B Appendix B Number Representations

> There is no art in being commonly intelligible,
> if one sacrifices all well-grounded insight.
>
> Immanuel Kant

We now discuss floating-point numbers more carefully than in chapter 2. As you may recall, computers use *binary digits or bits*: bits can take on only two possible values, by convention 0 or 1; compare with decimal digits, which can take on 10 different values, from 0 to 9. Bits are grouped to form *bytes*: 1 byte ≡ 1 B ≡ 8 bits ≡ 8 b.

B.1 Integers

The number of bytes needed to store variables of different type varies (by implementation, language, etc.), but nowadays it is common that characters use 1 B, integers use 4 B, and long integers use 8 B; trying to store an integer that's larger than that leads to overflow. This doesn't actually happen in Python, because Python employs arbitrary-precision integers.

For simplicity, let us start with an example for the case of 4 bits:

$$1100_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 12_{10} \tag{B.1}$$

We use subscripts to denote binary and decimal numbers; the equality sign is here used loosely. We're interested only in *unsigned integers*, where all bits are used to represent powers of 2. Applying the logic of this equation, you can also show that, e.g., $0011_2 = 3_{10}$. We see that, using 4 bits, the biggest integer we can represent is:

$$1111_2 = 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 15_{10} \tag{B.2}$$

Note that $2^4 - 1 = 15$. More generally, with n bits we can represent integers in the range $[0, 2^n)$, i.e., the largest integer possible is $2^n - 1$. Here's another example with $n = 6$ bits:

$$100101_2 = 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 37_{10} \tag{B.3}$$

Explicitly check that the maximum integer representable with 6 bits is $2^6 - 1 = 63$.

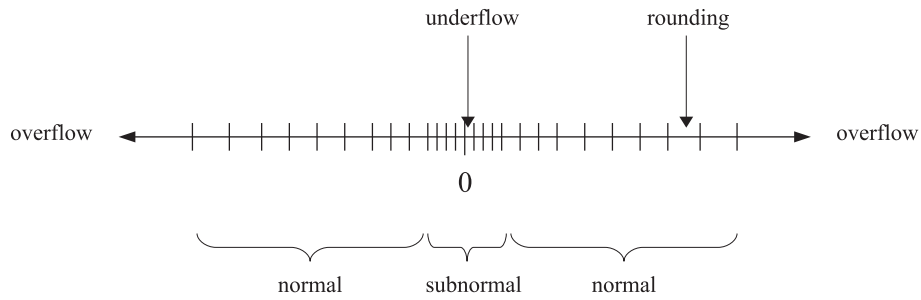


Illustration of exactly representable floating-point numbers **Fig. B.1**

B.2 Real Numbers

Several ways of storing real numbers on a computer exist. The simplest is known as *fixed-point representation*, but this is not commonly used in computational science. Most often, real numbers on a computer employ *floating-point representation*: ± mantissa × 10^{exponent}. For example, the speed of light in scientific notation is $+2.997\,924\,58 \times 10^8$ m/s. The mantissa here has nine significant figures.

Computers store only a finite number of bits, so cannot store exactly all possible real numbers. As a result, there are "only" finitely many exact representations/machine numbers; this refers to finitely many decimal numbers that can be stored exactly using a floating-point representation. There are three ways of losing precision, as shown qualitatively in Fig. B.1 (which you also encountered as Fig. 2.2): *underflow* for very small numbers, *overflow* for very large numbers, and *rounding* for decimal numbers whose value falls between two exactly representable numbers.

In the past, there existed a polyphony of conventions (in programming languages and operating systems) on how to treat floating-point representations; nowadays, the Institute of Electrical and Electronics Engineers has established a Standard for Floating-Point Arithmetic (known as *IEEE 754*), which is widely implemented. As usual, we go over only selected features.[1] The general convention is:

$$x_{\text{floating point}} = (-1)^s \times 1.f \times 2^{e-bias} \tag{B.4}$$

where: (a) s is the *sign bit*, (b) $1.f$ follows the convention of starting with a 1 (called a *hidden bit*) and then having the *fraction* of the *mantissa*,[2] and (c) the (fixed-value) *bias* allows us to use an *exponent* e that's always positive, i.e., it is an unsigned integer (but then $e - bias$ can range from negative to positive values). We need to store s, f, and e, following the above convention. We go over two cases: singles and doubles.

[1] We do not discuss the alternative concept of universal numbers, *unum*s, and their latest incarnation as *posit*s. This mention of them should be enough for you if you want to delve into this topic further.

[2] What we colloquially refer to as the mantissa is more properly called the *significand*.

B.2.1 Single-Precision Floating-Point Numbers

These are also called "singles" or "floats"; note that *Python floats are actually doubles*.[3] The storage of singles uses 4 B or 32 b in total. These are distributed as follows: 1 b for the sign s, 8 b for the exponent e, and 23 b for the fraction of the mantissa f (the *bias* doesn't have to be stored for any given single, as it has the same value for all singles.) Specifically, we store the sign s in the most-significant bit, then the exponent e, and then the fraction of the mantissa f, so the 32 bits are stored as:

s	e	f
31	30 . . . 23	22 . . . 0

(B.5)

If $s = 0$ we have $(-1)^s = +1$, whereas if $s = 1$ we have $(-1)^s = -1$.

Exponent The e is always positive (unsigned) and stored in 8 bits so, given that $2^8 - 1 = 255$, the exponent values go from $0000\ 0000_2 = 0_{10}$ to $1111\ 1111_2 = 255_{10}$. Thus, we have $0 \leq e \leq 255$. In reality, $e = 0$ and $e = 255$ are special cases, so we have $1 \leq e \leq 254$. The bias for all singles is $bias = 127$ so the actual exponent in 2^{e-bias} takes the values $-126 \leq e - 127 \leq 127$. In other words, numbers with $1 \leq e \leq 254$ are called *normal* and are represented as per Eq. (B.4):

$$x_{\text{normal single}} = (-1)^s \times 1.f \times 2^{e-127} \tag{B.6}$$

Now for the special cases:

- $e = 255$ with $f = 0$: this is either $+\infty$ (for $s = 0$) or $-\infty$ (for $s = 1$).
- $e = 255$ with $f \neq 0$: this is NaN (not a number).

We return to the remaining special case, $e = 0$, below.

Mantissa We have at our disposal 23 bits for the fraction f of the mantissa. Denoting them by $m_{22}, m_{21}, m_{20}, \ldots, m_1, m_0$, they are combined together as follows:

$$\text{normal single mantissa} = 1.f = 1 + m_{22} \times 2^{-1} + m_{21} \times 2^{-2} + \cdots + m_0 \times 2^{-23} \tag{B.7}$$

This expression only includes powers of inverse 2, i.e., it cannot be used to represent numbers that cannot be expressed as a sum of powers of inverse 2. You'll do an example of a "middle-of-the-road" floating-point single-precision number in a problem.

Largest normal single For now, let us examine the largest possible positive normal single. The largest possible f is all 1s, so:[4]

$$\begin{aligned} \text{normal single mantissa}\Big|_{max} &= 1.1111\ 1111\ 1111\ 1111\ 1111\ 111 \\ &= 1 + 0.5 + 0.25 + \cdots \approx 2 \end{aligned} \tag{B.8}$$

[3] Core Python has no singles, but NumPy allows you to control the data type in finer detail, see section 1.6.
[4] We stop explicitly denoting numbers as binary or decimal; the distinction should be clear from the context.

On the other hand, we already know that the largest possible normal exponent e is 254 (since 255 is a special case, as seen above). In binary, this is 1111 1110. Putting it all together, the largest normal single is stored as:

$$x_{\text{normal single}}\Big|_{max} = 0\ \ 1111\ 1110\ \ 1111\ 1111\ 1111\ 1111\ 1111\ 111 \tag{B.9}$$

This starts with a positive sign bit, then moves on to the largest normal exponent, and then on to the largest possible fraction of the mantissa. Its value is:

$$(-1)^s \times 1.f \times 2^{e-127} \approx +1 \times 2 \times 2^{127} = 2^{128} \approx 3.4 \times 10^{38} \tag{B.10}$$

Subnormal singles We left the special case $e = 0$ for last. This is used to represent numbers that are smaller than the smallest normal number, as per Fig. B.1. These are called *subnormal* numbers and are represented as follows:

$$x_{\text{subnormal single}} = (-1)^s \times 0.f \times 2^{-126} \tag{B.11}$$

Comparing this with Eq. (B.6) we observe, first, that the sign convention is the same. Second, the subnormal mantissa convention is slightly different: it's written $0.f$, so the bits in the fraction of the mantissa for subnormals are interpreted as:

$$\text{subnormal single mantissa} = 0.f = m_{22} \times 2^{-1} + m_{21} \times 2^{-2} + \cdots + m_0 \times 2^{-23} \tag{B.12}$$

where there is no leading 1 (since there is a leading 0). Third, the exponent term is 2^{-126}, which is different from what we would get from Eq. (B.6) if we plugged in $e = 0$. The vertical lines in Fig. B.1 are not equidistant from each other: this is most easily seen in the case of subnormal numbers, which are more closely spaced than normal numbers.

Incidentally, note that if all the bits in f are 0 we are left with the number zero: since we still have a sign bit, s, this gives rise to a *signed zero*, meaning that +0 and −0 are two different ways of representing the same real number.

Smallest subnormal single The smallest possible (non-zero) positive subnormal single is also the smallest possible positive single-precision floating-point number (since subnormals are smaller than normals). The smallest possible positive subnormal single has only a single 1 in the mantissa:

$$\text{subnormal single mantissa}\Big|_{min} = 0.0000\ 0000\ 0000\ 0000\ 0000\ 001 = 2^{-23} \tag{B.13}$$

On the other hand, we already know that the exponent is $e = 0$, since we're dealing with a subnormal. Putting it all together:

$$x_{\text{subnormal single}}\Big|_{min} = 0\ \ 0000\ 0000\ \ 0000\ 0000\ 0000\ 0000\ 0000\ 001 \tag{B.14}$$

This starts with a positive sign bit, then moves on to an exponent that is zero, and then on to the smallest possible fraction of the mantissa. Its value is:

$$(-1)^s \times 0.f \times 2^{-126} = +1 \times 2^{-23} \times 2^{-126} = 2^{-149} \approx 1.4 \times 10^{-45} \tag{B.15}$$

Summary Singles can represent:

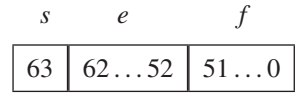
$$\pm 1.4 \times 10^{-45} \leftrightarrow \pm 3.4 \times 10^{38} \tag{B.16}$$

This refers to the ability to store very large or very small numbers; most of this ability is found in the term corresponding to the exponent. If we try to represent a number that's larger than $2^{128} \approx 3.4 \times 10^{38}$ we get *overflow* (up to infinity). Similarly, if we try to represent a number that's smaller than $2^{-149} \approx 1.4 \times 10^{-45}$ we get *underflow* (down to 0).

Being able to represent 1.4×10^{-45} does *not* mean that we are able to store 45 significant figures in a single. The number of significant figures (and the related concept of *precision*) is found in the mantissa. For singles, the precision is 1 part in $2^{23} \approx 1.2 \times 10^{-7}$, which amounts to six or seven decimal digits.

B.2.2 Double-Precision Floating-Point Numbers

These are also called "doubles"; keep in mind that Python floats are actually doubles. You will probe these more deeply in a problem – here we just summarize several facts. Their storage uses 8 B or 64 b in total. These are distributed as follows: 1 b for the sign s, 11 b for the exponent e, and 52 b for the fraction of the mantissa f.[5] Specifically, we store the sign s in the most-significant bit, then the exponent e, and then the fraction of the mantissa f. In other words, similarly to what we did for singles, the 64 bits of a double are stored as follows:



(B.17)

Exponent The e is always positive (unsigned) and stored in 11 bits so, given that $2^{11} - 1 = 2047$, the exponent goes from $0000\ 0000\ 000_2 = 0_{10}$ to $1111\ 1111\ 111_2 = 2047_{10}$. Thus, we have $0 \leq e \leq 2047$. As you may have guessed, $e = 0$ and $e = 2047$ are special cases, so we have $1 \leq e \leq 2046$. The bias for all doubles is $bias = 1023$ so the actual exponent in 2^{e-bias} takes the values $-1022 \leq e - 1023 \leq 1023$. In other words, numbers with $1 \leq e \leq 2046$ are called *normal* and are represented as per Eq. (B.4):

$$x_{\text{normal double}} = (-1)^s \times 1.f \times 2^{e-1023} \tag{B.18}$$

Mantissa The fraction of the mantissa is interpreted analogously to what we did for singles. There are 52 bits, $m_{51}, m_{50}, m_{49}, \ldots, m_1, m_0$, which appear in:

$$\text{normal double mantissa} = 1.f = 1 + m_{51} \times 2^{-1} + m_{50} \times 2^{-2} + \cdots + m_0 \times 2^{-52} \tag{B.19}$$

for normal doubles. We do not provide more details here because everything works by analogy to the case of singles.

[5] Again, the *bias* doesn't have to be stored, as it has the same value for all doubles.

Summary Doubles can represent:

$$\pm\, 4.9 \times 10^{-324} \leftrightarrow \pm 1.8 \times 10^{308} \tag{B.20}$$

This refers to the ability to store very large or very small numbers. As mentioned above, most of this ability is found in the term corresponding to the exponent. If we try to represent a number that's larger than 1.8×10^{308} we get *overflow*. Similarly, if we try to represent a number that's smaller than 4.9×10^{-324} we get *underflow*.

Again, being able to represent 4.9×10^{-324} does *not* mean that we are able to store 324 significant figures in a double. The number of significant figures (and the related concept of *precision*) is found in the mantissa. For doubles, the precision is 1 part in $2^{52} \approx 2.2 \times 10^{-16}$, which amounts to 15 or 16 decimal digits. Observe that both the digits of precision and the range in Eq. (B.20) are much better than the corresponding results for singles.

B.3 Problems

1. We saw how to convert binary integers to decimal, for example:
   $0101_2 \rightarrow 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5_{10}$
   By analogy to this, write a program that breaks decimal numbers down, e.g.:
   $5192_{10} \rightarrow 5 \times 10^3 + 1 \times 10^2 + 9 \times 10^1 + 2 \times 10^0$
   Do this for any four-digit integer the user inputs. (Use * instead of $\times$). Can you generalize this to work for an integer of any number of digits?
2. Convert the following single-precision floating point numbers from binary to decimal:
   (a) `0 0000 0011 0111 1011 0000 0000 0000 000`
   (b) `1 0101 0101 0111 0110 0000 0000 0001 011`
3. Find the smallest possible (non-zero) *normal* single-precision floating-point number.
4. Repeat our singles argument for doubles and thereby explicitly show Eq. (B.20).