Numerical_Methods_Week1_roundingoff

October 11, 2022

0.0.1 Notebook 1 - Your computer is a liar

Whenever you want to compare numbers your computer may lie to you, it becomes specially evident with simple operations like the following:

```
[1]: 0.1 + 0.2 == 0.3
```

```
[1]: False
```

Does it always lie or not?

```
[2]: 1.3 + 255.0 == 256.3
```

```
[2]: True
```

```
[3]: 1.2 + 2.4 + 3.6 == 7.2
```

```
[3]: False
```

Which way does it lie to you?

```
[4]: 0.1 + 0.2 <= 0.3
```

```
[4]: False
```

```
[5]: 10.4 + 20.8 > 31.2
```

```
[5]: True
```

```
[6]: (20.8-10.4 ) -10.4
```

```
[6]: 0.0
```

```
[7]: #0.8-0.1>0.7 ??
```

```
[8]: #0.1-0.8<-7 ??
```

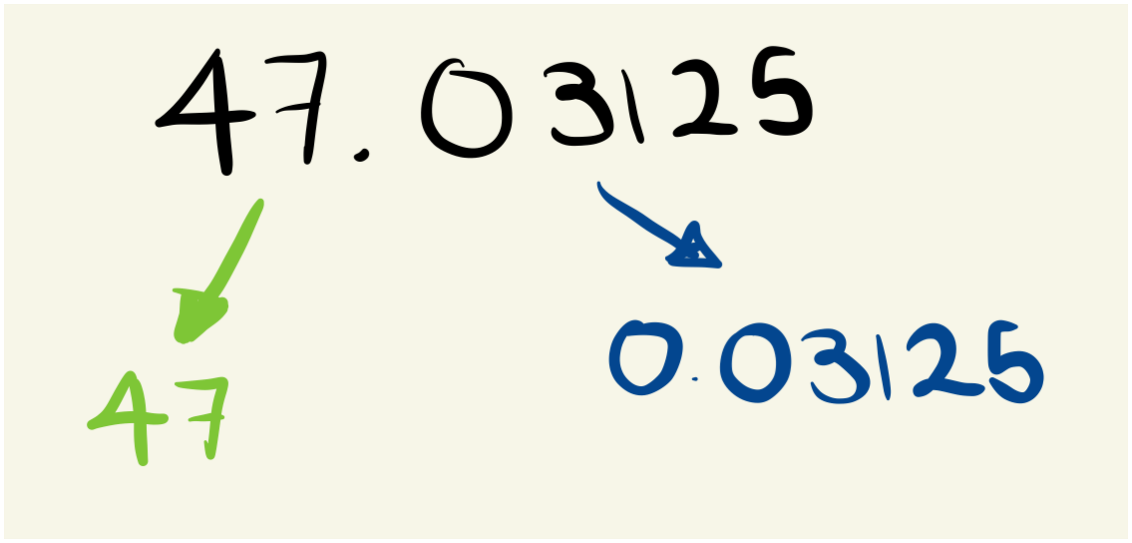
So what's going on? Why is your computer lying to you?

Well it's not exactly that it's lying it's just the fact that it gets some little extra digits when translating for you. Now wait a sec? What do you mean by translating? Well your computer speaks binary natively while you speak decimal so when you type the number 0.1 into the Python interpreter, it gets stored in memory as a floating-point number (IEEE-754 standard).

Here's a conversion that takes place when this happens. 0.1 is a decimal in base 10, but floating-point numbers are stored in binary. In other words, 0.1 gets converted from base 10 to base 2. When your computer translates the resulting binary number may not accurately represent the original base 10 number. 0.1 is one example. To understand this let us go briefly to the way this conversion works Consider the following number 47.03125 . Do you think it can be represent accurately?

Let us find out (for this example we will use single precision [1]:

1. The first step is to separate the number into the whole number and its decimal part



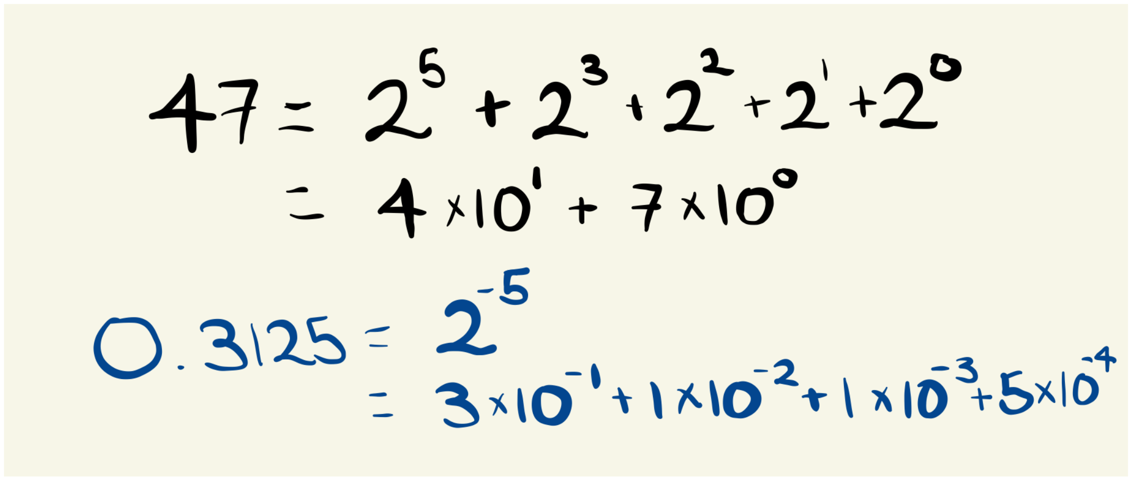
2. Then we represent the whole part in binary

$47 = 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 101111$

3. Then we represent the decimal part in binary -> This was perhaps the major point of confusion during the lecture

$0.03125 = 0 \times 2^{-1} + 0 \times 2^{-2} + 0 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5} = 0.00001$

If you're having trouble understanding why it's done this way, think about how you would do it in the decimal system before looking at the image below


$47 = 2^5 + 2^3 + 2^2 + 2^1 + 2^0$
$= 4 \times 10^1 + 7 \times 10^0$
$0.3125 = 2^{-5}$
$= 3 \times 10^{-1} + 1 \times 10^{-2} + 1 \times 10^{-3} + 5 \times 10^{-4}$

3. Write things in base 2 “scientific notation”

$101111.00001 = 1.0111100001 * 2^5$

Since the First digit in binary scientific notation we may forget about this bit and only use the ones after the decimal point in our Mantissa

4.Add bias to the exponent [1] and convert it to binary

$E = 127 + 5 = 132 = 10000100$

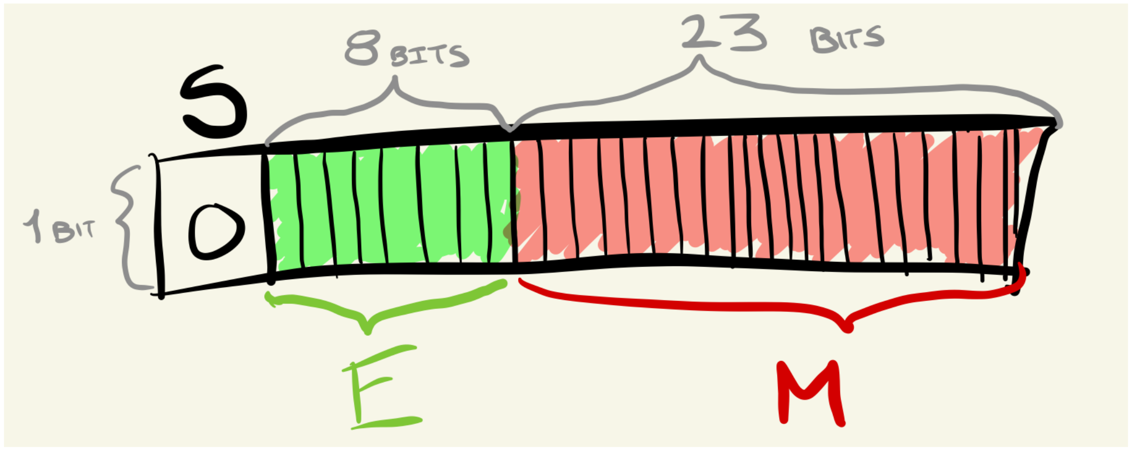
5. Determine the sign of the number

The sign is plot so our sign bit is set to 0

6. Combine Numbers together according to the standard

In this example we will use the single precision (32 bits), however if you work with double precision the format is the same, however how many bits are allocated is different for single precision we have

- S = 1 Bit ->Sign
- E = 8 bits -> Exponent
- b = 127 -> Bias
- M = 23 bits -> Mantissa



While for double precision we have

- S = 1 Bit ->Sign
- E = 11 bits -> Exponent
- b = 127 -> Bias
- M = 52 bits -> Mantissa

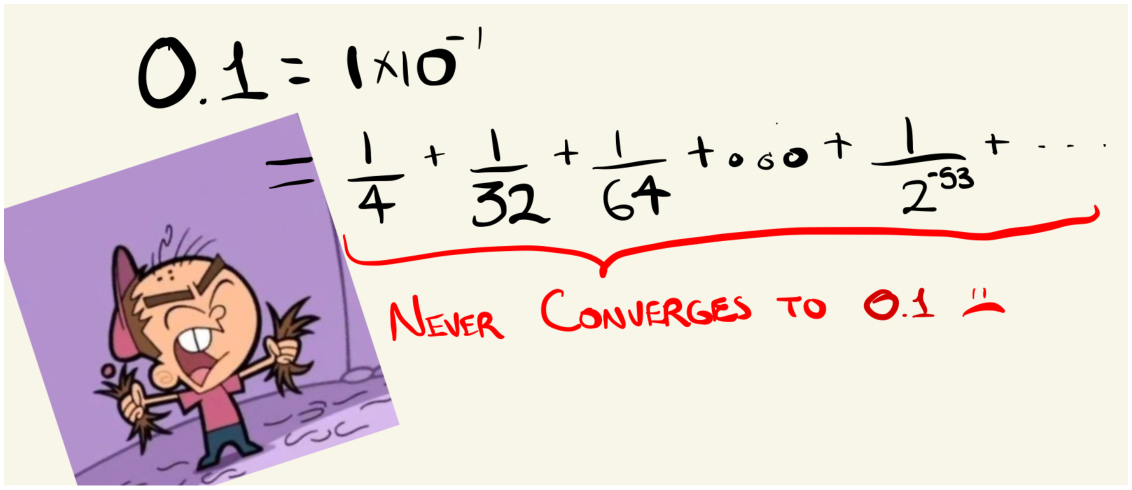
So 47.03125 is represented as :

$$\overbrace{0}^{S}\underbrace{10000100}_{E}\underbrace{01111000001000000000000}_{M}$$

The previous number could be represented acurately on the computer while this is not always the case, take the number 0.1 for example, it's binary representation is:

$$0.1 = 0\times2^{-1}+0\times2^{-2}+0\times2^{-3}+1\times2^{-4}+1\times2^{-5}+0\times2^{-6}+0\times2^{-7}+1\times2^{-8}+1\times2^{-9}+0\times2^{-10}+0\times2^{-11}+1\times2^{-12}+1\times2^{-1}$$

We just ran out of bits in the mantissa to express 0.1 so what we do now is to round up the last bit we use to express 0.1 to 1, so we aproximate 0.1 to a slightly higher number, the same thing would happen with double precision



0.3, also has the same issue. Computer memory is finite, so the infinitely repeating binary fraction representation of 0.3 gets rounded to a finite fraction. That is to say we truncate the above number and fit it into our format, knowing the mantissa is 53 bits long which do you think is the difference bewtween 0.3 and 0.1+0.2 ?

Task

Represent 30.274 in single precision according to the IEEE-754 standard

```
[139]: ((0.1+0.2)-0.3)
```

```
[139]: 5.551115123125783e-17
```

```
[10]: # The mantissa in doble precision is 52 bits long, however when using the␣
       ↪scientific notation we loose one bit (0 * 2^{-1}) and another is not
      #considered in the mantissa, then the bit we round up is the 54th
      2**(-54)
```

```
[10]: 5.551115123125783e-17
```

If you want to take a look at the fraction to wich the number is approximated you may use the function below, accordingly it is an example of how to write proper code, evethough from now we are not going to use python anymore the same should hold, you should annotate what the function does and an example in octave as well

```
[2]: # Example of how to write a proper function
     def fraction_rep(x:float)->str:
         """
         This function takes a floar x as input and returns the approximate binary
         fraction
         Example:
         >>> fraction_rep(0.1)
             '0.1   3602879701896397 / 36028797018963968'
         """
         numerator, denominator = (x).as_integer_ratio()
         return f"{x}   {numerator} / {denominator}"
```

```
[3]: fraction_rep(0.3)
```

```
[3]: '0.3   5404319552844595 / 18014398509481984'
```

```
[4]: fraction_rep(0.8)
```

```
[4]: '0.8   3602879701896397 / 4503599627370496'
```

```
[5]: fraction_rep(2.0) # Sanity check
```

```
[5]: '2.0   2 / 1'
```

Equalities or inequealities don't work right, what should I do for comparisons?

Basicly we consider how close a number is given a certain ϵ value. by default it is 1×10^{-9} in python

```
[7]: from math import isclose
```

```
[8]: isclose((0.1 + 0.2),0.3)
```

```
[8]: True
```

```
[9]: isclose(0.1 + 0.2, 0.3, rel_tol=1e-20) #decreasing the epsilon makes this the␣
     ↪same as equality, you should not just decrease it carelessly
```

```
[9]: False
```

What about arrays?

```
[10]: import numpy as np
```

```
[11]: array1=np.array([0.1, 0.2]) + np.array([0.2, 0.4])
      array2=np.array([0.3, 0.6])
      array1==array2
```

```
[11]: array([False, False])
```

```
[13]: np.isclose(array1,array2)
```

```
[13]: array([ True,  True])
```

Due to this issue Can we trust limits?

```
[14]: limit=lambda x:  np.sin(x)/(x)
```

```
[15]: limit(1e-12)
```

```
[15]: 1.0
```

if $f(x) = \frac{1-cos(x)}{x^2}$ will the limit $\lim_{x->0} f(x)$ work ?

```
[19]: limit2=lambda x: (1-np.cos(x))/(x**2)
```

```
[26]: limit2(1e-12)
```

```
[26]: 0.0
```

We see that it's false when the number is extremely small but what about, not so small numbers

```
[30]: limit2(1e-7)
```

```
[30]: 0.4996003610813205
```

Then the limit is well approximated. When dealing with limits numerically it's important to check them since actually approaching the limit may make the computer make mistakes, while not being so close may give you a nice approximation

Task Can you tell whether the limit $f(x) = xcoth(x)$ will work?

0.0.2 Does the order of summation matter?

Take for example the following expression

$x = \sum_{n=0}^{M}(0.9)^n = 1 + 0.9 + (0.9)^2 + ... + (0.9)^M = (0.9)^M + (0.9)^{M-1} + ... + 1$

Do you think that it will matter whether we start doing the sum from the left that is $1 + 0.9 + (0.9)^2 + ... + (0.9)^M$ or from the right meaning $(0.9)^M + (0.9)^{M-1} + ... + 1$?

Let us explore this situation

[34]:
```
def suma(M=200,right= False):
    """
    This function performs the sum of (0.9)^{n} with n from 0 to M
    if right is False and M to 0 if right is True
    >>> suma(right=True)
        8.99999999365043
    """
    rg=range(0,M,1) if not right else range(M,0,-1)
    result=np.array([(0.9)**i for i in rg]).sum()
    return result
```

[39]:
```
suma(right=True)
```

[39]:
```
8.99999999365043
```

[40]:
```
suma()
```

[40]:
```
9.999999992944922
```

Notice that there is a big difference in them. **Task** Which one is a better implementation?

Tip: You may find the answer with the help of this link 1

Exercise Alternate signs using 0.99999 what happens? What is the percentage error?