

King of Dumbs

el rey de los tontos

Nivel: ♠ Técnico

¿Por qué el ordenador dice que $0,1 + 0,2$ no es $0,3$?

$0,1$ no cabe en binario y el ordenador lo redondea. Qué es la coma flotante, por qué el error se acumula y qué usan la IA y el dinero. En tres niveles y con widget.

Español Publicado 15 de octubre de 2026 Actualizado 16 de octubre de 2026



Escanea para abrir la lección interactiva

<https://kingofdumbs.com/es/lecciones/coma-flotante>

Qué vas a aprender

- Por qué 0,1 no se puede guardar exacto en binario
- La coma flotante: notación científica en binario con cifras fijas
- Cómo se acumula el redondeo y cómo se compara con tolerancia
- float32, float16 y bfloat16 en la IA, y decimales para el dinero

Antes de esta lección

Si algo te suena a chino, estas lecciones lo explican desde cero:

1. [¿Cuánto es \$7 \times 10^{27}\$? Números enormes sin miedo](#)

Símbolos que aparecen

Δ incremento: cuánto ha cambiado (♠) · E_a subíndice: la etiqueta de abajo (♠) · Σ sumatorio: súmalos todos (♠) · x_i índice i: el elemento número i (♠) · $\approx$ aproximadamente igual (♠) · $\leq \geq$ menor o igual / mayor o igual (♠) · $|x|$ valor absoluto: sin signo (♠) · 10^{-3} exponente negativo: dividir (♦♠) · $\log$ logaritmo: ¿cuántas veces multiplico? (♠) · $\lfloor x \rfloor$ parte entera: redondear hacia abajo (♠) · ϵ épsilon: algo muy pequeño (♠) · δ delta minúscula: un error pequeño (♠) · $\in$ pertenece a (♠)

Abre el navegador en el ordenador, pulsa F12 para abrir la consola, escribe `0.1 + 0.2` y pulsa Enter. La respuesta es `0.30000000000000004`. Prueba en Python: lo mismo. Una máquina hecha para calcular falla una suma de primaria.

La pregunta tonta de esta lección es: **¿por qué el ordenador dice que `0,1 + 0,2` no es `0,3`?** No está roto. Guarda los números de una manera en la que 0,1 no cabe exacto. Esta lección explica cuál es esa manera, por qué casi nunca se nota y cuándo sí.

Elige tu carta: ♣ si empiezas de cero, ♦ si no te asusta una fórmula sencilla, ♠ si quieres la notación. Puedes cambiarla en cualquier momento.



0,1 no cabe en binario

♦ → ♠ Del intermedio al técnico

En ♦ decíamos	En ♠ se escribe
binario	base 2; $(\cdot)_2$ marca un número escrito en base 2
el grupo 0011 se repite para siempre	$0,0(0011)_2 = \sum_{k \geq 1} (2^{-4k} + 2^{-4k-1})$, desarrollo periódico
Le pasa lo mismo que a 1/3 en decimal	$p/q \in \mathbb{Q}$ (racional) irreducible: desarrollo finito en base 2 $\Leftrightarrow q = 2^k$
53 cifras binarias	precisión $p = 53$ bits
redondea al número más cercano	$\text{fl}(x)$: redondeo al más cercano, con los empates al par
La suma	$x \oplus y = \text{fl}(x + y)$, la suma en coma flotante
el siguiente	el vecino a un ulp (unit in the last place), aquí 2^{-54}

$$0,1 = 0,0(0011)_2 = \sum_k (2^{-4k} + 2^{-4k-1})$$

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

$$\text{fl}(0,1) = 3602879701896397 / 2^{55}$$

$$\text{fl}(0,1) - 0,1 = 1/(5 \cdot 2^{55}) \approx 5,55 \times 10^{-18}$$

$$0,1 \oplus 0,2 = \text{fl}(0,3) + 2^{-54}$$

Una fracción irreducible p/q tiene desarrollo finito en base 2 solo si q es potencia de 2. $1/10$ lleva un 5 en el denominador y queda periódico: $0,1 = 0,0(0011)_2 = \sum_{k \geq 1} (2^{-4k} + 2^{-4k-1})$. La serie geométrica lo confirma: $(3/32) / (15/16) = 1/10$.

Con $p = 53$, $\text{fl}(0,1)$ es la fracción $J/2^N$ más cercana con J de 53 bits: $J = 7205759403792794$ y $N = 56$, que simplificada queda $3602879701896397 / 2^{55}$ (tutorial de Python). Como $10 \times 3602879701896397 = 36028797018963970$ y $2^{55} = 36028797018963968$:

$$\text{fl}(0,1) - 0,1 = 2 / (10 \cdot 2^{55}) = 1 / (5 \cdot 2^{55}) \approx 5,55 \times 10^{-18}$$

Se redondeó hacia arriba. En $0,1 \oplus 0,2$, la suma exacta $\text{fl}(0,1) + \text{fl}(0,2)$ cae justo a medio camino entre dos números de 53 bits. La regla de empates al par la manda al de arriba, que está un ulp, $2^{-54} \approx 5,55 \times 10^{-17}$, por encima de $\text{fl}(0,3)$. Por eso $0.1 + 0.2 == 0.3$ es falso en Python, en JavaScript y en cualquier lenguaje que use este formato (cálculo propio con aritmética exacta; en JavaScript, $0.1 + 0.2 - 0.3$ da 2^{-54}).

El Rey de los Tontos dice:

El ordenador suma bien lo que tiene. El problema es que lo que tiene no es 0,1: no le cabe.



Coma flotante: notación científica en binario

♦ → ♠ Del intermedio al técnico

En ♦ decíamos	En ♠ se escribe
signo	$s \in \{0, 1\}$
exponente	E, de 11 bits, con sesgo 1023: el exponente real es $E - 1023$
mantisa	$(1.f)_2$, con f los 52 bits guardados y el 1 implícito
binary64	formato de IEEE 754-2019 con $p = 53$, $e_{\min} = -1022$ y $e_{\max} = 1023$
la separación entre dos números seguidos	$\text{ulp}(x) = 2^{(\lfloor \log_2 x \rfloor - 52)}$, con $\lfloor \cdot \rfloor$ la parte entera por abajo
se redondea al más cercano	$\text{fl}(x) = x(1 + \delta)$, $ \delta \leq u = 2^{-53}$ (u: unidad de redondeo)
Justo por encima de 1 es de $2,2 \times 10^{-16}$	$\epsilon = \text{ulp}(1) = 2^{-52}$ (Number.EPSILON, sys.float_info.epsilon)
el infinito y los resultados sin sentido	$\pm\infty$ y NaN (not a number), con E = 2047; E = 0, para el cero y los subnormales

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

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

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

$$\varepsilon = 2^{-52} \approx 2,22 \times 10^{-16}$$

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

Un binary64 normal vale $x = (-1)^s \cdot (1.f)_2 \cdot 2^{(E - 1023)}$, con $1 \leq E \leq 2046$. Los valores $E = 0$ y $E = 2047$ se reservan para el cero, los subnormales (sin el 1 implícito), $\pm\infty$ y NaN. Los parámetros $p = 53$, $e_{\min} = -1022$ y $e_{\max} = 1023$ son los de la tabla D-1 de Goldberg (1991); la revisión actual del estándar es IEEE 754-2019.

Entre 2^e y 2^{e+1} hay 2^{52} números igualmente espaciados, separados $\text{ulp}(x) = 2^{(\lfloor \log_2 |x| \rfloor - 52)}$. Con redondeo al más cercano, $\text{fl}(x) = x(1 + \delta)$ con $|\delta| \leq u = 2^{-53}$, y cada operación básica cumple lo mismo: $x \oplus y = (x + y)(1 + \delta)$ (Goldberg).

Cuidado con los nombres. MDN, Python y PyTorch llaman épsilon a $\varepsilon = 2^{-52} = \text{ulp}(1)$, la distancia de 1 al siguiente número. Goldberg llama *machine epsilon* a la cota del error relativo, que es la mitad: $u = 2^{-53}$. Los enteros son exactos hasta $2^{53} = 9\,007\,199\,254\,740\,992$; desde ahí, $\text{ulp} \geq 2$ (MDN, `Number.MAX_SAFE_INTEGER = 253 - 1`).



El redondeo se acumula

♦ → ♠ Del intermedio al técnico

En ♦ decíamos	En ♠ se escribe
Cada operación redondea	$x \oplus y = \text{fl}(x + y)$, $y \ominus x = \text{fl}(y - x)$
los errores se van acumulando	suma recursiva $s \leftarrow s \oplus x_j$: el sumando j se perturba hasta $(n - j)u$
el orden cambia el resultado	$\oplus$ es conmutativa pero no asociativa
Comparar con tolerancia	$ a - b \leq \max(\text{rel_tol} \cdot \max(a , b), \text{abs_tol})$, con $\text{rel_tol} = 10^{-9}$ (<code>math.isclose</code>)
Sumar con cuidado	suma compensada de Kahan, con c el error que arrastra cada paso
compensa los errores por dentro	suma de Neumaier, una variante de la de Kahan (gh-100425)
redondea una sola vez	un único redondeo al final (<code>math.fsum</code>); $O(nu^2)$: términos del orden de nu^2 o menores

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

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

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

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

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

En la suma recursiva, cada sumando x_j entra con un factor $(1 + \delta_j)$, con $|\delta_j| < (n - j)u$: el error crece con n (Goldberg, 1991, que llama ϵ a $u = 2^{-53}$, como se vio en el bloque anterior). La suma compensada de Kahan guarda en c lo que ha perdido cada $\oplus$ y lo devuelve en el paso siguiente. Su error queda en $(2u + O(nu^2)) \sum |x_j|$, casi sin depender de n (Goldberg, teorema 8; Kahan, *CACM* 8, 40, 1965).

Sin asociatividad, el orden importa: $(10^{30} \oplus -10^{30}) \oplus 1 = 1$, pero $10^{30} \oplus (-10^{30} \oplus 1) = 0$ (Goldberg). Una consecuencia: repartir una suma larga entre varios procesadores cambia el orden, y el resultado puede variar en las últimas cifras.

Para comparar, `math.isclose(a, b)` comprueba $|a - b| \leq \max(\text{rel_tol} \cdot \max(|a|, |b|), \text{abs_tol})$, con `rel_tol = 10-9` por defecto: unas 9 cifras de acuerdo (documentación de Python). `sum()` usa la suma de Neumaier desde Python 3.12 (gh-100425) y `math.fsum` hace un solo redondeo al final.

El Rey de los Tontos dice:

Nunca preguntes a un número con decimales si es igual a otro. Pregúntale si está muy cerca.



Menos bits para la IA, decimales para el dinero

♦ → ♠ Del intermedio al técnico

En ♦ decíamos	En ♠ se escribe
unas 7 cifras	<code>p · log₁₀2</code> cifras decimales: 7,22 con <code>p = 24</code>
nada por encima de 65 504	máximo finito $(2 - 2^{1-p}) \cdot 2^{\text{emax}}$, con <code>emax</code> el mayor exponente
el mismo rango que float32	el mismo exponente de 8 bits: <code>emax = 127</code>
precisión mixta	<code>mixed precision</code> : productos en FP16 o BF16 (<code>float16</code> , <code>bfloat16</code>); pesos maestros y acumulación en FP32 (<code>float32</code>)
los pesos de la red	pesos <code>w</code> ; gradiente: cuánto cambia la pérdida (el error de la red) al mover cada peso
Google	TPU (Tensor Processing Unit): el chip de Google para redes neuronales
un cambio pequeño puede perderse	<code>w ⊕ Δw = w</code> si <code> Δw < ulp(w)/2</code>
2048 veces mayor	$2^{11} = 2^p$ con <code>p = 11</code>
aritmética decimal	coma flotante en base $\beta = 10$ (módulo decimal de Python)

	p	ϵ	max
binary64	53	$2,2 \times 10^{-16}$	$1,8 \times 10^{308}$
binary32	24	$1,2 \times 10^{-7}$	$3,4 \times 10^{38}$
binary16	11	$9,8 \times 10^{-4}$	65504
bfloat16	8	$7,8 \times 10^{-3}$	$3,4 \times 10^{38}$

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

Con p bits de precisión y exponente máximo $e_{\max}$, $\epsilon = 2^{1-p}$ y el máximo finito es $(2 - 2^{1-p}) \cdot 2^{e_{\max}}$. Los bits salen de NumPy, PyTorch y Google; los valores de la tabla son cálculo propio. Para binary16, $(2 - 2^{-10}) \cdot 2^{15} = 65\,504$, el mismo valor que dan Micikevicius et al. (2018).

$w \oplus \Delta w = w$ en cuanto $|\Delta w| < \text{ulp}(w)/2$, y en FP16 eso puede pasar desde $|w| \geq 2^{11}|\Delta w|$ (Micikevicius et al.). Su receta de precisión mixta guarda los pesos maestros en FP32, multiplica en FP16 y escala la pérdida para que los gradientes pequeños no caigan a cero: en FP16 no hay ningún número positivo por debajo de 2^{-24} , y en su detector de objetos (SSD) bastó con un factor 8. En las TPU de Google, cada multiplicación de matrices multiplica en bfloat16 y acumula en FP32 (Google Cloud, 2019). El widget lo enseña con 0,1: en float16, desde 256 el ulp es 0,25 y $0,1 < 0,125$.

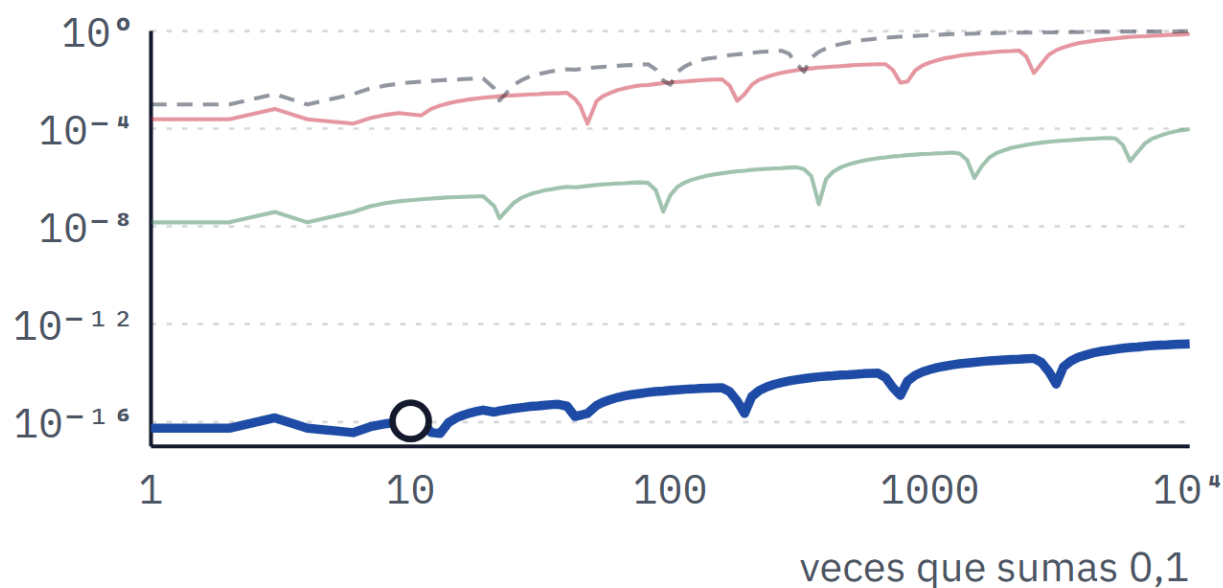
El módulo decimal de Python es coma flotante en base 10: 0,1 se guarda exacto y $0,1 + 0,1 + 0,1 - 0,3$ da 0. Para dinero también valen los enteros de céntimos, exactos en binary64 hasta 2^{53} .



Prueba tú: suma 0,1 una y otra vez

Elige cómo guarda los números el ordenador y cuántas veces suma 0,1. Compara lo que da con lo que debería dar y mira cómo crece el error en la gráfica. Prueba float16 con 5000 veces.

error relativo



Una curva por formato; la del elegido, en grueso.

Cómo guarda los números

- ☒ binary64, lo normal (64 bits)
- ☐ float32 (32 bits)
- ☐ float16 (16 bits)
- ☐ bfloat16 (16 bits, el de la IA)

Veces que sumas 0,1 10 veces

Debería dar 1

Da **0,9999999999999999**

Error **$1,1 \times 10^{-16}$**

Así se guarda 0,1 0,1000000000000000055511151231257827021181583404541015625

Se desvía $1,1 \times 10^{-16}$: una parte entre 9×10^{15} .



Widget interactivo. Pruébalo en la web: <https://kingofdumbs.com/es/lecciones/coma-flotante>



Reto final

1. ¿Por qué el ordenador no puede guardar 0,1 exacto?

- a. Porque es un número demasiado pequeño
- b. Porque en binario 0,1 tiene infinitas cifras, como 1/3 en decimal
- c. Porque los programas están mal hechos

2. Sumas 0,1 diez veces en un bucle y compruebas si el resultado es igual a 1. ¿Qué pasa?

- a. Sí, es igual a 1
- b. No: da 0,9999999999999999, así que hay que comparar con tolerancia
- c. El programa da un error

3. Para guardar el saldo de una cuenta bancaria, ¿qué es mejor?

- a. float16, para ahorrar memoria
- b. Contar céntimos con números enteros, o usar aritmética decimal
- c. binary64, que nunca redondea



En tres frases

1. El ordenador cuenta en binario, y en binario 0,1 no tiene final, como 1/3 en decimal: se guarda redondeado, un pelín por encima.
2. La coma flotante es notación científica en binario con cifras fijas (unas 16 en binary64): cada operación redondea, el error se acumula y el orden de las sumas importa.
3. Por eso los números con decimales se comparan con tolerancia; la IA usa formatos de 16 bits porque ahorran memoria, y el dinero va en céntimos enteros o en decimal.



Fuentes

- Por qué 0,1 no es exacto, su valor guardado, `math.isclose` y `math.fsum`: Python 3.15, [tutorial, capítulo 15](#) y [módulo math](#). `sum()` con suma de Neumaier: [novedades de Python 3.12](#). Aritmética decimal: [módulo decimal](#).
- Representación, redondeo, no asociatividad y suma de Kahan: 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](#), [reimpresión de Oracle](#); W. Kahan, *Communications of the ACM* 8(1), 40 (1965), [doi:10.1145/363707.363723](#). Estándar: [IEEE 754-2019](#).
- Formatos y bits: [MDN, Number](#); [NumPy 2.5, tipos de datos](#); [PyTorch 2.14, torch.dtype](#); [Excel y la coma flotante \(Microsoft\)](#).
- bfloat16: [blog de Google Cloud \(2019\)](#) y [documentación de Cloud TPU](#). Precisión mixta: P. Micikevicius et al., “Mixed Precision Training”, ICLR 2018, [arXiv:1710.03740](#).

Cómo está hecha esta lección: investigada y redactada con ayuda de Claude a partir de las fuentes de arriba, comprobadas y contrastadas con una segunda fuente el 15 de octubre de 2026. Los resultados de sumar 0,1 en un bucle, en cada formato, se midieron en JavaScript (Node 24) y los comprueba un test; las cifras sin enlace propio son cálculos a partir de las fuentes. Los dibujos son propios.

Soluciones

1. **b)** Porque en binario $0,1$ tiene infinitas cifras, como $1/3$ en decimal. Eso es. Cuenta en mitades, y una décima no se acaba nunca.
2. **b)** No: da $0,9999999999999999$, así que hay que comparar con tolerancia. Bien. Cada suma redondea y el error se acumula.
3. **b)** Contar céntimos con números enteros, o usar aritmética decimal. Exacto. Con el dinero no se redondea en binario.

Lección de King of Dumbs, kingofdumbs.com. Versión para leer sin conexión; la original, con widgets que se tocan, está en: <https://kingofdumbs.com/es/lecciones/coma-flotante>

Actualizada el 16 de octubre de 2026.