Lecture 4: Number Representations

Number Representations

Why do we care how numbers are stored?

- Finite precision of stored numbers source round-off errors. (next weeke)
- Each data type allowed range of values is based on memory architecture

Integer Representations

bit = ☐ = smallest unit of information for a computer

2 possible values = [0] or [1] (can store Boolean or True/False)

byte = 8 bits = ☐☐☐☐☐☐☐☐ all 0's & 1's

Ex: [1][0][1][1][0][0][1][0]

Integers can be stored as a byte or a set of bytes using base-2 (or binary) numbers

Binary numbers (how computers count)

$$N = \sum_{i=0}^{n-1} b_i \times 2^i$$

- n = number of bits
- b_i: 0 or 1
- 2^i would be a 10 in decimal numbers

Integers represented like this are "unsigned 8-bit integers"

example: represent 01001101 in decimal

$$01001101 = 0\times2^7 + 1\times2^6 + 0\times2^5 + 0\times2^4 + 1\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0$$
$$= 64 + 8 + 4 + 1 = 77$$

Ex: Convert 158 to binary (base-2).

- biggest power of 2 less than 158: $128 = 2^7$
- $158 - 128 = 30$
- biggest power of 2 less than 30: $16 = 2^4$
- $30 - 16 = 14$
- biggest power of 2 less than 14: $8 = 2^3$
- $14 - 8 = 6$
- biggest power of 2 less than 6: $4 = 2^2$ ($6 - 4 = 2 = 2^1$)
- $158 = 2^7 + 2^4 + 2^3 + 2^2 + 2^1$

$= \boxed{1}\boxed{0}\boxed{0}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{0}$

1 byte stores 256 integers

What's the largest unsigned 8-bit integer?

$N_{max} = \boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}$

$= \underbrace{\sum_{i=0}^{8-1} 1 \times 2^i}_{\text{geometric series}} = \frac{1-2^8}{1-2} = 255$ (calculate another way?)

Incrementing past $N = 255$ will go back to zero.

$\boxed{255 + 1 = 0}$

"Signed 8-bit integer" — use to store positive + negative integers using 1st bit — $\boxed{0} = +$, $\boxed{1} = -$

$N_{max} = \boxed{0}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1}\boxed{1} = \sum_{i=0}^{7-1} 2^i = 127$

You can't just treat the lower 7 bits as the magnitude of the integer; otherwise adding 1 bitwise would reduce the value.

Instead, we set $\boxed{1}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0} = -128$

Higher binary numbers are used to store negative numbers

$\boxed{127 + 1 = -128}$

-128 127

10000000 + 01111111 = 11111111 = -1

Python 2 standard: signed 32-bit, range $\approx [-2, 2] \times 10^9$, "int"

others available: 16-bit ("short")

64-bit ("long")

Python 3: Starts with 64-bits ($N_{max} \sim 10^{19}$) but adds bytes for higher numbers (unlimited range)

Floating-Point Representations

Bits can also store non-integers called "floats"

Python standard: double precision, 64 bits ("float 64" or "float")

can also use single precision, 32 bits ("float 32")

Floats are stored in exponential notation

$$\pm 2.383 \times 10^{-45}$$

sign — mantissa — exponent

these 3 parts are stored in bits

32-bit: sign gets 1 bit (bit 31)

exponent gets 8 bits (bits 30-23)

mantissa gets 23 bits (bits 22-0)

$$float(X) = \pm \left(1 + \sum_{i=0}^{22} f_i \, 2^{-(23-i)}\right) \times 2^{e-127}$$

normal numbers have $1 \le e \le 254$

Ex:

	s	e	f
Bit position	31	30 ... 23	22 ... 0
Value	1	0001 1010	1100 0100 0000 0000 0000 000

$s=1 \rightarrow$ negative, $e = 00011010 = 2^4 + 2^3 + 2^1 = 16+8+2 = 26$

$$\sum_{i=0}^{22} f_i \times 2^{-(23-i)} = 2^{-1} + 2^{-2} + 2^{-6} = 0.765625$$

$$\Rightarrow \text{number} = -1.765625 \times 2^{-101} \approx -6.964162679 \times 10^{-31}$$

Show

Bit position	s	e	f	
	31	30 ... 23	22 ...	0
Value	0	1010 1100	1101 0000 0000 0000 0000 000	

$= 1.8125 \times 2^{45} \approx 6.3771 \times 10^{13}$

special cases: $e=0, f \neq 0$ – "subnormal numbers"
- replace "1" with 0 in mantissa & "127" with 126 in exponent

$e=0, f=0$ – underflow

$e=255, f=0$ – overflow (+ or −)

$e=255, f \neq 0$ – NaN or "Not a Number" (usually $0/0$ or $\sqrt{-1}$)

Ex:

Bit position	s	e	f	
	31	30 ... 23	22 ...	0
Value	0	0000 0000	1100 0100 0000 0000 0000 000	

$= 0.765625 \times 2^{-126} = 9.000172497 \times 10^{-39}$

$$X_{max} \text{ (single)} = \left(1 + \sum_{i=0}^{22} 1 \times 2^{-(23-i)}\right) \times 2^{254-127} = \left[1 + 2^{-23}\left(2^{22+1} - 1\right)\right] 2^{127}$$

$$\approx \boxed{2^{128} = 3.4 \times 10^{38}}$$

$$X_{min} \text{ (single, positive)} = \left(0 + 2^{-23}\right) \times 2^{0-126} = \boxed{2^{-149} = 1.4 \times 10^{-45}}$$

$f_{i=0} = 1, f_{i=1-22} = 0$

$e = 0$

What's the lowest normal (positive) number? the highest subnormal?

Double precision - standard for Python 3

$$float(x) = (-1)^s \left[1 + \sum_{i=0}^{51} f_i \times 2^{-(52-i)}\right] \times 2^{e-1023}$$

exponent gets 11 bits
mantissa gets 52 bits

$e=0, f\neq 0$ – subnormal
- remove "1." in mantissa
- replace 1023 → 1022 in exponent

$$X_{max} \simeq (1+1) \times 2^{2046-1023} = 2^{1024} \simeq 1.8 \times 10^{308}$$
$$X_{min} = (0+2^{-52}) \times 2^{-1022} = 2^{-1074} \simeq 4.94 \times 10^{-324}$$

Round-off Error

How do you add $3.5 \times 10^4 + 86. \times 10^2$ like a computer (without expanding exponent)? Move powers of 10 in the smaller number from the exponent to the mantissa until exponents agree!

$$3.5 \times 10^4 + 8.6 \times 10^2 = 3.5 \times 10^4 + 0.086 \times 10^4 = \boxed{3.586 \times 10^4}$$

For computations, this is limited by the smallest mantissa that can be stored.
- single precision – $\delta x = 2^{-23} \simeq 10^{-7}$ – holds 7 decimals
- double precision – $\delta x = 2^{-52} \simeq 10^{-16}$ – holds 16 decimals

How about $1.0 + 10^{-8}$ in single-precision?

$1.0 + 10^{-8} = 1.0 + 0.00000000$ ✗ – 8th digit cannot be stored, is dropped
$= 1.0 + 0.0000000$
$= 1.0 \neq \boxed{1.0000000\,1}$ – cannot be stored in single precision

adding 2 numbers separated by several orders of magnitude can give you a "round-off error".

Also present for other operations. (multiply, power, etc.)

Leads to really large errors when subtracting 2 numbers that are really close together

This is even a problem when adding lots of small numbers; one at a time is not good. Why?

See Jupyter notebook for examples