

¿Por qué obtengo resultados inesperados con los float? 🤔 ¿Cómo lidiar con eso? 🦷

 josuegranados1219
7985

Seguro conocerás los números de punto flotante ¿Verdad? pero, ¿Me puedes decir que retornará la siguiente expresión en la consola de Python?

```
>>> (0.1 + 0.1 + 0.1) == 0.3
```

Tal vez no lo parezca, pero esa expresión devuelve False



```
josue@MagnusGates73: ~  
>>> (0.1 + 0.1 + 0.1) == 0.3  
False  
>>> 
```

¿Porque pasa exactamente esto?

Aquí te explico porqué y como lidiar con esto 👤

Sistemas de numeración decimal y binario

Para empezar entendamos como es que representamos los números.

Un sistema de numeración es un conjunto de reglas para expresar cantidades válidas. Ambos el sistema Decimal y Binario cuentan con un número determinado de dígitos, a eso se le llama base.

Sistema Decimal

Base 10 (0,1,2,3,4,5,6,7,8,9)

Sistema Binario

Base 2 (0,1)

Los dos son sistemas **posicionales**, eso quiere decir que el valor de un dígito en una cantidad expresada depende de la posición en la que se encuentre.

Para expresar números enteros en ambos sistemas esas posiciones comienzan desde la derecha y van hacia la izquierda desde la posición 0 y pueden terminar en la posición infinito.

La manera de calcular el valor del dígito en un número expresado es la siguiente:

$\text{dígito} \times (\text{base}^{\text{posición}})$

y el valor total de la cantidad expresada es igual a la suma del valor de cada dígito.

Veamos un ejemplo en ambos sistemas.

$$\begin{array}{r} 1 \quad 2 \quad 5 \\ \hline 100 \quad 10 \quad 1 \end{array}$$

$$\begin{array}{r} 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 0 \quad 1 \\ \hline 64 \quad 32 \quad 16 \quad 8 \quad 4 \quad 2 \quad 1 \end{array}$$

Los 2 sistemas representan la cantidad **125**.

Los números por encima de la línea son los dígitos y los números debajo de la línea son las potencias de su respectiva base. Quizá te llame la atención el **1** que aparece al inicio pero recuerda que comenzamos desde la posición 0 y cualquier número a la 0 es **1**

$$10^0 = 1$$

$$2^0 = 1$$

Ahora calculemos el valor total en cada sistema:

$$(1 \times 100 + 2 \times 10 + 5 \times 1) = 125$$

$$(1 \times 64 + 1 \times 32 + 1 \times 16 + 1 \times 8 + 1 \times 4 + 0 \times 2 + 1 \times 1) = 125$$

Números decimales

Todo bien con los números enteros, pero ¿cómo expresamos los números que no lo son?. De la misma manera que en los números enteros, los decimales se expresan con dígitos en diferentes posiciones pero esta vez van de izquierda a derecha, además no tenemos ese **1** al principio dado que está en el lado de los enteros.

Imagina la recta de números positivos y negativos, así es como la notación científica se usa para expresar números muy grandes o muy pequeños:

$$23 \times 10^{46} \text{ (un número bastante grande)}$$

ó

$$3.8 \times 10^{-15} \text{ (un número muy pequeño)}$$

Veamos un ejemplo de cómo expresaríamos un decimal en ambos sistemas:

$$\begin{array}{r} .1 \quad 2 \quad 5 \\ \hline 10 \quad 100 \quad 1000 \end{array}$$

$$\begin{array}{r} .0 \quad 0 \quad 1 \\ \hline 2 \quad 4 \quad 8 \end{array}$$



De nuevo tenemos los dígitos en la parte superior de la línea y las potencias de la base debajo. En este caso el valor del dígito se calcula **dividiendo**, tiene sentido ¿no?, la operación inversa a la que utilizamos con los enteros ¿cierto?.

Bueno lo que en realidad pasa es esto:

1×10^{-1} es equivalente a $1 \times (1/10^1)$ que a su vez es equivalente a $(1/1) \times (1/10^1)$

y tu ya sabes multiplicar fracciones, ¿verdad?

Entonces para efectos prácticos solo dividamos el dígito entre la potencia de la base:

Sistema Decimal

$$1/10 + 2/100 + 5/1000 = .125$$

Sistema Binario

$$0/2 + 0/4 + 1/8 = .125$$

Entonces, ¿Cuál es el problema?

Ya vimos como los humanos representamos números de punto flotante y como lo hacen las computadoras sin embargo; hay algunos números bastante largos, muuuuuuuuy largos. Por ejemplo:

$$1/3$$

Los humanos podríamos representar esa fracción así

$$0.3$$

o aún mejor

$$0.33$$

inclusive mucho mejor

$$0.333$$

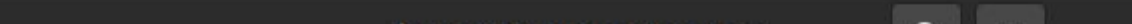
.000110011001100110011001100110011001100110011001100110011...

```
def binary(binary_num):
    sums = 0.0
    pos = 1
    for i in binary_num:
        if i == '1':
            sums += round(1/(2**pos),10)
            pos += 1
    return round(sums, 10)

def main():
    #ingresa solo los bits despues del punto y sin los tres puntos finales
    bin_num = input('Give me the binary num: ')
    result = binary(bin_num)
    print(f'{bin_num} is {result} in decimal')

if __name__ == '__main__':
    main()
```

`(0.1 + 0.1 + 0.1 == 0.3)` retorna **False**



A terminal window titled "josue@MagnusGates73: ~" with search, menu, and window control icons. The prompt is ">>>". The user enters "0.1 + 0.1 + 0.1", and the output is "0.30000000000000004". The prompt is then followed by an empty box.

```
>>> 0.1 + 0.1 + 0.1
0.30000000000000004
>>> 
```

Relajate, esto no quiere decir que ya no puedes confiar en los números de punto flotante



El que haya inexactitudes a la hora de representarlos no es un bug en Python, ni tampoco hay nada mal en tu código, esto es propio de la naturaleza de los binarios a la hora de guardar flotantes y pasa en muchos otros lenguajes. Lo importante es que entiendas que la aritmetica que se lleva acabo en las computadoras es binaria, no decimal.

Si bien hay algunas inexactitudes, estas son muy poquissimas y en la mayoria de cálculos comunes obtendrás lo que esperas si redondeas los resultados finales

```
Josue@MagnusGates73: ~  
>>> round(0.1 + 0.1 + 0.1, 1) == round(0.3, 1)  
True  
>>> █
```

esto le dirá a Python que solo considere una parte del número y no todo lo que esta guardado.

Si lo que necesitas son representaciones precisas de numeros decimales utiliza el modulo **decimal** en Python

👉 <https://docs.python.org/3/library/decimal.html#module-decimal>

Tambien existe el modulo **fractions** para representar exactamente fracciones

👉 <https://docs.python.org/3/library/fractions.html#module-fractions>