Extended Introduction to Computer Science
CS1001.py

Chapter B Floating Point Representation
Lecture 7a

Amir Rubinstein, Michal Kleinbort

School of Computer Science
Tel-Aviv University
Fall Semester 2023-24
http://tau-cs1001-py.wikidot.com

* Slides based on a course designed by Prof. Benny Chor

Wonders of “Real Numbers” in Python

- Look at this, a very disturbing phenomenon:

```
>>> 0.1+0.1 == 0.2
True
>>> 0.1+0.1+0.1 == 0.3
False
```

- And indeed,

```
>>> 0.1+0.1
0.2
>>> 0.1+0.1+0.1
0.30000000000000004
```

- We need some understanding of how decimal point numbers are represented in the computer’s memory.

Fixed Point

- A simple way to represent decimal point numbers in the computer's memory would be a fixed point representation.

- Suppose we allocate n decimal digits for such numbers
- We designate a fixed number of digits to the right of the decimal point. Denote this number k ($0 < k < n-1$):

$$\underbrace{d_0 d_1 d_2 \dots d_{k-1}}_{n-k} . \underbrace{d_k d_{k+1} \dots d_{n-1}}_{k}$$

- For example, if $n = 7$ and $k = 2$, then 1498523 represents 14985.23
- The (fixed) value of k can be regarded as an implicit "scaling factor."

  (in the example above the scaling factor is $1/100$).

~~Fixed Point~~

- However, there are some major disadvantages to this method, which is why it is hardly used today in modern systems.

- First, this method bounds numbers to a fixed order of magnitude and precision.
  - For example, distances between galaxies and diameters of atomic nucleus cannot be both expressed with the same fixed scaling factor.

- Also, frequent rounding due to arithmetical operations may cause loss of precision.
  - For example, if the scaling factor is 1/100, multiplying two numbers is likely to yield a number with 4 digits of precision, which then must be rounded to conform with the fixed precision (look at $0.01 \cdot 0.01 = 0.0001$)

Floating Point

- Today, most decimal point numbers are represented using floating point representation. This method allows different orders of magnitude and precision within the same type.

- Over the years, a variety of floating point representations have been used in computers. However, since the 1990s, the most commonly encountered representation is that defined by the IEEE 754 Standard, some of which will be presented next.

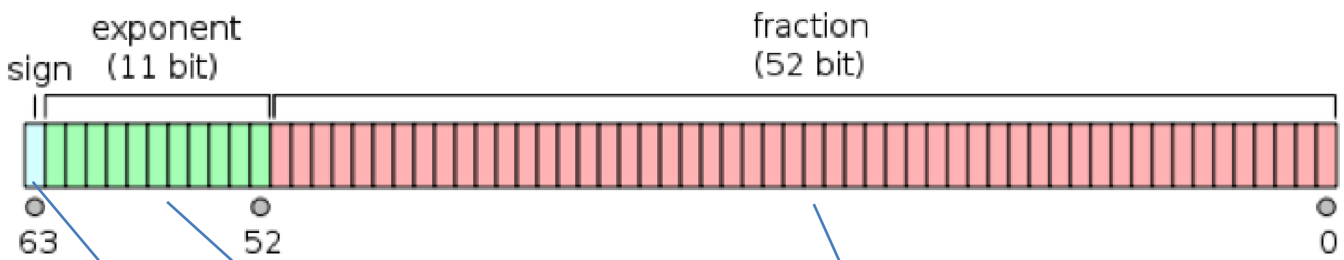
- The basic idea:

  14985.23 can be represented as 1.498523×10^{4}

  0.001498523 can be represented as 1.498523×10^{-3}

IEEE 754 Standard for Floating Point Numbers

- Suppose we deal with a machine of 64 bit words. A floating point number is represented by 64 bits:



(figure from Wikipedia)

and is coded by:

$$(-1)^{sign} \cdot 2^{exponent-1023} \cdot (1 + fraction)$$

- The $sign$ bit: 0 indicates non-negative, 1 indicates negative
- The $exponent$ is an 11 bit integer, so $-1023 \leq exponent - 1023 \leq 1024$
- The $fraction$ is a sum of negative powers of 2, represented by 52 bits:

$$0 \leq fraction \leq \sum_{i=1}^{52} \frac{1}{2^i} = 1 - 2^{-52}$$

Example #1

Which number is represented here?

0 01111111110 1000000000000000000000000000000000000000000000000000

- The sign bit is 0 → +
- exponent = 01111111110_2 = 1022_{10}
- The fraction bits are 100...0 →

  fraction = $\sum_{i=1}^{52} b_i \cdot \frac{1}{2^i} = 1 \cdot \frac{1}{2} + 0 \cdot \frac{1}{4} + \ ... + 0 \cdot \frac{1}{2^{52}} = \frac{1}{2}$

$$\boxed{(-1)^{sign} \cdot 2^{exponent-1023} \cdot (1 + fraction)}$$

$$(+) \quad \cdot 2^{1022\,-1023} \quad \cdot \left(1 + \frac{1}{2}\right) \quad = 2^{-1} \cdot 1.5 = 0.75$$

- Look at: https://float.exposed/0x3fe8000000000000

Example #2

Which number is represented here?

1 10000000001 0110000000000000000000000000000000000000000000000000

- The sign bit is 1 → -
- exponent = 10000000001_2 = 1025_{10}
- The fraction bits are 01100...0 →

$$\text{fraction} = \Sigma_{i=1}^{52} b_i \cdot \frac{1}{2^i} = 0 \cdot \frac{1}{2} + 1 \cdot \frac{1}{4} + 1 \cdot \frac{1}{8} + 0 \cdot \frac{1}{16} + \cdots + 0 \cdot \frac{1}{2^{52}}$$
$$= \frac{1}{4} + \frac{1}{8} = 0.375$$

$$\boxed{(-1)^{sign} \cdot 2^{exponent-1023} \cdot (1 + fraction)}$$
$$(-) \quad \cdot 2^{1025-1023} \quad \cdot (1 + 0.375) \quad = -2^2 \cdot 1.375 = -5.5$$

- Look at: https://float.exposed/0xc016000000000000

Floating Point has Bounded Accuracy

- Accuracy is of course bounded (determined by the "word size'', the operating system you are using ,the version of the interpreter, etc., etc.).

- Indeed, floating point arithmetic carries many surprises for the unwary. This follows from the fact that floating numbers are represented as a number in binary, namely the sum of a fixed number of powers of two.

- The bad news is that even very simple rational numbers cannot be represented this way with complete accuracy.
  - For example, the decimal $0.1 = \frac{1}{10}$ cannot be represented as a sum of powers of two, since the denominator has prime factors other than 2, in this case, 5.

The 0.1 Example

- A confusing issue is that when we type 0.1 (or, equivalently, 1/10) to the interpreter, the reply is 0.1.

```
>>> 1/10
0.1
```

- This does not mean that 0.1 is represented exactly as a floating point number. It just means that Python's designers have built the display function to act this way.

- In fact the inner representation of 0.1 on most machines today is (see here):

$$+2^{-4} * 1.6000000000000000888178419700125232338905334472656 25$$

- And so is the inner representation of 0.10000000000000001. Since the two have the same inner representation, no wonder that display treats them the same:

```
>>> 0.10000000000000001
0.1
```

Arithmetic of Floating Point Numbers (for reference only)

- The speed of floating point operations, commonly measured in terms of FLOPS, is an important characteristic of a computer system, especially for applications that involve intensive numerical calculations.

- Addition is done by first shifting both numbers to have the same exponent, then adding the fractions, then converting back so that the fraction is smaller than 1 (recall $0 \leq \text{fraction} \leq 1\text{-}2^{-52}$).

- Multiplication is done by multiplying the two fractions, and adding the two exponents.

- Subtraction and division are analogous to addition and multiplication, correspondingly.

- Arithmetical operations may lead to substantial loss of precision

Arithmetic of Floating Point Numbers

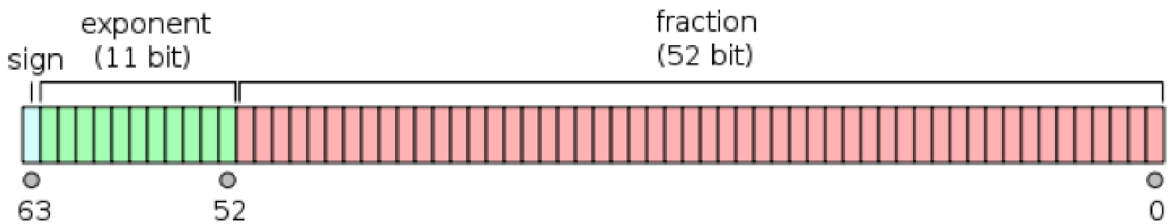
- Arithmetical operations may lead to substantial loss of precision:

```
>>> a = 10.0**40
>>> a
1e+40
>>> b = 10.0**4
>>> a-b
1e+40
>>> a-b == a
True     ☹

>>> a%b
752.0    ☹
```

Exercise

- How many different floating point values are in $[1,2)$?



$$\boxed{(-1)^{sign} \cdot 2^{exponent-1023} \cdot (1 + fraction)}$$

  - Sign bit must be 0 (+)
  - Recall $0 \leq fraction < 1$, and so $1 \leq 1 + fraction < 2$
    - 2^{52} different fractions possible
  - $exponent$ must be 1023

- We get 2^{52} numbers in $[1,2)$
- What about [2,4)? [4,8)? $[2^n, 2^{n+1})$?

Uneven Spread of Floating Point

- Indeed, for every range of numbers between adjacent powers of 2, there are an equal number of representable numbers, so floating point numbers become more sparse as they increase in magnitude.

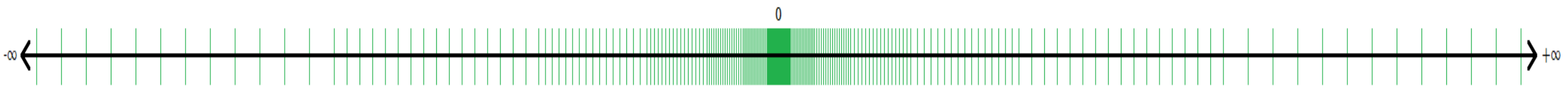


Image from Wikipedia

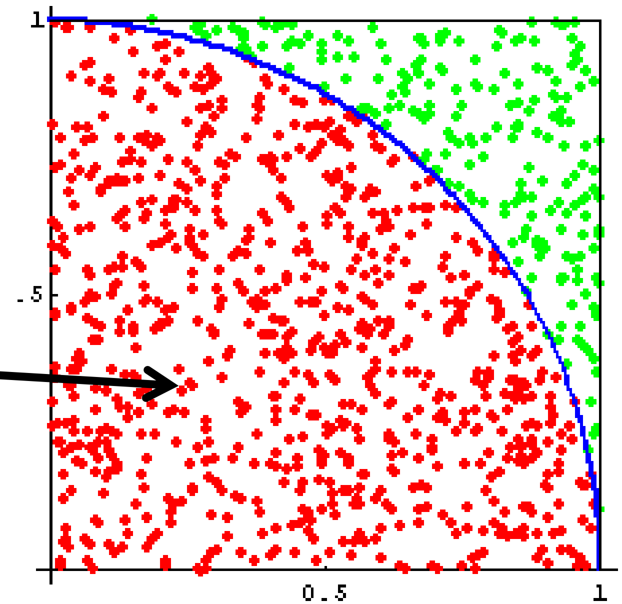
Life is (a bit) More Complicated (for reference only)

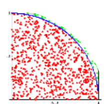
- There are several special cases, which deviate from the formula we saw. These are used for special values such as 0.0, and occur when either the exponent is all 0s' or all 1's, or when the fraction is all 0's.

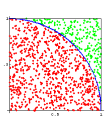
$fraction$ / $exponent$	all 0's	Otherwise (not all 0's)
all 0's	0.0	"Subnormal numbers"
all 1's	∞	NaN

Example: Estimating π

- Randomly choose points (x, y) in the unit square ($0 \leq x, y < 1$)
- Count how many are located inside the quarter circle of radius 1 centered in the origin
- The ratio is an approximation for $\frac{\pi}{4}$







$$= \frac{\frac{1}{4}\pi r^2}{r^2} = \frac{\pi}{4}$$

Figure taken from http://mathfaculty.fullerton.edu/mathews/n2003/montecarlopimod.html

Estimating π in Python

```
import random

def estimate_Pi(sample_size=1000):
    """ estimate pi, using sample_size random choices """
    count=0
    for n in range(sample_size):
        x = random.random()
        y = random.random()
        if x**2 + y**2 <= 1.0: # inside circle
            count += 1
    return 4*count/sample_size # 4*(pi/4) = pi
```

```
>>> estimate_Pi()
3.156
>>> estimate_Pi(100000)
3.12864
>>> estimate_Pi(10**8)
3.14175412
```

default: 1000 samples

took ~1 min

```
>>> import math
>>> math.pi
3.141592653589793
```

Comic Relief *

https://www.youtube.com/shorts/TXl4q-ZEjvA

* אני מזמין אתכם לשלוח לי הצעות לתמונות שיופיעו על שקפים אלו לאורך הסמסטר