Mysterious World of Python Float Arithmetics

Syed Muhammad Dawoud
Sheraz Ali

Agenda

- Problem Statement
- Float representations in base 10 and 2
- Float representation on hardware
- Python and Floats

whoami

- Software Engineer @ Arbisoft, Lahore, Pakistan
- Python & Django

Medium

@dawoud.sheraz

SCAN ME

LinkedIn

syed-muhammad-dawoud-sheraz-ali

SCAN ME

Syed M. Dawoud Sheraz Ali

Guess the outcome

```
>>> 0.1 + 0.1 + 0.2 == 0.4
>>> 0.1 + 0.1 + 0.1 == 0.3
>>> 100 * 0.357
>>> 64.8 - 35.8
>>> 0.3 - 0.2 >= 0.1
>>> 500.50 - 400.05
>>> 1000 * 0.357
>>> 64.8 - 35.7
```

Expectations?

```
>>> 0.1 + 0.1 + 0.2 == 0.4
```

TRUE

```
>>> 0.1 + 0.1 + 0.1 == 0.3
```

TRUE

```
>>> 100 * 0.357
```

35.7

```
>>> 64.8 - 35.8
```

29.0

```
>>> 0.3 - 0.2 >= 0.1
```

TRUE

```
>>> 500.50 - 400.05
```

100.45

```
>>> 1000 * 0.357
```

357

```
>>> 64.8 - 35.7
```

29.1

Actual

```
>>> 0.1 + 0.1 + 0.2 == 0.4
True
```

```
>>> 0.3 - 0.2 >= 0.1
False
```

```
>>> 0.1 + 0.1 + 0.1 == 0.3
False
```

```
>>> 500.50 - 400.05
100.44999999999999
```

```
>>> 100 * 0.357
35.699999999999996
```

```
>>> 1000 * 0.357
357.0
```

```
>>> 64.8 - 35.8
29.0
```

```
>>> 64.8 - 35.7
29.099999999999994
```

PyCon PAKISTAN

What’s Going On?

Fractions

- Fraction → Representing part/portion of whole thing
- a/b → a=Numerator, b=Denominator
- ½, ⅗, 9/10, 99/101

Decimals

- Type of numbers that have a whole and a fractional part, separated by a decimal point
- 3.1415, 9.98, 1.05, 34.901, etc.
- 3.1415
  - 3 → whole part
  - 1415 → fractional part
  - . → decimal point

Fractions and Decimals

- Both fractions and decimals are mostly interconvertible
  - Focus here would be on Fraction -> Decimal
- Fraction -> Decimal
  - finite repeating decimals
  - infinite repeating decimals

Fractions and Decimals

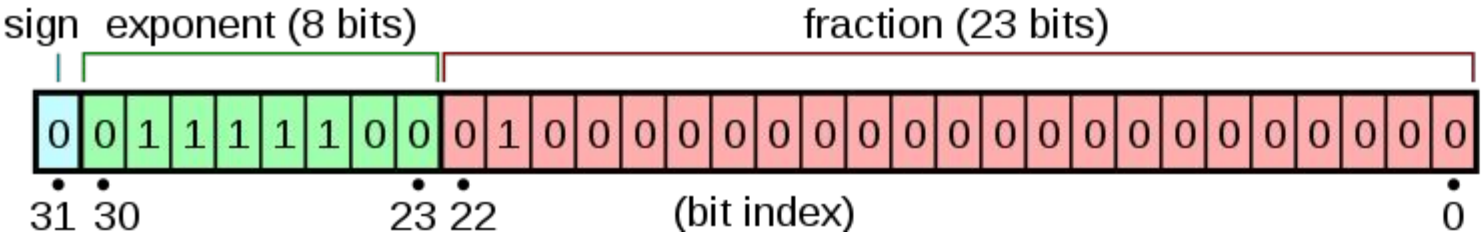
- ½ → 0.50
- ¼ → 0.25
- ⅓ → 0.333 or 0.3333333 or 0.333333333
- ⅙ → 0.166666666667
- 3/11 → 0.2727272727

Float approximations - Base 2

- Like other numbers, floats represented as base-2 on hardware
- Finite and infinite repeating exists in binary as well
- 0.25 → 0.01
- 0.1 → 0.0001100110011001100110011001100…
- 0.1667 → 0.00101010101010110011011001111010000…
- 0.333333 → 0.010101010101010101010100111110111101101011010111…

IEEE 754

- IEEE 754 standard used for 32 bits float representation
- Rounding off the fractional/mantissa to 23 bits leads to rounding off errors



PyCon PAKISTAN

IEEE 754 Conversion – Example

- Representing 10.1 in IEEE 754
- Start with converting whole and fractional parts into binary
- 10
  - 10 / 2 → **R=0**, Q=5
  - 5/2 → **R=1,** Q=2
  - 2/2 → **R=0**, Q=1
  - 1/2 → **R=1**, Q=0
- $10_{10} \rightarrow 1010_2$

PyCon PAKISTAN

IEEE 754 Conversion – Example

- 0.1
  - 0.1 * 2 = **0**.2
  - 0.2 * 2 = **0**.4
  - 0.4 * 2 = **0**.8
  - 0.8 * 2 = **1**.6
  - 0.6 * 2 = **1**.2
  - 0.2 * 2 = **0**.4
  - 0.4 * 2 = **0**.8
  - 0.8 * 2 = **1**.6
  - 0.6 * 2 = **1**.2
  - 0.2 * 2 = **0**.4
  - … (infinite repeating)
- $0.1_{10} = 00011001100110011001100110011$

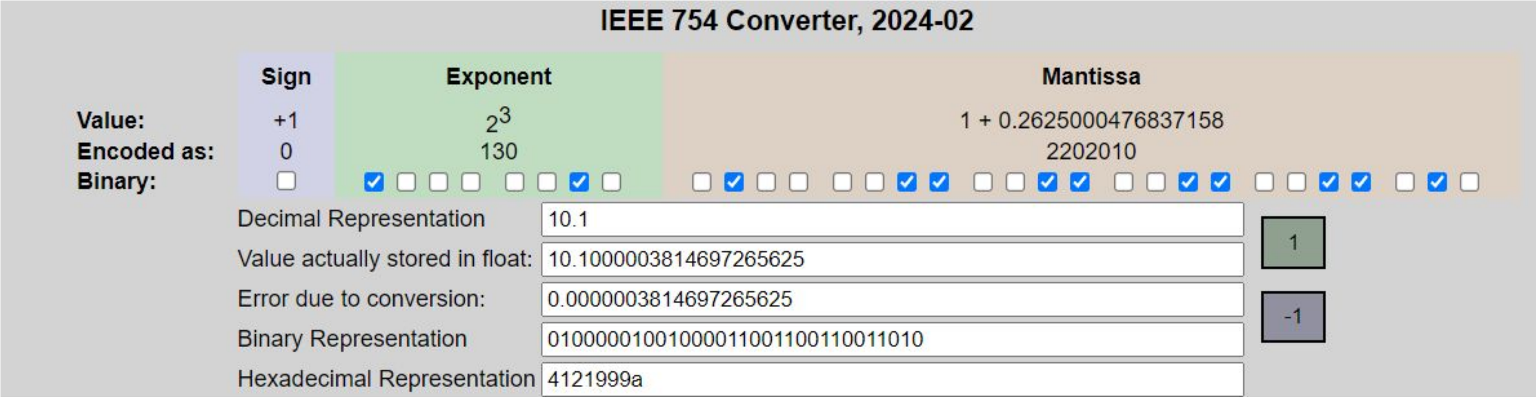
IEEE 754 Conversion – Example

1. 10.1_{10} = 1010.00011001100110011001100110011 (not done yet)
2. Next, shift the decimal either right or left until there is only '1' before decimal
   a. 1010.00011001100110011001100110011 → 1.01000011001100110011001100110011
   b. 3 points moved left
3. Round off decimal portion to 23 bits
   a. **1.01000011001100110011001**10011 → 1.01000011001100110011010
4. Calculate biased exponent
   a. IEEE 754, constant bias value = 127
   b. Bias exponent calculated = 127 + 3 → 130 (add +3 as decimal point was shifted 3 points left, the value would be subtracted if shifted right)
   c. $130_{10} = 10000010_2$

PyCon PAKISTAN

IEEE 754 Conversion – Example

- 10.1
  - Sign = 0
  - Exponent = 10000010
  - Mantissa = 01000011001100110011010
- Assembled → 0 10000010 01000011001100110011010

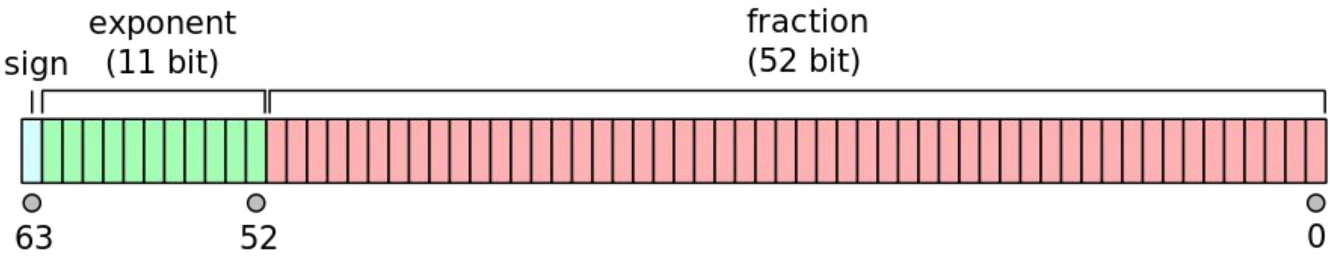


Summarized

1. Convert both whole and fractional parts to binary
2. Shift decimal so there is only one 1 in front of the point
3. Round off fractional part
4. Add bias to the exponent value
5. Convert exponent to the binary system
6. Assemble floating point number with sign bit at start

IEEE 754-2008 Standard

- Double precision floating point format
- Also known as float64 or binary64



- Floats represented as approximations in most cases
- A degree of precision error is there regardless
- Due to hardware, the approximation issue is part of nearly every programming language
- How do we work around that?

Workarounds in Python

- Fraction
- Decimal
- round
- isclose

Workarounds

- Built-in fractions module
- Represent rational numbers as Fractions
- perform arithmetics on fractions
- Format fractions to floats when needed

```
>>> from fractions import Fraction
>>> fc = Fraction(2, 10)
>>> fc + 1 == Fraction(6, 5)
True
```

```
>>> fraction = Fraction(4, 5) + 2
>>> fraction
Fraction(14, 5)
>>> f"{fraction:.2f}"
'2.80'
>>> f"{fraction:.4f}"
'2.8000'
```

```
>>> Fraction(2) / 10
Fraction(1, 5)
>>> Fraction(10, 100)
Fraction(1, 10)
>>> Fraction(4, 5) + 1
Fraction(9, 5)
```

```
>>> Fraction('5/6')
Fraction(5, 6)
>>> Fraction('9/50')
Fraction(9, 50)
>>> Fraction.from_float(1.80)
Fraction(8106479329266893, 4503599627370496)
>>> Fraction.from_float(1.80).limit_denominator(1000)
Fraction(9, 5)
>>> Fraction.from_float(1.80).limit_denominator(100)
Fraction(9, 5)
```

Workarounds

- Built-in decimal module
- Designed on same standards as floats but allow programmers to handle precisions explicitly
- Create module from various data types and perform arithmetics

```
>>> from decimal import *
>>> getcontext()
Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999, capitals=1, clamp=0, flags=[], traps=[InvalidOpera
tion, DivisionByZero, Overflow])
>>> Decimal(2)/7
Decimal('0.2857142857142857142857142857')
>>> getcontext().prec = 8
>>> Decimal(2)/7
Decimal('0.28571429')
>>>
```

```
>>> (Decimal(2)/7) * 4
Decimal('1.1428572')
>>> ((Decimal(2)/7) * 4).as_integer_ratio()
(2857143, 2500000)
```

```
>>> getcontext()
Context(prec=8, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999, capitals=1, clamp=0, flags=[Inexact, FloatOperation
, Rounded], traps=[InvalidOperation, DivisionByZero, Overflow])
>>> Decimal(3.1415)
Decimal('3.14150000000000018118839761882554739713668823242187 5')
>>> Decimal('3.1415904543565')
Decimal('3.1415904543565')
>>> Decimal(105)
Decimal('105')
>>> Decimal(2/7)
Decimal('0.28571428571428569842538536249776370823383331298828125')
>>> Decimal(2/7) + Decimal('3.1415904543565')
Decimal('3.4273047')
```

Workarounds

- Built-in round function
- Takes 2 arguments; number and precision
- Round off manually to avoid unexpected results

```
>>> round(0.1 + 0.1 + 0.1, 2) == round (0.3, 2)
True
```

```
>>> round(145.95-45.45, 1)
100.5
```

```
>>> round(0.1, 1) + round(0.1, 1) + round(0.1, 1) == round(0.3, 1)
False
>>> round(0.1 + 0.1 + 0.1, 1) == round(0.3, 1)
True
```

Workarounds

- Built-in math.isclose
- Verify if the provided values are close to each other
  - Closeness is calculated based on absolute and relative tolerance
- Relative tolerance → maximum allowed difference between values, greater than zero
- Absolute tolerance → minimum absolute difference
- Under the hood → abs(a-b) <= max(rel_tol * max(abs(a), abs(b)), abs_tol).

```
>>> isclose(0.5, 0.3, rel_tol=0.05)
False
>>> isclose(0.5, 0.3, rel_tol=0.2)
False
>>> isclose(0.5, 0.3, rel_tol=0.4)
True
```

```
>>> isclose(0.6, 0.65, abs_tol=0.01, rel_tol=0.1)
True
>>> isclose(0.8, 0.65, abs_tol=0.01, rel_tol=0.1)
False
```

Unit Testing

- Unittest's built-in assertions
  - assertAlmostEqual
  - assertNotAlmostEqual
- Signature
  - First, second, decimal places (default 7), msg=None, delta=None
- How does it work?
  - computing the difference of two numbers
  - rounding to the given number of decimal places
  - Comparing to zero
  - Raise assertion depending upon the type
- If delta is supplied, the difference between should be less or equal to or greater than delta.

```python
def test_almost_equal(self):
    # 0.15-0.14 <= 0.05
    self.assertAlmostEqual(0.14, 0.15, delta=0.05)
    # 0.12 - 0.09 <= 0.04
    self.assertAlmostEqual(0.12, 0.09, delta=0.04)
    # round(.501-0.50, 2) = 0
    self.assertAlmostEqual(0.501, 0.50, places=2)
    # round(0.50-0.4999, 3)
    self.assertAlmostEqual(0.4999, 0.50, places=3)

def test_almost_not_equal(self):
    # round(0.501 - 0.50, 3) = 0.001
    self.assertNotAlmostEqual(0.501, 0.50, places=3)
    # round(0.50-0.4999, 4) = 0.0001
    self.assertNotAlmostEqual(0.4999, 0.50, places=4)
    # 0.01 >= 0.001
    self.assertNotAlmostEqual(0.5, 0.51, delta=0.001)
```

Closing Thoughts

- Floating approximation is weird and can cause apps to behave unexpectedly
- Align or set expectations explicitly in the code using workarounds to avoid surprises

Helpful Links

- https://medium.com/python-in-plain-english/mysterious-world-of-pythons-floating-numbers-subtraction-42e157b4bd77
- https://nisal-pubudu.medium.com/how-to-deal-with-floating-point-rounding-error-5f77347a9549
- https://en.wikipedia.org/wiki/IEEE_754
- https://en.wikipedia.org/wiki/IEEE_754-2008_revision
- https://betterprogramming.pub/floating-point-numbers-are-weird-in-python-heres-how-to-fix-them-51336e4ad51a
- https://docs.python.org/3/tutorial/floatingpoint.html
- https://jvns.ca/blog/2023/01/13/examples-of-floating-point-problems/
- https://numeral-systems.com/ieee-754-converter/
- https://www.h-schmidt.net/FloatConverter/IEEE754.html

Medium

@dawoud.sheraz

SCAN ME

LinkedIn

syed-muhammad-dawoud-sheraz-ali

SCAN ME

DawoudSheraz

SCAN ME