15. Floating-Point Arithmetic: Issues and Limitations

Floating-point numbers are represented in computer hardware as base 2 (binary) fractions. For example, the **decimal** fraction "0.625" has value 6/10 + 2/100 + 5/1000, and in the same way the **binary** fraction "0.101" has value 1/2 + 0/4 + 1/8. These two fractions have identical values, the only real difference being that the first is written in base 10 fractional notation, and the second in base 2.

Unfortunately, most decimal fractions cannot be represented exactly as binary fractions. A consequence is that, in general, the decimal floating-point numbers you enter are only approximated by the binary floating-point numbers actually stored in the machine.

The problem is easier to understand at first in base 10. Consider the fraction 1/3. You can approximate that as a base 10 fraction:

0.3

or, better,

0.33

or, better,

0.333

and so on. No matter how many digits you're willing to write down, the result will never be exactly 1/3, but will be an increasingly better approximation of 1/3.

In the same way, no matter how many base 2 digits you're willing to use, the decimal value 0.1 cannot be represented exactly as a base 2 fraction. In base 2, 1/10 is the infinitely repeating fraction

0.0001100110011001100110011001100110011001100110011...

Stop at any finite number of bits, and you get an approximation. On most machines today, floats are approximated using a binary fraction with the numerator using the first 53 bits starting with the most significant bit and with the denominator as a power of two. In the case of 1/10, the binary fraction is "3602879701896397 / 2 ** 55" which is close to but not exactly equal to the true value of 1/10.

Many users are not aware of the approximation because of the way values are displayed. Python only prints a decimal approximation to the true decimal value of the binary approximation stored by the machine. On most machines, if Python were to print the true decimal value of the binary approximation stored for 0.1, it would have to display:

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

That is more digits than most people find useful, so Python keeps the

number of digits manageable by displaying a rounded value instead:

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

Just remember, even though the printed result looks like the exact value of 1/10, the actual stored value is the nearest representable binary fraction.

Interestingly, there are many different decimal numbers that share the same nearest approximate binary fraction. For example, the numbers "0.1" and "0.10000000000000001" and "0.1000000000000000055511151231257827021181583404541015625" are all approximated by "3602879701896397 / 2 ** 55". Since all of these decimal values share the same approximation, any one of them could be displayed while still preserving the invariant "eval(repr(x)) == x".

Historically, the Python prompt and built-in "repr()" function would choose the one with 17 significant digits, "0.10000000000000001". Starting with Python 3.1, Python (on most systems) is now able to choose the shortest of these and simply display "0.1".

Note that this is in the very nature of binary floating point: this is not a bug in Python, and it is not a bug in your code either. You'll see the same kind of thing in all languages that support your hardware's floating-point arithmetic (although some languages may not *display* the difference by default, or in all output modes).

For more pleasant output, you may wish to use string formatting to produce a limited number of significant digits:

```
>>> format(math.pi, '.12g') # give 12 significant digits
'3.14159265359'

>>> format(math.pi, '.2f') # give 2 digits after the point
'3.14'

>>> repr(math.pi)
'3.141592653589793'
```

It's important to realize that this is, in a real sense, an illusion: you're simply rounding the *display* of the true machine value.

One illusion may beget another. For example, since 0.1 is not exactly 1/10, summing three values of 0.1 may not yield exactly 0.3, either:

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

Also, since the 0.1 cannot get any closer to the exact value of 1/10 and 0.3 cannot get any closer to the exact value of 3/10, then pre-rounding with "round()" function cannot help:

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

Though the numbers cannot be made closer to their intended exact values, the "math.isclose()" function can be useful for comparing inexact values:

```
>>> math.isclose(0.1 + 0.1 + 0.1, 0.3)
True
```

Alternatively, the "round()" function can be used to compare rough approximations:

```
>>> round(math.pi, ndigits=2) == round(22 / 7, ndigits=2)
True
```

Binary floating-point arithmetic holds many surprises like this. The problem with "0.1" is explained in precise detail below, in the "Representation Error" section. See Examples of Floating Point Problems for a pleasant summary of how binary floating point works and the kinds of problems commonly encountered in practice. Also see The Perils of Floating Point for a more complete account of other common surprises.

As that says near the end, "there are no easy answers." Still, don't be unduly wary of floating point! The errors in Python float operations are inherited from the floating-point hardware, and on most machines are on the order of no more than 1 part in 2**53 per operation. That's more than adequate for most tasks, but you do need to keep in mind that it's not decimal arithmetic and that every float operation can suffer a new rounding error.

While pathological cases do exist, for most casual use of floating-point arithmetic you'll see the result you expect in the end if you simply round the display of your final results to the number of decimal digits you expect. "str()" usually suffices, and for finer control see the "str.format()" method's format specifiers in Format String Syntax.

For use cases which require exact decimal representation, try using the "decimal" module which implements decimal arithmetic suitable for accounting applications and high-precision applications.

Another form of exact arithmetic is supported by the "fractions" module which implements arithmetic based on rational numbers (so the numbers like 1/3 can be represented exactly).

If you are a heavy user of floating-point operations you should take a look at the NumPy package and many other packages for mathematical and statistical operations supplied by the SciPy project. See <https://scipy.org>.

Python provides tools that may help on those rare occasions when you really *do* want to know the exact value of a float. The "float.as_integer_ratio()" method expresses the value of a float as a fraction:

```
>>> x = 3.14159
>>> x.as_integer_ratio()
```

(3537115888337719, 1125899906842624)

Since the ratio is exact, it can be used to losslessly recreate the
original value:

```
>>> x == 3537115888337719 / 1125899906842624
True
```

The "float.hex()" method expresses a float in hexadecimal (base 16),
again giving the exact value stored by your computer:

```
>>> x.hex()
'0x1.921f9f01b866ep+1'
```

This precise hexadecimal representation can be used to reconstruct the
float value exactly:

```
>>> x == float.fromhex('0x1.921f9f01b866ep+1')
True
```

Since the representation is exact, it is useful for reliably porting
values across different versions of Python (platform independence) and
exchanging data with other languages that support the same format
(such as Java and C99).

Another helpful tool is the "sum()" function which helps mitigate
loss-of-precision during summation. It uses extended precision for
intermediate rounding steps as values are added onto a running total.
That can make a difference in overall accuracy so that the errors do
not accumulate to the point where they affect the final total:

```
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 == 1.0
False
>>> sum([0.1] * 10) == 1.0
True
```

The "math.fsum()" goes further and tracks all of the "lost digits" as
values are added onto a running total so that the result has only a
single rounding. This is slower than "sum()" but will be more
accurate in uncommon cases where large magnitude inputs mostly cancel
each other out leaving a final sum near zero:

```
>>> arr = [-0.10430216751806065, -266310978.67179024, 143401161448607.16,
... -143401161400469.7, 266262841.31058735, -0.003244936839808227]
>>> float(sum(map(Fraction, arr))) # Exact summation with single rounding
8.042173697819788e-13
>>> math.fsum(arr) # Single rounding
8.042173697819788e-13
>>> sum(arr) # Multiple roundings in extended precision
8.042178034628478e-13
>>> total = 0.0
>>> for x in arr:
... total += x # Multiple roundings in standard precision
...
>>> total # Straight addition has no correct digits!
-0.0051575902860057365
```

15.1. Representation Error

This section explains the "0.1" example in detail, and shows how you can perform an exact analysis of cases like this yourself. Basic familiarity with binary floating-point representation is assumed.

Representation error refers to the fact that some (most, actually) decimal fractions cannot be represented exactly as binary (base 2) fractions. This is the chief reason why Python (or Perl, C, C++, Java, Fortran, and many others) often won't display the exact decimal number you expect.

Why is that? 1/10 is not exactly representable as a binary fraction. Since at least 2000, almost all machines use IEEE 754 binary floating-point arithmetic, and almost all platforms map Python floats to IEEE 754 binary64 "double precision" values. IEEE 754 binary64 values contain 53 bits of precision, so on input the computer strives to convert 0.1 to the closest fraction it can of the form *J*/2***N* where *J* is an integer containing exactly 53 bits. Rewriting

1 / 10 ~= J / (2**N)

as

J ~= 2**N / 10

and recalling that *J* has exactly 53 bits (is ">= 2**52" but "< 2**53"), the best value for *N* is 56:

```
>>> 2**52 <= 2**56 // 10 < 2**53
True
```

That is, 56 is the only value for *N* that leaves *J* with exactly 53 bits. The best possible value for *J* is then that quotient rounded:

```
>>> q, r = divmod(2**56, 10)
>>> r
6
```

Since the remainder is more than half of 10, the best approximation is obtained by rounding up:

```
>>> q+1
7205759403792794
```

Therefore the best possible approximation to 1/10 in IEEE 754 double precision is:

7205759403792794 / 2 ** 56

Dividing both the numerator and denominator by two reduces the fraction to:

```
3602879701896397 / 2 ** 55
```

Note that since we rounded up, this is actually a little bit larger
than 1/10; if we had not rounded up, the quotient would have been a
little bit smaller than 1/10. But in no case can it be *exactly*
1/10!

So the computer never "sees" 1/10: what it sees is the exact fraction
given above, the best IEEE 754 double approximation it can get:

```
>>> 0.1 * 2 ** 55
3602879701896397.0
```

If we multiply that fraction by 10**55, we can see the value out to 55
decimal digits:

```
>>> 3602879701896397 * 10 ** 55 // 2 ** 55
1000000000000000055511151231257827021181583404541015625
```

meaning that the exact number stored in the computer is equal to the
decimal value
0.1000000000000000055511151231257827021181583404541015625. Instead of
displaying the full decimal value, many languages (including older
versions of Python), round the result to 17 significant digits:

```
>>> format(0.1, '.17f')
'0.10000000000000001'
```

The "fractions" and "decimal" modules make these calculations easy:

```
>>> from decimal import Decimal
>>> from fractions import Fraction

>>> Fraction.from_float(0.1)
Fraction(3602879701896397, 36028797018963968)

>>> (0.1).as_integer_ratio()
(3602879701896397, 36028797018963968)

>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')

>>> format(Decimal.from_float(0.1), '.17')
'0.10000000000000001'
```