

运 算 符

在 Python 中,运算符(operator)是一种特殊的符号,表示应该执行某种计算。运算符作用的对象称为操作数(operand):

```
>>> a = 1
>>> b = 2
>>> a + b                                     #“+”叫作运算符,变量 a 和 b 叫作操作数
3
```

用运算符和括号将操作数连接起来的、符合 Python 语法规则的式子,称为表达式(expression),如 $a+b-5$ 。

3.1 算术运算符

Python 语言中一共有 9 个算术运算符,如表 3-1 所示。

表 3-1 算术运算符

运算符	类别	意义	示例	说 明
+	一元	正号	$+a$	正号
+	二元	求和	$a + b$	求 a 与 b 的和
-	一元	负号	$-a$	负号
-	二元	求差	$a - b$	求 a 和 b 的差
*	二元	乘法	$a * b$	求 a 和 b 的乘积
/	二元	除法	a / b	求 a 除以 b 的商,结果为浮点数
%	二元	求余数	$a \% b$	求 a 除以 b 的余数
//	二元	求整商	$a // b$	求 a 除以 b 的商(F 取整)
**	二元	幂运算	$a ** b$	求 a 的 b 次幂

```

>>> a = 3
>>> b = 2
>>> +a
3
>>> -a
-3
>>> a + b
5
>>> a - b
1
>>> a * b
6
>>> a / b
1.5
>>> a % b
1
>>> a ** b
9

```

注意：除法运算的结果总是一个浮点数：

```

>>> 6 / 3                                #结果不是 2 而是 2.0,与 c 语言不同
2.0
>>> 5 / 2                                #与 c 语言不同
2.5

```

注意：求整商“//”的计算结果：

```

>>> 5 // 2                                #2.5 下取整为 2
2
>>> 5 // -2                               #-2.5 下取整为-3
-3
>>> -5 // 2                               #-2.5 下取整为-3
-3
>>> -5 // -2                              #2.5 下取整为 2
2

```

$n \% m$ 的计算结果是 $[0, m-1]$ 。求余运算的一个应用场景是求一个多位数各个数位上的数字：

```

>>> num = 351
>>> num % 10                               #得到 num 个位上的数字
1
>>> num // 100                             #得到 num 百位上的数字
3
>>> num // 10 % 10                         #得到 num 十位上的数字
5

```

Python 语言的内置函数 `divmod()`, 用来同时计算整商和余数; 函数 `pow()` 用来执行幂运算:

```
>>> a, b = divmod(41, 7)
>>> a                                # a = 41 // 7
5
>>> b                                # b = 41 % 7
6
>>> pow(5, 2)                        # 相当于计算 5 ** 2, 求幂 (Power)
25
>>> pow(5, 2, 3)                    # 相当于计算 5 ** 2 % 3
1
```

3.2 比较运算符

Python 语言中一共有 6 个比较运算符, 如表 3-2 所示。

表 3-2 比较运算符

运算符	意 义	示例	说 明
<code>==</code>	等于	<code>a == b</code>	如果 a 等于 b, 则结果为 True; 否则为 False
<code>!=</code>	不等于	<code>a != b</code>	如果 a 不等于 b, 则结果为 True; 否则为 False
<code><</code>	小于	<code>a < b</code>	如果 a 小于 b, 则结果为 True; 否则为 False
<code><=</code>	小于或等于	<code>a <= b</code>	如果 a 小于或等于 b, 则结果为 True; 否则为 False
<code>></code>	大于	<code>a > b</code>	如果 a 大于 b, 则结果为 True; 否则为 False
<code>>=</code>	大于或等于	<code>a >= b</code>	如果 a 大于或等于 b, 则结果为 True; 否则为 False

```
>>> a = 1
>>> b = 2
>>> a == b
False
>>> a != b
True
>>> a <= b
True
>>> a >= b
False
```

有些浮点数在计算机内部不能精确地存储:

```
>>> x = 1.1 + 2.2
>>> x == 3.3                                # 不能精确存储导致两者不相等
```

```
False
```

因此,确定两个浮点数是否“相等”的首选方法,在给定的误差(Tolerance)的情况下,计算它们是否彼此接近:

```
>>> tolerance = 0.00001
>>> x = 1.1 + 2.2
>>> abs(x - 3.3) < tolerance      #求绝对值函数 abs()
True
```

此时可以认为 x 等于 3.3。

3.3 逻辑运算符

Python 语言中一共有 3 个逻辑运算符,如表 3-3 所示。

表 3-3 逻辑运算符

运算符	示例	说 明
not	not x	如果 x 为 False,则结果为 True;否则结果为 False
or	x or y	如果 x 和 y 都为 False,则结果为 False;否则结果为 True
and	x and y	如果 x 和 y 都为 True,则结果为 True;否则结果为 False

在逻辑表达式的求解过程中,并不是所有的表达式都会被执行,只有在必须执行下一个表达式才能得到最终的解时,才会执行该表达式,这种特性叫作短路求值(Short-Circuit Evaluation)。

```
>>>x = 1
>>> not x < 2      #x < 2 为 True,因此结果为 False
False
>>> x < 2 or 0 > 1  #x < 2 为 True,因此结果为 True
True              #短路求值,不执行第二个表达式 0 > 1
>>> x < 0 or 2 < 1  #x < 0 为 False,2 < 1 为 False,因此结果为 False
False
>>> x < 0 and 2 > 1  #x < 0 为 False,因此结果为 False
False            #短路求值,不执行第二个表达式 2 > 1
>>> x > 0 and 2 > 1  #x > 0 为 True,2 > 1 为 True,因此结果为 True
True
```

在布尔上下文求值时,下列所有数据都被看作 False:

- (1) 布尔值 False;
- (2) 任何数值为零的值,如 0、0.0、0.0+0.0j;
- (3) 空字符串;

(4) 空的组合数据类型,如空列表、空元组、空集合、空字典;

(5) None。

```
>>> bool(0), bool(0.0), bool(0.0+0j)    #0、0.0 和 0.0+0j 都为 False
(False, False, False)
>>> bool(-1), bool(1.1), bool(1+1j)      #-1、1.1 和 1+1j 都为 True
(True, True, True)
>>> bool(''), bool(None), bool(False)    #空字符串'',None 和 False 都为 False
(False, False, False)
>>> bool('fun'), bool(' ')               #字符串 fun 和空格“ ”都为 True
(True, True)
>>> bool(list()), bool(tuple())          #空列表 list() 和空元组 tuple() 都为 False
(False, False)
>>> bool(set()), bool(dict())            #空集合 set() 和空字典 dict() 都为 False
(False, False)
```

下列逻辑表达式中使用了非布尔操作数:

```
>>> x = 1
>>> y = 2
>>> x and y
2
>>> x or y                                #短路求值,因为 x 为 True
1
>>> bool(x)                              #x 的值为 1,等价于 True
True
>>> not x
False
>>> x = 0.0
>>> y = 1.1
>>> x and y                              #短路求值
0.0
>>> x or y
1.1
>>> bool(x)                              #x 的值为 0.0,等价于 False
False
>>> not x
True
```

在某些情况下,利用短路求值可以编写出既简洁又高效的代码。假如有两个变量 a 和 b,想要知道表达式 b / a 是否大于 0:

```
>>> a = 2
>>> b = 1
>>> (b / a) > 0
True
```

考虑到 a 可能为 0,此时,解释器会引发异常:

```
>>> a = 0
>>> b = 1
>>> (b / a) > 0
Traceback (most recent call last):
  File "<pysHELL#136>", line 1, in <module>
    (b / a) > 0
ZeroDivisionError: division by zero
```

为了避免引发异常,可以这样修改代码:

```
>>> a != 0 and (b / a) > 0
False
```

当 a 为 0 时, a != 0 为 False,短路求值特性确保第二个表达式 (b / a) > 0 不被计算,从而避免引发异常。实际上,上述代码还可以编写得更简洁。当 a 为 0 时, a 本身就等价于 False,因此不需要显式地比较 a != 0:

```
>>> a and (b / a) > 0                                # a 非 0 时,才会计算第 2 个表达式
0
```

3.4 位运算符

位运算符将操作数看作二进制数字序列,并对其逐位操作。Python 语言支持的位运算符如表 3-4 所示。

表 3-4 位运算符

运算符	示 例	意 义	说 明
&	a & b	按位与	对应位的与运算,只有两者都是 1,才为 1;否则为 0
	a b	按位或	对应位的或运算,只要有一个为 1,则为 1;否则为 0
~	~a	按位取反	将每一位都逻辑求反,如果为 0,则为 1;如果为 1,则为 0
^	a ^ b	按位异或	对应位的异或运算,如果两者不同,则为 1;否则为 0
>>	a >> n	按位右移	将每一位都向右移动 n 位
<<	a << n	按位左移	将每一位都向左移动 n 位

```
>>> 0b1100 & 0b1010
8
>>> bin(8)                                #将十进制数 8 转换为二进制数
'0b1000'
>>> 0b1100 | 0b1010
```

```

14
>>> bin(14)                                #将十进制数 14 转换为二进制数
'0b1110'
>>> 0b1100 ^ 0b1010
6
>>> bin(6)                                #将十进制数 6 转换为二进制数
'0b110'
>>> 0b1100 >>2
3
>>> bin(3)
'0b11'
>>> 0b1100 <<2
48
>>> bin(48)
'0b110000'
>>> ~0b1100                                #二进制数 1100 等于十进制数 12
-13                                         #因此~12 等于-13
>>> bin(-13)
'-0b1101'

```

读者可以验证一下~12 等于-13 这一事实。注意,数据在计算机内部是以补码的形式存储的。

3.5 恒等运算符

Python 提供了两个恒等运算符(identity operator)is 和 is not,用于确定两个操作数是否恒等,即引用同一个对象。下面是两个相等但不相同的对象(不恒等):

```

>>> x = 1001
>>> y = 1000+1
>>> print(x, y)
1001 1001
>>> x == y
True
>>> x is y
False
>>> y is x
False

```

变量 x 和 y 都指向值为 1001 的对象,它们是相等的,但它们引用的不是同一个对象,这可以使用 id()函数进行验证:

```

>>> id(x)
2685066139824

```

```
>>> id(y)
2685066141040
```

当执行 $y=x$ 这样的赋值操作时,Python 只创建对同一个对象 x 的第二个引用 y :

```
>>> a = "just for test"
>>> b = a
>>> id(a)
2685066109872
>>> id(b)
2685066109872                                #变量 a 和 b 指向同一段内存地址
>>> a is b                                    #变量 a 和 b 指向同一个对象
True
>>> a == b                                    #变量 a 和 b 的值相等
True
```

运算符 `is not` 与 `is` 的逻辑功能相反:

```
>>> x = 5
>>> y = 10
>>> x is not y
True
```

3.6 运算符的优先级

在求解表达式的值时,必须按照运算符的优先级从高到低的顺序执行(括号除外)。
表 3-5 按照优先级从低到高的顺序列出 Python 的部分运算符。

表 3-5 运算符的优先级

运 算 符	说 明
or	逻辑或
and	逻辑与
not	逻辑非
=,!=,<,<=,>,>=,is,is not	优先级相同
	按位或
^	按位异或
&	按位与
<<,>>	按位左移、按位右移
+, -	加、减
*,/,//,%	乘、除、求整商、求余数
+x, -x, ~x	正号、负号、按位取反
**	幂运算

```
>>> 2 * 3 ** 2 * 4
72
```

#先计算 3 ** 2, 然后自左向右按顺序计算

可以使用小括号改变运算符的执行顺序:

```
>>> 10 + 2 * 5
20
>>> (10 + 2) * 5
60
```

有时使用小括号不是想要改变计算的先后顺序,而是为了提高代码的可读性,这是一种好的编程实践,避免了读者记忆运算符优先级的烦恼:

```
>>> (a < 10) and (b > 20)
>>> a < 10 and b > 20
```

#推荐的做法,尽管小括号是多余的
#不推荐

3.7 复合赋值运算符

在赋值符“=”之前加上其他运算符,就构成了复合赋值运算符。如在“=”前加一个“+”运算符就构成复合赋值运算符“+=”。

```
a += 1 等价于 a = a + 1
a %= 3 等价于 a = a % 3
a ^= 5 等价于 a = a ^ 5
```

算术运算符支持此种用法:

```
+=、-=、*、/=、%=、//=、**=
```

位运算符也支持此种用法:

```
&=、|=、^=、>>=和<<=
>>> a = 1
>>> a += 5
>>> a
6
```

3.8 小 结

本章讲述了各种运算符的用法,包括算术运算符、比较运算符、逻辑运算符、位运算符等。将赋值符“=”与其他运算符相结合就构成了复合赋值运算符,如“+=”。另外,逻辑运算符还有一个短路求值特性,利用这一特性可以编写更简洁、更高效的代码。

练习题 3

1. 已知 $x = 3$, 执行 $x *= 2 + 1$ 语句后 x 的值等于_____。
2. 如果数据在计算机内部用 1 个字节存储, 则 -5 的补码为_____。
3. 代码 $5 > 3+2$ 的执行结果是_____。
4. 请举例说明短路求值特性。
5. 请问代码 $(0.4-0.1) == 0.3$ 的执行结果是什么? 为什么会有这样的结果?
6. 请写出下列各表达式的值:
 - (1) `bool(1.0+1.0j)`
 - (2) `bool(0,0)`
 - (3) `bool(None)`
 - (4) `bool(list())`
 - (5) `bool('hello')`
 - (6) `bool(1)`
7. 已知 $x = 5, y = 3$, 求下列各表达式的值:
 - (1) $x \text{ or } y$
 - (2) $x \text{ and } y$
 - (3) `not x`
8. 已知 $a = 0b110, b = 0b101$, 求下列各表达式的值:
 - (1) $a \& b$
 - (2) $a | b$
 - (3) $a \wedge b$
 - (4) $\sim a$
 - (5) $a >> 1$
 - (6) $b << 1$
9. 已知 $a = \text{'hello'}, b = a$, 求 $a == b$ 的执行结果。
10. 写出整数 254 的个位数字、十位数字和百位数字的表达式。