

King of Dumbs

Level: ♠ Technical

Why does the computer say $0.1 + 0.2$ isn't 0.3 ?

0.1 doesn't fit in binary, so the computer rounds it. What floating point is, why the error adds up and what AI and money use. In three levels, with a widget.

English Published October 15, 2026 Updated October 16, 2026



Scan to open the interactive lesson

<https://kingofdumbs.com/en/lessons/floating-point>

What you'll learn

- Why 0.1 can't be stored exactly in binary
- Floating point: scientific notation in binary with a fixed number of digits
- How rounding adds up and how to compare with a tolerance
- float32, float16 and bfloat16 in AI, and decimals for money

Before this lesson

If something sounds like gibberish, these lessons explain it from scratch:

1. [How much is \$7 \times 10^{27}\$? Huge numbers without fear](#)

Symbols you'll meet

Δ change: how much it moved (♠) · E_a subscript: the label underneath (♠) · Σ sum: add them all up (♠) · x_i index i: item number i (♠) · $\approx$ approximately equal (♠) · $\leq \geq$ less or equal / greater or equal (♠) · $|x|$ absolute value: drop the sign (♠) · 10^{-3} negative exponent: divide (♦♠) · $\log$ logarithm: how many times do I multiply? (♠) · $\lfloor x \rfloor$ floor: round down (♠) · ϵ epsilon: something tiny (♠) · δ small delta: a small error (♠) · $\in$ belongs to (♠)

Open the browser on your computer, press F12 to open the console, type `0.1 + 0.2` and press Enter. The answer is `0.30000000000000004`. Try Python: same thing. A machine built to calculate gets a primary-school sum wrong.

This lesson's dumb question is: **why does the computer say `0.1 + 0.2` isn't `0.3`?** It isn't broken. It stores numbers in a way where 0.1 doesn't fit exactly. This lesson explains what that way is, why you almost never notice, and when you do.

Pick your card: ♣ if you're starting from scratch, ♦ if a simple formula doesn't scare you, ♠ if you want the notation. You can switch at any time.



0.1 doesn't fit in binary

♦ → ♠ From intermediate to technical

In ♦ we said	In ♠ we write
binary	base 2; $(\cdot)_2$ marks a number written in base 2
the group 0011 repeats forever	$0.0(0011)_2 = \sum_{k \geq 1} (2^{-4k} + 2^{-4k-1})$, a repeating expansion
It's the same thing that happens to 1/3 in decimal	$p/q \in \mathbb{Q}$ (rational) in lowest terms: finite expansion in base 2 $\Leftrightarrow q = 2^k$
53 binary digits	precision $p = 53$ bits
rounds to the nearest number	$\text{fl}(x)$: round to nearest, ties to even
The sum	$x \oplus y = \text{fl}(x + y)$, floating-point addition
the next one	the neighbour one ulp (unit in the last place) away, here 2^{-54}

$$0.1 = 0.0(0011)_2 = \sum_k (2^{-4k} + 2^{-4k-1})$$

$$p/q \in \mathbb{Q} \text{ finite in base 2} \Leftrightarrow q = 2^k$$

$$\text{fl}(0.1) = 3602879701896397 / 2^{55}$$

$$\text{fl}(0.1) - 0.1 = 1/(5 \cdot 2^{55}) \approx 5.55 \times 10^{-18}$$

$$0.1 \oplus 0.2 = \text{fl}(0.3) + 2^{-54}$$

A fraction p/q in lowest terms has a finite expansion in base 2 only if q is a power of 2. $1/10$ has a 5 in the denominator, so it repeats: $0.1 = 0.0(0011)_2 = \sum_{k \geq 1} (2^{-4k} + 2^{-4k-1})$. The geometric series confirms it: $(3/32) / (15/16) = 1/10$.

With $p = 53$, $\text{fl}(0.1)$ is the nearest fraction $J/2^N$ with a 53-bit J : $J = 7205759403792794$ and $N = 56$, which reduces to $3602879701896397 / 2^{55}$ (Python tutorial). Since $10 \times 3602879701896397 = 36028797018963970$ and $2^{55} = 36028797018963968$:

$$\text{fl}(0.1) - 0.1 = 2 / (10 \cdot 2^{55}) = 1 / (5 \cdot 2^{55}) \approx 5.55 \times 10^{-18}$$

It was rounded up. In $0.1 \oplus 0.2$, the exact sum $\text{fl}(0.1) + \text{fl}(0.2)$ lands exactly halfway between two 53-bit numbers. The ties-to-even rule sends it to the upper one, which sits one ulp, $2^{-54} \approx 5.55 \times 10^{-17}$, above $\text{fl}(0.3)$. That's why $0.1 + 0.2 == 0.3$ is false in Python, in JavaScript and in any language that uses this format (own calculation with exact arithmetic; in JavaScript, $0.1 + 0.2 - 0.3$ gives 2^{-54}).

The King of Dumbs says:

The computer adds what it has just fine. The problem is that what it has isn't 0.1: it doesn't fit.



Floating point: scientific notation in binary

♦ → ♠ From intermediate to technical

In ♦ we said	In ♠ we write
sign	$s \in \{0, 1\}$
exponent	E , 11 bits, with bias 1023: the actual exponent is $E - 1023$
mantissa	$(1.f)_2$, with f the 52 stored bits and the implicit 1
binary64	IEEE 754-2019 format with $p = 53$, $e_{\min} = -1022$ and $e_{\max} = 1023$
the gap between two neighbouring numbers	$\text{ulp}(x) = 2^{(\lfloor \log_2 x \rfloor - 52)}$, with $\lfloor \cdot \rfloor$ the floor
it's rounded to the nearest one	$\text{fl}(x) = x(1 + \delta)$, $ \delta \leq u = 2^{-53}$ (u : unit roundoff)
Just above 1 it's 2.2×10^{-16}	$\epsilon = \text{ulp}(1) = 2^{-52}$ (<code>Number.EPSILON</code> , <code>sys.float_info.epsilon</code>)
infinity and meaningless results	$\pm\infty$ and NaN (not a number), with $E = 2047$; $E = 0$ for zero and subnormals

$$x = (-1)^s \cdot (1.f)_2 \cdot 2^{(E - 1023)}$$

$$\text{ulp}(x) = 2^{(\lfloor \log_2 |x| \rfloor - 52)}$$

$$\text{fl}(x) = x(1 + \delta), \quad |\delta| \leq u = 2^{-53}$$

$$\epsilon = 2^{-52} \approx 2.22 \times 10^{-16}$$

$$2^{53} \approx 9 \times 10^{15}$$

A normal binary64 is $x = (-1)^s \cdot (1.f)_2 \cdot 2^{(E - 1023)}$, with $1 \leq E \leq 2046$. The values $E = 0$ and $E = 2047$ are reserved for zero, subnormals (without the implicit 1), $\pm\infty$ and NaN. The parameters $p = 53$, $\text{emin} = -1022$ and $\text{emax} = 1023$ are those of Goldberg's table D-1 (1991); the current revision of the standard is IEEE 754-2019.

Between 2^e and 2^{e+1} there are 2^{52} equally spaced numbers, $\text{ulp}(x) = 2^{(\lfloor \log_2 |x| \rfloor - 52)}$ apart. With round to nearest, $\text{fl}(x) = x(1 + \delta)$ with $|\delta| \leq u = 2^{-53}$, and every basic operation obeys the same bound: $x \oplus y = (x + y)(1 + \delta)$ (Goldberg).

Watch the names. MDN, Python and PyTorch call $\epsilon = 2^{-52} = \text{ulp}(1)$, the distance from 1 to the next number, epsilon. Goldberg calls the bound on the relative error *machine epsilon*, which is half that: $u = 2^{-53}$. Integers are exact up to $2^{53} = 9,007,199,254,740,992$; from there on, $\text{ulp} \geq 2$ (MDN, `Number.MAX_SAFE_INTEGER = 253 - 1`).



Rounding adds up

♦ → ♠ From intermediate to technical

In ♦ we said	In ♠ we write
Every operation rounds	$x \oplus y = \text{fl}(x + y)$, and $x \ominus y = \text{fl}(x - y)$
the errors pile up	recursive summation $s \leftarrow s \oplus x_j$: term j is perturbed by up to $(n - j)u$
the order changes the result	$\oplus$ is commutative but not associative
Compare with a tolerance	$ a - b \leq \max(\text{rel_tol} \cdot \max(a , b), \text{abs_tol})$, with $\text{rel_tol} = 10^{-9}$ (<code>math.isclose</code>)
Add carefully	Kahan compensated summation, with c the error each step carries
compensates for the errors internally	Neumaier summation, a variant of Kahan's (gh-100425)
rounds only once	a single rounding at the end (<code>math.fsum</code>); $O(nu^2)$: terms of order nu^2 or smaller

$$s \leftarrow s \oplus x_j : |\delta_j| < (n - j)u \quad (\text{naïve})$$

$$\begin{aligned} \text{Kahan: } y &= x_j \ominus c; \quad t = s \oplus y \\ c &= (t \ominus s) \ominus y; \quad s = t \end{aligned}$$

$$|\text{error}| \leq (2u + O(nu^2)) \sum |x_j|$$

$$(10^{30} \oplus -10^{30}) \oplus 1 = 1$$

$$10^{30} \oplus (-10^{30} \oplus 1) = 0$$

In recursive summation, each term x_j comes in with a factor $(1 + \delta_j)$, with $|\delta_j| < (n - j)u$: the error grows with n (Goldberg, 1991, who calls $u = 2^{-53}$ epsilon, as seen in the previous block). Kahan's compensated summation stores in c what each $\oplus$ lost and gives it back in the next step. Its error stays at $(2u + O(nu^2)) \sum |x_j|$, almost independent of n (Goldberg, theorem 8; Kahan, *CACM* 8, 40, 1965).

Without associativity, order matters: $(10^{30} \oplus -10^{30}) \oplus 1 = 1$, but $10^{30} \oplus (-10^{30} \oplus 1) = 0$ (Goldberg). One consequence: splitting a long sum across several processors changes the order, and the result may vary in the last digits.

To compare, `math.isclose(a, b)` checks $|a - b| \leq \max(\text{rel_tol} \cdot \max(|a|, |b|), \text{abs_tol})$, with `rel_tol = 10-9` by default: about 9 digits of agreement (Python documentation). `sum()` has used Neumaier summation since Python 3.12 (gh-100425), and `math.fsum` rounds only once at the end.

The King of Dumbs says:

Never ask a number with decimals whether it equals another. Ask whether it's very close.



Fewer bits for AI, decimals for money

♦ → ♠ From intermediate to technical

In ♦ we said	In ♠ we write
about 7 digits	$p \cdot \log_{10} 2$ decimal digits: 7.22 with $p = 24$
nothing above 65,504	largest finite value $(2 - 2^{1-p}) \cdot 2^{\text{emax}}$, with <code>emax</code> the largest exponent
the same range as float32	the same 8-bit exponent: <code>emax = 127</code>
mixed precision	products in FP16 or BF16 (float16, bfloat16); master weights and accumulation in FP32 (float32)
the network's weights	weights <code>w</code> ; gradient: how much the loss (the network's error) changes when each weight moves
Google	TPU (Tensor Processing Unit): Google's chip for neural networks
a small change can get lost	$w \oplus \Delta w = w$ if $ \Delta w < \text{ulp}(w)/2$
2048 times larger	$2^{11} = 2^p$ with $p = 11$
decimal arithmetic	floating point in base $\beta = 10$ (Python's decimal module)

	p	ϵ	max
binary64	53	2.2×10^{-16}	1.8×10^{308}
binary32	24	1.2×10^{-7}	3.4×10^{38}
binary16	11	9.8×10^{-4}	65504
bfloat16	8	7.8×10^{-3}	3.4×10^{38}

$$|\Delta w| < \text{ulp}(w)/2 \Rightarrow w \oplus \Delta w = w$$

With p bits of precision and largest exponent $\epsilon_{\max}$, $\epsilon = 2^{1-p}$ and the largest finite value is $(2 - 2^{1-p}) \cdot 2^{\epsilon_{\max}}$. The bit layouts come from NumPy, PyTorch and Google; the values in the table are my own calculation. For binary16, $(2 - 2^{-10}) \cdot 2^{15} = 65,504$, the same value Micikevicius et al. (2018) give.

$w \oplus \Delta w = w$ as soon as $|\Delta w| < \text{ulp}(w)/2$, and in FP16 that can happen from $|w| \geq 2^{11}|\Delta w|$ on (Micikevicius et al.). Their mixed-precision recipe keeps master weights in FP32, multiplies in FP16 and scales the loss so that small gradients don't drop to zero: in FP16 there is no positive number below 2^{-24} , and for their object detector (SSD) a factor of 8 was enough. On Google's TPUs, each matrix multiplication multiplies in bfloat16 and accumulates in FP32 (Google Cloud, 2019). The widget shows it with 0.1: in float16, from 256 on the ulp is 0.25 and $0.1 < 0.125$.

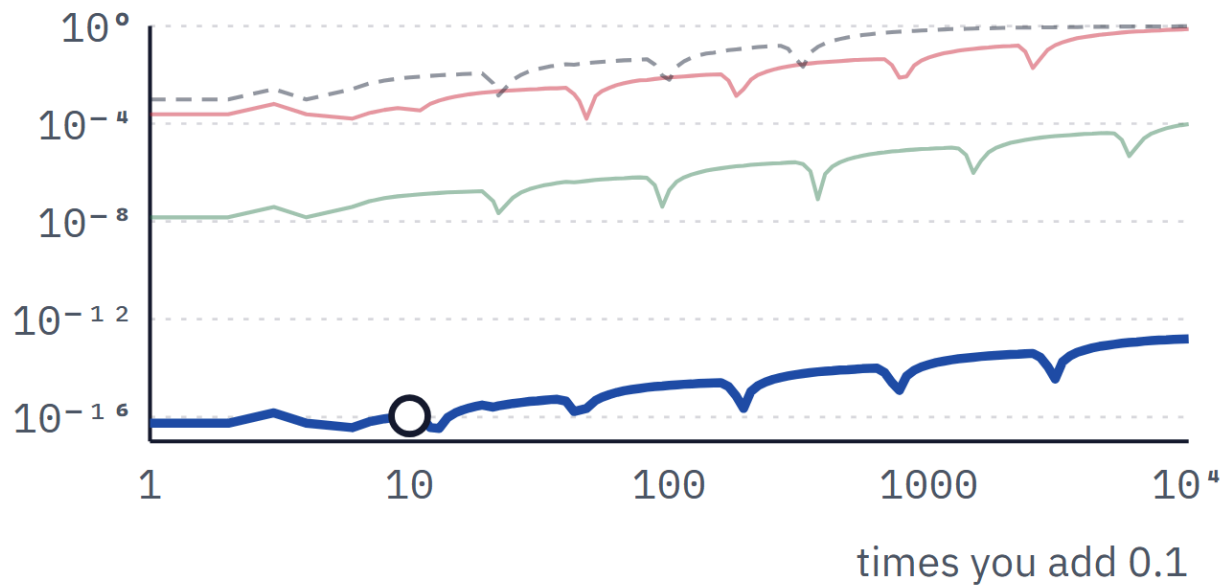
Python's decimal module is floating point in base 10: 0.1 is stored exactly and $0.1 + 0.1 + 0.1 = 0.3$ gives 0. For money, whole numbers of cents also work, exact in binary64 up to 2^{53} .



Try it: add 0.1 over and over

Choose how the computer stores numbers and how many times it adds 0.1. Compare what it gives with what it should give, and watch the error grow on the chart. Try float16 with 5000 times.

relative error



One curve per format; the chosen one is bold.

How it stores numbers

- ☒ **binary64**, the usual (64 bits)
- ☐ float32 (32 bits)
- ☐ float16 (16 bits)
- ☐ bfloat16 (16 bits, the AI one)

Times you add 0.1 10 times

Should give 1

Gives **0.9999999999999999**

Error 1.1×10^{-16}

How 0.1 is stored 0.1000000000000000055511151231257827021181583404541015625

Off by 1.1×10^{-16} : one part in 9×10^{15} .



Interactive widget. Try it on the web: <https://kingofdumbs.com/en/lessons/floating-point>



Final challenge

1. Why can't the computer store 0.1 exactly?

- a. Because it's too small a number
- b. Because in binary 0.1 has infinitely many digits, like 1/3 in decimal
- c. Because the programs are badly written

2. You add 0.1 ten times in a loop and check whether the result equals 1. What happens?

- a. Yes, it equals 1
- b. No: it gives 0.9999999999999999, so you need to compare with a tolerance
- c. The program throws an error

3. To store a bank account balance, what's best?

- a. float16, to save memory
- b. Counting cents with whole numbers, or using decimal arithmetic
- c. binary64, which never rounds



In three sentences

1. The computer counts in binary, and in binary 0.1 has no end, like 1/3 in decimal: it's stored rounded, a hair too high.
2. Floating point is scientific notation in binary with a fixed number of digits (about 16 in binary64): every operation rounds, the error adds up and the order of the sums matters.
3. That's why numbers with decimals are compared with a tolerance; AI uses 16-bit formats because they save memory, and money goes in whole cents or in decimal.



Sources

- Why 0.1 isn't exact, its stored value, `math.isclose` and `math.fsum`: Python 3.15, [tutorial, chapter 15](#) and [math module](#). `sum()` with Neumaier summation: [What's new in Python 3.12](#). Decimal arithmetic: [decimal module](#).
- Representation, rounding, non-associativity and Kahan summation: D. Goldberg, "What every computer scientist should know about floating-point arithmetic", *ACM Computing Surveys* 23(1), 5–48 (1991), [doi:10.1145/103162.103163](#), [Oracle reprint](#); W. Kahan, *Communications of the ACM* 8(1), 40 (1965), [doi:10.1145/363707.363723](#). Standard: [IEEE 754-2019](#).
- Formats and bits: [MDN, Number](#); [NumPy 2.5, data types](#); [PyTorch 2.14, torch.dtype](#); [Excel and floating point \(Microsoft\)](#).
- bfloat16: [Google Cloud blog \(2019\)](#) and [Cloud TPU documentation](#). Mixed precision: P. Micikevicius et al., "Mixed Precision Training", ICLR 2018, [arXiv:1710.03740](#).

How this lesson was made: researched and written with Claude's help from the sources above, checked and cross-checked against a second source on 15 October 2026. The results of adding 0.1 in a loop, in each format, were measured in JavaScript (Node 24) and a test checks them; figures without their own link are calculations from the sources. The drawings are original.

Answers

1. **b)** Because in binary 0.1 has infinitely many digits, like $1/3$ in decimal. That's it. It counts in halves, and a tenth never ends.
2. **b)** No: it gives 0.9999999999999999, so you need to compare with a tolerance. Good. Every sum rounds and the error adds up.
3. **b)** Counting cents with whole numbers, or using decimal arithmetic. Exactly. With money, you don't round in binary.

A King of Dumbs lesson, kingofdumbs.com. Offline version; the original, with widgets you can play with, is at: <https://kingofdumbs.com/en/lessons/floating-point>

Updated on October 16, 2026.