We Need To Talk About Floating Point Arithmetic

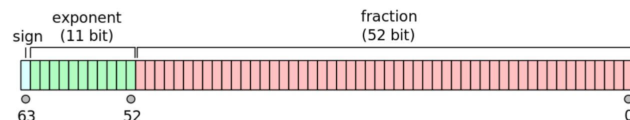
Fact of life #1: Not all real numbers can be expressed exactly as a finite decimal expansion.
So,
`1/2 = 0.5` (exactly)
But,
`1/3 = 0.333333333...`
With finite storage, we can only store a maximum number of these 3's. If we store three of them,
`1/3 ~ (3/10)+(3/100)+(3/1000)=0.333`
and we have only an approximation to `1/3`.

We Need To Talk About Floating Point Arithmetic

Fact of life #2: The same is true for the binary representation of a real number.
So,
`3/4 = (1/2) + (1/4) = 0.11` (in binary, exactly)
But,
`1/10 = 0.00011001100110011...`
an infinitely repeating series. Python stores 53 bits for the mantissa of each floating point number, so this series is truncated. The truncated series, converted back to decimal is:
`1/10 ~ 0.10000000000000009`
This is the *nearest representable number* (using a 53-bit mantissa) to `1/10`.

We Need To Talk About Floating Point Arithmetic

The IEEE-754 Standard for storing 'double-precision' floating point numbers:



64 bits = 8 bytes
The 53-bit mantissa is *normalized* so that it starts with 1. This one isn't stored explicitly, so only 52 bits are needed
$\log_{10}(2^{53}) = 15.95$, so Python's floats can be stored with 15-digit precision

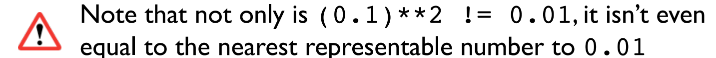
We Need To Talk About Floating Point Arithmetic

Fact of life #3: It is extremely dangerous to compare floating point numbers for equality (or inequality). Consider squaring `1/10`. In Python, the result is:

```
>>> (0.1)**2
0.010000000000000002
```

The exact answer is `1/100 = 0.01` of course, but `0.01` is stored using a 53 mantissa as the equivalent of decimal:
`0.010000000000000009`
The unfortunate consequence is:

```
>>> 0.1**2 == 0.01
False
```


Note that not only is `(0.1)**2 != 0.01`, it isn't even equal to the nearest representable number to `0.01`

We Need To Talk About Floating Point Arithmetic

Fact of life #4: Most floating point operations (and especially addition and subtraction) result in a *loss of significance*. To see why this happens, consider a hypothetical system working in decimal with a 6-digit mantissa. Perform the calculation:
`0.12345678 - 0.123456`
The answer, accurate to 8 digits is `0.00000078`.
On our hypothetical system, however:
`0.123457 - 0.123456 = 0.000001`
Whereas the original numbers are accurate to 6 digits, their floating-point difference is only accurate in its first significant digit: this is an example of *catastrophic cancellation*.

note that it isn't the case that we can't represent the true answer on our machine:
`0.00000078` = 7.8×10^{-7} has only 2 significant digits, well within the 6 available to us.

We Need To Talk About Floating Point Arithmetic

Fact of life #5: Loss of significance can also occur when a small number is subtracted (or added) to a much larger one. For example :
`12345.6 - 0.123456 = 12345.476544` (exactly),
but using 6-digit precision
`12345.6 - 0.123456 = 12345.5`
Repeatedly performing such operations grows the rounding error (possibly by a lot!). For example, in Python:

```
>>> a = 0.
>>> for i in range(10000000):
>>>     a += 0.1
>>> a
999999.9998389754
```

which should be `1000000`, of course.
We're out by over 1.61×10^{-4}.

We Need To Talk About Floating Point Arithmetic

The Patriot Missile defence system was designed to shoot to track and intercept Scud missiles during the first Iraq War

We Need To Talk About Floating Point Arithmetic

The Patriot Missile defence system was designed to shoot to track and intercept Scud missiles during the first Iraq War

Its internal clock counted in an integral number of 1/10ths of a second, which was converted to floating point using a 24-bit register, resulting in an error of `0.000000095` decimal.

We Need To Talk About Floating Point Arithmetic

The Patriot Missile defence system was designed to shoot to track and intercept Scud missiles during the first Iraq War

Its internal clock counted in an integral number of 1/10ths of a second, which was converted to floating point using a 24-bit register, resulting in an error of `0.000000095` decimal.
After the system had been on for 100 hours, the accumulated error was `100*60*60*10*0.000000095 = 0.34` secs. In this time, a Scud missile travels 570 m

We Need To Talk About Floating Point Arithmetic

The Patriot Missile defence system was designed to shoot to track and intercept Scud missiles during the first Iraq War

Its internal clock counted in an integral number of 1/10ths of a second, which was converted to floating point using a 24-bit register, resulting in an error of `0.000000095` decimal.
After the system had been on for 100 hours, the accumulated error was `100*60*60*10*0.000000095 = 0.34` secs. In this time, a Scud missile travels 570 m
On one occasion, the result of this error was that a Patriot battery tracking a Scud was looking for it outside its 'range gate' and failed to intercept it before it hit an American army barracks, killing 28 people.

We Need To Talk About Floating Point Arithmetic

Fact of life #5: An algorithm can be stable (round-off errors tend to cancel) or unstable (round-off errors tend to accumulate). Use stable algorithms.

e.g. find the roots of $ax^2 + bx + c = 0$ $(a \neq 0)$
If $b^2 >> |4ac|$, the formula

$x_\pm = [-b \pm \mathrm{sqrt}(b^2-4ac)]/2a$ is unstable:
if $b < 0$, $-b \approx \mathrm{sqrt}(b^2-4ac)$: rounding error in x_-
if $b > 0$, $b \approx \mathrm{sqrt}(b^2-4ac)$: rounding error in x_+
leading to potential round-off error.

Better:
for $b < 0$, calculate x_+ and then $x_- = c/ax_+$
for $b > 0$, calculate x_- and then $x_+ = c/ax_-$

We Need To Talk About Floating Point Arithmetic

Fact of life #6: Don't think that because you run your algorithms in double precision you're immune to round-off error and loss of precision.

If your algorithm is unstable, it will still come to grief just later rather than sooner ...
... and sometimes not even that much later

We Need To Talk About Floating Point Arithmetic

Further Reading

"What Every Computer Scientist Should Know About Floating-Point Arithmetic": http://floating-point-gui.de/references/

From the Python docs: "Floating Point Arithmetic: Issues and Limitations": http://docs.python.org/tutorial/floatingpoint.html

S. Oliveira and D. Stewart, "Writing Scientific Software: A Guide to Good Style", CUP (2006)

James Gleick on the Ariane 5 launch failure: http://www.around.com/ariane.html

N. J. Higham, "Accuracy and Stability of Numerical Algorithms", 2nd ed., SIAM (2002)

The Zen of Python

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!