Rounding Presentation

December 4, 2017

1 Floating Point Arithematic

1.1 Base-2 Representation

Floating-point numbers are represented in computer hardware as base 2 (binary) fractions. While Decimal Fraction is in base 10. Consider the simple number 0.125, which has 2 different represenations in base-10 and base-2. ##### Base-10

$$0.125 = \frac{1}{10} + \frac{2}{100} + \frac{5}{1000}$$

Base-2

$$0.125 = \frac{0}{2} + \frac{0}{4} + \frac{1}{8}$$

Similar to the representation of $1/3$ in base-10, there are a number of decimals that has infinite representation in base-2 format, like $\frac{1}{10}$ or 0.1. 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.

One of the glaring cases is

$$.1 + .1 + .1! = .3$$

Since the .1 can't get any closer to $1/10$, and .3 can't get any closer to $1/3$, even using the rounding function won't make a difference.

```
In [1]: 0.1 + 0.1 + 0.1 == 0.3
```

```
Out[1]: False
```

1.2 Precision

Almost all machines today use IEEE-754 floating point arithmetic, and almost all platforms map Python floats to IEEE-754 "double precision". 754 doubles 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.

```
In [2]: 0.1.as_integer_ratio()
```

```
Out[2]: (3602879701896397, 36028797018963968)
```

1.3 Problems in Rounding

Given that the floating point is broken, this creates room for many rounding errors in the actual values since the values represented are different from the values stored. Extremly minor erros in percision can change the rounding result thus yielding a wrong value.

One of the most critical and glaring problems is the case is the round half up for $n + 0.5$ where $n \in \mathbb{Z}$. Rounding to the nearest integer would actually floor the number in many cases instead of ceiling it.

```
In [3]: round(.5)

Out[3]: 0

In [4]: round(1.5)

Out[4]: 2
```

1.4 Moving the Rounding Function to the IB tool

We are proposing moving the rounding function from the Python Script to the IB tool for the following reasons: * Consistancey across the Key and Distractors * More robust implementation * Managing the rounding from instance to another * Limiting the parameters file to values and moving the display to the IB tool

1.5 Approach

Our approach is to work on a rounding function with students in mind with the notion that we should do rounding same way students learn to do it at school, giving the proper attention to the decimal places and significant figures.

The following cells show case the decimal module in Python which does just that.

```
In [5]: from decimal import ROUND_UP, Decimal, getcontext
        from sympy import cos, sin, pi

In [6]: getcontext()

Out[6]: Context(prec=28, rounding=ROUND_HALF_EVEN, Emin=-999999, Emax=999999, capitals=1, clamp=

In [7]: getcontext().rounding = 'ROUND_HALF_UP'

In [8]: Decimal('0.5').quantize(Decimal('1'))

Out[8]: Decimal('1')

In [9]: Decimal('4.44444444').quantize(Decimal('1.0'))

Out[9]: Decimal('4.4')

In [10]: Decimal('5.000005').quantize(Decimal('1.00000'))

Out[10]: Decimal('5.00001')
```

```
In [11]: Decimal(-68.44499999999996).quantize(Decimal('1.00'))

Out[11]: Decimal('-68.44')

In [12]: Decimal('-68.44499999999996').quantize(Decimal('1.000'))

Out[12]: Decimal('-68.445')

In [13]: Decimal('-51.005').quantize(Decimal('1.00'))

Out[13]: Decimal('-51.01')

In [14]: Decimal('-51.004999999999974').quantize(Decimal('1.00'))

Out[14]: Decimal('-51.00')

In [15]: cos(pi/3)

Out[15]: 1/2

In [16]: cos(2*pi/3)

Out[16]: -1/2
```