

表3-5 运算符优先级

运算符	说明
or	布尔“或”
and	布尔“与”
not	布尔“非”
in，not in	成员测试（字符串、列表、元组、字典中常用）
is，is not	身份测试
<，<=，>，>=，!=，==	比较
|	按位或

^	按位异或
&	按位与
<<，>>	按位左移、按位右移
+，-	加法，减法
*，/，%	乘法、除法，取余
+x，-x	正负号
~	按位取反
**	指数

默认情况下，运算符的优先级决定了复杂表达式中的哪个单一表达式先执行，但用户可使用圆括号“()”改变表达式的执行顺序。通常圆括号中的表达式先执行，如对于表达式“3+4*5”，若想让加法先执行，可写为“(3+4)*5”。此外，若有多层圆括号，则最内层圆括号中的表达式先执行。

运算符一般按照自左向右的顺序结合，如在表达式“3+5-4”中，+、-优先级相同，解释器会先执行“3+5”，再将 3+5 的执行结果 8 与操作数 4 一起，执行“8-4”，即执行顺序等同于“(3+5)-4”；但赋值运算符的结合性为自右向左，如表达式“a = b = c”，Python 解释器会先将 c 的值赋给 b，再将 b 的值赋给 a，即执行顺序等同于“a = (b = c)”。

3.8 类型转换

在程序中，最常对数字进行的操作便是运算。例如在终端输入如下语句：

```
>>>3 + 5
8
```

实际就是进行了一次算术运算。

当然，如上示例的意义并不大，因为这样的程序只能单一地执行两个固定数据的运算。程序中更常设定的功能，是提示用户进行输入，交互地获取对象的值。示例如下：

```
num_01 = input("请输入变量 a：")
num_02 = input("请输入变量 b：")
```

如上的两条语句可利用 input()函数获取两个用户输入，并将获取到的数据分别赋给 num_01 和 num_02。假设用户根据第一条提示输入的数据为“1”，根据第二条提示输入的数据为“5”，那么在终端执行求和语句：

```
>>> sum = num_01 + num_02
>>> print(sum)
```

将会获取如下结果：

```
15
```

这个结果由字符串“1”和字符串“5”拼接而来，若用户本意为计算两个数字类型的和，那么此结果显然与期望不符。这是因为，在 Python3 版本中，使用 input()函数获取到的数据是一个字符串，即便用户输入的是形如“3”、“3.4”、“3+2j”、“False”等的数值类型的数据，解释器也会将其视为字符串。

3.8.1 类型转换函数

综上，若用户想要将从终端获取的数据作为数值类型进行计算，就必须先进行类型转换。

Python 中内置了一系列可实现强制类型转换的方法，保证用户在有需求时，可使用 int()、float()、complex()和 str()函数，将目标内容转换为指定类型。这四个函数的功能分别如表 3-6 所示。

表3-6 类型转换函数

函数	说明
int()	将浮点型、布尔类型和符合数值类型规范的字符串转换为整型
float()	将整型和符合数值类型规范的字符串转换为浮点型
complex()	将其它数值类型或符合数值类型规范的字符串转换为复数类型
str()	将数值类型转换为字符串

3.8.2 类型转换注意事项

在使用以上函数时有两点需要注意：

（1）int()函数、float()函数和 complex()函数只能转换符合数值类型格式规范的字符串；

（2）使用 int()函数将浮点数转换为整数时，若有必要会发生截断（取整）而非四舍五入。

用户在使用以上函数时，必须考虑到以上两点，否则可能会因字符串不符合要求而导致在转换时产生错误，或因截断而产生预期之外的计算结果。

3.8.3 类型转换示例

int()、float()、complex()和 str()函数的简单用法分别如下：

```
>>> int(3.6)              #浮点型转整型
3                         #小数部分被截断
>>> float(3)              #整型转浮点型
3.0
>>> complex('3+2j')       #字符串转复数
(3+2j)
>>> demo_str = '345'          #字符串转整型
>>> type(demo_str)
<class 'str'>
>>> type(int(demo_str))
<class 'int'>
>>> str(2+3j)             #复数转字符串
'(2+3j)'
```

掌握以上函数后，想对从终端获取的数据，或两个符合数值类型格式的字符串进行算术运算便尤为简单。例如要求对两个符合数值类型格式的字符串进行求和运算，示例如下：

```
>>> str_01 = "2+4j"
>>> str_02 = "5+2j"
>>> sum = complex(str_01)+complex(str_02)
>>> print(sum)
(7+6j)
```

由执行结果“(7+6j)”可知，字符串 str1 和 str2 中存储的字符串成功被转换为复数类型，

并进行了求和计算。

值得一提的是，在经过以上操作后，str_01 和 str_02 仍为字符串，这是因为使用 complex() 转换的结果只是一个临时对象，并未被存储。如若通过 type()测试 str_01、str_02 和 sum 的类型，获得的结果如下：

```
>>> type(str_01)
<type 'str'>
>>> type(str_02)
<type 'str'>
>>> type(sum)
<type 'complex'>
```

3.9 对象和引用

3.9.1 对象

在 Python 中定义的数据一般被称为对象（Object）。计算机中的数据是分块存储的，我们可以把计算机的内存空间视为一个被等分为多个格子的储物柜，储物柜中的每个格子顺序编号，当有对象被定义时，Python 将对象的数值放入储物柜的某个格子中，并在格子上贴上标签，如此便完成了对象的定义。

上述过程中，标签可视为“对象名（name）”，存储到格子中的内容视为对象的“值（value）”，而对象所在格子的编号则可视为对象在内存中的地址，Python 中将对象的内存地址称为“身份（id）”。值和身份是 Python 中对象的重要特性，此外，对象还有一个重要的特性——类型（type），类型决定了对象在“储物柜”中占据“格子”的数量。

Python 对象的身份可唯一标识一个变量，任何对象的身份都可使用内建函数 id()获取，如在 Python 中定义一个整型对象 a，并获取其身份，则可使用如下方法：

```
>>> a = 3
>>> id(a)
8939712
```

3.9.2 引用

Python 对象的身份是只读的，用户不能直接更改对象的身份。Python 中部分对象的值也是不可改变的，值不能被改变的对象称为不可变对象，Python 的数值类型就是不可变对象。

读者或许会有疑问，在对一个已经定义好的数值类型对象重新赋值时明明可以操作成功，如在进行下列操作时并不会报错：

```
>>> a = 5
>>> a = 6
>>> print(a)
6
```

且对象 a 的值也确实从 5 更改为 6，为什么却说数值类型是不可改变的呢？这与 Python 中变量的赋值方式有关。实际上，当进行以上操作时，改变的并非对象 a 的值，而是对象 a 的引用。

Python 中的赋值通过引用实现。当用户在定义对象时，解释器将对象的值放入内存地址，并将该地址的引用赋给对象，经此过程，对象名便等同于内存地址的别名，用户可通过变量名获取对象的值。而在对对象进行修改时，解释器实际上会将新数值放入新内存地址，再将新内存地址的引用赋给待修改对象，如图 3-1 所示。

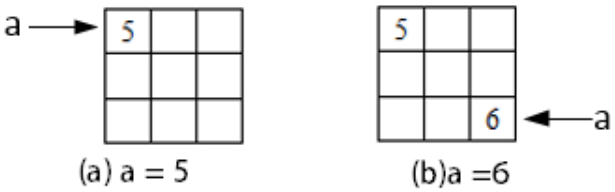

图3-10　对象定义与再赋值

由图 3-1 可知，该对象原地址中存储的数值并没有被修改。下面在 Python 解释器中进行验证：

```
>>> a = 5              #定义对象 a
>>> id(a)              #查看对象身份
8939776
>>> a = 6              #重新赋值
>>> id(a)              #查看对象身份
8939808
```

观察以上各表达式的执行结果，可知重新赋值后对象的身份发生了改变。

3.9.3 身份运算符

Python 中的身份运算符为：is 和 is not，用于判断两个对象是否相同。Python 中对象的唯一标识即身份，因此身份运算符的运算过程即对象身份的比较过程。身份运算符的功能如表 3-7 所示。

表3-7 身份运算符

函数	说明
is	测试两个对象是否相同，相同返回 True，否则返回 False
is not	测试两个对象是否不同，不同返回 True，否则返回 False

3.9.4 身份运算符的使用

下面通过示例来展示身份运算符的用法。

```
>>> a = 5              #定义对象 a
>>> b = a              #使用 a 定义对象 b，b 获取 a 的引用
>>> a is b             #判断 a 是否与 b 相同
True
>>> a is not b         #判断 a 是否与 b 不同
False
>>> c = 6              #定义对象 c
>>> b = c              #使用对象 c 为 b 赋值
>>> a is b
False                  #对象 b 的身份被修改，与对象 a 不再相同
```

```
>>> b is c
True                    #对象 b 的身份与对象 c 的身份相同
```