APPENDIX E

BINARY NUMBERS AND BIT OPERATIONS

Binary Numbers

Decimal notation represents numbers as powers of 10, for example

$$1729_{\text{decimal}} = 1 \times 10^3 + 7 \times 10^2 + 2 \times 10^1 + 9 \times 10^0$$

There is no particular reason for the choice of 10, except that several historical number systems were derived from people's counting with their fingers. Other number systems, using a base of 12, 20, or 60, have been used by various cultures throughout human history. However, computers use a number system with base 2 because it is far easier to build electronic components that work with two values, which can be represented by a current being either off or on, than it would be to represent 10 different values of electrical signals. A number written in base 2 is also called a *binary* number.

For example,

$$1101_{\text{binary}} = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 8 + 4 + 1 = 13$$

For digits after the "decimal" point, use negative powers of 2.

$$\begin{aligned} 1.101_{\text{binary}} &= 1 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} \\ &= 1 + \frac{1}{2} + \frac{1}{8} \\ &= 1 + 0.5 + 0.125 = 1.625 \end{aligned}$$

In general, to convert a binary number into its decimal equivalent, simply evaluate the powers of 2 corresponding to digits with value 1, and add them up. Table 1 shows the first powers of 2.

To convert a decimal integer into its binary equivalent, keep dividing the integer by 2, keeping track of the remainders. Stop when the number is 0. Then write the remainders as a binary number, starting with the *last* one. For example,

$$\begin{aligned} 100 \div 2 &= 50 \text{ remainder } 0 \\ 50 \div 2 &= 25 \text{ remainder } 0 \\ 25 \div 2 &= 12 \text{ remainder } 1 \\ 12 \div 2 &= 6 \text{ remainder } 0 \\ 6 \div 2 &= 3 \text{ remainder } 0 \\ 3 \div 2 &= 1 \text{ remainder } 1 \\ 1 \div 2 &= 0 \text{ remainder } 1 \end{aligned}$$

Therefore, $100_{\text{decimal}} = 1100100_{\text{binary}}$.

Conversely, to convert a fractional number less than 1 to its binary format, keep multiplying by 2. If the result is greater than 1, subtract 1. Stop when the number is 0. Then use the digits before the decimal points as the binary digits of the fractional part, starting with the *first* one. For example,

$$0.35 \cdot 2 = 0.7$$
$$0.7 \cdot 2 = 1.4$$
$$0.4 \cdot 2 = 0.8$$
$$0.8 \cdot 2 = 1.6$$
$$0.6 \cdot 2 = 1.2$$
$$0.2 \cdot 2 = 0.4$$

Here the pattern repeats. That is, the binary representation of 0.35 is 0.01 0110 0110 0110 . . .

To convert any floating-point number into binary, convert the whole part and the fractional part separately.

Table 1 Powers of Two

Power	Decimal Value
2^0	1
2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
2^7	128
2^8	256
2^9	512
2^{10}	1,024
2^{11}	2,048
2^{12}	4,096
2^{13}	8,192
2^{14}	16,384
2^{15}	32,768
2^{16}	65,536

Two’s Complement Integers

Python uses the “two’s complement” representation for negative integers. To form the negative of an integer,

- Flip all bits.
- Then add 1.

For example, to compute −13 as an 8-bit value, first flip all bits of 00001101 to get 11110010. Then add 1:

$$-13 = 11110011$$

No special rules are required for adding negative numbers. Simply follow the normal rule for addition, with a carry to the next position if the sum of the digits and the prior carry is 2 or 3. For example,

```
       1 1111 111
+13      0000 1101
-13      1111 0011
       -----------
       1 0000 0000
```

But only the last 8 bits count, so +13 and −13 add up to 0, as they should.

In particular, −1 has two’s complement representation 1111 . . . 1111, with all bits set.

The leftmost bit of a two’s complement number is 0 if the number is positive or zero, 1 if it is negative.

Bit and Shift Operations

There are four bit operations in Python: the unary negation (~) and the binary *and* (&), *or* (|), and *exclusive or* (^), often called *xor*.

Table 2 and Table 3 show the truth tables for the bit operations in Python. When a bit operation is applied to integer values, the operation is carried out on corresponding bits.

For example, suppose you want to compute `46 & 13`. First convert both values to binary. $46_{\text{decimal}} = 101110_{\text{binary}}$ (actually 00000000000000000000000000101110 as a 32-bit integer), and $13_{\text{decimal}} = 1101_{\text{binary}}$. Now combine corresponding bits:

```
  0.....0101110
& 0.....0001101
---------------
  0.....0001100
```

The answer is $1100_{\text{binary}} = 12_{\text{decimal}}$.

Table 2 The Unary Negation Operation

a	~a
0	1
1	0

Table 3 The Binary And, Or, and Xor Operations

a	b	a & b	a | b	a ^ b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

You sometimes see the `|` operator being used to combine two bit patterns. For example, the symbolic constant `BOLD` is the value 1, and the symbolic constant `ITALIC` is 2. The binary *or* combination `BOLD | ITALIC` has both the bold and the italic bit set:

```
  0.....0000001
| 0.....0000010
---------------
  0.....0000011
```

Besides the operations that work on individual bits, there are two *shift* operations that take the bit pattern of a number and shift it to the left or right by a given number of positions.

The left shift (<<) moves all bits to the left, filling in zeroes in the least significant bits. Shifting to the left by n bits yields the same result as multiplication by 2^n. The right shift (>>) moves all bits to the right, propagating the sign bit. Therefore, the result is the same as integer division by 2^n, both for positive and negative values.

The expression

```
1 << n
```

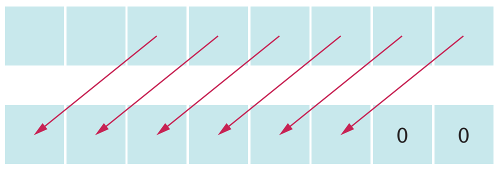
yields a bit pattern in which the `n`th bit is set (where the `0` bit is the least significant bit).

To set the `n`th bit of a number, carry out the operation

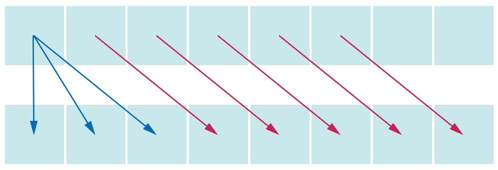
```
x = x | 1 << n
```

To check whether the `n`th bit is set, execute the test

```
if (x & 1 << n) != 0 :
```



Left shift (<<)



Right shift with sign extension (>>)

Figure 1 The Shift Operations