

COMP1730/COMP6730

Programming for Scientists

Floating point numbers



Outline

- * Numbers in binary and other bases
- * Floating point numbers



Representing Integers

Sequential encoding

- * A *sequential encoding system* represents each item (words, numbers, etc) by a sequence of symbols; the order (position) of a symbol in the sequence carries meaning, as much as the symbol itself.
- * For example,
 - "representation" \neq "interpret as one"
 - 007 \neq 700

Positional number system

- * The position of a digit is the power of the *base* that it adds to the number.
- * For example, in base 10:

1864

= 1 thousand 8 hundreds 6 tens 4 ones

$$= 1 \times 10^3 + 8 \times 10^2 + 6 \times 10^1 + 4 \times 10^0$$

- * The position of the least significant digit is 0.
($b^0 = 1$ for any base b .)
- * The representation of any (non-negative integer) number is unique, except for leading zeros.

We can count in any base

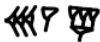
- * For example, in base 3:

$$\begin{aligned} & 2120001_3 \\ = & 2 \times 3^6 \\ & + 1 \times 3^5 + 2 \times 3^4 + 0 \times 3^3 \\ & + 0 \times 3^2 + 0 \times 3^1 + 1 \times 3^0 \\ = & 2 \times 729 + 243 + 2 \times 81 + 1 \\ = & 1864 \end{aligned}$$



- * Each digit is one of $0, \dots, b - 1$.
- * (“ $nnnn_b$ ” means a number in base b .)

- * Ancient Babylonians
(ca 2,000 BC)
counted in base 60.



$$\begin{aligned}
 &= 31 \times 60^1 + 4 \times 60^0 \\
 &= 1864
 \end{aligned}$$

𐎶 1	𐎶𐎶 11	𐎶𐎶𐎶 21	𐎶𐎶𐎶𐎶 31	𐎶𐎶𐎶𐎶𐎶 41	𐎶𐎶𐎶𐎶𐎶𐎶 51
𐎶𐎶 2	𐎶𐎶𐎶 12	𐎶𐎶𐎶𐎶 22	𐎶𐎶𐎶𐎶𐎶 32	𐎶𐎶𐎶𐎶𐎶𐎶 42	𐎶𐎶𐎶𐎶𐎶𐎶𐎶 52
𐎶𐎶𐎶 3	𐎶𐎶𐎶𐎶 13	𐎶𐎶𐎶𐎶𐎶 23	𐎶𐎶𐎶𐎶𐎶𐎶 33	𐎶𐎶𐎶𐎶𐎶𐎶𐎶 43	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 53
𐎶𐎶𐎶𐎶 4	𐎶𐎶𐎶𐎶𐎶 14	𐎶𐎶𐎶𐎶𐎶𐎶 24	𐎶𐎶𐎶𐎶𐎶𐎶𐎶 34	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 44	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 54
𐎶𐎶𐎶𐎶𐎶 5	𐎶𐎶𐎶𐎶𐎶𐎶 15	𐎶𐎶𐎶𐎶𐎶𐎶𐎶 25	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 35	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 45	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 55
𐎶𐎶𐎶𐎶𐎶𐎶 6	𐎶𐎶𐎶𐎶𐎶𐎶𐎶 16	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 26	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 36	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 46	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 56
𐎶𐎶𐎶𐎶𐎶𐎶𐎶 7	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 17	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 27	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 37	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 47	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 57
𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 8	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 18	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 28	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 38	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 48	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 58
𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 9	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 19	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 29	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 39	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 49	𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶𐎶 59
𐎶 10	𐎶𐎶 20	𐎶𐎶𐎶 30	𐎶𐎶𐎶𐎶 40	𐎶𐎶𐎶𐎶𐎶 50	

- * However, they did not have a symbol for 0: 𐎶
can mean 1, 60, 3600, $1/60$, etc.

Binary numbers

- * Binary numbers are simply numbers in base 2.

$$\begin{aligned} & 11101001000_2 \\ = & 1 \times 2^{10} \\ & + 1 \times 2^9 + 1 \times 2^8 + 0 \times 2^7 + 1 \times 2^6 + 0 \times 2^5 \\ & + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\ = & 1024 + 512 + 256 + 64 + 8 \\ = & 1864 \end{aligned}$$

Arithmetic

- * Long (multi-digit) addition, subtraction, multiplication, division and comparison (of non-negative numbers) work the same way in any base.

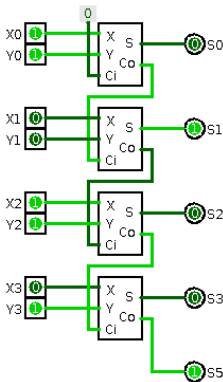
$0_2 + 0_2 = 0_2$
$0_2 + 1_2 = 1_2$
$1_2 + 0_2 = 1_2$
$1_2 + 1_2 = 10_2$

$$\begin{array}{r}
 111 \\
 0101_2 \\
 + 0111_2 \\
 \hline
 1100_2
 \end{array}$$

$$\begin{array}{r}
 1001_2 \\
 \times 101_2 \\
 \hline
 1001_2 \\
 00000_2 \\
 100100_2 \\
 \hline
 101101_2
 \end{array}$$

Bits and bytes

- * In the electronic computer, a single binary digit (*bit*) is represented by the presence or absence of current in a circuit element.
- * 8 bits make an *octet*, or *byte*.
- * Digital hardware works with *fixed-width* number representations (“*words*”).
- * Common word sizes: 32-bit, 64-bit.



Note: Python allows arbitrarily large integers!



Floating point numbers

Representing fractional numbers

- * Extend the number system to negative positions; decimal point marks position zero.

$$0.25_{10}$$

$$= 0 \times 10^0 + 2 \times 10^{-1} + 5 \times 10^{-2}$$

$$= 0 \times 1 + 2 \times 1/10 + 5 \times 1/100$$

$$0.01_2$$

$$= 0 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2}$$

$$= 0 \times 1 + 0 \times 1/2 + 1 \times 1/4$$

$$= 0.25_{10}$$

- ★ Not every fraction has a finite decimal expansion in a given base.
- ★ For example,
 - $1/3 = 0.3333 \dots$ in base 10
 - $1/5 = 0.001100110011 \dots$ in base 2
 - $1/3 = 0.1$ in base 3.
- ★ Because digital computers work with numbers of fixed width, decimal representation of fractions have *finite precision*.

Floating point representation

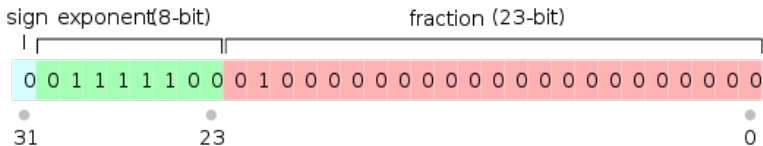
- * A floating point number in base b ,

$$x = \pm m \times b^e$$

consists of three components:

- the sign (+ or -);
 - the *significand* (m);
 - the *exponent* (e);
- * The number is *normalised* iff $1 \leq m < b$.
 - * **Is 0 a normalised number?**

- * Floating point types, as implemented in computers, use *fixed-width* binary integer representation of the significand and exponent.
- * In a normalised binary number the first digit is 1, so only the fraction is represented ($m = 1.f$).
- * The exponent is biased by a negative constant.
- * IEEE standard formats:
 - single: 23-bit fraction, 8-bit exponent.
 - double: 52-bit fraction, 11-bit exponent.
- * Standard also specifies how to represent 0, $+\infty$, $-\infty$ and nan (“not a number”).



$$\begin{aligned}
 x &= (-1)^s (1.f)_2 2^{(e-127)} \\
 &= (-1)^0 (1.01)_2 2^{01111100_2-127} \\
 &= (1 + 1 \cdot 2^{-2}) 2^{(64+32+16+8+4)-127} \\
 &= (1.25)2^{-3} = (1.25)/8 = 0.15625
 \end{aligned}$$

(Image from wikipedia.org)

- * Type `float` can represent infinity:

```
>>> 1 / 1e-320  
inf
```

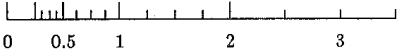
- * Most math functions raise an error rather than return `inf`.
 - For example, `1 / 0`, or `math.log(0)`.
- * `nan` (“not a number”) is a special value used to indicate errors or undefined results.

```
>>> (1 / 1e-320) - (1 / 1e-320)  
nan
```

- * `math.isinf` and `math.isnan` functions.

Floating point number systems

- * A floating point number system (b, p, L, U) is defined by four parameters:
 - the base (b) ;
 - the *precision*: number of digits in the fraction of the significand (p) ; and
 - the lower (L) and upper (U) limit of the exponent.
- * IEEE double-precision is $(2, 52, -1023, 1024)$ (with some tweaks).

- * The numbers that can be represented (exactly) in a floating point number system are not evenly distributed on the real line.
- * E.g., $(2, 2, -2, 1)$: 
- * E.g., in a $(2, 52, -1023, 1024)$ system,
 - the smallest number > 0 is $2^{-1023} \approx 10^{-308}$,
 - (Actual IEEE double standard can represent numbers down to $\approx 4 \cdot 10^{-324}$.)
 - the smallest number > 1 is $1 + 2^{-52} \approx 1 + 2 \cdot 10^{-16}$.
- * Rounding the significand to $p + 1$ digits causes a discrepancy, called the rounding error.

- ★ Because of rounding, mathematical laws do not always hold for floating point arithmetic.

```
>>> a = 11111113.0
>>> b = -11111111.0
>>> c = 7.51111111
>>> (a + b) + c == a + (b + c)
False
>>> ((a + b) + c) - (a + (b + c))
4.488374116817795e-10
```

Example from Punch & Enbody

- ★ *(Almost) never compare floats with ==.*



Take home message

- * Computer numbers are not the same as mathematical numbers!