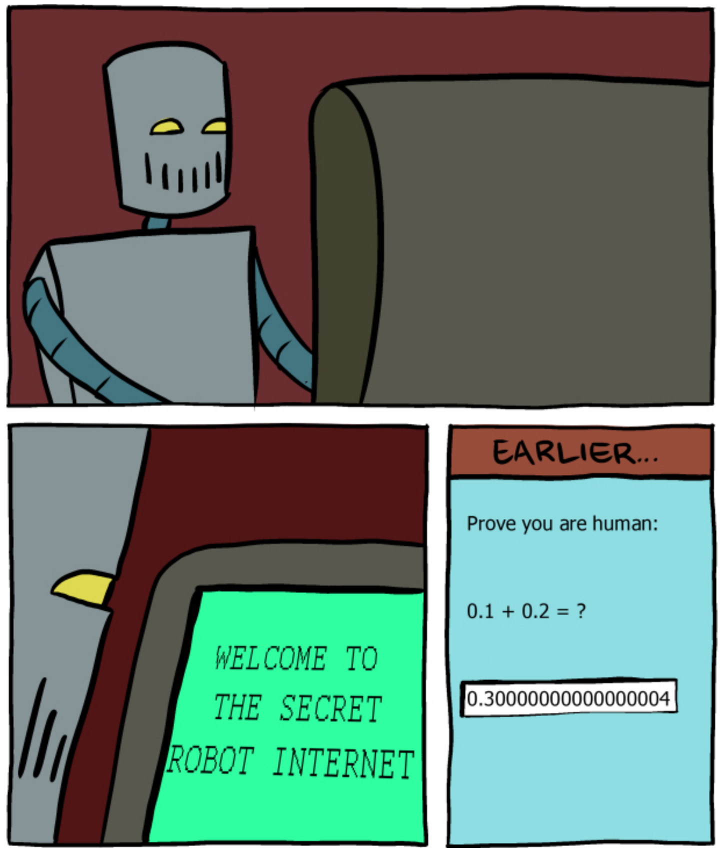

To understand the meme, just type in Python

```python
0.1 + 0.2
```

```
0.30000000000000004
```

And then, well, obviously

$$0.1 + 0.2 = 0.3,$$

so it seems weird that a computer returns 0.30000000000000004 while the answer is "obviously" (for us humans) 0.3.

Here is an explanation.
We did the 64-bit floating-point machine number representation of 0.1 and 0.2 in EX.0.3.7 (see url), and we saw that, in a 64-bit floating point IEEE system, 0.1 and 0.2 are stored as the machine numbers, respectively as:

$$\begin{aligned}
\mathrm{fl}(0.1) &= +2^{-4}(1.1001100110011001100110011001100110011001100110011010)_2 && \text{\# this is a machine number} \\
\mathrm{fl}(0.2) &= +2^{-3}(1.1001100110011001100110011001100110011001100110011010)_2 && \text{\# this is a machine number}
\end{aligned}$$

If we prefer to see this in base 10:

$$\begin{aligned}
\mathrm{fl}(0.1) &= 0.1000000000000000055511151231257827021181583404541015625 && \text{\# this is a machine number} \\
\mathrm{fl}(0.2) &= 0.2000000000000000111022302462515654042363166809082031250 && \text{\# this is a machine number}
\end{aligned}$$

(Base 10 is not really useful. Base 2 is just fine. But base 10 might help for comprehension.)

Then, we do (in exact arithmetic) the base-two addition, $x = \text{fl}(0.1) + \text{fl}(0.2)$, and we get

$$
\begin{array}{rr l}
 & 2^{-4}(1.1001100110011001100110011001100110011001100110011010)_2 & \text{\# this is fl(0.1), this is a machine number} \\
+ & 2^{-4}(11.0011001100110011001100110011001100110011001100110100)_2 & \text{\# this is fl(0.2), this is a machine number} \\
\hline
 & 2^{-4}(100.1100110011001100110011001100110011001100110011001110)_2 & \text{\# this is x = fl(0.1) + fl(0.2)} \\
 & & \text{\# this is not a machine number}
\end{array}
$$

So in other words

$$x = \text{fl}(0.1) + \text{fl}(0.2) \quad = \quad +2^{-2}(1.0011001100110011001100110011001100110011001100110011001110)_2$$

x is not a machine number. We cannot store x exactly on the computer since x has 55 bits in the scientific notation. And we can only store 53 bits: 1 hidden bit and 52 mantissa bits.

We can do the same addition in base 10. (This is not really useful. Base 2 is just fine. But base 10 might help for comprehension.)

$$
\begin{array}{rr l}
 & 0.1000000000000000055511151231257827021181583404541015625 & \text{\# this is fl(0.1), this is a machine number} \\
+ & 0.2000000000000000111022302462515654042363166809082031250 & \text{\# this is fl(0.2), this is a machine number} \\
\hline
 & 0.3000000000000000166533453693773481063544750213623046875 & \text{\# this is x = fl(0.1) + fl(0.2)} \\
 & & \text{\# this is not a machine number}
\end{array}
$$

Now, $x = (\text{fl}(0.1) + \text{fl}(0.2))$ is stored in the computer as the machine number $\text{fl}(x) = \text{fl}(\text{fl}(0.1) + \text{fl}(0.2))$. $\text{fl}(x)$ will either be x_-, the machine number just smaller than x; or x_+, the machine number just bigger than x.

$$
\begin{array}{lcl l}
x_- & = & +2^{-2}(1.0011001100110011001100110011001100110011001100110011\ )_2 & \text{\# this is a machine number} \\
x & = & +2^{-2}(1.0011001100110011001100110011001100110011001100110011|10\ )_2 & \text{\# this is not a machine number} \\
x_+ & = & +2^{-2}(1.0011001100110011001100110011001100110011001100110100\ )_2 & \text{\# this is a machine number}
\end{array}
$$

We need to take the machine numbers x_- or x_+ whichever is closer from x. But we see that x is right in the middle of x_- and x_+. Dang it! ☺. Right in the middle. So then we use the "ties to even" rule: "if the number falls midway, it is rounded to the nearest value with an even least significant digit." (Please note that since x_- and x_+ are two consecutive machine numbers, the last bit of one of two must be a 0, while last bit of the other must be a 1. The rule says: "pick the one with the last bit a 0".) The 52-nd bit of x_- is 1. The 52-nd bit of x_+ is 0. So, then, we round x is x_+. So $\text{fl}(x)$ is x_+. So we get:

$$\text{fl}(\text{fl}(0.1) + \text{fl}(0.2)) = +2^{-2}(1.0011001100110011001100110011001100110011001100110100)_2$$

We can also look at x_-, x, and x_+ in base 10. We get

$$
\begin{array}{lcl l}
x_- & = & 0.299999999999999988897769753748434595763683319091796875 & \text{\# this is a machine number} \\
x & = & 0.3000000000000000166533453693773481063544750213623046875 & \text{\# this is not a machine number} \\
x_+ & = & 0.3000000000000000444089209850062616169452667236328125000 & \text{\# this is a machine number}
\end{array}
$$

So that

$$\text{fl}(\text{fl}(0.1) + \text{fl}(0.2)) = 0.3000000000000000444089209850062616169452667236328125000$$

We can quickly check all this in python with

```python
import struct
print(f"{struct.unpack('<Q',struct.pack('<d',( 0.1 + 0.2 )))[0]:#066b}")
```

```
0b0011111111010011001100110011001100110011001100110011001100110100
```

We remove the `0b` at the start and are left with the 64 bits. Breaking the 64 bits with 1 bit (sign), 11 bits (exponent) and 52 bits (mantissa), this reads:

$$0 \mid 01111111101 \mid 0011001100110011001100110011001100110011001100110100$$

The first bit `0` represents the sign $+$. ✓ The next 8 bit bit `01111111101` represents the exponent -2, We have that $(01111111101)_2 = 1021$ and then $1021 - 1023 = -2$, so we find that the exponent is -2. ✓ The last 52 bits `0011001100110011001100110011001100110011001100110100` is the mantissa, they represent $(1.0011001100110011001100110011001100110011001100110100)_2$. ✓

And here you go. This is why when we type

```
0.1 + 0.2
```

we get

```
0.30000000000000004
```

Fun!

<u>Note:</u> I used **mpmath** for extended precision accuracy for the following computations. The goal was to obtain the exact base-10 representation of numbers. This is not needed in practice but base-10 might help for some understanding.

```
from mpmath import *
mp.dps = 100

# In base 10, fl( 0.1 ) is
print("fl(0.1)                  = ", mpf(2**(-4)*(  2**(  0) + 2**( -1) + 2**( -4)
         + 2**(-20) + 2**(-21) + 2**(-24) + 2**(-25) + 2**(-28) + 2**(-29) + 2**(
         + 2**(-40) + 2**(-41) + 2**(-44) + 2**(-45) + 2**(-48) + 2**(-49) + 2**(

# In base 10, fl( 0.2 ) is
print("fl(0.2)                  = ", mpf(2**(-3)*(  2**(  0) + 2**( -1) + 2**( -4)
         + 2**(-20) + 2**(-21) + 2**(-24) + 2**(-25) + 2**(-28) + 2**(-29) + 2**(
         + 2**(-40) + 2**(-41) + 2**(-44) + 2**(-45) + 2**(-48) + 2**(-49) + 2**(

# In base 10, fl(0.1) + fl( 0.2 ) is
print("fl(0.1) + fl(0.2)        = ", mpf(2**(-4)*(  2**(  0) + 2**( -1) + 2**( -4)
         + 2**(-20) + 2**(-21) + 2**(-24) + 2**(-25) + 2**(-28) + 2**(-29) + 2**(
         + 2**(-40) + 2**(-41) + 2**(-44) + 2**(-45) + 2**(-48) + 2**(-49) + 2**(
         + mpf(2**(-3)*(  2**(  0) + 2**( -1) + 2**( -4) + 2**( -5) + 2**( -8) +
         + 2**(-20) + 2**(-21) + 2**(-24) + 2**(-25) + 2**(-28) + 2**(-29) + 2**(
         + 2**(-40) + 2**(-41) + 2**(-44) + 2**(-45) + 2**(-48) + 2**(-49) + 2**(

# In base 10, fl( fl( 0.1 ) + fl( 0.2 ) ) is
print("fl( fl(0.1) + fl(0.2) ) = ", mpf(2**(-2)*(  2**(  0) + 2**( -3) + 2**( -4)
                  + 2**(-23) + 2**(-24) + 2**(-27) + 2**(-28) + 2**(-31) + 2**(-
                  + 2**(-43) + 2**(-44) + 2**(-47) + 2**(-48) + 2**(-50) )))
```

```
fl(0.1)                 =  0.1000000000000000055511151231257827021181583404541015
fl(0.2)                 =  0.2000000000000000111022302462515654042363166809082031
fl(0.1) + fl(0.2)       =  0.3000000000000000166533453693773481063544750213623046
fl( fl(0.1) + fl(0.2) ) =  0.3000000000000000444089209850062616169452667236328125
```