Beancount Precision & Tolerances

[Martin Blais](http://plus.google.com/+MartinBlais), May 2015

<http://furius.ca/beancount/doc/tolerances>

*This document describes how Beancount handles the limited precision of numbers in transaction balance checks and balance assertions. It also documents rounding that may occur in inferring numbers automatically.*

# Motivation

Beancount automatically enforces that the amounts on the Postings of Transactions entered in an input file sum up to zero. In order for Beancount to verify this in a realistic way, it must tolerate a small amount of imprecision. This is because Beancount lets you **replicate what happens in real world account transactions**, and in the real world, institutions round amounts up or down for practical reasons.

Here’s an example: Consider the following transaction which consists in a transfer between two accounts denominated in different currencies (US dollars and Euros):

2015-05-01 \* "Transfer from secret Swiss bank account"  
 Assets:CH:SBS:Checking **-9000.00 CHF**  
 Assets:US:BofA:Checking **9643.82 USD @ 0.93324 CHF**

In this example, the exchange rate used was 0.93324 USD/CHF, that is, 0.93324 Swiss Francs per US dollar. This rate was quoted to 5 digits of precision by the bank. A full-precision conversion of 9000.00 CHF / 0.93324 CHF yields **9643.82152501...** USD. Similarly, converting the US dollars to Francs using the given rate yields an imprecise result as well: 9643.82 x 0.93324 = **8999.9985768…** .

Here is another example where this type of rounding may occur: A transaction for a fractional number of shares of a mutual fund:

2013-04-03 \* "Buy Mutual Fund - Price as of date based on closing price"  
 Assets:US:Vanguard:RGAGX **10.22626 RGAGX {37.61 USD}**  
 Assets:US:Vanguard:Cash **-384.61 USD**

Once again, rounding occurs in this transaction: not only the Net Asset Value of the fund is rounded to its nearest penny value ($37.61), but the number of units is also rounded and accounted for by Vanguard with a fixed number of digits (10.22626 units of VPMBX). And the balance of the entire transaction needs to tolerate some imprecision, whether you compute the value of the shares (10.22626 x $37.61 = **$384.6096386**) or whether you compute the number of shares from the desired dollar amount of the contribution ($384.61 / $37.61 = **10.2262696091**).

From Beancount’s point-of-view, both of the examples above are balancing transactions. Clearly, if we are to try to represent and reproduce the transactions of external accounts to our input file, there needs to be some tolerance in the balance verification algorithm.

# How Precision is Determined

Beancount attempts to derive the precision from each transaction **automatically**, from the input, for each Transaction **in isolation**[[1]](#footnote-0). Let us inspect our last example again:

2013-04-03 \* "Buy Mutual Fund - Price as of date based on closing price"  
 Assets:US:Vanguard:RGAGX **10.22626** RGAGX{37.61 USD}  
 Assets:US:Vanguard:Cash **-384.61** USD

In this transaction, Beancount will infer the tolerance of

* RGAGX at 5 fractional digits, that is, **0.000005 RGAGX**, and
* USD at 2 fractional digits, that is, **0.005 USD**.

Note that the tolerance used is **half of the last digit of precision** provided by the user. This is entirely inferred from the input, without having to fetch any global tolerance declaration. Also note how the precision is calculated **separately for each currency**.

Observe that although we are inferring a tolerance for units of RGAGX, it is actually not used in the balancing of this transaction, because the “weight” of the first posting is in USD (10.22626 x 37.61 = 384.6096386 USD).

So what happens here? The weights of each postings are calculated:

* 384.6096386 USD for the first posting
* -384.61 USD for the second

These are summed together, by currency (there is only USD in the weights of this transaction) which results in a *residual* value of -0.0003614 USD. This value is compared to the tolerance for units of USD: |-0.0003614| < 0.005, and this transaction balances.

## Prices and Costs

For the purpose of inferring the tolerance to be used, the price and cost amounts declared on a transaction’s Postings **are ignored**. This makes sense if you consider that these are usually specified at a higher precision than the base amounts of the postings—and sometimes this extra precision is necessary to make the transaction balance. These should not be used in setting the precision of the whole transaction.

For example, in the following transaction:

1999-09-30 \* "Vest ESPP - Bought at discount: 18.5980 USD"  
 Assets:US:Schwab:ESPP **54 HOOL** {21.8800 USD}  
 Income:CA:ESPP:PayContrib **-1467.84 CAD** @ 0.6842 USD  
 Income:CA:ESPP:Discount **-259.03 CAD** @ 0.6842 USD

The only tolerance inferred here is 0.005 for CAD. (54 HOOL does not yield anything in this case because it is integral; the next section explains this). There is no tolerance inferred for USD, neither from the cost from the first posting (21.8800 USD), nor from the prices of the remaining postings (0.6842 USD).

## Integer Amounts

For integer amounts in the input, the precision is **not** inferred to 0.5, that is, this should fail to balance:

2013-04-03 \* "Buy Mutual Fund - Price as of date based on closing price"  
 Assets:US:Vanguard:RGAGX 10.21005RGAGX{37.61 USD}  
 Assets:US:Vanguard:Cash **-384** USD

In other words, integer amounts do not contribute a number of digits to the determination of the tolerance for their currency.

By default, the tolerance used on amounts without an inferred precision is **zero**. So in this example, because we cannot infer the precision of USD (recall that the cost is ignored), this transaction will fail to balance, because its residual is non-zero (|-0.0003614| > 0).

You can customize what the default tolerance should be for each currency separately and for any currency as well (see section below on how to do this).

This treatment of integer amounts implies that the **maximum amount of precision** that one can specify just by inputting numbers is 0.05 unitsof the currency, for example, by providing a number such as 10.7 as input[[2]](#footnote-1). On the other hand, the settings for the default tolerance to use allows specifying arbitrary numbers.

## Resolving Ambiguities

A case that presents itself rarely is one where multiple different precisions are being input for the same currency. In this case, the **largest** (coarsest)of the inferred input tolerances is used.

For example, if we wanted to track income to more than pennies, we might write this:

1999-08-20 \* "Sell"  
 Assets:US:BRS:ESPP -81 HOOL {26.3125 USD}  
 Assets:US:BRS:Cash **2141.36** USD  
 Expenses:Financial:Fees **0.08** USD  
 Income:CA:ESPP:PnL **-10.125** USD

The amounts we have for USD in this case are 2141.36, 0.08 and -10.125, which infer tolerances of either 0.005 or 0.0005. We select the coarsest amount: this transaction tolerates an imprecision of 0.005 USD.

## Default Tolerances

When a transaction’s numbers do not provide enough information to infer a tolerance *locally*, we fall back to some default tolerance value. As seen in previous examples, this may occur either because (a) the numbers associated with the currency we need it for are integral, or (b) sufficient numbers are simply absent from the input.

By default, this default tolerance is **zero** for all currencies. This can be specified with an option, like this:

option "inferred\_tolerance\_default" "\*:0.001"

The default tolerance can be further refined for each currency involved, by providing the currency to the option, like this:

option "inferred\_tolerance\_default" "USD:0.003"

If provided, the currency-specific tolerance will be used over the global value.

The general form for this option is:

option "inferred\_tolerance\_default" "*<currency>*:*<tolerance>*"

Just to be clear: this option is *only* used when the tolerance cannot be inferred. If you have overly large rounding errors and the numbers in your transactions do infer some tolerance value, this value will be ignored (e.g., setting it to a larger number to try to address that fix will not work). If you need to loosen up the tolerance, see the “inferred\_tolerance\_multiplier” in the next section.

*(Note: I’ve been considering dedicating a special meta-data field to the Commodity directive for this, but this would break from the invariant that meta-data is only there to be used by users and plugins, so I’ve refrained so far.)*

## Tolerance Multiplier

We’re shown previously that when the tolerance value isn’t provided explicitly, that it is inferred from the numbers on the postings. By default, the smallest digit found on those numbers is divided by half to obtain the tolerance because we assume that the institutions which we’re reproducing the transactions apply rounding and so the error should never be more than half.

But in reality, you may find that the rounding errors sometime exceed this value. For this reason, we provide an option to set the multiplier for the inferred tolerance:

option "inferred\_tolerance\_multiplier" "1.2"

This value overrides the default multiplier. In this example, for a transaction with postings only with values such as 24.45 CHF, the inferred tolerance for CHF would be +/- 0.012 CHF.

## Inferring Tolerances from Cost

There is also a feature that expands the maximum tolerance inferred on transactions to include values on cost currencies inferred by postings held at-cost or converted at price. Those postings can imply a tolerance value by multiplying the smallest digit of the unit by the cost or price value and taking half of that value.

For example, if a posting has an amount of "2.345 RGAGX {45.00 USD}" attached to it, it implies a tolerance of 0.001 x 45.00 / 2 = 0.045 USD and the sum of all such possible rounding errors is calculate for all postings held at cost or converted from a price, and the resulting tolerance is added to the list of candidates used to figure out the tolerance we should use for the given commodity (we use the maximum value of all the inferred tolerances).

You turn on the feature like this:

option "infer\_tolerance\_from\_cost" "TRUE"

Enabling this flag only makes the tolerances potentially wider, never smaller.

# Balance Assertions & Padding

There are a few other places where approximate comparisons are needed. Balance assertions also compare two numbers:

2015-05-08 balance Assets:Investments:RGAGX 4.27**1** RGAGX

This asserts that the accumulated balance for this account has 4.271 units of RGAGX, plus or minus 0.001 RGAGX. So accumulated values of 4.270 RGAGX up to 4.272 RGAGX will check as asserted.

The tolerance is inferred automatically to be 1 unit of the least significant digit of the number on the balance assertion. If you wanted a looser assertion, you could have declared:

2015-05-08 balance Assets:Investments:RGAGX **4.27** RGAGX

This assertion would accept values from 4.26 RGAGX to 4.28 RGAGX.

Note that the inferred tolerances are also expanded by the inferred tolerance multiplier discussed above.

## Tolerances that Trigger Padding

Pad directives automatically insert transactions to bring account balances in-line with a subsequent balance assertion. The insertion only triggers if the balance differs from the expected value, and the tolerance for this to occur behaves exactly the same as for balance assertions.

## Explicit Tolerances on Balance Assertions

Beancount supports the specification of an explicit tolerance amount, like this:

2015-05-08 balance Assets:Investments:RGAGX 4.271 **~ 0.01** RGAGX

This feature was added because of some observed peculiarities in Vanguard investment accounts whereby rounding appears to follow odd rules and balances don’t match.

# Saving Rounding Error

As we saw previously, transactions don’t have to balance exactly, they allow for a small amount of imprecision. This bothers some people. If you would like to track and measure the residual amounts allowed by the tolerances, Beancount offers an option to automatically insert postings that will make each transaction balance exactly.

You enable the feature like this:

option "**account\_rounding**" "**Equity:RoundingError**"

This tells Beancount to insert postings to compensate for the rounding error to an “Equity:RoundingError” account. For example, with the feature enabled, the following transaction:

2013-02-23 \* "Buying something"  
 Assets:Invest 1.245 RGAGX {43.23 USD}  
 Assets:Cash -53.82 USD

will be automatically transformed into this:

2013-02-23 \* "Buying something"  
 Assets:Invest 1.245 RGAGX {43.23 USD}  
 Assets:Cash -53.82 USD  
 **Equity:RoundingError -0.00135 USD**

You can verify that this transaction balances exactly. If the transaction already balances exactly (this is the case for most transactions) no posting is inserted.

Finally, if you require that all accounts be opened explicitly, you should remember to declare the rounding account in your file at an appropriate date, like this:

2000-01-01 **open Equity:RoundingError**

# Precision of Inferred Numbers

Beancount is able to infer some missing numbers in the input. For example, the second posting in this transaction is “interpolated” automatically by Beancount:

2014-05-06 \* "Buy mutual fund"  
 Assets:Investments:RGXGX 4.27 RGAGX {53.21 USD}  
 Assets:Investments:Cash

The calculated amount to be inserted from the first posting is -227.2067 USD. Now, you might ask, to which precision is it inserted at? Does it insert 227.2067 USD at the full precision or does the number get rounded to a penny, e.g. 227.21 USD?

It depends on the tolerance inferred for that currency. In this example, no tolerance is able to get inferred (there is no USD amount provided other than the cost amount, which is ignored for the purpose of inferring the tolerance), so we have to defer to the default tolerance.

If the default tolerance is not overridden in the input file—and therefore is zero—the full precision will be used; no rounding occurs. This will result in the following transaction:

2014-05-06 \* "Buy mutual fund"  
 Assets:Investments:RGXGX 4.27 RGAGX {53.21 USD}  
 Assets:Investments:Cash **-227.2067** USD

Note that if a tolerance could be inferred from other numbers on that transaction, it would be used for rounding, such as in this example where the Cash posting is rounded to two digits because of the 9.95 USD number on the Commissions posting:

2014-05-06 \* "Buy mutual fund"  
 Assets:Investments:RGXGX 4.27 RGAGX {53.21 USD}  
 Expenses:Commissions 9.95 USD  
 Assets:Investments:Cash **-237.16** USD

However, if no inference is possible, and the default tolerance for USD is set to 0.001, the number will be quantized to 0.001 before insertion, that is, 227.207 USD will be stored:

option **"default\_tolerance" "USD:0.001"**  
  
2014-05-06 \* "Buy mutual fund"  
 Assets:Investments:RGXGX 4.27 RGAGX {53.21 USD}  
 Assets:Investments:Cash **-227.207** USD

Finally, if you enabled the accumulation of rounding error, the posting’s amount will reflect the correct residual, taking into account the rounded amount that was automatically inserted:

option "default\_tolerance" "USD:0.01"  
option **"account\_rounding" "Equity:RoundingError"**  
  
2014-05-06 \* "Buy mutual fund"  
 Assets:Investments:RGXGX 4.27 RGAGX {53.21 USD}  
 Assets:Investments:Cash -227.207 USD  
 Equity:RoundingError **0.0003** USD

# Porting Existing Input

The inference of tolerance values from the transaction’s numbers is generally good enough to keep existing files working without changes. There may be new errors appearing in older files once we process them with the method described in this document, but they should either point to previously undetected errors in the input, or be fixable with simple addition of a suitable number of digits.

As a testimony, porting the author’s very large input file has been a relatively painless process that took less than 1 hour.

In order to ease the transition, you will probably want to change the default tolerance for all currencies to match the previous value that Beancount had been using, like this:

option "inferred\_tolerance\_default" "\*:0.005"

I would recommend you start with this and fix all errors in your file, then proceed to removing this and fix the rest of errors. This should make it easier to adapt your file to this new behavior.

As an example of how to fix a new error… converting this newly failing transaction from the Integer Amounts section:

2013-04-03 \* "Buy Mutual Fund - Price as of date based on closing price"  
 Assets:US:Vanguard:RGAGX 10.21005RGAGX{37.61 USD}  
 Assets:US:Vanguard:Cash **-384** USD

by inserting zero’s to provide a locally inferred value like this:

2013-04-03 \* "Buy Mutual Fund - Price as of date based on closing price"  
 Assets:US:Vanguard:RGAGX 10.21005RGAGX{37.61 USD}  
 Assets:US:Vanguard:Cash **-384.00** USD

is sufficient to silence the balance check.

# Representational Issues

Internally, Beancount uses a decimal number representation (not a binary/float representation, neither rational numbers). Calculations that result in a large number of fractional digits are carried out to 28 decimal places (the default precision from the context of Python’s IEEE decimal implementation). This is plenty sufficient, because the method we propose above rarely trickles these types of numbers throughout the system: the tolerances allows us to post the precise amounts declared by users, and only automatically derived prices and costs will possibly result in precisions calculated to an unrealistic number of digits that could creep into aggregations in the rest of the system.

# References

The [original proposal](http://furius.ca/beancount/doc/proposal-rounding) that led to this implementation can be [found here](http://furius.ca/beancount/doc/proposal-rounding). In particular, the proposal highlights on the other systems have attempted to deal with this issue. There are also [some discussions](https://groups.google.com/forum/#!msg/ledger-cli/m-TgILbfrwA/YjkmOM3LHXIJ) on the mailing-list dedicated to this topic.

Note that for the longest time, Beancount used a fixed precision of 0.005 across all currencies. This was eliminated once the method described in this document was implemented.

Also, for Balance and Pad directives, there used to be a “tolerance” option that was set by default to 0.015 of any units. This option has been deprecated with the merging of the changes described in this document.

# Historical Notes

Here’s an overview of the status of numbers rendering in Beancount as of March 2016, [from the mailing-list](https://groups.google.com/d/msg/beancount/frfN1zc6TEc/d5OjuDnREgAJ):

First, it's important to realize how these numbers are represented in memory. They are using the Decimal representation which beyond being able to accurately representing decimal numbers (as opposed to the approximation that binary floats provides) also contains a specific precision. That is, the number 2.00 is represented differently than the numbers 2.0 and 2.000. The numbers "remember" which precision they are represented up to. This is important. When I say rendering the numbers to their "natural precision" I mean the precision with which they are represented, i.e., 2.0 renders as "2.0", 2.000 renders as "2.000".

Then, there are two DISTINCT topics: (1) tolerances, and (2) precision.

* "Tolerances" are values used to determine how much imprecision is acceptable in balancing transactions. This is used in the verification stage, to determine how much looseness to allow. It should not affect how numbers are rendered.
* "Precision" is perhaps a bit of a misnomer: By that I'm referring to is how many digits the numbers are to be rendered with.

Once upon a time - after the shell was already written - these concepts weren't well defined in Beancount and I wasn't dealing with these things consistently. At some point it became clear what I needed to do and I created a class called "DisplayContext" which could contain appropriate settings for rendering the precision of numbers for each currency (each currency tends to have its own most common rendering precision, e.g. two digits for USD, one digit for MXN, no digits for JPY and in reports we're typically fine rounding the actual numbers to that precision). So an instance of this DisplayContext is automatically instantiated in the parser and in order to avoid the user having to set these values manually - for Beancount to "do the right thing" by default - it is able to accumulate (<https://bitbucket.org/blais/beancount/src/c349150908078720c7e5ee6bfda644b51ac9543e/src/python/beancount/core/display_context.py?at=default&fileviewer=file-view-default#display_context.py-190>) the numbers seen and to deduce the most common and maximum number of digits used from the input, and to use that as the default number of digits for rendering numbers. The most common format/number of digits is used to render the number of units, and the maximum number of digits seen is used to render costs and prices. In addition, this class also has capabilities for aligning to the decimal dot and to insert commas on thousands as well. It separates the control of the formatting from the numbers themselves.

MOST of the code that renders numbers uses the DisplayContext (via the to\_string() methods) to convert the numbers into strings, such as the web interface and explicit text reports. But NOT ALL... there's a bit of HISTORY here... the SQL shell uses some old special-purpose code (<https://bitbucket.org/blais/beancount/src/c349150908078720c7e5ee6bfda644b51ac9543e/src/python/beancount/query/query_render.py?at=default&fileviewer=file-view-default#query_render.py-166>) to render numbers that I never bothered to convert to the DisplayContext class. There's a TODO item here:<https://bitbucket.org/blais/beancount/src/c349150908078720c7e5ee6bfda644b51ac9543e/TODO?at=default&fileviewer=file-view-default#TODO-312>. It needs to get converted at some point, but I've neglected doing this so far because I have much bigger plans for the SQL query engine that involve a full rewrite of it with many improvements and I figured I'd do that then. If you recall, the SQL query engine was a prototype, and actually it works, but it is not well covered by unit tests. My purpose with it was to discover through usage what would be useful and to then write a v2 of it that would be much better.

Now, about that PRINT command... this is not intended as a reporting tool. The printer's purpose is to print input that accurately represents the content of the transactions. In order to do this, it needs to render the numbers at their "natural" precision, so that when they get read back in, they parse into the very same number, that is, with the same number of digits (even if zeros). For this reason, the PRINT command does not attempt to render using the DisplayContext instance derived from the input file - this is on purpose. I could change that, but then round-trip would break: the rounding resulting from formatting using the display context may output transactions which don't balance anymore.

As you can see, it's not an obvious topic... Hopefully this should allow you to understand what comes out of Beancount in terms of the precision of the numbers it renders.

Note: "default\_tolerances" has been renamed to "inferred\_tolerance\_default" recently because the name was too general and confusing. Old name will work but generate a warning.

I just noticed from your comments and some grepping around that the "render\_commas" option is not used anymore. I'm not sure how that happened, but I'll go ad fix that right away and set the default value of the DisplayContext derived from the input.

I should probably also convert the SQL shell rendering to use the display context regardless of future plans, so that it renders consistently with all the rest. Not sure I can do that this weekend, but I'll log a ticket.

<https://bitbucket.org/blais/beancount/issues/105/context-the-query-rendering-routines-to>

I hope this helps. You're welcome to ask questions if the above isn't clear. I'm sorry if this isn't entirely obvious... there's been a fair bit of history there and there's a lot of code. I should review the naming of options, I think the tolerance options all have "tolerance" in their name, but there aren't options to override the rendering and when I add them they should all have a common name as well.

# Further Reading

[What Every Computer Scientist Should Know About Floating-Point Arithmetic](http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html#689)

1. This stands in contrast to Ledger which attempts to infer the precision based on other transactions recently parsed in the file, in file order. This has the unfortunate effect of creating “cross-talk” between the transactions in terms of what precision can be used. [↑](#footnote-ref-0)
2. Note that due to the way Beancount represents numbers internally, it is also not able to distinguish between “230” and “230.”; these parse into the same representation for Beancount. Therefore, we are not able to use that distinction in the input to support a precision of 0.5. [↑](#footnote-ref-1)