Floating-point numbers are represented in computer hardware as base 2 (binary) fractions. Most decimal fractions cannot be represented exactly as binary fractions. A consequence is that, in general, the decimal floating-point numbers you enter are only approximated by the binary floating-point numbers actually stored in the machine.

The core idea of floating-point representations is that a number **x** is written as $m*b^e$ where ***m*** is a **mantissa** or fractional part, ***b*** is a **base**, and ***e*** is an **exponent**.

Convert 2.625 to our 8-bit floating point format:

A. The integral part is easy, $2_{10} = 10_2$.
B. For the fractional part:

$0.625 \times 2 = 1.25$	1	*Generate 1 and continue with the rest.*
$0.25 \times 2 = 0.5$	0	*Generate 0 and continue.*
$0.5 \times 2 = 1.0$	1	*Generate 1 and nothing remains.*

C. So $0.625_{10} = 0.101_2$, and $2.625_{10} = 10.101_2$.
D. Add an exponent part: $10.101_2 = 10.101_2 \times 2^0$.
E. Normalize: $10.101_2 \times 2^0 = 1.0101_2 \times 2^1$.
F. Mantissa: 0101
G. Exponent: $1 + 3 = 4 = 100_2$. The bias is $2^{k-1} - 1$, where k is the number of bits in the exponent field. For the eight-bit format, $k = 3$, so the bias is $2^{3-1} - 1 = 3$. Note, the 8-bit format is useful for instruction, not of much practical value for representing numbers. In practice the 32-bit IEEE standard format is used.
H. Sign bit is 0.

The result is

0	1	0	0	0	1	0	1

Convert decimal 1.7 to our 8-bit floating point format:

A. The integral part is easy, $1_{10} = 1_2$.
B. For the fractional part:

$0.7 \times 2 = 1.4$	1	*Generate 1 and continue with the rest.*
$0.4 \times 2 = 0.8$	0	*Generate 0 and continue.*
$0.8 \times 2 = 1.6$	1	*Generate 1 and continue with the rest.*
$0.6 \times 2 = 1.2$	1	*Generate 1 and continue with the rest.*
$0.2 \times 2 = 0.4$	0	*Generate 0 and continue.*
$0.4 \times 2 = 0.8$	0	*Generate 0 and continue.*
$0.8 \times 2 = 1.6$	1	*Generate 1 and continue with the rest.*
$0.6 \times 2 = 1.2$	1	*Generate 1 and continue with the rest.*

…

- The reason why the process seems to continue endlessly is that it does. We cannot represent this exactly as a floating-point number. The closest we can come in four bits is .1011. Since we already have a leading 1, the best eight-bit number we can make is 1.1011.

C. Already normalized: $1.1011_2 = 1.1011_2 \times 2^0$.
D. Mantissa is 1011, exponent is $0 + 3 = 3 = 011_2$, sign bit is 0.

The result is

0	0	1	1	1	0	1	1

Convert the 8-bit floating-point binary number 00111011 to decimal:

A. Exponent: $011_2 = 3_{10}$; $3 - 3 = 0$.
B. It is already normalized: $1.1011_2 \times 2^0 = 1.1011$.
C. Convert:

Exponents	2^0		2^{-1}		2^{-2}		2^{-3}		2^{-4}		
Place Values	1		0.5		0.25		0.125		0.0625		
Bits	1	.	1		0		1		1		
Value	1	+	0.5	+	0	+	0.125	+	0.0625	=	1.6875

D. Sign: positive

Result: 0111011 is 1.6875, which, however is not equal to 1.7

It's easy to forget that the stored value is an approximation to the original decimal fraction, because of the way that floats are displayed at the interpreter prompt. It's important to realize that this is, in a real sense, an illusion: the value in the machine is not exactly 17/10, you're simply rounding the *display* of the true machine value.

A **round-off error**, also called **rounding error**, is the difference between the calculated approximation of a number and its exact mathematical value due to rounding. It is occurring when using finitely many digits (due to memory limits) to represent real numbers (which in theory have infinitely many digits).