Floating point

```
Python 3.7.3 (default, Jul 25 2020, 13:03:44)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 0.3 - 0.2
0.09999999999999998
>>> 
```

```
Python 3.7.3 (default, Jul 25 2020, 13:03:44)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> x = 1000000000000000000000000.0
>>> x == x + 1
True
>>> 
```

Decimal places

$125_{10} =$

```
  1 * 100
+ 2 * 10
+ 5 * 1
```

$0.125_{10} =$

```
  1 * 1/10
+ 2 * 1/100
+ 5 * 1/1000
```

$101_2 =$

```
  1 * 4
+ 0 * 2
+ 1 * 1
```

$0.101_2 =$

```
  1 * 1/2
+ 0 * 1/4
+ 1 * 1/8
```

Imprecise decimal places

$1/3_{10} =$

```
  3 * 1/10
+ 3 * 1/100
+ 3 * 1/1000
+ …
```

$\cong 0.333_{10}\ldots$

```
       .333…
3 / 1.000
    -.9
     .100
    -.090
     .010
    -.009
     .001
      …
```

0.3 = not enough
0.4 = too much
0.33 = not enough
0.34 = too much
0.333 = not enough
0.334 = too much
etc

Imprecise decimal places

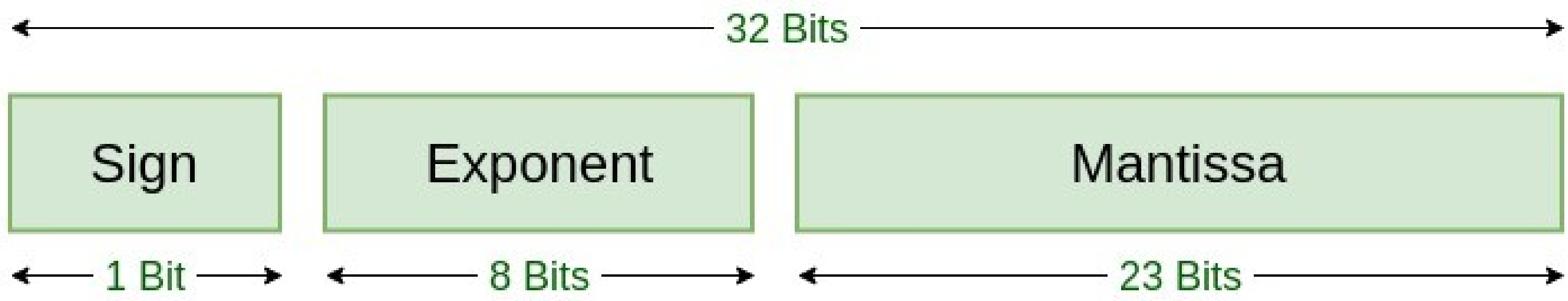
$1/10_{10} =$

```
  0 * 1/2
+ 0 * 1/4
+ 0 * 1/8
+ 1 * 1/16
+ 1 * 1/32
+ 0 * 1/64
+ …
```

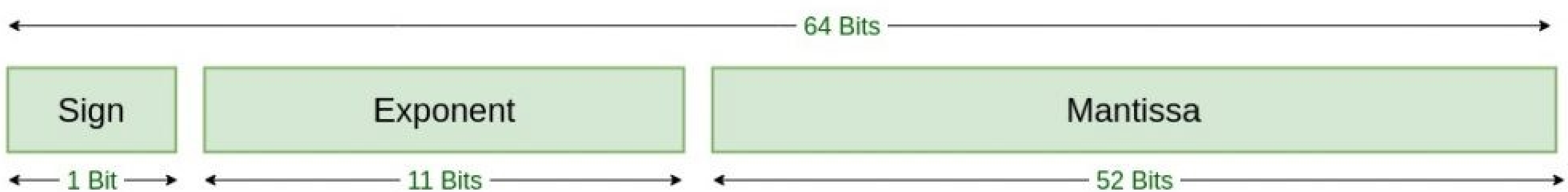
$\cong 0.000110011_2\ldots$

```
          .000110011…
1010 / 1.000000000
      - .1010
        .01100
      - .01010
        .00010
        …
```

$0.1_2 = 1/2_{10}$ = too much
$0.01_2 = 1/4_{10}$ = too much
$0.001_2 = 1/8_{10}$ = too much
$0.0001_2 = 1/16_{10}$ = not enough
$0.00011_2 = 3/32_{10}$ = not enough
$0.000111_2 = 7/64_{10}$ = too much
$0.0001101_2 = 13/128_{10}$ = too much
$0.00011001_2 = 25/256_{10}$ = not enough
$0.000110011_2 = 51/512_{10}$ = not enough
$0.0001100111_2 = 103/1024_{10}$ = too much
etc



Single Precision
IEEE 754 Floating-Point Standard



Double Precision
IEEE 754 Floating-Point Standard

General form of floating point values

$$(-1)^{\mathtt{sign}} * \mathtt{mantissa} * 2^{\mathtt{exponent}}$$

	5/8	1/10
Single-precision (23-bit mantissa)	sign = 0 mantissa = 5_{10} (101_2) exponent = -3	sign = 0 mantissa = 1677721 (110011001100110011001_2) exponent = -24
Double-precision (52-bit mantissa)	sign = 0 mantissa = 5_{10} (101_2) exponent = -3	sign = 0 mantissa = 3602879701896397 ($11001100110011001100110011001100$ $110011001100110011001101_2$) exponent = -55

Binary representations may not be exact!

sign = 0
mantissa = 5_{10} (101_2)
exponent = -3

sign = 0
mantissa = 1677721 (110011001100110011001_2)
exponent = -24

sign = 0
mantissa = 3602879701896397 ($1100110011001100110011001100110011001100110011001101_2$)
exponent = -55

```
>>> 5 * 2**-3
0.625
>>> 1677721 * 2**-24
0.09999996423721313
>>> 3602879701896397 * 2**-55
0.1
>>> 
```

Binary representations may not be exact!

sign = 0
mantissa = 5_{10} (101_2)
exponent = -3

sign = 0
mantissa = 1677721 (110011001100110011001_2)
exponent = -24

sign = 0
mantissa = 3602879701896397 (110
exponent = -55

Exact representation

Approximate representation

```
>>> 5 * 2**-3
0.625
>>> 1677721 * 2**-24
0.0999999642372131З
>>> 3602879701896397 * 2**-55
0.1
>>>
```

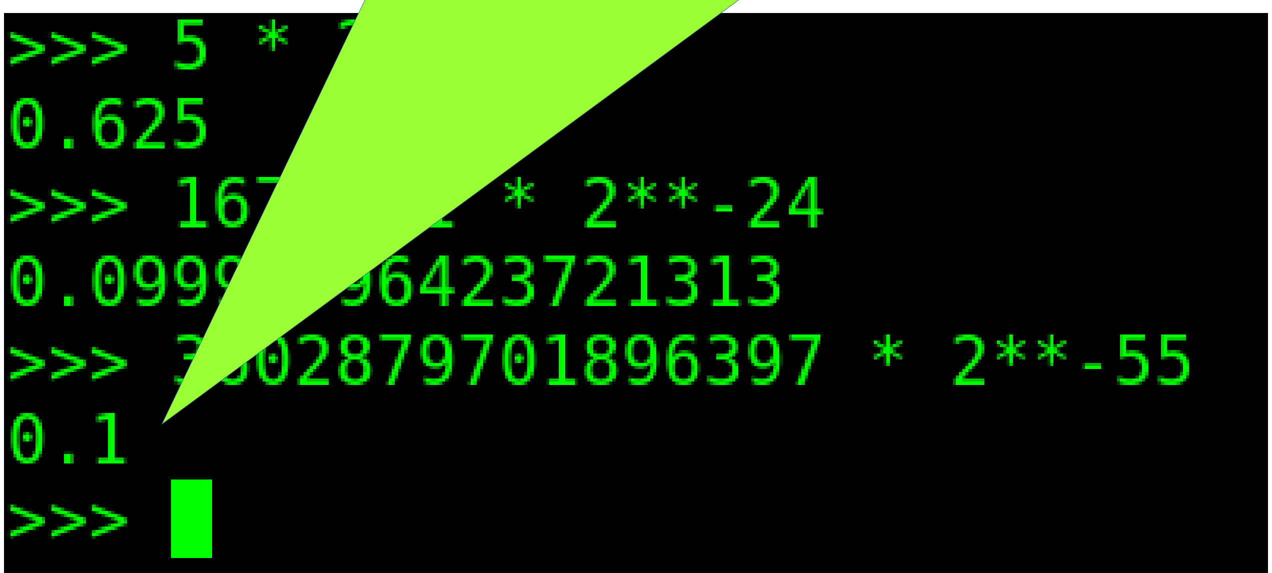
Binary representations may not be exact!

sign = 0
mantissa = 5_{10} (101_2)
exponent = -3

sign
man
exp

Approximate representation,
automatically rounded so it *looks* exact.
But the computed value is not exactly equal to 0.1.
Computed value is
0.1000000000000000055511151231257827021181583404541015625

sign = 0
mantissa = 3602879 … 101_2)
exponent = -55



Math with floats

0.1 + 0.1 + 0.1

⇕

3602879701896397*2**-55 + 3602879701896397*2**-55 + 3602879701896397*2**-55

⇕

0.3000000000000000166533453693773481063544750213623046875

Is that a good approximation of 0.3?

Binary conversion

$3/10_{10} =$

```
  0 * 1/2
+ 1 * 1/4
+ 0 * 1/8
+ 0 * 1/16
+ 1 * 1/32
+ 1 * 1/64
+ …
```

$\cong$

$0.0100110011_2\ldots$

```
           .01001
1010 / 11.000000000
        -10.10
           .10000
        -  .01010
           .00010
           …
```

$0.1_2 = 1/2_{10}$ = too much
$0.01_2 = 1/4_{10}$ = not enough
$0.011_2 = 3/8_{10}$ = too much
$0.0101_2 = 5/16_{10}$ = too much
$0.01001_2 = 9/32_{10}$ = not enough
$0.010011_2 = 19/64_{10}$ = not enough
$0.0100111_2 = 39/128_{10}$ = too much
$0.01001101_2 = 77/256_{10}$ = too much
$0.010011001_2 = 153/512_{10}$ = not enough
etc

	0.3	**0.1 + 0.1 + 0.1**
Double-precision (52-bit mantissa)	sign = 0 mantissa = 5404319552844595_{10} ($10011001100110011001100110011001100110011001100110011_2$) exponent = -54	sign = 0 mantissa = 1351079888211149_{10} ($1001100110011001100110011001100110011001100110011001101_2$) exponent = -52

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

Floaty McFloatface says: remember kids, never use floating point numbers to represent money!

Binary conversion

$500000000000000000000000000.0_{10}$ =

$11001110110010111000111100100111111 1$
$01000010000000000111100111010000000 0$
000000000000000000_2

What's the mantissa? What's the exponent?

Binary conversion

$500000000000000000000000000.0_{10} =$

$110011101100101110001111001001111111$
$01000010000000001111001110100000000$
000000000000000000_2

$500000000000000000000000001.0_{10} =$

$110011101100101110001111001001111111$
$01000010000000001111001110100000000$
000000000000000001_2

Binary conversion

$5000000000000000000000000000.0_{10}$ =

$110011101100101110001111001001111111$
$010000100000000001111001110100000000$
0000000000000000000_2

The mantissa is finite: this is *not* a repeating fractional value.

Nevertheless, there are more than 52 bits in the mantissa, so we need to truncate them for single and double precision.

	5000000000000000000000000000	5000000000000000000000000001
Single-precision (23-bit mantissa)	sign = 0 mantissa = 6776263_{10} = $11001110110010111000111_2$ exponent = 66	sign = 0 mantissa = 6776263_{10} = $11001110110010111000111_2$ exponent = 66
Double-precision (52-bit mantissa)	sign = 0 mantissa = 3637978807091712_{10} = 1100111011001011100011110010011 111110100001000000000_2 exponent = 37	sign = 0 mantissa = 3637978807091712_{10} = 11001110110010111000111100100111 11110100001000000000_2 exponent = 37

```
>>> x = 500000000000000000000000000.0
>>> x == x + 1
True
```

Floaty McFloatface says: floats don't have unlimited significant digits!

Normalization

These values are equivalent:

```
12.8 * 2**-2
6.4 * 2**-1
3.2 * 2**0
1.6 * 2**1
0.8 * 2**2
0.4 * 2**3
```

A *normalized* float value is one whose mantissa is in the range [1,2). That is, the mantissa is *x*, such that $1 \le x < 2$.

Normalization does not change the value of the float, it merely changes its representation.

Of the above equally-valued numbers, only one representation is normalized.

Single precision:

1 bit	8 bits	23 bits
Sign 0 for positive, 1 for negative	Exponent, biased by +127	Mantissa, normalized fractional component

Double precision:

1 bit	11 bits	52 bits
Sign 0 for positive, 1 for negative	Exponent, biased by +1023	Mantissa, normalized fractional component

$0.625_{10} =$

```
  1 * 1/2
+ 0 * 1/4
+ 1 * 1/8
```

$= 0.101_2$

In normalized form: 1.01_2 * 2**-1

Sign = positive
Normalized mantissa = 1.01
Normalized mantissa's fractional component = .01
Exponent = -1
Biased exponent = 126

1 bit	8 bits	23 bits
Sign 0 for positive, 1 for negative	Exponent, biased by +127	Mantissa, normalized fractional component
0	126	01

```
sign  exponent  mantissa
   0  01111110  01000000000000000000000

00111111001000000000000000000000
```

$5.3_{10} =$

```
   1 * 4
+ 0 * 2
+ 1 * 1
+ 0 * 1/2
+ 1 * 1/4
+ 0 * 1/8
+ 0 * 1/16
+ 1 * 1/32
+ 1 * 1/64
+ ...
```

$\cong 101.0100110011..._2$

In normalized form: $1.010100110011..._2 * 2**2$

Sign = positive
Normalized mantissa = 1.010100110011
Normalized mantissa's fractional component
= .010100110011
Exponent = 2
Biased exponent = 129

1 bit	8 bits	23 bits
Sign 0 for positive, 1 for negative	Exponent, biased by +127	Mantissa, normalized fractional component
0	**129**	**010100110011...**

```
sign   exponent   mantissa
   0   10000001   01010011001100110011001

01000000101010011001100110011001
```

- What is the decimal value of the following single-precision floating point number, given in binary?

1 10000001 11100000000000000000000

1 10000001 11100000000000000000000

1 bit	8 bits	23 bits
Sign 0 for positive, 1 for negative	Exponent, biased by +127	Mantissa, normalized fractional component
1	**129**	**11100000000000000000000**

Sign = negative
Normalized mantissa's fractional component = .111
Normalized mantissa = 1.111
Biased exponent = 129
Exponent = 2

In normalized form: -1.111_2 * 2**2
= -(1 + (1/2) + (1/4) + (1/8)) * 2**2
= -7.5

We want to convert 3.2 to single precision IEEE 754.

((-1)**0) * 1.10011001100110011001101 * (2**(128-127))

```
= 1
+ 1 * (1/2)
+ 0 * (1/4)
+ 0 * (1/8)
+ 1 * (1/16)
+ 1 * (1/32)
+ 0 * (1/64)
+ 0 * (1/128)
+ 1 * (1/256)
+ 1 * (1/512)
+ ....                = 1.60000002384185791015625
```

```
sign  exponent  mantissa
  0   10000000  10011001100110011001101

 01000000010011001100110011001101
```

We want to convert 3.2 to single precision IEEE 754.

((-1)**0) * 1.10011001100110011001101 * (2**(128-127))

```
= 1.60000002384185791015625 * 2
= 3.2000000476837158203125
```

This is not an exact representation!
Can we do better?

((-1)**0) * 1.10011001100110011001100 * (2**(128-127))

```
= 1.59999990463256835937 5 * 2
= 3.1999998092651367187 5
```

Not really.

```
3.2000000476837158203125-3.1999998092651367187 5
  =2.384185791015625e-07
  =2**(-22)
```

Special cases

sign exponent mantissa

0 00000000 00000000000000000000000

00000000000000000000000000000000

= +0.0

---

sign exponent mantissa

1 00000000 00000000000000000000000

10000000000000000000000000000000

= -0.0

---

sign exponent mantissa

0 11111111 00000000000000000000000

01111111100000000000000000000000

= +Inf

---

sign exponent mantissa

1 11111111 00000000000000000000000

11111111100000000000000000000000

= -Inf

---

sign exponent mantissa

0 11111111 11111111111111111111111

01111111111111111111111111111111

= NaN

For NaN, the sign can be 1 or 0. The mantissa can be anything except all zeros. The exponent must be all ones.

Adjacent floats

```
sign  exponent  mantissa
   0  10000001  10100000000000000000000

   01000000110100000000000000000000
```

= 6.5

```
sign  exponent  mantissa
   0  10000001  10100000000000000000001

   01000000110100000000000000000001
```

= 6.50000047684

These floats are *adjacent*: there is no representable number between them.

Their binary representations are also adjacent.
If we interpret their binary form as ints, the difference is 1.

There is always true: adjacent floats of the same sign are represented in binary as adjacent ints.

Adjacent floats

= 7.99999952316

sign **exponent** **mantissa**

0 **10000001** **11111111111111111111111**

01000000111111111111111111111111

---

sign **exponent** **mantissa**

0 **10000010** **00000000000000000000000**

01000001000000000000000000000000

= 8.0

The adjacent-float rule is true even when the exponent changes.

Adjacent floats

= 0.999999940395

```
sign  exponent  mantissa
   0  01111110  11111111111111111111111
    00111111011111111111111111111111
```

```
sign  exponent  mantissa
   0  01111111  00000000000000000000000
    00111111100000000000000000000000
```

= 1.0

The adjacent-float rule takes into account *bias*, i.e. the exponent is represented as +127 relative to its actual value (+1023 in doubles).

The first number's exponent in -1, stored as 126.
The second number's exponent is 0, stored as 127.

Adjacent floats

= 0.999999940395

```
sign   exponent  mantissa
  0    01111110  11111111111111111111111

  00111111011111111111111111111111
```

```
sign   exponent  mantissa
  0    01111111  00000000000000000000000

  00111111100000000000000000000000
```

= 1.0

Why do floating-point representations use exponent bias? Why not store the exponent in two's complement, i.e. so that -1 is stored as 11111111 instead of 01111110, and 0 is stored as 00000000 instead of 01111111?

Note that if we interpret the binary representations of the above numbers as ints, their order is preserved:

```
00111111011111111111111111111111 = 1065353215
00111111100000000000000000000000 = 1065353216
```

This allows us to correctly sort floats of the same sign even on hardware that doesn't handle floating point numbers. The benefit of such a design is historical.

In an alternative universe without exponent bias, the order would NOT be preserved:

```
01111111111111111111111111111111 = 2147483647
00000000000000000000000000000000 = 0
```

Alternatives to float: int

- Any int will store whole numbers precisely
- Magnitude may be limited
  - In C/C++, ints are limited to the size of machine words
  - In Python, ints have arbitrary number of significant digits, limited only by memory
- Don't use float for money
  - Instead, use int to store a whole number of cents

Alternatives to float: ratios

- Python's `fractions` module stores fractional values as a ratio of ints

```
>>> from fractions import Fraction
>>> Fraction(1, 50) + Fraction(2, 3)
Fraction(103, 150)
>>> Fraction(1, 1000000000)
Fraction(1, 1000000000)
>>>
```

Alternatives to float: fixed-point

- Python's `decimal` module implements a fixed-point data type, which represents fractional values with configurable precision

```
>>> from decimal import Decimal
>>> Decimal('0.1') + Decimal('0.1') + Decimal('0.1')
Decimal('0.3')
>>> decimal.getcontext().prec = 6
>>> Decimal(1) / Decimal(7)
Decimal('0.142857')
>>> decimal.getcontext().prec = 28
>>> Decimal(1) / Decimal(7)
Decimal('0.1428571428571428571428571429')
```

Alternatives to float: fixed-point

- Fixed-point is stored as an integer *i* and has an implicit scaling factor. Here, the scaling factor (i.e. the point position) is 2.
  - 3.14
    - i=314
  - 900
    - i=90000
  - 0.01
    - i=1
- Pros: avoids imprecision of floats
- Cons: less space efficient, especially for very high- and low-magnitude values.
  - With scale of 2, it's impossible store a number of magnitude less than 0.01, even though we have unused bis
- Selecting the scaling factor influences available range: if scaling factor is small, you can't express low-magnitude numbers; if scaling factor is large, you can't express high-magnitude numbers.