

King of Dumbs

Level: ♦ Intermediate

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

each box is worth

$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$	$\frac{1}{128}$	$\frac{1}{256}$	$\frac{1}{512}$
0	0	0	1	1	0	0	1	1

nine boxes: $\frac{1}{16} + \frac{1}{32} + \frac{1}{256} + \frac{1}{512} = 0.099609375$

$0.1 = 0.0001100110011\dots_2$ (0011 repeats)

The computer stores numbers in **binary**, using only zeros and ones. In decimal, each digit after the point is worth a tenth of the one before: tenths, hundredths... In binary, each digit is worth **half** the one before: $\frac{1}{2}$, $\frac{1}{4}$, $\frac{1}{8}$, $\frac{1}{16}$...

To write 0.1 in binary you check which pieces fit: $\frac{1}{2}$ no, $\frac{1}{4}$ no, $\frac{1}{8}$ no, $\frac{1}{16}$ yes... You get 0.0001100110011..., and **the group 0011 repeats forever**. With nine digits you reach 0.099609375; with more you get closer, but you never finish. **It's the same thing that happens to $\frac{1}{3}$ in decimal.**

An ordinary computer keeps **53 binary digits** and **rounds to the nearest number** it can write with them. Its 0.1 is exactly 0.1000000000000000055511151231257827021181583404541015625, a little more than 0.1 (Python tutorial).

The same happens with 0.2 and 0.3, each with its own small error. **The sum** of the first two doesn't land on the number the computer uses for 0.3, but on **the next one**: 0.300000000000000004.

The King of Dumbs says:

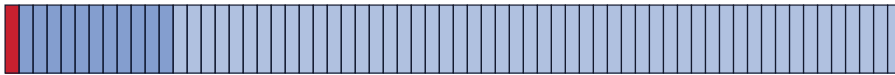
The computer adds what it has just fine. The problem is that what it has isn't 0.1: it doesn't fit.

Floating point: scientific notation in binary

sign

exponent (11)

mantissa (52)



+ 2^{-4}

1.6

$0.1 = + 1.6 \times 2^{-4}$ · about 16 decimal digits

A **floating-point** number is stored in three parts, like scientific notation but in base 2:

- the **sign**, + or −;
- the **exponent**, which says what power of 2 it's multiplied by;
- the **mantissa**, which holds the digits.

The usual format is called **binary64** (or “double precision”): 64 bits, 1 for the sign, 11 for the exponent and 52 for the mantissa (MDN; NumPy). The mantissa always starts with a 1 that doesn't need storing, so it has 53 binary digits, about 16 decimal ones. It's the format of JavaScript numbers, Python floats and Excel. For example, $0.1 = + 1.6 \times 2^{-4}$.

Since there are always the same number of digits, **the gap between two neighbouring numbers** grows with size. **Just above 1 it's 2.2×10^{-16}** . Near 10^{16} it's 2, which is why $10^{16} + 1$ gives 10^{16} . For the same reason, whole numbers are only exact up to about 9×10^{15} (MDN).

Some bit patterns are reserved for zero, **infinity and meaningless results**, such as the square root of −1 (Microsoft, on Excel).

When the result of an operation falls between two numbers that can be stored, **it's rounded to the nearest one**. Each rounding is off by at most half a gap: less than one part in 9×10^{15} .



Rounding adds up

0.1 added n times

1	0.1	6	0.6
2	0.2	7	0.7
3	0.30000000000000004	8	0.7999999999999999
4	0.4	9	0.8999999999999999
5	0.5	10	0.9999999999999999

$10^{16} + 1 = 10^{16}$: the 1 doesn't fit

Every operation rounds, and the errors pile up. Adding 0.1 in a loop, in JavaScript:

Times	Result
3	0.30000000000000004
8	0.7999999999999999
10	0.9999999999999999
10,000	1000.0000000001588

On top of that, **the order changes the result**. Adding a small number to a very large one may change nothing ($10^{16} + 1 = 10^{16}$), so $(a + b) + c$ doesn't always give the same as $a + (b + c)$ (Goldberg, 1991).

Three common remedies:

- **Compare with a tolerance** instead of with `==`. In Python, `math.isclose(0.1 + 0.1 + 0.1, 0.3)` gives `True` (Python tutorial).
- **Add carefully**. `math.fsum` keeps track of the digits that get lost and **rounds only once**. And since Python 3.12, `sum()` **compensates for the errors internally**: `sum([0.1] * 10) == 1.0` gives `True` on almost every install, while the hand-written loop gives `0.9999999999999999`.
- **Round when displaying**, not when calculating: two decimals for a price.

The King of Dumbs says:

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

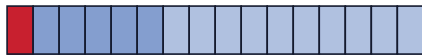


Fewer bits for AI, decimals for money

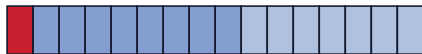
float32 ≈ 7 digits · up to 3.4×10^{38}



float16 ≈ 3 digits · up to 65,504



bfloat16 ≈ 2 digits · up to 3.4×10^{38}



Besides binary64 there are smaller formats:

- **float32**, 32 bits (1 + 8 + 23): **about 7 digits**.
- **float16**, 16 bits (1 + 5 + 10): about 3 digits, and **nothing above 65,504**.
- **bfloat16**, 16 bits (1 + 8 + 7): only about 2 digits, but **the same range as float32**. It was designed at Google for machine learning (Google Cloud blog, 2019).

Neural networks use these small formats a lot: they take half the memory and can go faster. The trick is called **mixed precision**: do the multiplications in 16 bits and keep a copy of the network's weights in 32. In float16, **a small change can get lost** if the number it's added to is **2048 times larger** or more (Micikevicius et al., 2018). It's the truck scale again.

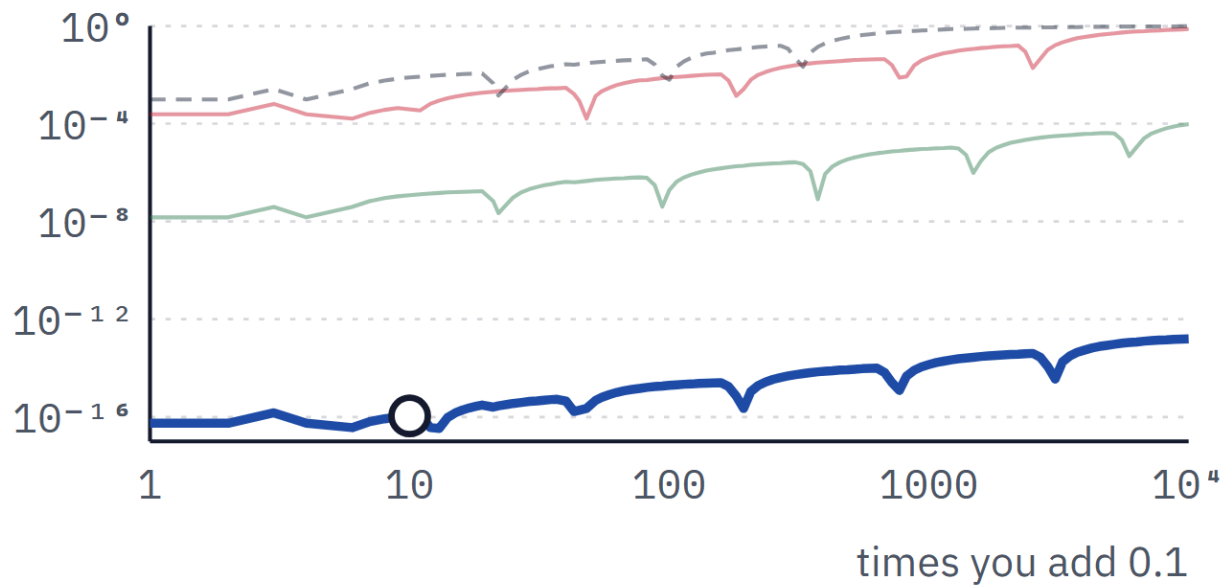
For money, you do the opposite: **decimal arithmetic**, which counts in tenths and hundredths like we do, or whole numbers of cents. In decimal, $0.1 + 0.1 + 0.1 - 0.3$ gives exactly 0; in binary, 5.55×10^{-17} (documentation of Python's decimal module, designed for accounting).



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.