Beancount Documentation

This is the top-level page for all documentation related to Beancount.

http://furius.ca/beancount/doc/index

Documentation for Users

Command-line Accounting in Context: A motivational document that explains what command-line accounting is, *why* I do it, with a simple example of *how* I do it. Read this as an introduction.

The Double-Entry Counting Method: A gentle introduction to the double-entry method, in terms compatible with the assumptions and simplifications that command-line accounting systems make.

Installing Beancount: Instructions for download and installation on your computer, and software release information.

Running Beancount and Generating Reports: How to run the Beancount executables and generate reports with it. This contains technical information for Beancount users.

Getting Started with Beancount: A text that explains the basics of how to create, initialize and organize your input file, and the process of declaring accounts and adding padding directives.

Beancount Language Syntax: A full description of the language syntax with examples. This is the main reference for the Beancount language.

Beancount Options Reference: A description and explanation of all the possible option values.

Precision & Tolerances: Transactions and Balance assertions tolerance small amounts of imprecision which are inferred from the context. This document explains how this works.

Beancount Query Language: A high-level overview of the bean-query command-line client tool that allows you to extract various tables of data from your Beancount ledger file.

Beancount Cheat Sheet: A single page "cheat sheet" that summarizes the Beancount syntax.

How Inventories Work: An explanation of inventories, their representation, and the detail of how lots are matched to positions held in an inventory. This is preliminary reading for the trading doc.

Exporting Your Portfolio: How to export your portfolio to external portfolio tracking websites.

Tutorial & Example: A realistic example ledger file that contains a few years of a hypothetical Beancount user's financial life. This can be used to kick the tires on Beancount and see what reports or its web interface look like. This can give a quick idea of what you can get out of using Beancount.

Beancount Mailing-list: Questions and discussions specific to Beancount occur there. Of related interest is also the Ledger-CLI Forum which contains lots of more general discussions and acts as a meta-list for command-line accounting topics.

Beancount History and Credits: A description of the development history of Beancount and a shout out to its contributors.

A Comparison of Beancount and Ledger & HLedger: A qualitative feature comparison between CLI accounting systems.

Fetching Prices in Beancount: How to fetch and maintain a price database for your assets using Beancount's tools.

Importing External Data: A description of the process of importing data from external files that can be downloaded from financial institutions and the tools and libraries that Beancount provides to automate some of this.

Cookbook & Examples

These documents are examples of using the double-entry method to carry out specific tasks. Use these documents to develop an intuition for how to structure your accounts.

Command-line Accounting Cookbook: Various examples of how to book many different kinds of financial transactions. This is undoubtedly the best way to build an intuition for how to best use the double-entry method.

Trading with Beancount: An explanation of trading P/L and worked examples of how to deal with various investing and trading scenarios with Beancount. This is a complement to the cookbook.

Stock Vesting in Beancount: An example that shows how to deal with restricted stock units and vesting events typical of those offered by technology companies & startups.

Sharing Expenses with Beancount: A realistic and detailed example of using the double-entry method for sharing expenses for a trip or project with other people.

How We Share Expenses: A more involved description of a continuous system for (a) sharing expenses between partners when both are contributing and (b) sharing expenses to a specific project (our son) who has a ledger of his own. This describes our real system.

Documentation for Developers

Beancount Scripting & Plugins: A guide to writing scripts that load your ledger contents in memory and process the directives, and how to write plugins that transform them.

Beancount Design Doc: Information about the program's architecture and design choices, code conventions, invariants and methodology. Read this if you want to get a deeper understanding.

LedgerHub Design Doc: The design and architecture of the importing tools and library implemented in beancount.ingest. This used to be a separate project called LedgerHub (now defunct), whose useful parts have been eventually folded into Beancount. This is the somewhat dated original design doc.

Source Code: The official repository of the Beancount source code lives at BitBucket.

External Contributions: A list of plugins, importers and other codes that build on Beancount's libraries that other people have made and shared.

Enhancement Proposals & Discussions

I occasionally write proposals for enhancements in order to organize and summarize my thoughts before moving forward, and solicit feedback from other users. This is good material to find out what features I'm planning to add, how things work in detail, or compare the differences with other similar software such as Ledger or HLedger.

A Proposal for an Improvement on Inventory Booking: A proposal in which I outline the current state of affairs in Ledger and Beancount regarding inventory booking, and a better method for doing it that will support average cost booking and calculating capital gains without commissions.

Settlement Dates in Beancount: A discussion of how settlement or auxiliary dates could be used in Beancount, and how they are used in Ledger.

Balance Assertions in Beancount: A summary of the different semantics for making balance assertions in Beancount and Ledger and a proposal to extend Beancount's syntax to support more complex assertions.

Fund Accounting with Beancount: A discussion of fund accounting and how best to go about using Beancount to track multiple funds in a single ledger.

Rounding & Precision in Beancount: A discussion of important rounding and precision issues for balance checks and a proposal for a better method to infer required precision. (Implemented)

External Links

Documents, links, blog entries, writings, etc. about Beancount written by other authors.]

Beancount Source Code Documentation (Dominik Aumayr): Sphinx-generated source code documentation of the Beancount codebase. The code to produce this is located here.

Beancount ou la comptabilité pour les hackers (Cyril Deguet): An overview blog entry (in french).

About this Documentation

You may have noticed that I'm using Google Docs. I realize that this is unusual for an open source project. If you take offense to this, so you know, I do like text formats too: in graduate school I used LaTeX extensively, and for the last decade I was in love with the reStructuredText format, I even wrote Emacs' support for it. But something happened around 2013: Google Docs became good enough to write solid technical documentation and I've begun enjoying its revision and commenting facilities extensively: I am hooked, I love it. For users to be able to suggest a correction or place a comment in context is an incredibly useful feature that helps improve the quality of my writing a lot more than patches or comments on a mailing-list. It also gives users a chance to point out passages that need improvement. I have already received orders of magnitude more feedback on documentation than on any of my other projects; it works.

It also looks really *good*, and this is helping motivate me to write more and better. I think maybe I'm falling in love again with WYSIWYG... Don't take me wrong: LaTeX and no-markup formats are great, but the world is changing and it is more important to me to have community collaboration and a living document than a crufty TeX file slowly aging in my repository. My aim is to produce quality text that is easy to read, that prints nicely, and most especially these days, that can render well on a mobile device or a tablet so you can read it anywhere. I'm also able to edit it while I'm on the go, which is fun for jotting down ideas or making corrections. Finally, Google Docs has an API that provides access to the doc, so should it be needed, I could eventually download the meta-data and convert it all to another format. I don't like everything about it, but what it offers, I really do like.

If you want to leave comments, I'd appreciate if you can **log into your Google account** while you read, so that your name appears next to your comment. Thank you!

—Martin Blais

Command-line Accounting in Context

Martin Blais, July 2014

http://furius.ca/beancount/doc/motivation

This text provides context and motivation for the usage of double-entry command-line accounting systems to manage personal finances.

Motivation

When I tell people about my command-line accounting hacking activities, I get a variety of reactions.

Sometimes I hear echoes of desperation about their own finances: many people wish they had a better handle and understanding of their finances, they sigh and wish they were better organized. Other times, after I describe my process I get incredulous reactions: “Why do you even bother?” Those are the people who feel that their financial life is so simple that it can be summarized in 5 minutes.

These are most often blissfully unaware how many accounts they have and have only an imprecise idea of where they stand. They have *at least* 20 accounts to their name; it's only when we go through the detail that they realize that their financial ecosystem is more complex than they thought. Finally, there are those who get really excited about the idea of using a powerful accounting system but who don't actually understand what it is that I'm talking about. They might think it is a system to support investment portfolios, or something that I use to curate a budget like a stickler. Usually they end up thinking it is too complicated to actually get started with.

This document attempts to explain what command-line bookkeeping is about in concrete terms, what you will get out of doing it, how the various software pieces fit together, and what kinds of results you can expect from this method. The fact is, the double-entry method is a basic technique that everyone should have been taught in high school. And using it for yourself is a simple and powerful process you can use to drive your entire financial life.

What exactly is "Accounting"?

When one talks about "accounting," they often implicitly refer to one or more of various distinct financial processes:

- **Bookkeeping.** Recording past transactions in a single place, a "ledger," also called "the books," as in, "the books of this company." Essentially, this means copying the amounts and category of financial transactions that occur in external account statements to a single system that includes all of the "accounts" relating to an entity and links them together. This is colloquially called "keeping books," or the activity of "bookkeeping."
- **Invoices.** Preparing invoices and tracking payments. Contractors will often bring this up because it is a major concern of their activity: issuing requests to clients for payments for services rendered, and checking whether the corresponding payments have actually been received later on (and taking collection actions in case they haven't). If you're managing a company's finances, processing payroll is another aspect of this which is heavy in bookkeeping.
- **Taxes.** Finding or calculating taxable income, filling out tax forms and filing taxes to the various governmental authorities the entity is subject to. This process can be arduous, and for people who are beginning to see an increase in the complexity of their personal assets (many accounts of different types, new reporting requirements), it can be quite stressful. This is often the moment that they start thinking about organizing themselves.
- **Expenses.** Analyzing expenses, basically answering the question: "Where is my money going?" A person with a small income and many credit cards may want to precisely track and calculate how much they're spending every month. This just develops awareness. Even in the

presence of abundant resources, it is interesting to look at how much one is spending regularly, and where most of one's regular income is going, it often brings up surprises.

- **Budgeting.** Forecasting future expenses, allocating limited amounts to spend in various categories, and tracking how close one's actual spending is to those allocations. For individuals, this is usually in the context of trying to pay down debt or finding ways to save more. For companies, this occurs when planning for various projects.
- **Investing.** Summarizing and computing investment returns and capital gains. Many of us now manage our own savings via discount brokers, investing via ETFs and individually selected mutual funds. It is useful to be able to view asset distribution and risk exposure, as well as compute capital gains.
- **Reporting.** Public companies have regulatory requirements to provide transparency to their investors. As such, they usually report an annual *income statement* and a beginning and ending *balance sheet*. For an individual, those same reports are useful when applying for a personal loan or a mortgage at a bank, as they provide a window to someone's financial health situation.

In this document, I will describe how command-line accounting can provide help and support for these activities.

What can it do for me?

You might legitimately ask: sure, but why bother? We can look at the uses of accounting in terms of the questions it can answer for you:

- **Where's my money, Lebowski?** If you don't keep track of stuff, use cash a lot, have too many credit cards or if you are simply a disorganized and brilliant artist with his head in the clouds, you might wonder why there aren't as many beans left at the end of the month as you would like. An income statement will answer this question.
- **I'd like to be like Warren Buffet. How much do I have to save every month?** I personally don't do a budget, but I know many people who do. If you set specific financial goals for a project or to save for later, the first thing you need to do is allocate a budget and set limits on your spending. Tracking your money is the first step towards doing that. You could even compute your returns in a way that can be compared against the market.
- **I have some cash to invest. Given my portfolio, where should I invest it?** Being able to report on the totality of your holdings, you can determine your asset class and currency exposures. This can help you

decide where to place new savings in order to match a target portfolio allocation.

- **How much am I worth?** You have a 401k, an IRA and taxable accounts. A remaining student loan, or perhaps you're sitting on real-estate with two mortgages. Whatever. What's the total? It's really nice to obtain a single number that tells you how far you are from your retirement goals. Beancount can easily compute your net worth (to the cent).

- **I still hurt from 2008. Are my investments safe?** For example, I'm managing my own money with various discount brokers, and many people are. I'm basically running a miniature hedge fund using ETFs, where my goal is to be as diversified as possible between asset classes, sector exposure, currency exposure, etc. Plus, because of tax-deferred accounts I cannot do that in a single place. With Beancount you can generate a list of holdings and aggregate them by category.

- **Taxes suck.** Well, enough said. Why do they suck? Mostly because of uncertainty and doubt. It's not fun to not know what's going on. If you had all the data at your fingertips instantly it wouldn't be nearly as bad. What do you have to report? For some people there's a single stream of income, but for many others, it gets complicated (you might even be in that situation and you don't know it). Qualified vs. ordinary dividends, long-term vs. short-term capital gains, income from secondary sources, wash sales, etc. When it's time to do my taxes, I bring up the year's income statement, and I have a clear list of items to put in and deductions I can make.

- **I want to buy a home. How much can I gather for a down payment?** If you needed a lot of cash all of a sudden, how much can you afford? Well, if you can produce a list of your holdings, you can aggregate them "by liquidity." This gives you an idea of available cash in case of an urgent need.

- **Mr. Banker, please lend me some money.** Impress the banker geek: just bring your balance sheet. Ideally the one generated fresh from that morning. You should have your complete balance sheet at any point in time.

- **Instructions for your eventual passing.** Making a will involves listing your assets. Make your children's lives easier should you pass by having a complete list of all the beans to be passed on or collected. The amounts are just part of the story: being able to list all the accounts and institutions will make it easier for someone to clear your assets.

- **I can't remember if they paid me.** If you send out invoices and have receivables, it's nice to have a method to track who has paid and who just says they'll pay.

I'm sure there is a lot more you can think of. Let me know.

Keeping Books

Alright, so here's the part where I give you the insight of the method. The central and most basic activity that provides support for all the other ones is *bookkeeping*. A most important and fundamental realization is that this relatively simple act—that of copying all of the financial transactions into a single integrated system—allows you to produce reports that solve all of the other problems. You are building a corpus of data, a list of dated transaction objects that each represents movements of money between some of the accounts that you own. This data provides a full timeline of your financial activity.

All you have to do is enter each transaction while respecting a simple constraint, that of the *double-entry method*, which is this:

```
Each time an amount is posted to an account, it must have a corresponding inverse amount pos
```

That's it. This is the essence of the method, and an ensemble of transactions that respects this constraint acquires nice properties which are discussed in detail in some of my other documents. In order to generate sensible reports, all accounts are further labeled with one of four categories: *Assets*, *Liabilities*, *Income* and *Expenses*. A fifth category, *Equity*, exists only to summarize the history of all previous income and expense transactions.

Command-line accounting systems are just simplistic computer languages for conveniently entering, reading and processing this transaction data, and ensure that the constraint of the double-entry method is respected. They're simple counting systems, that can add any kind of "thing" in dedicated counters ("accounts"). They're basically calculators with multiple buckets. Beancount and Ledger may differ slightly in their syntax and on some of their semantics, but the general principle is the same, and the simplifying assumptions are very similar.

Here is a concrete example of what a transaction looks like, just so you can get a feeling for it. In a text file—which is the input to a command-line accounting system—something like the following is written and expresses a credit card transaction:

```
2014-05-23 * "CAFE MOGADOR  NEW YO" "Dinner with Caroline"
  Liabilities:US:BofA:CreditCard    -98.32 USD
  Expenses:Restaurant
```

This is an example transaction with two "postings," or "legs." The "Expenses:Restaurant" line is an account, not a category, though accounts often act like categories (there is no distinction between these two concepts). The amount on the expenses leg is left unspecified, and this is a convenience allowed by the input language. The software determines its amount automatically with the remainder of the balance which in this case will be 98.32 USD, so that -98.32 USD + 98.32 USD = 0 (remember that the sum of all postings must balance to zero—this is enforced by the language).

Most financial institutions provide some way for you to download a summary

of account activity in some format or other. Much of the details of your transactions are automatically pulled in from such a downloaded file, using some custom script you write that converts it into the above syntax:

- A transaction date is always available from the downloadable files.
- The "CAFE MOGADOR NEW YO" bit is the "memo," also provided by the downloaded file, and those names are often good enough for you to figure out what the business you spent at was. Those memos are the same ugly names you would see appear on your credit card statements. This is attached to a transaction as a "payee" attribute.
- I manually added "Dinner with Caroline" as a comment. I don't have to do this (it's optional), but I like to do it when I reconcile new transactions, it takes me only a minute and it helps me remember past events if I look for them.
- The importer brought in the Liabilities:US:BofA:CreditCard posting with its amount automatically, but I've had to insert the Expenses:Restaurant account myself: I typed it in. I have shortcuts in my text editor that allow me to do that using account name completion, it takes a second and it's incredibly easy. Furthermore, pre-selecting this account could be automated as well, by running a simple learning algorithm on the previous history contains in the same input file (we tend to go to the same places all the time).

The syntax gets a little bit more complicated, for example it allows you to represent stock purchases and sales, tracking the cost basis of your assets, and there are many other types of conveniences, like being able to define and count any kind of "thing" in an account (e.g., "vacation hours accumulated"), but this example captures the *essence* of what I do. I replicate all the transactions from all of the accounts that I own in this way, for the most part in an automated fashion. I spend about 1-2 hours every couple of weeks to update this input file, and only for the most used accounts (credit card and checking accounts). Other accounts I'll update every couple of months, or when I need to generate some reports. Because I have a solid understanding of my finances, this is not a burden anymore… it has become *fun*.

What you obtain is a full history, a complete timeline of all the financial transactions from all the accounts over time, often connected together, your financial life, *in a single text file*.

Generating Reports

So what's the point of manicuring this file? I have code that reads it, parses it, and that can serve various views and aggregations of it on a local web server running on my machine, or produce custom reports from this stream of data. The most useful views are undeniably the *balance sheet* and *income statement*, but there are others:

- **Balance Sheet.** A *balance sheet* lists the final balance of all of your assets and liabilities accounts on a single page. This is a **snapshot** of all your accounts at a single point in time, a well-understood overview of your financial situation. This shows your net worth (very precisely, if you update all your accounts) and is also what a banker would be interested in if you were to apply for a loan. Beancount can produce my balance sheet at any point in time, but most often I'm interested in the "current" or "latest" balance sheet.

- **Income Statement.** An *income statement* is a summary of all income and expenses that occur between two points in time. It renders the final balance for these accounts in format familiar to any accountant: income accounts on the left, expense accounts on the right. This provides insights on the changes that occurred during a time period, a **delta** of changes. This tells you how much you're earning and where your money is going, in detail. The difference between the two is how much you saved. Beancount can render such a statement for any arbitrary period of time, e.g., this year, or month by month.

- **Journals.** For each account (or category, if you like to think of them that way), I can render a list of all the transactions that posted changes to that account. I call this a *journal* (Ledger calls this a "register"). If you're looking at your credit card account's journal, for instance, it should match that of your bank statement. On the other hand, if you look at your "restaurants expense" account, it should show all of your restaurant outings, across all methods of payment (including cash, if you choose to enter that). You can easily fetch any detail of your financial history, like "Where was that place I had dinner with Arthur in March three years ago?" or "I want to sell that couch… how much did I pay for it again?"

- **Payables and Receivables.** If you have amounts known to be received, for example, you filed your taxes or sent an invoice and are expecting a payment, or you mailed a check and you are expecting it to be cashed at some point in the future, you can track this with dedicated accounts. Transactions have syntax that allows you to link many of them together and figure out what has been paid or received.

- **Calculating taxes.** If you enter your salary deposits with the detail from your pay stub, you can calculate exactly how much taxes you've paid, and the amounts should match *exactly* those that will be reported on your employer's annual income reporting form (e.g., W2 form in the USA, T4 if in Canada, Form P60 in the UK, or otherwise if you live elsewhere). This is particularly convenient if you have to make tax installments, for instance. If you're a contractor, you will have to track many such things, including company expensable expenses. It is also useful to count dividends that you have to report as taxable income.

- **Investing.** I'm not quite rich, but I manage my own assets using discount

brokers and mainly ETFs. In addition, I take advantage of a variety of tax sheltered accounts, with limited choices in assets. This is pretty common now. You could say that I'm managing a miniature hedge fund and you wouldn't be too far from reality. Using Beancount, I can produce a detailed list of all holdings, with reports on daily market value changes, capital gains, dividends, asset type distribution (e.g. stocks vs. fixed income), currency exposure, and I can figure out how much of my assets are liquid, e.g., how much I have available towards a downpayment on a house. Precisely, and at any point in time. This is nice. I can also produce various aggregations of the holdings, i.e., value by account, by type of instrument, by currency.

- **Forecasting.** The company I work for has an employee stock plan, with a vesting schedule. I can forecast my approximate expected income including the vesting shares under reasonable assumptions. I can answer the question: "At this rate, at which date do I reach a net worth of X?" Say, if X is the amount which you require for retirement.
- **Sharing Expenses.** If you share expenses with others, like when you go on a trip with friends, or have a roommate, the double-entry method provides a natural and easy solution to reconciling expenses together. You can also produce reports of the various expenses incurred for clarity. No uncertainty.
- **Budgeting.** I make enough money that I don't particularly set specific goals myself, but I plan to support features to do this.

You basically end up managing your personal finances like a company would… but it's very easy because you're using a simple and cleverly designed computer language that makes a lot of simplifications (doing away with the concepts of "credit and debits" for example), reducing the process to its essential minimum.

Custom Scripting

The applications are endless. I have all sorts of wild ideas for generating reports for custom projects. These are useful and fun experiments, "challenges" as I call them. Some examples:

- I once owned a condo unit and I've been doing double-entry bookkeeping throughout the period I owned it, through selling it. All of the corresponding accounts share the Loft4530 name in them. This means that I could potentially compute the precise internal rate of return on all of the cash flows related to it, including such petty things as replacement light bulbs expenses. To consider it as a pure investment. Just for fun.
- I can render a tree-map of my annual expenses and assets. This is a good visualization of these categories, that preserve their relative importance.
- I could look at average expenses with a correction for the time-value of

money. This would be fun, tell me how my cost of living has changed over time.

The beauty of it is that once you have the corpus of data, which is relatively easy to create if you maintain it incrementally, you can do all sorts of fun things by writing a little bit of Python code. I built Beancount to be able to do that in two ways:

1. **By providing a "plugins" system** that allows you to filter a parsed set of transactions. This makes it possible for you to hack the syntax of Beancount and prototype new conveniences in data entry. Plugins provided by default provide extended features just by filtering and transforming the list of parsed transactions. And it's simple: all you have to do is implement a callback function with a particular signature and add a line to your input file.

2. **By writing scripts.** You can parse and obtain the contents of a ledger with very little code. In Python, this looks no more complicated than this:

   ```
   import beancount.loader
   ...
   entries, errors, options = beancount.loader.load_file('myfile.ledger')
   for entry in entries:
   ...
   ```

Voila. You're on your way to spitting out whatever output you want. You have access to all the libraries in the Python world, and my code is mostly functional, heavily documented and thoroughly unit-tested. You should be able to find your way easily. Moreover, if you're uncertain about using this system, you could just use it to begin entering your data and later write a script that converts it into something else.

Filing Documents

If you have defined accounts for each of your real-world accounts, you have also created a natural method for organizing your statements and other paper documents. As we are communicating more and more by email with our accountants and institutions, it is becoming increasingly common to scan letters to PDFs and have those available as computer files. All the banks have now gone paperless, and you can download these statements if you care (I tend to do this once at the end of the year, for preservation, just in case). It's nice to be able to organize these nicely and retrieve those documents easily.

What I do is keep a directory hierarchy mirroring the account names that I've defined, something that looks like this:

```
.../documents/
              Assets/
                     US/
```

```
                TDBank/
                       Checking/
                                2014-04-08.statement.march.pdf
                                2014-05-07.statement.april.pdf
                                …
Liabilities/
               US/
                  Amex/
                       Platinum/
                                2014-04-17.March.pdf
                                2014-04-19.download.ofx
                                …
Expenses/
            Health/
                   Medical/
                           2014-04-02.anthem.eob-physiotherapy.pdf
                           …
```

These example files would correspond to accounts with names Assets:US:TDBank:Checking, Liabilities:US:Amex:Platinum, and Expenses:Health:Medical. I keep this directory under version control. As long as a file name begins with a date, such as "2014-04-02", Beancount is able to find the files automatically and insert directives that are rendered in its web interface as part of an account's journal, which you can click on to view the document itself. This allows me to find all my statements in one place, and if I'm searching for a document, it has a well-defined place where I know to find it.

Moreover, the importing software I wrote is able to identify downloaded files and automatically move them into the corresponding account's directory.

How the Pieces Fit Together

So I've described what I do to organize my finances. Here I'll tell you about the various software pieces and how they fit together:

- Beancount is the core of the system. It reads the text file I update, parses the computer language I've defined and produces reports from the resulting data structures. This software *only* reads the input file and does not communicate with other programs on purpose. It runs in isolation.
- Beancount's ingest package and tools help automate the updating of account data by extracting transactions from file downloads from banks and other institutions. These tools orchestrate the running of importers which you implement (this is where all the messy importing code lives, the code you need to make it easier to keep your text file up-to-date, which can parse OFX and CSV files, for instance). See bean-extract, bean-identify tools.

- The ingest package also helps with filing documents (see bean-file tool). Because it is able to identify which document belongs to which account, it can move the downloaded file to my documents archive automatically. This saves me time.
- Finally, in order to provide market values, a Beancount input file should have suitable price directives. Beancount also contains code to fetch latest or historical prices for the various commodities present in one's ledger file (see the bean-price tool). Like the extraction of transactions from OFX downloads, it also spits out Beancount input syntax used to define prices.

See the diagram below for a pretty picture that illustrates how these work together.

Why not just use a spreadsheet?

This is indeed a good question, and spreadsheets are incredibly useful for sure. I certainly would not be writing my own software if I could track my finances with a spreadsheet.

The problem is that the intrinsic *structure* of the double-entry transactional data does not lend itself to a tabular representation. Each transaction has multiple attributes (date, narration, tags), and two or more legs, each of which has an associated amount and possibly a cost. If you put the dates on one axis and the accounts on the other, you would end up with a very sparse and very large table; that would not be very useful, and it would be incredibly difficult to edit. If on the other hand, you had a column dedicated to account names for each row, all of your computations would have to take the account cell into the calculation logic. It would be very difficult to deal with, if not impossible. All matters of data aggregations are performed relative to account names.

Moreover, dealing with the accumulation of units with a cost basis would require difficult gymnastics. I don't even know how I would proceed forward to do this in a spreadsheet. A core part of command-line accounting systems is the inventory logic that allows you to track a cost basis for every unit held in an account and book reductions of positions against existing lots only. This allows the system to compute capital gains automatically. And this is related to the codes that enforce the constraint that all postings on a transaction balance out to zero.

I believe that the "transaction <-> postings" data representation combined with a sensible method for updating inventories is the essence of this system. The need for these is the justification to create a dedicated method to build this data, such as a computer language. Finally, having our own syntax offers the opportunity to provide other types of useful directives such as balance assertions, open/close dates, and sophisticated filtering of subsets of transactions. You just cannot do what we're doing in a spreadsheet.

Why not just use a commercial app?

How about Mint.com?

Oftentimes people tell me they're using Mint.com to track their finances, usually with some amount of specific complaints following close behind. Online services such as Mint provide a subset of the functionality of double-entry accounting. The main focus of these services is the reporting of expenses by category, and the maintenance of a budget. As such, they do a great job at automating the download of your credit card and bank transactions. I think that if you want a low-maintenance option to tracking your finances and you don't have a problem with the obvious privacy risks, this is a *fantastic* option: it comes with a web interface, mobile apps, and updating your accounts probably requires clicking some buttons, providing some passwords, and a small amount of manual corrections every couple of weeks.

I have several reservations about using it for myself, however:

- **Passwords.** I'm just not comfortable enough with any commercial company to share the passwords to *all* my bank, credit card, and investment accounts. This sounds like an insane idea to me, a very scary one, and I'm just not willing to put that much trust in any one company's hands. I don't care what they say: I worked for several software companies and I'm aware of the disconnect between promises made by salespeople, website PR, and engineering reality. I also know the power of determined computer hackers.
- **Insufficient reporting.** Reporting is probably beautiful and colorful—it's gorgeous on the website—but certainly insufficient for all the reporting *I* want to do on my data. Reporting that doesn't do what they want it to is a common complaint I hear about these systems. With my system, I can always write a script and produce *any* kind of report I might possibly want in the future. For instance, spitting out a treemap visualization of my expenses instead of a useless pie chart. I can slice and dice my transactions in all kinds of unique ways. I can tag subsets of transactions for projects or trips and report on them. It's more powerful than generic reporting.
- **Perennity.** What if the company is not in business in 5 years? Where will my data be? If I spend any time at all curating my financial data, I want to ensure that it will be available to me forever, in an open format. In their favor, some of these sites probably have a downloadable option, but who uses it? Does it even work well? Would it include all of the transactions that they downloaded? I don't know.
- **Not international.** It probably does not work well with an international, multi-currency situation. These services target a "majority" of users, most of which have all their affairs in a single country. I live in the USA, have a past history in Canada, which involves remaining tax-sheltered investment

accounts, occasional expenses during visits which justify maintaining a credit card there, and a future history which might involve some years spent in another country such as Hong Kong or Australia. Will this work in Mint? Probably not. They might support Canada, but will they support accounts in both places? How about some other country I might want to move to? I want a single integrated view of all my accounts across all countries in all currencies forever, nothing less. My system supports that very well. (I'm not aware of any double-entry system that deals with the international problem in a currency agnostic way as well as Beancount does.)

- **Inability to deal with particular situations.** What if I own real estate? Will I be able to price the value of my home at a reasonable amount, so it creates an accurate balance sheet? Regularly obtaining "comparables" for my property from an eager real estate agent will tell me much more precisely how much my home is worth than services like Zillow ever could. I need to be able to input that for my balance sheet. What about those stock options from that privately held Australian company I used to work for? How do I price that? How about other intangible things, such as receivables from a personal loan I made to a friend? I doubt online services are able to provide you with the ability to enter those. If you want the whole picture with precision, you need to be able to make these adjustments.

- **Custom tracking.** Using "imaginary currencies", I'm able to track all kinds of other things than currencies and stocks. For example, by using an "IRAUSD" commodity in Beancount I'm able to track how many 401k contributions I've made at any point during the year. I can count the after-tax basis of a traditional IRA account similarly. I can even count my vacation hours using an imaginary "VACHR" currency and verify it against my pay stubs.

- **Capital gains reporting.** I haven't tried it myself, but I've heard some discontent about Mint from others about its limited capabilities for capital gains reporting. Will it maintain trade lots? Will it handle average cost booking? How about PnL on FOREX gains? What about revenue received for that book I'm selling via PayPal? I want to be able to use my accounting system as an input for my tax reporting.

- **Cash transactions.** How difficult is it to enter cash transactions? Do I have to log in, or start a slow, heavy program that will only work under Windows? With Beancount I bring up a file in a text editor, this is instant and easy. Is it even possible to enter custom cash entries in online services?

In other words, I'm a very sophisticated user and yes, a bit of a *control freak*. I'm not lying about it. I don't have any ideological objections about using a commercial service, it is probably not for me. If it's good enough for you, suit yourself. Despite their beautiful and promising websites, I haven't heard of

anyone being completely satisfied with these services. I hear a fair amount of complaining, some mild satisfaction, but I have yet to meet anyone who raves about it.

How about Quicken?

Quicken is a single-entry system, that is, it replicates the transactions of a remote account locally, and allows you to add a label to each transaction to place it in a category. I believe it also has support for synchronizing investment accounts. This is not enough for me, I want to track all kinds of things, and I want to use the double-entry method, which provides an intrinsic check that I've entered my data correctly. Single-entry accounting is just not good enough if you've already crossed the bridge of understanding the double-entry method.

How about Quickbooks?

So let's talk about sophisticated software that is good enough for a company. Why wouldn't I use that instead? If it's good enough for small businesses, it should be good enough for me, no? There's Quickbooks and other ones. Why don't I use them:

- **It costs money.** Commercial software comes at a price. Ok, I probably could afford to pay a few hundred dollars per year (Quickbooks 2014 looks like around 300$/year for the full set of features), but I don't really want to.
- **Platform.** These softwares usually run on Microsoft Windows and sometimes on Mac OS X. I'm a software developer, I mostly use Linux, and a Macbook Air for a laptop, on which I get annoyed running anything else than tmux and a web browser. I'm not going to reboot just to enter a quick cash transaction.
- **Slow startup.** I cannot speak specifically to Quickbooks' implementation, but virtually every software suite of commercial scope I've had to use had a splash screen and a slow, slow startup that involved initializing tons of plugins. They assume you're going to spend hours in it, which is reasonable for commercial users, but not for me, if I want to do a quick update of my ledger.
- **UIs are inadequate.** I haven't seen their UI but given the nature of transactions and my desire to input precisely and search quickly and organize things, I want to be able to edit in as text. I imagine it would be inconvenient for me. With Emacs and org-mode, I can easily i-search my way to any transaction within seconds after opening my ledger file.
- **Inflexible.** How would I go about re-organizing all my account names? I think I'm still learning about the double-entry bookkeeping method and I have made mistakes in the past, mistakes where I desire to revisit the way I organize my accounts in a hierarchy. With my text file, I was able

to safely rename a large number of accounts several times, and evolve my chart-of-accounts to reflect my improving understanding of how to organize my financial tracking system. Text is powerful!

How about GnuCash?

I don't like UIs; they're inconvenient. There's nothing quite like editing a text file if you are a programmer. Moreover, GnuCash does not deal with multiple currencies well. I find bugs in it within the first hour every time I kick the tires on it, which I do every couple of years, out of curiosity. Other programs, such as Skrooge, also take the heavy-handed big UI approach.

Why build a computer language?

A bookkeeping system provides conditions for a solution that involves a simple computer language for many reasons.

Single-entry bookkeeping is largely insufficient if you're trying to track everything holistically. Existing systems either limit themselves to expense categories with little checking beyond "reconciling" which sometimes involves freezing the past. If you're not doing the bookkeeping for a company, sometimes just changing the past and fixing the mistakes where they occurred makes more sense. More importantly, the single-entry method leaves us wanting for the natural error-checking mechanism involved in the double-entry system.

The problem is also not solvable elegantly by using spreadsheets; the simple data structure that forms the basis of the double-entry system infers either a very sparse spreadsheet with accounts on one dimension and transactions on the other. For real-world usage, this is impractical. Another iteration on this theme would involve inserting the postings with two columns, one with the account and one with the amount, but the varying number of columns and the lookup code makes this inelegant as well. Plus, it's not obvious how you would deal with a large number of currencies.

Programs that provide fancy graphical or web-based user interfaces are inevitably awkward, due to the nature of the problem: each transaction is organized by viewing it through the lens of one account's journal, but any of the accounts present in its postings provide equally valid views. Ideally, what you want, is just to look at the transaction. Organizing them for most convenient input has little to do with the order in which they are to be presented. Using text has a lot of advantages:

- You can easily used search-and-replace and/or sed to make global changes, for example, rename or reorganize your accounts;
- You can organize the transactions in the order that is most convenient for data entry;
- There are a number of existing tools to search the text;

- You can easily write various little tools to spit out the data syntax, i.e., for importing from other file types, or converting from other systems.
- Text is inherently **open**, that is the file format is one that you can read your data from and store anywhere else, not a blob of incomprehensible data that becomes unusable when the company that makes the software stops supporting it.

Finally, systems that attempt to automate the process of importing all your data from automated sources (e.g., mint.com) have one major shortfall: most often it's not very easy or even possible to add information for accounts that aren't automated. It is my experience that in practice you will have some entries and accounts to track that will not have a nice downloadable file format, or that simply don't have a real-world counterpart. In order to produce a complete view of one's balance sheet, it is important to be able to enter all of an individual's account within a single system. In other words, custom accounts and manually entered transactions do matter a lot.

For all the reasons mentioned above, I feel that a computer language is more appropriate to express this data structure than a heavy piece of software with a customized interface. Being able to easily bring up a text file and quickly type in a few lines of text to add a transaction is great–it's fast and easy. The ledger file provides a naturally open way to express one's data, and can be source-controlled or shared between people as well. Multiple files can be merged together, and scripted manipulations on a source file can be used to reorganize one's data history. Furthermore, a read-only web interface that presents the various reports one expects and allows the user to explore different views on the dataset is sufficient for the purpose of viewing the data.

Advantages of Command-Line Bookkeeping

In summary, there are many advantages to using a command-line accounting system over a commercial package that provides a user-interface:

- **Fast.** You don't have to fire up a slow program with a splash screen that will take a while to initialize in order to add something to your books. Bringing up a text file from a bookmark in your favorite editing program (I use Emacs) is easy and quick. And if you're normally using a text editor all day long, as many programmers do, you won't even blink before the file is in front of your eyes. It's quick and easy.
- **Portable.** It will work on all platforms. Beancount is written in Python 3 with some C extensions, and as such will work on Mac, Linux and Windows platforms. I am very careful and determined to keep external dependencies on third-party packages to an absolute minimum in order to avoid installation problems. It should be easy to install and update, and work everywhere the same.
- **Openness.** Your data is open, and will remain open *forever*. I plan on

having my corpus of data until the day I die. With an open format you will never end up in a situation where your transactional data is sitting in a binary blob with an unknown format and the software goes unsupported. Furthermore, your data can be converted to other languages easily. You can easily invoke the parser I provide and write a script that spits it out in another program's input syntax. You can place it under version control. You can entirely reorganize the structure of your accounts by renaming strings with sed. Text is empowering.

- **Customized.** You can produce very customized reports, that address *exactly* the kinds of problems you are having. One complaint I often hear from people about other financial software is that it doesn't quite do what they want. With command-line accounting systems you can at least write it yourself, as a small extension that uses your corpus of data.

Why not just use an SQL database?

I don't like to reinvent the wheel. If this problem could be solved by filling up an SQL database and then making queries on it, that's exactly what I would do. Creating a language is a large overhead, it needs to be maintained and evolved, it's not an easy task, but as it turns out, necessary and justified to solve this problem.

The problem is due to a few reasons:

- Filtering occurs on a two-level data structure of transactions vs. their children postings, and it is inconvenient to represent the data in a single table upon which we could then make manipulations. You cannot simply work only with postings: in order to ensure that the reports balance, you need to be selecting complete transactions. When you select transactions, the semantics is "select all the postings for which the transaction has *this* or *that* property." One such property is "all transactions that have some posting with account *X*." These semantics are not obvious to implement with a database. The nature of the data structure makes it inconvenient.

- The wide variety of directives makes it difficult to design a single elegant table that can accommodate all of their data. Ideally we would want to define two tables for each type of directive: a table that holds all common data (date, source filename & lineno, directive type) and a table to hold the type-specific data. While this is possible, it steps away from the neat structure of a single table of data.

- Operations on inventories—the data structure that holds the incrementally changing contents of accounts—require special treatment that would be difficult to implement in a database. Lot reductions are constrained against a running inventory of lots, each of which has a specific cost basis. Some lot reductions trigger the merging of lots (for average cost booking). This requires some custom operations on these inventory objects.

- Aggregations (breakdowns) are hierarchical in nature. The tree-like structure of accounts allows us to perform operations on subtrees of postings. This would also not be easy to implement with tables.

Finally, input would be difficult. By defining a language, we side-step the problem of having to build a custom user interface that would allow us to create the data.

Nevertheless, I really like the idea of working with databases. A script (bean-sql) is provided to convert the contents of a Beancount file to an SQL database; it's not super elegant to carry out computations on those tables, but it's provided nonetheless. You should be able to play around with it, even if operations are difficult. I might even pre-compute some of the operation's outputs to make it easier to run SQL queries.

But... I just want to do *X*?

Some may infer that what we're doing must be terribly complicated given that they envision they might want to use such a system only for a single purpose. But the fact is, how many accounts you decide to track is a personal choice. You can choose to track as little as you want in as little detail as you deem sufficient. For example, if all that you're interested in is investing, then you can keep books on only your investment accounts. If you're interested in replacing your usage of Mint.com or Quicken, you can simply just replicate the statements for your credit cards and see your corresponding expenses.

The simplicity of having to just replicate the transactions in all your accounts over doing a painful annual "starting from scratch" evaluation of all your assets and expenses for the purpose of looking at your finances will convince you. Looking at your finances with a spreadsheet will require you to at least copy values from your portfolio. Every time you want to generate the report you'll have to update the values… with my method, you just update each account's full activity, and you can obtain the complete list holdings as a by-product. It's easy to bring everything up-to-date if you have a systematic method.

To those I say: try it out, accounting just for the bits that you're interested in. Once you'll have a taste of how the double-entry method works and have learned a little bit about the language syntax, you will want to get a fuller picture. You will get sucked in.

Why am I so Excited?

Okay, so I've become an accounting nerd. I did not ask for it. Just kind-of happened while I wasn't looking (I still don't wear brown socks though). Why is it I can't stop talking about this stuff?

I used to have a company. It was an umbrella for doing contract work (yes, I originally had bigger aspirations for it, but it ended up being just that. Bleh.)

As a company owner in Canada, I had to periodically make five different kinds of tax installments to the Federal and Provincial governments, some every month, some every three months, and then it varied. An accountant was feeding me the "magic numbers" to put in the forms, numbers which I was copying like a monkey, without fully understanding how they were going to get used later on. I had to count separately the various expenses I would incur in relation to my work (for deductions), and those which were only personal —I did not have the enlightened view of getting separate credit cards so I was trying to track personal vs. company expenses from the same accounts. I also had to track transfers between company and personal accounts that would later on get reported as dividends or salary income. I often had multiple pending invoices that would take more than two months to get paid, from different clients. It was a demi-hell. You get the idea. I used to try to track all these things manually, using systems composed of little text files and spreadsheets and doing things very carefully and in a particular way.

The thing is, although I am by nature quite methodical, I would sometimes, just *sometimes* forget to update one of the files. Things would fall out of sync. When this happened it was incredibly frustrating, I had to spend time digging around various statements and figure out where I had gone wrong. This was time-consuming and unpleasant. And of course, when something you have to do is unpleasant, you tend not to do it so well. I did not have a solid idea of the current state of affairs of my company during the year, I just worked and earned money for it. My accountant would draw a balance sheet once a year after doing taxes in July, and it was always a tiny bit of a surprise. I felt permanently somewhat confused, and frankly annoyed every time I had to deal with "moneythings." It wasn't quite a nightmare, but it was definitely making me tired. I felt my life was so much simpler when I just had a job.

But one day… one day… I saw the light: I discovered the double-entry method. I don't recall exactly how. I think it was on a website, yes… this site: dwm-beancounter.com (it's a wonderful site). I realized that using a *single system*, I could account for all of these problems at the same time, and in a way that would impose an inherent error-checking mechanism. I was blown away! I think I printed the whole thing on paper at the time and worked my way through every tutorial. This simple counting trick is what exactly what I was in dire need for.

But all the softwares I tried were either disappointing, broken, or too complicated. So I read up on Ledger. And I got in touch with its author. And then shortly after I started on Beancount[1]. I'll admit that going through the effort of designing my own system just to solve my accounting problems is a bit overkill, but I've been known to be a little more than extreme about certain things, and I've really enjoyed solving this problem. My life is fulfilled when I maintain a good balance of "learning" and "doing," and this falls squarely in the "doing" domain.

Ever since, I feel so excited about anything related to personal finance. Probably

because it makes me so happy to have such a level of awareness about what's going on with mine. I even sometimes find myself *loving* spending money in a new way, just from knowing that I'll have to figure out how to account for it later. I feel so elated by the ability to solve these financial puzzles that I have had bouts of engaging complete weekends in building this software. They are small challenges with a truly practical application and a tangible consequence. Instant gratification. I get a feeling of *empowerment*. And I wish the same for you.

[1] For a full list of differences, refer to this document.

The Double-Entry Counting Method

Martin Blais, December 2016

http://furius.ca/beancount/doc/double-entry

The Table Perspective

Introduction

This document is a gentle introduction to the double-entry counting method, as written from the perspective of a computer scientist. It is an attempt to explain basic bookkeeping using as simple an approach as possible, doing away with some of the idiosyncrasies normally involved in accounting. It is also representative of how Beancount works, and it should be useful to all users of plain-text accounting.

Note that I am not an accountant, and in the process of writing this document I may have used terminology that is slightly different or unusual to that which is taught in perhaps more traditional training in accounting. I granted myself license to create something new and perhaps even unusual in order to explain those ideas as simply and clearly as possible to someone unfamiliar with them.

I believe that the method of double-entry counting should be taught to everyone at the high school level everywhere as it is a tremendously useful organizational skill, and I hope that this text can help spread its knowledge beyond professional circles.

Basics of Double-Entry Bookkeeping

The double-entry system is just a simple *method of counting*, with some simple rules.

Let's begin by defining the notion of an **account**. An account is something that can contain things, like a bag. It is used to count things, to accumulate things. Let's draw a horizontal arrow to visually represent the evolving contents of an account over time:

On the left, we have the past, and to the right, increasing time: the present, the future, etc.

For now, let's assume that accounts can contain only one kind of thing, for example, *dollars*. All accounts begin with an empty content of zero dollars. We will call the number of units in the account the **balance** of an account. Note that it represents its contents at a particular point in time. I will draw the balance using a number above the account's timeline:

The contents of accounts can change over time. In order to change the content of an account, we have to add something to it. We will call this addition a **posting** to an account, and I will draw this change as a circled number on the account's timeline, for example, adding $100 to the account:

Now, we can draw the updated balance of the account after the posting with another little number right after it:

The account's balance, after adding $100, is now $100.

We can also remove from the contents of an account. For example, we could remove $25, and the resulting account balance is now $75:

Account balances can also become *negative*, if we remove more dollars than there are in the account. For example, if we remove $200 from this account, the balance now becomes $-125:

It's perfectly fine for accounts to contain a negative balance number. Remember that all we're doing is counting things. As we will see shortly, some accounts will remain with a negative balance for most of their timeline.

Statements

Something worthy of notice is how the timeline notation I've written in the previous section is analogous to paper account statements institutions maintain for each client and which you typically receive through the mail:

Sometimes the amount column is split into two, one showing the positive amounts and the other the negative ones:

Here, "debit" means "removed from your account" and "credit" means "deposited in your account." Sometimes the words "withdrawals" and "deposits" will be used. It all depends on context: for checking and savings accounts it is usual to have both types of postings, but for a credit card account typically it shows only positive numbers and then the occasional monthly payment so the single column format is used.

In any case, the "balance" column always shows the resulting balance *after* the amount has been posted to the account. And sometimes the statements are rendered in decreasing order of time.

Single-Entry Bookkeeping

In this story, this account belongs to someone. We'll call this person the **owner** of the account. The account can be used to represent a real world account, for example, imagine that we use it to represent the content of the owner's checking account at a bank. So we're going to label the account by giving it a name, in this case "Checking":

Imagine that at some point, this account has a balance of $1000, like I've drawn on the picture. Now, if the owner spends $79 of this account, we would represent it like this:

Furthermore, if the expense was for a meal at a restaurant, we could flag the posting with a **category** to indicate what the change was used for. Let's say, "Restaurant", like this:

Now, if we have a lot of these, we could write a computer program to accumulate all the changes for each category and calculate the sums for each of them. That

would tell us how much we spent in restaurants in total, for example. This is called the **single-entry method** of accounting.

But we're not going to do it this way; we have a better way. Bear with me for a few more sections.

Double-Entry Bookkeeping

An owner may have multiple accounts. I will represent this by drawing many similar account timelines on the same graphic. As before, these are labeled with unique names. Let's assume that the owner has the same "Checking" account as previously, but now also a **"Restaurant"** account as well, which can be used to accumulate all food expenses at restaurants. It looks like this:

Now, instead of *categorizing* the posting to a "restaurant category" as we did previously, we could create a matching posting on the "Restaurant" account to record how much we spent for food, with the amount spent ($79):

The "Restaurant" account, like all other accounts, also has an accumulated balance, so we can find out how much we spent in "Restaurant" in total. This is entirely symmetrical to counting changes in a checking account.

Now, we can associate the two postings together, by creating a kind of "parent" box that refers to both of them. We will call this object a **transaction**:

Notice here that we've also associated a description to this transaction: "Dinner at Uncle Boons". A transaction also has a **date**, and all of its postings are recorded to occur on that date. We call this the transaction date.

We can now introduce the fundamental rule of double-entry bookkeeping system:

```
The sum of all the postings of a transaction must equal zero.
```

Remember this, as this is the foundation of the double-entry method, and its most important characteristic. It has important consequences which I will discuss later in this document.

In our example, we remove $79 from the "Checking" account and "give it" to the "Restaurant" account. ($79) + ($-79) = $0. To emphasize this, I could draw a little summation line under the postings of the transaction, like this:

Many Accounts

There may be many such transactions, over many different accounts. For example, if the owner of the accounts had a lunch the next day which she paid using a credit card, it could be represented by creating a "Credit Card" account dedicated to tracking the real world credit card balance, and with a corresponding transaction:

In this example, the owner spent $35 at a restaurant called "Eataly." The previous balance of the owner's credit card was $-450; after the expense, the

new balance is $-485.

For each real world account, the owner can create a bookkeeping account like we did. Also, for each category of expenditure, the owner also creates a bookkeeping account. In this system, there are no limits to how many accounts can be created.

Note that the balance in the example is a negative number; this is not an error. Balances for credit card accounts are normally negative: they represent an amount *you owe*, that the bank is lending you *on credit*. When your credit card company keeps track of your expenses, they write out your statement from their perspective, as positive numbers. For you, those are amounts you need to eventually pay. But here, in our accounting system, we're representing numbers from the owner's point-of-view, and from her perspective, this is money she owes, not something she has. What we have is a meal sitting in our stomach (a positive number of $ of "Restaurant").

Multiple Postings

Finally, transactions may have more than two postings; in fact, they may have any number of postings. The only thing that matters is that the sum of their amounts is zero (from the rule of double-entry bookkeeping above).

For example, let's look at what would happen if the owner gets her salary paid for December:

Her gross salary received in this example is recorded as $-2,905 (I'll explain the sign in a moment). $905 is set aside for taxes. Her "net" salary of $2,000, the remainder, is deposited in her "Checking" account and the resulting balance of that account is $2,921 (the previous balance of $921 + $2,000 = $2,921). This transaction has three postings: (+2,000) + (-2,905) + (+905) = 0. The double-entry rule is respected.

Now, you may ask: Why is her salary recorded as a negative number? The reasoning here is similar to that of the credit card above, though perhaps a bit more subtle. These accounts exist to track all the amounts from the owner's point-of-view. The owner gives out work, and receives money and taxes in exchange for it (positive amounts). The work given away is denominated in dollar units. It "leaves" the owner (imagine that the owner has *potential work* stored in her pocket and as she goes into work every day sprinkles that work potential giving it to the company). The owner *gave* $2,905's worth of work away. We want to track how much work was given, and it's done with the "Salary" account. That's her gross salary.

Note also that we've simplified this paycheck transaction a bit, for the sake of keeping things simple. A more realistic recording of one's pay stub would have many more accounts; we would separately account for state and federal tax amounts, as well as social security and medicare payments, deductions, insurance paid through work, and vacation time accrued during the period. But

it wouldn't be much more complicated: the owner would simply translate all the amounts available from her pay stub into a single transaction with more postings. The structure remains similar.

Types of Accounts

Let's now turn our attention to the different types of accounts an owner can have.

Balance or Delta. First, the most important distinction between accounts is about whether we care about the balance **at a particular point** in time, or whether it only makes sense to care about differences **over a period** of time. For example, the balance of someone's Checking or Savings accounts is a meaningful number that both the owner and its corresponding bank will care about. Similarly, the total amount owed on someone's Credit Card account is also meaningful. The same goes with someone's remaining Mortgage amount to pay on a house.

On the other hand, the total amount of Restaurant expenses since the beginning of somebody's life on earth is not particularly interesting. What we might care about for this account is the amount of Restaurant expenses incurred *over a particular period of time*. For example, "how much did you spend in restaurants last month?" Or last quarter. Or last year. Similarly, the total amount of gross salary since the beginning of someone's employment at a company a few years ago is not very important. But we would care about the total amount earned during a tax year, that is, for that time period, because it is used for reporting one's income to the tax man.

- Accounts whose balance at a point in time is meaningful are called **balance sheet accounts**. There are two types of such accounts: "**Assets**" and "**Liabilities**."
- The other accounts, that is, those whose balance is not particularly meaningful but for which we are interested in calculating changes over a period of time are called **income statement accounts**. Again, there are two kinds: "**Income**" and "**Expenses**."

Normal sign. Secondly, we consider *the usual sign of an account's balance*. The great majority of accounts in the double-entry system tend to have a balance with always a positive sign, or always a negative sign (though as we've seen previously, it is not impossible that an account's balance could change signs). This is how we will distinguish between the pairs of accounts mentioned before:

- For a balance sheet account, Assets normally have positive balances, and Liabilities normally have negative balances.
- For income statement accounts, Expenses normally have a positive balance, and Income accounts normally have a negative balance.

This situation is summarized in the following table:

Let's discuss each type of account and provide some examples, so that it doesn't remain too abstract.

- **Assets. (+)** Asset accounts represent *something the owner has*. A canonical example is banking accounts. Another one is a "cash" account, which counts how much money is in your wallet. Investments are also assets (their units aren't dollars in this case, but rather some number of shares of some mutual fund or stock). Finally, if you own a home, the home itself is considered an asset (and its market value fluctuates over time).

- **Liabilities. (-)** A liability account represents *something the owner owes*. The most common example is a credit card. Again, the statement provided by your bank will show positive numbers, but from your own perspective, they are negative numbers. A loan is also a liability account. For example, if you take out a mortgage on a home, this is money you owe, and will be tracked by an account with a negative amount. As you pay off the mortgage every month the negative number goes up, that is, its absolute value gets smaller and smaller over time (e.g., -120,000 -> -117,345).

- **Expenses. (+)** An expense account represents *something you've received*, perhaps by exchanging something else to purchase it. This type of account will seem pretty natural: food, drinks, clothing, rent, flights, hotels and most other categories of things you typically spend your disposable income on. However, taxes are also typically tracked by an expense account: when you receive some salary income, the amount of taxes withheld at the source is recorded immediately as an expense. Think of it as paying for government services you receive throughout the year.

- **Income. (-)** An income account is used to count *something you've given away* in order to receive something else (typically assets or expenses). For most people with jobs, that is the value of their time (a salary income). Specifically, here we're talking about the *gross* income. For example, if you're earning a salary of $120,000/year, that number is $120,000, not whatever amount remains after paying for taxes. Other types of income includes dividends received from investments, or interest paid from bonds held. There are also a number of oddball things received you might record as income, such the value of rewards received, e.g., cash back from a credit card, or monetary gifts from someone.

In Beancount, all account names, without exception, must be associated to one of the types of accounts described previously. Since the type of an account never changes during its lifetime, we will make its type a part of an account's name, as a *prefix*, by convention. For example, the qualified account name for restaurant will be "Expenses:Restaurant". For the bank checking account, the qualified account name will be "Assets:Checking".

Other than that, you can select any name you like for your accounts. You can create as many accounts as you like, and as we will see later, you can organize

them in a hierarchy. As of the writing of this document, I'm using more than 700 accounts to track my personal affairs.

Let us now revisit our example and add some more accounts:

And let's imagine there are more transactions:

… and even more of them:

Finally, we can label each of those accounts with one of the four types of accounts by prepending the type to their account names:

A realistic book from someone tracking all of their personal affairs might easily contain thousands of transactions per year. But the principles remain simple and they remain the same: postings are applied to accounts over time, and must be parented to a transaction, and within this transaction the sum of all the postings is zero.

When you do **bookkeeping** for a set of accounts, you are essentially describing all the postings that happen on all the accounts over time, subject to the constraint of the rule. You are creating a database of those postings in a **book**. You are "keeping the book," that is, traditionally, the book which contains all those transactions. Some people call this "maintaining a journal."

We will now turn our attention to obtaining useful information from this data, summarizing information from the book.

Trial Balance

Take our last example: we can easily reorder all the accounts such that all the Asset accounts appear together at the top, then all the Liabilities accounts, then Income, and finally Expenses accounts. We are simply changing the order without modifying the structure of transactions, in order to group each type of accounts together:

We've reordered the accounts with Assets accounts grouped at the top, then Liabilities, then some Equity accounts (which we have just introduced, more about them is discussed later), then Income and finally Expenses at the bottom.

If we sum up the postings on all of the accounts and render just the account name and its final balance on the right, we obtain a report we call the "trial balance."

This simply reflects the balance of each account at a particular point in time. And because each of the accounts began with a zero balance, and each transaction has itself a zero balance, we know that the sum of all those balances must equal zero.[1] This is a consequence of our constraining that each of the postings be part of a transaction, and that each transaction have postings that balance each other out.

Income Statement

One kind of common information that is useful to extract from the list of transactions is a summary of changes in income statement accounts during a particular period of time. This tells us how much money was earned and spent during this period, and the difference tells us how much profit (or loss) was incurred. We call this the "net income."

In order to generate this summary, we simply turn our attention to the balances of the accounts of types Income and Expenses, summing up just the transactions for a particular period, and we draw the Income balances on the left, and Expenses balances on the right:

It is important to take note of the signs here: Income numbers are negative, and Expenses numbers are positive. So if you earned more than you spent (a good outcome), the final sum of Income + Expenses balances will be a negative number. Like any other income, a net income that has a negative number means that there is a corresponding amount of Assets and/or Expenses with positive numbers (this is good for you).

An Income Statement tells us what changed during a particular period of time. Companies typically report this information **quarterly** to investors and perhaps the public (if they are a publicly traded company) in order to share how much profit they were able to make. Individuals typically report this information on their **annual** tax returns.

Clearing Income

Notice how in the income statement only the transactions within a particular interval of time are summed up. This allows one, for instance, to compute the sum of all income earned during a year. If we were to sum up all of the transactions of this account since its inception we would obtain the total amount of income earned since the account was created.

A better way to achieve the same thing is to zero out the balances of the Income and Expenses accounts. Beancount calls this basic transformation "clearing[2]." It is carried out by:

1. Computing the balances of those accounts from the beginning of time to the start of the reporting period. For example, if you created your accounts in year 2000 and you wanted to generate an income statement for year 2016, you would sum up the balances from 2000 to Jan 1, 2016.

2. Inserting transactions to empty those balances and transfer them to some other account that isn't Income nor Expenses. For instance, if the restaurant expense account for those 16 years amounts to $85,321 on Jan 1, 2016, it would insert a transaction of $-85,321 to restaurants and $+85,321 to "previous earnings". The transactions would be dated Jan 1, 2016. Including this transaction, the sum of that account would zero on that date.

This is what we want.

Those transactions inserted for all income statement accounts are pictured in green below. Now summing the entire set of transactions through the end of the ledger would yield only the changes during year 2016 because the balances were zero on that date:

This is the semantics of the "CLEAR" operation of the bean-query shell.

(Note that another way to achieve the same thing for income statement accounts would be to segregate and count amounts only for the transactions after the clearing date; however, jointly reporting on income statement accounts and balance sheet accounts would have incorrect balances for the balance sheet accounts.)

Equity

The account that receives those previously accumulated incomes is called "Previous Earnings". It lives in a fifth and final type of accounts: **Equity**. We did not talk about this type of accounts earlier because they are most often only used to transfer amounts to build up reports, and the owner usually doesn't post changes to those types of accounts; the software does that automatically, e.g., when clearing net income.

The account type "equity" is used for accounts that hold a summary of the net income implied by all the past activity. The point is that if we now list together the Assets, Liabilities and Equity accounts, because the Income and Expenses accounts have been zero'ed out, the sum total of all these balances should equal exactly zero. And summing up all the Equity accounts clearly tells us what's our stake in the entity, in other words, if you used the assets to pay off all the liabilities, how much is left in the business… how much it's worth.

Note that the normal sign of the Equity accounts is *negative*. There is no particular meaning to that, just that they are used to counterbalance Assets and Liabilities and if the owner has any value, that number should be negative. (A negative Equity means some positive net worth.)

There are a few different Equity accounts in use in Beancount:

- **Previous Earnings** or **Retained Earnings.** An account used to hold the sum total of Income & Expenses balances from the beginning of time until the *beginning* of a reporting period. This is the account we were referring to in the previous section.
- **Current Earnings**, also called **Net Income.** An account used to contain the sum of Income & Expenses balances incurred *during* the reporting period. They are filled in by "clearing" the Income & Expenses accounts *at the end* of the reporting period.
- **Opening Balances.** An equity account used to counterbalance deposits used to initialize accounts. This type of account is used when we trun-

cate the past history of transactions, but we also need to ensure that an account's balance begins its history with a particular amount.

Once again: you don't need to define nor use these accounts yourself, as these are created for the purpose of summarizing transactions. Generally, the accounts are filled in by the clearing process described above, or filled in by Pad directives to "opening balances" equity accounts, to account for summarized balances from the past. They are created and filled in automatically by the software. We'll see how these get used in the following sections.

Balance Sheet

Another kind of summary is a listing of the owner's assets and debts, for each of the accounts. This answers the question: "*Where's the money?*" In theory, we could just restrict our focus to the Assets and Liabilities accounts and draw those up in a report:

However, in practice, there is another closely related question that comes up and which is usually answered at the same time: "*Once all debts are paid off, how much are we left with?*" This is called the **net worth**.

If the Income & Expenses accounts have been cleared to zero and all their balances have been transferred to Equity accounts, the net worth should be equal to the sum of all the Equity accounts. So in building up the Balance Sheet, it it customary to clear the net income and then display the balances of the Equity accounts. The report looks like this:

Note that the balance sheet can be drawn for *any point in time*, simply by truncating the list of transactions following a particular date. A balance sheet displays a snapshot of balances at one date; an income statement displays the difference of those balances between two dates.

Summarizing

It is useful to summarize a history of past transactions into a single equivalent deposit. For example, if we're interested in transactions for year 2016 for an account which has a balance of $450 on Jan 1, 2016, we can delete all the previous transactions and replace them with a single one that deposits $450 on Dec 31, 2015 and that takes it from somewhere else.

That somewhere else will be the Equity account **Opening Balances**. First, we can do this for all Assets and Liabilities accounts (see transactions in blue):

Then we delete all the transactions that precede the opening date, to obtain a truncated list of transactions:

This is a useful operation when we're focused on the transactions for a particular interval of time.

(This is a bit of an implementation detail: these operations are related to how Beancount is designed. Instead of making all the reporting operations with parameters, all of its reporting routines are simplified and instead operate on the entire stream of transactions; in this way, we convert the list of transactions to include only the data we want to report on. In this case, summarization is just a transformation which accepts the full set of transactions and returns an equivalent truncated stream. Then, from this stream, a journal can be produced that excludes the transactions from the past.

From a program design perspective, this is appealing because the only state of the program is a stream of transactions, and it is never modified directly. It's simple and robust.)

Period Reporting

Now we know we can produce a statement of changes over a period of time, by "clearing" and looking at just the Income & Expenses accounts (the Income Statement). We also know we can clear to produce a snapshot of Assets, Liabilities & Equity at any point in time (the Balance Sheet).

More generally, we're interested in inspecting a particular period of time. That implies an income statement, but also *two* balance sheet statements: the balance sheet *at the beginning* of the period, and the balance sheet *at the end* of the period.

In order to do this, we apply the following transformations:

- **Open.** We first clear net income at the beginning of the period, to move all previous income balances to the Equity **Previous Earnings** account. We then summarize up to the beginning of the period. We call the combination of clearing + summarizing: "Opening."
- **Close.** We also truncate all the transactions following the end of the reporting period. We call this operation "Closing."

These are the meaning of the "OPEN" and "CLOSE" operations of the beanquery shell[3]. The resulting set of transactions should look like this.

"Closing" involves two steps. First, we remove all transactions following the closing date:

We can process this stream of transactions to produce an Income Statement for the period.

Then we clear again at the *end* date of the desired report, but this time we clear the net income to "Equity:Earnings:Current":

From these transactions, we produce the Balance Sheet at the end of the period.

This sums up the operations involved in preparing the streams of transactions to produce reports with Beancount, as well as a basic introduction to those types

of reports.

Chart of Accounts

New users are often wondering how much detail they should use in their account names. For example, should one include the payee in the account name itself, such as in these examples?

```
Expenses:Phone:Mobile:VerizonWireless
Assets:AccountsReceivable:Clients:AcmeInc
```

Or should one use simpler names like the following, relying instead on the "payee", "tags", or perhaps some other metadata in order to group the postings?

```
Expenses:Phone
Assets:AccountsReceivable
```

The answer is that *it depends on you*. This is an arbitrary choice to make. It's a matter of taste. Personally I like to abuse the account names a bit and create long descriptive ones, other people prefer to keep them simple and use tags to group their postings. Sometimes one doesn't even need to filter subgroups of postings. There's no right answer, it depends on what you'd like to do.

One consideration to keep in mind is that account names implicitly define a hierarchy. The ":" separator is interpreted by some reporting code to create an in-memory tree and can allow you to collapse a node's children subaccounts and compute aggregates on the parent. Think of this as an additional way to group postings.

Country-Institution Convention

One convention I've come up with that works well for my assets, liabilities and income accounts is to root the tree with a code for the country the account lives in, followed by a short string for the institution it corresponds to. Underneath that, a unique name for the particular account in that institution. Like this:

```
<type> : <country> : <institution> : <account>
```

For example, a checking account could be chosen to be "Assets:US:BofA:Checking", where "BofA" stands for "Bank of America." A credit card account could include the name of the particular type of card as the account name, like "Liabilities:US:Amex:Platinum", which can be useful if you have multiple cards.

I've found it doesn't make sense for me to use this scheme for expense accounts, since those tend to represent generic categories. For those, it seems to make more sense to group them by category, as in using "Expenses:Food:Restaurant" instead of just "Expenses:Restaurant".

In any case, Beancount doesn't enforce anything other than the root accounts; this is just a suggestion and this convention is not coded anywhere in the soft-

ware. You have great freedom to experiment, and you can easily change all the names later by processing the text file. See the Cookbook for more practical guidance.

Credits & Debits

At this point, we haven't discussed the concepts of "credits" and "debits." This is on purpose: Beancount largely does away with these concepts because it makes everything else simpler. I believe that it is simpler to just learn that the signs of Income, Liabilities and Equity accounts are normally negative and to treat all accounts the same way than to deal with the debits and credits terminology and to treat different account categories differently. In any case, this section explains what these are.

As I have pointed out in previous sections, we consider "Income", "Liabilities" and "Equity" accounts to normally have a negative balance. This may sound odd; after all, nobody thinks of their gross salary as a negative amount, and certainly your credit-card bill or mortgage loan statements report positive numbers. This is because in our double-entry accounting system we consider all accounts to be held *from the perspective of the owner of the account*. We use signs consistent from this perspective, because it makes all operations on account contents straightforward: they're all just simple additions and all the accounts are treated the same.

In contrast, accountants traditionally keep all the balances of their accounts as positive numbers and then handle postings to those accounts differently depending on the account type upon which they are applied. The sign to apply to each account is entirely dictated by its type: Assets and Expenses accounts are debit accounts and Liabilities, Equity and Income accounts are credit accounts and require a sign adjustment. Moreover, posting a positive amount on an account is called "crediting" and removing from an account is called "debiting." See this external document, for example, which nearly makes my head explode. This way of handling postings makes everything much more complicated than it needs to be.

The problem with this approach is that summing of amounts over the postings of a transaction is not a straightforward sum anymore. For example, let's say you're creating a new transaction with postings to two Asset accounts, an Expenses account and an Income account and the system tells you there is a $9.95 imbalance error somewhere. You're staring at the entry intently; which of the postings is too small? Or is one of the postings too large? Also, maybe a new posting needs to be added, but is it to a debit account or to a credit account? The mental gymnastics required to do this are taxing. Some double-entry accounting software tries to deal with this by creating separate columns for debits and credits and allowing the user enter an amount only in the column that corresponds to each posting account's type. This helps visually, but why not just use signs instead?

Moreover, when you look at the accounting equations, you have to consider their signs as well. This makes it awkward to do transformations on them and make what is essentially a simple summation over postings into a convoluted mess that is difficult to understand.

In plain-text accounting, we would rather just do away with this inconvenient baggage. We just use additions everywhere and learn to keep in mind that Liabilities, Equity and Income accounts normally have a negative balance. While this is unconventional, it's much easier to grok. And If there is a need to view a conventional report with positive numbers only, we will be able to trigger that in reporting code[4], inverting the signs just to render them in the output.

Save yourself some pain: Flush your brain from the "debit" and "credit" terminology.

Accounting Equations

In light of the previous sections, we can easily express the accounting equations in signed terms. If,

- A = the sum of all Assets postings
- L = the sum of all Liabilities postings
- X = the sum of all Expenses postings
- I = the sum of all Income postings
- E = the sum of all Equity postings

We can say that:

```
A + L + E + X + I = 0
```

This follows from the fact that

```
sum(all postings) = 0
```

Which follows from the fact that each transaction is guaranteed to sum up to zero (which is enforced by Beancount):

```
for all transactions t, sum(postings of t) = 0
```

Moreover, the sum of postings from Income and Expenses is the Net Income (NI):

```
NI = X + I
```

If we adjust the equity to reflect the total Net Income effect by clearing the income to the Equity retained earnings account, we get an updated Equity value (E'):

```
E' = E + NI = E + X + I
```

And we have a simplified accounting equation:

```
A + L + E' = 0
```

If we were to adjust the signs for credits and debits (see previous section) and have sums that are all positive number, this becomes the familiar accounting equation:

```
Assets - Liabilities = Equity
```

As you can see, it's much easier to just always add up the numbers.

Plain-Text Accounting

Ok, so now we understand the method and what it can do for us, at least in theory. The purpose of a double-entry bookkeeping system is to allow you to replicate the transactions that occur in various real world accounts into a single, unified system, in a common representation, and to extract various views and reports from this data. Let us now turn our attention to how we record this data in practice.

This document talks about Beancount, whose purpose is "double-entry bookkeeping using text files." Beancount implements a parser for a simple syntax that allows you to record transactions and postings. The syntax for an example transaction looks something like this:

```
2016-12-06 * "Biang!" "Dinner"
  Liabilities:CreditCard   -47.23 USD
  Expenses:Restaurants
```

You write many of declarations like these in a file, and Beancount will read it and create the corresponding data structures in memory.

Verification. After parsing the transactions, Beancount also verifies the rule of the double-entry method: it checks that the sum of the postings on all your transactions is zero. If you make a mistake and record a transaction with a non-zero balance, an error will be displayed.

Balance Assertions. Beancount allows you to replicate balances declared from external accounts, for example, a balance written on a monthly statement. It processes those and checks that the balances resulting from your input transactions match those declared balances. This helps you detect and find mistakes easily.

Plugins. Beancount allows you to build programs which can automate and/or process the streams of transactions in your input files. You can build custom functionality by writing code which directly processes the transaction stream.

Querying & Reporting. It provides tools to then process this stream of transactions to produce the kinds of reports we discussed earlier in this document.

There are a few more details, for example, Beancount allows you to track cost basis and make currency conversions, but that's the essence of it.

The Table Perspective

Almost always, questions asked by users on the mailing-list about how to calculate or track some value or other can be resolved easily simply by thinking of the data as a long list of rows, some of which need to be filtered and aggregated. If you consider that all that we're doing in the end is deriving "sums" of these postings, and that the attributes of transactions and postings are what allows us to filter subsets of postings, it always becomes very simple. In almost all the cases, the answer is to find some way to disambiguate postings to select them, e.g. by account name, by attaching some tag, by using some metadata, etc. It can be illuminating to consider how this data can be represented as a table.

Imagine that you have two tables: a table containing the fields of each Transaction such as date and description, and a table for the fields of each Posting, such as account, amount and currency, as well as a reference to its parent transaction. The simplest way to represent the data is to **join** those two tables, replicating values of the parent transaction across each of the postings.

For example, this Beancount input:

```
2016-12-04 * "Christmas gift"
  Liabilities:CreditCard       -153.45 USD
  Expenses:Gifts

2016-12-06 * "Biang!" "Dinner"
  Liabilities:CreditCard   -47.23 USD
  Expenses:Restaurants

2016-12-07 * "Pouring Ribbons" "Drinks with friends"
  Assets:Cash     -25.00 USD
  Expenses:Tips     4.00 USD
  Expenses:Alcohol
```

could be rendered as a table like this:

Notice how the values of Transaction fields are replicated for each posting. This is exactly like a regular database join operation. The posting fields begin at column "Account." (Also note that this example table is simplified; in practice there are many more fields.)

If you had a joined table just like this you could filter it and sum up amounts for arbitrary groups of postings. This is exactly what the bean-query tool allows you to do: You can run a SQL query on the data equivalent to this in-memory table and list values like this:

```
SELECT date, payee, number WHERE account = "Liabilities:CreditCard";
```

Or sum up positions like this:

```
SELECT account, sum(position) GROUP BY account;
```

This simple last command generates the trial balance report.

Note that the table representation does not inherently constrain the postings to sum to zero. If your selection criteria for the rows (in the WHERE clause) always selects ***all*** the postings for each of the matching transactions, you are ensured that the final sum of all the postings is zero. If not, the sum may be anything else. Just something to keep in mind.

If you're familiar with SQL databases, you might ask why Beancount doesn't simply process its data in order to fill up an existing database system, so that the user could then use those database's tools. There are two main reasons for this:

- **Reporting Operations.** In order to generate income statements and balance sheets, the list of transactions needs to be preprocessed using the clear, open and close operations described previously. These operations are not trivial to implement in database queries and are dependent on just the report and ideally don't need to modify the input data. We'd have to load up the posting data into memory and then run some code. We're already doing that by parsing the input file; the database step would be superfluous.
- **Aggregating Positions.** Though we haven't discussed it in this document so far, the contents of accounts may contain different types of commodities, as well as positions with an attached cost basis. The way that these positions are aggregated together requires the implementation of a custom data type because it obeys some rules about how positions are able to cancel each other out (see How Inventories Work for details). It would be very difficult to build these operations with an SQL database beyond the context of using just a single currency and ignoring cost basis.

This is why Beancount provides a custom tool to directly process and query its data: It provides its own implementation of a SQL client that lets you specify open and close dates and leverages a custom "Inventory" data structure to create sums of the positions of postings. This tools supports columns of Beancount's core types: Amount, Position and Inventory objects.

(In any case, if you're not convinced, Beancount provides a tool to export its contents to a regular SQL database system. Feel free to experiment with it if you like, knock yourself out.)

[1] Please don't pay attention to the numbers in these large figures, they were randomly generated and don't reflect this. We're just interested in showing the structure, in these figures.

[2] Note that this is unrelated to the term "clearing transactions" which means acknowledging or marking that some transactions have been eyeballed by the bookkeeper and checked for correctness.

[3] Note that operations have nothing to do with the Open and Close directives

Beancount provides.

[4] This is not provided yet in Beancount, but would be trivial to implement. All we'd need to do is invert the signs of balances from Liabilities, Income and Equity accounts. It's on the roadmap to provide this eventually.

Installing Beancount

Martin Blais - Updated: June 2015

http://furius.ca/beancount/doc/install

Instructions for downloading and installing Beancount on your computer.

Releases

Beancount is a mature project: the first version was written in 2008. The current rewrite of Beancount is stable. (Technically, this is what I call version 2.x beta).

I'm still working on this Beancount code every weekend these days, so it is very much in active development and evolving, though the great majority of the basic features are basically unchanging. I've built an extensive suite of tests so you can consider the "default" branch of the repository as stable. New features are developed in branches and only merged in the "default" branch when fully stable (the entire battery of tests passes without failures). Changes to "default" are posted to the CHANGES file and a corresponding email is sent to the mailing-list.

So I'm not cutting releases yet. You have to install or run from source. I don't update the PyPI page actively either.

Until development slows down and I decide to bake numbered releases, I recommend cloning and updating regularly from the bitbucket repository and building and running in place the "default" branch (if you're not familiar with Mercurial, "default" is the default, same as master under Git).

Where to Get It

This is the official location for the source code:

```
https://bitbucket.org/blais/beancount/
```

Download it like this, by using Mercurial to make a clone on your machine:

```
hg clone https://bitbucket.org/blais/beancount
```

The author tries to maintain a GitHub clone for this project, but it's often a bit out-of-date:

```
git clone https://github.com/beancount/beancount
```

Other GitHub clones that can be found in the wild are undoubtedly out of date. Don't use them.

How to Install

Installing Python

Beancount uses Python 3.5[1] or above, which is a pretty recent version of Python (as of this writing), and a few common library dependencies. I try to minimize dependencies, but you do have to install a few. This is very easy.

First, you should have a working Python install. Install the latest stable version >=3.5 using the download from python.org. Make sure you have the development headers and libraries installed as well (e.g., the "Python.h" header file). For example, on a Debian/Ubuntu system you would install the **python3-dev** package.

Beancount supports setuptools since Feb 2016, and you will need to install dependencies. You will want to have the "pip3" tool installed. It's probably installed by default along with Python3—test this out by invoking "pip3" command. In any case, under a Debian/Ubuntu system you would simply install the **python3-pip** package.

```
Note for Arch Linux: Arch's Python package does not correctly install all of its runtime dep
```

Python Dependencies

Note that in order to build a full working Python install from source, you will probably need to install a host of other development libraries and their corresponding header files, e.g., libxml2, libxslt1, libgdbm, libmp, libssl, etc. Installing those is dependent on your particular distribution and/or OS. Just make sure that your Python installation has all the basic modules compiled for its default configuration.

Installing Beancount using pip

This is the easiest way to install Beancount. You just install Beancount using

```
sudo pip3 install beancount
```

This should automatically download and install all the dependencies.

Note, however, that this will install the latest version that was pushed to the PyPI repository and not the very latest version available from source. Releases to PyPI are made sporadically but frequently enough not to be too far behind. Consult the CHANGES file if you'd like to find out what is not included since the release date.

Installing Beancount from Source

Installing from source offers the advantage of providing you with the very latest version of the stable branch ("default"). The default branch is as stable as the released version.

Obtain the Source Code

Get the source code from the official repository:

```
hg clone https://bitbucket.org/blais/beancount
```

Install third-party dependencies

You might need to install some non-Python library dependencies, such as libxml2-dev, libxslt1-dev, and perhaps a few more. Try to build, it should be obvious what's missing. If on Ubuntu, use apt-get.

If installing on Windows, see the Windows section below.

Install Beancount from source using pip3

You can then install all the dependencies and Beancount itself using pip:

```
cd beancount
sudo pip3 install .
```

Install Beancount from source using setup.py

First you'll need to install dependent libraries. You can do this using pip:

```
pip3 install python-dateutil bottle ply lxml python-magic beautifulsoup4
```

Or equivalently, you may be able to do that using your distribution, e.g., on Ubuntu/Debian:

```
sudo apt-get install python3-dateutil python3-bottle python3-ply python3-lxml python3-bs4 …
```

Then, you can install the package in your Python library using the usual setup.py invocation:

```
cd beancount
sudo python3 setup.py install
```

Or you can install the package in your user-local Python library using this:

```
sudo python3 setup.py install --user
```

Remember to add ~/.local/bin to your path to access the local install.

Installing for Development

If you want to execute the source in-place for making changes to it, you can use the setuptools "develop" command to point to it:

```
cd beancount
sudo python3 setup.py develop
```

Warning: This modifies a .pth file in your Python installation to point to the path to your clone. You may or may not want this.

The way I work myself is *old-school*; I build it locally and setup my environment to find its libraries. You build it like this:

```
cd beancount
python3 setup.py build_ext -i
```

Finally, both the PATH and PYTHONPATH environment variables need to be updated for it:

```
export PATH=$PATH:/path/to/beancount/bin
export PYTHONPATH=$PYTHONPATH:/path/to/beancount
```

Installing from Packages

Various distributions may package Beancount. Here are links to those known to exist:

- Arch: https://aur.archlinux.org/packages/beancount/

Windows Installation

Native

Installing this package by pip requires compiling some C++ code during the installation procedure which is only possible if an appropriate compiler is available on the computer, otherwise you will receive an error message about missing *vsvarsall.bat* or *cl.exe*.

To be able to compile C++ code for Python you will need to install the same major version of the C++ compiler as your Python installation was compiled with. By running *python* in a console and looking for a text similar to *[MSC v.1900 64 bit (AMD64)]* you can determine which compiler was used for your particular Python distribution. In this example it is *v.1900*.

Using this number find the required Visual C++ version here. Since different versions seem to be compatible as long as the first two digits are the same you can in theory use any Visual C++ compiler between 1900 and 1999.

According to my experience both Python 3.5 and 3.6 was compiled with MSC v.1900 so you can do either of the following to satisfy this requirement:

- Install the standalone Build Tools for Visual Studio 2017 or

- Install the standalone Visual C++ Build Tools 2015 or
- Modify an existing Visual Studio 2017 installation
  - Start the Visual Studio 2017 installer from *Add or remove programs*
  - Select *Individual components*
  - Check *VC++ 2017 version 15.9 v14.16 latest v141 tools* or newer under *Compilers, build tools, and runtimes*
  - Install
- Visual Studio 2019
  - add C++ build tools: C++ core features, MSVC v142 build tools

If cl.exe is not in your path after installation, run Developer Command Prompt for Visual Studio and run the commands there.

With Cygwin

Windows installation is, of course, a bit different. It's a breeze if you use Cygwin. You just have to prep your machine first. Here's how.

- Install the latest Cygwin. This may take a while (it downloads a lot of stuff), but it is well worth it in any case. But before you kick off the install, make sure the following packages are all manually enabled in the interface provided by setup.exe (they're not selected by default):
  - python3
  - python3-devel
  - python3-setuptools
  - mercurial
  - make
  - gcc-core
  - flex
  - bison
  - lxml
  - ply
- Start a new Cygwin bash shell (there should be a new icon on your desktop) and install the pip3 installer tool by running this command:
  **easy_install-3.4 pip
  **Make sure you invoke the version of easy_install which matches your Python version, e.g. easy_install-3.5 if you have Python 3.5 installed, or more.

At this point, you should be able to follow the instructions from the previous sections as is, starting from "Install Beancount using pip".

With WSL

The newly released Windows 10 Anniversary Update brings WSL 'Windows Subsystem for Linux' with bash on Ubuntu on Windows (installation instructions and more at https://msdn.microsoft.com/commandline/wsl/about).

This makes beancount installation easy, from bash:

```
sudo apt-get install python3-pip
sudo apt-get install python3-lxml
sudo pip3 install m3-cdecimal
sudo pip3 install beancount --pre
```

This is not totally "Windows compatible", as it is running in a pico-process, but provides a convenient way to get the Linux command-line experience on Windows. *(Contrib: willwray)*

Notes on lxml

Some users have reported problems installing lxml, and a solution: when installing lxml with pip (under Cygwin), using this may help:

```
STATIC_DEPS=true pip install lxml
```

Checking your Install

You should be able to run the binaries from this document. For example, running bean-check should produce something like this:

```
$ bean-check
```

If this works, you can now go to the tutorial and begin learning how Beancount works.

Reporting Problems

If you need to report a problem, either send email on the mailing-list or file a ticket on Bitbucket. Running the following command lists the presence and versions of dependencies installed on your computer and it might be useful to include the output of this command in your bug report:

```
bean-doctor checkdeps
```

Editor Support

There is support for some editors available:

- Emacs support is provided in the distribution. See the Getting Started text for installation instruction.
- Support for editing with Sublime has been contributed by Martin Andreas Andersen. See his github repo.
- Support for editing with Vim has been contributed by Nathan Grigg. See his GitHub repo.
- Visual Studio Code currently has two extensions available. Both have been tested on Linux.
  - Beancount, with syntax checking (bean-check) and support for accounts, currencies, etc. It not only allows selecting existing open accounts but also displays their balance and other metadata. Quite helpful!
  - Beancount Formatter, which can format the whole document, aligning the numbers, etc. using bean-format.

If You Have Problems

If you run into any installation problems, file a ticket, email me, or hit the mailing-list.

Post-Installation Usage

Normally the scripts located under beancount/bin should be automatically installed somewhere in your executable path. For example, you should be able to just run "bean-check" on the command-line.

Appendix

If everything worked, you can stop reading here. Here I just discuss the various dependencies and why you need them (or why you don't, some of them are optional). This is of interest to developers and some of this info might help troubleshoot problems if you encounter any.

Notes on Dependencies

Python 3.5 or greater

Python 3.5 is widely available at this point, released more than a year ago. However, my experience with open source distribution tells me that a lot of users are running on old machines that won't be upgraded for a while. So for those users, you might have to install your own Python… don't worry, installing Python manually is pretty straightforward.

On Mac, it can also be installed with one of the various package management suits, like Brew, e.g., with "brew install python3".

On an old Linux, download the source from python.org, and then build it like this:

```
tar zxcf Python-3.5.2.tgz
cd Python-3.5.2
./configure
make
sudo make install
```

This should just work. I recommend you install the latest 3.x release (3.5.2 at the time of this writing).

Note: The reason I require at least version 3.5 is because the cdecimal library which supplies Beancount with its implementation of a Decimal representation has been added to the standard library. We need this library to represent all our numbers. Also, Beancount uses the "typing" type annotations which are included in 3.5 (note: You may be able to install "typing" explicitly if you use an older version; no guarantees however).

Some users have reported that there are distributions which package the cdecimal support for Python3 separately. This is the case for Arch, and I've witnessed missing cdecimal support in some Ubuntu installs as well. A check has been inserted into Beancount in December 2015 for this, and a warning should be issued if your Python installation does not have fast decimal numbers. If you are in such a situation, you can try to install cdecimal explicitly, like this:

```
sudo pip3 install m3-cdecimal
```

and Beancount will then use it.

Python Libraries

If you need to install Python libraries, there are a few different ways. First, there is the easy way: there is a package management tool called "pip3" (or just "pip" for version 2 of Python) and you install libraries like this:

```
sudo pip3 install <library-name>
```

Installing libraries from their source code is also pretty easy: download and unzip the source code, and then run this command from its root directory:

```
sudo python3 setup.py install
```

Just make sure the "python3" executable you run when you do that is the same one you will use to run Beancount.

Here are the libraries Beancount depends on and a short discussion of why.

python-dateutil

This library provides Beancount with the ability to parse dates in various formats, it is very convenient for users to have flexible options on the command-line and in the SQL shell.

bottle

The Beancount web interface (invoked via bean-web) runs a little self-contained web server accessible locally on your machine. This is implemented using a tiny library that makes it easy to implement such web applications: bottle.py.

ply

The query client (bean-query) which is used to extract data tables from a ledger file depends on a parser generator: I use Dave Beazley's popular PLY library (version 3.4 or above) because it makes it really easy to implement a custom SQL language parser.

lxml

A tool is provided to bake a static HTML version of the web interface to a zip file (bean-bake). This is convenient to share files with an accountant who may not have your software installed. The web scraping code that is used to do that used the lxml HTML parsing library.

Python Libraries for Export (optional)

google-api-python-client

Some of the scripts I use to export outputs to Google Drive (as well as scripts to maintain and download documentation from it) are checked into the codebase. You don't have to install these libraries if you're not exporting to Google Drive; everything else should work fine without them. These are thus optional.

If you do install this library, it require recent (as of June 2015) installs of

- google-api-python-client: A Python client library for Google's discovery based APIs.
- oauth2client: This is a client library for accessing resources protected by OAuth 2.0, used by the Google API Python client library.
- httplib2: A comprehensive HTTP client library. This is used by oauth2client.

These are best installed using the pip3 tool.

Update: as of 2016, you should be able to install all of the above like this:

```
pip3 install google-api-python-client
```

IMPORTANT: The support for Python3 is fairly recent, and if you have an old install you might experience some failures, e.g. with missing dependencies, such as a missing "gflags" import. Please install those from recent releases and you should be fine. You can install them manually.

Python Libraries for Imports (Optional)

Support for importing identifying, extracting and filing transactions from externally downloaded files is built into Beancount. (This used to be in the Ledger-Hub project, which has been deprecated.) If you don't take advantage of those importing tools and libraries, you don't need these imports.

python-magic (optional)

Beancount attempt to identify the types of files using the stdlib "mimetypes" module and some local heuristics for types which aren't supported, but in addition, it is also useful to install libmagic, which it will use if it is not present:

```
pip3 install python-magic
```

Note that there exists another, older library which provides libmagic support called "filemagic". You need "python-magic" and not "filemagic." More confusingly, under Debian the "python-magic" library is called "filemagic."

Other Tools

It is expected that the user will build their own importers. However, Beancount will provide some modules to help you invoke various external tools. The dependencies you need for these depends on which tool you've configured your importer to use.

Virtualenv Installation

If you'd like to use virtualenv, you can try this (suggestion by Remy X). First install Python 3.5 or beyond[2]:

```
sudo add-apt-repository ppa:fkrull/deadsnakes
sudo apt-get update
sudo apt-get install python3.5
sudo apt-get install python3.5-dev
sudo apt-get install libncurses5-dev
```

Then install and activate virtualenv:

```
sudo apt-get install virtualenv
virtualenv -p /usr/bin/python3.5 bean
source bean/bin/activate
pip install package-name
```

```
sudo apt-get install libz-dev  # For lxml to build
pip install lxml
pip install beancount
```

Development Setup

Installation for Development

For development, you will want to avoid installing Beancount in your Python distribution and instead just modify your PYTHONPATH environment variable to run it from source:

```
export PYTHONPATH=/path/to/install/of/beancount
```

Beancount has a few compiled C extension modules. This is just portable C code and should work everywhere. For development, you want to compile the C modules in-place, within the source code, and load them from there. Build the C extension module in-place like this:

```
python3 setup.py build_ext -i
```

Or equivalently, the root directory has a Makefile that does the same thing and that will rebuild the C code for the lexer and parser if needed (you need flex and bison installed for this):

```
make build
```

With this, you should be able to run the executables under the bin/ subdirectory. You may want to add this to your PATH as well.

Dependencies for Development

Beancount needs a few more tools for development. If you're reading this, you're a developer, so I'll assume you can figure out how to install packages, skip the detail, and just list what you might need:

- **nosetests** (**nose**, or **python3-nose**): This is the test driver to use for running all the unit tests. You definitely need this.
- **GNU flex**: This lexer generator is needed if you intend to modify the lexer. It generated C code that chops up the input files into tokens for the parser to consume.
- **GNU bison**: This old-school parser generator is needed if you intend to modify the grammar. It generates C code that parses the tokens generated by flex. (I like using old tools like this because they are pretty low on dependencies, just C code. It should work everywhere.)
- **GNU tar**: You need this to test the archival capabilities of bean-bake.
- **InfoZIP zip** (comes with Ubuntu): You need this to test the archival capabilities of bean-bake.

- **lxml**: This XML parsing library is used in the web tests. (Unfortunately, the built-in one fails to parse large XML files.)
- **pylint** >= 1.2: I'm running all the source code through this linter. If you contribute code with the intent of it being integrated and released, it has to pass the linter tests (or I'll have to make it pass myself).
- **pyflakes**: I'm running all the source code through this logical error detector. If you contribute code with the intent of it being integrated and released, it has to pass those tests (or I'll have to make it pass myself).
- **snakefood**: I use snakefood to analyze the dependencies between the modules and enforce some ordering, e.g. core cannot import from plugins, for example. There's a target to run it from the root Makefile.
- **graphviz**: The dependency tree of all the modules is generated by snakefood but graphed by the graphviz tool. You need to install it if you want to look at dependencies.

I think that's about it. You certainly don't need *everything* above, but that's the list of tools I use. If you find anything missing, please leave a comment, I may have missed something.

[1] Some people have reported bugs with the cdecimal library in 3.4. I would recommend actually installing 3.5, which appears to have fixed the problem. Technically, 3.3 will still run, but I'll deprecate it for 3.5 at some point, probably when the Ubuntu and Mac OS X distributions have it installed by default.

[2] Installing py3.5: http://www.jianshu.com/p/4f4b2ed568f4

Running Beancount & Generating Reports

Martin Blais, July-Sep 2014

http://furius.ca/beancount/doc/tools

Introduction

Tools

bean-check

bean-report

bean-query

bean-web

Global Pages

View Reports Pages

bean-bake

Introduction

This document describes the tools you use to process Beancount input files, and many of the reports available from it. The syntax of the language is described in the Beancount Language Syntax document. This manual only covers the technical details for using Beancount from the command-line.

Tools

bean-check

bean-check is the program you use to verify that your input syntax and transactions work correctly. All it does is load your input file and run the various plugins you configured in it, plus some extra validation checks. It report errors (if any), and then exits. You run it on your input file, like this:

```
bean-check /path/to/my/file.beancount
```

If there are no errors, there should be no output, it should exit quietly. If there were errors, they will be printed to stderr with the filename, line number and error description (in a format that is understood by Emacs, so you can just use next-error and previous-error to navigate to the file if you want):

```
/home/user/myledger.beancount:44381:   Transaction does not balance: 34.46 USD

   2014-07-12 * "Black Iron Burger" ""
     Expenses:Food:Restaurant               17.23 USD
     Assets:Cash                            17.23 USD
```

You should always fix all the errors before producing reports.

bean-report

This is the main tool used to extract specialized reports to the console in text or one of the various other formats. You invoke it like this:

```
bean-report /path/to/my/file.beancount <report-name>
```

For example:

```
bean-report /path/to/my/file.beancount balances
```

There are many reports available. See the section on reports for a description of the main ones. If you want to produce the full list of reports, ask it for help:

```
bean-report --help-reports
```

Report names can sometimes accept arguments. At the moment the arguments are specified as part of the report name itself, often separated by a colon (:), like this:

```
bean-report /path/to/my/file.beancount balances:Vanguard
```

There are a few special reports you should know about:

- check, or validate: This is the same as running the bean-check command.
- print: This simply prints out the entries that Beancount has parsed, in Beancount syntax. This can be used to confirm that Beancount has read and interpreted your input data correctly (if you're debugging something difficult).

The other reports are what you'd expect: they print out various tables of aggregations of amounts. The reports you can generate are described in a dedicated section below.

bean-query

Beancount's parsed list of transactions and postings is like an in-memory database. **bean-query** is a command-line tool that acts like a client to that in-memory database in which you can type queries in a variant of SQL. You invoke it like this:

```
bean-query /path/to/my/file.beancount
Input file: "/path/to/my/file.beancount"
Ready with 14212 directives (21284 postings in 8879 transactions).
beancount> _
```

More details are available in its own document.

bean-web

bean-web serves all the reports on a web server that runs on your computer. You run it like this:

```
bean-web /path/to/my/file.beancount
```

It will serve all pages on port 8080 on your machine. Navigate to http://localhost:8080 with a web browser. You should be able to click your way through all the reports easily.

The web interface provides a set of global pages and a set of report pages for each "view."

Global Pages

The top-level table of contents page provides links to all the global pages at the top:

- The table of contents (the page you're looking at)
- A list of the errors that occurred in your ledger file
- A view of the source code of your Ledger file (this is used by various other links when referring to a location in your input file.)

The table of contents provides a convenient list of links to all the common views, such as

- "view by year",
- "view by tag", and of course,
- "view all transactions."

There are a few more.

View Reports Pages

When you click on a view report page, you enter a set of pages for the subset of transactions for that view. Various reports about those transactions are available from here:

- Opening balances (a balance sheet at the beginning of the view)
- Balance sheet (a balance sheet at the end of the view)
- Income statement (for the period of the view)
- Various journals for each account (just click on an account name)
- Various reports of holdings at the end of the view
- Lists of documents and prices included in the view's entries
- Some statistics about the view data

… and much more. There should be an index of all the available view reports.

bean-bake

bean-bake runs a **bean-web** instance and bakes all the pages to a directory:

```
bean-bake /path/to/my/file.beancount myfinances
```

It also support baking directly to an archive file:

```
bean-bake /path/to/my/file.beancount myfinances.zip
```

Various compression methods are supported, e.g. .tar.gz.

This is useful to share the web interface with your accountant or other people, who usually don't have the ability to run Beancount. The page links have all been converted to relative links, and they should be able to extract the archive to a directory and browse all the reports the same way you do with **bean-web**.

bean-doctor

This is a debugging tool used to perform various diagnostics and run debugging commands, and to help provide information for reporting bugs. For example, it can do the following:

- List the Beancount dependencies that are installed on your machine, and the ones that are missing. It can tell you what you're missing that you should be installing.
- Print a dump of the parser's lexer tokens for your input file. If you report a parsing bug, it can be useful to look at the lexer output (if you know what you're doing).
- It can run your input file through a parsing round-trip, that is, print out the file and re-read it again and compare it. This is a useful parser and printer test.
- It can check that a directory hierarchy corresponds to a Beancount input file's chart-of-accounts, and reporting directories that do not comply. This is useful in case you decide to change some account names and are maintaining a corresponding archive of documents which needs to be adjusted accordingly.

Context

It can list the context upon which a transaction is applied, i.e., the inventory balances of each account the transaction postings modify, before and after it is applied. Use it like this:

```
$ bean-doctor context /home/blais/accounting/blais.beancount 28514
/home/blais/accounting/blais.beancount:28513:

;   Assets:US:ETrade:BND             100.00 BND {81.968730000000 USD}
;   Assets:US:ETrade:Cash           8143.97 USD

2014-07-25 * "(TRD) BOT +50 BND @82.10" ^273755872
 Assets:US:ETrade:BND                 50.00 BND {82.10 USD}
 Assets:US:ETrade:Cash             -4105.00 USD

;   Assets:US:ETrade:BND                     100.00 BND {81.968730000000 USD}

; ! Assets:US:ETrade:BND               50.00 BND {82.10 USD}
; ! Assets:US:ETrade:Cash            4038.97 USD
```

There is a corresponding Emacs binding (C-c x) to invoke this around the cursor.

bean-format

This pure text processing tool will reformat Beancount input to right-align all the numbers at the same, minimal column. It left-aligns all the currencies. It only modifies whitespace. This tool accepts a filename as arguments and outputs the aligned file on stdout (similar to UNIX cat).

bean-example

This program generates an example Beancount input file. See the Tutorial for more details about the contents of this example file.

Filtering Transactions

In order to produce different views of your financial transactions, we select a subset of full list of parsed transactions, for example, "all the transactions that occurred in year 2013", and then use that to produce the various available reports that Beancount provides.

At the moment, only a preset list of filters are available as "views" from the web interface. These views include:

- All transactions
- Transactions that occur in a particular year
- Transactions with a particular tag
- Transactions with a particular payee
- Transactions that involve at least one account with a particular name component

At the moment, in order to access reports from these subsets of transactions, you need to use the **bean-web** web interface, and click on the related keyword in the root global page, which enters you into a set of reports for that view.

Reports

The whole point of entering your transactions in a single input file in the first place is that it allows you to sum, filter, aggregate and arrange various subsets of your data into well-known reports. There are three distinct ways to produce reports from Beancount: by using **bean-web** and browsing to a view and then to a specific report (this is the easy way), by using **bean-query** and providing the name of a desired report (and possibly some report-specific arguments), and by using **bean-query** and requesting data by specifying a SQL statement.

Reports can sometimes be rendered in different file formats. Each report type will support being rendered in a list of common ones, such as console text, HTML and CSV. Some reports render in Beancount syntax itself, and we simply call this format name "beancount."

There are many types of reports available, and there will be many more in the future, as many of the features on the roadmap involve new types of output. This section provides an overview of the most common ones. Use bean-report to inquire about the full list supported by your installed version:

```
bean-report --help-reports
```

Balance Reports

All the balance reports are similar in that they produce tables of some set of accounts and their associated balances: [output]

```
|-- Assets
|   `-- US
|       |-- BofA
|       |   `-- Checking                        596.05 USD
|       |-- ETrade
|       |   |-- Cash                          5,120.50 USD
|       |   |-- GLD                              70.00 GLD
|       |   |-- ITOT                             17.00 ITOT
|       |   |-- VEA                              36.00 VEA
|       |   `-- VHT                             294.00 VHT
|       |-- Federal
|       |   `-- PreTax401k
|       |-- Hoogle
|       |   `-- Vacation                        337.26 VACHR
|       `-- Vanguard
...
```

Balances for commodities held "at cost" are rendered at their book value. (Unrealized gains, if any, are inserted as separate transactions by an optional plugin and the result amounts get mixed in with the cost basis if rendered in the same account.)

If an account's balance contains many different types of currencies (commodities not held "at cost", such as dollars, euros, yen), each gets printed on its own line. This can render the balance column a bit too busy and messes with the vertical regularity of account names. This is to some extent an unavoidable compromise, but in practice, there are only a small number of commodities that form the large majority of a user's ledger: the home country's currency units. To this extent, amounts in common currencies can be broken out into their own column using the "operating_currency" option:

```
option "operating_currency" "USD"
option "operating_currency" "CAD"
```

You may use this option multiple times if you have many of them (this is my case, for instance, because I am an expat and hold assets in both my host and home countries). Declaring operating currencies also hold the advantage that the name of the currency need not be rendered, and it can thus more easily be imported into a spreadsheet.

Finally, some accounts are deemed "active" if they have not been closed. A closed account with no transactions in the filtered set of transactions will not be rendered.

Trial Balance (balances)

A trial balance report simply produces a table of final balances of all the active accounts, with all the accounts rendered vertically.

The sum total of all balances is reported at the bottom. Unlike a balance sheet, it may not always balance to zero because of currency conversions. (This is the equivalent of Ledger's bal report.)

The equivalent bean-query command is:

```
SELECT account, sum(position)
GROUP BY account
ORDER BY account;
```

Balance Sheet (balsheet)

A balance sheet is a snapshot of the balances of the Assets, Liabilities and Equity accounts at a particular point in time. In order to build such a report, we have to move balances from the other accounts to it:

1. compute the balances of the Income and Expenses account at that point in time,
2. insert transactions that will zero out balances from these accounts by transferring them to an equity account (Equity:Earnings:Current),
3. render a tree of the balances of the Assets accounts on the left side,
4. render a tree of the Liabilities accounts on the right side,
5. render a tree of the Equity accounts below the Liabilities accounts.

See the introduction document for an example.

Note that the Equity accounts include the amounts reported from the Income and Expenses accounts, also often called "Net Income."

Also, in practice, we make *two* transfers because we're typically looking at a reporting period, and we want to differentiate between Net Income amounts transferred before the beginning of the period (Equity:Earnings:Previous) and during the period itself (Equity:Earnings:Current). And similar pair of transfers is carried out in order to handle currency conversions (this is a bit of a hairy topic, but one with a great solution; refer to the dedicated document if you want all the details).

The equivalent bean-query command is:

```
SELECT account, sum(position)
FROM CLOSE ON 2016-01-01
GROUP BY account ORDER BY account;
```

Opening Balances (openbal)

The opening balances report is simply a balance sheet drawn at the beginning of the reporting period. This report only makes sense for a list of filtered entries that represents a period of time, such as "year 2014." The balance sheet is generated using only the summarization entries that were synthesized when the transactions were filtered (see the double-entry method document).

Income Statement (income)

An income statement lists the final balances of the Income and Expenses accounts. It represents a summary of the transient activity within these accounts. If the balance sheet is the snapshot at a particular point in time, this is the difference between the beginning and the end of a period (in our case: of a filtered set of transactions). The balances of the active Income accounts are rendered on the left, and those of the active Expenses accounts on the right. See the introduction document for an example.

The difference between the total of Income and Expenses balances is the Net Income.

Note that the initial balances of the Income and Expenses accounts should have been zero'ed out by summarization transactions that occur at the beginning of the period, because we're only interested in the changes in these accounts.

Journal Reports

The reports in this section render lists of transactions and other directives in a linear fashion.

Journal (journal)

This report is the equivalent of an "account statement" from an institution, a list of transactions with at least one posting in that account. This is the equivalent of Ledger's register report (reg).

You generate a journal report like this:

```
bean-report myfile.beancount journal …
```

By default, this renders a journal of *all* the transactions, which is unlikely to be what you want. Select a particular account to render like this:

```
bean-report myfile.beancount journal -a Expenses:Restaurant
```

At the moment, the "-a" option accepts only a complete account name, or the name of one of the parent accounts. Eventually we will extend it to handle expressions.

Rendering at Cost

The numbers column on the right displays the changes from the postings of the selected account. Notice that only the balances for the postings affecting the given account are rendered.

The change column renders the changes in the units affected. For example, if this posting is selected:

```
Assets:Investments:Apple       2 AAPL {402.00 USD}
```

The value reported for the change will be "2 AAPL". If you would like to render the values at cost, use the "--at-cost" or "-c" option, which will in this case render "804.00 USD" instead.

There is no "market value" option. Unrealized gains are automatically inserted at the end of the history by the "beancount.plugins.unrealized" plugin. See options for that plugin to insert its unrealized gains.

Note that if the sum of the selected postings is zero, no amount is rendered in the change column.

Adding a Balance Column

If you want to add a column that sums up the running balance for the reported changes, use the "--render-balance" or "-b" option. This does not always make sense to report, so it is up to you to decide whether you want a running balance.

Character Width

By default, the report will be as wide as your terminal allows. Restrict the width to a set number of characters with the "-w" option.

Precision

The number of fractional digits for the number rendering can be specified via "--precision" or "-k".

Compact, Normal or Verbose

In its normal operation, Beancount renders an empty line between transactions. This helps delineate transactions where there are multiple currencies affected, as they render on separate lines. If you want a more compact rendering, use the "--compact" or "-x" option.

On the other hand, if you want to render the affected postings under the transaction line, use the "--verbose" or "-X" option.

Summary

Here is a summary of its arguments:

```
optional arguments:
  -h, --help            show this help message and exit
  -a ACCOUNT, --account ACCOUNT
                        Account to render
  -w WIDTH, --width WIDTH
                        The number of characters wide to render the report to
  -k PRECISION, --precision PRECISION
                        The number of digits to render after the period
  -b, --render-balance, --balance
                        If true, render a running balance
  -c, --at-cost, --cost
                        If true, render values at cost
  -x, --compact         Rendering compactly
  -X, --verbose         Rendering verbosely
```

Equivalent SQL Query

The equivalent bean-query command is:

```
SELECT date, flag, description, account, cost(position), cost(balance);
```

Conversions (conversions)

This report lists the total of currency conversions that result from the selected transactions. (Most people won't need this.)

Documents (documents)

This report produces an HTML list of all the external documents found in the ledger, either from explicit directives or from plugins that automatically find the documents and add them to the stream of transactions.

Holdings Reports

These reports produces aggregations for assets held at cost.

Holdings & Aggregations (holdings*)

This report produces a detailed list of all holdings found in the ledger. You can produce aggregates by commodity and accounts using the "-g" option.

Net Worth (networth)

This report produces a short summary of the net worth (equity) of the ledger, in each of the operating currencies.

Other Report Types

Cash

This report renders balances in commodities not held at cost, in other words, cash:

```
bean-report example.beancount cash -c USD
```

The report allows you to convert all currencies to a common currency (in the example above, "convert everything to USD"). There's also an option to report only on the operating currencies. I use this to get an overview of all uninvested cash.

Prices (prices)

This report renders a list of price points for a base currency in terms of a quote currency. The list is sorted by date. You can output this table in beancount format as well. This is convenient to save a price database to a file, that can then be combined and loaded into another input file.

Statistics (stats)

This report simply provides various statistics on the parsed entries.

Update Activity (activity)

This table renders for each account the date of the last entry.

Getting Started with Beancount

Martin Blais, July 2014

http://furius.ca/beancount/doc/getting-started

Introduction

This document is a gentle guide to creating your first Beancount file, initializing it with some options, some guidelines for how to organize your file, and instructions for declaring accounts and making sure their initial balance does not raise errors. It also contains some material on configuring the Emacs text editor, if you use that.

You will probably want to have read some of the User's Manual first in order to familiarize yourself with the syntax and kinds of available directives, or move on to the Cookbook if you've already setup a file or know how to do that. If you're familiar with Ledger, you may want to read up on the differences between Ledger and Beancount first.

Editor Support

Emacs

First, you will want a bit of support for your text editor. I'm using Emacs, so I'll explain my setup for this, but certainly you can use any text editor to compose your input file.

Beancount's Emacs support provides a *Minor Mode* (as opposed to a *Major Mode*) on purpose, so that you can combine it with your favorite text editing mode. I like to use *Org-Mode*, which is an outline mode for Emacs which allows you to fold and unfold its sections to view the outline of your document. This makes it much easier to edit a very long text file. (I don't use Org-Mode's literal programming blocks, I only use it to fold and unfold sections of text.)

You can configure Emacs to automatically open files with a ".beancount" extension to enable beancount-mode by adding this code to your ~/.emacs file:

```
(add-to-list 'load-path "/path/to/beancount/src/elisp")
(require 'beancount)
(add-to-list 'auto-mode-alist '("\\.beancount\\'" . beancount-mode))
```

Beancount-mode provides some nice editing features:

- In order to quickly and painlessly insert account names, you need completion on the account names. You will probably get frustrated with all the typing if you don't have completion. Beancount-mode automatically recognizes account names already present in the input file (refresh the list with "C-c r") and you can insert a new account name with "C-c '".
- It's also nice to align the amounts in a transaction nicely; this formatting can be done automatically with "C-c ;" with the cursor on the transaction you want to align.

Vim

Nathan Grigg implement support for vim in this github repo. It supports:

- Syntax highlighting (not exhaustive, but all the basics)
- Account name completion (using omnifunc, C-X C-O)
- Aligning the decimal points across transactions (:AlignCommodity)

Sublime

Support for editing with Sublime has been contributed by Martin Andreas Andersen. See his github repo.

Creating your First Input File

To get started, let's create a minimal input file with two accounts and a single transaction. Enter or copy the following input to a text file:

```
2014-01-01 open Assets:Checking
2014-01-01 open Equity:Opening-Balances

2014-01-02 * "Deposit"
  Assets:Checking           100.00 USD
  Equity:Opening-Balances
```

Brief Syntax Overview

A few notes and an ultra brief overview of the Beancount syntax:

- Currencies must be entirely in capital letters (allowing numbers and some special characters, like "_" or "-"). Currency symbols (such as $ or €) are not supported.
- Account names do not admit spaces (though you can use dashes), and must have at least two components, separated by colons.
- Description strings must be quoted, like this: "AMEX PMNT".
- Dates are only parsed in ISO8601 format, that is, YYYY-MM-DD.
- Tags must begin with "#", and links with "^".

For a complete description of the syntax, visit the User's Manual.

Validating your File

The purpose of Beancount is to produce reports from your input file (either on the console or serve via its web interface). However, there is a tool that you can use to simply load its contents and make some validation checks on it, to ensure that your input does not contain errors. Beancount can be quite strict; this is a tool that you use while you're entering your data to ensure that your input file is legal. The tool is called "bean-check" and you invoke it like this:

```
bean-check /path/to/your/file.beancount
```

Try it now on the file you just created from the previous section. It should exit with no output. If there are errors, they will be printed on the console. The errors are printed out in a format that Emacs recognizes by default, so you can use Emacs' next-error and previous-error built-in functions to move the cursor to the location of the problem.

Viewing the Web Interface

A convenient way to view reports is to bring up the "bean-web" tool on your input file. Try it:

```
bean-web /path/to/your/file.beancount
```

You can then point a web browser to http://localhost:8080 and click your way around the various reports generated by Beancount. You can then modify the input file and reload the web page your browser is pointing to—bean-web will automatically reload the file contents.

At this point, you should probably read some of the Language Syntax document.

How to Organize your File

In this section we provide general guidelines for how to organize your file. This assumes you've read the Language Syntax document.

Preamble to your Input File

I recommend that you begin with just a single file[1]. My file has a header that tells my editor (Emacs) what "mode" to open the file with, followed by some common options:

```
;; -*- mode: org; mode: beancount; coding: utf-8; fill-column: 400; -*-
option "title" "My Personal Ledger"
option "operating_currency" "USD"
option "operating_currency" "CAD"
```

The title options is used in reports. The list of "operating currencies" identify those commodities which you use most commonly as "currencies" and which warrant rendering in their own dedicated columns in reports (this declaration has no other effect on the behavior of any of the calculations).

Sections & Declaring Accounts

I like to organize my input file in sections that correspond to each real-world account. Each section defines all the accounts related to this real-world account by using an Open directive. For example, this is a checking account:

```
2007-02-01 open Assets:US:BofA:Savings              USD
2007-02-01 open Income:US:BofA:Savings:Interest     USD
```

I like to declare the currency constraints as much as possible, to avoid mistakes. Also, note how I declare an income account specific to this account. This helps break down income in reporting for taxes, as you will likely receive a tax document in relation to that specific account's income (in the US this would be a 1099-INT form produced by your bank).

Here's what the opening accounts might look like for an investment account:

```
2012-03-01 open Assets:US:Etrade:Main:Cash            USD
2012-03-01 open Assets:US:Etrade:Main:ITOT            ITOT
2012-03-01 open Assets:US:Etrade:Main:IXUS            IXUS
2012-03-01 open Assets:US:Etrade:Main:IEFA            IEFA
2012-03-01 open Income:US:Etrade:Main:Interest        USD
2012-03-01 open Income:US:Etrade:Main:PnL             USD
2012-03-01 open Income:US:Etrade:Main:Dividend        USD
2012-03-01 open Income:US:Etrade:Main:DividendNoTax   USD
```

The point is that all these accounts are related somehow. The various sections of the cookbook will describe the set of accounts suggested to create for each section.

Not all sections have to be that way. For example, I have the following sections:

- **Eternal accounts.** I have a section at the top dedicated to contain special and "eternal" accounts, such as payables and receivables.
- **Daybook.** I have a "daybook" section at the bottom that contains all cash expenses, in chronological order.
- **Expense accounts.** All my expenses accounts (categories) are defined in their own section.
- **Employers.** For each employer I've defined a section where I put the entries for their direct deposits, and track vacations, stock vesting and other job-related transactions.
- **Taxes.** I have a section for taxes, organized by taxation year.

You can organize it any way you like, because Beancount doesn't care about the ordering of declarations.

Closing Accounts

If a real-world account has closed, or is never going to have any more transactions posted to it, you can declare it "closed" at a particular date by using a Close directive:

```
; Moving to another bank.
2013-06-13 close Assets:US:BofA:Savings
```

This tells Beancount not to show the account in reports that don't include any date where it was active. It also avoids errors by triggering an error if you do try to post to it at a later date.

De-duping

One problem that will occur frequently is that once you have some sort of code or process set up to automatically extract postings from downloaded files, you will end up importing postings which provide two separate sides of the same

transaction. An example is the payment of a credit card balance via a transfer from a checking account. If you download the transactions for your checking account, you will extract something like this:

```
2014-06-08 * "ONLINE PAYMENT - THANK YOU"
  Assets:CA:BofA:Checking           -923.24 USD
```

The credit card download will yield you this:

```
2014-06-10 * "AMEX EPAYMENT    ACH PMT"
  Liabilities:US:Amex:Platinum       923.24 USD
```

Many times, transactions from these accounts need to be booked to an expense account, but in this case, these are two separate legs of the same transaction: a transfer. When you import one of these, you normally look for the other side and merge them together:

```
;2014-06-08 * "ONLINE PAYMENT - THANK YOU"
2014-06-10 * "AMEX EPAYMENT    ACH PMT"
  Liabilities:US:Amex:Platinum       923.24 USD
  Assets:CA:BofA:Checking           -923.24 USD
```

I often leave one of the description lines in comments—just my choice, Beancount ignores it. Also note that I had to choose one of the two dates. I just choose the one I prefer, as long as it does not break any balance assertion.

In the case that you would forget to merge those two imported transactions, worry not! That's what balance assertions are for. Regularly place a balance assertion in either of these accounts, e.g., every time you import, and you will get a nice error if you ended up entering the transaction twice. This is pretty common and after a while it becomes second nature to interpret that compiler error and fix it in seconds.

Finally, when I know I import just one side of these, I select the other account manually and I mark the posting I know will be imported later with a flag, which tells me I haven't de-duped this transaction yet:

```
2014-06-10 * "AMEX EPAYMENT    ACH PMT"
  Liabilities:US:Amex:Platinum       923.24 USD
  ! Assets:CA:BofA:Checking
```

Later on, when I import the checking account's transactions and go fishing for the other side of this payment, I will find this and get a good feeling that the world is operating as it should.

(If you're interested in more of a discussion around de-duping and merging transaction, see this feature proposal. Also, you might be interested in the "effective_date" plugin external contribution, which splits transactions in two.)

Which Side?

So if you organize your account in sections the way I suggest above, which section of the file should you leave such "merged" transactions in, that is, transactions that involve two separate accounts? Well, it's your call. For example, in the case of a transfer between two accounts organized such that they have their own dedicated sections, it would be nice to be able to leave both transactions there so that when you edit your input file you see them in either section, but unfortunately, the transaction must occur in only one place in your document. You have to choose one.

Personally I'm a little careless about being consistent which of the section I choose to leave the transaction in; sometimes I choose one section of my input file, or that of the other account, for the same pair of accounts. It hasn't been a problem, as I use Emacs and i-search liberally which makes it easy to dig around my gigantic input file. If you want to keep your input more tidy and organized, you could come up with a rule for yourself, e.g. "credit card payments are always left in the paying account's section, not in the credit card account's section", or perhaps you could leave the transaction in both sections and comment one out[2].

Padding

If you're just starting out—and you probably are if you're reading this—you will have no historical data. This means that the balances of your Assets and Liabilities accounts in Beancount will all be zero. But the first thing you should want to do after defining some accounts is establish a balance sheet and bring those amounts to their actual current value.

Let's take your checking account as an example, say you opened it a while back. You don't remember exactly when, so let's use an approximate date:

```
2000-05-28 open Assets:CA:BofA:Checking  USD
```

The next thing you do is look up your current balance and put a balance assertion for the corresponding amount:

```
2014-07-01 balance Assets:CA:BofA:Checking    1256.35 USD
```

Running Beancount on just this will correctly produce an error because Beancount assumes an implicit balance assertion of "empty" at the time you open an account. You will have to "pad" your account to today's balance by inserting a *balance adjustment* at some point in time between the opening and the balance, against some equity account, which is an arbitrary place to book "where you received the initial balance from." For this purpose, this is usually the "Equity:Opening-Balances" account. So let's include this padding transaction and recap what we have so far:

```
2000-05-28 open Assets:CA:BofA:Checking  USD
```

```
2000-05-28 * "Initialize account"
  Equity:Opening-Balances                    -1256.35 USD
  Assets:CA:BofA:Checking                     1256.35 USD

2014-07-01 balance Assets:CA:BofA:Checking    1256.35 USD
```

From here onwards, you would start adding entries reflecting everything that happened after 7/1. However, what if you wanted to go *back* in time? It is perfectly reasonable that once you've got your chart-of-accounts set up you might want to fill in the missing history until at least the beginning of this year.

Let's assume you had a single transaction in June 2014, and let's add it:

```
2000-05-28 open Assets:CA:BofA:Checking  USD

2000-05-28 * "Initialize account"
  Equity:Opening-Balances                    -1256.35 USD
  Assets:CA:BofA:Checking                     1256.35 USD

2014-06-28 * "Paid credit card bill"
  Assets:CA:BofA:Checking                     -700.00 USD
  Liabilities:US:Amex:Platinum                 700.00 USD

2014-07-01 balance Assets:CA:BofA:Checking    1256.35 USD
```

Now the balance assertion fails! You would need to adjust the initialization entry to fix this:

```
2000-05-28 open Assets:CA:BofA:Checking  USD

2000-05-28 * "Initialize account"
  Equity:Opening-Balances                    -1956.35 USD
  Assets:CA:BofA:Checking                     1956.35 USD

2014-06-28 * "Paid credit card bill"
  Assets:CA:BofA:Checking                     -700.00 USD
  Liabilities:US:Amex:Platinum                 700.00 USD

2014-07-01 balance Assets:CA:BofA:Checking    1256.35 USD
```

Now this works. So basically, every single time you insert an entry in the past, you would have to adjust the balance. Isn't this annoying? Well, yes.

Fortunately, we can provide some help: you can use a Pad directive to replace and automatically synthesize the balance adjustment to match the next balance check, like this:

```
2000-05-28 open Assets:CA:BofA:Checking  USD

2000-05-28 pad Assets:CA:BofA:Checking Equity:Opening-Balances
```

```
2014-06-28 * "Paid credit card bill"
  Assets:CA:BofA:Checking                      -700.00 USD
  Liabilities:US:Amex:Platinum                  700.00 USD

2014-07-01 balance Assets:CA:BofA:Checking    1256.35 USD
```

Note that this is only needed for balance sheet accounts (Assets and Liabilities) because we don't care about the initial balances of the Income and Expenses accounts, we only care about their transitional value (the changes they post during a period). For example, it makes no sense to bring up the Expenses:Restaurant account to the sum total value of all the costs of the meals you consumed since you were born.

So you will probably want to get started with Open & Pad directives for each Assets and Liabilities accounts.

What's Next?

At this point you will probably move onwards to the Cookbook, or read the User's Manual if you haven't already done that.

[1] It is tempting to want to break down a large file into many smaller ones, but especially at first, the convenience of having everything in a single place is great.

[2] Some people have suggested that Beancount automatically detect duplicated transactions based on a heuristic and automatically ignore (remove) one of the two, but this has not been tried out yet. In particular, this would lend itself well to organizing transactions not just per section, but in separate files, i.e., all files would contain all the transactions for the accounts they represent. If you're interested in adding this feature, you could easily implement this as a plugin, without disrupting the rest of the system.

Beancount Language Syntax

Martin Blais, Updated: April 2016

http://furius.ca/beancount/doc/syntax

Introduction

Syntax Overview

- Directives
- Ordering of Directives
- Accounts
- Commodities / Currencies

Introduction

This is a user's manual to the language of Beancount, the command-line double-entry bookkeeping system. Beancount defines a computer language that allows you to enter financial transactions in a text file and extract various reports from it. It is a generic counting tool that works with multiple currencies, commodities held at cost (e.g., stocks), and even allows you to track unusual things, like vacation hours, air miles and rewards points, and anything else you might want to count, even beans.

This document provides an introduction to Beancount's syntax and some of the technical details needed for one to understand how it carries out its calculations. This document does *not* provide an introduction to the double-entry method, a motivation, nor examples and guidelines for entering transactions in your input file, nor how to run the tools. These subjects have their own dedicated documents, and it is recommended that you have had a look at those before diving into this user's manual. This manual covers the technical details for using Beancount.

Syntax Overview

Directives

Beancount is a declarative language. The input consists of a text file containing mainly a list of **directives**, or **entries** (we use these terms interchangeably in the code and documentation); there is also syntax for defining various **options**. Each directive begins with an associated *date*, which determines the point in time at which the directive will apply, and its *type*, which defines which kind of event this directive represents. All the directives begin with a syntax that looks like this:

```
YYYY-MM-DD <type> …
```

where YYYY is the year, MM is the numerical month, and DD the numerical date. All digits are required, for example, the 7th of May 2007 should be "2007-05-07", including its zeros. Beancount supports the international ISO 8601 standard format for dates, with dashes (e.g., "2014-02-03"), or the same ordering with slashes (e.g., "2014/02/03").

Here are some example directives, just to give you an idea of the aesthetics:

```
2014-02-03 open Assets:US:BofA:Checking
```

The end product of a parsed input file is a simple list of these entries, in a data structure. All operations in Beancount are performed on these entries.

Each particular directive type is documented in a section below.

Ordering of Directives

The order of declaration of the directives is not important. In fact, the entries are re-sorted chronologically after parsing and before being processed. This is an important feature of the language, because it makes it possible for you to organize your input file any way you like without having to worry about affecting the meaning of the directives.

Except for transactions, each directive is assumed to occur at the *beginning* of each day. For example, you could declare an account being opened on the same day as its first transaction:

```
2014-02-03 open Assets:US:BofA:Checking

2014-02-03 * "Initial deposit"
  Assets:US:BofA:Checking          100 USD
  Assets:Cash                     -100 USD
```

However, if you hypothetically closed that account immediately, you could not declare it closed on the same day, you would have to fudge the date forward by declaring the close on 2/4:

```
2014-02-04 close Assets:US:BofA:Checking
```

This also explains why balance assertions are verified before any transactions that occur on the same date. This is for consistency.

Accounts

Beancount accumulates commodities in accounts. The names of these accounts do not have to be declared before being used in the file, they are recognized as "accounts" by virtue of their syntax alone[1]. An account name is a colon-separated list of capitalized words which begin with a letter, and whose first word must be one of five account types:

```
Assets Liabilities Equity Income Expenses
```

Each component of the account names begin with a capital letter or a number and are followed letters, numbers or dash (-) characters. All other characters are disallowed.

Here are some realistic example account names:

```
Assets:US:BofA:Checking
Liabilities:CA:RBC:CreditCard
Equity:Retained-Earnings
Income:US:Acme:Salary
Expenses:Food:Groceries
```

The set of all names of accounts seen in an input file implicitly define a hierarchy of accounts (sometimes called a **chart-of-accounts**), similarly to how files are organized in a file system. For example, the following account names:

```
Assets:US:BofA:Checking
Assets:US:BofA:Savings
Assets:US:Vanguard:Cash
Assets:US:Vanguard:RGAGX
Assets:Receivables
```

implicitly declare a tree of accounts that looks like this:

```
`-- Assets
    |-- Receivables
    `-- US
        |-- BofA
        |   |-- Checking
        |   `-- Savings
        `-- Vanguard
            |-- Cash
            `-- RGAGX
```

We would say that "Assets:US:BofA" is the parent account of "Assets:US:BofA:Checking", and that the latter is a child account of the former.

Commodities / Currencies

Accounts contain **currencies**, which we sometimes also call **commodities** (we use both terms interchangeably). Like account names, currency names are recognized by their syntax, though, unlike account names, they need not be declared before being used). The syntax for a currency is a word all in capital letters, like these:

```
USD
CAD
EUR
MSFT
```

```
IBM
AIRMILE
```

(Technically: up to 24 characters long, beginning with a capital letter and ending with a capital letter or a number. The middle characters may include "_-'.".) The first three might evoke real world currencies to you (US dollars, Canadian dollars, Euros); the next two, stock ticker names (Microsoft and IBM). And the last: rewards points (airmiles). Beancount knows of no such thing; from its perspective all of these instruments are treated similarly. There is no built-in notion of any previously existing currency. These currency names are just names of "things" that can be put in accounts and accumulated in inventories associated with these accounts.

There is something elegant about the fact that there is no "special" currency unit, that all commodities are treated equally the same: Beancount is inherently a multi-currency system. You will appreciate this if, like many of us, you are an expat and your life is divided between two or three continents. You can handle an international ledger of accounts without any problems.

And your use of currencies can get quite creative: you can create a currency for your home, for example (e.g. MYLOFT), a currency to count accumulated vacation hours (VACHR), or a currency to count potential contributions to your retirement accounts allowed annually (IRAUSD). You can actually solve many problems this way. The cookbook describes many such concrete examples.

Beancount does not support the dollar sign syntax, e.g., "$120.00". You should always use symbols for currencies in your input file. This makes the input more regular and is a design choice. For monetary units, I suggest that you use the standard ISO 4217 currency code as a guideline; these quickly become familiar. However, you may include some other characters in currency names, like underscores (_) or dashes (-), but no spaces.

Finally, you will notice that there exists a "Commodity" directive that can be used to declare currencies. It is entirely optional: currencies come into being as you use them. The purpose of the directive is simply to attach metadata to it.

Strings

Whenever we need to insert some free text as part of an entry, it should be surrounded by double-quotes. This applies to the payee and narration fields, mainly; basically anything that's not a date, a number, a currency, an account name.

Strings may be split over multiple lines. (Strings with multiple lines will include their newline characters and those need to be handled accordingly when rendering.)

Comments

The Beancount input file isn't intended to contain only your directives: you can be liberal in placing comments and headers in it to organize your file. Any text on a line after the character ";" is ignored, text like this:

```
; I paid and left the taxi, forgot to take change, it was cold.
2015-01-01 * "Taxi home from concert in Brooklyn"
  Assets:Cash      -20 USD  ; inline comment
  Expenses:Taxi
```

You can use one or more ";" characters if you like. Prepend on all lines if you want to enter a larger comment text. If you prefer to have the comment text parsed in and rendered in your journals, see the Note directive elsewhere in this document.

Any line that does not begin as a valid Beancount syntax directive (e.g. with a date) is silently ignored. This way you can insert markup to organize your file for various outline modes, such as org-mode in Emacs. For example, you could organize your input file by institution like this and fold & unfold each of the sections independently,:

```
* Banking
** Bank of America

2003-01-05 open Assets:US:BofA:Checking
2003-01-05 open Assets:US:BofA:Savings

;; Transactions follow …

** TD Bank

2006-03-15 open Assets:US:TD:Cash

;; More transactions follow …
```

The unmatching lines are simply ignored.

Note to visiting Ledger users: In Ledger, ";" is used both for marking comments and for attaching "Ledger tags" (Beancount metadata) to postings. This is not the case in Beancount. In Beancount comments are always just comments. Metadata has its own separate syntax.

Directives

For a quick reference & overview of directive syntax, please consult the Syntax Cheat Sheet.

Open

All accounts need to be declared "open" in order to accept amounts posted to them. You do this by writing a directive that looks like this:

```
2014-05-01 open Liabilities:CreditCard:CapitalOne     USD
```

The general format of the Open directive is:

```
YYYY-MM-DD open Account [ConstraintCurrency,...] ["BookingMethod"]
```

The comma-separated optional list of constraint currencies enforces that all changes posted to this account are in units of one of the declared currencies. Specifying a currency constraint is recommended: the more constraints you provide Beancount with, the less likely you will be to make data entry mistakes because if you do, it will warn you about it.

Each account should be declared "opened" at a particular date that precedes (or is the same as) the date of the first transaction that posts an amount to that account. Just to be clear: an Open directive does not have to appear before transactions *in the file*, but rather, the *date* of the Open directive must precede the date of postings to that account. The order of declarations in the file is not important. So for example, this is a legal input file:

```
2014-05-05 * "Using my new credit card"
  Liabilities:CreditCard:CapitalOne         -37.45 USD
  Expenses:Restaurant

2014-05-01 open Liabilities:CreditCard:CapitalOne     USD
1990-01-01 open Expenses:Restaurant
```

Another optional declaration for opening accounts is the "booking method", which is the algorithm that will be invoked if a reducing lot leads to an ambiguous choice of matching lots from the inventory (0, 2 or more lots match). Currently, the possible values it can take are:

- **STRICT**: The lot specification has to match exactly one lot. This is the default method. If this booking method is invoked, it will simply raise an error. This ensures that your input file explicitly selects all matching lots.
- **NONE**: No lot matching is performed. Lots of any price will be accepted. A mix of positive and negative numbers of lots for the same currency is allowed. (This is similar to how Ledger treats matching… it ignores it.)

Close

Similarly to the Open directive, there is a Close directive that can be used to tell Beancount that an account has become inactive, for example:

```
; Closing credit card after fraud was detected.
2016-11-28 close Liabilities:CreditCard:CapitalOne
```

The general format of the Close directive is:

```
YYYY-MM-DD close Account
```

This directive is used in a couple of ways:

- An error message is raised if you post amounts to that account after its closing date (it's a sanity check). This helps avoid mistakes in data entry.
- It helps the reporting code figure out which accounts are still active and filter out closed accounts outside of the reporting period. This is especially useful as your ledger accumulates much data over time, as there will be accounts that stop existing and which you just don't want to see in reports for years that follow their closure.

Note that a Close directive does not currently generate an implicit zero balance check. You may want to add one just before the closing date to ensure that the account is correctly closed with empty contents.

At the moment, once an account is closed, you cannot reopen it after that date. (Though you can, of course, delete or comment-out the directive that closed it.) Finally, there are utility functions in the code that allow you to establish which accounts are open on a particular date. I strongly recommend that you close accounts when they actually close in reality, it will keep your ledger more tidy.

Commodity

There is a "Commodity" directive that can be used to declare currencies, financial instruments, commodities (different names for the same thing in Beancount):

```
1867-01-01 commodity CAD
```

The general format of the Commodity directive is:

```
YYYY-MM-DD commodity Currency
```

This directive is a late arrival, and is entirely optional: you can use commodities without having to really declare them this way. The purpose of this directive is to attach commodity-specific metadata fields on it, so that it can be gathered by plugins later on. For example, you might want to provide a long descriptive name for each commodity, such as "Swiss Franc" for commodity "CHF", or "Hooli Corporation Class C Shares" for "HOOL", like this:

```
1867-01-01 commodity CAD
  name: "Canadian Dollar"
  asset-class: "cash"

2012-01-01 commodity HOOL
  name: "Hooli Corporation Class C Shares"
  asset-class: "stock"
```

For example, a plugin could then gather the metadata attribute names and perform some aggregations per asset class.

You can use any date for a commodity… but a relevant date is the date at which it was created or introduced, e.g. Canadian dollars were first introduced in 1867, ILS (new Israeli Shekels) are in use since 1986-01-01. For a company, the date the corporation was formed and shares created could be a good date. Since the main purpose of this directive is to collect per-commodity information, the particular date you choose doesn't matter all that much.

It is an error to declare the same commodity twice.

Transactions

Transactions are the most common type of directives that occur in a ledger. They are slightly different than the other ones, because they can be followed by a list of postings. Here is an example:

```
2014-05-05 txn "Cafe Mogador" "Lamb tagine with wine"
  Liabilities:CreditCard:CapitalOne         -37.45 USD
  Expenses:Restaurant
```

As for all the other directives, a transaction directive begins with a date in the *YYYY-MM-DD* format and is followed by the directive name, in this case, "txn". However, because transactions are the *raison d'être* for our double-entry system and as such are by far the most common type of directive that should appear in a Beancount input file, we make a special case and allow the user to elide the "txn" keyword and just use a flag instead of it:

```
2014-05-05 * "Cafe Mogador" "Lamb tagine with wine"
  Liabilities:CreditCard:CapitalOne         -37.45 USD
  Expenses:Restaurant
```

A flag is used to indicate the status of a transaction, and the particular meaning of the flag is yours to define. We recommend using the following interpretation for them:

- *: Completed transaction, known amounts, "this looks correct."
- !: Incomplete transaction, needs confirmation or revision, "this looks incorrect."

In the case of the first example using "txn" to leave the transaction unflagged, the default flag ("*") will be set on the transaction object. (I nearly always use the "*" variant and never the keyword one, it is mainly there for consistency with all the other directive formats.)

You can also attach flags to the postings themselves, if you want to flag one of the transaction's legs in particular:

```
2014-05-05 * "Transfer from Savings account"
```

```
  Assets:MyBank:Checking             -400.00 USD
  ! Assets:MyBank:Savings
```

This is useful in the intermediate stage of de-duping transactions (see Getting Started document for more details).

The general format of a Transaction directive is:

```
YYYY-MM-DD [txn|Flag] [[Payee] Narration] [Flag] Account Amount [{Cost}] [@ Price] [Flag] Ac
```

The lines that follow the first line are for "Postings." You can attach as many postings as you want to a transaction. For example, a salary entry might look like this:

```
2014-03-19 * "Acme Corp" "Bi-monthly salary payment"
  Assets:MyBank:Checking             3062.68 USD     ; Direct deposit
  Income:AcmeCorp:Salary            -4615.38 USD     ; Gross salary
  Expenses:Taxes:TY2014:Federal       920.53 USD     ; Federal taxes
  Expenses:Taxes:TY2014:SocSec        286.15 USD     ; Social security
  Expenses:Taxes:TY2014:Medicare       66.92 USD     ; Medicare
  Expenses:Taxes:TY2014:StateNY       277.90 USD     ; New York taxes
  Expenses:Taxes:TY2014:SDI             1.20 USD     ; Disability insurance
```

The crucial and only constraint on postings is that the sum of their balance amounts must be zero. This is explained in full detail below.

Metadata

It's also possible to attach metadata to the transaction and/or any of its postings, so the fully general format is:

```
YYYY-MM-DD [txn|Flag] [[Payee] Narration] [Key: Value] ... [Flag] Account Amount [{Cost}] [@
```

See the dedicated section on metadata below.

Payee & Narration

A transaction may have an optional "payee" and/or a "narration." In the first example above, the payee is "Cafe Mogador" and the narration is "Lamb tagine with wine".

The payee is a string that represents an external entity that is involved in the transaction. Payees are sometimes useful on transactions that post amounts to Expense accounts, whereby the account accumulates a category of expenses from multiple businesses. A good example is "Expenses:Restaurant", which will include all postings for the various restaurants one might visit.

A narration is a description of the transaction that you write. It can be a comment about the context, the person who accompanied you, some note about the product you bought... whatever you want it to be. It's for you to insert whatever you like. I like to insert notes when I reconcile, it's quick and I can

refer to my notes later on, for example, to answer the question "What was the name of that great little sushi place I visited with Andreas on the West side last winter?"

If you place a single string on a transaction line, it becomes its narration:

```
2014-05-05 * "Lamb tagine with wine"
  …
```

If you want to set just a payee, put an empty narration string:

```
2014-05-05 * "Cafe Mogador" ""
  …
```

For legacy reasons, a pipe symbol ("|") is accepted between those strings (but this will be removed at some point in the future):

```
2014-05-05 * "Cafe Mogador" | ""
  …
```

You may also leave out either (but you must provide a flag):

```
2014-05-05 *
  …
```

Note for Ledger users. Ledger does not have separate narration and payee fields, it has only one field, which is referred to by the "Payee" metadata tag, and the narration ends up in a saved comment (a "persistent note"). In Beancount, a Transaction object simply has two fields: payee and narration, where payee just happens to have an empty value a lot of the time.

For a deeper discussion of how and when to use payees or not, see Payees, Subaccounts, and Assets.

Costs and Prices

Postings represent a single amount being deposited to or withdrawn from an account. The simplest type of posting includes only its amount:

```
2012-11-03 * "Transfer to pay credit card"
  Assets:MyBank:Checking            -400.00 USD
  Liabilities:CreditCard             400.00 USD
```

If you converted the amount from another currency, you must provide a conversion rate to balance the transaction (see next section). This is done by attaching a "price" to the posting, which is the rate of conversion:

```
2012-11-03 * "Transfer to account in Canada"
  Assets:MyBank:Checking            -400.00 USD @ 1.09 CAD
  Assets:FR:SocGen:Checking          436.01 CAD
```

You could also use the "@@" syntax to specify the total cost:

```
2012-11-03 * "Transfer to account in Canada"
  Assets:MyBank:Checking            -400.00 USD @@ 436.01 CAD
  Assets:FR:SocGen:Checking          436.01 CAD
```

Beancount will automatically compute the per-unit price, that is 1.090025 CAD (note that the precision will differ between the last two examples).

After the transaction, we are not interested in keeping track of the exchange rate for the units of USD deposited into the account; those units of "USD" are simply deposited. In a sense, the rate at which they were converted at has been forgotten.

However, some commodities that you deposit in accounts must be "held at cost." This happens when you want to keep track of the *cost basis* of those commodities. The typical use case is investing, for example when you deposit shares of a stock in an account. What you want in that circumstance is for the acquisition cost of the commodities you deposit to be attached to the units, such that when you remove those units later on (when you sell), you should be able to identify by cost which of those units to remove, in order to control the effect of taxes (and avoid errors). You can imagine that for each of the units in the account there is an attached cost basis. This will allow us to automatically compute capital gains later.

In order to specify that a posting to an account is to be held at a specific cost, include the cost in curly brackets:

```
2014-02-11 * "Bought shares of S&P 500"
  Assets:ETrade:IVV                10 IVV {183.07 USD}
  Assets:ETrade:Cash         -1830.70 USD
```

This is the subject of a deeper discussion. Refer to "How Inventories Work" and "Trading with Beancount" documents for an in-depth discussion of these topics.

Finally, you can include both a cost and a price on a posting:

```
2014-07-11 * "Sold shares of S&P 500"
  Assets:ETrade:IVV               -10 IVV {183.07 USD} @ 197.90 USD
  Assets:ETrade:Cash          1979.90 USD
  Income:ETrade:CapitalGains
```

The price will only be used to insert a price entry in the prices database (see Price section below). This is discussed in more details in the Balancing Postings section of this document.

Important Note. Amounts specified as either per-share or total prices or costs are *always unsigned*. It is an error to use a negative sign or a negative cost and Beancount will raise an error if you attempt to do so.

Balancing Rule - The "weight" of postings

A crucial aspect of the double-entry method is ensuring that the sum of all the amounts on its postings equals ZERO, in all currencies. This is the central, non-negotiable condition that engenders the accounting equation, and makes it possible to filter any subset of transactions and drawing balance sheets that balance to zero.

But with different types of units, the previously introduced price conversions and units "held at cost", what does it all mean?

It's simple: we have devised a simple and clear rule for obtaining an amount and a currency from each posting, to be used for balancing them together. We call this the "weight" of a posting, or the balancing amount. Here is a short example of weights derived from postings using the four possible types of cost/price combinations:

```
YYYY-MM-DD
  Account       10.00 USD                          -> 10.00 USD
  Account       10.00 CAD @ 1.01 USD               -> 10.10 USD
  Account       10 SOME {2.02 USD}                 -> 20.20 USD
  Account       10 SOME {2.02 USD} @ 2.50 USD      -> 20.20 USD
```

Here is the explanation of how it is calculated:

1. If the posting has **only an amount** and no cost, the balance amount is just the amount and currency on that posting. Using the first example from the previous section, the amount is -400.00 USD, and that is balanced against the 400.00 USD of the second leg.

2. If the posting has **only a price**, the price is multiplied by the number of units and the price currency is used. In the second example from the preceding section, that is -400.00 USD x 1.09 CAD(/USD) = -436.00 CAD, and that is balanced against the other posting of 436.00 CAD[2].

3. If the posting **has a cost**, the cost is multiplied by the number of units and the cost currency is used. In the third example from the preceding section, that is 10 IVV x 183.08 USD(/IVV) = 1830.70 USD. That is balanced against the cash leg of -1830.70 USD, so all is good.

4. Finally, if a posting **has both a cost and a price**, we simply ignore the price. This optional price is used later on to generate an entry in the in-memory prices database, but it is not used in balancing at all.

With this rule, you should be able to easily balance all your transactions. Moreover, this rule makes it possible to let Beancount automatically calculate capital gains for you (see Trading with Beancount for details).

Reducing Positions

When you post a *reduction* to a position in an account, the reduction must always match an existing lot. For example, if an account holds 3200 USD and

a transaction posts a -1200 USD change to that account, the 1200 USD match against the existing 3200 USD, and the result is a single position of 2000 USD. This also works for negative values. For example, if an account has a -1300 USD balance and you post a +2000 USD change to it, you obtain a 700 USD balance.

A change posted to an account, regardless of the account type, can result in a positive or negative balance; there are no limitations on the balances of simple commodity amounts (that is, those with no cost associated to them). For example, while Assets accounts *normally* have a positive balance and Liabilities accounts *usually* a negative one, you can legally credit an Assets account to a negative balance, or debit a Liabilities account to a positive balance. This is because in the real world these things do happen: you might write a check too many and obtain temporary credit from your bank's checking account (usually along with an outrageous "overdraft" fee), or pay that credit card balance twice by mistake.

For commodities held at cost, the cost specification of the posting must match one of the lots held in the inventory before the transaction is applied. The list of lots is gathered, and matched against the specification in the {...} part of the of posting. For example, if you provide a cost, only those lots whose cost match that will remain. If you provide a date, only those lots which match that date will remain. And you can use a label as well. If you provide a cost and a date, both of these are matched against the list of candidate lots to reduce. This is essentially a filter on the list of lots.

If the filtered list results in a single lot, that lot is chosen to be reduced. If the list results in multiple lots, but the total amount being reduced equals the total amount in the lots, all those lots are reduced by that posting.

For example, if in the past you had the following transactions:

```
2014-02-11 * "Bought shares of S&P 500"
  Assets:ETrade:IVV                20 IVV {183.07 USD, "ref-001"}
  …
```

Each of the following reductions would be unambiguous:

```
2014-05-01 * "Sold shares of S&P 500"
  Assets:ETrade:IVV               -20 IVV {183.07 USD}
  …
```

However, the following would be ambiguous:

```
2014-05-01 * "Sold shares of S&P 500"
  Assets:ETrade:IVV               -20 IVV {}
  …
```

If multiple lots match against the reducing posting and their number is not the total number, we are in a situation of *ambiguous matches*. What happens then, is that the account's *booking method* is invoked. There are multiple booking

methods, but by default, all accounts are set to use the "STRICT" booking method. This method simply issues an error in an ambiguous situation.

You may set the account's booking method to "FIFO" to instruct Beancount to select the oldest of the lots. Or "LIFO" for the latest (youngest) of the lots. This will automatically select all the necessary matching lots to fulfill the reduction.

For such postings, a change that results in a negative number of units is usually impossible. Beancount does not currently allow holding a negative number of a commodity held at cost. For example, an input with just this transaction will fail:

```
2014-05-23 *
  Assets:Investments:MSFT        -10 MSFT {43.40 USD}
  Assets:Investments:Cash     434.00 USD
```

If it did not, this would result in a balance of -10 units of MSFT. On the other hand, if the account had a balance of 12 units of MSFT held at 43.40 USD on 5/23, the transaction would book just fine, reducing the existing 12 units to 2. Most often, the error that will occur is that the account will be holding a balance of 10 or more units of MSFT at a *different cost*, and the user will specify an incorrect value for the cost. For instance, if the account had a positive balance of 20 MSFT {42.10 USD}, the transaction above would still fail, because there aren't 10 or more units of MSFT at 43.40 USD to remove from.

This constraint is enforced for a few reasons:

- Mistakes in data entry for stock units are not uncommon, they have an important negative impact on the correctness of your Ledger—the amounts are usually large—and they will typically trigger an error from this constraint. Therefore, the error check is a useful way to detect these types of errors.
- Negative numbers of units held at cost are fairly rare. Chances are you don't need them at all. Exceptions include: short sales of stock, holding spreads on futures contracts, and depending on how you account for them, short positions in currency trading.

This is why this check is enabled by default.

For more details of the inventory booking algorithm, see the How Inventories Work document.

Amount Interpolation

Beancount is able to fill in some of the details of a transaction automatically. You can currently elide the amount of *at most* one posting within a transaction:

```
2012-11-03 * "Transfer to pay credit card"
  Assets:MyBank:Checking            -400.00 USD
  Liabilities:CreditCard
```

In the example above, the amount of the credit card posting has been elided. It is automatically calculated by Beancount at 400.00 USD to balance the transaction. This also works with multiple postings, and with postings with costs:

```
2014-07-11 * "Sold shares of S&P 500"
  Assets:ETrade:IVV                 -10 IVV {183.07 USD}
  Assets:ETrade:Cash             1979.90 USD
  Income:ETrade:CapitalGains
```

In this case, the units of IVV are sold at a higher price ($197.90) than they were bought for ($183.07). The cash first posting has a weight of -10 x 183.07 = -1830.70 and the second posting a straightforward $1979.90. The last posting will be ascribed the difference, that is, a balance of -149.20 USD, which is to say, a *gain* of $149.20.

When calculating the amount to be balanced, the same balance amounts that are used to check that the transaction balances to zero are used to fill in the missing amounts. For example, the following would not trigger an error:

```
2014-07-11 * "Sold shares of S&P 500"
  Assets:ETrade:IVV                 -10 IVV {183.07 USD} @ 197.90 USD
  Assets:ETrade:Cash
```

The cash account would receive 1830.70 USD automatically, because the balance amount of the IVV posting is -1830.70 USD (if a posting has both a cost and a price, the cost basis is always used and the optional price ignored). While this is accepted and correct from a balancing perspective, this would be incomplete from an accounting perspective: the capital gain on a sale needs to be accounted for separately and besides, the amount deposited to the cash account if you did as above would fall short of the real deposit (1979.00 USD) and hopefully a subsequent balance assertion in the cash account would indicate this oversight by triggering an error.

Finally, this also works when the balance includes multiple commodities:

```
2014-07-12 * "Uncle Bob gave me his foreign currency collection!"
  Income:Gifts                 -117.00 ILS
  Income:Gifts                -3000.00 INR
  Income:Gifts                 -800.00 JPY
  Assets:ForeignCash
```

Multiple postings (one for each commodity required to balance) will be inserted to replace the elided one.

Tags

Transactions can be tagged with arbitrary strings:

```
2014-04-23 * "Flight to Berlin" #berlin-trip-2014
  Expenses:Flights              -1230.27 USD
```

```
  Liabilities:CreditCard
```

This is similar to the popular idea of "hash tagging" on Twitter and such. These tags essentially allow you to mark a subset of transactions. They can then be used as a filter to generate reports on only this subset of transactions. They have numerous uses. I like to use them to mark all my trips.

Multiple tags can be specified as well:

```
2014-04-23 * "Flight to Berlin" #berlin-trip-2014 #germany
  Expenses:Flights              -1230.27 USD
  Liabilities:CreditCard
```

(If you want to store key-value pairs on directives, see the section on metadata below.)

The Tag Stack

Oftentimes multiple transactions related to a single tag will be entered consecutively in a file. As a convenience, the parser can automatically tag transactions within a block of text. How this works is simple: the parser has a "stack" of current tags which it applies to all transactions as it reads them one-by-one. You can push and pop tags onto/from this stack, like this:

```
pushtag #berlin-trip-2014

2014-04-23 * "Flight to Berlin"
  Expenses:Flights              -1230.27 USD
  Liabilities:CreditCard

poptag #berlin-trip-2014
```

This way, you can also push multiple tags onto a long, consecutive set of transactions without having to type them all in.

Links

Transactions can also be linked together. You may think of the link as a special kind of tag that can be used to group together a set of financially related transactions over time. For example you may use links to group together transactions that are each related with a specific invoice. This allows to track payments (or write-offs) associated with the invoice:

```
2014-02-05 * "Invoice for January" ^invoice-pepe-studios-jan14
  Income:Clients:PepeStudios           -8450.00 USD
  Assets:AccountsReceivable

2014-02-20 * "Check deposit - payment from Pepe" ^invoice-pepe-studios-jan14
  Assets:BofA:Checking                  8450.00 USD
  Assets:AccountsReceivable
```

Or track multiple transfers related to a single nefarious purpose:

```
2014-02-05 * "Moving money to Isle of Man" ^transfers-offshore-17
  Assets:WellsFargo:Savings          -40000.00 USD
  Assets:WellsFargo:Checking          40000.00 USD

2014-02-09 * "Wire to FX broker" ^transfers-offshore-17
  Assets:WellsFargo:Checking         -40025.00 USD
  Expenses:Fees:WireTransfers            25.00 USD
  Assets:OANDA:USDollar               40000.00

2014-03-16 * "Conversion to offshore beans" ^transfers-offshore-17
  Assets:OANDA:USDollar          -40000.00 USD
  Assets:OANDA:GBPounds           23391.81 GBP @ 1.71 USD

2014-03-16 * "God save the Queen (and taxes)" ^transfers-offshore-17
  Assets:OANDA:GBPounds             -23391.81 GBP
  Expenses:Fees:WireTransfers          15.00 GBP
  Assets:Brittania:PrivateBanking    23376.81 GBP
```

Linked transactions can be rendered by the web interface in their own dedicated journal, regardless of the current view/filtered set of transactions (the list of links is a global page).

Balance Assertions

A balance assertion is a way for you to input your statement balance into the flow of transactions. It tells Beancount to verify that the number of units of a particular commodity in some account should equal some expected value at some point in time. For instance, this

```
2014-12-26 balance Liabilities:US:CreditCard   -3492.02 USD
```

says "Check that the number of USD units in account "Liabilities:US:CreditCard" on the *morning* of December 26th, 2014 is -3492.02 USD." When processing the list of entries, if Beancount encounters a different balance than this for USD it will report an error.

If no error is reported, you should have some confidence that the list of transactions that precedes it in this account is highly likely to be correct. This is useful in practice, because in many cases some transactions can get imported separately from the accounts of each of their postings (see the de-duping problem). This can result in you booking the same transaction twice without noticing, and regularly inserting a balance assertion will catch that problem every time.

The general format of the Balance directive is:

```
YYYY-MM-DD balance Account Amount
```

Note that a balance assertion, like all other non-transaction directives, applies at the **beginning** of its date (i.e., midnight at the start of day). Just imagine that the balance check occurs right after midnight on that day.

Balance assertions only make sense on balance sheet accounts (Assets and Liabilities). Because the postings on Income and Expenses accounts are only interesting because of their transient value, i.e., for these accounts we're interested in sums of changes over a period of time (not the absolute value), it makes little sense to use a balance assertion on income statement accounts.

Also, take note that each account benefits from an implicit balance assertion that the account is empty after it is opened at the date of its Open directive. You do not need to explicitly assert a zero balance when opening accounts.

Multiple Commodities

A Beancount account may contain more than one commodity (although in practice, you will find that this does not occur often, it is sensible to create dedicated sub-accounts to hold each commodity, for example, holding a portfolio of stocks). A balance assertion applies *only* to the commodity of the assertion; it leaves the other commodities in the balance unchecked. If you want to check multiple commodities, use multiple balance assertions, like this:

```
; Check cash balances from wallet
2014-08-09 balance Assets:Cash     562.00 USD
2014-08-09 balance Assets:Cash     210.00 CAD
2014-08-09 balance Assets:Cash      60.00 EUR
```

There is currently no way to exhaustively check the full list of commodities in an account (a proposal is underway).

Note that in this example if an exhaustive check really matters to you, you could circumvent by defining a subaccount of the cash account to segregate each commodity separately, like this Assets:Cash:USD, Assets:Cash:CAD.

Lots Are Aggregated

The balance assertion applies to the sum of units of a particular commodities, irrespective of their cost. For example, if you hold three lots of the same commodity in an account, for example, 5 HOOL {500 USD} and 6 HOOL {510 USD}, the following balance check should succeed:

```
2014-08-09 balance Assets:Investing:HOOL     11 HOOL
```

All the lots are aggregated together and you can verify their number of units.

Checks on Parent Accounts

Balance assertions may be performed on parent accounts, and will include the balances of theirs and their sub-accounts:

```
2014-01-01 open Assets:Investing
2014-01-01 open Assets:Investing:Apple       AAPL
2014-01-01 open Assets:Investing:Amazon      AMZN
2014-01-01 open Assets:Investing:Microsoft   MSFT
2014-01-01 open Equity:Opening-Balances

2014-06-01 *
  Assets:Investing:Apple       5 AAPL {578.23 USD}
  Assets:Investing:Amazon      5 AMZN {346.20 USD}
  Assets:Investing:Microsoft   5 MSFT {42.09 USD}
  Equity:Opening-Balances

2014-07-13 balance Assets:Investing 5 AAPL
2014-07-13 balance Assets:Investing 5 AMZN
2014-07-13 balance Assets:Investing 5 MSFT
```

Note that this does require that a parent account have been declared as Open, in order to be legitimately used in the balance assertions directive.

Before Close

It is useful to insert a balance assertion for 0 units just before closing an account, just to make sure its contents are empty as you close it. The Close directive does not insert that for you automatically (we may eventually build a plug-in for it).

Pad

A padding directive automatically inserts a transaction that will make the subsequent balance assertion succeed, if it is needed. It inserts the difference needed to fulfill that balance assertion. (What "rubber space" is in LaTeX, Pad directives are to balances in Beancount.)

Note that by "subsequent," I mean in *date order*, not in the order of the declarations in the file. This is the conceptual equivalent of a transaction that will automatically expand or contract to fill the difference between two balance assertions over time. It looks like this:

```
2014-06-01 pad Assets:BofA:Checking Equity:Opening-Balances
```

The general format of the Pad directive is:

```
YYYY-MM-DD pad Account AccountPad
```

The first account is the account to credit the automatically calculated amount to. This is the account that should have a balance assertion following it (if the account does not have a balance assertion, the pad entry is benign and does nothing).

The second account is for the other leg of the transaction, it is the source where the funds will come from, and this is almost always some Equity account. The reason for this is that this directive is generally used for initializing the balances of new accounts, to save us from having to either insert such a directive manually, or from having to enter the full past history of transactions that will bring the account to its current balance.

Here is a realistic example usage scenario:

```
; Account was opened way back in the past.
2002-01-17 open Assets:US:BofA:Checking

2002-01-17 pad Assets:US:BofA:Checking Equity:Opening-Balances

2014-07-09 balance Assets:US:BofA:Checking  987.34 USD
```

This will result in the following transaction being inserted right after the Pad directive, on the same date:

```
2002-01-17 P "(Padding inserted for balance of 987.34 USD)"
  Assets:US:BofA:Checking        987.34 USD
  Equity:Opening-Balances       -987.34 USD
```

This is a normal transaction—you will see it appear in the rendered journals along with the other ones. (Note the special "P" flag, which can be used by scripts to find these.)

Observe that without balance assertions, Pad directives make no sense. Therefore, like balance assertions, they are normally only used on balance sheet accounts (Assets and Liabilities).

You could also insert Pad entries *between* balance assertions, it works too. For example:

```
2014-07-09 balance Assets:US:BofA:Checking  987.34 USD
… more transactions…
2014-08-08 pad Assets:US:BofA:Checking Equity:Opening-Balances

2014-08-09 balance Assets:US:BofA:Checking  1137.23 USD
```

Without intervening transactions, this would insert the following padding transaction:

```
2014-08-08 P "(Padding inserted for balance of 1137.23 USD)"
  Assets:US:BofA:Checking        149.89 USD
  Equity:Opening-Balances       -149.89 USD
```

In case that's not obvious, 149.89 USD is the difference between 1137.23 USD and 987.34 USD. If there were more intervening transactions posting amounts to the checking account, the amount would automatically have been adjusted to make the second assertion pass.

Unused Pad Directives

You may not currently leave unused Pad directives in your input file. They will trigger an error:

```
2014-01-01 open Assets:US:BofA:Checking

2014-02-01 pad Assets:US:BofA:Checking Equity:Opening-Balances

2014-06-01 * "Initializing account"
  Assets:US:BofA:Checking                    212.00 USD
  Equity:Opening-Balances

2014-07-09 balance Assets:US:BofA:Checking  212.00 USD
```

(Being this strict is a matter for debate, I suppose, and it could eventually be moved to an optional plugin.)

Commodities

Note that the Pad directive does not specify any commodities at all. All commodities with corresponding balance assertions in the account are affected. For instance, the following code would have a padding directive insert a transaction with separate postings for USD and CAD:

```
2002-01-17 open Assets:Cash

2002-01-17 pad Assets:Cash Equity:Opening-Balances

2014-07-09 balance Assets:Cash    987.34 USD
2014-07-09 balance Assets:Cash    236.24 CAD
```

If the account contained other commodities that aren't balance asserted, no posting would be inserted for those.

Cost Basis

At the moment, Pad directives do not work with accounts holding positions held at cost. The directive is really only useful for cash accounts. (This is mainly because balance assertions do not yet allow specifying a cost basis to assert. It is possible that in the future we decide to support asserting the total cost basis, and that point we could consider supporting padding with cost basis.)

Multiple Paddings

You cannot currently insert multiple padding entries for the same account and commodity:

```
2002-01-17 open Assets:Cash
```

```
2002-02-01 pad Assets:Cash Equity:Opening-Balances

2002-03-01 pad Assets:Cash Equity:Opening-Balances

2014-04-19 balance Assets:Cash    987.34 USD
```

(There is a proposal pending to allowing this and spread the padding amount evenly among all the intervening Pad directives, but it is as of yet unimplemented.)

Notes

A Note directive is simply used to attach a dated comment to the journal of a particular account, like this:

```
2013-11-03 note Liabilities:CreditCard "Called about fraudulent card."
```

When you render the journal, the note should be rendered in context. This can be useful to record facts and claims associated with a financial event. I often use this to record snippets of information that would otherwise not make their way to a transaction.

The general format of the Note directive is:

```
YYYY-MM-DD note Account Description
```

The description string may be split among multiple lines.

Documents

A Document directive can be used to attach an external file to the journal of an account:

```
2013-11-03 document Liabilities:CreditCard "/home/joe/stmts/apr-2014.pdf"
```

The filename gets rendered as a browser link in the journals of the web interface for the corresponding account and you should be able to click on it to view the contents of the file itself. This is useful to integrate account statements and other downloads into the flow of accounts, so that they're easily accessible from a few clicks. Scripts could also be written to obtain this list of documents from an account name and do something with them.

The general format of the Document directive is:

```
YYYY-MM-DD document Account PathToDocument
```

Documents from a Directory

A more convenient way to create these entries is to use a special option to specify directories that contain a hierarchy of sub-directories that mirrors that of the chart of accounts. For example, the Document directive shown in the previous section could have been created from a directory hierarchy that looks like this:

```
stmts
`-- Liabilities
    `-- CreditCard
        `-- 2014-04-27.apr-2014.pdf
```

By simply specifying the root directory as an option (note the absence of a trailing slash):

```
option "documents" "/home/joe/stmts"
```

The files that will be picked up are those that begin with a date as above, in the *YYYY-MM-DD* format. You may specify this option multiple times if you have many such document archives. In the past I have used one directory for each year (if you scan all your documents, the directory can grow to a large size, scanned documents tend to be large files).

Organizing your electronic document statements and scans using the hierarchy of your ledger's accounts is a fantastic way to organize them and establish a clear, unambiguous place to find these documents later on, when they're needed.

Prices

Beancount sometimes creates an in-memory data store of prices for each commodity, that is used for various reasons. In particular, it is used to report unrealized gains on account holdings. Price directives can be used to provide data points for this database. A Price directive establishes the rate of exchange between one commodity (the base currency) and another (the quote currency):

```
2014-07-09 price HOOL  579.18 USD
```

This directive says: "The price of one unit of HOOL on July 9th, 2014 was 579.18 USD." Price entries for currency exchange rates work the same way:

```
2014-07-09 price USD  1.08 CAD
```

The general format of the Price directive is:

```
YYYY-MM-DD price Commodity Price
```

Remember that Beancount knows nothing about what HOOL, USD or CAD are. They are opaque "things." You attach meaning to them. So setting a price per hour for your vacation hours is perfectly valid and useful too, if you account for your unused vacations on such terms:

```
2014-07-09 price VACHR  38.46 USD  ; Equiv. $80,000 year
```

Prices from Postings

If you use the beancount.plugins.implicit_prices plugin, every time a Posting appears that has a cost or an optional price declared, it will use that cost or price to automatically synthesize a Price directive. For example, this transaction:

```
2014-05-23 *
  Assets:Investments:MSFT        -10 MSFT {43.40 USD}
  Assets:Investments:Cash     434.00 USD
```

automatically becomes this after parsing:

```
2014-05-23 *
  Assets:Investments:MSFT        -10 MSFT {43.40 USD}
  Assets:Investments:Cash     434.00 USD

2014-05-23 price MSFT  43.40 USD
```

This is convenient and if you enable it, will probably be where most of the price points in your ledger's price database will come from. You can print a table of the parsed prices from a ledger (it is just another type of report).

Prices on the Same Day

Notice that there is no notion of time; Beancount is not designed to solve problems for intra-day traders, though of course, it certainly is able to handle multiple trades per day. It just stores its prices per day. (Generally, if you need many features that require a notion of intra-day time, you're better off using another system, this is not the scope of a bookkeeping system.)

When multiple Price directives do exist for the same day, the last one to appear in the file will be selected for inclusion in the Price database.

Events

Event directives[3] are used to track the value of *some variable of your choice* over time. For example, your location:

```
2014-07-09 event "location" "Paris, France"
```

The above directive says: "Change the value of the 'location' event to 'Paris, France' as of the 9th of July 2014 and onwards." A particular event type only has a single value per day.

The general format of the Event directive is:

```
YYYY-MM-DD event Name Value
```

The event's *Name* string does not have to be declared anywhere, it just begins to exist the first time you use the directive. The *Value* string can be anything you like; it has no prescribed structure.

How to use these is best explained by providing examples:

- **Location**: You can use events for tracking the city or country you're in. This is sensible usage within a ledger because the nature of your expenses are heavily correlated to where you are. It's convenient: if you use Beancount regularly, it is very little effort to add this bit of information to your

file. I usually have a "cash daybook" section where I put miscellaneous cash expenses, and that's where I enter those directives.

- **Address**: If you move around a lot, it's useful to keep a record of your past home addresses. This is sometimes requested on government forms for immigration. The Green Card application process in the US, for instance,
- **Employer**: You can record your date of employment and departure for each job this way. Then you can count the number of days you worked there.
- **Trading window**: If you're an employee of a public company, you can record the dates that you're allowed to trade its stock. This can then be used to ensure you did not trade that stock when the window is closed.

Events are often used to report on numbers of days. For example, in the province of Quebec (Canada) you are insured for free health care coverage if "you spend 183 days of the calendar year or more, excluding trips of 21 days or less." If you travel abroad a lot, you could easily write a script to warn you of remaining day allocations to avoid losing your coverage. Many expats are in similar situations.

In the same vein, the IRS recognizes US immigrants as "resident alien"—and thus subject to filing taxes, yay!—if you pass the Substantial Presence Test, which is defined as "being physically present in the US on at least 31 days during the current year, and 193 days during the 3 year period that includes the current year and the 2 years immediately before that, counting: all the days you were present in the current year, and of the days of the year before that, and of the year preceding that one." Ouch… my head hurts. This gets complicated. Armed with your location over time, you could report it automatically.

Events will also be usable in the filtering language, to specify non-contiguous periods of time. For example, if you are tracking your location using Event directives, you could produce reports for transactions that occur only when you are in a specific location, e.g., "Show me my expenses on all my trips to Germany," or "Give me a list of payees for restaurant transactions when I'm in Montreal."

It is worth noticing that the Price and Event directives are the only ones not associated to an account.

Query

It can be convenient to be able to associate SQL queries in a Beancount file to be able to run these as a report automatically. This is still an early development / experimental directive. In any case, you can insert queries in the stream of transactions like this:

```
2014-07-09 query "france-balances" "
  SELECT account, sum(position) WHERE 'trip-france-2014' in tags"
```

The grammar is

```
YYYY-MM-DD query Name SqlContents
```

Each query has a name, which will probably be used to invoke its running as a report type. Also, the date of the query is intended to be the date at which the query is intended to be run for, that is, transactions following it should be ignored. If you're familiar with the SQL syntax, it's an implicit CLOSE.

Custom

The long-term plan for Beancount is to allow plugins and external clients to define their own directive types, to be declared and validated by the Beancount input language parser. In the meantime, a generic directive is provided for clients to **prototype** new features, e.g., budgeting.

```
2014-07-09 custom "budget" "..." TRUE 45.30 USD
```

The grammar for this directive is flexible:

```
YYYY-MM-DD custom TypeName Value1 ...
```

The first argument is a string and is intended to be unique to your directive. Think of this as the type of your directive. Following it, you can put an arbitrary list of strings, dates, booleans, amounts, and numbers. Note that there is no validation that checks that the number and types of arguments following the *TypeName* is consistent for a particular type.

(See this thread for the origin story around this feature.)

Metadata

You may attach arbitrary data to each of your entries and postings. The syntax for it looks like this:

```
2013-03-14 open Assets:BTrade:HOOLI
  category: "taxable"
```

In this example, a "category" attribute is attached to an account's Open directive, a "statement" attribute is attached to a Transaction directive (with a string value that represents a filename) and a "decision" attribute has been attached to the first posting of the transaction (the additional indentation from the posting is not strictly necessary but it helps with readability).

Metadata can be attached to any directive type. Keys must begin with a lowercase character from a-z and may contain (uppercase or lowercase) letters, numbers, dashes and underscores. Moreover, the values can be any of the following data types:

- Strings
- Accounts

- Currency
- Dates (datetime.date)
- Tags
- Numbers (Decimal)
- Amount (beancount.core.amount.Amount)

There are two ways in which this data can be used:

1. Query tools that come with Beancount (such as bean-query) will allow you to make use of the metadata values for filtering and aggregation.
2. You can access and use the metadata in custom scripts. The metadata values are accessible as a ".meta" attribute on all directives and is a Python dict.

There are no special meanings attached to particular attributes, these are intended for users to define. However, all directives are guaranteed to contain a 'filename' (string) and a 'lineno' (integer) attribute which reflect the location they were created from.

Finally, attributes without a value will be parsed and have a value of 'None'. If an attribute is repeated multiple times, only the first value for this attribute will be parsed and retained and the following values ignored.

Options

The great majority of a Beancount input file consists in directives, as seen in the previous section. However, there are a few global options that can be set in the input file as well, by using a special undated "option" directive:

```
option "title" "Ed's Personal Ledger"
```

The general format of the Option directive is:

```
option Name Value
```

where *Name* and *Value* are both strings.

Note that depending the option, the effect may be to set a single value, or add to a list of existing values. In other words, some options are lists.

There are three ways to view the list of options:

- In this document, which I update regularly.
- To view the definitive list of options supported by your installed version, use the following command: bean-doctor list-options
- Finally, you can peek at the source code as well.

Operating Currencies

One notable option is "operating_currency". By default Beancount does not treat any of the commodities any different from each other. In particular, it doesn't know that there's anything special about the most common of commodities one uses: their currencies. For example, if you live in New Zealand, you're going to have an overwhelming number of NZD commodities in your transactions.

But useful reports try to reduce all non-currency commodities into one of the main currencies used. Also, it's useful to break out the currency units into their own dedicated columns. This may also be useful for exporting in order to avoid having to specify the units for that column and import to a spreadsheet with numbers you can process.

For this reason, you are able to declare the most common currencies you use in an option:

```
option "operating_currency" "USD"
```

You may declare more than one.

In any case, this option is only ever used by reporting code, it never changes the behavior of Beancount's processing or semantics.

Plugins

In order to load plugin Python modules, use the dedicated "plugin" directive:

```
plugin "beancount.plugins.module_name"
```

The name of a plugin should be the name of a Python module in your PYTHONPATH. Those modules will be imported by the Beancount loader and run on the list of parsed entries in order for the plugins to transform the entries or output errors. This allows you to integrate some of your code within Beancount, making arbitrary transformations on the entries. See Scripting & Plugins for details.

Plugins also optionally accept some configuration parameters. These can be provided by an optional final string argument, like this:

```
plugin "beancount.plugins.module_name" "configuration data"
```

The general format of the Option directive is:

```
plugin ModuleName StringConfig
```

The format of the configuration data is plugin-dependent. At the moment, an arbitrary string is passed provided to the plugin. See the plugins' documentation for specific detail on what it can accept.

Also see the "plugin processing mode" option which affects the list of built-in plugins that get run.

Includes

Include directives are supported. This allows you to split up large input files into multiple files. The syntax looks like this:

```
include "path/to/include/file.beancount"
```

The general format is

```
include Filename
```

The specified path can be an absolute or a relative filename. If the filename is relative, it is relative to the including filename's directory. This makes it easy to put relative includes in a hierarchy of directories that can be placed under source control and checked out anywhere.

Include directives are not processed strictly (as in C, for example). The include directives are accumulated by the Beancount parser and processed separately by the loader. This is possible because the order of declarations of a Beancount input file is not relevant.

However, for the moment, options are parsed per-file. The options-map that is kept for post-parse processing is the options-map returned for the top-level file. This is probably subject to review in the future.

What's Next?

This document described all the possible syntax of the Beancount language. If you haven't written any Beancount input yet, you can head to the Getting Started guide, or browse through a list of practical use cases in the Command-line Accounting Cookbook.

[1] Note that there exists an "Open" directive that is used to provide the start date of each account. That can be located anywhere in the file, it does not have to appear in the file somewhere before you use an account name. You can just start using account names in transactions right away, though all account names that receive postings to them will eventually have to have a corresponding Open directive with a date that precedes all transactions posted to the account in the input file.

[2] Note that this is valid whether the price is specified as a per-unit price with the @ syntax or as a total price using the @@ syntax.

[3] I really dislike the name "event" for this directive. I've been trying to find a better alternative, so far without success. The name "register" might be more appropriate, as it resembles that of a processor's register, but that could be confused with an *account register* report. "variable" might work, but somehow that just sounds too computer-sciency and out of context. If you can think of something better, please make a suggestion and I'll seriously entertain a complete rename (with legacy support for "event").

option ”title” ”Joe Smith’s Personal Ledger”
The title of this ledger / input file. This shows up at the top of
every page.
option ”name_assets” ”Assets”
option ”name_liabilities” ”Liabilities”
option ”name_equity” ”Equity”
option ”name_income” ”Income”
option ”name_expenses” ”Expenses”
Root names of every account. This can be used to customize your
category names, so that if you prefer ”Revenue” over ”Income” or
”Capital” over ”Equity”, you can set them here. The account names in
your input files must match, and the parser will validate these. You
should place these options at the beginning of your file, because
they affect how the parser recognizes account names.
option ”account_previous_balances” ”Opening-Balances”
Leaf name of the equity account used for summarizing previous
transactions into opening balances.
option ”account_previous_earnings” ”Earnings:Previous”
Leaf name of the equity account used for transferring previous
retained earnings from income and expenses accrued before the
beginning of the exercise into the balance sheet.
option ”account_previous_conversions” ”Conversions:Previous”
Leaf name of the equity account used for inserting conversions that
will zero out remaining amounts due to transfers before the opening
date. This will essentially ”fixup” the basic accounting equation
due to the errors that priced conversions introduce.
option ”account_current_earnings” ”Earnings:Current”
Leaf name of the equity account used for transferring current
retained earnings from income and expenses accrued during the
current exercise into the balance sheet. This is most often called
”Net Income”.

option ”account_current_conversions” ”Conversions:Current”
Leaf name of the equity account used for inserting conversions that
will zero out remaining amounts due to transfers during the exercise
period.
option ”account_rounding” ”Rounding”
The name of an account to be used to post to and accumulate rounding
error. This is unset and this feature is disabled by default;
setting this value to an account name will automatically enable the
addition of postings on all transactions that have a residual
amount.
option ”conversion_currency” ”NOTHING”
The imaginary currency used to convert all units for conversions at
a degenerate rate of zero. This can be any currency name that isn't
used in the rest of the ledger. Choose something unique that makes
sense in your language.
option ”inferred_tolerance_default” ”CHF:0.01”
option ”default_tolerance” ”CHF:0.01”
THIS OPTION IS DEPRECATED: This option has been renamed to
'inferred_tolerance_default'
Mappings of currency to the tolerance used when it cannot be
inferred automatically. The tolerance at hand is the one used for
verifying (1) that transactions balance, (2) explicit balance checks
from 'balance' directives balance, and (3) in the precision used for
padding (from the 'pad' directive). The values must be strings in
the following format: <currency>:<tolerance> for example,
'USD:0.005'. By default, the tolerance used for currencies without
an inferred value is zero (which means infinite precision). As a
special case, this value, that is, the fallabck value used for all
currencies without an explicit default can be overridden using the
'*' currency, like this: '*:0.5'. Used by itself, this last example
sets the fallabck tolerance as '0.5' for all currencies. (Note: The

new value of this option is ”inferred_tolerance_default”; it renames the option which used to be called ”default_tolerance”. The latter name was confusing.) For detailed documentation about how precision is handled, see this doc: http://furius.ca/beancount/doc/tolerances
(This option may be supplied multiple times.)
option ”inferred_tolerance_multiplier” ”1.1”
A multiplier for inferred tolerance values. When the tolerance values aren't specified explicitly via the 'inferred_tolerance_default' option, the tolerance is inferred from the numbers in the input file. For example, if a transaction has posting with a value like '32.424 CAD', the tolerance for CAD will be inferred to be 0.001 times some multiplier. This is the muliplier value. We normally assume that the institution we're reproducing this posting from applies rounding, and so the default value for the multiplier is 0.5, that is, half of the smallest digit encountered. You can customize this multiplier by changing this option, typically expanding it to account for amounts slightly beyond the usual tolerance, for example, if you deal with institutions with bad of unexpected rounding behaviour. For detailed documentation about how precision is handled, see this doc:
http://furius.ca/beancount/doc/tolerances
option ”infer_tolerance_from_cost” ”True”
Enable 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 * M = 0.045 USD (where M is the inferred_tolerance_multiplier) and this is added to the mix to enlarge the tolerance allowed for units of USD on that

transaction. All the normally inferred tolerances (see http://furius.ca/beancount/doc/tolerances) are still taken into account. Enabling this flag only makes the tolerances potentially wider.
option "tolerance" "0.015"
THIS OPTION IS DEPRECATED: The 'tolerance' option has been deprecated and has no effect.
The tolerance allowed for balance checks and padding directives. In the real world, rounding occurs in various places, and we need to allow a small (but very small) amount of tolerance in checking the balance of transactions and in requiring padding entries to be auto-inserted. This is the tolerance amount, which you can override.
option "use_legacy_fixed_tolerances" "True"
Restore the legacy fixed handling of tolerances. Balance and Pad directives have a fixed tolerance of 0.015 units, and Transactions balance at 0.005 units. For any units. This is intended as a way for people to revert the behavior of Beancount to ease the transition to the new inferred tolerance logic. See http://furius.ca/beancount/doc/tolerances for more details.
option "documents" "/path/to/your/documents/archive"
A list of directory roots, relative to the CWD, which should be searched for document files. For the document files to be automatically found they must have the following filename format: YYYY-MM-DD.(.*)
(This option may be supplied multiple times.)
option "operating_currency" "USD"
A list of currencies that we single out during reporting and create dedicated columns for. This is used to indicate the main currencies that you work with in real life. (Refrain from listing all the possible currencies here, this is not what it is made for; just list the very principal currencies you use daily only.) Because our

system is agnostic to any unit definition that occurs in the input
file, we use this to display these values in table cells without
their associated unit strings. This allows you to import the numbers
in a spreadsheet (e.g, ”101.00 USD” does not get parsed by a
spreadsheet import, but ”101.00” does). If you need to enter a list
of operating currencies, you may input this option multiple times,
that is, you repeat the entire directive once for each desired
operating currency.
(This option may be supplied multiple times.)
option ”render_commas” ”TRUE”
A boolean, true if the number formatting routines should output
commas as thousand separators in numbers.
option ”plugin_processing_mode” ”raw”
A string that defines which set of plugins is to be run by the
loader: if the mode is ”default”, a preset list of plugins are
automatically run before any user plugin. If the mode is ”raw”, no
preset plugins are run at all, only user plugins are run (the user
should explicitly load the desired list of plugins by using the
'plugin' option. This is useful in case the user wants full control
over the ordering in which the plugins are run).
option ”plugin” ”beancount.plugins.module_name”
THIS OPTION IS DEPRECATED: The 'plugin' option is deprecated; it
should be replaced by the 'plugin' directive
A list of Python modules containing transformation functions to run
the entries through after parsing. The parser reads the entries as
they are, transforms them through a list of standard functions, such
as balance checks and inserting padding entries, and then hands the
entries over to those plugins to add more auto-generated goodies.
The list is a list of pairs/tuples, in the format (plugin-name,
plugin-configuration). The plugin-name should be the name of a
Python module to import, and within the module we expect a special

'___plugins___' attribute that should list the name of transform functions to run the entries through. The plugin-configuration argument is an optional string to be provided by the user. Each function accepts a pair of (entries, options_map) and should return a pair of (new entries, error instances). If a plugin configuration is provided, it is provided as an extra argument to the plugin function. Errors should not be printed out the output, they will be converted to strings by the loader and displayed as dictated by the output medium.
(This option may be supplied multiple times.)
option "long_string_maxlines" "64"
The number of lines beyond which a multi-line string will trigger a overly long line warning. This warning is meant to help detect a dangling quote by warning users of unexpectedly long strings.
option "experiment_explicit_tolerances" "True"
Enable an EXPERIMENTAL feature that supports an explicit tolerance value on Balance assertions. If enabled, the balance amount supports a tolerance in the input, with this syntax: <number> ~ <tolerance> <currency>, for example, "532.23 ~ 0.001 USD". See the document on tolerances for more details:
http://furius.ca/beancount/doc/tolerances WARNING: This feature may go away at any time. It is an exploration to see if it is truly useful. We may be able to do without.
option "booking_method" "SIMPLE"
The booking method to apply, for interpolation and for matching lot specifications to the available lots in an inventory at the moment of the transaction. Values may be 'SIMPLE' for the original method used in Beancount, or 'FULL' for the newer method that does fuzzy matching against the inventory and allows multiple amounts to be interpolated (see http://furius.ca/beancount/doc/proposal-booking for details).

Beancount Precision & Tolerances

Martin Blais, 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]. 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    10.21005 RGAGX {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 units of the currency, for example, by providing a number such as 10.7 as input[2]. 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       4.271 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    10.21005 RGAGX {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    10.21005 RGAGX {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 that led to this implementation can be found here. In particular, the proposal highlights on the other systems have attempted to deal with this issue. There are also some discussions 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:

```
First, it's important to realize how these numbers are represented in memory. They are using
```

- "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/be
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/be
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/TOD
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

[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.

[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.

Beancount Query Language

Martin Blais, January 2015

http://furius.ca/beancount/doc/query

Introduction

The purpose of Beancount is to allow the user to create an accurate and error-free representation of financial transactions, typically those occurring in a user or in an institution's associated set of accounts, to then extract various reports from this list of transactions. Beancount provides a few tools to extract reports from the corpus of transactions: custom reports (using the Beancount bean-

report tool), a web interface (using the bean-web tool) and the ability for the user to write their own scripts to output anything they want.

The repository of financial transactions is always read from the text file input, but once parsed and loaded in memory, extracting information from Beancount could be carried out much like you would another database, that is, instead of using custom code to generate output from the data structures, a query language could be compiled and run over the relatively regular list of transactions.

In practice, you could flatten out the list of postings to an external SQL database and make queries using that database's own tools, but the results of this approach are quite disappointing, mainly due to the lack of operations on inventories which is the basis of balancing rules in Beancount. By providing a slightly specialized query engine that takes advantage of the structure of the double-entry transactions we can easily generate custom reports specific to accounting purposes.

This document describes our specialized SQL-like query client. It assumes you have at least a passing knowledge of SQL syntax. If not, you may want to first read something about it.

Motivation

So one might ask: Why create another SQL client? Why not output the data to a SQLite database and allow the user to use that SQL client?

Well, we have done that (see the bean-sql script which converts your Beancount ledger into a SQLite database) and the results are not great. Writing queries is painful and carrying out operations on lots that are held at cost is difficult. By taking advantage of a few aspects of our in-memory data structures, we can do better. So Beancount comes with its own SQL-like query client called "bean-query".

The clients implements the following "extras" that are essential to Beancount:

- It allows to easily filter at two levels simultaneously: You can filter whole transactions, which has the benefit of respecting the accounting equation, and then, usually for presentation purposes, you can also filter at the postings level.
- The client supports the semantics of inventory booking implemented in Beancount. It also supports aggregation functions on inventory objects and rendering functions (e.g., COST() to render the cost of an inventory instead of its contents).
- The client allows you to flatten multiple lots into separate postings to produce lists of holdings each with their associated cost basis.
- Transactions can be summarized in a manner useful to produce balance sheets and income statements. For example, our SQL variant explicitly

supports a "close" operation with an effect similar to closing the year, which inserts transactions to clear income statement accounts to equity and removes past history.

See this post as well for a similar answer.

Warning & Caveat

Approximately 70% of the features desired in the original design doc were implemented in the query language in late 2014. More work will be needed to cover the full feature set, but the current iteration supports most of the use cases covered by Ledger, and I suspect by Beancount users. More feedback is desired on the current version before moving on, and I would like to move forward improving some of the more fundamental aspects of Beancount (namely, inventory booking) before spending more time on the query language. It is functional as it is, but a second revision will be made later on, informed by user feedback and prolonged use.

Therefore, a first release of the query language has been merged in the default stable branch. This document presents this first iteration on the Beancount query language.

Making Queries

The custom query client that we provide is called bean-query. Run it on your ledger file, like this:

```
$ bean-query myfile.beancount
Input file: "My Ledger's Title"
Ready with 13996 directives (21112 postings in 8833 transactions).
beancount> _
```

This launches the query tool in interactive mode, where you can enter multiple commands on the dataset loaded in memory. bean-query parses the input file, spits out a few basic statistics about your ledger, and provides a command prompt for you to enter query commands. You can type "help" here to view the list of available commands.

If any errors in your ledger are incurred, they are printed before the prompt. To suppress error printing, run the tool with the "no-errors" option:

```
$ bean-query -q myfile.beancount
```

Batch Mode Queries

If you'd like to run queries directly from the command-line, without an interactive prompt, you can provide the query directly following your filename:

```
$ bean-query myfile.beancount 'balances from year = 2014'
                     account                          balance
```

```
--------------------------------------------------------------------
… <balances follow> …
```

All the interactive commands are supported.

Shell Variables

The interactive shell has a few "set" variables that you can customize to change some of the behavior of the shell. These are like environment variables. Type the "set" command to see the list of available variables and their current value.

The variables are:

- format (string): The output format. Currently, only "text" is supported.
- boxed (boolean): Whether we should draw a box around the output table.
- spaced (boolean): Whether to insert an empty line between every result row. This is only relevant because postings with multiple lots may require multiple lines to be rendered, and inserting an empty line helps delineate those as separate.
- pager (string): The name of the pager program to pipe multi-page output to when the output is larger than the screen. The initial value is copied from the PAGER environment variable.
- expand (boolean): If true, expand columns that render to lists on multiple rows.

Transactions and Postings

The structure of transactions and entries can be explained by the following simplified diagram:

The contents of a ledger is parsed into a list of directives, most of which are "Transaction" objects which contain two or more "Posting" objects. Postings are always linked only to a single transaction (they are never shared between transactions). Each posting refers to its parent transaction but has a unique account name, amount and associates lot (possibly with a cost), a price and some other attributes. The parent transaction itself contains a few useful attributes as well, such as a date, the name of a payee, a narration string, a flag, links, tags, etc.

If we ignore the list of directives other than transactions, you can view the dataset as a single table of all postings joined with their parent transaction. It is mainly on this joined table of postings that we want to perform filtering and aggregation operations.

However, because of the double-entry bookkeeping constraint, that is, the sum of amounts on postings attached to a transaction is zero, it is also quite useful to perform filtering operations at the transaction level. Because any isolated

transaction has a total impact of zero on the global balance, any subset of transactions will also respect the accounting equation (Assets + Liabilities + Equity + Income + Expenses = 0), and producing balance sheets and income statements on subset of transactions provides meaningful views, for example, "all asset changes and expenses incurred during a trip to the Bahamas" which could be selected by a tag.

For this reason, we modify the SQL SELECT syntax to provide a two-level filtering syntax: since we have a single table of data, we replace the table name in FROM by a filtering expression which applies over transactions, and the WHERE clause applies to data pulled from the resulting list of postings:

```
SELECT <target1>, <target2>, …
FROM <entry-filter-expression>
WHERE <posting-filter-expression>;
```

Both filtering expressions are optional. If no filtering expressions are provided, all postings will be enumerated over. Note that since the transactions are always filtered in date order, the results will be processed and returned in this order by default.

Posting Data Columns

The list of targets refers to attributes of postings or of their parent transaction. The same list of "columns" is made available in the *<posting-filter-expression>*, to filter by posting attributes. For example, you could write the following query:

```
SELECT date, narration, account, position
WHERE account ~ ".*:Vacation" AND year >= 2014;
```

Here, the "date", "year" and "narration" columns refer to attributes of the parent transaction, the "account" and "position" columns refer to attributes of the posting itself.

You may name targets explicitly with the familiar AS operator:

```
SELECT last(date) as last_date, cost(sum(position)) as cost;
```

The full list of posting columns and functions available on them is best viewed by querying your actual client using "help targets" or "help where", which prints out the list and data type of each available data column. You may also refer to the following diagram of the structure of a Posting object for the correspondence between the columns and the data structure attributes.

Entry Data Columns

A different list of column names is available on the *<entry-filter-expression>* of the FROM clause. These columns refer to attributes of the Transaction objects. This clause is intended to filter whole transactions (i.e., all their postings or none at all). Available attributes include the date, transaction flag, the optional

payee, narration, set of tags and links. Use the "help from" command to find the complete list of columns and functions available in this clause.

A Beancount input file consists of many different types of entries, not just transactions. Some of these other types of entries (such as Open, Close, Balance, etc.) may also provide attributes that can be access from the FROM clause. This is embryonic at this point. (It's unclear yet how these will be used in the future, but I suspect we will find some interesting applications for them eventually. The FROM clause provides access to the type of the data entry via column "type". It's still an exploration how much we can make pragmatic use of the SQL language for other types of directives.)

The "id" Column

A special column exists that identifies each transaction uniquely: "id". It is a unique hash automatically computed from the transaction and should be stable between runs.

```
SELECT DISTINCT id;
```

This hash is derived from the contents of the transaction object itself (if you change something about the transaction, e.g. you edit the narration, the id will change).

You can print and select using this column. It can be used for debugging, e.g.

```
PRINT FROM id = '8e7c47250d040ae2b85de580dd4f5c2a';
```

The "balance" Column

One common desired output is a journal of entries over time (also called a "register" in Ledger):

```
SELECT date, account, position WHERE account ~ "Chase:Slate";
```

For this type of report, it is convenient to also render a column of the cumulative balance of the selected postings rows. Access to the previous row is not a standard SQL feature, so we get a little creative and provide a special column called "balance" which is automatically calculated based on the previous selected rows:

```
SELECT date, account, position, balance WHERE account ~ "Chase:Slate";
```

This provides the ability to render typical account statements such as those mailed to you by a bank. Output might look like this:

```
$ bean-query $T "select date, account, position, balance where account ~ 'Expenses:Food:Rest
   date            account          position   balance
---------- ---------------------- --------- ----------
2012-01-02 Expenses:Food:Restaurant 31.02 USD  31.02 USD
2012-01-04 Expenses:Food:Restaurant 25.33 USD  56.35 USD
2012-01-08 Expenses:Food:Restaurant 67.88 USD 124.23 USD
2012-01-09 Expenses:Food:Restaurant 35.28 USD 159.51 USD
```

```
2012-01-14 Expenses:Food:Restaurant 25.84 USD 185.35 USD
2012-01-17 Expenses:Food:Restaurant 36.73 USD 222.08 USD
2012-01-21 Expenses:Food:Restaurant 28.11 USD 250.19 USD
2012-01-22 Expenses:Food:Restaurant 21.12 USD 271.31 USD
```

Wildcard Targets

Using a wildcard as the target list ("*") select a good default list of columns:

```
SELECT * FROM year = 2014;
```

To view the actual list of columns selected, you can use the EXPLAIN prefix:

```
EXPLAIN SELECT * FROM year = 2014;
```

Data Types

The data attributes extracted from the postings or transactions have particular types. Most of the data types are regular types as provided by the underlying Python implementation language, types such as

- String (Python str)
- Date (a datetime.date instance). You can parse a date with the #"..." syntax; this uses Python's dateutil module and is pretty liberal in the formats it accepts.
- Integer (Python int)
- Boolean (Python bool object), as TRUE, FALSE
- Number (a decimal.Decimal object)
- Set of Strings (a Python set of str objects)
- Null objects (NULL)

Positions and Inventories

However, one reason that our SQL-like client exists in the first place is for its ability to carry out aggregation operations on inventories of positions, the data structures at the core of Beancount, that implements its balancing semantics. Internally, Beancount defines Position and Inventory objects and is able to aggregate them together in an instance of Inventory. On each Posting, the "position" column extracts an object of type Position, which when summed over produces an instance of Inventory.

The shell is able to display those appropriately. More specifically, Inventory objects can contain multiple different lots of holdings, and each of these will get rendered on a separate line.

Quantities of Positions and Inventories

Objects of type Position are rendered in their full detail by default, including not just their number and currency, but the details of their lot. Inventories consist of a list of lots, and as such are rendered similarly, as a list of positions (one per line) each with their full detail by default. This is generally too much detail.

The shell provides functions that allow the user to summarize the positions into one of the various derived quantities. The types of derived quantities are:

- "raw": render the position in its full detail, including cost and lot date
- "units": render just the number and currency of the position
- "cost": render the total cost of the position, that is the number of units x the per-unit cost
- "weight": render the amount that is used to balance the postings of a transaction. The main distinction between cost and weight is for postings with a price conversion.
- "value": render the amount at the market value of the last entry rendered.

Functions with the same names are available to operate on position or inventory objects. For example, one could generate a table of final balances for each account like this:

```
SELECT account, units(sum(position)), cost(sum(position)) GROUP BY 1;
```

Refer to the table below for explicit examples of each type of posting and how it would get converted and rendered.

Operators

Common comparison and logical operators are provided to operate on the available data columns:

- = (equality), != (inequality)
- < (less than), <= (less than or equal)
- > (greater than), >= (greater than or equal)
- AND (logical conjunction)
- OR (logical disjunction)
- NOT (logical negation)
- IN (set membership)

We also provide a regular expression search operator into a string object:

- ~ (search regexp)

At the moment, matching groups are ignored.

You can use string, number and integer constants with those operators, and parentheses to explicitly state precedence. You can use the #"..." literal syntax to input dates (valid contents for the string are pretty liberal, it supports anything Python's dateutil.parser supports).

Here is an example query that uses a few of these:

```
SELECT date, payee
WHERE account ~ 'Expenses:Food:Restaurant'
  AND 'trip-new-york' IN tags
  AND NOT payee = 'Uncle Boons'
```

Unlike SQL, bean-query does not implement three-valued logic for NULL. This means that e.g. the expression NULL = NULL yields TRUE instead of NULL, which simplifies things, but may come as a surprise to veteran SQL users.

Simple Functions

The shell provides a list of simple function that operate on a single data column and return a new value. These functions operate on particular types. The shell implements rudimentary type verification and should be able to warn you on incompatible types.

Some example functions follow:

- COST(Inventory), COST(Position): Return an Amount, the cost of the position or inventory.
- UNITS(Inventory), UNITS(Position): Return the units of the position or inventory.
- DAY(date), MONTH(date), YEAR(date): Return an integer, the day, month or year of the posting or entry's date.
- LENGTH(list): Computes the length of a list or a set, e.g. on tags.
- PARENT(account-string): Returns the name of the parent account.

These are just examples; for the complete list, see "help targets", "help where", "help from".

Note that it is exceedingly easy to add new functions to this list. As of December 2014, we are just beginning using the shell widely and we expect to be adding new functions as needed. If you need a function, please add a comment here or log a ticket and we will consider adding it to the list (we understand that the current list is limited). I intend to be liberal about adding new functions; as long as they have generic application, I don't think it should be a problem. Otherwise, I may be able to provide a mechanism for user to register new functions as part of Python plugins that could live outside the Beancount codebase.

Aggregate Functions

Some functions operate on more than a single row. These functions aggregate and summarize the multiple values for the data column that they operate on. A prototypical usage of such a function is to sum the positions in an inventory:

```
SELECT account, sum(position) WHERE account ~ 'Income' GROUP BY account;
```

If a query target has at least one aggregating function, the query becomes an aggregated query (see relevant section for details). Note that you cannot use aggregation functions in the FROM or WHERE clauses.

Examples of aggregate functions include:

- COUNT(...): Computes the number of postings selected (an integer).
- FIRST(...), LAST(...): Returns first or last value seen.
- MIN(...), MAX(...): Computes the minimum or maximum value seen.
- SUM(...): Sums up the values of each set. This works on amounts, positions, inventories, numbers, etc.

As for simple functions, this is just a starting list. We will be adding more as needed. Use "help targets" to access the full list of available aggregate functions.

Note: You cannot filter (using a WHERE clause) the results of aggregation functions; this requires the implementation an HAVING clause, and at the moment, HAVING filtering is not yet implemented.

Simple vs. Aggregated Queries

There are two types of queries:

- **Simple queries**, which produce a row of results for each posting that matches the restricts in the WHERE clause.
- **Aggregate queries**, which produce a row of results for each *group* of postings that match the restricts in the WHERE clause.

A query is "aggregate" if it has at least one aggregate function in its list of targets. In order to identify the aggregation keys, all the non-aggregate columns have to be flagged using the GROUP BY clause, like this:

```
SELECT payee, account, COST(SUM(position)), LAST(date)
GROUP BY payee, account;
```

You may also use the positional order of the targets to declare the group key, like this:

```
SELECT payee, account, COST(SUM(position)), LAST(date)
GROUP BY 1, 2;
```

Furthermore, if you name your targets, you can use the explicit target names:

```
SELECT payee, account as acc, COST(SUM(position)), LAST(date)
GROUP BY 1, acc;
```

This should all feel familiar if you have preliminary knowledge of SQL.

Finally, because we implement a limited version of SQL, and that the simple columns must always be specified, omitting the GROUP BY clause should also eventually work and we should group by those columns implicitly, as a convenience.

Distinct

There is a post-filtering phase that supports uniquifying result rows. You can trigger this unique filter with the DISTINCT flag after SELECT, as is common in SQL, e.g.

```
SELECT DISTINCT account;
```

Controlling Results

Order By

Analogous to the GROUP BY clause is an ORDER BY clause that controls the final ordering of the result rows:

```
SELECT …
GROUP BY account, payee
ORDER BY payee, date;
```

The clause is optional. If you do not specify it, the default order of iteration of selected postings is used to output the results (that is, the order of transactions-sorted by date- and then their postings).

As in SQL, you may reverse the order of sorting by a DESC suffix (the default is the same as specifying ASC):

```
SELECT …
GROUP BY account, payee
ORDER BY payee, date DESC;
```

Limit

Our query language also supports a LIMIT clause to interrupt output row generation:

```
SELECT … LIMIT 100;
```

This would output the first 100 result rows and then stop. While this is a common clause present in the SQL language, in the context of double-entry bookkeeping it is not very useful: we always have relatively small datasets to work from. Nevertheless, we provide it for completeness.

Format

For SELECT, JOURNAL and BALANCES queries, the output format is a table of text by default. We support CSV output. (*We could easily add support for XLS or Google Sheets output.)*

However, for PRINT queries, the output format is Beancount input text format.

Statement Operators

The shell provides a few operators designed to facilitate the generation of balance sheets and income statements. The particular methodology used to define these operations should be described in detail in the "introduction to double-entry bookkeeping" document that accompanies Beancount and is mostly located in the source code in the summarize module.

These special operators are provided on the FROM clause that is made available on the various forms of query commands in the shell. These further transform the set of entries selected by the FROM expression at the transaction levels (not postings).

Please note that these are not from standard SQL; these are extensions provided by this shell language only.

Opening a Period

Opening an exercise period at a particular date replaces all entries before that date by summarization entries that book the expected balance against an Equity "opening balances" account and implicitly clears the income and expenses to zero by transferring their balances to an Equity "previous earnings" account (see beancount.ops.summarize.open() for implementation details).

It is invoked like this:

```
SELECT … FROM <expression> OPEN ON <date> …
```

For example:

```
SELECT * FROM has_account("Invest") OPEN ON 2014-01-01;
```

If you want, you can view just the inserted summarization entries like this:

```
PRINT FROM flag = "S" AND account ~ "Invest" OPEN ON 2014-01-01;
```

Closing a Period

Closing an exercise period involves mainly truncating all entries that come *after* the given date and ensuring that currency conversions are correctly corrected for (see beancount.ops.summarize.close() for implementation details).

It is invoked like this:

```
SELECT … FROM <expression> CLOSE [ON <date>] …
```

For example:

```
SELECT * FROM has_account("Invest") CLOSE ON 2015-04-01;
```

Note that the closing date should be one day after the last transaction you would like to include (this is in line with the convention we use everywhere in Beancount whereby starting dates are inclusive and ending dates exclusive).

The closing date is optional. If the date is not specified, the date one day beyond the date of the last entry is used.

Closing a period leaves the Income and Expenses accounts as they are, that is, their balances are not cleared to zero to Equity. This is because closing is also used to produce final balances for income statements. "Clearing", as described in the next section, is only needed for balance sheets.

Clearing Income & Expenses

In order to produce a balance sheet, we need to transfer final balances of the Income and Expenses to an Equity "current earnings" account (sometimes called "retained earnings" or "net income"; you can select the specific account name to use using options in the input file). The resulting balances of income statement accounts should be zero. (see beancount.ops.summarize.clear() for implementation details.)

You can clear like this:

```
SELECT … FROM <expression> CLEAR …
```

For example:

```
SELECT * FROM has_account("Invest") CLOSE ON 2015-04-01 CLEAR;
```

This is a statement suitable to produce a list of accounts to build a balance sheet. The "Equity:Earnings:Current" (by default) will contain the net income accumulated during the preceding period. No balances for the Income nor Expenses accounts should appear in the output.

Example Statements

The statement operators of course may be combined. For instance, if you wanted to output data for an income statement for year 2013, you could issue the following statement:

```
SELECT account, sum(position)
FROM OPEN ON 2013-01-01 CLOSE ON 2014-01-01
WHERE account ~ "Income|Expenses"
GROUP BY 1
ORDER BY 1;
```

This would produce a list of balances for Income and Expenses accounts.

To generate a balance sheet, you would add the CLEAR option and select the other accounts:

```
SELECT account, sum(position)
FROM OPEN ON 2013-01-01 CLOSE ON 2014-01-01 CLEAR
WHERE not account ~ "Income|Expenses"
GROUP BY 1
ORDER BY 1;
```

Note that if you added the CLEAR operator to the statement of income statement accounts, all the balances would show at zero because of the inserted transactions that move those balances to the Equity net income account at the end of the period.

It is relevant to notice that the examples above do not filter the transactions any further. If you are selecting a subset of transactions you may want to leave the accounts unopened, unclosed and uncleared because applying only some of the transactions on top of the opening balances of Assets and Liabilities accounts will not produce correct balances for those accounts. It would be more useful to leave them all opened, and to interpret the balances of the balance sheet accounts as the *changes* in those accounts for the subset of transactions selected. For example, if you selected all transactions from a trip (using a tag), you would obtain a list of changes in Expenses (and possibly Income) tagged as being included in this trip, and the Assets and Liabilities accounts would show where the funds for those Expenses came from.

Consult the "introduction to double-entry method" document for a pictorial representation of this. (Granted, this is probably worth of a dedicated document and I might produce one at some point.)

Example Fetching Cost Basis

*"... is there currently an easy way to determine what my cost basis is for an account on a given date (other than manually adding up UNITS * COST for every contribution, which is kind of a pain)? I'm trying to estimate the tax implications of potential stock sales." [Question from Matthew Harris]*

For a detailed report of share movements:

```
SELECT account, currency, position, COST(position)
WHERE year <= 2015
  AND account ~ "Assets:US:Schwab"
  AND currency != "USD"
```

Or this for the sum total of the cost bases:

```
SELECT sum(cost(position))
WHERE year <= 2015
```

```
AND account ~ "Assets:US:Schwab"
AND currency != "USD"
```

High-Level Shortcuts

There are two types of queries that are very common for accounting applications: journals and balances reports. While we have explicit implementations of such reports that can be produced using the bean-report tool, we are also able to synthesize good approximations of such reports using SELECT statements. This section describes a few additional selection commands that translate directly into SELECT statements and which are then run with the same query code. These are intended as convenient shortcuts.

Selecting Journals

A common type of query is one that generates a linear journal of entries (Ledger calls this a "register"). This roughly corresponds to an account statement, but with our language, such a statement can be generated for any subset of postings.

You can generate a journal with the following syntax:

```
JOURNAL <account-regexp> [AT <function>] [FROM …]
```

The regular expression *account-regexp* is used to select which subset of accounts to generate a journal for. The optional "AT *<function>*" clause is used to specify an aggregation function for the amounts rendered (typically UNITS or COST). The FROM clause follows the same rules as for the SELECT statement and is optional.

Here is an example journal-generating query:

```
JOURNAL "Invest" AT COST FROM HAS_ACCOUNT("Assets:US");
```

Selecting Balances

The other most common type of report is a table of the balances of various accounts at a particular date. This can be viewed as a SELECT query aggregating positions grouping by account.

You can generate a balances report with the following syntax:

```
BALANCES [AT <function>] [FROM …]
```

The optional "AT *<function>*" clause is used to specify an aggregation function for the balances rendered (usually UNITS or COST). The FROM clause follows the same rules as for the SELECT statement and is optional.

To generate your balances at a particular date, close your set of entries using the "FROM… CLOSE ON" form described above.

Observe that typical balance sheets and income statements seen in an accounting context are subsets of tables of balances such as reported by this query. An income statement reports on just the transactions that appears during a period of time, and a balance sheet summarizes transactions before its reporting before and clears the income & expenses accumulated during the period to an equity account. Then some minor reformatting is carried out. Please consult the introduction document on double-entry bookkeeping for more details, and the section above that discusses the "open", "close" and "clear" operations.

We will also be providing a separate text processing tool that can accept balance reports and reformat them in a two-column format similar to that you would see balance sheets and income statements.

Print

It can be useful to generate output in Beancount format, so that subsets of transactions can be saved to files, for example. The shell provides that ability via the PRINT command:

```
PRINT [FROM …]
```

The FROM clause obeys the usual semantics as described elsewhere in this document. The resulting filtered stream of Beancount entries is then printed out on the output in Beancount syntax.

In particular, just running the "PRINT" command will spit out the parsed and loaded contents of a Beancount file. You can use this for troubleshooting if needed, or to expand transactions generated from a plugin you may be in the process of developing.

Debugging / Explain

If you're having trouble getting a particular statement to compile and run,. you can prefix any query statement with the EXPLAIN modifier, e.g.:

```
EXPLAIN SELECT …
```

This will not run the statement, but rather print out the intermediate AST and compiled representation as well as the list of computed statements. This can be useful to report bugs on the mailing-list.

Also, this shows you the translated form of the JOURNAL and BALANCES statements.

Future Features

The following list of features were planned for the first release but I've decided to make a first cut without them. I'll be adding those during a revision.

Flattening Inventories

If you provide the FLATTEN option after a query, it tells the query engine to flatten inventories with multiple lots into separate rows for each lot. For example, if you have an inventory balance with the following contents:

3 AAPL {102.34 USD} 4 AAPL {104.53 USD} 5 AAPL {106.23 USD}

Using the following query:

```
SELECT account, sum(position) GROUP BY account;
```

It should return a single row of results, rendered over three lines. However, adding the option:

```
SELECT account, sum(position) GROUP BY account FLATTEN;
```

This should return three separate rows, with all the selected attributes, as if there were that many postings.

Sub-Selects

The ability to select from the result of another SELECT is not currently supported, but the internals of the query code are prepared to do so.

Pivot By

A special PIVOT BY clause can be used to convert the output from a one-dimensional list of results to a two-dimensional table. For example, the following query:

```
SELECT
  account,
  YEAR(date) AS year,
  SUM(COST(position)) AS balance
WHERE
   account ~ 'Expenses:Food' AND
   currency = 'USD' AND
   year >= 2012
GROUP BY 1,2
ORDER BY 1,2;
```

Might generate the following table of results:

```
        account          year   balance
----------------------- ---- -----------
Expenses:Food:Alcohol    2012   57.91 USD
Expenses:Food:Alcohol    2013   33.45 USD
Expenses:Food:Coffee     2012   42.07 USD
```

```
Expenses:Food:Coffee     2013  124.69 USD
Expenses:Food:Coffee     2014   38.74 USD
Expenses:Food:Groceries  2012 2172.97 USD
Expenses:Food:Groceries  2013 2161.90 USD
Expenses:Food:Groceries  2014 2072.36 USD
Expenses:Food:Restaurant 2012 4310.60 USD
Expenses:Food:Restaurant 2013 5053.61 USD
Expenses:Food:Restaurant 2014 4209.06 USD
```

If you add a PIVOT clause to the query, like this:

```
…
PIVOT BY account, year;
```

You would get a table like this:

```
      account/year              2012        2013        2014
------------------------ ----------- ----------- -----------
Expenses:Food:Alcohol      57.91 USD   33.45 USD
Expenses:Food:Coffee       42.07 USD  124.69 USD   38.74 USD
Expenses:Food:Groceries  2172.97 USD 2161.90 USD 2072.36 USD
Expenses:Food:Restaurant 4310.60 USD 5053.61 USD 4209.06 USD
```

If your table has more than three columns, the non-pivoted column would get expanded horizontally, like this:

```
SELECT
  account,
  YEAR(date),
  SUM(COST(position)) AS balance,
  LAST(date) AS updated
WHERE
   account ~ 'Expenses:Food' AND
   currency = 'USD' AND
   year >= 2012
GROUP BY 1,2
ORDER BY 1, 2;
```

You would get a table like this:

```
        account          year   balance    updated
------------------------ ---- ----------- ----------
Expenses:Food:Alcohol    2012   57.91 USD 2012-07-17
Expenses:Food:Alcohol    2013   33.45 USD 2013-12-13
Expenses:Food:Coffee     2012   42.07 USD 2012-07-19
Expenses:Food:Coffee     2013  124.69 USD 2013-12-16
Expenses:Food:Coffee     2014   38.74 USD 2014-09-21
Expenses:Food:Groceries  2012 2172.97 USD 2012-12-30
Expenses:Food:Groceries  2013 2161.90 USD 2013-12-31
Expenses:Food:Groceries  2014 2072.36 USD 2014-11-20
```

```
Expenses:Food:Restaurant 2012 4310.60 USD 2012-12-30
Expenses:Food:Restaurant 2013 5053.61 USD 2013-12-29
Expenses:Food:Restaurant 2014 4209.06 USD 2014-11-28
```

Pivoting, this would generate this table:

```
account/balance,updated  2012/balanc 2012/updat 2013/balanc 2013/updat 2014/balanc 2014/upda
------------------------ ---------- ---------- ---------- ---------- ---------- ---------
Expenses:Food:Alcohol      57.91 USD 2012-07-17   33.45 USD 2013-12-13
Expenses:Food:Coffee       42.07 USD 2012-07-19  124.69 USD 2013-12-16   38.74 USD 2014-09-2
Expenses:Food:Groceries  2172.97 USD 2012-12-30 2161.90 USD 2013-12-31 2072.36 USD 2014-11-2
Expenses:Food:Restaurant 4310.60 USD 2012-12-30 5053.61 USD 2013-12-29 4209.06 USD 2014-11-2
```

More Information

This document attempts to provide a good high-level summary of the features supported in our query language. However, should you find you need more information, you may take a look at the original proposal, or consult the source code under the beancount.query directory. In particular, the parser will provide insight into the specifics of the syntax, and the environments will shed some light on the supported data columns and functions. Feel free to rummage in the source code and ask questions on the mailing-list.

Beancount Syntax Cheat Sheet

How Inventories Work

Martin Blais, December 2016

http://furius.ca/beancount/doc/booking

This document explains how we accumulate commodities and match sales (reductions) against accumulated inventory contents.

Introduction

Beyond the ability to track and list the postings made to each of the accounts (an operation that produces a *journal* of entries), one of the most common and useful operations of Beancount is to sum up the positions of arbitrary sets of postings. These aggregations are at the heart of how Beancount works, and are implemented in an object called “inventory.” This document explains how this aggregation process works. If you’re going to track investments, it’s necessary to understand what follows.

Matches & Booking Methods

In order to get the big picture, let's walk through the various booking features by way of a simple examples. This should expose you to all the main ideas in one go.

Simple Postings — No Cost

Consider a set of simple postings to an account, e.g.,

```
2016-04-24 * "Deposit check"
  Assets:Bank:Checking      221.23 USD
  …

2016-04-29 * "ATM Withdrawal"
  Assets:Bank:Checking     -100.00 USD
  …

2016-04-29 * "Debit card payment"
  Assets:Bank:Checking      -45.67 USD
  …
```

The inventory of the Checking account begins empty. After the first transaction, its contents are 221.23 USD. After the second, 121.23 USD. Finally, the third transaction brings this balance to 75.56 USD. This seems very natural; the numbers simply add to.

It might be obvious, but note also that the numbers are allowed to change the sign (go negative).

Multiple Commodities

An inventory may contain more than one type of commodity. It is equivalent to a mapping from commodity to some number of units. For example,

```
2016-07-24 * "Dinner before leaving NYC"
  Expenses:Restaurants     34.58 USD
  …

2016-07-26 * "Food with friends after landing"
  Expenses:Restaurants     62.11 CAD
  …
```

After those two transactions, the Restaurants account contains 34.58 USD and 62.11 CAD. Its contents are said to be of mixed commodities. And naturally, postings are applied to just the currencies they affect. For instance, the following transaction

```
2016-07-27 * "Brunch"
  Expenses:Restaurants     23.91 CAD
```

```
  …
```

brings the balance of that account to 34.58 USD and 86.02 CAD. The number of units USD hasn't changed.

Note that accounts may contain any number of commodities, and this is also true for commodities held at cost, which we'll see shortly. While this is made possible, I recommend that you define enough accounts to keep a single commodity in each; this can be enforced with the "onecommodity" plugin.

Cost Basis

Things get a little more hairy when we consider the tracking of investments with a cost basis. Beancount allows you to associate a cost basis and an optional label with a particular lot acquired. Consider these two purchases to an investment account:

```
2015-04-01 * "Buy some shares of Hooli in April"
  Assets:Invest            25 HOOL {23.00 USD, "first-lot"}
  …

2015-05-01 * "Buy some more shares of Hooli in May"
  Assets:Invest            35 HOOL {27.00 USD}
  …
```

So now, the investment account's inventory contains

```
          units ccy   cost  cost-ccy lot-date    label
             25 HOOL {23.00 USD,     2015-04-01, "first-lot"}
             35 HOOL {27.00 USD,     2015-05-01, None}
```

Those two lots were not merged, they are still two distinct positions in the inventory. Inventories merge lots together and adjust the units only if the commodity and *all* of its cost attributes exactly match. (In practice, it's pretty rare that two augmentations will have the same cost and date attributes.)

Note how Beancount automatically associated the acquisition date to each lot; you can override it if desired, by adding the date similar to the optional label. this is useful for making cost basis adjustments).

Postings that *add* to the content of an inventory are called **augmentations**.

Reductions

But how do we remove commodities from an inventory?

You could eat away at an existing lot by selling some of it. You do this by posting a **reduction** to the account, like this:

```
2015-05-15 * "Sell some shares"
  Assets:Invest           -12 HOOL {23.00 USD}
```

```
  …
```

Just to be clear, what makes this posting a **reduction** is the mere fact that the sign (-) is opposite that of the balance of the account (+25) for that commodity. This posting tells Beancount to find all lots with a cost basis of 23.00 USD and remove 12 units from it. The resulting inventory will be

```
          units ccy   cost  cost-ccy lot-date    label
             13 HOOL {23.00 USD,     2015-04-01, "first-lot"}
             35 HOOL {27.00 USD,     2015-05-01, None}
```

Note how the first posting was reduced by 12 units. We didn't have to specify all of the lot's attributes, just the cost. We could have equivalently used the date to specify which lot to reduce:

```
2015-05-15 * "Sell some shares"
  Assets:Invest           -12 HOOL {2015-04-01}
  …
```

Or the label:

```
2015-05-15 * "Sell some shares"
  Assets:Invest           -12 HOOL {"first-lot"}
  …
```

Or a combination of these. Any combination of attributes will be matched against the inventory contents to find which lot to reduce. In fact, if the inventory happened to have just a single lot in it, you could reduce it like this:

```
2015-05-15 * "Sell some shares"
  Assets:Invest           -12 HOOL {}
  …
```

Ambiguous Matches

But what happens if multiple lots match the reduction? For example, with the previous inventory containing two lots, if you wrote your sale like this:

```
2015-05-15 * "Sell some shares"
  Assets:Invest           -12 HOOL {}
  …
```

Beancount wouldn't be able to figure out which lot needs to get reduced. We have an ambiguous match.

Partially ambiguous matches are also possible. For example, if you have the following inventory to reduce from:

```
          units ccy   cost  cost-ccy lot-date    label
             25 HOOL {23.00 USD,     2015-04-01, None}
             30 HOOL {25.00 USD,     2015-04-01, None}
             35 HOOL {27.00 USD,     2015-05-01, None}
```

And you attempted to reduce like this:

```
2015-05-15 * "Sell some shares"
  Assets:Invest           -12 HOOL {2015-04-01}
  …
```

The first two lots are selected as matches.

Strict Booking

What does Beancount do with ambiguous matches? By default, it issues an error.

More precisely, what happens is that Beancount invokes the **booking method** and it handles the ambiguous match depending on what it is set. The default booking method is "**STRICT**" and it just gives up and spits out an error, telling you you need to refine your input in order to disambiguate your inventory reduction.

FIFO and LIFO Booking

Other booking methods are available. They can be configured using options, like this:

```
option "booking_method" "FIFO"
```

The "**FIFO**" method automatically select the oldest of the matching lots up to the requested size of the reduction. For example, given our previous inventory:

```
          units ccy   cost  cost-ccy lot-date    label
             25 HOOL {23.00 USD,     2015-04-01, "first-lot"}
             35 HOOL {27.00 USD,     2015-05-01, None}
```

Attempting to reduce 28 shares like this:

```
2015-05-15 * "Sell some shares"
  Assets:Invest           -28 HOOL {}
  …
```

would match both lots, completely reduce the first one to zero units and remove the remaining 3 units from the second lot, to result in the following inventory:

```
          units ccy   cost  cost-ccy lot-date    label
             32 HOOL {27.00 USD,     2015-05-01, None}
```

The "LIFO" method works similarly, but consumes the youngest (latest) lots first, working its way backward in time to remove the volume.

Per-account Booking Method

You don't have to make all accounts follow the same booking method; the option in the previous section sets the default method for all accounts. In order

to override the booking method for a particular account, you can use an optional string option on the account's Open directive, like this:

```
2014-01-01 open Assets:Invest    "FIFO"
```

This allows you to treat different accounts with a different booking resolution.

Total Matches

There is an exception to strict booking: if the entire inventory is being reduced by exactly the total number of units, it's clear that all the matching lots are to be selected and this is considered unambiguous, even under "STRICT" booking. For example, under "STRICT" booking, this reduction would empty up the previous inventory without raising an error, because there are 25 + 35 shares matching:

```
2015-05-15 * "Sell all my shares"
  Assets:Invest           -60 HOOL {}
  …
```

Average Booking

Retirement accounts created by government incentive programs (such as the 401k plan in the US or the RRSP in Canada) typically consist in pre-tax money. For these types of accounts, brokers usually disregard the calculation of cost basis because the taxation is to be made upon distributing money outside the account. These accounts are often managed to the extent that they are fully invested; therefore, fees are often taken as shares of the investment portfolio, priced on the day the fee is paid out. This makes it awkward to track the cost basis of individual lots.

The correct way to deal with this is to treat the number of units and the cost basis of each commodity's entire set of lots separately. For example, the following two transactions:

```
2016-07-28 * "Buy some shares of retirement fund"
  Assets:Invest       45.0045 VBMPX {11.11 USD}
  …

2016-10-12 * "Buy some shares of retirement fund"
  Assets:Invest       54.5951 VBMPX {10.99 USD}
  …
```

Should result in a single lot with the total number of units and the averaged cost:

```
          units ccy    cost    cost-ccy lot-date    label
       99.5996 VBMPX {11.0442 USD,     2016-07-28, None}
```

A fee taken in this account might look like this:

```
2016-12-30 * "Quarterly management fee"
  Assets:Invest       -1.4154 VBMPX {10.59 USD}
  Expenses:Fees
```

Even with negative units the number and cost get aggregated separately:

```
          units ccy    cost    cost-ccy lot-date    label
        98.1842 VBMPX {11.0508 USD,     2016-07-28, None}
```

No Booking

However, there is another way to deal with non-taxable accounts in the meantime: you can simply disable the booking. There is a booking method called "**NONE**" which implements a very liberal strategy which accepts any new lot.. New lots are always appended unconditionally to the inventory. Using this strategy on the transactions from the previous section would result in this inventory:

```
          units ccy    cost  cost-ccy lot-date    label
        45.0045 VBMPX {11.11 USD,     2016-07-28, None}
        54.5951 VBMPX {10.99 USD,     2016-10-12, None}
        -1.4154 VBMPX {10.59 USD,     2016-12-30, None}
```

Observe how the resulting inventory has a mix of signs; normally this is not allowed, but it is tolerated under this degenerate booking method. Note that under this method, the only meaningful values are the *total* number of units and the *total* or *average* cost amounts. The individual lots aren't really lots, they only represent the list of all postings made to that account.

Note: If you are familiar with Ledger, this is the default and only booking method that it supports.

Summary

In summary, here's what we presented in the walkthrough. *Augmentations* are never problematic; they always add a new position to an existing inventory. On the other hand, *reductions* may result in a few outcomes:

- **Single match.** Only one position matches the reduction; it is reduced.
- **Total match.** The total number of units requested matches the total number of units of the positions matched. These positions are reduced away.
- **No match.** None of the positions matches the reducing posting. An error is raised.
- **Ambiguous matches.** More than one position in the inventory matches the reducing posting; *the booking method is invoked to handle this.*

There are a few booking methods available to handle the last case:

- **STRICT.** An error is raised.

- **FIFO.** Units from oldest (earliest) lots are selected until the reduction is complete.
- **LIFO.** Units from youngest (latest) lots are selected until the reduction is complete.
- **AVERAGE.** After every reduction, all the units of the affected commodity are merged and their new average cost is recalculated.
- **NONE.** Booking is disabled; the reducing lots is simply added to the inventory. This results in an inventory with mixed signs and only the total number of units and total cost basis are sensible numbers.

Beancount has a default booking method for all accounts, which can be overridden with an option:

```
option "booking_method" "FIFO"
```

The default value for the booking method is "STRICT". I recommend that you leave it that way and override the booking method for specific accounts.

The method can be specified for each account by adding a string to its Open directive:

```
2016-05-01 open Assets:Vanguard:RGAGX  "AVERAGE"
```

How Prices are Used

The short answer is that prices aren't used nor affect the booking algorithm at all. However, it is relevant to discuss what they do in this context because users invariably get confused about their interpretation.

There are two use cases for prices: making conversions between commodities and tagging a reducing lot with its sale price in order to record it and optionally balance the proceeds.

Commodity Conversions

Conversions are used to exchange one currency for another. They look like this:

```
2016-04-24 * "Deposit check"
  Assets:Bank:Checking      220.00 USD @ 1.3 CAD
  Income:Payment           -286.00 CAD
```

For the purpose of booking it against the Checking account's inventory, the posting with the price attached to it is treated just the same as if there was no price: the Checking account simply receives a deposit of 220.00 units of USD and will match against positions of commodity "USD". The price is used only to verify that the transaction balances and ensure the double-entry accounting rule is respected (220.00 x 1.3 CAD + -286.00 CAD = 0.00). It is otherwise ignored for the purpose of modifying the inventory contents. In a sense, after

the postings have been applied to the account inventories, the price is forgotten and the inventory balance retains no memory of the deposit having occurred from a conversion.

Price vs. Cost Basis

One might wonder how the price is used if there is a cost basis specification, like this:

```
2015-05-15 * "Sell some shares"
  Assets:Invest:HOOL           -12 HOOL {23.00 USD} @ 24.70 USD
  Assets:Invest:Cash        296.40 USD
  Income:Invest:Gains
```

The answer is often surprising to many users: the price is **not used** by the balancing algorithm if there is a cost basis; the cost basis is the number used to balance the postings. This is a useful property that allows us to compute capital gains automatically. In the previous example, the balance algorithm would sum up -12 x 23.00 + 296.40 = -20.40 USD, which is the capital gain, (24.70 - 23.00) * 12. It would complete the last posting and assign it this value.

In general, the way that profits on sales are calculated is by weighing the proceedings, i.e., the cash deposits, against the cost basis of the sold lots, and this is sufficient to establish the gain difference. Also, if an *augmenting* posting happens to have a price annotation on it, it is also unused.

The price is an annotation for your records. It remains attached to the Posting objects and if you want to make use of it somehow, you can always do that by writing some Python code. There are already two plugins which make use of this annotation:

- **beancount.plugins.implicit_prices**: This plugin takes the prices attached to the postings and automatically creates and inserts Price directives for each of them, in order to feed the global price database.
- **beancount.plugins.sellgains**: This plugin implements an additional balancing check: it uses the prices to compute the expected proceeds and weighs them against all the other postings of the transaction *excluding* any postings to Income accounts. In our example, it would check that (-12 x 24.70 + 296.40) = 0. This provides yet another means of verifying the correctness of your input.

See the Trading with Beancount document for more details on this topic.

Trades

The combination of acquiring some asset and selling it back is what we call a "trade." In Beancount we consider only assets with a cost basis to be the subject of trades. Since booking reductions against accumulated inventory contents

happens during the booking process, this is where trades should be identified and recorded.

The way trades will be implemented is by allowing the booking process to insert matching metadata with unique UUIDs on both the augmenting and reducing postings, in the stream of transactions. Functions and reports will be provided that are able to easily extract the pairs of postings for each reducing postings and filter those out in different ways. Ultimately, one should be able to extract a list of all trades to a table, with the acquisition and sale price, as well as other fees.

Debugging Booking Issues

If you're experiencing difficulties in recording your sales due to the matching process, there are tools you can use to view an account's detailed inventory contents before and after applying a Transaction to it. To do this, you can use the bean-doctor command. You invoke the program providing it with the file and line number close to the Transaction you want to select, like this:

```
bean-doctor context <filename> <line-no>
```

The resulting output will show the list of inventory contents of all affected accounts prior to the transaction being applied, including cost basis, acquisition date, and optional label fully rendered. Note that some failures are typically handled by throwing away a invalid Transaction's effects (but never quietly).

From Emacs or VI, placing the cursor near a transaction and invoking the corresponding command is the easiest way to invoke the command, as it inserts the line number automatically.

Appendix

The rest of this document delves into more technical details. You should feel free to ignore this entirely, it's not necessary reading to understand how Beancount works. Only bother if you're interested in the details.

Data Representation

It is useful to know how positions are represented in an inventory object. A *Position* is essentially some number of units of a commodity with some optional information about its acquisition:

- **Cost.** Its per-unit acquisition cost (the "cost basis").
- **Date.** The date at which the units were acquired.
- **Label.** Some user-specified label which can be used to refer to the lot).

We often refer to these position objects as "lots" or "legs." Schematically, a position object looks like this:

The structure of a *Position* object.

There are two different types of positions, discussed in detail in the sections that follow:

- **Simple positions.** These are positions with no cost basis. The "cost" attribute is set to a null value. ("None" in Python.)
- **Positions held at cost.** These are positions with an associated cost basis and acquisition details.

An *Inventory* is simply an accumulation of such positions, represented as a list. We sometimes talk of the *ante-inventory* to refer to the contents of the inventory before a transaction's postings have been applied to it, and the *ex-inventory* to the resulting inventory after they have been applied.

A *Posting* is an object which is a superset of a position: in addition to units and cost, it has an associated account and an optional price attributes. If present, the price has the same type as units. It represents one of the legs of a transaction in the input. Postings imply positions, and these positions are added to inventories. We can say that a position is *posted* to an account.

For more details on the internal data structures used in Beancount, please refer to the Design Doc which expands on this topic further.

Why Booking is Not Simple

The complexity of the reduction process shows itself when we consider how to keep track of the cost basis of various lots. To demonstrate how this works, let us consider a simple example that we shall reuse in the different sections below:

- 25 shares are bought on 4/1 at $23/share.
- 35 shares are bought on 5/1 at $27/share.
- 30 shares are sold on 5/15; at that time, the price is $26/share.

We'll ignore commissions for now, they don't introduce any additional complexity. In Beancount, this scenario would be recorded like this:

```
2015-04-01 * "Buy some shares of Hooli in April"
  Assets:Invest:HOOL            25 HOOL {23.00 USD}
  Assets:Invest:Cash       -575.00 USD

2015-05-01 * "Buy some more shares of Hooli in May"
  Assets:Invest:HOOL            35 HOOL {27.00 USD}
  Assets:Invest:Cash       -945.00 USD

2015-05-15 * "Sell 30 shares"
  Assets:Invest:HOOL           -30 HOOL {...} @ 26.00 USD
  Assets:Invest:Cash        780.00 USD
```

```
  Income:Invest:HOOL:Gains
```

Now, the entire question revolves around *which* of the shares are selected to be sold. I've rendered this input as a red ellipsis ("…"). Whatever the user puts in that spot will be used to determine which lot we want to use.

Whichever lot(s) we elect to be the ones sold will determine the amount of gains, because that is a function of the cost basis of those shares. This is why this matters.

Augmentations vs. Reductions

The most important observation is that there are two distinct kinds of lot specifications which look very similar in the input but which are processed very differently.

When we buy, as in the first transaction above, the {...} cost basis syntax provides Beancount with information about a *new* lot:

```
2015-05-01 * "Buy some more shares of Hooli in May"
  Assets:Invest:HOOL            35 HOOL {27.00 USD}
  Assets:Invest:Cash       -945.00 USD
```

We call this an "augmentation" to the inventory, because it will simply add a new position to it. The **cost basis** that you provide is attached to this position and preserved through time, in the inventory. In addition, there are a few other pieces of data you can provide for an augmenting lot. Let's have a look at all the data that can be provided in the cost spec:

- **Cost basis.** This consists in per-unit and total cost numbers—which are combined into a single per-unit number—and a currency.
- **Acquisition date.** A lot has an acquisition date. By default, the date attached of its parent transaction will be set as its acquisition date automatically. You may override this date by providing one. This comes in handy to handle stock splits or wash sales and preserve the original acquisition date of the replacement shares, as we'll see later.
- **Label.** You can provide a unique label for it, so that you can more easily refer to it later on, when you sell some or all of it.
- **Merge.** An indicator (a flag) that the lot should be merged (this will be useful for average cost booking which will be implemented later).

For an **augmenting postings**, these informations must be either provided or inferred automatically. They can be provided in any order:

```
2015-05-01 * "Buy some more shares of Hooli in May"
  Assets:Invest:HOOL            35 HOOL {2015-04-25, 27.00 USD, "hooli-123"}
  Assets:Invest:Cash       -945.00 USD
```

If you omit the date, the date of the Transaction is attached to it. If you omit the cost, the rest of the postings must be filled in such that the cost amount can be inferred from them. Since the label is optional anyway, an unspecified label field is left as a null value.

You might wonder why it is allowed to override the date of an augmentation; it is useful when making cost basis adjustments to preserve the original acquisition date of a posting: You remove the posting, and then replace it with its original date and a new cost basis.

Now, when we sell those shares, we will refer to the posting as a **"reducing" posting**, a "**reduction**". Note that the terms "augmenting" and "reducing" are just terminology I'm came up with in the context of designing how Beancount processes inventories; they're not general accounting terms.

It's a "reduction" because we're removing shares from the inventory that has been accumulated up to the date of its transaction. For example, if we were to sell 30 shares from that lot of 35 shares, the input might look like this:

```
2015-05-15 * "Sell 30 shares"
  Assets:Invest:HOOL           -30 HOOL {27.00 USD} @ 26.00 USD
  Assets:Invest:Cash        780.00 USD
  Income:Invest:HOOL:Gains
```

While the input looks the same as on the augmenting posting, Beancount handles this quite differently: it looks at the state of the account's inventory before applying the transaction and finds all the positions that match the lot data you provided. It then uses the details of the matched lots as the cost basis information for the reducing posting.

In this example, it would simply match all the positions which have a cost basis of $27.00. This example's inventory before the sale contains a single lot of 35 shares at $27.00, so there is a single position matching it and that lot is reduced by 30 shares and 5 shares remain. We'll see later what happens in the case of multiple lots matching the specification.

Note that you could have provide other subsets of lot information to match against, like just providing the label, for example:

```
2015-05-15 * "Sell 30 shares"
  Assets:Invest:HOOL           -30 HOOL {"hooli-123"} @ 26.00 USD
  Assets:Invest:Cash        780.00 USD
  Income:Invest:HOOL:Gains
```

This is also a valid way to identify the particular lot you wish to reduce. If you had provided a date here, it would also only be used to match against the inventory contents, to disambiguate between lots acquired at different dates, not to attach the date anywhere. And furthermore, if there was a single lot in the inventory you could have also just provided just an empty cost basis spec like

this: "{}". The Booking Methods section below will delve into the detail of what happens when the matches are ambiguous.

In summary:

- When you're adding something to an account's inventory (augmenting), the information you provide is used to create a new lot and is attached to it.
- When you're removing from an account's inventory (reducing), the information you provide is used to filter the inventory contents to select which of the lot(s) to reduce, and information from the selected lots is filled in.

Homogeneous and Mixed Inventories

So far in the example and in the vast majority of the examples in the documentation, "augmenting" means adding a positive number of shares. But in Beancount many of the accounts normally have a negative balance, e.g., liabilities accounts. It's fair to ask if it makes sense to hold a negative balance of commodities held at cost.

The answer is yes. These would correspond to "short" positions. Most people are unlikely to be selling short, but Beancount inventories support it. How we define "augmenting" is in relation to the existing balance of lots of a particular commodity. For example, if an account's inventory contains the following positions:

```
 25 HOOL {23.00 USD, 2016-04-01}
 35 HOOL {27.00 USD, 2016-05-01}
```

Then "adding" means a positive number of shares. On the other hand, if the account contains only short positions, like this:

```
-20 HOOL {23.00 USD, 2016-04-15}
-10 HOOL {27.00 USD, 2016-05-15}
```

Then "adding" means a negative number of shares, and "reducing" would be carried out by matching a positive number of shares against it.

The two inventories portrayed above are homogeneous in units of HOOL, that is, all of the positions have the same sign. With of the most booking methods we will see further, Beancount makes it impossible to create a non-homogeneous, or "mixed," inventory. But the "NONE" method allows it. A mixed inventory might have the following contents, for example:

```
 25 HOOL {23.00 USD, 2016-04-01}
-20 HOOL {23.00 USD, 2016-04-15}
```

As you may intuit, the notion of "augmenting" or "reducing" only makes sense for homogeneous inventories.

Original Proposal

If you're interested in the design doc that led to this implementation, you can find the document here. I hope the resulting implementation is simple enough yet general.

Exporting Your Portfolio

Martin Blais, December 2015 (v2)

http://furius.ca/beancount/doc/export

Overview

This document explains how to export your portfolio of holdings from Beancount to a Google Finance portfolio (and eventually to other portfolio tracking websites).

Note: This is the second version of this document, rewritten in Dec 2015, after greatly simplifying the process of exporting portfolios and completely separating the specification of stock tickers for price downloads. This new, simplified version only uses a single metadata field name: "export". The previous document can be found here.

Portfolio Tracking Tools

There are multiple websites on the internet that allow someone to create a portfolio of investments (or upload a list of transactions to creates such a portfolio) and that reports on the changes in the portfolio due to price movements, shows you unrealized capital gains, etc. One such website is the Google Finance portal. Another example is the Yahoo Finance one. These are convenient because they allow you to monitor the impact of price changes on your entire portfolio of assets, across all accounts, during the day or otherwise.

However, each of these sites expects their users to use their interfaces and workflows to painfully enter each of the positions one-by-one. A great advantage of using Beancount is that you should never have to enter this type of information manually; instead, you should be able to extract it and upload it to one of these sites. You can be independent of the particular portfolio tracking service you use and should be able to switch between them without losing any data; Beancount can serve as your pristine source for your list of holdings as your needs evolve.

Google Finance supports an "import" feature to create portfolio data which supports the Microsoft OFX financial interchange data format. In this document, we show how we built a Beancount report that exports the portfolio of holdings to OFX for creating a Google Finance portfolio.

Exporting to Google Finance

Exporting your Holdings to OFX

First, create an OFX file corresponding to your Beancount holdings. You can use this command to do this:

```
bean-report file.beancount export_portfolio > portfolio.ofx
```

See the report's own help for options:

```
bean-report file.beancount export_portfolio --help
```

Importing the OFX File in Google Finance

Then we have to import that OFX file in a web-based portfolio.

1. **Visit http://finance.google.com and click on "Portfolios" on the left (or simply visit https://www.google.com/finance/portfolio, this works as of Jan 2015)**

2. If you have an existing, previously imported portfolio, click on "Delete Portfolio" to get rid of it.

3. Click on "Import Transactions", then "Choose File" and select the *portfolio.ofx* file you exported to, then click on "Preview Import".

4. You should see a list of imported lots, with familiar stock symbols and names, and Type "Buy" with realistic Shares and Price columns. If not, see the note below. Otherwise, scroll to the bottom of the page and click "Import".

5. Your portfolio should now appear. You are done.

You should never bother updating this portfolio directly using the website… instead, update your Beancount ledger file, re-export to a new OFX file, **delete** the previous portfolio and re-import a brand new one over it. Your pristine source is always your Beancount file, ideally you should never have to be worried about corrupting or deleting the portfolio data in any external website.

Controlling Exported Commodities

Declaring Your Commodities

Generally, we recommend that you explicitly declare each of the commodities used in your input file. It is a neat place to attach information about those commodities, metadata that you should be able to use later on from bean-query or in scripts that you make. For example, you could declare a human-readable description of the commodity, and some other attributes, like this:

```
2001-09-06 commodity XIN
  name: "iShares MSCI EAFE Index ETF (CAD-Hedged)"
  class: "Stock"
  type: "ETF"
  ...
```

Beancount will work with or without these declarations (it automatically generates Commodity directives if you haven't provided them). If you like to be strict and have a bit of discipline, you can *require* that each commodity be declared by using a plugin that will issue an error when an undeclared commodity appears:

```
plugin "beancount.plugins.check_commodity"
```

You can use any date for that Commodity directive. I recommend using the date of the commodity's inception, or perhaps when it was first introduced by the issuing country, if it is a currency. You can find a suitable date on Wikipedia or on the issuer's websites. Google Finance may have the date itself.

What Happens by Default

By default, all holdings are exported as positions with a ticker symbol named the same as the Beancount commodity that you used to define them. If you have a holding of "AAPL" units, it will create an export entry for "AAPL". The export code attempts to export all holdings by default.

However, in any but the simplest unambiguous cases, this is probably not good enough to produce a working Google Finance portfolio. The name for each commodity that you use in your Beancount input file may or may not correspond to a financial instrument in the Google Finance database; due to the very large number of symbols supported in its database, just specifying the ticker symbol is often ambiguous. Google Finance attempts to resolve an ambiguous symbol string to the most likely instrument in its database. It is possible that it resolves it to a different financial instrument from the one you intended. So even if you use the same basic symbol that is used by the exchange, you often still need to disambiguate the symbol by specifying which exchange or symbology it lives in. Google provides a list of these symbol spaces.

Here is a real-life example. The symbol for the "CAD-Hedged MSCI EAFE Index" ETF product issued by iShares/Blackrock is "XIN" on the Toronto Stock Exchange (TSE). If you just looked up "XIN" on Google Finance, it would choose to resolve it by default to the more likely "NYSE:XIN" symbol (Xinyuan Real Estate Co. on the New York Stock Exchange). So you need to disambiguate it by specifying that the desired ETF ticker for this instrument is "TSE:XIN".

Explicitly Specifying Exported Symbols

You can specify which exchange-specific symbol is used to export a commodity by attaching an "export" metadata field to each of your Commodity directives, like this:

```
2001-09-06 commodity XIN
  ...
  export: "TSE:XIN"
```

The "export" field is used to map your commodity name to the corresponding instrument in the Google Finance system. If a holding in that commodity needs to be exported, this code is used instead of the Beancount currency name.

The symbology used by Google Finance appears to follow the following syntax:

Exchange:Symbol

where *Exchange* is a code either for the exchange where the stock trades, or for another source of financial data, e.g. "MUTF" for "mutual funds in the US", and more. *Symbol* is a name that is unique within that exchange. I recommend searching for each of your financial instruments in Google Finance, confirming that the instrument corresponds to your instrument (by inspecting the full name, description and price), and inserting the corresponding code like this.

Exporting to a Cash Equivalent

To account for positions that aren't supported in Google Finance, the export report can convert a holding to its cash-equivalent value. This is also useful for cash positions (e.g., cash sitting idle in a savings or checking account).

For example, I hold units of an insurance policy investment vehicle (this is common in Canada, for example, with London Life). This is a financial instrument, but each particular policy issuance has its own associated value—there is no public source of data for each of those products, it's rather opaque, I can obtain its value with my annual statement, but definitely not in Google Finance. But I'd still like for the asset's value to be reflected in my portfolio.

The way you tell the export code to make this conversion is to specify a special value of "CASH" for the "export" field, like this:

```
1878-01-01 commodity LDNLIFE
  export: "CASH"
```

This would convert holdings in LDNLIFE commodities to their corresponding quoted value before exporting, using the price nearest to the date of exporting. Note that attempting to convert to cash a commodity that does not have a corresponding cost or price available for us to determine its value will generate an error. A price must be present to make the conversion.

Simple currencies should also be marked as cash in order to be exported:

```
1999-01-01 commodity EUR
  name: "European Union Euro currency"
  export: "CASH"
```

Finally, all converted holdings are agglomerated into a single cash position. There is no point in exporting these cash entries to separate OFX entries because the Google Finance code will agglomerate them to a single one anyhow.

Declaring Money Instruments

There is a small hiccup in this cash conversion story: the Google Finance importer does not appear to correctly grok an OFX position in "cash" amounts in the importer; I think this is probably just a bug in Google Finance's import code (or perhaps I haven't found the correct OFX field values to make this work).

Instead, in order to insert a cash position the exporter uses a cash-equivalent commodity which always prices at 1 unit of the currency, e.g. $1.00 for US dollars. For example, for US dollars I I use VMMXX which is a Vanguard Prime Money Market Fund, and for Canadian dollars I use IGI806. A good type of commodity for this is some sort of Money Market fund. It doesn't matter so much which one you use, as long as it prices very close to 1. Find one.

If you want to include cash commodities, you need to find such a commodity for each of the cash currencies you have on your books and tell Beancount about them. Typically that will be only one or two currencies.

You declare them by append the special value "MONEY" for the "export" field, specifying which currency this commodity represents, like this:

```
1900-01-01 commodity VMMXX
  export: "MUTF:VMMXX (MONEY:USD)"

1900-01-01 commodity IGI806
  export: "MUTF_CA:IGI806 (MONEY:CAD)"
```

Ignoring Commodities

Finally, some commodities held in a ledger should be ignored. This is the case for the imaginary commodities used in mirror accounting, for example, to track unvested shares of an employment stock plan, or commodities used to track amounts contributed to a retirement account, like this:

```
1996-01-01 commodity RSPCAD
  name: "Canada Registered Savings Plan Contributions"
```

You tell the export code to ignore a commodity specifying the special value "IGNORE" for the "export" field, like this:

```
1996-01-01 commodity RSPCAD
```

```
name: "Canada Registered Savings Plan Contributions"
export: "IGNORE"
```

All holdings in units of RSPCAD will thus not be exported.

The question of whether some commodities should be exported or not sometimes presents interesting choices. Here is an example: I track my accumulated vacation hours in an asset account. The units are "VACHR". I associate with this commodity a price that is roughly equivalent to my net hourly salary. This gives me a rough idea how much vacation time money is accumulated on the books, e.g. if I quit my job, how much I'd get paid. Do I want to them included in my total net worth? Should the value from those hours be reflected in the value of my exported portfolio? I think that largely depends on whether I plan to use up those vacations before I leave this job or not, whether I want to have this accumulated value show up on my balance sheet.

Comparing with Net Worth

The end result is that the sum total of all your exported positions plus the cash position should approximate the value of all your assets, and the total value calculated by the Google Finance website should be very close to the one reported by this report:

```
bean-report file.beancount networth
```

As a point of comparison, the value of my own portfolio is usually close to within a few hundred US dollars.

Details of the OFX Export

Import Failures

Exporting a portfolio with symbols that Google Finance does not recognize **fatally** trips up Google's import feature. Google Finance then proceeds to fail to recognize your **entire** file. I recommend that you use explicit exchange:symbol names on all commodities that get exported in order to avoid this problem, as is described further in this document.

Google Finance can also be a little bit finicky about the format of the particular OFX file you give it to import. The export_portfolio command attempts to avoid OFX features that would break it but it's fragile, and it's possible that the particulars of your portfolio's contents triggers output that fails to import. If this is the case, at step (4) above, instead of a list of stock symbols you would see a long list of positions that look like XML tags (this is how failure manifests itself). If that is the case, send email to the mailing-list (best if you can isolate the positions that trigger breakage and have the capability to diff files and do some troubleshooting).

Mutual Funds vs. Stocks

The OFX format distinguishes between stocks and mutual funds. In practice, the Google Finance importer does not appear to distinguish between these two (at least it appears to behave the same way), so this is likely an irrelevant implementation detail. Nevertheless, the export code is able to honor the OFX convention of distinguishing between "BUYMF" vs. "BUYSTOCK" XML elements.

To this effect, the export code attempts to classify which commodities represent mutual funds by inspecting whether the ticker associated with the commodity begins with the letters "MUTF" and is followed by a colon. For example, "MUTF:RGAGX" and "MUTF_CA:RBF1005" will both be detected as mutual funds, for example.

Debugging the Export

In order to debug how each of your holdings gets exported, use the --debug flag, which will print a detailed account of how each holding is handled by the export script to **stderr**:

```
bean-report file.beancount export_portfolio --debug 2>&1 >/dev/null | more
```

The script should print the list of exported positions and their corresponding holdings, then the list of converted positions and their corresponding holdings (usually many cash positions are aggregated together) and finally, the list of ignored holdings. This should be enough to explain the entire contents of the exported portfolio.

Purchase Dates

Beancount does not currently have all the lot purchase dates, so the purchase dates are exported as if purchased the day before the export.

Eventually, when the purchase date is available in Beancount (pending the inventory booking changes) the actual lot purchase date will probably be used in the export format. However, it's not yet clear that using the correct date is the right thing to do, because Google Finance might insist on inserting cash for dividends since the reported purchase date… but Beancount already takes care of inserting a special lot for cash that should already include this. We shall see when we get there.

Disable Dividends

Under the "Edit Portfolio" option there is a checkbox that appears to disable the calculation of dividends offered. It would be nice to find a way to automatically disable this checkbox upon import.

Automate Upload

It would be nice to automate the replacement of the portfolio with a Python script. Unfortunately, the Google Finance API has been deprecated. Maybe someone can write a screen-scraping routine to do this.

Summary

Each holding's export can be controlled by how its commodity is treated, in one of the following ways:

1. **Exported** to a portfolio position. This is the default, but you should specify the ticker symbol using the "ticker" or "export" metadata fields, in "*ExchangeCode:Symbol*" format.
2. **Converted** to cash and exported to a money market cash-equivalent position, by setting the value of the "export" metadata field to the special value "CASH".
3. **Ignored** by specifying the "export" metadata field to the special value "IGNORE".
4. Provided as **Money Instrument**, to be used for cash-equivalent value of each holding intended to be converted to cash and included in the portfolio. These are identified by a special value "(MONEY:<currency>)" in the "export" metadata field.

Beancount Example & Tutorial

Martin Blais, October 2014

http://furius.ca/beancount/doc/example

Other Reports

Other Formats

Viewing Reports through the Web Interface

The Future of Beancount Reports

Example File Generator

Beancount has a command to generate a few years of a realistic user's historical entries:

```
bean-example > example.beancount
```

The script checks if the output fails to process cleanly in Beancount (if this occurs, try rerunning or contact the author at blais@furius.ca). Note that there is an option to fix the random generator's seed to be used to generate the entries.

You can generate multiple years of ledger data for this user; see available options with

```
bean-example --help
```

The purpose of the script is:

- To provide a realistic corpus of anonymous data for users to experiment with, to kick the tires of Beancount, generate reports, etc.
- To compare reports with those generated by Ledger.
- As an example file to be used in this tutorial.
- As input for stress testing Beancount.

Converting to Ledger Input

It should be possible to convert example files generated from the script to Ledger syntax, like this:

```
bean-report example.beancount ledger > example.lgr
```

If you encounter any problems with the conversion, please let me know. While I have an automated test to check that the conversion succeeds, I'm focusing my efforts on Beancount itself and I'm not using Ledger much other than for the occasional comparison or for illustrating something on the mailing-list.

Profile of Example User

Here's a high-level profile of this user. In the text that follows, I'll use the masculine "he" for simplicity's sake.

- He lives somewhere in the USA and uses a single operating currency: "USD". He rents an apartment.

- **Income**: He is a software engineer (sorry I could not resist), receives a modest salary income from a software company.
  - He is paid bi-weekly.
  - He maximizes contributions to his company's 401k plan. His company offers a match to his contributions.
  - His salary is paid as a direct deposit to his checking account.
  - He benefits and pays premiums for a health insurance plan provided by his employer.
  - He tracks his vacation hours explicitly.
- **Expenses**:
  - **Checking account**: He pays his apartment's rent by check. He also pays his electricity, internet ISP and mobile phone bills directly from his checking account. His bank also takes a monthly bank fee.
  - **Credit cards**: He also has other regular expenses, such as groceries and restaurant outings with friends, which he pays from his credit card.
- **Assets**:
  - **Banking**: He has an account at a major US bank.
  - **Retirement account**: His 401k account is held at Fidelity or Vanguard and is allocated in its entirety to two funds, a mix of stocks and bonds. There are never any sales in this account. This account is held at average cost.
  - **Other investments**: Extra money that he is able to save is transferred from his checking account towards a taxable investment account in which he purchases a mix of stocks and ETFs, and occasionally makes sales and realizes profits.
- **Liabilities**:
  - **Credit cards**: He has one or two credit cards issued by another bank.
- **Events/Tags**: He takes a short vacation or two per year, where he travels somewhere fun.

Future Additions

Note that as I beef up the example generator script, this profile will increase in complexity. If there is a particular situation from the cookbook or otherwise that you'd like to see in the example file, please let me know (a patch is the best way to let me know).

- Generate fake PDF files for documents (from an option) along with suitable document directives for them, most of them implicit but some explicit.
- Generate more expenses:
  - Some amount of realistic cash expenses.
  - Add a vehicle: car repairs, gas, etc. Remove tram example, it is too urban-specific.
  - Add donations for charity.
  - Add a second credit card and vary sources of expenses.
- Add some real uses for links, for the 401k match, for instance, the employer contributions should be linked with the employee contributions. Or other ones.
- Generate a loan and payments for a student loan. Having a large liability is a common case, I'd like to represent that, even if it's not a mortgage.
- Convert retirement accounts to book using the average cost method, and add corresponding fees which would otherwise be impossible to track with explicit lots.
- Add tracking of health care expenses as per the cookbook.
- Add foreign accounts in another currency and transactions and transfers in that currency.

Tutorial

This section provides basic examples of generating reports with Beancount, on the typical example file that would be output by it. The list of reports here is not meant to be exhaustive.

You may follow the tutorial and generate your own example file, or look at the files from beancount/examples/tutorial if you prefer.

Generate an Example File

Begin by generating the example file:

```
bean-example > example.beancount
```

Open example.beancount and examine it.

Next, before we begin generating reports, verify that the file loads without any errors:

```
bean-check example.beancount
```

It should return quietly, without outputting anything (bean-check only writes errors when there are some, otherwise on success it writes nothing).

Generating Reports

Let's generate a report of the final balances of all accounts [output]:

```
bean-report example.beancount balances
```

As you can see, the bean-report script has subcommands for the various reports it generates. To list the available reports, use --help-reports [output]:

```
bean-report --help-reports
```

To list the options available for a particular report, use --help on it [output]:

```
bean-report example.beancount balances --help
```

The report-specific options vary for each report type.

You can also list the global options for the script [output]:

```
bean-report --help
```

The global options allow you to specify the desired output format for the reports. To see which formats are available for which report, use --help-formats [output]:

```
bean-report --help-formats
```

Generating Balances

Good, so we know how to generate a report of balances for all accounts. This is a pretty detailed list of accounts though. Let's just restrict the output to the accounts that we're interested in [output]:

```
bean-report example.beancount balances -e ETrade
```

Here you can view the number of units held in each of the investment subaccounts. To render the cost, use the --cost option [output]:

```
bean-report example.beancount balances -e ETrade --cost
```

Formatting Tools

Sometimes it's nice to render a hierarchical list of accounts as a tree. You can use the "treeify" tool provided by Beancount to do this:

```
bean-report example.beancount balances | treeify
```

This tool will work on any column of data that looks like a column of account names (you can also configure it work with filenames as well, or other patterns).

Generating a Balance Sheet and Income Statement

Let us generate a balance sheet:

```
bean-report example.beancount balsheet
```

Unfortunately, the only output format supported for it at this point is HTML. Also, filtering for balance sheet from the command-line is not supported. Generate this to a file and open a browser to it [output]:

```
bean-report example.beancount balsheet > /tmp/balsheet.html
```

You can to the same for income statements:

```
bean-report example.beancount income > /tmp/income.html
```

Journals

You can also generate journals (in Ledger parlance, these are "registers"). Let's look at a checking account postings, for instance [output]:

```
bean-report example.beancount journal -a Assets:US:BofA:Checking
```

To render a column of running balances, add the --balance option [output]:

```
bean-report example.beancount journal -a Assets:US:BofA:Checking --balance
```

The inventories are rendered in the number of units they're holding [output]:

```
bean-report example.beancount journal -a Assets:US:ETrade:GLD
```

If you want to render the values at cost, use the --cost option [output]:

```
bean-report example.beancount journal -a Assets:US:ETrade:GLD --cost
```

To render journals, you will typically restrict the set of postings that you want to view. If you did not, the postings would render no change, because all the legs of the transactions would be applied to the running balance and the total would be zero. But try it, nonetheless [output]:

```
bean-report example.beancount journal --balance
```

Holdings

There are a variety of ways to obtain aggregations for the total list of holdings. List the detailed holdings [output]:

```
bean-report example.beancount holdings
```

To aggregate the holdings by account, do this [output]:

```
bean-report example.beancount holdings --by=account
```

Because it's common practice to create and use a dedicated Beancount subaccount name for each commodity held in a real-world account, we can refine

this further to aggregate to the parent ("root") accounts of those subaccounts [output]:

```
bean-report example.beancount holdings --by=root-account
```

We can aggregate by commodity [output]:

```
bean-report example.beancount holdings --by=commodity
```

Or by the currency in which the commodities are quoted, which gives us a glimpse into our currency exposure [output]:

```
bean-report example.beancount holdings --by=currency
```

Finally, we can aggregate all the holdings to obtain net worth [output]:

```
bean-report example.beancount networth
```

There are a few more options for converting amounts to a common currency. See help for details.

Other Reports

There are many other miscellaneous reports available. Try a few of those.

Listing all accounts [output]:

```
bean-report example.beancount accounts
```

Listing events [output]:

```
bean-report example.beancount events
```

Number of directives by type [output]:

```
bean-report example.beancount stats-directives
```

Number of postings by type [output]:

```
bean-report example.beancount stats-postings
```

Other Formats

Reports may support various output formats. You can change the output format with the --format (-f) global option [output]:

```
bean-report -f csv example.beancount holdings
```

Note that "beancount" and "ledger" are valid output formats: they refer to the Beancount and Ledger input syntaxes.

Viewing Reports through the Web Interface

The original way to access reports in Beancount is via its web interface, that serves to a local web server on your machine. Serve the example file like this:

```
bean-web example.beancount
```

Then navigate with a web browser to http://localhost:8080. From there, you can click on any number of filtered views and access some of the reports previously demonstrated. For example, click on a year view; that will provide balance sheets and income statements and various other reports for this subset of transactions.

The bean-web tool has many options for restricting what is being served. (Note that by default the server listens only for connections from your computer; if you want to host this on a web server and accept connections from anywhere, use --public).

Note: There exists a separate project which provides a better web interface than the one which comes with Beancount: Fava. You might want to check it out.

The Future of Beancount Reports

I find my command-line reports above to be rather unsatisfying, from the point-of-view of customizability. They bother me a lot. This is largely because I've been happy and satisfied with the capabilities of the web interface to Beancount. As I'm beginning to use the command-line interface more and more, I'm finding myself desiring for more explicit and well-defined way to apply all the filters I have available from the web interface, and more. For this reason, I'm currently implementing a query language that looks similar to SQL. This syntax will allow me to remove all the custom reports above and to replace them by a single consistent query syntax. This work is underway.

All the holdings reports and even their internal objects will be replaced by the SQL-like query syntax as well. Holdings need not be treated separately: they are simply the list of positions available at a particular point in time, and along with suitable accessors from the query syntax and support for aggregations over their many columns, we should be able to generate all of those reports equivalently and simplify the code.

— *Martin Blais, October 2014*

Beancount History and Credits

Martin Blais, July 2014

http://furius.ca/beancount/doc/history

A history of the development of Beancount and credits for contributors..

History of Beancount

John Wiegley's Ledger was the inspiration for the first version of Beancount. His system is where much of the original ideas for this system came from. When I

first learned about double-entry bookkeeping and realized that it could be the perfect method to solve many of the tracking problems I was having in counting various installments for my company, and after a quick disappointment in the solutions that were available at the time (including GnuCash, which I could break very easily), I was quickly led to the Ledger website. There, John laid out his vision for a text-based system, in particular, the idea of doing away with credits and debits and just the signs, and the basics of a convenient input syntax which allows you to omit the amount of one of the postings. I got really excited and had various enthusiastic discussions with him about Ledger and how I wanted to use it. There was some cross-pollination of ideas and John was very receptive to proposals for adding new features.

I was so intensely curious about bookkeeping that I began writing a Python interface to Ledger. But in the end I found myself rewriting the entire thing in Python–not for dislike of Ledger but rather because it was simple enough that I could do most of it in little time, and immediately add some features I thought would be useful. One reason for doing so was that instead of parsing the input file every time and generating one report to the console, I would parse it once and then serve the various reports from the in-memory database of transactions, requested via a web page. Therefore, I did not need processing speed, so having to use C++ for performance reasons was not necessary anymore, and I chose to just stick with a dynamic language, which allowed me to add many features quickly. This became Beancount version 1, which stood on its own and evolved its own set of experimental features.

My dream was to be able to quickly and easily manipulate these transaction objects to get various views and breakdowns of the data. I don't think the first implementation pushed the limits far enough, however; the code was sub-standard, to be honest—I wrote it really quickly—and making modifications to the system was awkward. In particular, the way I originally implemented the tracking of capital gains was inelegant and involved some manual counting. I was unhappy with this, but it worked. It was also using ugly ad-hoc parser code in order to remain reasonably compatible with Ledger syntax—I thought it would be interesting to be able to share some common input syntax and use either system for validation and maybe even to convert between them–and that made me wary of making modifications to the syntax to evolve new features, so it stabilized for a few years and I lost interest in adding new features.

But it was correct and it worked, mostly, so I used the system continuously from 2008 to 2012 to manage my own personal finances, my company's finances, and joint property with my wife, with detailed transactions; this was great. I learned a lot about how to keep books during that time (the cookbook document is meant to help you do the same). In the summer of 2013, I had an epiphany and realized a correct and generalizable way to implement capital gains, how to merge the tracking of positions held at a cost and regular positions, and a set of simple rules for carrying out operations on them sensibly (the design of how inventories work). I also saw a better way to factor out the internal

data structures, and decided to break from the constraint of compatibility with Ledger and redesign the input syntax in order to parse the input language using a lex/yacc generator, which would allow me to easily evolve the input syntax without having to deal with parsing issues, and to create ports to other languages more easily. In the process, a much simpler and more consistent syntax emerged, and in a fit of sweat and a few intense weekends I re-implemented the entire thing from scratch, without even looking at the previous version, clean-room. Beancount version 2 was born, much better than the last, modular, and easy to extend with plugins.

The result is what I think is an elegant design involving a small set of objects, a design that could easily be a basis for reimplementation in other computer languages. This is described in the accompanying design doc, for those who would have an interest in having a go at it (this would be welcome and I'm expecting this will happen). While Ledger remains an interesting project bubbling with ideas for expressing the problem of bookkeeping, the second version of Beancount proposes a simpler design that leaves out features that are not strictly necessary and aims at maximum usability through a simple web interface and a very small set of command-line options. Since I had more than 5 years worth of real-world usage experience with the first version, I set a goal for myself to remove all the features that I thought weren't actually useful and introduced unnecessary complexity (like virtual transactions, allowing accounts not in the five types, etc.), and to simplify the system as much as possible without compromising on its usability. The resulting language and software were much simpler to use and understand, the resulting data structures are much simpler to use, the processing more "functional" in nature, and the internals of Beancount are very modular. I converted all my 6 years worth of input data–thanks to some very simple Python scripts to manipulate the file–and began using the new version exclusively. It is now in a fully functional state.

Ledger's syntax implements many features that trigger a lot of implicitly-defined behaviour; I find this confusing and some the reasons for this are documented in the many improvement proposals. I don't like command-line options. In contrast, Beancount's design provides a less expressive, lower-level syntax but one that closely matches the generated in-memory data structure, and that is hopefully more explicit in that way. I think both projects have strengths and weaknesses. Despite its popularity, the latest version of Ledger remains with a number of shortcomings in my view. In particular, reductions in positions are not booked at the moment they occur, but rather they appear to simply accumulate and get matched only at display time, using different methods depending on the command-line options. Therefore, it is possible in Ledger to hold positive and negative lots of the same commodity in an account simultaneously. I believe that this can lead to confusing and even incorrect accounting for trading lots. I think this is still being figured out by the Ledger look-alikes community and that they will eventually converge to the same solution I have, or perhaps even figure out a better solution.

In the redesign, I separated out configuration directives that I had used for importing and over many iterations eventually figured out an elegant way to mostly automate imports and automatically detect the various input files and convert them into my input syntax. The result is the LedgerHub design doc. LedgerHub is currently implemented and I'm using it exclusively, but has insufficient testing for me to stamp a public release [July 2014]. You are welcome to try it out for yourself.

In June and July 2014, I decided to dump seven years' worth of thinking about command-line accounting in a set of Google Docs and this now forms the basis for the current documentation of Beancount. I hope to be evolving it gradually from here on.

Chronology

- Ledger was begun in August 2003
  http://web.archive.org/web/*/http://www.newartisans.com/ledger/
- Beancount was begun in 2008
  http://web.archive.org/web/*/furius.ca/beancount
- HLedger was also begun in 2008
  https://github.com/simonmichael/hledger/graphs/contributors?from=2007-01-21&to=2014-09-08&type=c

Credits

So far I've been contributing all the code to Beancount. Some users have made significant contributions in other ways:

- Daniel Clemente has been reporting all the issues he came across while using Beancount to manage his company and personal finances. His relentless perseverance and attention to detail has helped me put a focus on fixing the rough corners of Beancount that I knew to avoid myself.
- After many years of prodding, my old friend Filippo Tampieri has finally decided to convert his trading history in Beancount format. He has contributed a number of sophisticated reviews of my documentation and is working on adding various methods for evaluating returns on assets.

A Comparison of Beancount and Ledger

Martin Blais, September 2014

http://furius.ca/beancount/doc/comparison

The question of how Beancount differs from Ledger & HLedger has come up a few times on mailing-lists and in private emails. This document highlights

key differences between these systems, as they differ sharply in their design and implementations.

Keep in mind that this document is written from the perspective of Beancount and as its author, reflects my own biased views for what the design of a CLI accounting system should be. My purpose here is not to shoot down other systems, but rather to highlight material differences to help newcomers understand how these systems vary in their operation and capabilities, and perhaps to stimulate a fruitful discussion about design choices with the other developers.

Differences

Inventory Booking

Currency Conversions

Isolation of Inputs

Language Syntax

Order Independence

Account Types

Transactions Must Balance

Numbers and Precision of Operations

Filtering at the Transactional Level

Extension Mechanisms

Automated Transactions via Plugins

No Support for Time or Effective Dates

Documents

Simpler and More Strict

Web Interface

Missing Features

Console Output

Filtering Language

No Meta-data

No Arithmetic Expressions

Limited Support for Unicode

No Forecasting or Periodic Transactions

Philosophical Differences

(See this thread.)

First, Ledger is optimistic. It assumes it's easy to input correct data by a user. My experience with data entry of the kind we're doing is that it's impossible to do this right without many automated checks. Sign errors on unasserted accounts are very common, for instance. In contrast, Beancount is highly pessimistic. It assumes the user is unreliable. It imposes a number of constraints on the input. For instance, if you added a share of AAPL at $100 to an empty account it won't let you remove a share of AAPL at $101 from it; you just don't have one. It doesn't assume the user is able or should be relied upon to input transactions in the correct order (dated assertions instead of file-order assertions). It optionally checks that proceeds match sale price (sellgains plugin). And it allows you to place extra constraints on your chart of accounts, e.g. a plugin that refuses postings to non-leaf accounts, or that refuses more than one commodity per account, or that requires you declare all accounts with Open directives; choose your level of pedanticity a-la-carte. It adds more automated cross-checks than the double-entry method provides. After all, cross-checking is why we choose to use the DE method in the first place, why not go hardcore on checking for correctness? Beancount should appeal to anyone who does not trust themselves too much. And because of this, it does not provide support for unbalanced/virtual postings; it's not a shortcoming, it's on purpose.

Secondly, there's a design ethos difference. As is evidenced in the user manual, Ledger provides a myriad of options. This surely will be appealing to many, but to me it seems it has grown into a very complicated monolithic tool. How these options interact and some of the semantic consequences of many of these options are confusing and very subtle. Beancount offers a minimalistic approach: while there are some small number of options, it tries really hard not to have them. And those options that do affect the semantics of transactions always appear in the input file (nothing on the command-line) and are distinct from the options of particular tools. Loading a file always results in the same stream of transactions, regardless of the reporting tool that will consume them. The only command-line options present are those which affect the particular behavior of the reporting tool invoked; those never change the semantics of the stream itself.

Thirdly, Beancount embraces stream processing to a deeper extent. Its loader creates a single ordered list of directives, and all the directives share some common attributes (a name, a date, metadata). This is all the data. Directives that are considered "grammar" in Ledger are defined as ordinary directive objects, e.g. "Open" is nothing special in Beancount and does nothing by itself. It's simply used in some routines that apply constraints (an account appears, has an Open directive been witnessed prior?) or that might want to hang per-account metadata to them. Prices are also specified as directives and are embedded in the stream, and can be generated in this way. All internal operations are defined as processing and outputting a stream of directives. This makes it pos-

sible to allow a user to insert their own code inside the processing pipeline to carry out arbitrary transformations on this stream of directives—anything is possible, unlimited by the particular semantics of an expression language. It's a mechanism that allows users to build new features by writing a short add-on in Python, which gets run at the core of Beancount, not an API to access its data at the edges. If anything, Beancount's own internal processing will evolve towards doing less and less and moving all the work to these plugins, perhaps even to the extent of allowing plugins to declare the directive types (with the exception of Transaction objects). It is evolving into a shallow driver that just puts together a processing pipeline to produce a stream of directives, with a handy library and functional operations.

Specific Differences

Inventory Booking & Cost Basis Treatment

Beancount applies strict rules regarding reductions in positions from inventories tracking the contents of its accounts. This means that you can only take out of an account something that you placed in it previously (in time), or an error will be generated. This is enforced for units "held at cost" (e.g., stock shares), to ensure that

- no cost basis can ever leak from accounts,
- we can detect errors in data entry for trades (which are all too common), and
- we are able to correctly calculate capital gains.

In contrast, Ledger does not implement inventory booking checks over time: all lots are simply accumulated regardless of the previous contents of an inventory (there is no distinction between lot addition vs. reduction). In Beancount, reductions to the contents of an inventory are required to match particular lots with a specified cost basis.

In Ledger, the output is slightly misleading about this: in order to simplify the reporting output the user may specify one of a few types of lot merging algorithms. By default, the sum of units of all lots is printed, but by using these options you can tell the reporting generation to consider the cost basis (what it calls "prices") and/or the dates the lots were created at in which case it will report the set of lots with distinct cost bases and/or dates individually. You can select which type of merging occurs via command-line options, e.g. --lot-dates. Most importantly, this means that it is legal in Ledger to remove from an account a lot that was never added to it. This results in a mix of long and short lots, which do not accurately represent the actual changes that occur in accounts. However, it trivially allows for average cost basis reporting.

I believe that this is not only confusing, but also an incorrect treatment of account inventories and have argued it can lead to leakage and incorrect cal-

culations. More details and an example are available here. Furthermore, until recently Ledger was not using cost basis to balance postings in a way that correctly computes capital gains. For this reason I suspect Ledger users have probably not used it to compute and compare their gains against the values reported by their broker.

In order for Beancount to detect such errors meaningfully and implement a strict matching discipline when reducing lots, it turns out that the only constraint it needs to apply is that a particular account not be allowed to simultaneously hold long and short positions of the same commodity. For example, all it has to do is enforce that you could not hold a lot of 1 GOOG units and a lot of -1 GOOG units simultaneously in the same inventory (regardless of their acquisition cost). Any reduction of a particular set of units is detected as a "lot reduction" and a search is found for a lot that matches the specification of the reducing posting, normally, a search for a lot held at the same cost as the posting specifies. Enforcing this constraint is not of much concern in practice, as there are no cases where you would want both long and short positions in the same account, but if you ever needed this, you could simply resort to using separate accounts to hold your long and short positions (it is already recommended that you use a sub-account to track your positions in any one commodity anyway, this would be quite natural).

Finally, Beancount is implementing a proposal that significantly extends the scope of its inventory booking: the syntax will allow a loose specification for lot reductions that makes it easy for users to write postings where there is no ambiguity, as well as supporting booking lots at their average cost as is common in Canada and in all tax-deferred accounts across the world. It will also provide a new syntax for specifying cost per unit and total cost at the same time, a feature which will make it possible to correctly track capital gains without commissions.

Currency Conversions

Beancount makes an important semantic distinction between simple currency conversions and conversions of commodities to be held at cost, i.e., for which we want to track the cost basis. For example, converting 20,000 USD to 22,000 CAD is a currency conversion (e.g., between banks), and after inserting the resulting CAD units in the destination account, they are not considered "CAD at a cost basis of 1.1 USD each," they are simply left as CAD units in the account, with no record of the rate which was used to convert them. This models accurately how the real world operates. On the other hand, converting 5,000 USD into 10 units of GOOG shares with a cost basis of 500 USD each is treated as a distinct operation from a currency conversion: the resulting GOOG units have a particular cost basis attached to them, a memory of the price per unit is kept on the inventory lot (as well as the date) and strict rules are applied to the reduction of such lots (what I call "booking") as mentioned previously. This is used to calculate and report capital gains, and most importantly, it detects a *lot* of errors in data entry.

In contrast, Ledger does not distinguish between these two types of conversions. Ledger treats currency conversions the same way as for the conversion of commodities held at cost. The reason that this is not causing as many headaches as one might intuit, is because there is no inventory booking - all lots at whichever conversion rate they occur accumulate in accounts, with positive or negative values - and for simple commodities (e.g. currencies, such as dollars), netting the total amount of units without considering the cost of each unit provides the correct answer. Inspecting the full list of an account's inventory lots is provided as an option (See Ledger's "--lot-dates") and is not the default way to render account balances. I suspect few users make use of the feature: if you did render the list of lots for real-world accounts in which many currency conversions occurred in the past, you would observe a large number of irrelevant lots. I think the cost basis of currency conversions would be best elided instead.

HLedger does not parse cost basis syntax and as such does not recognize it.

Isolation of Inputs

Beancount reads its entire input *only* from the text file you provide for it. This isolation is by design. There is no linkage to external data formats nor online services, such as fetchers for historical price values. Fetching and converting external data is disparate enough that I feel it should be the province of separate projects. These problem domains also segment very well and quite naturally: Beancount provides an isolated core which allows you to ingest all the transactional data and derive reports of various aggregations from it, and its syntax is the hinge that connects it to external transaction repositories or price databases. It isolates itself from the ugly details of external sources of data in this way.

There are too many external formats for downloadable files that contains transactional information to be able to cover all of them. The data files you can obtain from most institutions are provided in various formats: OFX, Quicken, XLS or CSV, and looking at their contents makes it glaringly obvious that the programmers who built the codes that outputs them did not pay much attention to detail or the standard definition of their format; it's quite an ugly affair. Those files are almost always very messy, and the details of that messiness varies over time as well as these files evolve.

Fetching historical or current price information is a similarly annoying task. While Yahoo and Google Finance are able to provide some basic level of price data for common stocks on US exchanges, when you need to fetch information for instruments traded on foreign exchanges, or instruments typically not traded on exchanges, such as mutual funds, either the data is not available, or if it is, you need to figure out what ticker symbol they decided to map it to, there are few standards for this. You must manually sign the ticker. Finally, it is entirely possible that you want to manage instruments for which there is no existing external price source, so it is necessary that your bookkeeping software provide a mechanism whereby you can manually input price values (both Beancount and

Ledger provide a way). The same declaration mechanism is used for caching historical price data, so that Beancount need not require the network at all.

Most users will want to write their own scripts for import, but some libraries exist: the beancount.ingest library (within Beancount) provides a framework to automate the identification, extraction of transactions from and filing of downloadable files for various institutions. See its design doc for details.

In comparison, Ledger and HLedger support rudimentary conversions of transactions from CSV files as well as automated fetching of current prices that uses an external script you are meant to provide (getquote). CSV import is far insufficient for real world usage if you are to track all your accounts, so this needs to be extended. Hooking into an external script is the right thing to do, but Beancount favors taking a strong stance about this issue and instead not to provide code that would trigger any kind of network access nor support any external format as input. In Beancount you are meant to integrate price updates on your own, perhaps with your own script, and maybe bring in the data via an include file. (But if you don't like this, you could also write your own plugin module that could fetch live prices.)

Language Syntax

Beancount's syntax is somewhat simpler and quite a bit more restrictive than Ledger's. For its 2.0 version, the Beancount syntax was redesigned to be easily specifiable to a parser generator by a grammar. The tokens were simplified in order to make tokenization unambiguous. For example,

- Currencies must be entirely in capital letters (allowing numbers and some special characters, like "_" or "-"). Currency symbols (such as $ or €) are not supported. (On the other hand, currencies with numbers require quoting in Ledger.)
- Account names do not admit spaces (though you can use dashes), and must have at least two components, separated by colons.
- Description strings must be quoted, like this: "AMEX PMNT". No freestyle text as strings is supported anymore.
- Dates are only parsed from ISO8601 format, that is, "YYYY-MM-DD".
- Tags must begin with "#", and links with "^".
- Apart from the tag stack, all context information has been removed. There is no account alias, for example, nor is there a notion of "apply" as in Ledger (see "apply root" and "apply tag"). It requires a bit more verbose input—full account names—and so assumes that you have account name completion setup in your editor.

As a side effect, these changes make the input syntax look a bit more like a programming language. These restrictions may annoy some users, but overall

they make the task of parsing the contents of a ledger simpler and the resulting simplicity will pave the way for people to more easily write parsers in other languages. (Some subtleties remain around parsing indentation, which is meaningful in the syntax, but should be easily addressable in all contexts by building a custom lexer.)

Due to its looser and more user-friendly syntax, Ledger uses a custom parser. If you need to parse its contents from another language, the best approach is probably to create bindings into its source code or to use it to export the contents of a ledger to XML and then parse that (which works well). I suspect the parsing method might be reviewed in the next version of Ledger, because using a parser generator is liberating for experimentation.

Order Independence

Beancount offers a guarantee that the ordering of directives in an input file is irrelevant to the outcome of its computations. You should be able to organize your input file and reorder any declaration as is most convenient for you without having to worry about how the software will make its calculations. Not even directives that declare accounts ("Open") are required to appear before these accounts get used in the file. All directives are parsed and then basically stably sorted before any calculation or validation occurs. This also makes it trivial to implement inclusions of multiple files (you can just concatenate the files if you want).

In contrast, Ledger processes the amounts on its postings as it parses the input file. In terms of implementation, this has the advantage that only a single pass is needed to check all the assertions and balances, whereas in Beancount, numerous passes are made on the entire list of directives (this has not been much of a problem in Beancount, however, because even a realistically large number of transactions is rather modest for our computers; most of Beancount's processing time is due to parsing and numerical calculations).

An unfortunate side-effect of Ledger's method of calculation is that a user must be careful with the order in which the transactions appear in the file. This can be treacherous and difficult to understand when editing a very large input file. This difference is particularly visible in balance assertions. Ledger's balance assertions are attached to the postings of transaction directives and calculated in file order (I call these "file assertions"). Beancount's balance assertions are separate directives that are applied at the beginning of the date for which they are declared, regardless of their position in the file (call these "dated assertions"). I believe that dated assertions are more useful and less error-prone, as they do not depend on the ordering of declarations. On the other hand, Ledger-style file assertions naturally support balance checks on intra-day balances without having to specify the time on transactions, which is impossible with dated assertions.

For this reason, a proposal has been written up to consider implementing file assertions in Beancount (in addition to its dated assertions). This will probably

be carried out as a plugin. Ledger does not support dated assertions.

Account Types

Beancount accounts must have a particular type from one of five categories: Assets, Liabilities, Income, Expenses and Equity. Ledger accounts are not constrained in this way, you can define any root account you desire and there is no requirement to identify an account as belonging to one of these categories. This reflects the more liberal approach of Ledger: its design aims to be a more general "calculator" for anything you want. No account types are enforced or used by the system.

In my experience, I haven't seen any case where I could not classify one of my accounts in one of those categories. For the more exotic commodities, e.g., "US dollars allowable to contribute to an IRA", it might require a bit of imagination to understand which account fits which category, but there is a logic to it: if the account has an *absolute* value that we care about, then it is an Assets or Liabilities account; if we care only about the *transitional* values, or the value accumulated during a time period, then it should be an Income or Expenses account. If the sign is positive, then it should be an Assets or Expenses account; conversely, if the sign is negative, it should be a Liabilities or Income account. Equity accounts are almost never used explicitly, and are defined and used by Beancount itself to transfer opening balances, retained earnings and net income to the balance sheet report for a particular reporting period (any period you choose). This principle makes it easy to resolve which type an account should have. I have data from 2008 to 2014 and have been able to represent *everything* I ever wanted using these five categories. Also, I don't think asking the user to categorize their accounts in this way is limiting in any way; it just requires a bit of foresight.

The reason for requiring these account types is that it allows us to carry out logic based on their types. We can isolate the Income and Expenses accounts and derive an income statement and a single net income value. We can then transfer retained earnings (income from before the period under consideration) and net income (income during the period) to Equity accounts and draw up a balance sheet. We can generate lists of holdings which automatically exclude income and expenses, to compute net worth, value per account, etc. In addition, we can use account types to identify external flows to a group of accounts and compute the correct return on investments for these accounts in a way that can be compared with a target allocation's market return (note: not integrated yet, but prototyped and spec'ed out, it works). The bottom line is that having account types is a useful attribute, so we enforce that you should choose a type for each account.

The absence of account types is probably also why Ledger does not provide a balance sheet or income statement reports, only a trial balance. The advantage is an apparently looser and more permissive naming structure. But also note

that requiring types does not itself cause any difference in calculations between the two systems, you can still accumulate any kind of "beans" you may want in these accounts, it is no less general. The type is only extra information that Beancount's reporting makes use of.

Transactions Must Balance

Beancount transactions are required to balance, period. I make no compromise in this, there is no way out. The benefit of this is that the sum of the balance amounts of the postings of any subset of transactions is always precisely *zero* (and I check for it).

Ledger allows the user two special kinds of postings:

- Virtual postings: These are postings input with parentheses around them, and they are *not* considered in the transaction balance's sum, you can put any value in them without causing an error.
- Balanced virtual postings: These postings are input with square brackets around them. These are less permissive: the set of postings in square brackets is enforced to balance within itself.

The second case can be shown to be equivalent to two transactions: a transaction with the set of regular postings, and a separate transaction with only the balanced virtual ones (this causes no problem). The first case is the problematic one: in attempting to solve accounting problems, beginning users routinely revert to them as a cop-out instead of modeling their problem with the double-entry method. It is apparent that most adopters of CLI accounting systems are computer scientists and not accounting professionals, and as we are all learning how to create our charts of accounts and fill up our ledgers, we are making mistakes. Oftentimes it is truly not obvious how to solve these problems; it simply requires experience.

But the fact is that in 8 years' worth of usage, there isn't a single case I have come across that truly required having virtual postings. The first version of Beancount used to have support for virtual postings, but I've managed to remove all of them over time. I was always able to come up with a better set of accounts, or to use an imaginary currency that would allow me to track whatever I needed to track. And it has always resulted in a better solution with unexpected and sometimes elegant side-effects.

But these systems have to be easy to use, so how do we address this problem? The mailing-list is a good place to begin and ask questions, where people share information regarding how they have solved similar problems (there aren't that many "accounting problems" per-se in the first place). I am also in the process of documenting all the solutions that I have come up with to solve my own accounting problems in the Beancount Cookbook, basically everything I've learned so far; this is work in progress. I hope for this evolving document to become a helpful reference to guide others in coming up with solutions that fit

the double-entry accounting framework, and to provide ample examples that will act as templates for others to replicate, to fit in their own data.

In the words of Ledger's author:

```
"If people don't want to use them [virtual accounts], that's fine. But Ledger is not an acc
```

I respectfully beg to differ. Therefore Beancount takes a more radical stance and explicitly avoids supporting virtual postings. If you feel strongly that you should need them, you should use Ledger.

Numbers and Precision of Operations

Beancount, Ledger and HLedger all differ in how they represent their numbers internally, and in how they handle the precision of balance checks for a transaction's postings.

First, about how number are represented: Ledger uses rational numbers in an attempt to maintain the full precision of numbers resulting from mathematical operations. This works, but I believe this is perhaps not the most appropriate choice. The great majority of the cases where operations occur involve the conversion from a number of units and a price or a cost to the total value of an account's posted change (e.g., units x cost = total cost). Our task in representing transactional information is the replication of operations that take place mostly in institutions. These operations always involve the rounding of numbers for units and currencies (banks do apply stochastic rounding), and the *correct* numbers to be used from the perspective of these institutions, and from the perspective of the government, are indeed the *rounded* numbers themselves. It is a not a question of mathematical purity, but one of practicality, and our system should do the same that banks do. Therefore, I think that we should always post the rounded numbers to accounts. Using rational numbers is not a limitation in that sense, but we must be careful to store rounded numbers where it matters. I think the approach implemented by Ledger is to keep as much of the original precision as possible.

Beancount chooses a decimal number representation to store the numbers parsed from the input with the same precision they are written as. This method suffers from the same problem as using rational numbers does in that the result of mathematical operations between the decimal numbers will currently be stored with their full precision (albeit in decimal). Admittedly, I have yet to apply explicit quantization where required, which would be the correct thing to do. A scheme has to be devised to infer suitable precisions for automatically quantizing the numbers after operations. The decimal representation provides natural opportunities for rounding after operations, and it is a suitable choice for this, implementations even typically provide a context for the precision to take place. Also note that it will never be required to store numbers to an infinite precision: the institutions never do it themselves.

HLedger, oddly enough, selects "double" fractional binary representation for its

prices. This is an unfortunate choice, a worse one than using a precise representation: fractional decimal numbers input by the user are *never* represented precisely by their corresponding binary form. So all the numbers are incorrect but "close enough" that it works overall, and the only way to display a clean final result is by rounding to a suitable number of digits at the time of rendering a report. One could argue that the large number of digits provided by a 64-bit double representation is unlikely to cause significant errors given the quantity of operations we make… but binary rounding error could potentially accumulate, and the numbers are basically all incorrectly stored internally, rounded to their closest binary relative. Given that our task is accounting, why not just represent them correctly?

Secondly, when checking that the postings of a transaction balance to zero, with all three systems it is necessary to allow for some tolerance on those amounts. This need is clear when you consider that inputting numbers in a text file implies a limited decimal representation. For example, if you're going to multiply a number of units and a cost, say both written down with 2 fractional digits, you might end up with a number that has 4 fractional digits, and then you need to compare that result with a cash amount that would typically be entered with only 2 fractional digits. You need to allow for some looseness somehow.

The systems differ in how they choose that tolerance:

- Ledger attempts to automatically derive the precision to use for its balance checks by using recently parsed context (in file order). The precision to be used is that of the last value parsed for the particular commodity under consideration. This can be problematic: it can lead to unnecessary side-effects between transactions which can be difficult to debug.
- HLedger, on the other hand, uses global precision settings. The whole file is processed first, then the precisions are derived from the most precise numbers seen in the entire input file.
- At the moment, Beancount uses a constant value for the tolerance used in its balance checking algorithm (0.005 of any unit). This is weak and should, at the very least, be commodity-dependent, if not also dependent on the particular account in which the commodity is used. Ultimately, it depends on the numbers of digits used to represent the particular postings. We have a proposal en route to fix this.

I am planning to fix this: Beancount will eventually derive its precision using a method entirely *local* to each transaction, perhaps with a global value for defaults (this is still in the works - *Oct 2014*). Half of the most precise digit will be the tolerance. This will be derived similarly to HLedger's method, but for each transaction separately. This will allow the user to use an arbitrary precision, simply by inserting more digits in the input. Only fractional digits will be used to derive precision. No global effect implied by transactions will be applied. No transaction should ever affect any other transaction's balancing

context. Rounding error will be optionally accumulated to an Equity account if you want to monitor it.

As for automatically quantizing the numbers resulting from operations, I still need to figure out an automatic method for doing so.

Filtering at the Transactional Level

Another area where the systems differ is in that Beancount will not support filtering at the postings level but only at the transaction level. That is, when applying filters to the input data, even filtering that applies predicates to postings, only sets of *complete* transactions will be produced. This is carried out in order to produce sets of postings whose sum balances to exactly zero. We will not ignore only some postings of a transaction. We will nevertheless allow reports to cull out arbitrary subsets of accounts once all balances have been computed.

Ledger filtering works at either the transactional or postings level. I think this is confusing. I don't really understand why it works this way or how it actually works.

(Note that this is not a very relevant point at this moment, because I have yet to implement custom arbitrary filtering; the only filtering available at the moment are "views" you can obtain through the web interface, but I will soon provide a simple logical expression language to apply custom filters to the parsed transactions as in Ledger, as I think it's a powerful feature. Until recently, most reports were only rendered through a web interface and it wasn't as needed as it is now that Beancount implements console reports.)

Extension Mechanisms

Both Beancount and Ledger provide mechanisms for writing scripts against their corpus of data. These mechanisms are all useful but different.

Ledger provides

- A custom expression language which it interprets to produce reports
- A command to export contents to XML
- A command to export contents to LISP
- A library of Python bindings which provides access to its C++ data structures
- *(… others?)*

(Note: due to its dependencies, C++ features being used and build system, I have found it difficult to build Ledger on my vanilla Ubuntu machine or Mac OS computer. Building it with the Python bindings is even more difficult. If

you have the patience, the time and that's not a problem for you, great, but if you can find a pre-packaged version I would recommend using that instead.)

Beancount provides

- A native Python plugin system whereby you may specify lists of Python modules to import and call to filter and transform the parsed directives to implement new features;
- An easy loader function that allows you to access the internal data structures resulting from parsing and processing a Beancount ledger. This is also a native Python library.

So basically, you must write Python to extend Beancount. I'm planning to provide output to XML and SQL as well (due to the simple data structures I'm using these will be both very simple to implement). Moreover, using Beancount's own "printer" module produces text that is guaranteed to parse back into exactly the same data structure (Beancount guarantees round-tripping of its syntax and its data structures).

One advantage is that the plugins system allows you to perform in-stream arbitrary transformations of the list of directives produced, and this is a great way to prototype new functionality and easier syntax. For example, you can allocate a special tag that will trigger some arbitrary transformation on such tagged transactions. And these modules don't have to be integrated with Beancount: they can just live anywhere on your own PYTHONPATH, so you can experiment without having to contribute new features upstream or patch the source code.

(Note: Beancount is implemented in Python 3, so you might have to install a recent version in order to work with it, such as Python-3.4, if you don't have it. At this point, Python 3 is becoming pretty widespread, so I don't see this as a problem, but you might be using an older OS.)

Automated Transactions via Plugins

Ledger provides a special syntax to automatically insert postings on existing transactions, based on some matching criteria. The syntax allows the user to access some of a posting's data, such as amount and account name. The syntax is specialized to also allow the application of transaction and posting "tags."

Beancount allows you to do the same thing and more via its plugin extension mechanism. Plugins that you write are able to completely modify, create or even delete any object and attribute of any object in the parsed flow of transactions, allowing you to carry out as much automation and summarization as you like. This does not require a special syntax - you simply work in Python, with access to all its features - and you can piggyback your automation on top of existing syntax. Some examples are provided under beancount.plugins.*.

There is an argument to be made, however, for the availability of a quick and easy method for specifying the most common case of just adding some postings. I'm not entirely convinced yet, but Beancount may acquire this eventually (you could easily prototype this now in a plugin file if you wanted).

No Support for Time or Effective Dates

Beancount does not represent the intra-day time of transactions, its granularity is a day. Ledger allows you to specify the time of transactions down to seconds precision. I choose to limit its scope in the interest of simplicity, and I also think there are few use cases for supporting intra-day operations.

Note that while Beancount's maximum resolution is one day, when it sorts the directives it will maintain the relative order of all transactions that occurs within one day, so it is possible to represent multiple transactions occurring on the same day in Beancount and still do correct inventory bookings. But I believe that if you were to do day-trading you would need a more specialized system to compute intra-day returns and do technical analysis and intraday P/L calculations. Beancount is not suited for those (a lot of other features would be needed if that was its scope).

Ledger also has support for *effective dates* which are essentially an alternative date for the transaction. Reporting features allow Ledger users to use the main date or the alternative date. I used to have this feature in Beancount and I removed it, mainly because I did not want to introduce reporting options, and to handle two dates, say a transaction date and a settlement date, I wanted to enforce that at any point in time all transactions would balance. I also never made much use of it which indicates it was probably futile. Handling postings to occur at different points in time would have created imbalances, or I would have had to come up with a solution that involved "limbo" or "transfer" accounts. I preferred to just remove the feature: in 8 years of data, I have *always* been able to fudge the dates to make everything balance. This is not a big issue.

Note that handling those split transaction and merging them together will be handled; a proposal is underway.

Documents

Beancount offers support for integrating a directory hierarchy of documents with the contents of a ledger's chart of accounts. You can provide a directory path and Beancount will automatically find and create corresponding Document directives for filenames that begin with a date in directories that match mirror account names, and attach those documents to those accounts. And given a bit of configuration, the bean-file tool can automatically file downloaded documents to such a directory hierarchy.

Ledger's binding of documents to transactions works by generic meta-data. Users can attach arbitrary key-value pairs to their transactions, and those can

be filenames. Beyond that, there is no particular support for document organization.

Simpler and More Strict

Finally, Beancount has a generally simpler input syntax than Ledger. There are very few command-line options—and this is on purpose, I want to localize all the input within the file—and the directive syntax is more homogeneous: all transactions begin with a date and a keyword. If the argument of simplicity appeals to you, you might prefer to work with Beancount. I feel that the number of options offered in Ledger is daunting, and I could not claim to understand all of the possible ways they might interact with each other. If this does not worry you, you might prefer to use Ledger.

It is also more strict than Ledger. Certain kinds of Beancount inputs are not valid. Any transaction in an account needs to have an open directive to initiate the account, for example (though some of these constraints can be relaxed via optional plugins). If you maintain a Beancount ledger, you can expect to have to normalize it to fix a number of common errors being reported. I view this as a good thing: it detects many potential problems and applies a number of strict constraints to its input which allows us to make reasonable assumptions later on when we process the stream of directives. If you don't care about precision or detecting potential mistakes, Ledger will allow you to be more liberal. On the other hand if you care to produce a precise and flawless account of transactions, Beancount offers more support in its validation of your inputs.

Web Interface

Beancount has a built-in web interface, and there is an external project called Fava which significantly improves on the same theme. This is the default mode for browsing reports. I believe HLedger also has a web interface as well.

Missing Features

Beancount generally attempts to *minimize* the number of features it provides. This is in contrast with Ledger's implementation, which has received a substantial amount of feature addition in order to experiment with double-entry bookkeeping. There are a large number of options. This reflects a difference in approach: I believe that there is a small core of essential features to be identified and that forward progress is made when we are able to minimize and remove any feature that is not strictly necessary. My goal is to provide the smallest possible kernel of features that will allow one to carry out the full spectrum of bookkeeping activities, and to make it possible for users to extend the system to automate repetitive syntax.

But here is a list of features that Beancount does not support that are supported in Ledger, and that I find would be useful, that I think would be nice to have

eventually. The list below is unlikely to be exhaustive.

Console Output

Beancount's original implementation focused on providing a web view for all of its contents. During the 2.0 rewrite I began implementing some console/text outputs, mainly because I want to be able for reports to be exportable to share with others. I have a trial balance view (like Ledger's "bal" report) but for now the journal view isn't implemented.

Ledger, on the other hand, has always focused on console reports.

I'll make all the reports in Beancount support output to text format first thing after the initial release, as I'm increasingly enjoying text reports. Use bean-query --list-formats to view current status of this.

Filtering Language

Beancount does not yet have a filtering language. Until recently, its web interface was the main mode for rendering reports and exploring the contents of its ledger, and it provided limited subsets of transactions in the form of "views", e.g., per-year, per-tag, etc. Having a filtering language in particular allows one to do away with many sub-accounts. I want to simplify my chart of accounts so I need this.

I'm working on adding a simple logical expression language to do arbitrary filters on the set of Beancount transactions. This is straightforward to implement and a high priority.

No Meta-data

Beancount does not currently support meta-data. Ledger users routinely make liberal use of metadata. This has been identified as a powerful feature and a prototype has already been implemented. Meta-data will be supported on any directive type as well as on any posting. A dictionary of key-value pairs will be attached to each of these objects. Supported values will include strings, dates, numbers, currencies and amounts.

So far the plan is to restrict Beancount's own code to make no specific use of meta-data, on purpose. The meta-data will be strictly made available for user-plugins and custom user scripts to make use of.

No Arithmetic Expressions

Beancount does not support arbitrary expression evaluation in the syntax, in the places where numbers are allowed in the input. Ledger does. I have had no use for these yet, but I have no particular reason against adding this, I just haven't implemented it, as I don't need it myself.

I think an implementation would be straightforward and very low risk, a simple change to the parser and there is already a callback for numbers. I think there are many legitimate uses for it.

Limited Support for Unicode

Beancount support UTF8 or and other encodings ***in strings only*** (that is, for input that is in quotes). For example, you can enter payees and narrations with non-ASCII characters, but not account names (which aren't in quotes). Ledger supports other encodings over the entire file.

The reason for the lack of a more general encoding support in Beancount is current limitation of tokenizer tools. I've been using GNU flex to implement my lexer and it does not support arbitrary encodings. I just need to write a better lexer and make that work with Bison, it's not a difficult task. I will eventually write my own lexer manually—this has other advantages—and will write it to support Unicode (Python 3 has full support for this, so all that is required is to modify the lexer, which is one simple compilation unit). This is a relatively easy and isolated task to implement.

No Forecasting or Periodic Transactions

Beancount has no support for generating periodic transactions for forecasting, though there is a plugin provided that implements a simplistic form of it to be used as an example for how plugins work (see beancount.plugins.forecast). Ledger supports periodic transaction generation.

I do want to add this to core Beancount eventually, but I want to define the semantics very clearly before I do. Updating one's ledger is essentially the process of copying and replicating transactional data that happens somewhere else. I don't believe that regular transactions are that "regular" in reality; in my experience, there is always some amount of small variations in real transactions that makes it impossible to automatically generate series of transactions by a generator in a way that would allow the user to forego updating them one-by-one. What it is useful for in my view, is for generating *tentative* future transactions. I feel strongly that those transactions should be limited not to straddle reconciled history, and that reconciled history should replace any kind of automatically generated transactions.

I have some fairly complete ideas for implementing this feature, but I'm not using forecasting myself at the moment, so it's on the backburner. While in theory you can forecast using Ledger's periodic transactions, to *precisely* represent your account history you will likely need to adjust the beginning dates of those transactions every time you append new transactions to your accounts and replaced forecasted ones. I don't find the current semantics of automated transactions in Ledger to be very useful beyond creating an approximation of account contents.

(In the meantime, given the ease of extending Beancount with plugins, I suggest you begin experimenting with forecast transactions on your own for now, and if we can derive a generic way to create them, I'd be open to merging that into the main code.)

Prices in Beancount

Martin Blais, December 2015

http://furius.ca/beancount/doc/prices

Introduction

Processing a Beancount file is, by definition, constrained to the contents of the file itself. In particular, the latest prices of commodities are *never* fetched automatically from the internet. This is by design, so that any run of a report is deterministic and can also be run offline. No surprises.

However, we do need access to price information in order to compute market values of assets. To this end, Beancount provides a Price directive which can be used to fill its in-memory price database by inserting these price points inline in the input file:

```
2015-11-20 price ITOT 95.46 USD
2015-11-20 price LQD 115.63 USD
2015-11-21 price USD 1.33495 CAD
...
```

Of course, you could do this manually, looking up the prices online and writing the directives yourself. But for assets which are traded publicly you can automate it, by invoking some code that will download prices and write out the directives for you.

Beancount comes with some tools to help you do this. This document describes these tools.

The Problem

In the context of maintaining a Beancount file, we have a few particular needs to address.

First, we cannot expect the user to always update the input file in a timely manner. This means that we have to be able to fetch not only the latest prices, but also historical prices, from the past. Beancount provides an interface to provide these prices and a few implementations (fetching from Yahoo! Finance and from Google Finance).

Second, we don't want to store too many prices for holdings which aren't relevant to us at a particular point in time. The list of assets held in a file varies over

time. Ideally we would want to include the list of prices for just those assets, while they are held. The Beancount price maintenance tool is able to figure out which commodities it needs to fetch prices for at a particular date.

Third, we don't want to fetch the same price if it already appears in the input file. The tools detect this and skip those prices. There's also a caching mechanism that avoids redundant network calls.

Finally, while we provide some basic implementations of price sources, we cannot provide such codes for all possible online sources and websites. The problem is analogous to that of importing and extracting data from various institutions. In order to address that, we provide a mechanism for **extensibility**, a way that you can implement your own price source fetcher in a Python module and point to it from your input file by specifying the module's name as the source for that commodity.

The "bean-price" Tool

Beancount comes with a "bean-price" command-line tool that integrates the ideas above. By default, this script accepts a list of Beancount input filenames, and fetches prices required to compute latest market values for current positions held in accounts:

bean-price /home/joe/finances/joe.beancount

It is also possible to provide a list of specific price fetching jobs to run, e.g.,

bean-price **-e USD:google/TSE:XUS CAD:mysources.morningstar/RBF1005**

These jobs are run concurrently so it should be fairly fast.

Source Strings

The general format of each of these "job source strings" is

<quote-currency>:<module>/[^]<symbol>

For example:

USD:beancount.prices.sources.google/NASDAQ:AAPL

The "quote-currency" is the currency the Commodity is quoted in. For example, shares of Apple are quoted in US dollars.

The "module" is the name of a Python module that contains a Source class which can be instantiated and which connect to a data source to extract price data. These modules are automatically imported and a Source class therein instantiated in order to pull the price from the particular online source they support. This allows you to write your own fetcher codes without having to modify this script. Your code can be placed anywhere on your Python PYTHONPATH and you should not have to modify Beancount itself for this to work.

The “symbol” is a string that is fed to the price fetcher to lookup the currency. For example, Apple shares trade on the Nasdaq, and the corresponding symbol in the Google Finance source is “NASDAQ:AAPL”. Other price sources may have a different symbology, e.g., some may require the asset’s CUSIP.

Default implementations of price sources are provided; we provide fetchers for Yahoo! Finance or Google Finance, which cover a large universe of common public investment types (e.g. stock and some mutual funds). As a convenience, the module name is always first searched under the ”beancount.prices.sources” package, where those implementations live. This is how, for example, in order to use the provided Yahoo! Finance data fetcher you don’t have to write all of ”beancount.prices.sources.yahoo/AAPL” but you can simply use ”yahoo/AAPL”.

Fallback Sources

In practice, fetching prices online often fails. Data sources typically only support some limited number of assets and even then, the support may vary between assets. As an example, Google Finance supports historical prices for stocks, but does not return historical prices for currency instruments (these restrictions may be more related to contractual arrangements between them and the data providers upstream than with technical limitations).

To this extent, a source string may provide multiple sources for the data, separated with commas. For example:

USD:google/CURRENCY:GBPUSD,yahoo/GBPUSD

Each source is tried in turn, and if one fails to return a valid price, the next source is tried as a fallback. The hope is that at least one of the specified sources will work out.

Inverted Prices

Sometimes, prices are only available for the inverse of an instrument. This is often the case for currencies. For example, the price of Canadian dollars quoted in US dollars is provided by the USD/CAD market, which gives the price of a US dollar in Canadian dollars (the inverse). In order use this, you can prepend ”^” to the instrument name to instruct the tool to compute the inverse of the fetched price:

USD:google/**^**CURRENCY:USDCAD

If a source price is to be inverted, like this, the precision could be different than what is fetched. For instance, if the price of USD/CAD is 1.32759, for the above directive to price the “CAD” instrument it would output this:

2015-10-28 price CAD 0.753244601119 USD

By default, inverted rates will be rounded similarly to how other Price directives were rounding those numbers.

As you may now, Beancount's in-memory price database works in both directions (the reciprocals of all rates are stored automatically). So if you prefer to have the output Price entries with swapped currencies instead of inverting the rate number itself, you can use the --swap-inverted option. In the previous example for the price of CAD, it would output this:

2015-10-28 price USD 1.32759 CAD

Date

By default, the latest prices for the assets are pulled in. You can use an option to fetch prices for a desired date in the past instead:

bean-price –**date=2015-02-03** ...

If you are using an input file to specify the list of prices to be fetched, the tool will figure out the list of assets held on the books *at that time* and fetch historical prices for those assets only.

Caching

Prices are automatically cached (if current and latest, prices are cached for only a short period of time, about half an hour). This is convenient when you're having to run the script multiple times in a row for troubleshooting.

You can disable the cache with an option:

```
bean-price --no-cache
```

You can also instruct the script to clear the cache before fetching its prices:

bean-price --clear-cache

Prices from a Beancount Input File

Generally, one uses a Beancount input file to specify the list of currencies to fetch. In order to do that, you should have Commodity directives in your input file for each of the currencies you mean to fetch prices for, like this:

```
2007-07-20 commodity VEA
```

The "price" metadata should contain a list of price source strings. For example, a stock product might look like this:

```
2007-07-20 commodity CAD
```

While a currency may have multiple target currencies it needs to get converted to:

```
1990-01-01 commodity GBP
```

Which Assets are Fetched

There are many ways to compute a list of commodities with needed prices from a Beancount input file:

1. **Commodity directives.** The list of all Commodity directives with "price" metadata present in the file. For each of those holdings, the directive is consulted and its "price" metadata field is used to specify where to fetch prices from.

2. **Commodities of assets held at cost.** Prices for all the holdings that were seen held at cost at a particular date. Because these holdings are held at cost, we can assume there is a corresponding time-varying price for their commodity.

3. **Converted commodities.** Prices for holdings which were price-converted from some other commodity in the past (i.e., converting some cash in a currency from another).

By default, the list of tickers to be fetched includes only the intersection of these three lists. This is because the most common usage of this script is to fetch missing prices for a particular date, and only the needed ones.

Inactive commodities. You can use the "--inactive" option to fetch the entire set of prices from (1), regardless of asset holdings determined in (2) and (3).

Undeclared commodities. Commodities without a corresponding "Commodity" directive will be ignored by default. To include the full list of commodities seen in an input file, use the "--undeclared" option.

Clobber. Existing price directives for the same data are excluded by default, since the price is already in the file. You can use "--clobber" to ignore existing price directives and avoid filtering out what is fetched.

Finally, you can use "--all" to include inactive and undeclared commodities and allow clobbering existing ones. You probably don't want to use that other than for testing.

If you'd like to do some troubleshooting and print out the list of seen commodities, use the "--verbose" option twice, i.e., "-vv". You can also just print the list of prices to be fetched with the "--dry-run" option, which stops short of actually fetching the missing prices.

Conclusion

Writing Your Own Script

If the workflow defined by this tool does not fit your needs and you would like to cook up your own script, you should not have to start from scratch; you should be able to reuse the existing price fetching code to do that. I'm hoping to provide a few examples of such scripts in the experiments directory. For example, given

an existing file it would be convenient to fetch all prices of assets every Friday in order to fill up a history of missing prices. Another example would be to fetch all price directives required in order to correctly compute investment returns in the presence of contributions and distributions.

Contributions

If this workflow suits your needs well and you'd like to contribute some price source fetcher to Beancount, please contact the mailing-list. I'm open to including very general fetchers that have been very carefully unit-tested and used for a while.

Importing External Data in Beancount

Martin Blais, March 2016

http://furius.ca/beancount/doc/ingest

Introduction

This is the user's manual for the library and tools in Beancount which can help you **automate the importing of external transaction data** into your Beancount input file and manage the documents you download from your financial institutions' websites.

The Importing Process

People often wonder how we do this, so let me describe candidly and in more detail what we're talking about doing here.

The essence of the task at hand is to transcribe the transactions that occur in a person's entire set of accounts to a single text file: the Beancount input file. Having the entire set of transactions ingested in a single system is what we need to do in order to generate comprehensive reports about one's wealth and expenses. Some people call this "reconciling".

We could transcribe all the transactions manually from paper statements by typing them in. However nowadays most financial institutions have a website where you can download a statement of historical transactions in a number of data formats which you can parse to output Beancount syntax for them.

Importing transactions from these documents involves:

- Manually reviewing the transactions for **correctness** or even fraud;
- **Merging** new transactions with previous transactions imported from another account. For example, a payment from a bank account to pay off one's credit card will typically be imported from both the bank AND the credit card account. You must manually merge the corresponding transactions together[1].
- Assigning the right **category** to an expense transaction
- **Organizing** your file by moving the resulting directives to the right place in your file.

- **Verifying balances** either visually or inserting a Balance directive which asserts what the final account balance should be after the new transactions are inserted.

If my importers work without bugs, this is a process that takes me 30-60 minutes to update the majority of my active accounts. Less active accounts are updated every quarter or when I feel like it. I tend to do this on Saturday morning maybe twice per month, or sometimes weekly. If you maintain a well-organized input file with lots of assertions, mismatches are easily found, it's a pleasant and easy process, and after you're done generating an updated balance sheet is rewarding (I typically re-export to a Google Finance portfolio).

Automating Network Downloads

The downloading of files is not something I automate, and Beancount provides no tools to connect to the network and fetch your files. There is simply too great a variety of protocols out there to make a meaningful contribution to this problem[2]. Given the nature of today's secure websites and the castles of JavaScript used to implement them, it would be a nightmare to implement. Web scraping is probably too much to be a worthwhile, viable solution.

I **manually** log into the various websites with my usernames & passwords and click the right buttons to generate the downloaded files I need. These files are recognized automatically by the importers and extracting transactions and filing the documents in a well-organized directory hierarchy is automated using the tools described in this document.

While I'm not scripting the fetching, I think it's possible to do so on some sites. That work is left for you to implement where you think it's worth the time.

Typical Downloads

Here's a description of the typical kinds of files involved; this describes my use case and what I've managed to do. This should give you a qualitative sense of what's involved.

- **Credit cards and banks** provide fairly good quality historical statement downloads in OFX or CSV file formats but I need to categorize the other side of those transactions manually and merge some of the transactions together.
- **Investment accounts** provide me with great quality of processable statements and the extraction of purchase transactions is fully automated, but I need to manually edit sales transactions in order to associate the correct cost basis. Some institutions for specialized products (e.g., P2P lending) provide only PDF files and those are translated manually.
- **Payroll stubs and vesting events** are usually provided only as PDFs and I don't bother trying to extract data automatically; I transcribe those

manually, keeping the input very regular and with postings in the same order as they appear on the statements. This makes it easier.

- **Cash transactions**: I have to enter those by hand. I only book non-food expenses as individual transactions directly, and for food maybe once every six months I'll count my wallet balance and insert a summarizing transaction for each month to debit away the cash account towards food to make it balance. If you do this, you end up with surprisingly little transactions to type manually, maybe just a few each week (it depends on lifestyle choices, for me this works). When I'm on the go, I just note those on my phone in Google Keep and eventually transcribe them after they accumulate.

Extracting Data from PDF Files

I've made some headway toward converting data from PDF file, which is a common need, but it's incomplete; it turns out that fully automating table extraction from PDF isn't easy in the general case. I have some code that is close to working and will release it when the time is right. Otherwise, the best FOSS solution I've found for this is a tool called TabulaPDF but you still need to manually identify where the tables of data are located on the page; you may be able to automate some fetching its sister project tabula-java.

Nevertheless, I usually have good success with my importers grepping around PDF statements converted to ugly text in order to identify what institution they are for and extracting the date of issuance of the document.

Finally, there are a number of different tools used to extract text from PDF documents, such as PDFMiner, LibreOffice, the xpdf library, the poppler library[3] and more... but none of them works consistently on all input documents; you will likely end up installing many and relying on different ones for different input files. For this reason, I'm not requiring a dependency on PDF conversion tools from within Beancount. You should test what works on your specific documents and invoke those tools from your importer implementations.

Tools

There are three Beancount tools provided to orchestrate the three stages of importing:

1. **bean-identify**: Given a messy list of downloaded files (e.g. in ~/Downloads), automatically identify which of your configured importers is able to handle them and print them out. This is to be used for debugging and figuring out if your configuration is properly associating a suitable importer for each of the files you downloaded;

2. **bean-extract**: Extracting transactions and statement date from each file, if at all possible. This produces some Beancount input text to be moved to your input file;

3. **bean-file**: Filing away the downloaded files to a directory hierarchy which mirrors the chart of accounts, for preservation, e.g. in a personal git repo. The filenames are cleaned, the files are moved and an appropriate statement date is prepended to each of them so that Beancount may produce corresponding Document directives.

Invocation

All tools accept the same input parameters:

```
bean-<tool> <config> <downloads-dir>
```

For example,

```
bean-extract blais.config ~/Downloads
```

The filing tool accepts an extra option that lets the user decide where to move the files, e.g.,

```
bean-file -o ~/accounting/documents blais.config ~/Downloads
```

Its default behavior is to move the files to the same directory as that of the configuration file.

Configuration

The tools introduced previously orchestrate the processes, but they don't do all that much of the concrete work of groking the individual downloads themselves. They call methods on importer objects. You must provide a list of such importers; this list is the configuration for the importing process (without it, those tools don't do anything useful).

For each file found, each of the importers is called to assert whether it can or cannot handle that file. If it deems that it can, methods can be called to produce a list of transactions, extract a date, or produce a cleaned up filename for the downloaded file.

The configuration should be a Python3 module in which you instantiate the importers and assign the list to the module-level "CONFIG" variable, like this:

```
#!/usr/bin/env python3
from myimporters.bank import acmebank
from myimporters.bank import chase
…

CONFIG = [
  acmebank.Importer(),
  chase.Importer(),
  …
]
```

Of course, since you're crafting a Python script, you can insert whatever other code in there you like. All that matters is that this "CONFIG" variable refer to a list of objects which comply with the importer protocol (described in the next section). Their order does not matter.

In particular, it's a good idea to write your importers as generically as possible and to parameterize them with the particular account names you use in your input file. This helps keep your code independent of the particular accounts and forces you to define logical accounts, and I've found that this helps with clarity.

Or not… at the end of the day, these importer codes live in some of your own personal place, not with Beancount. If you so desire, you can keep them as messy and unshareable as you like.

Configuring from an Input File

An interesting idea that I haven't tested yet is to use one's Beancount input file to infer the configuration of importers. If you want to try this out and hack something, you can load your input file from the import configuration Python config, by using the API's beancount.loader.load_file() function.

Writing an Importer

Each of the importers must comply with a particular protocol and implement at least some of its methods. The full detail of this protocol is best found in the source code itself: importer.py. The tools above will take care of finding the downloads and invoking the appropriate methods on your importer objects.

Here's a brief summary of the methods you need to, or may want to, implement:

- name(): This method provides a unique id for each importer instance. It's convenient to be able to refer to your importers with a unique name; it gets printed out by the identification process, for instance.
- identify(): This method just returns true if this importer can handle the given file. You must implement this method, and all the tools invoke it to figure out the list of (file, importer) pairs.
- extract(): This is called to attempt to extract some Beancount directives from the file contents. It must create the directives by instantiating the objects defined in beancount.core.data and return them.
- file_account(): This method returns the root account associated with this importer. This is where the downloaded file will be moved by the filing script
- file_date(): If a date can be extracted from the statement's contents, return it here. This is useful for dated PDF statements… it's often possible using regular expressions to grep out the date from a PDF converted to

text. This allows the filing script to prepend a relevant date instead of using the date when the file was downloaded (the default).

- file_name(): It's most convenient not to bother renaming downloaded files. Oftentimes, the files generated from your bank either all have a unique name and they end up getting renamed by your browser when you download multiple ones and the names collide. This function is used for the importer to provide a "nice" name to file the download under.

So basically, you create some module somewhere on your PYTHONPATH—anywhere you like, somewhere private—and you implement a class, something like this:

```
from beancount.ingest import importer

class Importer(importer.ImporterProtocol):

    def identify(self, file):
        …

    # Override other methods…
```

Typically I create my importer module files in directories dedicated to each importer, so that I can place example input files all in that directory for regression testing.

Regression Testing your Importers

I've found over time that regression testing is *key* to maintaining your importer code working. Importers are often written against file formats with no official spec and unexpected surprises routinely occur. For example, I have XML files with some unescaped "&" characters, which require a custom fix just for that bank[4]. I've also witnessed a discount brokerage switching its dates format between MM/DD/YY and DD/MM/YY; that importer now needs to be able to handle both types. So you make the necessary adjustment, and eventually you find out that something else breaks; this isn't great. And the timing is particularly annoying: usually things break when you're trying to update your ledger: you have other things to do.

The easiest, laziest and most relevant way to test those importers is to use some **real data files** and compare what your importer extracts from them to expected outputs. For the importers to be at least somewhat reliable, you really need to be able to reproduce the extractions on a number of real inputs. And since the inputs are so unpredictable and poorly defined, it's not practical to write exhaustive tests on what they could be. In practice, I have to make at least *some* fix to *some* of my importers every couple of months, and with this process, it only sinks about a half-hour of my time: I add the new downloaded file which causes breakage to the importer directory, I fix the code by running it there

locally as a test. And I also run the tests over *all* the previously downloaded test inputs in that directory (old and new) to ensure my importer is still working as intended on the older files.

There is some support for automating this process in beancount.ingest.regression. What we want is some routine that will list the importer's package directory, identify the input files which are to be used for testing, and generate a suite of unit tests which compares the output produced by importer methods to the contents of "expected files" placed next to the test file.

For example, given a package with an implementation of an importer and two sample input files:

```
/home/joe/importers/acmebank/__init__.py   <- code goes here
/home/joe/importers/acmebank/sample1.csv
/home/joe/importers/acmebank/sample2.csv
```

You can place this code in the Python module (the ___init___.py file):

```
from beancount.ingest import regression
…
def test():
    importer = Importer(...)
    yield from regression.compare_sample_files(importer)
```

If your importer overrides the extract() and file_date() methods, this will generate four unit tests which get run automatically by nosetests:

1. A test which calls extract() on sample1.csv, prints the extracted entries to a string, and compares this string with the contents of sample1.csv.extract
2. A test which calls file_date() on sample1.csv and compares the date with the one found in the sample1.csv.file_date file.
3. A test like (1) but on sample2.csv
4. A test like (2) but on sample2.csv

Generating Test Input

At first, the files containing the expected outputs do not exist. When an expected output file is absent like this, the regression tests automatically generate those files from the extracted output. This would result in the following list of files:

```
/home/joe/importers/acmebank/__init__.py   <- code goes here
/home/joe/importers/acmebank/sample1.csv
/home/joe/importers/acmebank/sample1.csv.extract
/home/joe/importers/acmebank/sample1.csv.file_date
/home/joe/importers/acmebank/sample2.csv
/home/joe/importers/acmebank/sample2.csv.extract
```

```
/home/joe/importers/acmebank/sample2.csv.file_date
```

You should inspect the contents of the expected output files to visually assert that they represent the contents of the downloaded files.

If you run the tests again with those files present, the expected output files will be used as inputs to the tests. If the contents differ in the future, the test will fail and an error will be generated. (You can test this out now if you want, by manually editing and inserting some unexpected data in one of those files.)

When you edit your source code, you can always re-run the tests to make sure it still works on those older files. When a newly downloaded file fails, you repeat the process above: You make a copy of it in that directory, fix the importer, run it, check the expected files. That's it[5].

Making Incremental Improvements

Sometimes I make improvements to the importers that result in more or better output being generated even in the older files, so that all the old tests will now fail. A good way to deal with this is to keep all of these files under source control, locally delete all the expected files, run the tests to regenerate new ones, and then diff against the most recent commit to check that the changes are as expected.

Caching Data

Some of the data conversions for binary files can be costly and slow. This is usually the case for converting PDF files to text[6]. This is particularly painful, since in the process of ingesting our downloaded data we're typically going to run the tools multiple times—at least twice if everything works without flaw: once to extract, twice to file—and usually many more times if there are problems. For this reason, we want to cache these conversions, so that a painful 40 second PDF-to-text conversion doesn't have to be run twice, for example.

Beancount aims to provide two levels of caching for conversions on downloaded files:

1. An in-memory caching of conversions so that multiple importers requesting the same conversion runs them only once, and
2. An on-disk caching of conversions so that multiple invocations of the tools get reused.

In-Memory Caching

In-memory caching works like this: Your methods receive a wrapper object for a given file and invoke the wrapper's convert() method, providing a converter callable/function.

```
class MyImporter(ImporterProtocol):
    ...
    def extract(self, file):
        text = file.convert(slow_convert_pdf_to_text)
        match = re.search(..., text)
```

This conversion is automatically memoized: if two importers or two different methods use the same converter on the file, the conversion is only run once. This is a simple way of handling redundant conversions in-memory. Make sure to always call those through the .convert() method and share the converter functions to take advantage of this.

On-Disk Caching

At the moment. Beancount only implements (1). On-disk caching will be implemented later. *Track this ticket for status updates.*

Organizing your Files

The tools described in this document are pretty flexible in terms of letting you specify

- **Import configuration**: The Python file which provides the list of importer objects as a configuration;
- **Importers implementation**: The Python modules which implement the individual importers and their regression testing files;
- **Downloads directory**: Which directory the downloaded files are to be found in;
- **Filing directory**: Which directory the downloaded files are intended to be filed to.

You can specify these from any location you want. Despite this, some people are often asking how to organize their files, so I provide a template example under beancount/examples/ingest/office, and I describe this here.

I recommend that you create a Git or Mercurial[7] source-controlled repository following this structure:

```
office
  documents
     Assets
     Liabilities
     Income
     Expenses
  importers
     __init__.py
     …
```

```
        __init__.py
        sample-download-1.csv
        sample-download-1.extract
        sample-download-1.file_date
        sample-download-1.file_name
  personal.beancount
  personal.import
```

The root "office" directory is your repository. It contains your ledger file ("personal.beancount"), your importer configuration ("personal.import"), your custom importers source code ("importers/") and your history of documents ("documents/"), which should be well-organized by bean-file. You always run the commands from this root directory.

An advantage of storing your documents in the same repository as your importers source code is that you can just symlink your regression tests to some files under the documents/ directory.

You can check your configuration by running identify:

```
bean-identify example.import ~/Downloads
```

If it works, you can extract transactions from your downloaded files at once:

```
bean-extract -e example.beancount example.import ~/Downloads > tmp.beancount
```

You then open tmp.beancount and move its contents to your personal.beancount file.

Once you're finished, you can stash away the downloaded files for posterity like this:

```
bean-file example.import ~/Downloads -o documents
```

If my importers work, I usually don't even bother opening those files. You can use the --dry-run option to test moving destinations before doing so.

To run the regression tests of the custom importers, use the following command:

```
pytest -v importers
```

Personally, I have a Makefile in my root directory with these targets to make my life easier. Note that you will have to install "pytest", which is a test runner; it is often packaged as "python3-pytest" or "pytest".

Example Importers

Beyond the documentation above, I cooked up an example importer for a made-up CSV file format for a made-up investment account. See this directory.

There's also an example of an importer which uses an external tool (PDFMiner2) to convert a PDF file to text to identify it and to extract the statement date from it. See this directory.

Beancount also comes with some very basic generic importers. See this directory.

- There is a simple OFX importer that has worked for me for a long time. Though it's pretty simple, I've used it for years, it's good enough to pull info out of most credit card accounts.
- There are also a couple of mixin classes you can mix into your importer implementation to make it more convenient; these are relics from the LedgerHub project—you don't really need to use them—which can help in the transition to it.

Eventually I plan to build and provide a generic CSV file parser in this framework, as well as a parser for QIF files which should allow one to transition from Quicken to Beancount. (I need example inputs to do this; if you're comfortable sharing your file I could use it to build this, as I don't have any real input, I don't use Quicken.) It would also be nice to build a converter from GnuCash at some point; this would go here as well.

Cleaning Up

Automatic Categorization

A frequently asked question and common idea from first-time users is "How do I automatically assign a category to transactions I've imported which have only one side?" For example, importing transactions from a credit card account usually provides only one posting, like this:

```
2016-03-18 * "UNION MARKET"
  Liabilities:US:CreditCard    -12.99 USD
```

For which you must manually insert an Expenses posting, like this:

```
2016-03-18 * "UNION MARKET"
  Liabilities:US:CreditCard    -12.99 USD
  Expenses:Food:Grocery
```

People often have the impression that it is time-consuming to do this.

My standard answer is that while it would be fun to have, if you have a text editor with account name completion configured properly, it's a breeze to do this manually and you don't really need it. You wouldn't save much time by automating this away. And personally I like to go over each of the transactions to check what they are and sometimes add comments (e.g., who I had dinner with, what that Amazon charge was for, etc.) and that's when I categorize.

It's something that could eventually be solved by letting the user provide some simple rules, or by using the history of past transactions to feed into a simple learning classifier.

Beancount does not currently provide a mechanism to automatically categorize transactions. You can build this into your importer code. I want to provide a

hook for the user to register a completion function that could run across all the importers where you could hook that code in.

Cleaning up Payees

The payees that one can find in the downloads are usually ugly names:

- They are sometimes the legal names of the business, which often does not reflect the street name of the place you went, for various reasons. For example, I recently ate at a restaurant called the "Lucky Bee" in New York, and the memo from the OFX file was "KING BEE".
- The names are sometimes abbreviation or contain some crud. In the previous example, the actual memo was "KING BEE NEW YO", where "NEW YO" is a truncated location string.
- The amount of ugliness is inconsistent between data sources.

It would be nice to be able to normalize the payee names by translating them at import time. I think you can do most of it using some simple rules mapping regular expressions to names provided by the user. There's really no good automated way to obtain the "clean name" of the payee.

Beancount does not provide a hook for letting you do this this yet. It will eventually. You could also build a plugin to rename those accounts when loading your ledger. I'll build that too—it's easy and would result in much nicer output.

Future Work

A list of things I'd really want to add, beyond fortifying what's already there:

- A generic, configurable CSV importer which you can instantiate. I plan to play with this a bit and build a sniffer that could automatically figure out the role of each column.
- A hook to allow you to register a callback for post-processing transactions that works across all importers.

Related Discussion Threads

- Getting started; assigning accounts to bank .csv data
- Status of LedgerHub… how can I get started?
- Rekon wants your CSV files

Historical Note

There once was a first implementation of the process described in this document. The project was called LedgerHub and has been decommissioned in February 2016, rewritten and the resulting code integrated in Beancount itself, into this

beancount.ingest library. The original project was intended to include the implementation of various importers to share them with other people, but this sharing was not very successful, and so the rewrite includes only the scaffolding for building your own importers and invoking them, and only a very limited number of example importer implementations.

Documents about LedgerHub are preserved, and can help you understand the origins and design choices for Beancount's importer support. They can be found here:

- Original design
- Original instructions & final status (the old version of this doc)
- An analysis of the reasons why it the project was terminated (post-mortem)

[1] There are essentially three conceptual modes of entering such transactions: (1) a user crafts a single transaction manually, (2) another where a user inputs the two sides as a single transaction to transfer accounts, and (3) the two separate transactions get merged into a single one automatically. These are dual modes of each other. The twist in this story is that the same transaction often posts at different dates in each of its accounts. Beancount currently [March 2016] does not support multiple dates for a single transaction's postings, but a discussion is ongoing to implement support for these input modes. See this document for more details.

[2] The closest to universal downloader you will find in the free software world is ofxclient for OFX files, and in the commercial world, Yodlee provides a service that connects to many financial institutions.

[3] The 'pdftotext' utility in poppler provides the useful '-layout' flag which outputs a text file without mangling tables, which can be helpful in the normal case of 'transaction-per-row'

[4] After sending them a few detailed emails about this and getting no response nor seeing any change in the downloaded files, I have given up on them fixing the issue.

[5] As you can see, this process is partly why I don't share my importers code. It requires the storage of way too much personal data in order to keep them working.

[6] I don't really understand why, since opening them up for viewing is almost instant, but nearly all the tools to convert them to other formats are vastly slower.

[7] I personally much prefer Mercurial for the clarity of its commands and output and its extensibility, but an advantage of Git's storage model is that moving files within it comes for free (no extra copy is stored). Moving files in a Mercurial repository costs you a bit in storage space. And if you rename accounts or

change how you organize your files you will end up potentially copying many large files.

Command-line Accounting Cookbook

Martin Blais, July 2014

http://furius.ca/beancount/doc/cookbook

Introduction

The best way to learn the double-entry method is to look at real-world examples. The method is elegant, but it can seem unintuitive to the newcomer how transactions have to be posted in order to perform the various operations that one needs to do in counting for different types financial events. This is why I wrote this cookbook. It is not meant to be a comprehensive description of all the features supported, but rather a set of practical guidelines to help you solve problems. I think this will likely be the most useful document in the Beancount documentation set!

All the examples here apply to any double-entry accounting system: Ledger, GnuCash, or even commercial systems. Some of the details may differ only slightly. This cookbook is written using the syntax and calculation method of the Beancount software. This document also assumes that you are already familiar with the general balancing concepts of the double-entry method and with at least some of the syntax of Beancount which is available from its user's manual or its cheat sheet. If you haven't begun writing down your first file, you will want to read Getting Started with Beancount and do that first.

Command-line accounting systems are agnostic about the types of things they can count and allow you to get creative with the kinds of units that you can invent to track various kinds of things. For instance, you can count "IRA contribution dollars," which are not real dollars, but which correspond to "possible contributions in real dollars," and you obtain accounts of assets, income and expenses types for them - it works. Please do realize that some of those clever tricks may not be possible in more traditional accounting systems. In addition, some of the operations that would normally require a manual process in these systems can be automated away for us, e.g., "closing a year" is entirely done by the software at any point in time, and balance assertions provide a safeguard that allow us to change the details of past transactions with little risk, so there is no need to "reconcile" by baking the past into a frozen state. More flexibility is at hand.

Finally, if you have a transaction entry problem that is not covered in this document, please do leave a comment in the margin, or write up your problem to the Ledger mailing-list. I would like for this document to cover as many realistic scenarios as possible.

A Note of Caution

While reading this, please take note that the author is a dilettante: I am a computer scientist, not an accountant. In fact, apart from a general course I took in college and having completed the first year of a CFA program, I have no real training in accounting. Despite this, I do have some practical experience in maintaining three set of books using this software: my personal ledger (8 years worth of full financial data for all accounts), a joint ledger with my spouse, and the books of a contracting and consulting company I used to own. I also used my

double-entry system to communicate with my accountant for many years and he made suggestions. Nevertheless… I may be making fundamental mistakes here and there, and I would appreciate you leaving a comment in the margin if you find anything dubious.

Account Naming Conventions

You can define any account name you like, as long as it begins with one of the five categories: Assets, Liabilities, Income, Expenses, or Equity (note that you can customize those names with options - see the Language Syntax document for details). The accounts names are generally defined to have multiple name *components,* separated by a colon (:), which imply an accounts hierarchy, or "chart of accounts":

```
Assets:Component1:Component2:Component3:...
```

Over time, I've iterated over many ways of defining my account names and I have converged to the following convention for Assets, Liabilities, and Income accounts:

```
Type : Country : Institution : Account : SubAccount
```

What I like about this is that when you render a balance sheet, the tree that gets rendered nicely displays accounts by country first, then by institution.

Some example account names:

```
Assets:US:BofA:Savings       ; Bank of America "Savings" account
Assets:CA:RBC:Checking       ; Royal Bank of Canada "Checking" account
Liabilities:US:Amex:Platinum ; American Express Platinum credit card
Liabilities:CA:RBC:Mortgage  ; Mortgage loan account at RBC
Income:US:ETrade:Interest    ; Interest payments in E*Trade account
Income:US:Acme:Salary        ; Salary income from ACME corp.
```

Sometimes I use a further sub-account or two, when it makes sense. For example, Vanguard internally keeps separate accounts depending on whether the contributions were from the employee or the employer's matching amount:

```
Assets:US:Vanguard:Contrib401k:RGAGX  ; My contributions to this fund
Assets:US:Vanguard:Match401k:RGAGX    ; Employer contributions
```

For investment accounts, I tend organize all their contents by storing each particular type of stock in its own sub-account:

```
Assets:US:ETrade:Cash        ; The cash contents of the account
Assets:US:ETrade:FB          ; Shares of Facebook
Assets:US:ETrade:AAPL        ; Shares of Apple
Assets:US:ETrade:MSFT        ; Shares of Microsoft
…
```

This automatically organizes the balance sheet by types of shares, which I find really nice.

Another convention that I like is to use the same institution component name when I have different related types of accounts. For instance, the E*Trade assets account above has associated income streams that would be booked under similarly named accounts:

```
Income:US:ETrade:Interest    ; Interest income from cash deposits
Income:US:ETrade:Dividends   ; Dividends received in this account
Income:US:ETrade:PnL         ; Capital gains or losses from trades
…
```

For "Expenses" accounts, I find that there are generally no relevant institutions. For those it makes more sense to treat them as categories and just have a simple hierarchy that corresponds to the kinds of expenses they count, some examples:

```
Expenses:Sports:Scuba        ; All matters of diving expenses
Expenses:Transport:Train     ; Train (mostly Amtrak, but not always)
Expenses:Transport:Bus       ; Various "chinese bus" companies
Expenses:Transport:Flights   ; Flights (various airlines)
…
```

I have a *lot* of these, like 250 or more. It is really up to you to decide how many to define and how finely to aggregate or "categorize" your expenses this way. But of course, you should only define them as you need them; don't bother defining a huge list ahead of time. It's always easy to add new ones.

It is worth noting that the institution does not have to be a "real" institution. For instance, I owned a condo unit in a building, and I used the Loft4530 "institution" for all its related accounts:

```
Assets:CA:Loft4530:Property
Assets:CA:Loft4530:Association
Income:CA:Loft4530:Rental
Expenses:Loft4530:Acquisition:Legal
Expenses:Loft4530:Acquisition:SaleAgent
Expenses:Loft4530:Loan-Interest
Expenses:Loft4530:Electricity
Expenses:Loft4530:Insurance
Expenses:Loft4530:Taxes:Municipal
Expenses:Loft4530:Taxes:School
```

If you have all of your business in a single country and have no plans to move to another, you might want to skip the country component for brevity.

Finally, for "Equity" accounts, well, …. normally you don't end up defining many of them, because these are mostly created to report the net income and currency conversions from previous years or the current exercise period on the

balance sheet. Typically you will need at least one, and it doesn't matter much what you name it:

```
Equity:Opening-Balances           ; Balances used to open accounts
```

You can customize the name of the other Equity accounts that get automatically created for the balance sheet.

Choosing an Account Type

Part of the art of learning what accounts to book transactions to is to come up with relevant account names and design a scheme for how money will flow between those accounts, by jotting down some example transactions. It can get a bit creative. As you're working out how to "count" all the financial events in your life, you will often end up wondering what account type to select for some of the accounts. Should this be an "Assets" account? Or an "Income" account? After all, other than for creating reports, Beancount doesn't treat any of these account types differently…

But this does not mean that you can just use any type willy nilly. Whether an account appears in the balance sheet or income statement does matter—there is usually a correct choice. When in doubt, here are some guidelines to choose an account type.

```
First, if the amounts to be posted to the account are only relevant to be reported for a pe
```

Based on these two indicators, you should be able to figure out any case. Let's work through some examples:

- A restaurant meal represents something that you obtained in exchange for some assets (cash) or a liability (paid by credit card). Nobody ever cares what the "sum total of all food since you were born" amounts to. Only the *transitional* value matters: "How much did I spend in restaurants *this month*?" Or, *since the beginning of the year*? Or, *during this trip?* This clearly points to an Expenses account. But you might wonder… this is a positive number, but it is money I spent? Yes, the account that you spent from was subtracted from (debited) in exchange for the expense you *received*. Think of the numbers in the expenses account as things you received that vanish into the ether right after you receive them. These meals are consumed.. and then they go somewhere. Okay, we'll stop the analogy here.
- You own some shares of a bond, and receive an interest payment. This interest is cash deposited in an Assets account, for example, a trading account. What is the other leg to be booked to?

Choosing Opening Dates

Some of the accounts you need to define don't correspond to real world accounts. The Expenses:Groceries account represents the sum total of grocery expenses

since you started counting. Personally, I like to use my *birth date* on those. There's a rationale to it: it sums all the groceries you've ever spent money on, and this started only when you came to this world.

You can use this rationale on other accounts. For example, all the income accounts associated with an employer should probably be opened at the date you began the job, and end on the date you left. Makes sense.

How to Deal with Cash

Let's start with cash. I typically define two accounts at my birth date:

```
1973-04-27 open Assets:Cash
1973-04-27 open Assets:ForeignCash
```

The first account is for active use, this represents my wallet, and usually contains only units of my operating currencies, that is, the commodities I usually think of as "cash." For me, they are USD and CAD commodities.

The second account is meant to hold all the paper bills that I keep stashed in a pocket from trips around the world, so they're out of the way in some other account and I don't see them in my cash balance. I transfer those to the main account when I do travel to such places, e.g., if I return to Japan, I'll move my JPY from Assets:ForeignCash to Assets:Cash right before the trip and use them during that time.

Cash Withdrawals

An ATM withdrawal from a checking account to cash will typically look like this:

```
2014-06-28 * "DDA WITHDRAW 0609C"
  Assets:CA:BofA:Checking                      -700.00 USD
  Assets:Cash
```

You would see this transaction be imported in your checking account transactions download.

Tracking Cash Expenses

One mistake people make when you tell them you're tracking all of your financial accounts is to assume that you have to book every single little irrelevant cash transaction to a notebook. Not so! It is your choice to decide *how many* of these cash transactions to take down (or not).

Personally, I try to minimize the amount of manual effort I put into updating my Ledger. My rule for dealing with cash is this:

```
If it is for food or alcohol, I don't track it.
```

This works for me, because the great majority of my cash expenses tend to be food (or maybe I just make it that way by paying for everything else with credit cards). Only a few receipts pile up somewhere on my desk for a couple of months before I bother to type them in.

However, you will need to make occasional adjustments to your cash account to account for these expenses. I don't actually bother doing this very often… maybe once every three months, when I feel like it. The method I use is to take a snapshot of my wallet (manually, by counting the bills) and enter a corresponding balance assertion:

```
2014-05-12 balance Assets:Cash       234.13 USD
```

Every time I do this I'll also add a cash distribution adjusted to balance the account:

```
2014-06-19 * "Cash distribution"
  Expenses:Food:Restaurant           402.30 USD
  Expenses:Food:Alcohol              100.00 USD
  Assets:Cash                      ; -502.30 USD

2014-06-20 balance Assets:Cash       194.34 USD
```

If you wonder why the amounts in the cash account don't add up (234.13 -502.30 194.34), it is because between the two assertions I added to the cash account by doing some ATM withdrawals against the checking account, and those appear somewhere else (in the checking account section). The withdrawal increased the balance of the cash account. It would appear if I rendered a journal for Assets:Cash.

I could have made my life simpler and used a Pad directive if I had booked everything to food—pad entries don't work solely at the beginning of an account's history, but also between any two balance assertions on the same account—but I want to book 80% of it to food and 20% alcohol, to more accurately represent my real usage of cash[1].

Finally, if you end up with a long time period between the times that you do this, you may want to "spread out" your expenses by adding more than one cash distribution[2] manually, so that if you generate a monthly report, a large cash expense does not appear as a single lump in or outside that month.

Salary Income

Accounting for your salary is rewarding: you will be able to obtain a summary of income earned during the year as well as the detail of where the various deductions are going, and you will enjoy the satisfaction of seeing matching numbers from your Beancount reports when you receive your W-2 form from your employer (or on your T4 if you're located in Canada).

I put all entries related to an employer in their own dedicated section. I start it by setting an event to the date I began working there, for example, using the hypothetical company "Hooli" (from the Silicon Valley show):

```
2012-12-13 event "employer" "Hooli Inc."
```

This allows me to automatically calculate the number of days I've been working there. When I leave a job, I'll change it to the new one, or an empty string, if I don't leave for another job:

```
2019-03-02 event "employer" ""
```

This section will make several assumptions. The goal is to expose you to the various ideas you can use to account for your income correctly. You will almost certainly end up having to adapt these ideas to your specific situation.

Employment Income Accounts

Then you define accounts for your pay stubs. You need to make sure that you have an account corresponding to each line of your pay stub. For example, here are some of the income accounts I define for this employment income at Hooli Inc.:

```
2012-12-13 open Income:US:Hooli:Salary          USD ; "Regular Pay"
2012-12-13 open Income:US:Hooli:ReferralBonus   USD ; "Referral bonuses"
2012-12-13 open Income:US:Hooli:AnnualBonus     USD ; "Annual bonus"
2012-12-13 open Income:US:Hooli:Match401k       USD ; "Employer 401k match"
2012-12-13 open Income:US:Hooli:GroupTermLife   USD ; "Group Term Life"
```

These correspond to regular salary pay, bonuses received for employee referrals, annual bonus, receipts for 401k (Hooli in this example will match some percentage of your contributions to your retirement account), receipts for life insurance (it appears both as an income and an expense), and benefits paid for your gym subscription. There are more, but this is a good example. (In this example I wrote down the names used in the stub as a comment, but you could insert them as metadata instead if you prefer.)

You will need to book taxes withheld at the source to accounts for that year (see the tax section for details on this):

```
2014-01-01 open Expenses:Taxes:TY2014:US:Medicare  USD
2014-01-01 open Expenses:Taxes:TY2014:US:Federal   USD
2014-01-01 open Expenses:Taxes:TY2014:US:StateNY   USD
2014-01-01 open Expenses:Taxes:TY2014:US:CityNYC   USD
2014-01-01 open Expenses:Taxes:TY2014:US:SDI       USD
2014-01-01 open Expenses:Taxes:TY2014:US:SocSec    USD
```

These accounts are for Medicare taxes, Federal, New York State and NYC taxes (yes, New York City residents have an additional tax on top of state tax), state disability insurance (SDI) payments, and finally, taxes to pay for social security.

You will also need to have some accounts defined elsewhere for the various expenses that are paid automatically from your pay:

```
2012-12-13 open Expenses:Health:Life:GroupTermLife  USD ; "Life Ins."
2012-12-13 open Expenses:Health:Dental:Insurance    USD ; "Dental"
2012-12-13 open Expenses:Health:Medical:Insurance   USD ; "Medical"
2012-12-13 open Expenses:Health:Vision:Insurance    USD ; "Vision"
2012-12-13 open Expenses:Internet:Reimbursement     USD ; "Internet Reim"
2012-12-13 open Expenses:Transportation:PreTax      USD ; "Transit PreTax"
```

These correspond to typical company group plan life insurance payments, premiums for dental, medical and vision insurances, reimbursements for home internet usage, and pre-tax payments for public transit (the city of New York allows you to pay for your MetroCard with pre-tax money through your employer).

Booking Salary Deposits

Then, when I import details for a payment via direct deposit to my checking account, it will look like this:

```
2014-02-28 * "HOOLI INC       PAYROLL"
  Assets:US:BofA:Checking               3364.67 USD
```

If I haven't received my pay stub yet, I might book it temporarily to the salary account until I do:

```
2014-02-28 * "HOOLI INC       PAYROLL"
  Assets:US:BofA:Checking               3364.67 USD
  ! Income:US:Hooli:Salary
```

When I receive or fetch my pay stub, I remove this and complete the rest of the postings. A realistic entry for a gross salary of $140,000 would look something like this:

```
2014-02-28 * "HOOLI INC       PAYROLL"
  Assets:US:BofA:Checking               3364.67 USD
  Income:US:Hooli:GroupTermLife          -25.38 USD
  Income:US:Hooli:Salary               -5384.62 USD
  Expenses:Health:Dental:Insurance         2.88 USD
  Expenses:Health:Life:GroupTermLife      25.38 USD
  Expenses:Internet:Reimbursement        -34.65 USD
  Expenses:Health:Medical:Insurance       36.33 USD
  Expenses:Transportation:PreTax          56.00 USD
  Expenses:Health:Vision:Insurance         0.69 USD
  Expenses:Taxes:TY2014:US:Medicare       78.08 USD
  Expenses:Taxes:TY2014:US:Federal      1135.91 USD
  Expenses:Taxes:TY2014:US:CityNYC        75.03 USD
  Expenses:Taxes:TY2014:US:SDI             1.20 USD
  Expenses:Taxes:TY2014:US:StateNY       340.06 USD
```

```
  Expenses:Taxes:TY2014:US:SocSec         328.42 USD
```

It's quite unusual for a salary payment to have no variation at all from its previous one: rounding up or down from the payroll processor will often result in a difference of a penny, social security payments will cap to their maximum, and there are various deductions that will occur from time to time, e.g., deductions on taxable benefits received. Moreover, contributions to a 401k will affect that amounts of taxes withheld at the source. Therefore, you end up having to look at each pay stub individually to enter its information correctly. But this is not as time-consuming as it sounds! Here's a trick: it's a lot easier to update your transactions if you list your postings *in the same order as they appear* on your pay stub. You just copy-paste the previous entry, read the pay stub from top to bottom and adjust the numbers accordingly. It takes a minute for each.

It's worth noting some unusual things about the previous entry. The "group term life" entry has both a $25.38 income leg and an expense one. This is because Hooli pays for the premium (it reads exactly like that on the stubs.) Hooli also reimburses some of home internet, because I use it to deal with production issues. This appears as a *negative* posting to reduce the amount of my expense Expenses:Internet account.

Vacation Hours

Our pay stubs also include accrued vacation and the total vacation balance, in vacation hours. You can also track these amounts on the same transactions. You need to declare corresponding accounts:

```
2012-12-13 open Income:US:Hooli:Vacation           VACHR
2012-12-13 open Assets:US:Hooli:Vacation           VACHR
2012-12-13 open Expenses:Vacation:Hooli            VACHR
```

Vacation that accrues is something you receive and thus is treated as *Income* in units of "VACHR", and accumulates in an *Assets* account, which holds how many of these hours you currently have available to "spend" as time off. Updating the previous salary income transaction entry:

```
2014-02-28 * "HOOLI INC       PAYROLL"
  Assets:US:BofA:Checking               3364.67 USD
  …
  Assets:US:Hooli:Vacation                 4.62 VACHR
  Income:US:Hooli:Vacation                -4.62 VACHR
```

4.62 VACHR on a bi-weekly paycheck 26 times per year is 26 x 4.62 ~= 120 hours. At 8 hours per day, that is 15 work days, or 3 weeks, which is a standard vacation package for new US Hooligans in this example.

When you do take time off, you book an expense against your accumulated vacation time:

```
2014-06-17 * "Going to the beach today"
```

```
  Assets:US:Hooli:Vacation     -8 VACHR
  Expenses:Vacation:Hooli
```

The *Expenses* account tracks how much vacation you've used. From time to time you can check that the balance reported on your pay stub—the amount of vacation left that your employer thinks you have—is the same as that which you have accounted for:

```
2014-02-29 balance Assets:US:Hooli:Vacation   112.3400 VACHR
```

You can "price" your vacation hour units to your hourly rate, so that your vacations *Assets* account shows how much the company would pay you if you decided to quit. Assuming that $140,000/year salary, 40 hour weeks and 50 weeks of work, which is 2000 hours per year, we obtain a rate of $70/hour, which you enter like this:

```
2012-12-13 price VACHR  70.00 USD
```

Similarly, if your vacation hours expires or caps, you can calculate how much dollar-equivalent you're forfeiting by working too much and giving up your vacation time. You would write off some of the VACHR from your *Assets* account into an income account (representing losses).

401k Contributions

The 401k plan allows you to make contributions to a tax-deferred retirement account using pre-tax dollars. This is carried out via withholdings from your pay. To account for those, you simply include a posting with the corresponding contribution towards your retirement account:

```
2014-02-28 * "HOOLI INC       PAYROLL"
  Assets:US:BofA:Checking               3364.67 USD
  …
  Assets:US:Vanguard:Cash               1000.00 USD
  …
```

If you're accounting for your available contributions (see the tax section of this document), you will want to reduce your "401k contribution" *Assets* account at the same time. You would add two more postings to the transaction:

```
2014-02-28 * "HOOLI INC       PAYROLL"
  Assets:US:BofA:Checking                        3364.67 USD
  …
  Assets:US:Vanguard:Cash                        1000.00 USD
  Assets:US:Federal:PreTax401k                  -1000.00 US401K
  Expenses:Taxes:TY2014:US:Federal:PreTax401k    1000.00 US401K
  …
```

If your employer matches your contributions, this may not show on your pay stubs. Because these contributions are not taxable—they are deposited directly

to a tax-deferred account—your employer does not have to include them in the withholding statement. You will see them appear directly in your investment account as deposits. You can book them like this to the retirement account's tax balance:

```
2013-03-16 * "BUYMF - MATCH"
  Income:US:Hooli:Match401k           -1173.08 USD
  Assets:US:Vanguard:Cash              1173.08 USD
```

And then insert a second transaction when you invest this case, or directly purchasing assets from the contribution if you have specified an asset allocation and this is automated by the broker:

```
2013-03-16 * "BUYMF - MATCH"
  Income:US:Hooli:Match401k    -1173.08 USD
  Assets:US:Vanguard:VMBPX       106.741 VMBPX {10.99 USD}
```

Note that the fund that manages your 401k accounts may be tracking your contributions and your employer's contributions in separate buckets. You would declare sub-accounts for this and make the corresponding changes:

```
2012-12-13 open Assets:US:Vanguard:PreTax401k:VMBPX  VMBPX
2012-12-13 open Assets:US:Vanguard:Match401k:VMBPX   VMBPX
```

It is common for them to do this in order to track each source of contribution separately, because there are several constraints on rollovers to other accounts that depend on it.

Vesting Stock Grants

See the dedicated document on this topic for more details.

Other Benefits

You can go crazy with tracking benefits if you want. Here are a few wild ideas.

Points

If your employer offers a sponsored massage therapy program on-site, you could presumably book a massage out of your paycheck or even from some internal website (if the company is modern), and you could pay for them using some sort of internal points system, say, "Hooli points". You could track those using a made-up currency, e.g., "MASSA's" and which could be priced at 0.50 USD, the price at which you could purchase them:

```
2012-12-13 open Assets:US:Hooli:Massage    MASSA
2012-12-13 price MASSA  0.50 USD
```

When I purchase new massage points, I

```
2013-03-15 * "Buying points for future massages"
  Liabilities:US:BofA:CreditCard    -45.00 USD
  Assets:US:Hooli:Massage              90 MASSA {0.50 USD}
```

If you're occasionally awarded some of these points, and you can track that in an Income account.

Food Benefits

Like many of the larger technology companies, Hooli presumably provides free food for its employees. This saves time and encourages people to eat healthy. This is a bit of a trend in the tech world right now. This benefit does not show up anywhere, but if you want to price it as part of your compensation package, you can track it using an *Income* account:

```
2012-12-13 open Income:US:Hooli:Food
```

Depending on how often you end up eating at work, you could guesstimate some monthly allowance per month:

```
2013-06-30 * "Distribution for food eaten at Hooli"
  Income:US:Hooli:Food      -350 USD
  Expenses:Food:Restaurant
```

Currency Transfers & Conversions

If you convert between currencies, such as when performing an international transfer between banks, you need to provide the exchange rate to Beancount. It looks like this:

```
2014-03-03 * "Transfer from Swiss account"
  Assets:CH:UBS:Checking       -9000.00 CHF
  Assets:US:BofA:Checking      10000.00 USD @ 0.90 CHF
```

The balance amount of the second posting is calculated as 10,000.00 USD x 0.90 CHF/USD = 9,000 CHF, and the transaction balances. Depending on your preference, you could have placed the rate on the other posting, like this:

```
2014-03-03 * "Transfer from Swiss account"
  Assets:CH:UBS:Checking       -9000.00 CHF @ 1.11111 USD
  Assets:US:BofA:Checking      10000.00 USD
```

The balance amount of the first posting is calculated as -9000.00 CHF x 1.11111 USD/CHF = 10000.00 USD[3]. Typically I will choose the rate that was reported to me and put it on the corresponding side. You may also want to use the direction that F/X markets use for trading the rate, for example, the Swiss franc trades as USD/CHF, so I would prefer the first transaction. The price database converts the rates in both directions, so it is not that important[4].

If you use wire transfers, which is typical for this type of money transfer, you might incur a fee:

```
2014-03-03 * "Transfer from Swiss account"
  Assets:CH:UBS:Checking       -9025.00 CHF
  Assets:US:BofA:Checking      10000.00 USD @ 0.90 CHF
  Expenses:Fees:Wires             25.00 CHF
```

If you convert cash at one of these shady-looking currency exchange parlors found in tourist locations, it might look like this:

```
2014-03-03 * "Changed some cash at the airport in Lausanne"
  Assets:Cash             -400.00 USD @ 0.90 CHF
  Assets:Cash              355.00 CHF
  Expenses:Fees:Services     5.00 CHF
```

In any case, you should *never* convert currency units using the cost basis syntax, because the original conversion rate needs to be forgotten after depositing the units, and not kept around attached to simple currency. For example, this would be incorrect usage:

```
2014-03-03 * "Transfer from Swiss account"
  Assets:CH:UBS:Checking       -9000.00 CHF
  Assets:US:BofA:Checking      10000.00 USD {0.90 CHF}  ; <-bad!
```

If you did that by mistake, you would incur errors when you attempted to use the newly USD deposited: Beancount would require that you specify the cost of these "USD" in CHF, e.g., "debit from my USD that I changed at 0.90 USD/CHF". Nobody does this in the real world, and neither should you when you represent your transactions: *once the money has converted, it's just money in a different currency, with no associated cost.*

Finally, a rather subtle problem is that using these price conversions back and forth at different rates over time breaks the accounting equation to some extent: changes in exchange rate may create small amounts of money out of thin air and all the balances don't end up summing up to zero. However, this is not a problem, because Beancount implements an elegant solution to automatically correct for this problem, so you can use these conversions freely without having to worry about this: it inserts a special conversions entry on the balance sheet to invert the cumulative effect of conversions for the report and obtain a clean balance of zero. (A discussion of the conversions problem is beyond the scope of this cookbook; please refer to Solving the Conversions Problem if you'd like to know more.)

Investing and Trading

Tracking trades and associated gains is a fairly involved topic. You will find a more complete introduction to profit and loss and a detailed discussion of various scenarios in the Trading with Beancount document, which is dedicated to this topic. Here we will discuss how to setup your account and provide simple example transactions to get you started.

Accounts Setup

You should create an account prefix to root various sub-accounts associated with your investment account. Say you have an account at ETrade, this could be "Assets:US:ETrade". Choose an appropriate institution name.

Your investment account will have a cash component. You should create a dedicated sub-account will represent uninvested cash deposits:

```
2013-02-01 open Assets:US:ETrade:Cash     USD
```

I recommend that you further define a sub-account for each of the commodity types that you will invest in. Although this is not strictly necessary—Beancount accounts may contain any number of commodities—it is a nice way to aggregate all the positions in that commodity together for reporting. Say you will buy shares of LQD and BND, two popular bond ETFs:

```
2013-02-01 open Assets:US:ETrade:LQD      LQD
2013-02-01 open Assets:US:ETrade:BND      BND
```

This also helps produce nicer reports: balances are often shown at cost and it's nice to see the total cost aggregated by commodity for various reasons (i.e., each commodity provides exposure to different market characteristics). Using a dedicated sub-account for each commodity held within an institution is a good way to do that. Unless you have specific reasons not to do so, I highly suggest sticking with this by default (you can always change it later by renaming accounts).

Specifying commodity constraints on your accounts will help you detect data entry mistakes. Stock trades tend to be a bit more involved than regular transactions, and this is certainly going to be helpful.

Then, you will hopefully receive income in this account, in two forms: capital gains, or "P&L", and dividends. I like to account for these by institution, because this is how they have to be declared for taxes. You may also receive interest income. Define these:

```
2013-02-01 open Income:US:ETrade:PnL          USD
2013-02-01 open Income:US:ETrade:Dividends    USD
2013-02-01 open Income:US:ETrade:Interest     USD
```

Finally, to account for transaction fees and commissions, you will need some general accounts to receive these:

```
1973-04-27 open Expenses:Financial:Fees
1973-04-27 open Expenses:Financial:Commissions
```

Funds Transfers

You will normally add some initial money in this account by making a transfer from an external account, say, a checking account:

```
2014-02-04 * "Transferring money for investing"
  Assets:US:BofA:Checking        -2000.00 USD
  Assets:US:ETrade:Cash           2000.00 USD
```

Making a Trade

Buying stock should have a posting that deposits the new commodity in the commodity's sub-account, and debits the cash account to the corresponding amounts plus commissions:

```
2014-02-16 * "Buying some LQD"
  Assets:US:ETrade:LQD                 10 LQD {119.24 USD}
  Assets:US:ETrade:Cash          -1199.35 USD
  Expenses:Financial:Commissions     6.95 USD
```

Note that when you're buying units of a commodity, you are establishing a new trade lot in the account's inventory and it is necessary that you provide the cost of each unit (in this example, 119.24 USD per share of LQD). This allows us to account for capital gains correctly.

Selling some of the same stock work similarly, except that an extra posting is added to absorb the capital gain or loss:

```
2014-02-16 * "Selling some LQD"
  Assets:US:ETrade:LQD                 -5 LQD {119.24 USD} @ 123.40 USD
  Assets:US:ETrade:Cash            610.05 USD
  Expenses:Financial:Commissions     6.95 USD
  Income:US:Etrade:PnL
```

Note that the postings of shares removed from the Assets:US:ETrade:LQD account is a lot *reduction* and you must provide information to identify which lot you're reducing, in this case, by providing the per-share cost basis of 119.24 USD.

I normally let Beancount calculate the capital gain or loss for me, which is why I don't specify it in the last posting. Beancount will automatically balance the transaction by setting the amount of this posting to -20.80 USD, which is a *gain* of 20.80 USD (remember that the signs are inverted for income accounts). Specifying the sale price of 123.40 USD is optional, and it is *ignored* for the purpose of balancing the transaction, the cash deposit and commissions legs determine the profit.

Receiving Dividends

Receiving dividends takes on two forms. First, you can receive dividends in cash, which will go into the cash account:

```
2014-02-16 * "Dividends from LQD"
  Income:US:ETrade:Dividends      -87.45 USD
  Assets:US:ETrade:Cash            87.45 USD
```

Note that the source of the dividends isn't specified here. You could use a sub-account of the income account to count it separately.

Or you can receive dividends in shares reinvested:

```
2014-06-27 * "Dividends reinvested"
  Assets:US:Vanguard:VBMPX      1.77400 VBMPX {10.83 USD}
  Income:US:Vanguard:Dividends   -19.21 USD
```

This is booked similarly to a stock purchase, and you also have to provide the cost basis of the received units. This would typically happen in a non-taxable retirement account.

Refer to the Trading with Beancount document for a more thorough discussion and numerous and more complex examples.

Choosing a Date

Buying or selling a single lot of stock typically involves multiple events over time: the trade is placed, the trade is filled (usually on the same day), the trade is settled. Settlement usually occurs 2 or 3 business days after the trade is filled.

For simplicity, I recommend using the trade date as the date of your transaction. In the US, this is the date that is recognized for tax purposes, and settlement has no impact on your account (brokers typically won't allow you to trade without the corresponding cash or margin anyhow). So normally I don't bother creating separate entries for settlement, it's not very useful.

More complex schemes can be envisioned, e.g. you could store the settlement date as a metadata field and then use it in scripts later on, but that's beyond the scope of this document.

Conclusion

This document is incomplete. I have many more example use cases that I'm planning to add here as I complete them. I will be announcing those on the mailing-list as they materialize. In particular, the following topics will be discussed:

- Health Care Expenses, e.g., insurance premiums and rebates
- Taxes
- IRAs, 401k and other tax-deferred accounts
- Real Estate
- Options

[1] I am considering supporting an extended version of the Pad directive that can take a percentage value and make it possible to pad only a percentage of the full amount, to automate this.

[2] Yet another extension to Beancount involves support multiple Pad directives between two balance assertions and automatically support this spreading out of padding directives.

[3] If you're concerned about the issue of precision or rounding in balancing, see this document.

[4] Note that if the price database needs to invert the date its calculation may result in a price with a large number of digits. Beancount uses IEEE decimal objects and the default context of the Python implementation is 28 digits, so inverting 0.9 will result in 1.111111….111 with 28 digits.

Trading with Beancount

Martin Blais, July 2014

http://furius.ca/beancount/doc/trading

Introduction

This is a companion document for the Command-Line Accounting Cookbook that deals exclusively with the subject of trading and investments in Beancount. You probably should have read an introduction to the double-entry method before reading this document.

The subject of stock trading needs to be preceded by a discussion of "profit and loss," or P/L, for short (pronounce: "P and L"), also called capital gains or losses. The notion of P/L against multiple trades can be difficult for a novice to

understand, and I've even seen professional traders lack sophistication in their understanding of P/L over varying time periods. It is worth spending a bit of time to explain this, and necessary to understand how to book your trades in a double-entry system.

This discussion will be weaved with detailed examples of how to book these trades in Beancount, wherever possible. There is a related, active proposal for improving the booking methods in Beancount that you might also be interested in. Discussions of basis for tax-deferred accounts will not be treated here, but in the more general cookbook.

What is Profit and Loss?

Let's imagine you have an account at the E*Trade discount broker and you buy some shares of a company, say IBM. If you buy 10 shares of IBM when its price is 160$/share, it will cost you 1600$. That value is what we will call the "book value", or equivalently, "the cost." This is how much money you had to spend in order to acquire the shares, also called "the position." This is how you would enter this transaction in Beancount:

```
2014-02-16 * "Buying some IBM"
  Assets:US:ETrade:IBM                  10 IBM {160.00 USD}
  Assets:US:ETrade:Cash           -1600.00 USD
```

In practice you will probably pay some commission to E*Trade for this service, so let's put that in for completeness:

```
2014-02-16 * "Buying some IBM"
  Assets:US:ETrade:IBM                  10 IBM {160.00 USD}
  Assets:US:ETrade:Cash           -1609.95 USD
  Expenses:Financial:Commissions     9.95 USD
```

This is how you tell Beancount to deposit some units "at cost", in this case, units of "IBM at 160 USD/share" cost. This transaction balances because the sum of its legs is zero: 10 x 160 + -1609.95 + 9.95 = 0. Also note that we're choosing to use a subaccount dedicated to the shares of IBM; this is not strictly necessary but it is convenient for reporting in, for example, a balance sheet, because it will naturally aggregate all of your shares of each of your positions on their own line. Having a "cash" subaccount also emphasizes that uninvested funds you have there are not providing any return.

The next day, the market opens and IBM shares are going for 170$/share. In this context, we will call this "the price."[1] The "market value" of your position, your shares, is the number of them x the market price, that is, 10 shares x 170$/share = 1700$.

The difference between these two amounts is what we will call the P/L:

```
market value - book value = P/L
```

We will call a positive amount "a profit" and if the amount is negative, "a loss."

Realized and Unrealized P/L

The profit from the previous section is called an "unrealized profit." That is because the shares have not actually been sold yet - this is a hypothetical profit: *if* I can sell those shares at the market value, this is how much I *would* pocket. The 100$ I mentioned in the previous section is actually an "unrealized P/L."

So let's say you like this unrealized profit and you feel that it's temporary luck that IBM went up. You decide to sell 3 of these 10 shares to the market at 170$/share. The profit on these share will now be "realized":

```
market value - book value = P/L
```

This 30$ is a "realized P/L." The remaining portion of your position is still showing an unrealized profit, that is, the price could fluctuate some more until you sell it:

```
market value - book value = P/L
```

This is how you would book this partial sale of your position in Beancount (again including a commission):

```
2014-02-17 * "Selling some IBM"
  Assets:US:ETrade:IBM                -3 IBM {160.00 USD}
  Assets:US:ETrade:Cash            500.05 USD
  Expenses:Financial:Commissions     9.95 USD
```

Do you notice something funny going on here? -3 x 160 = -480, -480 + 500.05 + 9.95 = 30... This transaction does not balance to zero! The problem is that we received 510$ in cash in exchange for the 3 shares we sold. This is because the actual price we sold them at was 170$: 3 x 170 = 510$. This is where we need to account for the profit, by adding another leg which will absorb this profit, and conveniently enough, automatically calculate and track our profits for us:

```
2014-02-17 * "Selling some IBM"
  Assets:US:ETrade:IBM                -3 IBM {160.00 USD}
  Assets:US:ETrade:Cash            500.05 USD
  Expenses:Financial:Commissions     9.95 USD
  Income:US:ETrade:PnL
```

The last leg will be automatically filled in by Beancount to -30 USD, as we're allowed one posting without an amount (and remember that in the double-entry system without credits and debits, a profit is a negative number for "Income" accounts). This is the number the government is interested in for your taxes.

In summary, you now have:

```
A position of 7 "shares at book value of 160$" = 1120$ (its book value)
```

Now at this point, some of you will jump up and down and say: "But wait, waiiit! I sold at 170$/share, not 160$/share, why do you put 160$ here?" The answer is that you did not have shares held at 170$ to sell. In order to explain this, I need to make a little detour to explain how we keep track of things in accounts...

So how do we keep track of these shares?

It's actually easy: when Beancount stores things in accounts, we use something called "an inventory." Imagine that an "inventory" is a bag with the name of that account on it. Each account has one such bag to hold the things in the account at a particular point in time, the "balance" of this account at that time. Imagine that the things it contains have a little label attached to each of them, with their cost, that is, the price that was paid to acquire them. Whenever you put a thing in the bag, you attach a new label to the thing. For things to work right, all things need to be labeled[2]. In our example, the bag contained 10 items of "shares of IBM bought at 160$/share". The syntax we used to put the IBM in the account can seem a little misleading; we wrote:

```
  Assets:US:ETrade:IBM                    10 IBM {160.00 USD}
```

but really, this is understood by Beancount closer to the following syntax:

```
  Assets:US:ETrade:IBM                    10 {IBM 160.00 USD}
```

But … it would be annoying to write this, so we use a syntax more intuitive to humans.

So the thing is, you can't subtract units of {IBM at 170.00 USD}... because there just aren't any in that bag. What you have in the bag are units of {IBM at 160.00 USD}. You can only take out these ones.

Now that being said, do you see how it's the amount that was exchanged to us for the shares that really helps us track the P/L? Nowhere did we actually need to indicate the price at which we sold the shares. It's the fact that we received a certain amount of cash that is different than the cost of the position we're selling that triggers the imbalance, which we book to a capital gain.

Hmmm… Beancount maintains a price database, wouldn't it be nice to at least record and attach that price to the transaction for documentation purposes? Indeed. Beancount allows you to also attach a price to that posting, but for the purpose of balancing the transaction, it ignores it completely. It is mainly there for documentation, and you can use it if you write scripts. And if you use the beancount.plugins.implicit_prices plugin, it will be used to automatically synthesize a price entry that will enrich our historical price database, which may be used in reporting the market value of the account contents (more details on this follow).

So the complete and final transaction for selling those shares should be:

```
2014-02-17 * "Selling some IBM"
```

```
  Assets:US:ETrade:IBM          -3 IBM {160.00 USD} @ 170.00 USD
  Assets:US:ETrade:Cash          500.05 USD
  Expenses:Financial:Commissions   9.95 USD
  Income:US:ETrade:PnL
```

Trade Lots

In practice, the reality of trading gets a tiny bit more complicated than this. You might decide to buy some IBM multiple times, and each time, it is likely that you would buy them at a different price. Let's see how this works with another example trade. Given your previous position of 7 shares held at 160$ cost, the following day you see that the price went up some more, you change your mind on IBM and decide to "go long" and buy 5 more shares. The price you get is 180$/share this time:

```
2014-02-18 * "I put my chips on big blue!"
  Assets:US:ETrade:IBM                 5 IBM {180.00 USD}
  Assets:US:ETrade:Cash           -909.95 USD
  Expenses:Financial:Commissions     9.95 USD
```

Now, what do we have in the bag for Assets:US:ETrade:IBM? We have two kinds of things:

- 7 shares of "IBM held at 160 USD/share", from the first trade
- 5 shares of "IBM held at 180 USD/share", from this last trade

We will call these "lots," or "trade lots."

In fact, if you were to sell this entire position, say, a month later, the way to legally sell it in Beancount (that is, without issuing an error), is by specifying both legs. Say the price is 172$/share at that moment:

```
2014-03-18 * "Selling all my blue chips."
  Assets:US:ETrade:IBM          -7 IBM {160.00 USD} @ 172.00 USD
  Assets:US:ETrade:IBM          -5 IBM {180.00 USD}
  Assets:US:ETrade:Cash         2054.05 USD
  Expenses:Financial:Commissions   9.95 USD
  Income:US:ETrade:PnL
```

Now your final position of IBM would be 0 shares.

Alternatively, since you're selling the entire position, Beancount should be able to unambiguously match all the lots against an unspecified cost. This is equivalent:

```
2014-03-18 * "Selling all my blue chips."
  Assets:US:ETrade:IBM          -12 IBM {} @ 172.00 USD
  Assets:US:ETrade:Cash         2054.05 USD
  Expenses:Financial:Commissions   9.95 USD
  Income:US:ETrade:PnL
```

Note that this won't work if the total amount of shares doesn't match all the lots (this would be ambiguous… which subset of the lots should be chosen isn't obvious).

Booking Methods

But what if you decided to sell only some of those shares? Say you need some cash to buy a gift to your loved one and you want to sell 4 shares this time. Say the price is now 175$/share.

Now you have a choice to make. You can choose to sell the older shares and realize a larger profit:

```
2014-03-18 * "Selling my older blue chips."
  Assets:US:ETrade:IBM          -4 IBM {160.00 USD} @ 175.00 USD
  Assets:US:ETrade:Cash          690.05 USD
  Expenses:Financial:Commissions   9.95 USD
  Income:US:ETrade:PnL        ;; -60.00 USD (profit)
```

Or you may choose to sell the most recently acquired ones and realize a loss:

```
2014-03-18 * "Selling my most recent blue chips."
  Assets:US:ETrade:IBM          -4 IBM {180.00 USD} @ 175.00 USD
  Assets:US:ETrade:Cash          690.05 USD
  Expenses:Financial:Commissions   9.95 USD
  Income:US:ETrade:PnL        ;;  20.00 USD (loss)
```

Or you can choose to sell a mix of both: just use two legs.

Note that in practice this choice will depend on a number of factors:

- The tax law of the jurisdiction where you trade the shares may have a defined method for how to book the shares and you may not actually have a choice. For example, they may state that you must trade the oldest lot you bought, a method called "first-in-first out."
- If you have a choice, the various lots you're holding may have different taxation characteristics because you've held them for a different period of time. In the USA, for example, positions held for more than one year benefit from a lower taxation rate (the "long-term" capital gains rate).
- You may have other gains or losses that you want to offset in order to minimize your cash flow requirements on your tax liability. This is sometimes called "tax loss harvesting."

There are more… but I'm not going to elaborate on them here. My goal is to show you how to book these things with the double-entry method.

Dated lots

We've almost completed the whole picture of how this works. There is one more rather technical detail to add and it begins with a question: What if I bought multiple lots of share at the same price?

As we alluded to in the previous section, the duration for which you held a position may have an impact on your taxation, even if the P/L ends up being the same. How do we differentiate between these lots?

Well… I had simplified things a tiny bit earlier, just to make it simpler to understand. When we put positions in an inventory, on the label that we attach to the things we put in it, we also mark down the date that lot was acquired if you supply it. This is how you would book entering the position this way:

```
2014-05-20 * "First trade"
  Assets:US:ETrade:IBM          5 IBM {180.00 USD, 2014-05-20}
  Assets:US:ETrade:Cash           -909.95 USD
  Expenses:Financial:Commissions     9.95 USD
```

Now when you sell, you can do the same thing to disambiguate which lot's position you want to reduce:

```
2014-08-04 * "Selling off first trade"
  Assets:US:ETrade:IBM         -5 IBM {180.00 USD, 2014-05-20}
  Assets:US:ETrade:Cash            815.05 USD
  Expenses:Financial:Commissions     9.95 USD
  Income:US:ETrade:PnL
```

Note that it's really unlikely that your broker will provide the information in the downloadable CSV or OFX files from their website… you probably won't be able to automate the lot detail of this transaction, you might have to pick up the PDF trade confirmations your broker provides to enter this manually, if it ever happens. But how often does it happen that you buy two lots at the same price? I trade relatively frequently - about every two weeks - and in 8 years worth of data I don't have a single occurrence of it. In practice, unless you do thousands of trades per day- and Beancount isn't really designed to handle that kind of activity, at least not in the most efficient way - it just won't happen very much.

(*Technical Detail*: that we're working on bettering the mechanism for lot selection so that you never have to insert the lot-date yourself, and so that you could disambiguate lot selection by supplying a name instead. See upcoming changes.)

Reporting Unrealized P/L

Okay, so our account balances are holding the cost of each unit, and that provides us with the book value of these positions. Nice. But what about viewing the market value?

The market value of the positions is simply the number of units of these instruments x the market price at the time we're interested in. This price fluctuates. So we need the price.

Beancount supports a type of entry called a price entry that allows you to tell it what the price of an instrument was at a particular point in time, e.g.

```
2014-05-25 price IBM   182.27 USD
```

In order to keep Beancount simple and with few dependencies, the software does not automatically fetch these prices (you can check out LedgerHub for this purpose, or write your own script that will insert the latest prices in your input file if so desired… there are many libraries to fetch prices from the internet online). It only knows about market prices from all these price entries. Using these, it builds an in-memory historical database of prices over time and can query it to obtain the most current values.

Instead of supporting different reporting modes with options, you can trigger the insertion of unrealized gains by enabling a plugin:

```
plugin "beancount.plugins.unrealized" "Unrealized"
```

This will create a synthetic transaction at the date of the last of directives, that reflects the unrealized P/L. It books one side as Income and the other side as a change in Asset:

```
2014-05-25 U "Unrealized gain for 7 units of IBM (price:
              182.2700 USD as of 2014-05-25,
              average cost: 160.0000 USD)"
  Assets:US:ETrade:IBM:Unrealized          155.89 USD
  Income:US:ETrade:IBM:Unrealized         -155.89 USD
```

Note that I used an option in this example to specify a sub-account to book the unrealized gains to. The unrealized P/L shows up on a separate line in the balance sheet and the parent account should show the market value on its balance (which includes that of its sub-accounts).

Commissions

So far we have not discussed trading commissions. Depending on the tax law that applies to you, the costs associated with trading may be deductible from the raw capital gain as we've calculated it in the previous examples. These are considered expenses by the government, and it is often the case that you can deduct those trading commissions (it's entirely reasonable from their part, you did not pocket that money after all).

In the examples above, the capital gains and commission expenses get tracked into two separate accounts. For example, you could end up with reported balances that look like this:

```
Income:US:ETrade:PnL                  -645.02 USD
```

```
Expenses:Financial:Commissions        39.80 USD
```

(Just to be clear, this is to be interpreted as a *profit* of $645.02 and an expense of $39.80.) You could subtract these numbers to obtain an *approximation* of the P/L without costs: 645.02 - 39.80 = $605.22. However, this is only an approximation of the correct P/L value. To understand why, we need to look at an example where a partial number of shares are sold across a reporting period.

Imagine that we have an account with a commission rate of $10 per trade, 100 shares of ITOT were bought in 2013, 40 of those shares were later sold in that same year, and the remaining 60 were sold the year after, a scenario that looks like this:

```
2013-09-01 Buy 100 ITOT at $80, commission = 10$
```

If you computed the sum of commissions paid at the end of 2013, you would have $20, and using the approximate method outlined previously, for so 2013 and 2014 you would declare

```
2013: P/L of 40 x ($82 - $80) - ($10 + $10) = $60
```

However, strictly speaking, this is incorrect. The $10 commission paid on *acquiring* the 100 shares has to be pro-rated with respect to the number of shares sold. This means that on that first sale of 40 shares only 4$ of the commission is deductible: $10 x (40 shares / 100 shares), and so we obtain:

```
2013: P/L of 40 x ($82 - $80) - $(4 + 10) = $66
```

As you can see, the P/L declared for each year differs, even if the sum of the P/L for both years is the same ($290).

A convenient method to automatically allocate the acquisition costs to the pro-rata value of the number of shares sold is to add the acquisition trading cost to the total book value of the position. In this example, you would say that the position of 100 shares has a book value $8010 instead of $8000: 100 share x $80/share + $10, or equivalently, that the individual shares have a book value of $80.10 each. This would result in the following calculation:

```
2013: P/L of 40 x ($82 - $80.10) - $10 = $66
```

You could even go one step further and fold the commission on *sale* into the price of each share sold as well:

```
2013: P/L of 40 x ($81.75 - $80.10) = $66
```

This may seem overkill, but imagine that those costs were much higher, as is the case on large commercial transactions; the details do begin to matter to the tax man. Accurate accounting is important, and we need to develop a method to do this more precisely.

Stock Splits

Stock splits are currently dealt with by emptying an account's positions and recreating the positions at a different price:

```
2004-12-21 * "Autodesk stock splits"
  Assets:US:MSSB:ADSK          -100 ADSK {66.30 USD}
  Assets:US:MSSB:ADSK           200 ADSK {33.15 USD}
```

The postings balance each other, so the rule is respected. As you can see, this requires no special syntax feature. It also handles more general scenarios, such as the odd split of the Google company that occurred on the NASDAQ exchange in April 2014, into two different classes of stock (voting and non-voting shares, at 50.08% and 49.92%, respectively):

```
2014-04-07 * "Stock splits into voting and non-voting shares"
  Assets:US:MSSB:GOOG        -25 GOOG {1212.51   USD} ; Old GOOG
  Assets:US:MSSB:GOOG         25 GOOG { 605.2850 USD} ; New GOOG
  Assets:US:MSSB:GOOGL        25 GOOG { 607.2250 USD}
```

Ultimately, maybe a plug-in module should be provided to more easily create such stock split transactions, as there is some amount of redundancy involved. We need to figure out the most general way to do this. But the above will work for now.

One problem with this approach is that the *continuity* of the trade lots is lost, that is, the purchase date of each lot has now been reset as a result of the transaction above, and it becomes impossible to automatically figure out the duration of the trade and its associated impact on taxation, i.e. long-term vs. short-term trade. Even without this the profit is still calculated correctly, but it is an annoying detail nonetheless.

One way to handle this is by using the Dated Lots (see the appropriate section of this doc). That way, the original trade date can be preserved on the new lots. This provides accurate timing information in addition to the capital gain/loss based on the price.

Another method for solving this and for easily propagating the lot trade date has been proposed and will be implemented in Beancount later on.

A more important problem with the current implementation is that the meaning of a unit of ADSK before and after the stock split is different. The price graph for this commodity unit will show a radical discontinuity! This is a more general problem that has yet to be addressed in both Beancount and Ledger. The Commodity Definition Changes document has a discussion to address this topic.

Cost Basis Adjustment and Return of Capital

Readjustment in cost basis may occur in managed funds, due to the fund's internal trading activities. This will typically occur in tax-sheltered accounts

where the gain that occurs from such an adjustment has no impact on taxes, and where the cost basis is held at the average cost of all shares in each position.

If we have the specific lot prices being adjusted, it is doable to book these in the same manner as we dealt with stock splits:

```
2014-04-07 * "Cost basis adjustment for XSP"
  Assets:CA:RRSP:XSP           -100 ADSK {21.10 CAD}
  Assets:CA:RRSP:XSP            100 ADSK {23.40 CAD}
  Income:CA:RRSP:Gains      -230.00 CAD
```

However, this is really uncommon. The more common case of this is of an account using the average cost booking method, we don't currently have a way to deal with this. There is an active proposal in place to make this possible.

The cost basis adjustment is commonly found in Return of Capital events. These happen, for example, when funds are returning capital to the shareholders. This can be caused by winding down the operation. From the taxation point of view, these are non-taxable events and affect the cost basis of the equity in the fund. The number of shares might stay the same, but their cost basis needs to be adjusted for potential Gain/Loss calculation at the point of sale in the future.

Dividends

Dividends don't pose a particular problem. They are just income. They can be received as cash:

```
2014-02-01 * "Cash dividends received from mutual fund RBF1005"
  Assets:Investments:Cash            171.02 CAD
  Income:Investments:Dividends
```

Or they can be received as stock itself:

```
2014-02-01 * "Stock dividends received in shares"
  Assets:Investments:RBF1005          7.234 RBF1005 {23.64 CAD}
  Income:Investments:Dividends
```

In the case of dividends received as stock, as for stock purchases, you provide the cost basis at which the dividend was received (this should be available in your statements). If the account is held at average cost, this posting will simply merge with the other legs at the time an average cost booking is needed to be performed.

Average Cost Booking

At the moment, the only way to perform booking at average cost is painful: you would have to use the method outlined in the Stock Split section in order to revalue your inventory. This is impractical, however. There is an active proposal with an associated syntax to fully solve this problem.

Once the proposal is implemented, it will look like this:

```
2014-02-01 * "Selling 5 shares at market price 550 USD"
  Assets:Investments:Stock               -5 GOOG {*}
  Assets:Investments:Cash           2740.05 USD
  Expenses:Commissions                 9.95 USD
  Income:Investments:CapitalGains
```

Any posting with a cost of "*" acting on an inventory will select all the shares of that currency (GOOG), merge them into a single one at the average cost, and then reduce that position at this new average cost.

Future Topics

I'll be handling the following topics later on:

- **Mark-to-Market**: Handling end-of-year mark-to-market for Section 1256 instruments (i.e., futures and options), by re-evaluating the cost basis. This is similar to a cost basis readjustment applied at the end of each year for all of these types instruments.
- **Short Sales**: these require little changes. We just have to allow negative numbers of units held at cost. At the moment we spit a warning when units held at cost go negative in order to detect data entry errors, but it would be easy to extend the Open directive syntax to allow this to occur on specific accounts which can hold short sales, which should just show as negative shares. All the arithmetic should otherwise just work naturally. Interest payments on margins would show up as distinct transactions. Also, when you short the stock, you don't receive dividends for those positions, but rather you have to pay them out. You would an expense account for this, e.g., Expenses:StockLoans:Dividends.
- **Trading Options**: I have no idea how to do this at the moment, but I imagine these could be held like shares of stock, with no distinctions. I don't foresee any difficulty.
- **Currency Trading**: At the moment, I'm not accounting for the positions in my FOREX accounts, just their P/L and interest payments. This poses interesting problems:
  - Positions held in a FOREX account aren't just long or short the way that stocks are: they are actually offsetting two commodities at the same time. For example, a long position in USD/CAD should increase the exposure of USD and decrease the exposure in CAD, it can be seen as holding a long asset of USD and a short asset in CAD, at the same time. While it is possible to hold these positions as if they were distinct instruments (e.g., units of "USDCAD" with disregard for its components) but for large positions, especially if held over long periods of time for hedging purposes, it is important to deal with this

and somehow allow the user to reflect the net currency exposures of multiple currency positions against the rest of their assets and liabilities.

- We also need to deal with the gains generated by the closing of these positions: those generate a gain in the currency of the account, after conversion to this currency. For example, if you hold a currency account denominated in USD, and you go long EUR/JPY, when you close the position you will obtain a gain in EUR, and after conversion of the P/L from the EUR into the equivalent number of USD (via EUR/USD) the USD gain will be deposited in your account. This means that two rates are being used to estimate the current market value of any position: the differential between the current rate and the rate at the time of buying, and the rate of the base currency (e.g., EUR) in the account currency (e.g., USD).

Some of these involve new features in Beancount, but some not. Ideas welcome.

[1] This is a misleading notion, however. In reality, *there is no price*, there exist only markets where you get a "hint" of how much someone else might be willing to exchange your shares for (for different amounts to buy or to sell them, and for some limited number of them, we call this "a market"), but until you've actually completed selling your shares, you don't really know precisely how much you will be able to execute that trade at, only an estimate. If you're not intimately familiar with trading, this should give you pause, and hopefully a big "ah-ha! "moment about how the world works - there really does not exist a price for anything in the world - but in the context of this discussion, let's make abstraction of this and assume that you can buy or sell as many shares as you have instantly on the markets at the middle price, such as Google Finance or Yahoo Finance or your broker would report it. The process of deciding that we would be able to sell the shares at 170$ each is called "marking", that is, under reasonable assumptions, we believe that we would be able to actually sell those shares at that price (the term of art "marking to market" refers to the fact that we use the market price as the best indicator to our knowledge of our capability to realize the trade). For an individual buying and selling small amounts of shares that don't move the market and with an honest broker, this is mostly true in practice.

[2] As an aside, putting regular currencies in an account is just the degenerate case of a thing with an empty label. This is an implementation detail that works great in practice.

Stock Vesting in Beancount

Martin Blais, June 2015

http://furius.ca/beancount/doc/vesting

Introduction

This document explains the vesting of restricted stock units in Beancount, by way of an example. This example may not exactly match your situation, but enough detail is provided that you should be able to adapt it for your own particular differences.

A working example file can be found here to follow along with this text.

Restricted Stock Compensation

Many technology companies offer their employees incentive compensation in the form of "grants" (or "awards") of "restricted stock units" (RSU), which is essentially a promise for the "release" to you of actual shares in the future. The stock is "restricted" in the sense that you cannot access it—you only receive it when it "vests", and this happens based on a schedule. Typically, you are promised a fixed number of shares that vest every quarter or every month over a period of 3 or 4 years. If you leave the company, your remaining unvested shares are lost.

One way you can view these RSUs is as an asset, a receivable that arrives regularly over time. These RSUs are essentially compensation denominated in the currency of the company's shares itself. We want to track the unraveling of these unvested units, and correctly account for their conversion to real stock with a cost basis and including whatever taxes were paid upon vesting.

Tracking Awards

Commodities

First we want to define some commodities. In this example, I work for "Hooli Inc." and will eventually receive shares of that company (valued in US dollars):

```
1990-12-02 commodity HOOL
  name: "Common shares of Hooli Inc."
  quote: USD
```

We will also want to track the amount of unvested shares:

```
2013-01-28 commodity HOOL.UNVEST
  name: "Unvested shares of Hooli from awards."
```

Accounts for Awards

Grants received is income. I use "Income:US:Hooli" as the root for all income accounts from Hooli, but in particular, I define an account for the awards, which contains units of unvested stock:

```
2013-01-28 open Income:US:Hooli:Awards        HOOL.UNVEST
```

When the stock vests, we will need to book the other side of this income somewhere, so we define an expenses account to count how much stock has been vested over a period of time:

```
2014-01-28 open Expenses:Hooli:Vested          HOOL.UNVEST
```

Receiving Awards

When you receive a new award (this may occur every year, for example, some people call this a "stock refresh"), you receive it as income and deposit it into a fresh new account, used to track this particular award:

```
2014-04-02 * "Award S0012345"
  Income:US:Hooli:Awards                 -1680 HOOL.UNVEST
  Assets:US:Hooli:Unvested:S0012345       1680 HOOL.UNVEST

2014-04-02 open Assets:US:Hooli:Unvested:S0012345
```

You may have multiple active awards at the same time. It's nice to have a separate account per award, as it offers a natural way to list their contents and when the award expires, you can close the account—the list of open award accounts gives you the list of outstanding & actively vesting awards. In this example I used the number of the award (#S0012345) as the sub-account name. It's useful to use the number as the statements typically include it.

I like to keep all the awards in a small dedicated section.

Vesting Events

Then I have a different section that contains all the transactions that follow a vesting event.

Accounts

First, when we vest stock, it's a taxable income event. The cash value for the stock needs an Income account:

```
2013-04-04 open Income:US:Hooli:RSU
```

Taxes paid should be on the annual expenses accounts you should have defined somewhere else to account for that year's taxes (this is covered elsewhere in the cookbook):

```
2015-01-01 open Expenses:Taxes:TY2015:US:StateNY
2015-01-01 open Expenses:Taxes:TY2015:US:Federal
2015-01-01 open Expenses:Taxes:TY2015:US:SocSec
2015-01-01 open Expenses:Taxes:TY2015:US:SDI
2015-01-01 open Expenses:Taxes:TY2015:US:Medicare
2015-01-01 open Expenses:Taxes:TY2015:US:CityNYC
```

After paying taxes on the received income, the remaining cash is deposited in a limbo account before getting converted:

```
2013-01-28 open Assets:US:Hooli:RSURefund
```

Also, in another section we should have an account for the brokerage which holds and manages the shares for you:

```
2013-04-04 open Assets:US:Schwab:HOOL
```

Generally you don't have a choice for this broker because the company you work normally makes an arrangement with an external firm in order to administer the restricted stock program. Typical firms doing this type of administration are Morgan Stanley, Salomon Smith Barney, Schwab, even E*Trade. In the example we'll use a Schwab account.

And we also need some sort of checking account to receive cash in lieu of fractional shares. I'll assume a Bank of America account in the example:

```
2001-01-01 open Assets:US:BofA:Checking
```

Vesting

First, the vesting events themselves:

```
2015-05-27 * "Vesting Event - S0012345 - HOOL" #award-S0012345 ^392f97dd62d0
  doc: "2015-02-13.hooli.38745783.pdf"
  Income:US:Hooli:RSU                    -4597.95 USD
  Expenses:Taxes:TY2015:US:Medicare         66.68 USD
  Expenses:Taxes:TY2015:US:Federal        1149.48 USD
  Expenses:Taxes:TY2015:US:CityNYC         195.42 USD
  Expenses:Taxes:TY2015:US:SDI               0.00 USD
  Expenses:Taxes:TY2015:US:StateNY         442.32 USD
  Expenses:Taxes:TY2015:US:SocSec          285.08 USD
  Assets:US:Hooli:RSURefund               2458.97 USD
```

This corresponds line-by-line to a payroll stub that I receive each vesting event for each award. Since all the RSU awards at Hooli vest on the same day of the month, this means that I have clusters of these, one for each award (in the example file I show two awards).

Some observations are in order:

- The value of the Income posting consists of the number of shares vested times the FMV of the stock. In this case, that is 35 shares $131.37/share = $4597.95. I just write the dollar amount because it is provided to me on the pay stub.
- Receiving vested stock is a taxable income event, and Hooli automatically withholds taxes and pays them to the government on its employees' behalf. These are the Expenses:Taxes accounts corresponding to your W-2.

- Finally, I don't directly deposit the converted shares to the brokerage account; this is because the remainder of cash after paying tax expenses is not necessarily a round number of shares. Therefore, I used a limbo account (Assets:US:Hooli:RSURefund) and decouple the receipt from the conversion.

This way, each transaction corresponds exactly to one pay stub. It makes it easier to enter the data.

Also note that I used a unique link (^392f97dd62d0) to group all the transactions for a particular vesting event. You could also use tags if you prefer.

Conversion to Actual Stock

Now we're ready to convert the remaining cash to stock units. This happens in the brokerage firm and you should see the new shares in your brokerage account. The brokerage will typically issue a "stock release report" statement for each vesting event for each award, with the details necessary to make the conversion, namely, the actual number of shares converted from cash and the cost basis (the FMV on the vesting day):

```
2015-05-25 * "Conversion into shares" ^392f97dd62d0
  Assets:US:Schwab:HOOL                    18 HOOL {131.3700 USD}
  Assets:US:Hooli:RSURefund
  Assets:US:Hooli:Unvested:S0012345       -35 HOOL.UNVEST
  Expenses:Hooli:Vested                    35 HOOL.UNVEST
```

The first two postings deposit the shares and subtract the dollar value from the limbo account where the vesting transaction left the remaining cash. You should make sure to use the specific FMV price provided by the statement and not an approximate price, because this has tax consequences later on. Note that you cannot buy fractional shares, so the cost of the rounded amount of shares (18) will leave some remaining cash in the limbo account.

The last two postings deduct from the balance of unvested shares and I "receive" an expenses; that expenses account basically counts how many shares were vested over a particular time period.

Here again you will probably have one of these conversions for each stock grant you have. I enter them separately, so that one statement matches one transaction. This is a good rule to follow.

Refund for Fractions

After all the conversion events have moved cash out of the limbo account, it is left the fractional remainders from all the conversions. In my case, this remainder is refunded by Hooli 3-4 weeks after vesting as a single separate pay stub that includes all the remainders (it even lists each of them separately). I enter this as a transaction as well:

```
2015-06-13 * "HOOLI INC       PAYROLL" ^392f97dd62d0
  doc: "2015-02-13.hooli.38745783.pdf"
  Assets:US:Hooli:RSURefund            -94.31 USD
  Assets:US:Hooli:RSURefund             -2.88 USD ; (For second award in example)
  Assets:US:BofA:Checking               97.19 USD
```

After the fractions have been paid the limbo account should be empty. I verify this claim using a balance assertion:

```
2015-06-14 balance Assets:US:Hooli:RSURefund  0 USD
```

This provides me with some sense that the numbers are right.

Organizing your Input

I like to put all the vesting events together in my input file; this makes them much easier to update and reconcile, especially with multiple awards. For example, with two awards I would have multiple chunks of transactions like this, separated with 4-5 empty lines to delineate them:

```
2015-05-27 * "Vesting Event - S0012345 - HOOL" #award-S0012345 ^392f97dd62d0
  doc: "2015-02-13.hooli.38745783.pdf"
  Income:US:Hooli:RSU                  -4597.95 USD
  Assets:US:Hooli:RSURefund             2458.97 USD
  Expenses:Taxes:TY2015:US:Medicare       66.68 USD
  Expenses:Taxes:TY2015:US:Federal      1149.48 USD
  Expenses:Taxes:TY2015:US:CityNYC       195.42 USD
  Expenses:Taxes:TY2015:US:SDI             0.00 USD
  Expenses:Taxes:TY2015:US:StateNY       442.32 USD
  Expenses:Taxes:TY2015:US:SocSec        285.08 USD

2015-05-27 * "Vesting Event - C123456 - HOOL" #award-C123456 ^392f97dd62d0
  doc: "2015-02-13.hooli.38745783.pdf"
  Income:US:Hooli:RSU                  -1970.55 USD
  Assets:US:Hooli:RSURefund             1053.84 USD
  Expenses:Taxes:TY2015:US:Medicare       28.58 USD
  Expenses:Taxes:TY2015:US:Federal       492.63 USD
  Expenses:Taxes:TY2015:US:CityNYC        83.75 USD
  Expenses:Taxes:TY2015:US:SDI             0.00 USD
  Expenses:Taxes:TY2015:US:StateNY       189.57 USD
  Expenses:Taxes:TY2015:US:SocSec        122.18 USD

2015-05-25 * "Conversion into shares" ^392f97dd62d0
  Assets:US:Schwab:HOOL                      18 HOOL {131.3700 USD}
  Assets:US:Hooli:RSURefund
  Assets:US:Hooli:Unvested:S0012345         -35 HOOL.UNVEST
  Expenses:Hooli:Vested                      35 HOOL.UNVEST
```

```
2015-05-25 * "Conversion into shares" ^392f97dd62d0
  Assets:US:Schwab:HOOL                          9 HOOL {131.3700 USD}
  Assets:US:Hooli:RSURefund
  Assets:US:Hooli:Unvested:C123456             -15 HOOL.UNVEST
  Expenses:Hooli:Vested                         15 HOOL.UNVEST

2015-06-13 * "HOOLI INC       PAYROLL" ^392f97dd62d0
  doc: "2015-02-13.hooli.38745783.pdf"
  Assets:US:Hooli:RSURefund                 -94.30 USD
  Assets:US:Hooli:RSURefund                  -2.88 USD
  Assets:US:BofA:Checking                    97.18 USD

2015-02-14 balance Assets:US:Hooli:RSURefund    0 USD
```

Unvested Shares

Asserting Unvested Balances

Finally, you may occasionally want to assert the number of unvested shares. I like to do this semi-annually, for example. The brokerage company that handles the RSUs for Hooli should be able to list how many unvested shares of each award remain, so it's as simple as looking it up on a website:

```
2015-06-04 balance Assets:US:Hooli:Unvested:S0012345  1645 HOOL.UNVEST
2015-06-04 balance Assets:US:Hooli:Unvested:C123456    705 HOOL.UNVEST
```

Pricing Unvested Shares

You can also put a price on the unvested shares in order to estimate the unvested dollar amount. You should use a fictional currency for this, because we want to avoid a situation where a balance sheet is produced that includes these unvested assets as regular dollars:

```
2015-06-02 price HOOL.UNVEST                132.4300 USD.UNVEST
```

At the time of this writing, the bean-web interface does not convert the units if they are not held at cost, but using the SQL query interface or writing a custom script you should be able to produce those numbers:

```
$ bean-query examples/vesting/vesting.beancount
```

Selling Vested Stock

After each vesting event, the stock is left in your brokerage account. Selling this stock proceeds just as in any other trading transaction (see Trading with Beancount for full details). For example, selling the shares from the example would look something like this:

```
2015-09-10 * "Selling shares"
```

```
Assets:US:Schwab:HOOL        -26 HOOL {131.3700 USD} @ 138.23 USD
Assets:US:Schwab:Cash    3593.98 USD
Income:US:Schwab:Gains
```

Here you can see why it matters that the cost basis you used on the conversion event is the correct one: You will have to pay taxes on the difference (in Income:US:Schwab:Gains). In this example the taxable difference is (138.23 - 131.37) dollars per share.

I like to keep all the brokerage transactions in a separate section of my document, where other transactions related to the brokerage occur, such as fees, dividends and transfers.

Conclusion

This is a simple example that is modeled after how technology companies deal with this type of compensation. It is by no means comprehensive, and some of the details will necessarily vary in your situation. In particular, it does not explain how to deal with options (ISOs). My hope is that there is enough meat in this document to allow you to extrapolate and adapt to your particular situation. If you get stuck, please reach out on the mailing-list.

Sharing Expenses in Beancount

Martin Blais, May 2015

http://furius.ca/beancount/doc/shared

Introduction

This document presents a method for precisely and easily accounting for shared expenses in complicated group situations. For example, traveling with a group of friends where people pay for different things. We show it is possible to deal with expenses to be split evenly among the group as well as group expenses that should be allocated to specific persons.

The method uses the double-entry accounting system. The essence of the method consists in separating the accounting of expenses and their associated payments. This makes it much easier to deal with shared costs, because using this method it doesn't matter who pays for what: we reconcile the precise amounts owed for each individual at the end and automatically compute final correcting transfers for each.

This article is structured around a simple trip with expenses shared equally by two people. However, the same method generalizes to any type of project with incomes and expenses to be shared among multiple participants, and splitting expenses evenly or not.

A Travel Example

Martin (the author) and Caroline (his girlfriend) visited Mexico in March 2015. We visited the island of Cozumel for a SCUBA diving trip for three days and then headed to Tulum for two days to relax and dive in the cenotes.

Our assumptions are:

- **Chaotic payments**. We lead very busy lives… both of us are going to make payments ahead of time and during the trip, without any forethought regarding who will pay for what, though we will carefully record every payment. Each of us just pays for whatever preparation costs as needed to arrange this trip. For example, Caroline selected and booked flights early while I paid for the resort and booked the rental and activities at the local dive shop one week before departure.
- **Use of shared and individual assets.** Both of us are bringing cash, and are going to make payments during the trip from our respective wallets as well as from a shared pool of cash converted to local currency (Mexican pesos, for which I will use the "MXN" symbol), and we will use credit cards as necessary before and during the trip.
- **Multiple currencies.** Some of our expenses will be denominated in US dollars and some in Mexican pesos. For example, flights were paid in US dollars, local meals and the accommodation were paid in pesos, but the local dive shop charged us dollars. Converted amounts will come from both cash and credit cards sources.
- **Individual expenses in shared pool.** While most of the expenses are to be shared equally, some of the expenses will apply to only one of us, and we want to account for those explicitly. For example, Caroline took a SCUBA certification course (PADI Open Water) and will pay for that on her own; similarly, she should not be paying for Martin's expensive boat diving costs. To complicate matters, the dive shop issued us a single joint bill for everything at the end of our stay.

A Note About Sharing

I feel that something should be said about the "sharing" aspect of our expenses, as this topic has come up on previous discussions on the mailing-lists involving sharing examples.

We are nitpicking on purpose. For the purpose of this exercise, we are accounting for every little penny spent in an incredibly detailed manner. The point of this document is specifically to show how a complex set of transactions well accounted for can be efficiently and simply disentangled to a precise accounting of expenses for each participant, regardless of who actually makes the payments. We are not cheapskates.

We will assume that we've decided to split expenses evenly. Our "generosity" to each other is not a topic relevant to this document. We're both well compensated working professionals and you can assume that we've agreed to split the common costs for this trip evenly (50/50).

One of the attributes of the method we show here is that the decision of how we choose to split expenses can be made *separately* from the actual payment of costs. For example, we may decide in the end that I pay for rd of the trip, but that can be decided precisely rather than in an ad-hoc "oh, I think I remember I paid for this" manner. This is especially useful in a larger group of people because when expenses aren't tracked accurately, usually *everyone* is left with a feeling that they paid more than the others. The sum of the *perceived* fractions of expenses paid in a group is always greater than 100%...

Overview of the Method

In this section we present a brief illustrated overview of the method. A set of common Assets accounts that belong to the project, and book all our individual expenses and transfer for the trip as coming from external Income accounts:

During the trip, we use the common Assets to make expenses. Most of the expenses are attributed to both of us (and to be shared eventually), but some of the expenses are intended to be attributed to each of us individually:

After the trip, remaining Assets (like cash we walked home with) gets distributed back to ourselves to zero out the balances of the Assets accounts and we record this through contra postings to Income accounts:

Finally, the list of shared Expenses are split between each other—using a plugin that forks every posting that is intended to be a shared expense—and the final amount is used to make a final transfer between each other so that we've each paid for our respective expenses and we're square:

Note that the final balance of expenses for each participant may differ, and these are due to particular expenses that were attributed separately, or if we decide to split the total unevenly.

How to Track Expenses

In order to write down all expenses for this trip we will open a brand new Beancount input file. Despite the fact that the expenses will come from each person's personal accounts, it is useful to think of the trip as a special project, as if it were a separate entity, e.g., a company that exists only for the duration of the trip. The example file we wrote for our trip can be found here and should help you follow along this text.

Accounts

The set of accounts in the input file should not have to match your personal Beancount file's account names. The accounts we will use include accounts that correspond to Martin and Caroline's personal accounts but with generic names (e.g., Income:Martin:CreditCard instead of Liabilities:US:Chase), and expense accounts may not match any regular expenses in my personal Beancount file—it's not important.

As a convention, any account that pertains specifically to one of the travelers will **include that person's name** in the account name. For example, Caroline's credit card will be named "Income:**Caroline**:CreditCard". This is important, as we will use this later on to split contributions and expenses.

Let's examine the different types of accounts we will need to carry this out.

External Income Accounts

The "project" will receive income in the form of transfers from personal accounts of either traveler. These are accounts we will consider external to the project and so will define them as Income accounts:

```
;; External accounts for Martin.
2015-02-01 open Income:Martin:Cash
2015-02-01 open Income:Martin:Cash:Foreign
2015-02-01 open Income:Martin:Wallet
2015-02-01 open Income:Martin:CreditCard

;; External accounts for Caroline.
2015-02-01 open Income:Caroline:Cash
2015-02-01 open Income:Caroline:Wallet
2015-02-01 open Income:Caroline:MetroCard
2015-02-01 open Income:Caroline:CreditCard
```

Transactions carried out from these accounts must be copied from your personal Beancount file. Obviously, you must be careful to include all the transactions pertaining to the trip. I used a tag to do this in my personal file.

Assets & Liabilities Accounts

There will have a few Asset accounts that will be active and exist for the duration of the trip. These temporary accounts will be zero'ed out at the end of it. One example is a pool of petty cash in local currency:

```
2015-02-01 open Assets:Cash:Pesos
  description: "A shared account to contain our pocket of pesos"
```

We also carried cash in each of our pockets while traveling, so I created two separate accounts for that:

```
2015-02-01 open Assets:Cash:Martin
  description: "Cash for the trip held by Martin"

2015-02-01 open Assets:Cash:Caroline
  description: "Cash for the trip held by Caroline"
```

Note however that despite their individual names, those accounts are considered as part of the project. It was just convenient to separately track balances for the cash we each held during the trip.

Expenses Accounts

We will define various accounts to book our Expenses to. For example, "Expenses:Flights" will contains our costs associated with flight travel. For convenience, and because there are many types of expenses in this file, we chose to leverage the "auto-accounts" plugin and let Beancount automatically open these accounts:

```
plugin "beancount.ops.auto_accounts"
```

The great majority of these accounts are for shared expenses to be split between us. For example, shared SCUBA diving expenses will be booked to "Expenses:Scuba".

However, for expenses that are intended to be covered by one of us only, we simply include the name of the traveler in the account name. For example, Martin's extra costs for boat diving will be booked to the "Expenses:Scuba:Martin" account.

Example Transactions

Let's turn our attention to the different types of transactions present in the file. In this section I will walk you through some representative transactions (the names I give to these is arbitrary).

Contributing Transactions

Contributions to the project's costs are usually done in the form of expenses paid from external accounts. For example, Caroline paying for flights with her credit card looks like this:

```
2015-02-01 * "Flights to Cancun"
   Income:Caroline:CreditCard        -976.00 USD
   Expenses:Flights
```

Martin paying for the taxi to the airport looks like this:

```
2015-02-25 * "Taxi to airport" ^433f66ea0e4e
  Expenses:Transport:Taxi                   62.80 USD
  Income:Martin:CreditCard
```

Bringing Cash

We both brought some cash on us, as we were warned it might be difficult to use credit cards in Mexico. In my personal Beancount file, the cash account is "Assets:Cash" but here it must get booked as an external contribution with my name on it:

```
;; Initial cash on us.
2015-02-24 * "Getting cash for travel"
  Income:Martin:Cash                          -1200 USD
  Assets:Cash:Martin                           1200 USD
```

Caroline's cash is similar:

```
2015-02-24 * "Getting cash for travel"
  Income:Caroline:Cash                         -300 USD
  Assets:Cash:Caroline                          300 USD
```

Once again, note that the Assets:Cash:Martin and Assets:Cash:Caroline accounts are considered a part of the project, in this case it just refers to who is carrying it. (These accounts are cleared at the end, so it does not matter that our names are in them.)

Transfers

Before the trip, I was really busy. It looked like Caroline was going to make most of arrangements and upfront payments, so I made a transfer to her Google Wallet to help her cover for some expenses ahead of time:

```
2015-02-01 * "Transfer Martin -> Caroline on Google Wallet"
  Income:Martin:Wallet                        -1000 USD
  Income:Caroline:Wallet                       1000 USD
```

Cash Conversions

Obtaining local currency was done by changing a small amount of cash at the airport (at a very bad rate):

```
2015-02-25 * "Exchanged cash at XIC at CUN airport"
  Assets:Cash:Caroline                  -100.00 USD @ 12.00 MXN
  Assets:Cash:Pesos                        1200 MXN
```

The "Assets:Cash:Pesos" account tracks our common pool of local currency that we use for various small expenses.

Cash Expenses in US Dollars

Some local expenses will call for US money, which in this example I paid from my cash pocket:

```
2015-03-01 * "Motmot Diving" | "Deposit for cenote diving"
  Expenses:Scuba                              50.00 USD
  Assets:Cash:Martin
```

Cash Expenses in Local Currency

Paying cash using pesos from our shared stash of pesos looks like this:

```
2015-02-25 * "UltraMar Ferry across to Cozumel"
  Expenses:Transport:Bus                 326 MXN
  Assets:Cash:Pesos
```

Sometimes we even had to pay with a mix of US dollars and pesos. In this example, we ran out of pesos, so we have to give them dollars and pesos (all the restaurants and hotels in the beach part of Tulum accept US currency):

```
2015-03-01 * "Hartwood" | "Dinner - ran out of pesos"
  Expenses:Restaurant                   1880 MXN
  Assets:Cash:Pesos                    -1400 MXN
  Assets:Cash:Martin                  -40.00 USD @ 12.00 MXN
```

I used the unfavorable rate the restaurant was offering to accept dollars at (the market rate was 14.5 at the time).

Individual Expenses

Here is an example of booking individual expenses using shared money. In order for us have access to the reef for diving, we had to pay a "marine park" fee of $2.50 per day to the island. This was a short trip where I dove only three days there and Caroline's fee was included in her course except for one day:

```
2015-02-25 * "Marine Park (3 days Martin, 1 day Caroline)"
  Expenses:Scuba:ParkFees:Martin              7.50 USD
  Expenses:Scuba:ParkFees:Caroline            2.50 USD
  Assets:Cash:Martin
```

All that needs to be done is to book these to expense accounts with our names in them, which will get separated out at the end.

Here is a more complicated example: the dive shop at Scuba Club Cozumel charged us a single bill at the end of our stay for all our rental gear and extra dives. All I did here was translate the itemized bill into a single transaction, booking to each other:

```
2015-03-01 * "Scuba Club Cozumel" | "Dive shop bill" ^69b409189b37
  Income:Martin:CreditCard            -381.64 USD
  Expenses:Scuba:Martin                         27 USD ;; Regulator w/ Gauge
  Expenses:Scuba:Caroline                        9 USD ;; Regulator w/ Gauge
  Expenses:Scuba:Martin                         27 USD ;; BCD
  Expenses:Scuba:Caroline                        9 USD ;; BCD
```

```
  Expenses:Scuba:Martin                          6 USD ;; Fins
  Expenses:Scuba:Martin                         24 USD ;; Wetsuit
  Expenses:Scuba:Caroline                        8 USD ;; Wetsuit
  Expenses:Scuba:Caroline                        9 USD ;; Dive computer
  Expenses:Scuba:Martin                          5 USD ;; U/W Light
  Expenses:Scuba:Caroline                       70 USD ;; Dive trip (2 tank)
  Expenses:Scuba:Martin                         45 USD ;; Wreck Dive w/ Lite
  Expenses:Scuba:Martin                         45 USD ;; Afternoon dive
  Expenses:Scuba:Caroline                       45 USD ;; Afternoon dive
  Expenses:Scuba:Martin                      28.64 USD ;; Taxes
  Expenses:Scuba:Caroline                    24.00 USD ;; Taxes
```

Final Balances

Of course you can use balance assertions at any time during the trip. For example, just before leaving at the Cancun airport we knew we wouldn't spend any more Mexican pesos for a while, so I counted what we had left after Caroline decided to splurge the remainder of them into overpriced chocolate sold at the airport shop:

```
2015-03-04 balance Assets:Cash:Pesos                65 MXN
```

Ideally the bookkeeper should want to do this at a quiet moment at the end of every day or couple of days, which makes it easier to triangulate an expense you might have forgotten to enter (we are on vacation after all, in our relaxed state of mind we forget to write down stuff here and there).

Clearing Asset Accounts

At the end of the trip, you should clear the final balances of all Assets and Liabilities accounts by transferring the remaining funds out to the participants, i.e., to the Income accounts. This should leave all the balances of the trip at zero and ensures all the cash put forward for trip expenses has been moved out to travelers.

By the way, it doesn't matter much who keeps this money, because at the end we will end up making a single correcting transfer that should account for the balance required to create an even split. You can transfer it to anyone; the end result will be the same.

Our clearing transaction looked like this:

```
2015-03-06 * "Final transfer to clear internal balances to external ones"
  Assets:Cash:Pesos                               -65 MXN
  Income:Martin:Cash:Foreign                       60 MXN
  Income:Caroline:Cash                              5 MXN
  Assets:Cash:Martin                             -330 USD
  Income:Martin:Cash                              330 USD
  Assets:Cash:Caroline                           -140 USD
```

```
  Income:Caroline:Cash                                140 USD

2015-03-07 balance Assets:Cash:Pesos                    0 MXN
2015-03-07 balance Assets:Cash:Pesos                    0 USD
2015-03-07 balance Assets:Cash:Martin                   0 USD
2015-03-07 balance Assets:Cash:Caroline                 0 USD
```

We had three 20 peso bills, and I kept the bills for future travels. Caroline kept the 5 peso coin (forgot to hand it over as tip). We transferred out the respective cash amounts we had been carrying together during the trip.

How to Take Notes

During this trip I did not carry a laptop—this was vacation after all. I like to disconnect. I did not carry a notebook either. Instead, I just took notes at the end of the day on a few sheets of paper at the hotel. This process took about 5-10 minutes each night, just recalling from memory and writing things down.

These notes look like this:

I made a paper spreadsheet where each line had

- The narration for the transaction (a description that would allow me to select an Expenses account later on)
- Who paid (which Assets or Income account the money came from)
- The amount (either in USD or MXN)

After our trip, I sat down at the computer and typed the corresponding Beancount file. If I had a computer during my vacation I probably would have typed it as we went along. Of course, I had to do a few adjustments here and there because of mistakes.

The bottom line is: if you're organized well, the overhead of doing this is minimal.

Reconciling Expenses

Running bean-web on the trip's file:

```
bean-web beancount/examples/sharing/cozumel2015.beancount
```

You can view the balances in the "All Transactions" view (click on "All Transactions").

The Balance Sheet should show empty balances for Assets accounts:

The balances of the equity accounts should reflect the total amount of currency conversions made during the trip. You can verify this by calculating the amount-weight average rate like this: 7539.00 / 559.88 ~= 13.465 USD/MXN (which is about right).

Reviewing Contributions

The Income Statement should show a summary of all expenses and contributions to the project:

The Income account balances show the total amounts of contributions for each person. Note that in creating the Income accounts, I went through the extra step of creating some specific accounts for each source of payment, like "Caroline's Credit Card", etc.

From this view, we can see that we contributed a total of 4254.28 USD (and were left with 65 MXN in hand) for this trip. The expenses side should match, considering the currency exchanges: 3694.40 + 7474 / 13.465 ~= 4249 USD which is approximately right (the small difference can be explained by the varying currency conversions).

If you want to view the list of contribution payments and the final balance, click on a particular traveler's root account, e.g., "Income:Caroline" (click on "Caroline") which should take you to the Journal for that root account:

This journal includes all the transactions in its sub-accounts. The final value at the bottom should show the total balance of those accounts, and thus, the amount of money Caroline contributed to this trip: 415 USD, and kept 5 MXN (in coin). We can do the same for Martin and find the final balance of 3839.28 USD and kept 60 MXN (in bills).

You can also get the same amounts by using bean-query to achieve the same thing:

```
~/p/.../examples/sharing$ bean-query cozumel2015.beancount
```

Splitting Expenses

The expenses side of the Income Statement shows the breakdown of expenses. Note how some of the expense accounts are explicitly booked to each member separately by their account name, e.g., "Expenses:Scuba:Martin". The other accounts, e.g. "Expenses:Groceries" are intended to be split.

How we're going to carry this out is by adding a plugin that will transform all the transactions to *actually* split the postings which are intended to be shared, that is, postings without any member's name as a component. For example, the following input transaction:

```
2015-02-28 * "Scuba Club Cozumel" | "Room bill"
  Income:Martin:CreditCard         -1380.40 USD
  Expenses:Scuba:Martin              178.50 USD
  Expenses:Accommodation            1201.90 USD
```

Will be transformed by the plugin into one like this:

```
2015-02-28 * "Scuba Club Cozumel" | "Room bill"
```

```
  Income:Martin:CreditCard         -1380.40 USD
  Expenses:Scuba:Martin              178.50 USD
  Expenses:Accommodation:Martin      600.95 USD
  Expenses:Accommodation:Caroline    600.95 USD
```

Note that:

- Only Expenses accounts get split into multiple postings.
- Accounts with the name of a member are assumed by convention to be already attributed to that person. In order to achieve this, the plugin has to be provided with the names of the members.
- All the resulting Income and Expenses accounts include the name of a member.

The plugin is enabled like this:

```
plugin "beancount.plugins.split_expenses" "Martin Caroline"
```

Reloading the web page after uncommenting this line from the input file and visiting the Income Statement page should show a long list of Expenses accounts split by person. Now, in order to generate the total amount of expenses incurred for each person, we have to produce the balance of all Expenses accounts with a member's name in it, accounts like "Expenses:.*Martin".

The web tool does not provide such a filtering capability at the moment[1], but we can use the bean-query tool to produce the total of expenses for each person:

```
beancount> SELECT sum(position) WHERE account ~ '^Expenses:.*Martin'
```

This says "Martin accrued expenses of 2007.43 USD and 3837.0 MXN."

You can manually convert this to a dollar amount:

```
yuzu:~$ dc -e '2007.43 3837.0 13.465 / +p'
2291.43
```

Or you can use the recently introduced "CONVERT" function of bean-query to do this:

```
beancount> SELECT convert(sum(position), 'USD') WHERE account ~ '^Expenses:.*Martin'
```

(The difference between the 2291.43 and 2288.53 amounts can be attributed to the usage of slightly different exchange rates used in the converting transactions.)

Similarly, you can generate the list of payments made by each person, e.g.:

```
beancount> SELECT sum(position) WHERE account ~ '^Income:.*Caroline'
```

Final Transfer

In order to figure out the total amount owed by each member, the process is similar: Simply sum up the balances of all the accounts attributed to that particular member:

```
beancount> SELECT convert(sum(position), 'USD') WHERE account ~ 'Caroline'
```

Notice that this includes the Income and Expenses accounts for that person. It's as if we had two separate ledgers merged into one. (Once again, the small differences can be attributed to differences in exchange rate over time.)

We can now make a final transfer amount in order to account for each of our expenses; we've agreed to round this to 1500 USD:

```
2015-03-06 * "Final transfer from Caroline to Martin to pay for difference"
  Income:Caroline:Wallet                    -1500 USD
  Income:Martin:Wallet                       1500 USD
```

If you uncomment this transaction from the input file (near the end), you will find corrected balances:

```
beancount> SELECT convert(sum(position), 'USD') WHERE account ~ 'Martin'
```

Updating your Personal Ledger

So great! Now we've reconciled each other for the trip. But you still need to reflect these expenses in your personal ledger (if you have one). In this section, I will explain how you should reflect these transactions in that file.

Account Names

First, let us take note that the accounts in your personal ledger do not have to match the account names used in the trip's ledger file. I never process those files together. (A good argument for not attempting this is that each trip will include account names from other people and I prefer not to have those leak into my main ledger file.)

Using a Tag

I like to use a tag to report on the entire set of transactions related to the trip. In this example, I used the tag #trip-cozumel2015.

Booking Contributions

Your contributions into the project should be booked to a temporary holding account. I call mine "Assets:Travel:Pending". For example, this transaction from the trip's file:

```
2015-02-25 * "Yellow Transfers" | "SuperShuttle to Playa del Carmen"
  Expenses:Transport:Bus                  656 MXN
```

```
  Income:Martin:CreditCard            -44.12 USD @ 14.86854 MXN
```

will eventually be imported in my personal file and categorized like this:

```
2015-02-25 * "YELLOW TRANSFER MX  CO" #trip-cozumel2015
  Liabilities:US:BofA:CreditCard      -44.12 USD
  Assets:Travel:Pending                44.12 USD
```

You can conceptualize this as contributing money to a pending pool of cash which you will eventually receive expenses from.

The same goes for cash that I brought for the trip:

```
2015-02-24 * "Taking cash with me on the trip"
  Assets:Cash                              -1200.00 USD
  Assets:Travel:Pending                     1200.00 USD
```

Note that absolutely **all** of the transactions related to the trip should be booked to that one account, including the inter-account transfers:

```
2015-03-06 * "Final transfer from Mexico trip" #trip-cozumel2015
  Assets:Google:Wallet                        1500 USD
  Assets:Travel:Pending                       -1500 USD
```

Booking Expenses

After the trip is concluded, we want to convert the balance in the pending account to a list of Expenses. To find the list of expenses for yourself, you can issue a query like this:

```
beancount> SELECT account, sum(position) WHERE account ~ '^Expenses:.*:Martin'
           GROUP BY 1 ORDER BY 1
```

I suppose you could script away this part to remove the member's name from the account names and have the script spit output already formatted as a nicely formatted Transaction, but because the account names will not match the personal ledger file's 1-to-1, you will have to perform a manual conversion anyway, so I did not bother automating this. Furthermore, you only have a single one of these to make after your trip, so it's not worth spending too much time making this part easier[2].

Here's what the final transaction looks like; I have a section in my input file with this:

```
2015-02-25 event "location" "Cozumel, Mexico"    ;; (1)
pushtag #trip-mexico-cozumel-2015

… other transactions related to the trip…

2015-03-01 event "location" "Tulum, Mexico"      ;; (1)
```

```
2015-03-07 * "Final reconciliation - Booking pending expenses for trip"
  Expenses:Travel:Accommodation         735.45 USD
  Expenses:Food:Alcohol                  483.0 MXN
  Expenses:Sports:Velo                    69.5 MXN
  Expenses:Transportation:Flights       488.00 USD
  Expenses:Food:Grocery                  197.0 MXN
  Expenses:Fun:Museum                     64.0 MXN
  Expenses:Food:Restaurant               22.28 USD
  Expenses:Food:Restaurant              1795.5 MXN
  Expenses:Scuba:Dives                  506.14 USD
  Expenses:Scuba:Fees                     7.50 USD
  Expenses:Scuba:Tips                    225.0 MXN
  Expenses:Scuba:Tips                   189.16 USD
  Expenses:Transportation:Bus            709.0 MXN
  Expenses:Transportation:Taxi           53.90 USD
  Expenses:Transportation:Taxi           294.0 MXN
  Expenses:Transportation:Train           5.00 USD
  Assets:Cash:Foreign                    60.0 MXN                ;; (2)
  Assets:Cash                           330.00 USD               ;; (3)
  Assets:Travel:Pending               -2337.43 USD               ;; (4)
  Assets:Travel:Pending                -288.67 USD @@ 3897.0 MXN  ;; (5)

2015-03-07 * "Writing off difference as gift to Caroline"     ;; (6)
  Assets:Travel:Pending                     -43.18 USD
  Expenses:Gifts

2015-03-14 balance Assets:Travel:Pending     0 USD   ;; (7)
2015-03-14 balance Assets:Travel:Pending     0 MXN

poptag #trip-mexico-cozumel-2015
```

Observations:

1. I used "event" directives in order to track my location. I'm doing this because I will eventually need it for immigration purposes (and just for fun, to track the days I spend in places).
2. I moved the extra cash I kept to my "foreign cash pocket" account: Assets:Cash:Foreign
3. I "moved" back the cash I had with me after I returned from the trip.
4. This leg removes the USD expenses from the Pending account.
5. The leg removes the MXN expenses from the Pending account. I calculated its amount manually (3897 / 13.5 ~= 288.67 USD) and use the full amount as the exchange rate.
6. I want to completely zero out the Pending account after the trip, and so I

write off the excess amount we agreed not to pay (the difference between the imbalance and 1500.00 USD).

7. Finally, I assert that the Pending account is empty of USD and of MXN.

I found the missing amounts by running bean-check or using bean-doctor context on an incomplete transaction from Emacs.

Generating Reports

If you want to automate the generation of reports for each of the participants in a trip, there is a script that generates text (and eventually CSV) reports for the queries mentioned in the previous sections. You can use this script to provide expenses status after or even during the trip or project, for each of the participants.

The script lives in the split_expenses plugin itself, and you invoke it like this:

```
python3 -m beancount.plugins.split_expenses <beancount-filename> --text=<dir>
```

For each person, it will generate the following reports:

- A detailed journal of expenses
- A detailed journal of contributions
- A breakdown of expenses for each category (account type)

Finally, it will also generate a final balance for each person, which you can use to send final reconciling transfers to each other. Try it now on one of the example files provided with Beancount.

Other Examples

There is another example file that shows how to share expenses between three participants in duxbury2015.beancount. Look to more example files to be introduced over time.

Conclusion

There is more than just one way to carry out the task we describe here. However, the method we present extends nicely to a larger group of participants, allows us to account for situations where particular expenses are incurred for one individual as part of a group, and finally, allows for a non-even split between the participants. This is pretty general.

It's also a short step to accounting for an ongoing project with a longer term. The ideas presented in this document provide a nice use case for the usage of the double-entry method in a simple setting. I hope that working through this use case will help people develop the intuitions necessary in using the double-entry method.

[1] As our SQL-like query language matures, bean-web will eventually allow the user to create views from a set of transactions filtered from an expression. This will be implemented eventually.

[2] Note that we could implement a special "beancount" supported format to the bean-query tool in order to write out balances in Transaction form. I'm not sure how useful that would be but it's an idea worth considering for the future.

How We Share Expenses

This document explains how I share expenses with my wife. This is a bit involved and I've developed a good working system, but it's not so simple for most people to do that, so I figured I would take the time to describe it to help others with designing similar processes for themselves.

Context

We're both working professionals and have decided to share all expenses 2:1, where I pay 2 parts. Roughly this is what we shoot for, this is not a hard rule, but we track it as if it were rigid, and accept it as a good base approximation. We have two types of shared expenses:

- **Shared.** Common expenses between us, e.g., rent, a dinner out with friends, etc.
- **Kyle.** Expenses for our child, e.g., daycare, diapers, nanny, baby food.

These get handled very differently, because we book our child's expenses as if it were a separate project on its own. (If you prefer pictures, there's a diagram at the end of this document that provides an overview of the system.)

Shared Expenses

For our shared expenses, I maintain an account for her on my personal ledger file. This is simple and I'm not too interested in maintaining a separate ledger for the totality of our common shared expenses. It's sufficient for me to just keep track of her balance on that one account. You can imagine that account like a credit card (I'm the credit provider) that she pays off with transfers and also by making her own shared expenses.

I just declare a Assets:US:Share:Carolyn account on my personal ledger (blais.beancount).

My Shared Expenses

Whenever I incur an expense that is for both of us, I book it normally but I will tag it with #carolyn:

```
2018-12-23 * "WHISK" "Water refill" #carolyn
  Liabilities:US:Amex:BlueCash  -32.66 USD
  Expenses:Food:Grocery
```

This gets automatically converted to:

```
2018-12-23 * "WHISK" "Water refill" #carolyn
  Liabilities:US:Amex:BlueCash  -32.66 USD
  Expenses:Food:Grocery          19.60 USD
  Assets:US:Share:Carolyn       13.06 USD
    share: TRUE
```

This is done by the beancount.plugins.divert_expenses plugin. In this example, 40% of 32.66 (13.06) gets rerouted to her account. Note that this is an asset account for me, because she owes this.

Her Shared Expenses

We also have to keep track of the money she spends on her own for shared expenses. Since she's not a Beancount user, I've set up a Google Sheets doc in which she can add rows to a particular sheet. This sheet has fields: Date, Description, Account, Amount. I try to keep it simple.

Then, I built an extract_sheets.py script that can pull down this data automatically and it writes it to a dedicated file for this, overwriting the entire contents each time. The contents of this ledger (carolyn.beancount) look like this:

```
pushtag #carolyn
...
2017-05-21 * "Amazon" "Cotton Mattress Protector in White"
  Expenses:Home:Furniture    199.99 USD
  Assets:US:Share:Carolyn  -199.99 USD
...
poptag #carolyn
```

All of those transactions are tagged as #carolyn from the pushtag/poptag directive. They get translated by the plugin to reduce the amount by the portion I'm supposed to pay:

```
2017-05-21 * "Amazon" "Cotton Mattress Protector in White" #carolyn
  Expenses:Home:Furniture    119.99 USD
  Assets:US:Share:Carolyn  -119.99 USD
    share: TRUE
```

These postings typically reduce her asset account by that much, and thus remove the portion she paid for me on these shared expenses.

I generate the file with a single command like this:

```
extract_sheets.py --tag='carolyn' --sheet='Shared Expenses' '<id>' 'Assets:US:Share:Carolyn'
```

In my personal ledger, I include this file to merge those expenses with my own directives flow. I also define a query to generate a statement for her account. In blais.beancount:

```
include "carolyn.beancount"

2020-01-01 query "carolyn" "
 select date, description, position, balance
 from open on 2017-01-01
 where account ~ 'Assets:US:Share:Carolyn'
"
```

Reviewing & Statement

In order to produce a statement for her to review (and spot the occasional mistake in data entry), I simply produce a journal of that account to a CSV file and upload that to another sheet in the same Google Sheets doc she inputs her expenses:

```
bean-query -f csv -o carolyn.csv --numberify $L run carolyn
upload-to-sheets -v --docid="<id>" carolyn.csv:"Shared Account (Read-Only)"
```

This makes it easy for her to eyeball all the amounts posted to her balance with me and point out errors if they occur (some always appear). Moreover, all the information is in one place—it's important to keep it simple. Finally, we can use the collaborative features of Sheets to communicate, e.g. comments, highlighting text, etc.

Note that this system has the benefit of correctly accruing my expenses, by reducing my portion on categories for the stuff I pay and by including her portion on categories for the stuff she pays for.

Reconciling our Shared Expenses

Finally, in order to reconcile this account, my wife (or I, but usually she's behind) just makes a bank transfer to my account, which I book to reduce her running account balance:

```
2019-01-30 * "HERBANK EXT TRNSFR; DIRECTDEP"
  Assets:US:MyBank:Checking  3000 USD
  Assets:US:Share:Carolyn
```

Typically she's do this every month or two. She'll be fiddling on her laptop and ask casually "Hey, what's my balance I can do a transfer now?" It's all fine to relax about the particulars since the system is keeping track of everything precisely, so she can send some approximate amount, it doesn't matter, it'll post to her account.

Child Expenses

I designed a very different system to track our child's expenses. For Kyle, I'm definitely interested in tracking the total cash flows and expenses related to him, regardless of who paid for them. It's interesting to be able to ask (our ledger) a question like: "How much did his schooling cost?", for example, or "How much did we pay in diapers, in total?". Furthermore, we tend to pay for different things for Kyle, e.g. I deal with the daycare expenses (I'm the accounting nerd after all, so this shouldn't be surprising), and his mother tends to buy all the clothing and prepare his food. To have a global picture of all costs related to him, we need to account for these things correctly.

One interesting detail is that it would be difficult to do this with the previously described method, because I'd have to have a mirror of all expense accounts I'd use for him. This would make my personal ledger really ugly. For example, I want to book diapers to Expenses:Pharmacy, but I also have my own personal pharmacy expenses. So in theory, to do that I'd like to have separate accounts for him and me, e.g., Expenses:Kyle:Pharmacy and Expenses:Pharmacy. This would have to be done for all the accounts we use for him. I don't do that.

My Child Expenses on my Personal Ledger

Instead of doing that, what I want is for my personal ledger to book all the expenses I make for him to a single category: Expenses:Kyle, and to track all the detail in a shared ledger. But I still have to book all the expenses to some category, and these appear on my personal ledger, there's no option (I won't maintain a separate credit card to pay for his expenses, that would be overkill, so I have to find a way).

I accomplish this by booking the expenses to my own expenses account, as usual, but tagging the transaction with #kyle:

```
2019-02-01 * "AMAZON.COM" "MERCHANDISE - Diapers size 4 for Kyle" #kyle
  Liabilities:US:Amex:BlueCash  -49.99 USD
  Expenses:Pharmacy
```

And I have a different plugin that *automatically* makes the conversion of those transactions to:

```
2019-02-01 * "AMAZON.COM" "MERCHANDISE - Diapers size 4 for Kyle" #kyle
 Liabilities:US:Amex:BlueCash  -49.99 USD
 Expenses:Kyle                  49.99 USD
   diverted_account: "Expenses:Pharmacy"
```

So from my personal side, all those expenses get booked to my "Kyle project" account. This is accomplished by the divert_expenses plugin, with this configuration:

```
plugin "beancount.plugins.divert_expenses" "{
  'tag': 'kyle',
```

```
  'account': 'Expenses:Kyle'
}"
```

The "diverted_account" metadata is used to keep track of the original account, and this is used later by another script that generates a ledger file decided to my expenses for him (more below).

My Child Expenses in Kyle's own Ledger

Now, because we're considering Kyle's expenses a project of his own, I have to maintain a set of ledgers for him. I automatically pull the transactions I described in the previous section from my personal ledger and automatically convert them to a file dedicated to his dad (me). This is done by calling the extract_tagged script:

```
extract_tagged.py blais.beancount '#kyle' 'Income:Dad' --translate "Expenses:Kyle:Mom=Income
```

The matching transaction from the previous section would look like this in it:

```
2019-02-01 * "AMAZON.COM" "MERCHANDISE - Diapers size 4 for Kyle" #kyle
  Income:Dad      -49.99 USD
  Expenses:Pharmacy   49.99 USD
```

As you can see, the script was able to reconstruct the original account name in Kyle's ledger by using the metadata saved by the divert_expenses plugin in the previous section. It also books the source of the payment to a single account showing it came from his father (Income:Dad). There's no need for me to keep track of which payment method I used on Kyle's side (e.g. by credit card), that's not important to him.

This ledger contains the sum total of all expenses I've made for him to date. The file gets entirely overwritten each time I run this (this is a purely generated output, no hand editing is ever done here, if I change anything I change it in my personal ledger file).

Her Child Expenses

In order to keep track of her expenses for Kyle, we use the same method (and programs) as we use for our shared accounts in order to pull a set of "Carolyn's expenses for Kyle" from another sheet in the same Google Sheets doc:

```
extract_sheets.py --sheet='Kyle Expenses (Regular)' '<id>' 'Income:Mom' > mom.beancount
```

This pulls in transactions that look like this:

```
2018-09-23 * "SPROUT SAN FRANCISCO" "Clothing for Kyle 9-12 months"
  Expenses:Clothing   118.30 USD
  Income:Mom      -118.30 USD
```

The expenses accounts are pulled from the sheet—sometimes I have to go fix that by hand a bit, as they may not track precisely those from our ledger—and

the income shows that the contribution was made by his mother.

Putting it All Together

Finally, we need to put together all these files. I created a top-level kyle.beancount file that simple declares all of his account categories and includes his mom and dad files. We have three files:

```
dad.beancount
mom.beancount
kyle.beancount -> includes transactions from dad.beancount and mom.beancount
```

I can then run bean-web or bean-query on kyle.beancount. There are two things which are interesting to see on that ledger:

1. The distribution of Kyle's expenses, in other words, what's it costing us to raise a child (regardless of who pays).
2. The difference between our contributions.

In order to reconcile (2), we basically compare the balances of the Income:Mom and Income:Dad accounts. This can be done "by hand", visually (using a calculator), but since Beancount's API make it so easy to pull any number from a ledger, I wrote a simple script which does that, computes the total amount of Income contributions, breaks it down to the expected numbers based on our chosen breakdown, compares it to each parent's actual contribution, and simply prints out the difference:

```
$ reconcile_shared.py kyle.beancount
```

After reconciling, the final number should be zero.

Reconciling the Child Ledger

In order to account for the difference and make the contributions to Kyle's upbringing in line with our mutual arrangement of 60%/40%, in the previous section, my wife would need to transfer $301 to me. Of course, we don't *actually* transfer anything in practice, I just add a virtual transfer to book the difference to her shared account on my ledger. To do this, all I have to do is insert a transaction in blais.beancount that looks like this:

```
2019-02-09 * "Transfer to reconcile Kyle's expenses" #kyle
  Assets:US:Share:Carolyn         301.00 USD
  Expenses:Kyle:Mom
```

When this gets pulled into dad.beancount (as explained previously), it looks like this:

```
2019-02-09 * "Transfer to reconcile Kyle's expenses" #kyle
  Income:Mom             -301.00 USD
  Income:Dad              301.00 USD
```

After that, going back to the Kyle ledger, pulling in all the transactions again (this is done by using a Makefile) would show an updated balance of 0:

```
$ reconcile_shared.py kyle.beancount
```

Summary of the System

Here's a diagram that puts in perspective the entire system together:

I ("Dad") use Beancount via Emacs, exclusively. Carolyn ("Mom") only interacts with a single Google Sheets doc with three sheets in it. I pull in Carolyn's shared expenses from a sheet that she fills in to a ledger which gets included in my personal ledger. I also pull in her expenses for Kyle in a similar document, and from my personal ledger I generate my own expenses for Kyle. Both of these documents are merged in a top-level ledger dedicated to Kyle's expenses.

I'm not going to claim it's simple and that it's always up-to-date. I tend to update all of these things once/quarter and fix a few input errors each time we review it. I automate much of it via Makefiles, but frankly I don't update it frequently enough to just remember where all the moving pieces are, so every time, there's a bit of scrambling and reading my makefiles and figuring out what's what (in fact, I was motivated to write this to cement my understanding solidly for the future). Amazingly enough, it hasn't broken at all and I find all the pieces every time and make it work.

And we have enough loose change between the two of us that it's never that urgent to reconcile every week. My wife tends to pay more shared expenses and I tend to pay more Kyle's expenses, so I often end up doing a transfer from one to the other to even things out and it's relatively close—the shortfall in Kyle expenses makes up for my shortfall on the shared expenses.

Conclusion

I hope this was useful to some of you trying to solve problems using the double-entry bookkeeping method. Please write questions and comments on the Beancount mailing-list, or as comments on this doc.

Beancount Scripting & Plugins

Martin Blais, July 2014

http://furius.ca/beancount/doc/scripting

Introduction

This document provides examples and guidelines on how to write scripts that use the contents of your ledger. It also provides information on how to write your own "plugins," which are just Python functions that you can configure to transform your transactions or synthesize ones programmatically. These are the main two methods for extending Beancount's features and for writing your own custom reports. You simply use Python to do this.

Load Pipeline

You need to know a little bit about how Beancount processes its input files. Internally, the single point of entry to load an input file is the beancount.loader.load_file() function, which accepts an input file and carries out a list of transformation steps, as in this diagram:

The stages of loading are as follows:

1. **Parser.** Run the input file through the parser. The output of this stage is

    1. entries: A list of tuples (defined in beancount.core.data) corresponding to each directive exactly as it appeared in the file, and sorted by date and line number. Moreover, Transaction directives that occur on the same date as other directives are always guaranteed to be sorted after them. This prepares the entries for processing. This list of entries will get transformed and refined by the various subsequent stages.

    2. options_map: A Python dict of the option values from the input file. See beancount.parser.options for details. Once created, this will never be modified thereafter.

    3. errors: A list of error objects, if any occurred. At every stage, new errors generated are collected.

2. **Process plugins.** For each plugin, load the plugin module and call its functions with the list of entries and the options_map from the previous stage, replacing the current list by the ones returned by the plugin. This effectively allows the plugin to filter the entries.
   The list of plugins to run is composed of a set of default plugin modules

that implement some of the built-in features of Beancount, followed by the list provided by the user from the "plugin" options in the input file.

3. **Validation.** Run the resulting entries through a validation stage, to ensure that directives synthesized or modified by the plugins conform to some invariants that the codebase depends on. This mainly exists to generate errors.

The list of entries generated by this pipeline are of the various types defined in beancount.core.data, and in a typical input file, most of them will be of type *Transaction*. Beancount's own filtering and reporting programs directly process those, and so can you too. These entries are dumb read-only objects (Python namedtuples) and have no methods that modify their contents explicitly. All processing within Beancount is performed functional-style by processing lists of entries that are assumed immutable[1].

The list of user plugins to run is part of the load stage because that allows programs that monitor the file for changes to reload it and reapply the same list of plugins. It also allows the author of the input file to selectively enable various optional features that way.

Writing Plug-ins

As you saw in the previous section, loading a Beancount file essentially produces a list of directives. Many syntax extensions can be carried out by transforming the list of directives into a new list in the plug-ins processing stage. Here are some examples of transformations that you might want to carry out on some of the directives:

- Add some postings automatically
- Link some transactions with a common tag
- Synthesize new transactions
- Remove or replace some sets of transactions
- Modify the various fields

There is no limit to what you can do, as long as the entries your plugin produces fulfill certain constraints (all postings balance, all data types are as expected).

A plugin is added to the input file via the option syntax, for example, like this:

```
plugin "accounting.wash_sales"
```

With this directive, the loader will attempt to import the accounting.wash_sales Python module (the code must be Python-3.3 or above), look for a special ___plugins___ attribute which should be a sequence of functions to run, and then run those functions.

For running the plugins, see the Executing Plugins section below.

As an example, you would place code like this in a "accounting/wash_sales.py" file:

```
__plugins__ = ['wash_sales']

def wash_sales(entries, options_map):
    errors = []
    for entry in entries:
        print(type(entry))
    return entries, errors
```

This is a minimal example which does not modify the entries and prints them on the console. In practice, to do something useful, you would modify some of the entries in the list and output them.

You then invoke the usual tools provided by Beancount on your input file. The various filters and reports will then operate on the list of entries output by your plugin. Refer to the source code in beancount.core for details and examples of how to manipulate entries.

Plugin Configuration

Some plugins will require configuration. In order to provide a plugin some data specific to your file, you can provide a configuration string:

```
plugin "accounting.wash_sales" "days=31"
```

The plugin function will then receive an extra parameter, the configuration string. It is up to the plugin itself to define how it gets interpreted.

Writing Scripts

If you need to produce some custom analysis or visualization that cannot be achieved using the built-in filtering and reporting capabilities, you can just write a script that loads the directives explicitly. This gives you control over the flow of the program and you can do anything you want.

Loading from File

You can simply call the beancount.loader.load_file() loader function yourself. Here is an example minimal script:

```
#!/usr/bin/env python3
from beancount import loader

filename = "/path/to/my/input.beancount"
entries, errors, options = loader.load_file(filename)
…
```

At this point you can process the entries as you like, print them out, generate HTML, call out to Python libraries, etc. (I recommend that you use best programming practices and use docstrings on your script and a main function; the script above is meant to be minimal). Once again, refer to the source code in beancount.core for details and examples of how to manipulate entries.

Loading from String

You can also parse a string directly. Use beancount.loader.load_string():

```
#!/usr/bin/env python3
from beancount import loader

entries, errors, options = loader.load_string("""

   2014-02-02 open Assets:TestAccount   USD
   …

""")
```

The stdlib textwrap.dedent function comes in handy if you want to indent the Beancount directives and have it automatically remove indentation. For a source of many examples, see the various tests in the Beancount source code.

Printing Errors

By default, the loader will not print any errors upon loading; we prefer loading not to have any side-effect by default. You can provide an optional argument to print errors, which is the function to call to write error strings:

```
#!/usr/bin/env python3
import sys
from beancount import loader

filename = "/path/to/my/input.beancount"
entries, errors, options = loader.load_file(filename,
                                            log_errors=sys.stderr)
…
```

Or if you prefer to do it yourself explicitly, you can call the beancount.parser.printer.print_errors() helper function:

```
#!/usr/bin/env python3
from beancount import loader
from beancount.parser import printer

filename = "/path/to/my/input.beancount"
entries, errors, options = loader.load_file(filename)
printer.print_errors(errors)
```

…

Printing Entries & Round-Tripping

Printing namedtuple entries directly will output some readable though relatively poorly formatted output. It's best to use the beancount.parser.printer.print_entry() utility function to print out an entry in a readable way:

```
#!/usr/bin/env python3
from beancount import loader
from beancount.parser import printer

filename = "/path/to/my/input.beancount"
entries, errors, options = loader.load_file(filename)
for entry in entries:
    printer.print_entry(entry)
```

In particular, Beancount offers the guarantee that the output of the printer should always be parseable and should result in the same data structure when read back in. (It should be considered a bug if that is not the case.)

See the beancount.parser.printer module source code for more utility functions.

Executing Plugins

All that is required for the plug-in module to be found, is that it must be present in your PYTHONPATH environment variable (you need to make sure that the relevant ___init___.py files exist for import). It can live in your own code: you don't have to modify Beancount itself.

There is also an option, which can be added to your beancount file:

```
option "insert_pythonpath" "True"
```

This will add the folder which contains the beancount file to the PYTHONPATH. The result is that you can place the plugins along the beancount file and have them execute when you use this file.

Here is a brief example, using the wash_sales.py plugin we wrote above. Your beancount file would include the following lines:

```
option "insert_pythonpath" "True"
```

The Python file wash_sales.py would be stored in the same folder as the .beancount file.

Going Further

To understand how to manipulate entries, you should refer to the source code, and probably learn more about the following modules:

- beancount.core.data
- beancount.core.account
- beancount.core.number
- beancount.core.amount
- beancount.core.position
- beancount.core.inventory

Refer to the Design Doc for more details. Enjoy!

[1] Technically, Python does not prevent the modifications of namedtuple attributes that are themselves mutable such as lists and sets, but in practice, by convention, once an entry is created we never modify it in any way. Avoiding side-effects and using a functional style provides benefits in any language.

Beancount Design Doc

Martin Blais (blais@furius.ca)

http://furius.ca/beancount/doc/design-doc

A guide for developers to understand Beancount's internals. I hope for this document to provide a map of the main objects used in the source code to make it easy to write scripts and plugins on top of Beancount or even extend it.

Introduction

This document describes the principles behind the design of Beancount and a high-level overview of its codebase, data structures, algorithms, implementation and methodology. This is not a user's manual; if you are interested in just using Beancount, see the associated User's Manual and all the other documents available here.

However, if you just want to understand more deeply how Beancount works this document should be very helpful. This should also be of interest to developers. This is a place where I wrote about a lot of the ideas behind Beancount that did not find any other venue. Expect some random thoughts that aren't important for users.

Normally one writes a design document before writing the software. I did not do that. But I figured it would still be worthwhile to spell out some of the ideas that led to this design here. I hope someone finds this useful.

Invariants

Isolation of Inputs

Beancount should accept input *only* from the text file you provide to its tools. In particular, it should not contact any external networked service or open any "global" files, even just a cache of historical prices or otherwise. This isolation is by design. Isolating the input to a single source makes it easier to debug, reproduce and isolate problems when they occur. This is a nice property for the tool.

In addition, fetching and converting external data is very messy. External data formats can be notoriously bad, and they are too numerous to handle all of

them (handling just the most common subset would beg the question of where to include the implementation of new converters). Most importantly, the problems of representing the data vs. that of fetching and converting it segment from each other very well naturally: Beancount provides a core which allows you to ingest all the transactional data and derive reports from it, and its syntax is the hinge that connects it to these external repositories of transactions or prices. It isolates itself from the ugly details of external sources of data in this way.

Order-Independence

Beancount offers a guarantee that the ordering of its *directives* in an input file is irrelevant to the outcome of its computations. You should be able to organize your input text and reorder any declaration as is most convenient for you, without having to worry about how the software will make its calculations. This also makes it trivial to implement inclusions of multiple files: you can just concatenate the files if you want, and you can process them in any order.

Not even directives that declare accounts or commodities are required to appear before these accounts get used in the file. All directives are parsed and then basically *sorted* before any calculation or validation occurs (a minor detail is that the line number is used as a secondary key in the sorting, so that the re-ordering is stable).

Correspondingly, all the non-directive grammar that is in the language is limited to either setting values with global impact ("option", "plugin") or syntax-related conveniences ("pushtag", "poptag").

There is an exception:

- Options may have effects during the processing of the file and should appear at the beginning. At the moment, Beancount does not enforce this, e.g., by performing multiple passes of parsing, but it probably should.
- What's more, options in included files are currently ignored. You should put all your options in the top-level ledger file.

All Transactions Must Balance

All transactions must balance by "weight." There is no exception. Beancount transactions are required to have balanced, period.

In particular, there is no allowance for exceptions such as the "virtual postings" that can be seen in Ledger. The first implementation of Beancount allowed such postings. As a result, I often used them to "resolve" issues that I did not know how to model well otherwise. When I rewrote Beancount, I set out to convert my entire input file to avoid using these, and over time I succeeded in doing this and learned a lot of new things in the process. I have since become convinced that virtual postings are wholly unnecessary, and that make them available only provides a *crutch* upon which a lot of users will latch instead

of learning to model transfers using balanced transactions. I have yet to come across a sufficiently convincing case for them[1].

This provides the property that any subsets of transactions will sum up to zero. This is a nice property to have, it means we can generate balance sheets at any point in time. Even when we eventually support settlement dates or per-posting dates, we will split up transactions in a way that does not break this invariant.

Accounts Have Types

Beancount accounts should be constrained to be of one of five types: Assets, Liabilities, Income, Expenses and Equity. The rationale behind this is to realize that all matters of counting can be characterized by being either permanent (Assets, Liabilities) or transitory (Income, Expenses), and that the great majority of accounts have a usual balance sign: positive (Assets, Expenses) or negative (Liabilities, Income). Given a quantity to accumulate, we can always select one of these four types of labels for it.

Enforcing this makes it possible to implement reports of accumulates values for permanent accounts (balance sheet) and reports of changes of periods of time (income statement). Although it is technically possible to explicitly book postings against Equity accounts the usual purpose of items in that category is to account for past changes on the balance sheet (net/retained income or opening balances).

These concepts of time and sign generalize beyond that of traditional accounting. Not having them raises difficult and unnecessary questions about how to handle calculating these types of reports. We simply require that you label your accounts into this model. I'll admit that this requires a bit of practice and forethought, but the end result is a structure that easily allows us to build commonly expected outputs.

Account Lifetimes & Open Directives

Accounts have lifetimes; an account opens at a particular date and optionally closes at another. This is directed by the Open and Close directives.

All accounts are required to have a corresponding Open directive in the stream. By default, this is enforced by spitting out an error when posting to an account whose Open directive hasn't been seen in the stream. You can automate the generation of these directives by using the "auto_accounts" plugin, but in any case, the stream of entries will always contain the Open directives. This is an assumption that one should be able to rely upon when writing scripts.

(Eventually a similar constraint will be applied for Commodity directives so that the stream always include one before it is used; and they should be auto-generated as well. This is not the case right now [June 2015].)

Supports Dates Only (and No Time)

Beancount does not represent time, only dates. The minimal time interval is one day. This is for simplicity's sake. The situations where time would be necessary to disambiguate account balances exist, but in practice they are rare enough that their presence does not justify the added complexity of handling time values and providing syntax to deal with it.

If you have a use case whereby times are required, you may use metadata to add it in, or more likely, you should probably write custom software for it. This is outside the scope of Beancount by choice.

Metadata is for User Data

By default, Beancount will not make assumptions about metadata fields and values. Metadata is reserved for private usage by the user and for plugins. The Beancount core does not read metadata and change its processing based on it. However, plugins may define special metadata values that affect what they produce.

That being said, there are a few special metadata fields *produced* by the Beancount processing:

- filename: A string, the name of the file the transaction was read from (or the plugin that created it)
- lineno: An integer, the line number from the file where the transaction was read from.
- **tolerances**: A dict of currency to Decimal, the tolerance values used in balancing the transaction.
- **residual** (on Postings): A boolean, true if the posting has been added to automatically absorb rounding error.

There may be a few more produced in the future but in any case, the core should not read any metadata and affect its behavior as a consequence. (I should probably created a central registry or location where all the special values can be found in one place.)

Overview of the Codebase

All source code lives under a "beancount" Python package. It consists of several packages with well-defined roles, the dependencies between which are enforced strictly.

Beancount source code packages and approximate dependencies between them.

At the bottom, we have a beancount.core package which contains the data structures used to represent transactions and other directives in memory. This

contains code for accounts, amounts, lots, positions, and inventories. It also contains commonly used routines for processing streams of directives.

One level up, the beancount.parser package contains code to parse the Beancount language and a printer that can produce the corresponding text given a stream of directives. It should be possible to convert between text and data structure and round-trip this process without loss.

At the root of the main package is a beancount.loader module that coordinates everything that has to be done to load a file in memory. This is the main entry point for anyone writing a script for Beancount, it always starts with loading a stream of entries. This calls the parser, runs some processing code, imports the plugins and runs them in order to produce the final stream of directives which is used to produce reports.

The beancount.plugins package contains implementations of various plugins. Each of these plugin modules are independent from each other. Consult the docstrings in the source code to figure out what each of these does. Some of these are prototypes of ideas.

There is a beancount.ops package which contains some high-level processing code and some of the default code that always runs by default that is implemented as plugins as well (like padding using the Pad directive). This package needs to be shuffled a bit in order to clarify its role.

On top of this, reporting code calls modules from those packages. There are four packages which contain reporting code, corresponding to the Beancount reporting tools:

- beancount.reports: This package contains code specialized to produce different kinds of outputs (invoked by bean-report). In general I'd like to avoid defining custom routines to produce desired outputs and use the SQL query tool to express grouping & aggregation instead, but I think there will always be a need for at least some of these. The reports are defined in a class hierarchy with a common interface and you should be able to extend them. Each report supports some subset of a number of output formats.

- beancount.query: This package contains the implementation of a query language and interactive shell (invoked by bean-query) that allows you to group and aggregate positions using a SQL-like DSL. This essentially implements processing over an in-memory table of Posting objects, with functions defined over Amount, Position and Inventory objects.

- beancount.web: This package contains all the source code for the default web interface (invoked by bean-web). This is a simple Bottle application that serves many of the reports from beancount.report to HTML format, running on your machine locally. The web interface provides simple views and access to your data. (It stands to be improved greatly, it's in no way perfect.)

- beancount.projects: This package contains various custom scripts for particular applications that I've wanted to share and distribute. Wherever possible, I aim to fold these into reports. There are no scripts to invoke these; you should use "python3 -m beancount.projects.<name>" to run them.

There is a beancount.scripts package that contains the "main" programs for all the scripts under the bin directory. Those scripts are simple launchers that import the corresponding file under beancount.scripts. This allows us to keep all the source code under a single directory and that makes it easier to run linters and other code health checkers on the code—it's all in one place.

Finally, there is a beancount.utils package which is there to contain generic reusable random bits of utility code. And there is a relatively unimportant beancount.docs package that contains code used by the author just to produce and maintain this documentation (code that connects to Google Drive).

Enforcing the dependency relationships between those packages is done by a custom script.

Core Data Structures

This section describes the basic data structures that are used as building blocks to represent directives. Where possible I will describe the data structure in conceptual terms.

(Note for Ledger users: This section introduces some terminology for Beancount; look here if you're interested to contrast it with concepts and terminology found in Ledger.)

Number

Numbers are represented using decimal objects, which are perfectly suited for this. The great majority of numbers entered in accounting systems are naturally decimal numbers and binary representations involve representational errors which cause many problems for display and in precision. Rational numbers avoid this problem, but they do not carry the limited amount of precision that the user intended in the input. We must deal with tolerances explicitly.

Therefore, all numbers should use Python's "decimal.Decimal" objects. Conveniently, Python v3.x supports a C implementation of decimal types natively (in its standard library; this used to be an external "cdecimal" package to install but it has been integrated in the C/Python interpreter).

The default constructor for decimal objects does not support some of the input syntax we want to allow, such as commas in the integer part of numbers (e.g., "278**,**401.35 USD") or initialization from an empty string. These are important cases to handle. So I provide a special constructor to accommodate these

inputs: beancount.core.number.D(). This is the only method that should be used to create decimal objects:

```
from beancount.core.number import D
…
number = D("2,402.21")
```

I like to import the "D" symbol by itself (and not use number.D). All the number-related code is located under beancount.core.number.

Some number constants have been defined as well: ZERO, ONE, and HALF. Use those instead of explicitly constructed numbers (such as D("1")) as it makes it easier to grep for such initializations.

Commodity

A **commodity**, or **currency** (I use both names interchangeably in the code and documentation) is a string that represents a kind of "thing" that can be stored in accounts. In the implementation, it is represented as a Python str object (there is no module with currency manipulation code). These strings may only contain capital letters and numbers and some special characters (see the lexer code).

Beancount does not predefine any currency names or categories—all currencies are created equal. Really. It genuinely does not know anything special about dollars or euros or yen or anything else. The only place in the source code where there is a reference to those is in the tests. There is no support for syntax using "$" or other currency symbols; I understand that some users may want this syntax, but in the name of keeping the parser very simple and consistent I choose not to provide that option.

Moreover, Beancount does not make a distinction between commodities which represent "money" and others that do not (such as shares, hours, etc.). These are treated the same throughout the system. It also does not have the concept of a "home" currency[2]; it's a genuinely multi-currency system.

Currencies need *not* be defined explicitly in the input file format; you can just start using them in the file and they will be recognized as such by their unique syntax (the lexer recognizes and emits currency tokens). However, you *may* declare some using a Commodity directive. This is only provided so that a per-commodity entity exists upon which the user can attach some metadata, and some report and plugins are able to find and use that metadata.

Account

An account is basically just the name of a bucket associated with a posting and is represented as a simple string (a Python str object). Accounts are detected and tokenized by the lexer and have names with at least two words separated by a colon (":").

Accounts don't have a corresponding object type; we just refer to them by their unique name string (like filenames in Python). When per-account attributes are needed, we can extract the Open directives from the stream of entries and find the one corresponding to the particular account we're looking for.

Similar to Python's os.path module, there are some routines to manipulate account names in beancount.core.account and the functions are named similarly to those of the os.path module.

The first component of an account's name is limited to one of five types:

- Assets
- Liabilities
- Equity
- Income
- Expenses

The names of the account types as read in the input syntax may be customized with some "option" directives, so that you can change those names to another language, or even just rename "Income" to "Revenue" if you prefer that. The beancount.core.account_types module contains some helpers to deal with these.

Note that the set of account names forms an implicit hierarchy. For example, the names:

```
Assets:US:TDBank:Checking
Assets:US:TDBank:Savings
```

implicitly defines a tree of nodes with parent nodes "Assets", "US", "TDBank" with two leaf nodes "Checking" and "Savings". This implicit tree is never realized during processing, but there is a Python module that allows one to do this easily (see beancount.core.realization) and create linearized renderings of the tree.

Flag

A "flag" is a single-character string that may be associated with Transactions and Postings to indicate whether they are assumed to be correct ("reconciled") or flagged as suspicious. The typical value used on transaction instances is the character "*". On Postings, it is usually left absent (and set to a None).

Amount

An **Amount** is the combination of a number and an associated currency, conceptually:

```
Amount = (Number, Currency)
```

Amount instances are used to represent a quantity of a particular currency (the "units") and the price on a posting.

A class is defined in beancount.core.amount as a simple tuple-like object. Functions exist to perform simple math operations directly on instances of Amount. You can also create Amount instance using amount.from_string(), for example:

```
value = amount.from_string("201.32 USD")
```

Cost

A Cost object represents the cost basis for a particular lot of a commodity. Conceptually, it is

```
Cost = (Number, Currency, Date, Label)
```

The number and currency is that of the cost itself, not of the commodity. For example, if you bought 40 shares of AAPL stock at 56.78 USD, the *Number* is a "56.78" decimal and the *Currency* is "USD." For example:

```
Cost(Decimal("56.78"), "USD", date(2012, 3, 5), "lot15")
```

The *Date* is the acquisition date of the corresponding lot (a datetime.date object). This is automatically attached to the Cost object when a posting augments an inventory—the Transaction's date is automatically attached to the Cost object—or if the input syntax provides an explicit date override.

The *Label* can be any string. It is provided as a convenience for a user to refer to a particular lot or disambiguate similar lots.

On a Cost object, the number, currency and date attributes are always set. If the label is unset, it has a value of "None."

CostSpec

In the input syntax, we allow users to provide as little information as necessary in order to disambiguate between the lots contained in the inventory prior to posting. The data provided filters the set of matching lots to an unambiguous choice, or to a subset from which an automated booking algorithm will apply (e.g., "FIFO").

In addition, we allow the user to provide either the per-unit cost and/or the total-cost. This convenience is useful to let Beancount automatically compute the per-unit cost from a total of proceeds.

```
CostSpec = (Number-per-unit, Number-total, Currency, Date, Label, Merge)
```

Since any of the input elements may be omitted, any of the attributes of a CostSpec may be left to None. If a number is missing and necessary to be filled in, the special value "MISSING" will be set on it.

The *Merge* attribute it used to record a user request to merge all the input lots before applying the transaction and to merge them after. This is the method used for all transactions posted to an account with the "AVERAGE" booking method.

Position

A position represents some units of a particular commodity held at cost. It consists simply in

```
Position = (Units, Cost)
```

Units is an instance of *Amount*, and *Cost* is an instance of *Cost*, or a null value if the commodity is not held at cost. Inventories contain lists of *Position* instances. See its definition in beancount.core.position.

Posting

Each Transaction directive is composed of multiple Postings (I often informally refer to these in the code and on the mailing-list as the "legs" of a transaction). Each of these postings is associated with an account, a position and an optional price and flag:

```
Posting = (Account, Units, Cost-or-CostSpec, Price, Flag, Metadata)
```

As you can see, a *Posting* embeds its *Position* instance[3]. The *Units* is an *Amount*, and the 'cost' attribute refers to either a *Cost* or a *CostSpec* instance (the parser outputs *Posting* instances with an *CostSpec* attribute which is resolved to a *Cost* instance by the booking process; see How Inventories Work for details).

The *Price* is an instance of *Amount* or a null value. It is used to declare a currency conversion for balancing the transaction, or the current price of a position held at cost. It is the Amount that appears next to a "@" in the input.

Flags on postings are relatively rare; users will normally find it sufficient to flag an entire transaction instead of a specific posting. The flag is usually left to None; if set, it is a single-character string.

The Posting type is defined in beancount.core.data, along with all the directive types.

Inventory

An Inventory is a container for an account's balance in various lots of commodities. It is essentially a list of *Position* instances with suitable operations defined on it. Conceptually you may think of it as a mapping with unique keys:

```
Inventory = [Position1, Position2, Position3, … , PositionN]
```

Generally, the combination of a position's (*Units.Currency, Cost)* is kept unique in the list, like the key of a mapping. Positions for equal values of currency and cost are merged together by summing up their *Units.Number* and keeping a single position for it. And simple positions are mixed in the list with positions held at cost.

The *Inventory* is one of the most important and oft-used object in Beancount's implementation, because it is used to sum the balance of one or more accounts over time. It is also the place where the inventory reduction algorithms get applied to, and traces of that mechanism can be found there. The "How Inventories Work" document provides the full detail of that process.

For testing, you can create initialized instances of Inventory using inventory.from_string(). All the inventory code is written in beancount.core.inventory.

About Tuples & Mutability

Despite the program being written in a language which does not make mutability "difficult by default", I designed the software to avoid mutability in most places. Python provides a "collections.namedtuple" factory that makes up new record types whose attributes cannot be overridden. Well… this is only partially true: mutable attributes of immutable tuples can be modified. Python does not provide very strong mechanisms to enforce this property.

Regardless, functional programming is not so much an attribute of the language used to implement our programs than of the guarantees we build into its structure. A language that supports strong guarantees helps to enforce this. But if, even by just using a set of conventions, we manage to *mostly* avoid mutability, we have a *mostly* functional program that avoids *most* of the pitfalls that occur from unpredictable mutations and our code is that much easier to maintain. Programs with no particular regard for where mutation occurs are most difficult to maintain. By avoiding most of the mutation in a functional approach, we avoid most of those problems.

- Most of the data structures used in Beancount are namedtuples, which make the modification of their attributes inconvenient on purpose.
- Most of the code will attempt to avoid mutation, except for local state (within a function) or in a narrow scope that we can easily wrap our head around. Where mutation is possible, by convention I try to avoid it by default. When we do mutation, I try to document the effects.
- I avoid the creation of classes which mutate their internal state, except for a few cases (e.g. the web application). I prefer functions to objects and where I define classes I mostly avoid inheritance.

These properties are especially true for all the small objects: Amount, Lot, Position, Posting objects, and all the directives types from beancount.core.data.

On the other hand, the Inventory object is nearly always used as an accumulator and *does* allow the modification of its internal state (it would require a special, persistent data structure to avoid this). You have to be careful how you share access to Inventory objects… and modify them, if you ever do.

Finally, the loader produces lists of directives which are all simple namedtuple objects. These lists form the main application state. I've avoided placing these inside some special container and instead pass them around explicitly, on purpose. Instead of having some sort of big "application" object, I've trimmed down all the fat and all your state is represented in two things: A dated and sorted list of directives which can be the subject of stream processing and a list of constant read-only option values. I think this is simpler.

I credit my ability to make wide-ranging changes to a mid-size Python codebase to the adoption of these principles. I would love to have ***types*** *in order to safeguard against another class of potential errors, and I plan to experiment with Python 3.5's upcoming typing module.*

Summary

The following diagram explains how these objects relate to each other, starting from a Posting.

For example, to access the number of units of a postings you could use

```
posting.units.number
```

For the cost currency:

```
posting.cost.currency
```

You can print out the tuples in Python to figure out their structure.

Previous Design

For the sake of preservation, if you go back in time in the repository, the structure of postings was deeper and more complex. The new design reflects a flatter and simpler version of it. Here is what the old design used to look like:

Directives

The main production from loading a Beancount input file is a list of **directives**. I also call these **entries** interchangeably in the codebase and in the documents. There are directives of various types:

- Transaction
- Balance & Pad
- Open & Close
- Commodity

- Note
- Event
- Price
- Document

In a typical Beancount input file, the great majority of entries will be Transactions and some Balance assertions. There will also be Open & perhaps some Close entries. Everything else is optional.

Since these combined with a map of option values form the entire application state, you should be able to feed those entries to functions that will produce reports. The system is built around this idea of processing a stream of directives to extract all contents and produce reports, which are essentially different flavors of filtering and aggregations of values attached to this stream of directives.

Common Properties

Some properties are in common to *all* the directives:

- **Date.** A datetime.date object. This is useful and consistent. For a Transaction, that's the date at which it occurs. For an Open or Close directive, that's the date at which the account was opened or closed. For a Pad directive, that's the date at which to insert a transaction. For a Note or Document, it is the date at which to insert the note in the stream of postings of that account. For an Event, it is the date at which it occurs. For a Commodity directive which essentially provides a per-commodity hanging point for commodity-related metadata, the date isn't as meaningful; I choose to date those on the day the commodity was created.
- **Meta**. All the directives have metadata attribute (a Python dict object). The purpose of metadata is to allow the user to hang any kind of ancillary data they want and then use this in scripts or queries. Posting instances also have a metadata attribute.

Transactions

The Transaction directive is the subject of Beancount and is by far the most common directive found in input files and is worth of special attention. The function of a bookkeeping system is to organize the Postings attached to Transactions in various ways. All the other types of entries occupy supporting roles in our system.

A Transaction has the following additional fields.

Flag

The single-character flag is usually there to replace the “txn” keyword (Transaction is the only directive which allows being entered without entering its keyword). I’ve been considering changing the syntax definition somewhat to allow not entering the flag nor the keyword, because I’d like to eventually support that. Right now, either the flag or keyword is required. The flag attribute may be set to None.

Payee & Narration

The narration field is a user-provided description of the transaction, such as ”Dinner with Mary-Ann.” You can put any information in this string. It shows up in the journal report. Oftentimes it is used to enrich the transaction with context that cannot be imported automatically, such as ”transfer from savings account to pay for car repairs.”
The payee name is optional, and exists to describe the entity with which the transaction is conducted, such as ”Whole Foods Market” or ”Shell.”
Note that I want to be able to produce reports for all transactions associated with a particular payee, so it would be nice to enter consistent payee names. The problem with this is that the effort to do this right is sometimes too great. Either better tools or looser matching over names is required in order to make this work.
The input syntax also allows only a single string to be provided. By default this becomes the narration, but I’ve found that in practice it can be useful to have just the payee. It’s just a matter of convenience. At the moment, if you want to enter just the payee string you need to append an empty narration string. This should be revisited at some point.

Tags

Tags are sets of strings that can be used to group sets of transactions (or set to None if there are no tags). A view of this subset of transactions can then be generated, including all the usual reports (balance sheet, income statement, journals, etc.). You can tag transactions for a variety of purposes; here are some examples:

- All transactions during a particular travel might be tagged to later summarize your trip expenses. Those transactions will usually occur around the same date and therefore there is convenient syntax used to automatically tag many transactions that are declared together in a file.
- Transactions related to a particular project to be accomplished. I took some classes in an online program once, and tagged all related expenses to it. I use this to track all my immigration-related expenses for a particular stage (e.g. green card).
- Declaring a group of expenses to be paid back by an employer (or your own company) later on.

- Expenses related to moving between locations.

Typical tag names that I use for tags look like "#trip-israel-2012", "#conference-siggraph", and "#course-cfa".

In general, tags are useful where adding a sub-accounts won't do. This is often the case where a group of related expenses are of differing types, and so they would not belong to a single account.

Given the right support from the query tools, they could eventually be subsumed by metadata—I have been considering converting tags into metadata keys with a boolean value of True, in order to remove unnecessary complexity.

Links

Links are a unique sets of strings or None, and in practice will be usually empty for most transactions. They differ from tags only in purpose.

Links are there to chain together a list of related transactions and there are tools used to list, navigate and balance a subset of transactions linked together. They are a way for a transaction to refer to other transactions. They are not meant to be used for summarizing.

Examples include: transaction-ids from trading accounts (these often provide an id to a "related" or "other" transaction); account entries associated with related corrections, for example a reversed fee following a phone call could be linked to the original invalid fee from several weeks before; the purchase and sale of a home, and related acquisition expenses.

In contrast to tags, their strings are most often unique numbers produced by the importers. No views are produced for links; only a journal of a particular links transactions can be produced and a rendered transaction should be accompanied by an actual "link" icon you can click on to view all the other related transactions.

Postings

A list of Postings are attached to the Transaction object. Technically this list object is mutable but in practice we try not to modify it. A Posting can ever only be part of a single Transaction.

Sometimes different postings of the same transaction will settle at different dates in their respective accounts, so eventually we may allow a posting to have its own date to override the transaction's date, to be used as documentation; in the simplest version, we enforce all transactions to occur punctually, which is simple to understand and was not found to be a significant problem in practice. Eventually we might implement this by implicitly converting Transactions with multiple dates into multiple Transactions and using some sort of transfer account to hold the pending balance in-between dates. See the associated proposal for details.

Balancing Postings

The fundamental principle of double-entry book-keeping is enforced in each of the Transaction entries: the sum of all postings must be zero. This section describes the specific way in which we do this, and is the **key piece of logic** that allow the entire system to balance nicely. This is also one of the few places in this system where calculations go beyond simple filtering and aggregation.

As we saw earlier, postings are associated with

- A position, which is a number of units of a lot (which itself may or may not have a cost)
- An optional conversion price

Given this, how do we balance a transaction?

Some terminology is in order: for this example posting:

```
Assets:Investment:HOOL    -50 HOOL {700 USD} @ 920 USD
```

Units. The number of units is the number of position, or "50".

Currency. The commodity of the lot, that is, "HOOL".

Cost. The cost of this position is the number of units times the per-unit cost in the cost currency, that is 50 x 700 = 35000 USD.

(Total) Price. The price of a unit is the attached price amount, 920 USD. The total price of a position is its number of units times the conversion price, in this example 50 x 920 = 46000 USD.

Weight. The amount used to balance each posting in the transaction:

1. If the posting has an associated cost, we calculate the cost of the position (regardless of whether a price is present or not);
2. If the posting has no associated cost but has a conversion price, we convert the position into its total price;
3. Finally, if the posting has no associated cost nor conversion price, the number of units of the lot are used directly.

Balancing is really simple: each of the Posting's positions are first converted into their weight. These amounts are then grouped together by currency, and the final sums for each currency is asserted to be close to zero, that is, within a small amount of tolerance (as inferred by a combination of by the numbers in the input and the options).

The following example is one of case (2), with a price conversion and a regular leg with neither a cost nor a price (case 3):

```
2013-07-22 * "Wired money to foreign account"
  Assets:Investment:HOOL     -35350 CAD @ 1.01 USD        ;; -35000 USD (2)
  Assets:Investment:Cash      35000 USD                   ;;  35000 USD (3)
```

```
                                                        ;;------------------
                                                        ;;      0 USD
```

In the next example, the posting for the first leg has a cost, the second posting has neither:

```
2013-07-22 * "Bought some investment"
  Assets:Investment:HOOL     50 HOOL {700 USD}          ;;  35000 USD (1)
  Assets:Investment:Cash    -35000 USD                  ;; -35000 USD (3)
                                                        ;;------------------
                                                        ;;      0 USD
```

Here's a more complex example with both a price and a cost; the price here is ignored for the purpose of balance and is used only to enter a data-point in the internal price database:

```
2013-07-22 * "Sold some investment"
  Assets:Investment:HOOL    -50 HOOL {700 USD} @ 920 USD  ;; -35000 USD (1)
  Assets:Investment:Cash     46000 USD                    ;;  46000 USD (3)
  Income:CapitalGains       -11000 USD                    ;; -11000 USD (3)
                                                          ;;------------------
                                                          ;;      0 USD
```

Finally, here's an example with multiple commodities:

```
2013-07-05 * "COMPANY INC PAYROLL"
  Assets:US:TD:Checking                          4585.38 USD
  Income:US:Company:GroupTermLife                 -25.38 USD
  Income:US:Company:Salary                      -5000.00 USD
  Assets:US:Vanguard:Cash                         540.00 USD
  Assets:US:Federal:IRAContrib                   -540.00 IRAUSD
  Expenses:Taxes:US:Federal:IRAContrib            540.00 IRAUSD
  Assets:US:Company:Vacation                        4.62 VACHR
  Income:US:Company:Vacation                       -4.62 VACHR
```

Here there are three groups of weights to balance:

- USD: (4485.38) + (-25.38) + (-5000) + (540) ~= 0 USD
- IRAUSD: (540.00) + (-540.00) ~= 0 IRAUSD
- VACHR: (4.62) + (-4.62) ~= 0 VACHR

Elision of Amounts

Transactions allow for at most one posting to be elided and automatically set; if an amount was elided, the final balance of all the other postings is attributed to the balance.

With the inventory booking proposal, it should be extended to be able to handle more cases of elision. An interesting idea would be to allow the elided postings

to at least specify the currency they're using. This would allow the user to elide multiple postings, like this:

```
2013-07-05 * "COMPANY INC PAYROLL"
  Assets:US:TD:Checking                                  USD
  Income:US:Company:GroupTermLife                 -25.38 USD
  Income:US:Company:Salary                      -5000.00 USD
  Assets:US:Vanguard:Cash                         540.00 USD
  Assets:US:Federal:IRAContrib                   -540.00 IRAUSD
  Expenses:Taxes:US:Federal:IRAContrib                   IRAUSD
  Assets:US:Company:Vacation                        4.62 VACHR
  Income:US:Company:Vacation                             VACHR
```

Stream Processing

An important by-product of representing all state using a single stream of directives is that most of the operations in Beancount can be implemented by simple functions that accept the list of directives as input and output a modified list of directives.

For example,

- The "Pad" directive is implemented by processing the stream of transactions, accumulating balances, inserting a new padding transaction after the Pad directive when a balance assertion fails.
- Summarization operations (open books, close books, clear net income to equity) are all implemented this way, as functional operators on the list of streams. For example, opening the books at a particular date is implemented by summing all balances before a certain date and replacing those transactions by new transactions that pull the final balance from an Equity account. Closing the books only involves moving the balances of income statement account to Equity and truncation the list of transactions to that date. This is very elegant—the reporting code can be oblivious to the fact that summarization has occurred.
- Prices on postings held with a cost are normally ignored. The "implicit prices" option is implemented using a plugin which works its magic by processing the stream of operations and inserting otherwise banal Price directives automatically when such a posting is found.
- Many types of verifications are also implemented this way; the sellgains plugin verifies that non-income balances check against the converted price of postings disregarding the cost. This one does not insert any new directives, but may generate new errors.

These are just some examples. I'm aiming to make most operations work this way. This design has proved to be elegant, but also surprisingly flexible and it has given birth to the plugins system available in Beancount; you might be

surprised to learn that I had not originally thought to provide a plugins system... it just emerged over time teasing abstractions and trying to avoid state in my program.

I am considering experimenting with weaving the errors within the stream of directives by providing a new "Error" directive that could be inserted by stream processing functions. I'm not sure whether this would make anything simpler yet, it's just an idea at this point [July 2015].

Stream Invariants

The stream of directives comes with a few guarantees you can rest upon:

- All the directives are ordered by date. Because many dates have multiple directives, in order to have a stable sorting order the line number in the file is used as a secondary sort key.
- All uses of an account is preceded in the stream by a corresponding Open entry. If the input did not provide one, an Open directive is automatically synthesized.
- All Balance directives precede any other directive on a particular day. This is enforced in order to make the processing of balance assertions straightforward and to emphasize that their semantics is to occur at the beginning of the day.

Loader & Processing Order

The process of **loading** a list of entries from an input file is the heart of the project. It is important to understand it in order to understand how Beancount works. Refer to the diagram below.

It consists of

1. A **parsing** step that reads in all the input files and produces a stream of potentially incomplete directives.
2. The processing of this stream of entries by built-in and user-specified plugins. This step potentially produces new error objects.
3. A final validation step that ensures the invariants have not been broken by the plugins.

The beancount.loader module orchestrates this processing. **In the code and documents, I'm careful to distinguish between the terms "parsing" and "loading" carefully.** These two concepts are distinct. "Parsing" is only a part of "loading."

The user-specified plugins are run in the order they are provided in the input files.

Raw mode. Some users have expressed a desire for more control over which built-in plugins are run and in which order they are to be run, and so enabling the "raw" option disables the automatic loading of all built-in plugins (you have to replace those with explicit "plugin" directives if you want to do that). I'm not sure if this is going to be useful yet, but it allows for tighter control over the loading process for experimentation.

Loader Output

The parser and the loader produce three lists:

- entries: A list of directive tuples from beancount.core.data. This is the stream of data which should consists mostly of Transaction and Balance instances. As much as possible, all data transformations are performed by inserting or removing entries from this list. Throughout the code and documents, I refer to these as entries or directives interchangeably.
- errors: A list of error objects. In Beancount I don't use exceptions to report errors; rather, all the functions produce error objects and they are displayed at the most appropriate time. (These deserve a base common type but for now the convention they respect is that they all provide a source (filename, line-no), message and entry attributes.)
- options_map: A dict of the options provided in the file and derived during parsing. Though this is a mutable object, we never modify it once produced by the parser.

Parser Implementation

The Beancount parser is a mix of C and Python 3 code. This is the reason you need to compile anything in the first place. The parser is implemented in C using a flex tokenizer and a Bison parser generator, mainly for performance reasons.

I chose the C language and these crufty old GNU tools because they are portable and will work everywhere. There are better parser generators out there, but they would either require my users to install more compiler tools or exotic languages. The C language is a great common denominator. I want Beancount to work everywhere and to be easy to install. I want this to make it easy on others, but I also want this for myself, because at some point I'll be working on other projects and the less dependencies I have the lighter it will be to maintain aging software. Just looking ahead. (That's also why I chose to use Python instead of some of the other languages I'm more naturally attracted to these days (Clojure, Go, ML); I need Beancount to work... when I'm counting the beans I don't have time to debug problems. Python is careful about its changes and it will be around for a long long time as is.)

There is a lexer file lexer.l written in flex and a Bison grammar in grammar.y. These tools are used to generate the corresponding C source code (lexer.h/.c and

grammar.h/.c). These, along with some hand-written C code that defines some interface functions for the module (parser.h/.c), are compiled into an extension module (_parser.so).

Eventually we could consider creating a small dependency rule in setup.py to invoke flex and Bison automatically but at the moment, in order to minimize the installation burden, I check the generated source code in the repository (lexer.h/c and grammar.h/c).

The interaction between the Python and C code works like this:

- You import beancount.parser.parser and invoke either parse_file() or parse_string(). This uses code in grammar.py to create a Builder object which essentially provides callbacks in Python to process grammar rules.
- parser.py calls a C function in the extension module with the Builder object. That C code sets up flex to be able to read the input file or string then hands over control to the parser by calling the generated C function "yyparse()".
- yyparse() is the parser and will fetch input tokens from the lexer C code using successive calls to "yylex()" and attempt to reduce rules.
- When a rule or token is successfully reduced, the C code invokes callback methods on the Builder object you provided. The parser's rules communicate between each other by passing around "PyObject*" instances, so that code has to be a little careful about correctly handling reference counting. This allows me to run Python code on rule reduction, which makes it really easy to customize parser behaviour yet maintain the performance of the grammar rule processing in C.

Note that the grammar.py Builder derives from a similar Builder class defined in lexer.py, so that lexer tokens recognized calls methods defined in the lexer.py file to create them and parser rule correspondingly calls methods from grammar.py's Builder. This isolation is not only clean, it also makes it possible to just use the lexer to tokenize a file from Python (without parsing) for testing the tokenizer; see the tests where this is used if you're curious how this works.

I just came up with this; I haven't seen this done anywhere else. I tried it and the faster parser made a huge difference compared to using PLY so I stuck with that. I quite like the pattern, it's a good compromise that offers the flexibility of a parser generator yet allows me to handle the rules using Python. Eventually I'd like to move just some of the most important callbacks to C code in order to make this a great deal faster (I haven't done any real performance optimizations yet).

This has worked well so far but for one thing: There are several limitations inherent in the flex tokenizer that have proved problematic. In particular, in order to recognize transaction postings using indent I have had to tokenize the whitespace that occurs at the beginning of a line. Also, single-characters in

the input should be parsed as flags and at the moment this is limited to a small subset of characters. I'd like to eventually write a custom lexer with some lookahead capabilities to better deal with these problems (this is easy to do).

Two Stages of Parsing: Incomplete Entries

At the moment, the parser produces transactions which may or may not balance. The validation stage which runs after the plugins carries out the balance assertions. This degree of freedom is allowed to provide the opportunity for users to enter incomplete lists of postings and for corresponding plugins to automate some data entry and insert completing postings.

However, the postings on the transactions are always complete objects, with all their expected attributes set. For example, a posting which has its amount elided in the input syntax will have a filled in position by the time it comes out of the parser.

We will have to loosen this property when we implement the inventory booking proposal, because we want to support the elision of numbers whose values depend on the accumulated state of previous transactions. Here is an obvious example. A posting like this

```
2015-04-03 * "Sell stock"
  Assets:Investments:AAPL     -10 AAPL {}
  ...
```

should succeed if at the beginning of April 3rd the account contains exactly 10 units of AAPL, in either a single or multiple lots. There is no ambiguity: "sell all the AAPL," this says. However, the fact of its unambiguity assumes that we've computed the inventory balance of that account up until April 3rd…

So the parser will need to be split into two phases:

1. A simple parsing step that produces potentially incomplete entries with missing amounts
2. A separate step for the interpolation which will have available the inventory balances of each account as inputs. This second step is where the booking algorithms (e.g., FIFO) will be invoked from.

See the diagram above for reference. Once implemented, everything else should be the same.

The Printer

In the same package as the parser lives a printer. This isolates all the functionality that deals with the Beancount "language" in the beancount.parser package: the parser converts from input syntax to data structure and the printer does the reverse. No code outside this package should concern itself with the Beancount syntax.

At some point I decided to make sure that the printer was able to round-trip through the parser, that is, given a stream of entries produced by the loader, you should be able to convert those to text input and parse them back in and the resulting set of entries should be the same (outputting the re-read input back to text should produce the same text), e.g.,

Note that the reverse isn't necessarily true: reading an input file and processing it through the loader potentially synthesizes a lot of entries (thanks to the plugins), so printing it back may not produce the same input file (even ignoring reordering and white space concerns).

This is a nice property to have. Among other things, it allows us to use the parser to easily create tests with expected outputs. While there is a test that attempts to protect this property, some of the recent changes break it partially:

- Metadata round-tripping hasn't been tested super well. In particular, some of the metadata fields need to be ignored (e.g., filename and lineno).
- Some directives contain derived data, e.g. the Balance directive has a "diff_amount" field that contains the difference when there is a failure in asserting a balance. This is used to report errors more easily. I probably need to remove these exceptions at some point because it is the only one of its kind (I could replace it with an inserted "Error" directive).

This probably needs to get refined a bit at some point with a more complete test (it's not far as it is).

Uniqueness & Hashing

In order to make it possible to compare directives quickly, we support unique hashing of all directives, that is, from each directive we should be able to produce a short and unique id. We can then use these ids to perform set inclusion/exclusion/comparison tests for our unit tests. We provide a base test case class with assertion methods that use this capability. This feature is used liberally throughout our test suites.

This is also used to detect and remove duplicates. This feature is optional and enabled by the beancount.plugins.noduplicates plugin.

Note that the hashing of directives currently does not include user meta-data.

Display Context

Number are input with different numbers of fractional digits:

```
500    -> 0 fractional digits
520.23 -> 2 fractional digits
1.2357 -> 4 fractional digits
31.462 -> 3 fractional digits
```

Often, a number used for the same currency is input with a different number of digits. Given this variation in the input, a question that arises is *how to render the numbers consistently* in reports?

Consider that:

- We would like that the user not to have to specify the number of fractional digits to use for rendering by default.
- Rendering numbers may differ depending on the context: most of the time we want to render numbers using their most commonly seen number of digits, rounding if necessary, but when we are rendering conversion rates we would like to render numbers using the maximum number of digits ever seen.
- The number of digits used tends to vary with the commodity they represent. For example, US dollars will usually render with two digits of precision; the number of units of a mutual fund such as RGAGX might be recorded by your broker with 3 digits of precision. Japanese Yen will usually be specified in integer units.
- We want to render text report which need to align numbers with varying number of fractional digits together, aligning them by their period or rightmost digits, and possibly rendering a currency thereafter.

In order to deal with this thorny problem, I built a kind of accumulator which is used to record all numbers seen from some input and tally statistics about the precisions witnessed for each currency. I call this a DisplayContext. From this object one can request to build a DisplayFormatter object which can be used to render numbers in a particular way.

I refer to those objects using the variable names dcontext and dformat throughout the code. The parser automatically creates a DisplayContext object and feeds it all the numbers it sees during parsing. The object is available in the options_map produced by the loader.

Realization

It occurs often that one needs to compute the final balances of a list of filtered transactions, and to report on them in a hierarchical account structure. See diagram below.

For the purpose of creating hierarchical renderings, I provide a process called a "realization." A realization is a tree of nodes, each of which contains

- An account name,
- A list of postings and other entries to that account, and
- The final balance for that account,
- A mapping of sub-account names to child nodes.

The nodes are of type "RealAccount," which is also a dict of its children. All variables of type RealAccount in the codebase are by convention prefixed by "real_*". The realization module provides functions to iterate and produce hierarchical reports on these trees of balances, and the reporting routines all use these.

For example, here is a bit of code that will dump a tree of balances of your accounts to the console:

```
import sys
from beancount import loader
from beancount.core import realization

entries, errors, options_map = loader.load_file("filename.beancount")
real_root = realization.realize(entries)
realization.dump_balances(real_root, file=sys.stdout)
```

The Web Interface

Before the appearance of the SQL syntax to filter and aggregate postings, the only report produced by Beancount was the web interface provide by bean-web. As such, bean-web evolved to be good enough for most usage and general reports.

Reports vs. Web

One of the important characteristics of bean-web is that it should just be a thin dispatching shell that serves reports generated from the beancount.reports layer. It used to contain the report rendering code itself, but at some point I began to factor out all the reporting code to a separate package in order to produce reports to other formats, such as text reports and output to CSV. This is mostly finished, but at this time [July 2015] some reports remain that only support HTML output. This is why.

Client-Side JavaScript

I would like to eventually include a lot more client-side scripting in the web interface. However, I don't think I'll be able to work on this for a while, at least not until all the proposals for changes to the core are complete (e.g., inventory booking improvements, settlement splitting and merging, etc.).

If you'd like to contribute to Beancount, improving bean-web or creating your own visualizations would be a great venue.

The Query Interface

The current query interface is the result of a *prototype*. It has not yet been subjected to the amount of testing and refinement as the rest of the codebase.

I've been experimenting with it and have a lot of ideas for improving the SQL language and the kinds of outputs that can be produced from it. I think of it as "70% done".

However, it works, and though some of the reports can be a little clunky to specify, it produces useful results.

It will be the subject of a complete rewrite at some point and when I do, I will keep the current implementation for a little while so that existing scripts don't just break; I'll implement a v2 of the shell.

Design Principles

Minimize Configurability

First, Beancount should have as few options as possible. Second, command-line programs should have no options that pertain to the processing and semantic of the input file (options that affect the output or that pertain to the script itself are fine).

The goal is to avoid feature creep. Just a few options opens the possibility of them interacting in complex ways and raises a lot of questions. In this project, instead of providing options to customize every possible thing, I choose to bias the design towards providing the best behavior by default, even if that means less behavior. This is the Apple approach to design. I don't usually favor this approach for my projects, but I chose it for this particular one. What I've learned in the process is how far you can get and still provide much of the functionality you originally set out for.

Too many command-line options makes a program difficult to use. I think Ledger suffers from this, for instance. I can never remember options that I don't use regularly, so I prefer to just design without them. Options that affect semantics should live in the file itself (and should be minimized as well). Options provided when running a process should be only to affect this process.

Having less options also makes the software much easier to refactor. The main obstacle to evolving a library of software is the number of complex interactions between the codes. Guaranteeing a well-defined set of invariants makes it much easier to later on split up functionality or group it differently.

So.. by default I will resist making changes that aren't generic or that would not work for most users. On the other hand, large-scale changes that would generalize well and that require little configurability are more likely to see implementation.

Favor Code over DSLs

Beancount provides a simple syntax, a parser and printer for it, and a library of very simple data record types to allow a user to easily write their scripts

to process their data, taking advantage of the multitude of libraries available from the Python environment. Should the built-in querying tools fail to suffice, I want it to be trivially easy for someone to build what they need on top of Beancount.

This also means that the tools provided by Beancount don't have to support *all* the features everyone might ever possibly want. Beancount's strength should be representational coherence and simplicity rather the richness of its reporting features. Creating new kinds of reports should be easy; changing the internals should be hard. (That said, it does provide an evolving palette of querying utilities that should be sufficient for most users.)

In contrast, the implementation of the Ledger system provides high-level operations that are invoked as "expressions" defined in a domain-specific language, expressions which are either provided on the command-line or in the input file itself. For the user, this expression language is yet another thing to learn, and it's yet another thing that might involve limitations require expansion and work… I prefer to avoid DSLs if possible and use Python's well understood semantics instead, though this is not always sensible.

File Format or Input Language?

One may wonder whether Beancount's input syntax should be a computer language or just a data format. The problem we're trying to solve is essentially that of making it easy for a human being to create a transactions-to-postings-to-accounts-&-positions data structure.

What is the difference between a data file format and a simple declarative computer language?

One distinction is the intended writer of the file. Is it a human? Or is it a computer? The input files are meant to be manicured by a human, at the very least briefly eyeballed. While we are trying to automate as much of this process as possible via importing code—well, the unpleasant bits, anyway—we do want to insure that we review all the new transactions that we're adding to the ledger in order to ensure correctness. If the intended writer is a human one could file the input under the computer language rubric (most data formats are designed to be written by computers).

We can also judge it by the power of its semantics. Beancount's language is not one designed to be used to perform computation, that is, you cannot define new "Beancount functions" in it. But it has data types. In this regard, it is more like a file format.

Just something to think about, especially in the context of adding new semantics.

Grammar via Parser Generator

The grammar of its language should be compatible with the use of commonly used parser generator tools. It should be possible to parse the language with a simple lexer and some grammar input file.

Beancount's original syntax was modeled after the syntax of Ledger. However, in its attempt to make the amount of markup minimal, that syntax is difficult to parse with regular tools. By making simple changes to the grammar, e.g. adding tokens for strings and accounts and commodities, the Beancount v2 reimplementation made it possible to use a flex/bison parser generator.

This has important advantages:

- It makes it really easy to make incremental changes to the grammar. Using a custom parser without a well-defined grammar can make it quite difficult to prototype syntax changes.
- It makes it much easier to implement a parser in another computer language. I'd like to eventually be able to parse Beancount input files in other languages. The data structures are simple enough that it should be easy to reimplement the core in a different language.
- It's just a lot less code to write and maintain.

(Eventually I'd like to create a file format to generate parsers in multiple languages from a single input. That will be done later, once all the major features are implemented.)

Future Work

Tagged Strings

At the moment, account names, tags, links and commodities are represented as simple Python strings. I've tried to keep things simple. At some point I'd like to refine this a bit and create a specialization of strings for each of these types. I'd need to assess the performance impact of doing that beforehand. I'm not entirely sure how it could benefit functionality yet.

Errors Cleanup

I've been thinking about removing the separate "errors" list and integrating it into the stream of entries as automatically produced "Error" entries. This has the advantage that the errors could be rendered as part of the rest of the stream, but it means we have to process the whole list of entries any time we need to print and find errors (probably not a big deal, given how rarely we output errors).

Types

Python 3.5 introduces types. I plan to use that throughout Beancount sometime after 3.5 is released, and I hope for it to catch a different class of errors than I'm already handling by unit tests.

Conclusion

This document is bound to evolve. If you have questions about the internals, please post them to the mailing-list.

References

Nothing in Beancount is inspired from the following docs, but you may find them interesting as well:

- Accounting for Computer Scientists

[1] I have been considering the addition of a very constrained form of non-balancing postings that would preserve the property of transactions otherwise being balanced, whereby the extra postings would not be able to interact (or sum with) regular balancing postings.

[2] There is an option called "operating_currency" but it is only used to provide good defaults for building reports, never in the processing of transactions. It is used to tell the reporting code which commodities should be broken out to have their own columns of balances, for example.

[3] In previous version of Beancount this was not true, the Posting used to have a 'position' attribute and composed it. I prefer the flattened design, as many of the functions apply to either a Position or a Posting. Think of a Posting as *deriving* from a Position, though, technically it does not.

Design Doc for Ledgerhub

Martin Blais, February 2014

http://furius.ca/beancount/doc/ledgerhub-design-doc

Please note that this document is the original design doc for LedgerHub. LedgerHub is being transitioned back to Beancount. See this postmortem document for details [blais, 2015-12].

Motivation

Several open source projects currently exist that provide the capability to create double-entry transactions for bookkeeping from a text file input. These various double-entry bookkeeping projects include Beancount, Ledger, HLedger, Abandon, and they are independent implementations of a similar goal: the creation of an in-memory representation for double-entry accounting transactions from a text file, and the production of various reports from it, such as balance sheets, income statements, journals, and others. Each implementation explores slightly different feature sets, but essentially all work by reading their input from a file whose format is custom declarative language that describe the transactions, a language which is meant to be written by humans and whose syntax is designed with that goal in mind. While the languages do vary somewhat, the underlying data structures that they define are fairly similar.

An essential part of the process of regularly updating one's journal files is the replication of a real-world account's transaction detail to a single input file in a consistent data format. This is essentially a translation step, meant to bring the transaction details of many institutions' accounts into a single system. Various banks and credit card companies provide downloadable transaction data in either Quicken or Microsoft Money (OFX) formats, and many institutions provide custom CSV files with transaction detail. Moreover, many of these institutions also make regular statements available for download as PDF files, and these can be associated with one's ledger accounts.

The process of translating these external data formats can be automated to some extent. These various files can be translated to output text that can then be massaged by the user to be integrated into input file formats accepted by a double-entry bookkeeping package. Several projects have begun to make inroads in that domain: Ledger-autosync aims at fetching transactions automatically from OFX servers for and translating them for Ledger and HLedger, and Reckon converts CSV files for Ledger. Beancount includes code that can automate the identification of downloaded files to the accounts from a ledger, extract their transaction detail, and automatically file them to a directory hierarchy that mirrors the ledger's chart of accounts. This code should probably live outside

of Beancount. Ledger also sports a "convert" command that attempts to do similar things and a CSV2Ledger Perl script is available that can convert CSV files. HLedger also had a convert command which translated CSV files with optional conversion hints defined in a separate file; HLedger now does the same conversion on-the-fly when the input file is CSV (i.e., CSV is considered a first-class input format).

The programs that fetch and convert external data files do not have to be tied to a single system. Moreover, this is often cumbersome code that would benefit greatly from having a large number of contributors, which could each benefit each other from having common parsers ready and working for the various institutions that they're using or likely to use in the future. I - the author of Beancount - have decided to move Beancount's importing and filing source code outside of its home project and to decouple it from the Beancount source code, so that others can contribute to it, with the intent of providing project-agnostic functionality. This document describes the goals and design of this project.

Goals & Stages

This new project should address the following aspects in a project-agnostic manner:

- **Fetching**: Automate *obtaining* the external data files by connecting to the data sources directly. External tools and libraries such as ofxclient for OFX sources can be leveraged for this purpose. Web scraping could be used to fetch downloadable files where possible. The output of this stage is a list of institution-specific files downloaded to a directory.
  Note that fetching does not just apply to transaction data here; we will also support fetching *prices*. A list of (date, price) entries may be created from this data. We will likely want to support an intermediate format for expressing a list of positions (and appropriate support in the ledgerhub-Ledger/Beancount/HLedger interface to obtain it).
- **Identification**: Given a filename and its contents, automatically *guess* which institution and account configuration the file is for, and ideally be able to extract the date from the file or statement. This should also work with PDF files. The output of this stage is an association of each input file to a particular extractor and configuration (e.g. a particular account name).
- **Extraction**: *Parse each file* (if possible) and extract a list of information required to generate double-entry transactions data structures from it, in some sort of *generic* data structure, such as dicts of strings and numbers, independent of the underlying project's desired output. If possible, a verbatim snippet of the original text that generated the transaction should be attached to the output data structure. The output of this stage is a data structure, e.g., a list of Python dictionaries in some defined format.

- **Transform**: Given some information from the past transaction history contained in a journal, using simple learning algorithms, a program should be able to apply transformations on the transactional information extracted from the previous step. The most common use case for this is to automatically add a categorization posting to transactions that have a single posting only. For example, transactions from credit card statements typically include the changes in balance of the credit card account but all transactions are left to be manually associated with a particular expense account. Some of this process can be automated at this stage.
- **Rendering**: Convert the internal transactions data structures to the particular syntax of a double-entry bookkeeping project implementation and to the particular desired syntax variants (e.g. currency formatting, comma vs. dot decimal separator, localized input date format). This steps spits out text to be inserted into an input file compatible with the ledger software of choice.
- **Filing**: Sanitize the downloaded files' filenames and move them into a well organized and structured directory hierarchy corresponding to the identified account. This can run from the same associations derived in the identification step.

Apart from the Render stage, all the other stages should be implemented without regard for a particular project, this should work across all ledger implementations. The Rendering code, however, should specialize, import source code, and attempt to add as many of the particular features provided by each project to its output text.

Where necessary, interfaces to obtain particular data sets from each ledger implementation's input files should be provided to shield the common code from the particular implementation details of that project. For instance, a categorization Transform step would need to train its algorithm on some of the transaction data (i.e., the narration fields and perhaps some of the amounts, account names, and dates). Each project should provide a way to obtain the necessary data from its input data file, in the same format.

Details of Stages

Fetching

By default, a user should be able to click their way to their institution's website and download documents to their ~/Downloads directory. A directory with some files in it should be the reasonable default input to the identification stage. This directory should be allowed to have other/garbage files in it, the identification step should be able to skip those automatically.

A module that can automatically fetch the data needs to be implemented. Ideally this would not require an external tool. The data extracted should also have a copy saved in some Downloads directory.

This is the domain of the ledger-autosync project. Perhaps we should coordinate input/outputs or even integrate call some of its library code at this stage. The author notes that fetching data from OFX servers is pretty easy, though the begin/end dates will have to get processed and filtered.

Automatic fetching support will vary widely depending on where the institutions are located. Some places have solid support, some less. Use the data from ofxhome.com to configure.

Fetching Prices

For fetching prices, there are many libraries out there. Initially we will port Beancount's bean-prices to ledgerhub.

Identification

The identification stage consists in running a driver program that

- Searches for files in a directory hierarchy (typically your ~/Downloads folder)
- If necessary, converts the files into some text/ascii format, so that regular expressions can be matched against it (even if the output is messy, e.g., with PDF files converted to ASCII). This works well for PDF files: despite the fact that we cannot typically extract transactional data from them, we can generally pretty reliably identify which account they're for and almost always extract the statement date as well.
- Check a list of regular expressions against the ASCII'fied contents. If the regular expressions all match, the configuration is associated to the filename.

Note that more than one configuration may be associated to the same file because some files contain many sections, sections for which different importers may be called on to extract their data (e.g., OFX banking + OFX credit card can be mixed in the same file, and some institutions do).

The net result of this process is an association of each filename with the a specific importer object instantiated in the configuration file. These importer objects are created with a set of required account names which they use to produce the Ledger-like syntax from the downloaded file that was associated with it. Here is an example configuration for two importers:

from ledgerhub.sources.rbc import rbcinvesting, rbcpdf

CONFIG = [

...

(('FileType: application/vnd.ms-excel', r'Filename: .*Activity-123456789-',),
rbcinvesting.Importer({

```
'FILE' : 'Assets:CA:RBC-Investing:Taxable',
'cash' : 'Assets:CA:RBC-Investing:Taxable:Cash',
'positions' : 'Assets:CA:RBC-Investing:Taxable',
'interest' : 'Income:CA:RBC-Investing:Taxable:Interest',
'dividend' : 'Income:CA:RBC-Investing:Taxable:Dividends',
'fees' : 'Expenses:Financial:Fees',
'commission' : 'Expenses:Financial:Commissions',
'transfer' : 'Assets:CA:RBC:Checking',
})),

(('FileType: application/pdf',
'Filename:.*/123456789-\d\d\d\d[A-Z][a-z][a-z]\d\d-\d\d\d\d[A-Z][a-z][a-
z]\d\d.pdf'),
rbcpdf.Importer({
'FILE': 'Assets:CA:RBC-Investing:RRSP',
})),
```

The configuration consists in a list, for each possible importer, of a pair of 1) a list of regular expressions which all should match against a "match text", which is a "textified" version of the contents of a file to be imported, and 2) an importer object, configured with a specific set of accounts to use for producing transactions. Each importer requires a particular set of output accounts which it uses to create its transactions and postings. The ledger's filename, and a list of these (regexps, importer) pairs is all that is necessary for the driver to carry out all of its work.

The textification consists in a simple and imperfect conversion of downloaded file that are in binary format to something that we can run regular expressions against. For an OFX file or CSV file there is no conversion required for textification, we can just match against the text contents of those files; for an Excel/XLS file, we need to convert that to a CSV file, which can then be searched; for a PDF file, a number of different pdf-to-text converters are attempted until one succeeds (the tools for this are notoriously unreliable, so we have to try various ones). Note that this converted "match text" is only created temporarily and only for the purpose of identification; the importer will get the original binary file to do its work.

It is not entirely clear whether the regular expressions can be standardized to avoid having the user configure them manually. In practice, I have found it often necessary, or at least very convenient, to place an account id in my import configuration. It is true that configuring each of the possible downloads can be a hassle that requires the user to do a bit of guesswork while looking at the contents of each file, but this has been much more reliable in practice than attempts at normalizing this process, likely because it is a much easier problem to uniquely distinguish between all the files of a particular user than to distinguish between all the types of files. Using an account id in one of the regular expressions is the easy way to do that, and it works well. This also provides a clear place to attach the list of accounts to a particular importer,

something that necessarily requires user input anyway.

Extraction

Once the association is made, we run the importers on each of the files. Some data structure is produced. The importers each do what they do - this is where the ugly tricks go. Ideally, we should build a library of common utilities to help parsing similar file types.

Though each of the importer modules should be pretty much independent, some common functionality can be imagined, for example, how one deals with different stocks held in a single investment account, could be configured outside of each importer (e.g., preferred method could be to create a subaccount of that account, with the symbol of the stock, or otherwise).

Note [AMaffei]: This could output a generic and well-defined CSV file format if you want to have the option of running the various steps as separate UNIX-style tools and/or process the intermediate files with regular text processing tools.

Transform

Some transformations should be independent of importers. In particular, automatically categorizing incomplete transactions is not dependent on which importer created the transaction. I'd like to keep this step as general as possible so that other embellishment steps can be inserted here in the future. Right now, I can only think of the following uses:

1. Auto-categorization of transactions with only a single leg
2. Detection of duplicate transactions: imported files often contain transactions which are already in the ledger; those should be either ignored or marked as such. In practice, this is not as easy as it sounds, because a straightforward date + narration comparison will fail: if the same transaction comes from two input data files, one side always ends up getting merged to the other, and sometimes even the date differs a bit. Some amount of fuzzy matching is required.
3. Normalization of payee names: the imported names of payees are often cut short or include some irrelevant words, such as "LLC", city names, and/or number codes. It may be desirable to somehow clean those up automatically.

This step involves a bootstrapping phase, where we will extract some data from the actual ledger that the transactions are meant to be imported into. We will implement a generic interface that should allow each ledger language implementation to provide relevant data for training.

The output data here should be in the same format as its input, so that we can optionally skip this phase.

Rendering

An output renderer should be selected by the driver. This is where we convert the extracted data structures to the particular flavor of ledger implementation you're using. Each of the renderer implementations should be free to import modules from its particular implementation, and we should be careful to constraint these import dependencies to only these modules, to make sure that only a single ledger implementation is required in order for the code to run.

Options for rendering style could be defined here, for each renderer, because each of the languages have particularies.

[AMaffei] Also, it should be possible to provide a generic renderer that takes printf-style format strings to output in any desired format.

Filing

Importers should be able to look at the textified contents of the files and find the file/statement date. This is useful, because we can rename the file by prepending the date of the statement, and the date at which we download the statement or transaction files is rarely the same date at which it was generated. In the case where we are not able to extract a date from the file, we fall back on the filename's last modified time.

A target directory should be provided and we should move each file to the account with which it is associated. For example, a file like this:

```
~/Downloads/ofx32755.qbo
```

should be moved to a directory

```
.../Assets/US/RBC/Checking/2013-11-27.ofx32755.qbo
```

if it is associated by the identification step with an importer for the Assets:US:RBC:Checking account. For this purpose, all the importers should have a required "filing" account associated with them.

As far as I know only Beancount implements this at the moment, but I suspect this convenient mechanism of organizing and preserving your imported files will be found useful by others. Given a list of directories, Beancount automatically finds those files and using the date in the filename, is able to render links to the files as line items in the journal web pages, and serve their contents when the user clicks on the links. Even without this capability, it can be used to maintain a cache of your documents (I maintain mine in a repository which I sync to an external drive for backup).

Implementation Details

Notes about the initial implementation:

- The implementation of this project will be carried out in Python3. Why Python?
  - The performance of importing and extracting is largely irrelevant, a dynamic language works well for this type of task
  - Parsing in a dynamic language works great, there are many available libraries
  - Python3 is now widely distributed and all desired parsing libraries are commonly available for it at this point
- The project will be hosted at either http://hg.furius.ca/public/ledgerhub/ (or perhaps https://bitbucket.org/blais/ledgerhub/ ?)
- All modules should be tested, including testing with sample input. If you want to add a new module, you should need to provide an anonymized sample file for it. We will have to have an automated test suite, because past experience has shown this type of code to be quite brittle and fragile to new and unexpected inputs. It's easy to write, but it's also easy to break.
  - In order to test binary files that cannot be anonymized, we will provide the ability to test from match-text instead of from original binary statement PDF. Those files are generally not extractable anyhow and are only there for identification and filing (e.g. a PDF statement, we can't extract any meaningful data out of those except perhaps for the statement date).
- There should be a quick way to test a particular importer with a particular downloaded file with zero configuration, even if the output account names are a little wonky.
- There needs to be clean and readable tracing for what the importers are doing, including a debugging/verbose option.
- We provide a single function to call as the driver for your own import script. Your configuration is a script / your script is the configuration. You call a function at the end. We will also provide a script that imports a filename and fetches an attribute from it, for those who want a more traditional invocation.
- We should keep types simples, but use the standard datetime types for dates, decimal.Decimal for numbers, and strings for currencies/commodities.

This is obviously based on my current importers code in Beancount. I'm very open to new ideas and suggestions for this project. Collaborations will be most welcome. The more importers we can support, the better.

Importers Interface

Each importer should be implemented as a class that derives from this one:

class ImporterBase:
"Base class/interface for all source importers."

A dict of required configuration variables to their docstring.
This declares the list of options required for the importer
to be provided with, and their meaning.
REQUIRED_CONFIG = {}

def ___init___(self, config):
"""Create an importer.
Most concrete implementations can just use this without overriding.

Args:
config: A dict of configuration accounts, that must match the
REQUIRED_CONFIG values.
"""
a dict of Configuration values. This can be accessed publicly.
assert isinstance(config, dict)
self.config = config

# Check that the config has just the required configuration values.
if not verify_config(self, config, self.REQUIRED_CONFIG):
raise ValueError("Invalid config {}, requires {}".format(
config, self.REQUIRED_CONFIG))

def get_filing_account(self):
"""Return the account for moving the input file to.

Returns:
The name of the account that corresponds to this importer.
"""
return self.config['FILE']

def **import_file**(self, filename):
"""Attempt to import a file.

Args:
filename: the name of the file to be imported.
Returns:
A list of new, imported entries extracted from the file.
"""
raise NotImplementedError

def **import_date**(self, filename, text_contents):
"""Attempt to obtain a date that corresponds to the given file.

Args:

filename: the name of the file to extract the date from
text_contents: an ASCII text version of the file contents,
whatever format it is originally in.
Returns:
A date object, if successful, or None.
"""
raise NotImplementedError

For each importer, a detailed explanation of how the original input file on the institution's website is to be found and downloaded should be provided, to help those find the correct download when adding this importer (some institutions provide a variety of download formats). In addition, a one-line description of the input file support should be provided, so that we can render at runtime a list of the supported file types.

References

Other projects with the same goal as importing account data into Ledger are listed here.

- Ledger's "convert" command
- HLedger with its built-in readers
- Reckon
- OFXmate (GUI for ledger-autosync)
- CSV2Ledger
- icsv2ledger
- csv2ledger (seems to lack active maintainers)

Update (Nov 2015): This design doc has been implemented and the project is being transitioned back to Beancount. Read the details here.

External Contributions to Beancount

Martin Blais - Updated: April 2016

http://furius.ca/beancount/doc/contrib

Links to codes written by other people that build on top of or that are related to Beancount and/or Ledgerhub.

Plugins

split_transactions: Johann Klähn wrote a plugin that can split a single transaction into many against a limbo account, as would be done for depreciation.

zerosum: redstreet0 wrote a plugin to match up transactions that when taken together should sum up to zero and move them to a separate account.

effective_dates: redstreet0 wrote a plugin to book different legs of a transaction to different dates

depreciation: Alok Parlikar wrote a plugin to automatically add entries at the EOY for the depreciation of assets.

beancount-plugins: Dave Stephens created a repository to share various of his plugins related to depreciation.

beancount-plugins-zack: Stefano Zacchiroli created this repository to share his plugins.

beancount-oneliner: Akuukis created a plugin to write an entry in one line (PyPi).

beancount-interpolate: Akuukis created plugins for Beancount to interpolate transactions (recur, split, depr, spread) (PyPi).

metadata-spray: Add metadata across entries by regex expression rather than having explicit entries (by Vivek Gani).

Tools

alfred-beancount (Yue Wu): An add-on to the "Alfred" macOS tool to quickly enter transactions in one's Beancount file. Supports full account names and payees match.

bean-add (Simon Volpert): A Beancount transaction entry assistant.

hoostus/fincen_114 (Justus Pendleton): A FBAR / FinCEN 114 report generator.

ghislainbourgeois/beancount_portfolio_allocation (Ghislain Bourgeois): A quick way to figure out the asset allocations in different portfolios.

hoostus/portfolio-returns (Justus Pendleton): portfolio returns calculator

Importers

yodlee importer: redstreet0 wrote an importer for fetching data from the Yodlee account aggregator. Apparently you can get free access as per this thread.

plaid2text: An importer from Plaid which stores the transactions to a Mongo DB and is able to render it to Beancount syntax. By Micah Duke.

jbms/beancount-import: A tool for semi-automatically importing transactions from external data sources, with support for merging and reconciling imported transactions with each other and with existing transactions in the beancount

journal. The UI is web based. (Announcement, link to previous version). By Jeremy Maitin-Shepard.

awesome-beancount: A collection of importers for Chinese banks + tips and tricks. By Zhuoyun Wei.

beansoup: Filippo Tampieri is sharing some of his Beancount importers and auto-completer in this project.

montaropdf/beancount-importers: An importer to extract overtime and vacation from a timesheet format for invoicing customers.

siddhantgoel/beancount-dkb (Siddhant Goel): importer for DKB CSV files.

Converters

plaid2text: Python Scripts to export Plaid transactions and transform them into Ledger or Beancount syntax formatted files.

gnucash-to-beancount: A script from Henrique Bastos to convert a GNUcash SQLite database into an equivalent Beancount input file.

debanjum/gnucash-to-beancount: A fork of the above.

andrewStein/gnucash-to-beancount : A further fork from the above two, which fixes a lot of issues (see this thread).

hoostus/beancount-ynab : A converter from YNAB to Beancount.

ledger2beancount: A script to convert ledger files to beancount. It was developed by Stefano Zacchiroli and Martin Michlmayr.

smart_importer: A smart importer for beancount and fava, with intelligent suggestions for account names. By Johannes Harms.

beancount-export-patreon.js: JavaScript that will export your Patreon transactions so you can see details of exactly who you've been giving money to. By kanepyork@gmail.

alensiljak/pta-converters (Alen Šiljak): GnuCash -> Beancount converter (2019).

Price Sources

hoostus/beancount-price-sources : A Morningstar price fetcher which aggregates multiple exchanges, including non-US ones.

andyjscott/beancount-financequote : Finance::Quote support for bean-price.

aamerabbas/beancount-coinmarketcap: Price fetcher for coinmarketcap (see post).

Development

Py3k type annotations: Yuchen Ying is implementing python3 type annotations for Beancount.

Documentation

Beancount Source Code Documentation (Dominik Aumayr): Sphinx-generated source code documentation of the Beancount codebase. The code to produce this is located here.

SQL queries for Beancount (Dominik Aumayr): Example SQL queries.

Beancount —— (Zhuoyun Wei): A tutorial (blog post) in Chinese on how to use Beancount.

Managing my personal finances with Beancount (Alex Johnstone)

Counting beans—and more—with Beancount (LWN)

Interfaces / Web

fava: A web interface for Beancount (Dominik Aumayr): Beancount comes with its own simple web front-end ("bean-web") intended merely as a thin shell to invoke and display HTML versions of its reports. "Fava" is an alternative web application front-end with more & different features, intended initially as a playground and proof-of-concept to explore a newer, better design for presenting the contents of a Beancount file.

Inventory Booking

A Proposal for an Improvement on Command-Line Bookkeeping

==========

Martin Blais, June 2014

http://furius.ca/beancount/doc/proposal-booking

Motivation

Problem Description

Lot Date

Average Cost Basis

Capital Gains Sans Commissions

Cost Basis Adjustments

Dealing with Stock Splits

Previous Solutions

Motivation

The problem of "inventory booking," that is, selecting which of an inventory's trade lots to reduce when closing a position, is a tricky one. So far, in the command-line accounting community, relatively little progress has been made in supporting the flexibility to deal with many common real-life cases. This

document offers a discussion of the current state of affairs, describes the common use cases we would like to be able to solve, identifies a set of requirements for a better booking method, and proposes a syntax and implementation design to support these requirements, along with clearly defined semantics for the booking method.

Problem Description

The problem under consideration is the problem of deciding, for a double-entry transaction that intends to reduce the size of a position at a particular point in time in a particular account, which of the account inventory lots contained at that point should be reduced. This should be specified using a simple data entry syntax that minimizes the burden on the user.

For example, one could enter a position in HOOL stock by buying two lots of it at different points in time:

```
2014-02-01 * "First trade"
  Assets:Investments:Stock        10 HOOL {500 USD}
  Assets:Investments:Cash
  Expenses:Commissions          9.95 USD

2014-02-15 * "Second trade"
  Assets:Investments:Stock         8 HOOL {510 USD}
  Assets:Investments:Cash
  Expenses:Commissions          9.95 USD
```

We will call the two sets of shares "trade lots" and assume that the underlying system that counts them is keeping track of each of their cost and date of acquisition separately.

The question here is, if we were to sell some of this stock, *which of the shares should we select?* Those from the first trade, or those from the second? This is an important decision, because it has an impact on capital gains, and thus on taxes. (The account of capital gains is described in more detail in the "Stock Trading" section of the Beancount cookbook if you need an explanation of this.) Depending on our financial situation, this year's trading history, and the unrealized gains that a trade would realize, sometimes we may want to select one lot over another. By now, most discount brokers even let you select your specific lot when you place a trade that reduces or closes a position.

It is important to emphasize here the two directions of inventory booking:

1. **Augmenting a position.** This is the process of creating a new trading lot, or adding to an existing trading lot (if the cost and other attributes are identical). This is easy, and amounts to adding a record to some sort of mapping that describes an account's balance (I call this an "inventory").

2. **Reducing a position.** This is generally where booking complications

take place, and the problem is essentially to figure out which lot of an existing position to remove units from.

Most of this document is dedicated to the second direction.

Lot Date

Even if the cost of each lot is identical, the acquisition date of the position you're intending to close matters, because different taxation rules may apply, for example, in the US, positions held for more than 12 months are subject to a significantly lower tax rate ("the long-term capital gains rate"). For example, if you entered your position like this:

```
2012-05-01 * "First trade"
  Assets:Investments:Stock        10 HOOL {300 USD}
  Assets:Investments:Cash
  Expenses:Commissions          9.95 USD

2014-02-15 * "Second trade at same price"
  Assets:Investments:Stock         8 HOOL {300 USD}
  Assets:Investments:Cash
  Expenses:Commissions          9.95 USD
```

and we are booking a trade for 2014-03-01, selecting the first lot would result in a long-term (low) gain, because two years have passed since acquisition (position held from 2012-05-01 to 2014-03-01) while booking against the second would result in a short-term (high) capital gains tax. So it's not just a matter of the cost of the lots. Our systems don't aim to file taxes automatically at this point but we would like to enter our data in a way that eventually allows this kind of reporting to take place.

Note that this discussion assumes that you were able to *decide* which of the lots you could trade against. There are several taxation rules that may apply, and in some cases, depending on which country you live in, you may not have a choice, the government may dictate how you are meant to report your gains.

In some cases you may even need to be able to apply multiple booking methods to the same account (e.g., if an account is subject to a taxation claim from multiple countries). We will ignore this exotic case in this document, as it is quite rare.

Average Cost Basis

Things get more complicated for tax-sheltered accounts. Because there are no tax consequences to closing positions in these accounts, brokers will normally choose to account for the book value of your positions as if they were a single lot. That is, the cost of each share is computed using the weighted average of the cost of the position you are holding. This can equivalently be calculated by

summing the total cost of each position and then dividing by the total number of shares. To continue with our first example:

```
10 HOOL x 500 USD/HOOL = 5000 USD
```

So if you were to close some of your position and sell some shares, you would do so at a cost of 504.44 USD/HOOL. The gain you would realize would be relative to this cost; for example if you sold 5 shares at 520 USD/HOOL, your gain would be calculated like this:

```
(520 USD/HOOL - 504.44 USD/HOOL) x 5 HOOL = 77.78 USD
```

This type of booking method is made evident by two kinds of financial events that you might witness occur in these types of accounts:

1. Fees may be withdrawn by selling some shares reported at their current market price. You will see these if you hold a retirement account at Vanguard in the USA. The broker takes a fixed amount of dollars per quarter (5$, with my employer's plan) that is backed out in terms of a small, fractional number of shares of each fund to sell. That transaction is oblivious to capital gains: they simply figure out how many shares to sell to cover their fee and don't report a gain to you. You have to assume it doesn't matter. Most people don't track their gains in non-taxable accounts but freaks like us may want to compute the return nonetheless.

2. Cost basis readjustments may occur spontaneously. This is rare, but I have seen this once or twice in the case of mutual funds in a Canadian tax-deferred account: based on some undisclosed details of their trading activity, the fund management has had to issue an adjustment to the cost basis, which you are meant to apply to your positions. You get an email telling you how much the adjustment was for. 99% of people probably don't blink, don't understand and ignore this message… perhaps rightly so, as it has no impact on their taxes, but if you want to account for your trading returns in that account, you do need to count it. So we need to be able to add or remove to the cost basis of existing positions.

Note that the average cost of your positions changes on every trade and needs to get recalculated every time that you add to an existing position. A problem with this is that if you track the cost per share, these multiple recalculations may result in some errors if you store the amounts using a fixed numerical representation (typically decimal numbers). An alternative method to track the cost in our data structures, one that is more stable numerically, is to simply keep track of the total cost instead of the cost-per-share. Also note that using fractional numbers does not provide a sufficient answer to this problem: in practice, the broker may report a specific cost, one that is rounded to specific number of decimals, and we may want to be able to book our reducing trade using that reported number rather than maintain the idealized amount that a fractional representation would. Both using a fixed decimal or a fractional representation pose problems. We will largely ignore those problems here.

In summary: we need to be able to support book against lots at their average cost, and we need to be able to adjust the cost basis of an existing position. I've been thinking about a solution to implement this that would not force an account to be marked with a special state. I think we can do this using only inventory manipulations. All we need is a small change to the syntax that allows the user to indicate to the system that it should book against the average cost of all positions.

Using the first example above, acquiring the shares would take the same form as previously, that is after buying two lots of 10 and 8 HOOL units, we end up with an inventory that holds two lots, one of 10 HOOL {500 USD} and one of 8 HOOL {510 USD}. However, when it's time to sell, the following syntax would be used (this is an old idea I've been meaning to implement for a while):

```
2014-02-01 * "Selling 5 shares at market price 550 USD"
  Assets:Investments:Stock               -5 HOOL {*}
  Assets:Investments:Cash           2759.95 USD
  Expenses:Commissions                 9.95 USD
  Income:Investments:CapitalGains
```

When the "*" is encountered in lieu of the cost, like this, it would:

1. Merge all the lots together and recalculate the average price per share (504.44 USD)

2. Book against this merged inventory, reducing the resulting lot.

After the transaction, we would end up with a single position of 13 HOOL {504.44 USD). Adding to this position at a different price would create a new lot, and those would get merged again the next time a reduction occurs at average cost. We do not need to merge lots until there is a reduction. Apart from concerns of accumulating rounding error, this solution is correct mathematically, and maintains the property that accounts aren't coerced into any specific booking method—a mix of methods could be used on every reduction. This is a nice property, even if not strictly necessary, and even if we want to be able to just specify a "default" booking method to use per account and just stick with it throughout. It's always nice to have the flexibility to support exceptions, because in real life, they sometimes occur.

Capital Gains Sans Commissions

We would like to be able to support the automatic calculation of capital gains without the commissions. This problem is described in much detail in the Commissions section of the trading documentation and in a thread on the mailing-list. The essence of the complication that occurs is that one cannot simply subtract the commissions incurred during the reporting period from the gains that include commissions, because the commissions that was incurred to acquire the position must be pro-rated to the shares of the position that are sold. The simplest and most common way to implement this is to include the costs of

acquiring the position into the cost basis of the position itself, and deduct the selling costs from the market value when a position is reduced.

Whatever new design we come up with must allow us to count these adjusted gains as this is essential to various individuals' situations. In Beancount, I have figured out a solution for this problem, which luckily enough involves only an automated transformation of a transaction flagged by the presence of a special flag on its postings... if and only if I can find a way to specify which lot to book against without having to specify its cost, because once the cost of commissions gets folded into the cost of the position, the adjusted cost does not show up anywhere in the input file, the user would have to calculate it manually, which is unreasonable. In the solution I'm proposing, the following transactions would be transformed automatically, triggered by the presence of the "C" flag:

```
2014-02-10 * "Buy"
  Assets:US:Invest:Cash    -5009.95 USD
  C Expenses:Commissions       9.95 USD
  Assets:US:Invest:HOOL       10.00 HOOL {500 USD / aa2ba9695cc7}

2014-04-10 * "Sell #1"
  Assets:US:Invest:HOOL       -4.00 HOOL {aa2ba9695cc7}
  C Expenses:Commissions       9.95 USD
  Assets:US:Invest:Cash     2110.05 USD
  Income:US:Invest:Gains

2014-05-10 * "Sell #2"
  Assets:US:Invest:HOOL       -6.00 HOOL {aa2ba9695cc7}
  C Expenses:Commissions       9.95 USD
  Assets:US:Invest:Cash     3230.05 USD
  Income:US:Invest:Gains
```

They would be automatically transformed by a plugin into the following and replace the original transactions above:

```
2014-02-10 * "Buy"
  Assets:US:Invest:Cash        -5009.95 USD
  X Expenses:Commissions           9.95 USD
  X Income:US:Invest:Rebates      -9.95 USD
  X Assets:US:Invest:HOOL         10.00 HOOL {500.995 USD / aa2ba9695cc7}

2014-04-10 * "Sell #1"
  X Assets:US:Invest:HOOL         -4.00 HOOL {aa2ba9695cc7}
  X Expenses:Commissions           9.95 USD
  X Income:US:Invest:Rebates      -9.95 USD
  Assets:US:Invest:Cash         2110.05 USD
  Income:US:Invest:Gains ; Should be (530-500)*4 - 9.95*(4/10) - 9.95 =
                         ; 106.07 USD
```

```
2014-05-10 * "Sell #2"
  X Assets:US:Invest:HOOL         -6.00 HOOL {aa2ba9695cc7}
  X Expenses:Commissions           9.95 USD
  X Income:US:Invest:Rebates      -9.95 USD
  Assets:US:Invest:Cash         3230.05 USD
  Income:US:Invest:Gains ; Should be (540-500)*6 - 9.95*(6/10) - 9.95 =
                         ; 224.08 USD
```

The "X" flag would just mark the postings that have been transformed, so maybe they can be rendered with a special color or attribute later on. The rebates account is included as a separate counter just to make the transaction balance.

The reason I need to relax the disambiguation of trading lots is that the user needs to be able to specify the matching leg without having to specify the cost of the position, because at the point where the position is reduced (2014-04-10), there is no way to figure out what the cost of the original lot was. Just to be clear, in the example above, this means that if all the information have is 4 HOOL and 500 USD, there is no way to back out a cost of 500.995 USD that could specify a match with the opening trade lot, because that happened with 10 HOOLs.

So we need to be more explicit about booking. In the example above, I'm selecting the matching lot "by label," that is, the user has the option to provide a unique lot identifier in the cost specification, and that can later on be used to disambiguate which lot we want to book a reduction/sale against. The example above uses the string "aa2ba9695cc7" in this way.

(An alternative solution to this problem would involve keep track of *both* the original cost (without commissions), and the actual cost (with commissions), and then finding the lot against the former, but using the latter to balance the transaction. This idea is to allow the user to keep using the cost of a position to select the lot, but I'm not even sure if that is possible, in the presence of various changes to the inventory. More thought is required on this matter.)

Cost Basis Adjustments

In order to adjust the cost basis, one can replace the contents of an account explicitly like this:

```
2014-03-15 * "Adjust cost basis from 500 USD to 510 USD"
  Assets:US:Invest:HOOL      -10.00 HOOL {500 USD}
  Assets:US:Invest:HOOL       10.00 HOOL {510 USD}
  Income:US:Invest:Gains
```

This method works well and lets the system automatically compute the gains. But it would be nice to support making adjustments in price units against the total cost of the position, for example, "add 340.51 USD to the cost basis of this position". The problem is that the adjusted cost per share now needs to be

computed by the user… it's inconvenient. Here's one way we could support this well, however:

```
2014-03-15 * "Adjust cost basis from 500 USD to 510 USD"
  Assets:US:Invest:HOOL      -10.00 HOOL {500 USD}
  Assets:US:Invest:HOOL       10.00 HOOL {}
  Income:US:Invest:Gains    -340.51 USD
```

If all the legs are fully specified - they have a calculatable balance - we allow a single price to be elided. This solves this problem well.

Dealing with Stock Splits

Accounting for stock splits creates some complications in terms of booking. In general, the problem is that we need to deal with changes in the meaning of a commodity over time. For example, you could do this in Beancount right now and it works:

```
2014-01-04 * "Buy some HOOL"
  Assets:Investments:Stock        10 HOOL {1000.00 USD}
  Assets:Investments:Cash

2014-04-17 * "2:1 split into Class A and Class B shares"
  Assets:Investments:Stock       -10 HOOL {1000.00 USD}
  Assets:Investments:Stock        10 HOOL  {500.00 USD}
  Assets:Investments:Stock        10 HOOLL {500.00 USD}
```

One problem with splitting lots this way is that the price and meaning of a HOOL unit before and after the split differs, but let's just assume we're okay with that for now (this can be solved by *relabeling* the commodity by using another name, ideally in a way that is not visible to the user - we have a pending discussion elsewhere).

The issue that concerns booking is, when you sell that position, which cost do you use? The inventory will contain positions at 500 USD, so that's what you should be using, but is it obvious to the user? We can assume this can be learned with a recipe .

Now, what if we automatically attach the transaction date? Does it get reset on the split and now you would have to use 2014-04-17? If so, we could not automatically inspect the list of trades to determine whether this is a long-term vs. short-term trade. We need to somehow preserve some of the attributes of the original event that augmented this position, which includes the original trade date (not the split date) and the original user-specified label on the position, if one was provided.

Forcing a Single Method per Account

For accounts that will use the average booking method, it may be useful to allow specifying that an account should only use this booking method. The idea is to avoid data entry errors when we know that an account is only able to use this method.

One way to ensure this is to automatically aggregate inventory lots when augmenting a position. This ensures that at any point in time, there is only a single lot for each commodity. If we associated an inventory booking type to each account, we could define a special one that also triggers aggregation upon the addition of a position.

Another approach would be to not enforce these aggregations, but to provide a plugin that would check that for those accounts that are marked as using the average cost inventory booking method by default, that only such bookings actually take place.

Previous Solutions

This section reviews existing implementations of booking methods in command-line bookkeeping systems and highlights specific shortcomings that motivate why we need to define a new method.

Shortcomings in Ledger

(Please note: I'm not deeply familiar with the finer details of the inner workings of Ledger; I inferred its behavior from building test examples and reading the docs. If I got any of this wrong, please do let me know by leaving a comment.)

Ledger's approach to booking is quite liberal. Internally, Ledger does not distinguish between conversions held at cost and regular price conversions: all conversions are assumed held at cost, and the only place trading lots appear is at reporting time (using its --lots option).

Its "{}" cost syntax is meant to be used to disambiguate lots on a position reduction, not to be used when acquiring a position. The cost of acquiring a position is specified using the "@" price notation:

```
2014/05/01 * Buy some stock (Ledger syntax)
  Assets:Investments:Stock        10 HOOL @ 500 USD
  Assets:Investments:Cash
```

This will result in an inventory of 10 HOOL {500 USD}, that is, the cost is *always* stored in the inventory that holds the position:

```
$ ledger -f l1.lgr bal --lots
10 HOOL {USD500} [2014/05/01]
            USD-5000  Assets:Investments
            USD-5000    Cash
```

```
10 HOOL {USD500} [2014/05/01]    Stock
--------------------
10 HOOL {USD500} [2014/05/01]
            USD-5000
```

There is no distinction between conversions at cost and without cost—all conversions are tracked as if "at cost." As we will see in the next section, this behaviour is distinct from Beancount's semantics, which requires a conversion held at cost to be specified using the "{}" notation for both its augmenting and reducing postings, and distinguishes between positions held at cost and price conversions.

The advantage of the Ledger method is a simpler input mechanism, but it leads to confusing outcomes if you accumulate many conversions of many types of currencies in the same account. An inventory can easily become fragmented in its lot composition. Consider what happens, for instance, if you convert in various directions between 5 different currencies… you may easily end up with USD held in CAD, JPY, EUR and GBP and vice-versa… any possible combinations of those are possible. For instance, the following currency conversions will maintain their original cost against their converted currencies:

```
2014/05/01 Transfer from Canada
  Assets:CA:Checking              -500 CAD
  Assets:US:Checking               400 USD @ 1.25 CAD

2014/05/15 Transfers from Europe
  Assets:UK:Checking              -200 GBP
  Assets:US:Checking               500 USD @ 0.4 GBP

2014/05/21 Transfers from Europe
  Assets:DE:Checking              -100 EUR
  Assets:US:Checking               133.33 USD @ 0.75 EUR
```

This results in the following inventory in the Assets:US:Checking account:

```
$ ledger -f l2.lgr bal --lots US:Checking
500.00 USD {0.4 GBP} [2014/05/15]
133.33 USD {0.75 EUR} [2014/05/21]
400.00 USD {1.25 CAD} [2014/05/01]  Assets:US:Checking
```

The currencies are treated like stock. But nobody thinks of their currency balances like this, one normally thinks of currencies held in an account as units in and of themselves, not in terms of their price related to some other currency. In my view, this is confusing. After these conversions, I just think of the balance of that account as 1033.33 USD.

A possible rebuttal to this observation is that most of the time a user does not care about inventory lots for accounts with just currencies, so they just happen not print them out. Out of sight, out of mind. But there is no way to

distinguish when the rendering routine should render lots or not. If instructed to produce a report, a balance routine should ideally only render lots only for some accounts (e.g., investing accounts, where the cost of units matters) and not for others (e.g. accounts with currencies that were deposited sometimes as a result of conversions). Moreover, when I render a balance sheet, for units held at cost we need to report the book values rather than the number of shares. But for currencies, the currency units themselves need to get reported, always. If you don't distinguish between the two cases, how do you know which to render? I think the answer is to select which view you want to render using the options. The problem is that neither provides a single unified view that does the correct thing for both types of commodities. The user should not have to specify options to view partially incorrect views in one style or the other, this should be automatic and depend on whether we care about the cost of those units. This gets handled correctly if you distinguish between the two kinds of conversions.

But perhaps most importantly, this method does not particularly address the *conversion problem*: it is still possible to create money out of thin air by making conversions back and forth at different rates:

```
2014/05/01 Transfer to Canada
  Assets:US:Checking             -40000 USD
  Assets:CA:Checking              50000 CAD @ 0.80 USD

2014/05/15 Transfer from Canada
  Assets:CA:Checking             -50000 CAD @ 0.85 USD
  Assets:US:Checking              42500 USD
```

This results in a trial balance with just 2500 USD:

```
$ ledger -f l3.lgr bal
            2500 USD  Assets:US:Checking
```

This is not a desirable outcome. We should require that the trial balance always sum to zero (except for virtual transactions). After all, this is the aggregate realization of the elegance of the double-entry method: because all transactions sum to zero, the total sum of any subgroup of transactions also sums to zero… except it doesn't. In my view, any balance sheet that gets rendered should comply with the accounting equation, precisely.

In my explorations with Ledger, I was hoping that by virtue of its inventory tracking method it would somehow naturally deal with these conversions automatically and thus provide a good justification for keeping track of all units at cost, but I witness the same problem showing up. (The conversions issue has been a real bitch of a problem to figure out a solution for in Beancount, but it is working, and I think that the same solution could be applied to Ledger to adjust these sums, without changing its current booking method: automatically insert a special conversion entry at strategic points - when a balance sheet is rendered.) In the end, I have not found that tracking costs for every transaction

would appear to have an advantage over forgetting the costs for price conversions, the conversions problem is independent of this. I'd love to hear more support in favour of this design choice.

Now, because all conversions maintain their cost, Ledger needs to be quite liberal about booking, because it would be inconvenient to force the user to specify the original rate of conversion for each currency, in the case of commonly occurring currency conversions. This has an unfortunate impact on those conversions which we do want to hold at cost: arbitrary lot reductions are allowed, that is, reductions against lots that do not exist. For example, the following trade does not trigger an error in Ledger:

```
2014/05/01 Enter a position
  Assets:Investments:Stock        10 HOOL @ 500 USD
  Assets:Investments:Cash

2014/05/15 Sale of an impossible lot
  Assets:Investments:Stock       -10 HOOL {505 USD} @ 510 USD
  Assets:Investments:Cash
```

This results in an inventory with a long position of 10 HOOL and a short position of 10 HOOL (at a different cost):

```
$ ledger -f l4.lgr bal --lots stock
10 HOOL {USD500} [2014/05/01]
   -10 HOOL {USD505}  Assets:Investments:Stock
```

I think that the reduction of a 505 USD position of HOOL should not have been allowed; Beancount treats this as an error by choice.

Ledger clearly appears to support booking against an existing inventory, so surely the intention was to be able to reduce against existing positions. The following transactions do result in an empty inventory, for instance:

```
2014/05/01 Enter a position
  Assets:Investments:Stock      10 HOOL @ 500 USD
  Assets:Investments:Cash

2014/05/15 Sale of matching lot
  Assets:Investments:Stock     -10 HOOL {500 USD} [2014/05/01]  @ 510 USD
  Assets:Investments:Cash
```

With output:

```
$ ledger -f l5.lgr bal --lots
              USD100  Equity:Capital Gains
```

As you see, the HOOL position has been eliminated. (Don't mind the auto-inserted equity entry in the output, I believe this will be fixed in a pending patch.) However, both the date and the cost must be specified for this to work.

The following transactions results in a similar problematic long + short position inventory as mentioned above:

```
2014/05/01 Enter a position
  Assets:Investments:Stock      10 HOOL @ 500 USD
  Assets:Investments:Cash

2014/05/15 Sale of matching lot?
  Assets:Investments:Stock     -10 HOOL {500 USD} @ 510 USD
  Assets:Investments:Cash
```

With output:

```
$ ledger -f l6.lgr bal --lots
10 HOOL {USD500} [2014/05/01]
   -10 HOOL {USD500}  Assets:Investments:Stock
              USD100  Equity:Capital Gains
--------------------
10 HOOL {USD500} [2014/05/01]
   -10 HOOL {USD500}
              USD100
```

This is a bit surprising, I expected the lots to book against each other. I suspect this may be an unreported bug, and not intended behaviour.

Finally, the "{}" cost syntax is allowed be used on augmenting legs as well. The documentation points to these methods being equivalent. It results in an inventory lot that does not have a date associated with it, but the other leg is not converted to cost:

```
2014/05/01 Enter a position
  Assets:Investments:Stock      10 HOOL {500 USD}
  Assets:Investments:Cash
```

With output:

```
$ ledger -f l7.lgr bal --lots
                   0  Assets:Investments
   -10 HOOL {USD500}    Cash
    10 HOOL {USD500}    Stock
--------------------
                   0
```

I probably just don't understand how this is meant to be used; in Beancount the automatically computed leg would have posted a -5000 USD amount to the Cash account, not shares.

What I think is going on, is that Ledger (probably) accumulates all the lots without attempting to match them together to try to make reductions, and then at reporting time - and only then - it matches the lots together based on some reporting options:

- Group lots by both cost/price and date, using --lots
- Group lots by cost/price only, using --lot-prices
- Group lots by date only, using --lot-dates

It is not entirely obvious from the documentation how that works, but the behavior is consistent with this.

(If this is correct, I believe that to be a problem with its design: the mechanism by which an inventory reduction is booked to an existing set of lots should definitely not be a reporting feature. It needs to occur before processing reports, so that a single and definitive history of inventory bookings can be realized. If variants in bookings are supported - and I don't think this is a good idea - they should be supported before the reporting phase. In my view, altering inventory booking strategies from command-line options is a bad idea, the strategies in place should be fully defined by the language itself.)

Shortcomings in Beancount

Beancount also has various shortcomings, though different ones.

In contrast to Ledger, Beancount disambiguates between currency conversions and conversions "held at cost":

```
2014-05-01 * "Convert some Loonies to Franklins"
  Assets:CA:Checking          -6000 CAD
  Assets:Investments:Cash      5000 USD @ 1.2 CAD   ; Currency conversion

2014-05-01 * "Buy some stock"
  Assets:Investments:Stock        10 HOOL {500 USD} ; Held at cost
  Assets:Investments:Cash
```

In the first transaction, the Assets:Investment:Cash account results in an inventory of 5000 USD, with no cost associated to it. However, after the second transaction the Assets:Investment:Stock account has an inventory of 10 HOOL with a cost of 500 USD associated to it. This is perhaps a little bit more complex, and requires more knowledge from the user: there are two kinds of conversions and he has to understand and distinguish between these two cases, and this is not obvious for newcomers to the system. Some user education is required (I'll admit it *would* help if I wrote more documentation).

Beancount's method is also less liberal, it requires a strict application of lot reduction. That is, if you enter a transaction that reduces a lot that does not exist, it will output an error. The motivation for this is to make it difficult for the user to make a mistake in data entry. Any reduction of a position in an inventory has to match against exactly one lot.

There are a few downsides that result from this choice:

- It requires the users to always find the cost of the lot to match against. This means that automating the import of transactions requires manual intervention from the user, as he has to go search in the ledger to insert the matching lot. (Note that theoretically the import code could load up the ledger contents to list its holdings, and if unambiguous, could look for the cost itself and insert it in the output. But this makes the import code dependent on the user's ledger, which most import codes aren't doing). This is an important step, as finding the correct cost is required in order to correctly compute capital gains, but a more flexible method is desired, one that allows the user to be a bit more lazy and not have to put the cost of the existing position when the choice is obvious, e.g., when there is a single lot of that unit to match against. I'd like to relax this method somewhat.
- The strict requirement to reduce against an existing lot also means that both long and short positions of the same commodities "held at cost" in the same account cannot exist. This is not much of a problem in practice, however, because short positions are quite rare—few individuals engage in them—and the user could always create separate accounts to hold short positions if needed. In fact, it is already customary to create a dedicated subaccount for each commodity in an investment account, as it naturally organizes trading activity nicely (reports a bit of a mess otherwise, it's nice to see the total position value grouped by stock, since they offer exposure to different market characteristics). Different accounts should work well, as long as we treat the signs correctly in reductions, i.e., buying stock against an existing short position is seen as a position reduction (the signs are just reversed).

Another problem with the Beancount booking method is in how it matches against lots with dates. For example, if you were to try to disambiguate between two lots at the same price, the most obvious input would not work:

```
2012-05-01 * "First trade"
  Assets:Investments:Stock        10 HOOL {500 USD}
  Assets:Investments:Cash

2014-02-15 * "Second trade at very same price"
  Assets:Investments:Stock         8 HOOL {500 USD}
  Assets:Investments:Cash

2014-06-30 * "Trying to sell"
  Assets:Investments:Stock        -5 HOOL {500 USD / 2012-05-01}
  Assets:Investments:Cash
```

This would report a booking error. This is confusing. In this case Beancount, in its desire to be strict, applies strict matching against the inventory lots, and attempting to match units of a lot of (HOOL, 500 USD, 2012-05-01) with units of (HOOL, 500 USD, *None*) simply fails. Note that the lot-date syntax is accepted

by Beancount on both augmenting and reducing legs, so the "solution" is that the user is forced to specifically provide the lot-date on acquiring the position. This would work, for instance:

```
2012-05-01 * "First trade"
  Assets:Investments:Stock        10 HOOL {500 USD / 2012-05-01}
  Assets:Investments:Cash

2014-02-15 * "Second trade at very same price"
  Assets:Investments:Stock         8 HOOL {500 USD}
  Assets:Investments:Cash

2014-06-30 * "Trying to sell"
  Assets:Investments:Stock        -5 HOOL {500 USD / 2012-05-01}
  Assets:Investments:Cash
```

The inconvenience here is that when the user entered the first trade, he could not predict that he would enter another trade at the same price in the future and typically would not have inserted the lot-date. More likely than not, the user had to go back and revisit that transaction in order to disambiguate this. We can do better.

This is a shortcoming of its implementation, which reflects the fact that position reduction was initially regarded as an exact matching problem rather than a fuzzy matching one—it wasn't clear to me how to handle this safely at the time. This needs to be fixed after this proposal, and what I'm doing with this document is very much a process of clarifying for myself what I want this fuzzy matching to mean, and to ensure that I come up with an unambiguous design that is easy to enter data for. Ledger always automatically attaches the transaction date to the inventory lot, and in my view this is the correct thing to do (below we propose attaching more information as well).

Context

[Updated on Nov'2014]

It is important to distinguish between two types of booking:

1. **Booking.** The strict type of booking that occurs when we are considering the entire list of transactions. We will call this the "booking" process, and it should occur just once. This process should be strict: the failure of correctly book a reduction to an existing lot should trigger a fatal error.

2. **Inventory Aggregation.** When we are considering a subset of entries, it is possible that entries that add some lots are removed, and that other entries that reduce or remove those lots are kept. This creates a situation in which it is necessary to support aggregations over time of changes to an inventory that may not admit a matching lot. If we are to support

arbitrary subsets of transactions being aggregated, we must take this into account. This aggregation process should never lead to an error.

Let's take an example to justify why we need the non-booking summarization. Let's say we have a stock purchase and sale:

```
2013-11-03 * "Buy"
  Assets:Investments:VEA      10 AAPL {37.45 USD} ;; (A)
  Assets:Investments:Cash     -370.45 USD

2014-08-09 * "Sell"
  Assets:Investments:VEA     -5 AAPL {37.45 USD}
  Assets:Investments:Cash     194.40 USD
  Income:Investments:PnL
```

If we filter "by year" and only want to view transactions that occurred in 2014, AND we don't "close" the previous year, that is, we do not create opening balances entries that would deposit the lots in the account at the beginning of the year - this could happen if we filter by tag, or by some other combination of conditions - then the (A) lot is not present in the inventory to be debited from. We must somehow accommodate a -5 AAPL at that point. When reporting, the user could decide how to summarize and "match up as best it can" those legs, but converting them to units or cost, or convert them to a single lot at average-cost.

The booking process, however, should admit no such laxity. It should be strict and trigger an error if a lot cannot be matched. This would only be applied on the total set of transactions, and only once, never on any subset. The purpose of the booking stage should be threefold:

1. **Match** against existing lots and issue errors when that is impossible. Do this, using the method specified by the user.

2. **Replace** all partially specified lots by a full lot specification, the lot that was matched during the booking process.

3. **Insert links** on transactions that form a trade: a buy and corresponding sells should be linked together. This essentially identifies trades, and can then be used for reporting later on. (Trades at average cost will require special consideration.)

We can view Ledger's method as having an aggregation method only, lacking a booking stage. The particular method for aggregation is chosen using --lots or --lot-dates or --lot-prices. This would indicate that a "booking" stage could be tacked on to Ledger without changing much of its general structure: it could be inserted as a pre- or post- process, but it would require Ledger to sort the transactions by date, something it does not currently do (it runs a single pass over its data).

Beancount has had both methods for a while, but the separate nature of these

two operations has not been explicitly stated so far—instead, the inventory supported an optional flag to raise an error when booking was impossible. Also, inventory aggregation has not been thought as requiring much customization, it was meant to be unified to the booking process. It is now clear that they are two distinct algorithms, and that a few different methods for aggregating can be relevant (though they do not function the same as Ledger's do. Note: you can view the inventory aggregation method as a GROUP BY within an Inventory object's lots: which columns do you group over? This requires use cases). In any case, fully specified lots should match against each other by default: we need to support this as the degenerate case to use for simple currencies (not held at cost)... we would not want to accumulate all changes in the same currency as separate lots in an inventory, they need to cross each other as soon as they are applied.

We want to change this in order to make the code clearer: there should be a separate "booking" stage, provided by a **plugin**, which resolves the partially specific lot reduction using the algorithm outlined in this document, and a separate method for inventory aggregation should not bother with any of those details. In fact, the inventory aggregation could potentially simply become an accumulated list of lots, and the summarization of them could take place a posteriori, with some conceptual similarly to when Ledger applies its aggregation. Just using a simple aggregation becomes more relevant once we begin entering more specific data about lots, such as always having an acquisition date, a label, and possibly even a link to the entry that created the lot. In order to trigger summarization sensibly, functions to convert and summarize accumulated inventories could be provided in the shell, such as "UNITS(inventory)".

Precision for Interpolation

The interpolation capabilities will be extended to cover eliding a single number, any number, in any subset of postings whose "weights" resolve to a particular currency (e.g., "all postings with weights in USD"). The interpolation needs to occur at a particular precision. The interpolated number should be quantized automatically, and the number of fractional digits to quantize it to should be automatically inferred from the DisplayContext which is itself derived from the input file. Either the most common or the maximum number of digits witnessed in the file should be used.

For a use case, see this thread:

`On Mon, Nov 24, 2014 at 12:33 PM, ELI <eliptus@gmail.com> wrote:`

Requirements

The previous sections introduce the general problem of booking, and point out important shortcomings in both the Beancount and Ledger implementations.

This section will present a set of desires for a new and improved booking mechanism for command-line bookkeeping software implementations.

Here are reasonable requirements for a new method:

- **Explicit inventory lot selection** needs to be supported. The user should be able to indicate precisely which of the lots to reduce a position against, per-transaction. An account or commodity should not have to have a consistent method applied across all its trades.
- Explicit inventory lot selection should **support partial matching**, that is, in cases where the lots to match against are ambiguous, the user should be able to supply only minimal information to disambiguate the lots, e.g., the date, or a label, or the cost, or combinations of these. For instance, if only a single lot is available to match against, the user should be able to specify the cost as "{}", telling the system to book at cost, but essentially saying: "book against any lot." This should trigger an error only if there are multiple lots.
- A variety of informations should be supported to specify an existing lot, and they should be combinable:
  - The cost of acquisition of the position;
  - The date the position was acquired at;
  - A user-provided label.
- Cases where insufficient information is provided for explicit lot selection is available should admit for a **default booking method** to be invoked. The user should be able to specify **globally** and **per-account** what the default booking method should be. This paves the way for **implicit lot selection**. A degenerate method should be available that kicks off an error and requires the user to be explicit.
- **Average cost booking** needs to be supported, as it is quite common in retirement or tax-sheltered accounts. This is the default method used in Canadian investment accounts.
- **Cost basis adjustments** need to be supported as well. The problem should be able to be specified by providing a specific lot (or all of a commodity's lots at average cost) and a *dollar amount* to adjust the position by.
- **Stock splits** need to be able to **maintain some of the original attributes of the position**, specifically, the original trade date and the user-specified label, if one has been provided. This should allow a common syntax to specify an original trading lot when reducing a position after a split.
- Reducing a position needs to **always book against an existing lot**. I'm preserving the Beancount behavior here, which I've argued for previously,

as it works quite well and is a great method to avoid data entry mistakes. This naturally defines an invariant for inventories: *all positions of a particular commodity held at cost should be of the same sign in one inventory* (this can be used as a sanity check).

- **Bookings that change the sign of a number of units should raise an error**, unless an explicit exception is requested (and I'm not even convinced that we need it). Note again that bookings only occur for units held at cost, so currency conversions are unaffected by this requirement. Beancount has had this feature for a while, and it has proved useful to detect errors. For example, if an inventory has a position of 8 HOOL {500 USD} you attempt to post a change of -10 units to this lot, the resulting number of units is now negative: -2. This should indicate user error. The only use case I can think for allowing this is the trading of futures spreads and currencies, which would naturally be reported in the same account (a short position in currencies is not regarded as a different instrument in the same way that a short position would); this is the only reason to provide an exception, and I suspect that 99% of users will not need it.]

Debugging Tools

Since this is a non-trivial but greatly important part of the process of entering trading data, we should provide tools that list in detail the running balance for an inventory, including all of the detail of its lots.

Booking errors being reported should include sufficient context, that is:

- The transaction with offending posting
- The offending posting
- The current state of the inventory before the posting is applied, with all its lots
- The implicit booking method that is in effect to disambiguate between lots
- A detailed reason why the booking failed

Explicit vs. Implicit Booking Reduction

Another question that may come up in the design of a new booking method is whether we require the user to be *explicit* about whether he thinks this is an addition to a position, or a reduction. This could be done, for instance, by requiring a slightly different syntax for the cost, for example "{}" would indicate an addition and "[]" a reduction. I'm not convinced that it is necessary to make that distinction, maybe the extra burden on the user is not worth it, but it might be a way to cross-check an expectation against the actual calculations that occur. I'll drop the idea for now.

Design Proposal

This section presents a concrete description of a design that fulfills the previously introduced requirements. We hope to keep this as simple as possible.

Inventory

An inventory is simply a list of lot descriptors. Each inventory lot will be defined by:

`UNITS, (CURRENCY, COST-UNITS, COST-CURRENCY, DATE, LABEL)`

Fields:

- UNITS: the number of units of that lot that are held
- CURRENCY: the type of commodity held, a string
- COST-UNITS: the number in the price of that lot
- COST-CURRENCY: the pricing commodity of the cost
- DATE: the acquisition date of the lot
- LABEL: an arbitrary user-specified string used to identify this lot

If this represents a lot held at cost, after this processing, only the LABEL field should be optionally with a NULL value. Anyone writing code against this inventory could expect that all other values are filled in with non-NULL values (or are otherwise all NULL).

Input Syntax & Filtering

The input syntax should allow the specification of cost as any combination of the following informations:

- The **cost per unit**, as an amount, such as "500 USD"
- A **total cost**, to be automatically divided by the number of units, like this: {... +9.95 USD}. You should be able to use either cost per unit, total cost, or even combine the two, like this: {500 + 9.95 USD}. This is useful to enter commissions. This syntax also replaces the previous {{...}} syntax.
- The **lot-date**, as a YYYY-MM-DD date, such as "2014-06-20"
- A **label**, which is any non-numerical identifier, such as first-apple or some random uuid like "aa2ba9695cc7"
- A **special marker** "*" that indicates to book at the average cost of the inventory balance

These following postings should all be valid syntax:

```
...
  Assets:Investments:Stock        10 HOOL {500 USD}      ; cost
  Assets:Investments:Stock        10 HOOL {339999615d7a} ; label
  Assets:Investments:Stock        10 HOOL {2014-05-01}   ; lot-date
  Assets:Investments:Stock        10 HOOL {}             ; no information

... Combinations
  Assets:Investments:Stock        10 HOOL {500 USD, 2014-05-01}
  Assets:Investments:Stock        10 HOOL {2014-05-01, 339999615d7a}
```

Algorithm

All postings should be processed in a separate step after parsing, in the order of their date, against a running inventory balance for each account. The cost amount should become fully determined at this stage, and if we fail to resolve a cost, an error should be raised and the transaction deleted from the flow of directives (after the program loudly complaining about it).

When processing an entry, we should match and filter all inventory lots against all the filters that are provided by the user. In general, after filtering the lots with the user restrictions:

- If the set of lots is empty, an error should be raised;
- If there is a single lot, this lot should be selected (success case);
- If there are multiple lots, the **default implicitly booking method** for the corresponding account should be invoked.

Implicit Booking Methods

If there are multiple matching lots to choose from during booking, the following implicit booking methods could be invoked:

- STRICT: Select the only lot of the inventory. If there are more than one lots, raise an error.
- FIFO: Select the lot with the earliest date.
- LIFO: Select the lot with the latest date.
- AVERAGE: Book this transaction at average cost.
- AVERAGE_ONLY: Like AVERAGE but also trigger an aggregation on an addition to a position. This can be used to enforce that the only booking method for all lots is at the average cost.
- NONE: Don't perform any inventory booking on this account. Allow a mix of lots for the same commodity or positive and negative numbers in the inventory. (This essentially devolves to the Ledger method of booking.)

This method would *only* get invoked if disambiguation between multiple lots is required, after filtering the lots against the expression provided by the user. The STRICT method is basically the degenerate disambiguation which issues an error if there is any ambiguity and should be the default.

There should be a default method that applies to all accounts. The default value should be overridable. In Beancount, we would add a new "option" to allow the user to change this:

```
option "booking_method"  "FIFO"
```

The default value used in a particular account should be specifiable as well, because it is a common case that the method varies by account, and is fixed within the account. In Beancount, this would probably become part of the open directive, something like this:

```
2003-02-17 open Assets:Ameritrade:HOOL  HOOL  booking:FIFO
```

(I'm not sure about the syntax.)

Resolving Same Date Ambiguity

If the automatic booking method gets invoked to resolve an ambiguous lot reduction, e.g. FIFO, if there are multiple lots at the same date, something needs to be done to resolve which of the lots is to be chosen. The line number at which the transaction appears should be selected. For example, in the following case, a WIDGET Of 8 GBP would be selected:

```
2014-10-15 * "buy widgets"
  Assets:Inventory     10 WIDGET {} ;; Price inferred to 8 GBP/widget
  Assets:Cash         -80 GBP

2014-10-15 * "buy another widget"
  Assets:Inventory      1 WIDGET {} ;; Price inferred to 9 GBP/widget
  Assets:Cash          -9 GBP

2014-10-16 * "sell a widget"
  Assets:Cash           11 GBP
  Assets:Inventory      -1 WIDGET {} ;; Ambiguous lot
```

Dates Inserted by Default

By default, if an explicit date is not specified in a cost, the date of the transaction that holds the posting should be attached to the trading lot automatically. This date is overridable so that stock splits may be implemented by a transformation to a transaction like this:

```
2014-01-04 * "Buy some HOOL"
  Assets:Investments:Stock   10 HOOL {1000.00 USD}
  Assets:Investments:Cash
```

```
2014-04-17 * "2:1 split into Class A and Class B shares"
  Assets:Investments:Stock  -10 HOOL {1000.00 USD, 2014-01-04} ; reducing
  Assets:Investments:Stock   10 HOOL  {500.00 USD, 2014-01-04} ; augment
  Assets:Investments:Stock   10 HOOLL {500.00 USD, 2014-01-04} ; augment
```

Matching with No Information

Supplying no information for the cost should be supported and is sometimes useful: in the case of *augmenting* a position, if all the other legs have values specified, we should be able to automatically infer the cost of the units being traded:

```
2012-05-01 * "First trade"
  Assets:Investments:Stock            10 HOOL {}
  Assets:Investments:Cash       -5009.95 USD
  Expenses:Commissions              9.95 USD
```

In the case of *reducing* a position—and we can figure that out whether that is the case by looking at the inventory at the point of applying the transaction to its account balance—an empty specification should trigger the default booking method. If the method is STRICT, for instance, this would select a lot *only* if there is a single lot available (the choice is unambiguous), which is probably a common case if one trades infrequently. Other methods will apply as they are defined.

Note that the user still has to specify a cost of "{}" in order to inform us that this posting has to be considered "held at cost." This is important to disambiguate from a price conversion with no associated cost.

Reducing Multiple Lots

If a single trade needs to close multiple existing lots of an inventory, this can dealt with trivially by inserting one posting for each lot. I think this is a totally reasonable requirement. This could represent a single trade, for instance, if your brokerage allows it:

```
2012-05-01 * "Closing my position"
  Assets:Investments:Stock           -10 HOOL {500 USD}
  Assets:Investments:Stock           -12 HOOL {510 USD}
  Assets:Investments:Cash       12000.00 USD
  Income:Investments:Gains
```

Note: If the cost basis specification for the lot matches multiple lots of the inventory and the result is unambiguous, the lots will be correctly, selected. For example, if the ante-inventory contains just those two lots (22 HOOL in total), you should be able to have a single reducing posting like this:

```
2012-05-01 * "Closing my position"
```

```
  Assets:Investments:Stock           -22 HOOL {}
  Assets:Investments:Cash       12000.00 USD
  Income:Investments:Gains
```

and they will be both be included.

On the other hand, if the result is ambiguous (for example, if you have more than these two lots) the booking strategy for that account would be invoked. By default, this strategy will be "STRICT" which will generate an error, but if this account's strategy is set to "FIFO" (or is not set and the default global strategy is "FIFO"), the FIFO lots would be automatically selected for you, as per your wish.

Lot Basis Modification

Another idea is to support the modification of a specific lot's cost basis in a single leg. The way this could work, is by specifying the number of units to modify, and the "+ number currency" syntax would be used to adjust the cost basis by a specific number, like this:

```
2012-05-01 * "Adjust cost basis by 250 USD for 5 of the 500 USD units"
  Assets:Investments:Stock            ~5 HOOL {500 + 250 USD}
```

If the number of unit is smaller than the total number of units in the lot, split out the lot before applying the cost basis adjustment. *I'm not certain I like this.*

Not specifying the size could also adjust the entire position (I'm not sure if this makes sense as it departs from the current syntax significantly):

```
2012-05-01 * "Adjust cost basis by 250 USD for the 500 USD units"
  Assets:Investments:Stock            HOOL {500 + 250 USD}
```

In any case, all the fields other than the total cost adjustment would be used to select which lot to adjust.

Tag Reuse

We will have to be careful in the inventory booking to warn on reuse of lot labels. Labels should be unique.

Examples

Nothing speaks more clearly than concrete examples. If otherwise unspecified, we are assuming that the booking method on the Assets:Investments:Stock account is STRICT.

No Conflict

Given the following inventory lots:

```
21, (HOOL, 500, USD, 2012-05-01, null)
```

This booking should succeed:

```
2013-05-01 *
  Assets:Investments:Stock           -10 HOOL {}
  ...
```

This booking should fail (because the amount is not one of the valid lots):

```
2013-05-01 *
  Assets:Investments:Stock           -10 HOOL {520 USD}
  ...
```

This one too should fail (no currency matching lot):

```
2013-05-01 *
  Assets:Investments:Stock           -10 MSFT {80 USD}
  ...
```

So should this one (invalid date when a date is specified):

```
2013-05-01 *
  Assets:Investments:Stock           -10 HOOL {500 USD, 2010-01-01}
  ...
```

Explicit Selection By Cost

Given the following inventory:

```
21, (HOOL, 500, USD, 2012-05-01, null)
```

This booking should succeed:

```
2013-05-01 *
  Assets:Investments:Stock           -10 HOOL {510 USD}
  ...
```

This booking should fail if the stock account's method is STRICT (ambiguous):

```
2013-05-01 *
  Assets:Investments:Stock           -10 HOOL {500 USD}
  ...
```

This booking should succeed if the stock account's method is FIFO, booking against the lot at 2012-05-01 (earliest):

```
2013-05-01 *
  Assets:Investments:Stock           -10 HOOL {500 USD}
  ...
```

Explicit Selection By Date

Given the same inventory as previously, this booking should succeed:

```
2013-05-01 *
  Assets:Investments:Stock           -10 HOOL {2012-05-01}
  ...
```

This booking should fail if the method is STRICT (ambiguous)

```
2013-05-01 *
  Assets:Investments:Stock           -10 HOOL {2012-06-01}
  ...
```

Explicit Selection By Label

This booking should succeed, because there is a single lot with the “abc“ label:

```
2013-05-01 *
  Assets:Investments:Stock           -10 HOOL {abc}
  ...
```

If multiple lots have the same label, ambiguous cases may occur; with this inventory, for example:

```
32, (HOOL, 500, USD, 2012-06-01, "abc")
```

The same booking should fail.

Explicit Selection By Combination

With the initial inventory, this booking should succeed (unambiguous):

```
2013-05-01 *
  Assets:Investments:Stock           -10 HOOL {500 USD, 2012-06-01}
  ...
```

There is only one lot at a cost of 500 USD and at an acquisition date of 2012-06-01.

Not Enough Units

The following booking would be unambiguous, but would fail, because there aren't enough units of the lot in the inventory to subtract from:

```
2013-05-01 *
  Assets:Investments:Stock           -33 HOOL {500 USD, 2012-06-01}
  …
```

Redundant Selection of Same Lot

If two separate postings select the same inventory lot in one transaction, it should be able to work:

```
2013-05-01 *
  Assets:Investments:Stock           -10 HOOL {500 USD, 2012-06-01}
  Assets:Investments:Stock           -10 HOOL {abc}
  ...
```

Of course, if there are not enough shares, it should fail:

```
2013-05-01 *
  Assets:Investments:Stock           -20 HOOL {500 USD, 2012-06-01}
  Assets:Investments:Stock           -20 HOOL {abc}
  ...
```

Automatic Price Extrapolation

If all the postings of a transaction can have their balance computed, we allow a single price to be automatically calculated - this should work:

```
2014-03-15 * "Adjust cost basis from 500 USD to 510 USD"
  Assets:US:Invest:HOOL      -10.00 HOOL {500.00 USD}
  Assets:US:Invest:HOOL       10.00 HOOL {}
  Income:US:Invest:Gains    -340.51 USD
```

And result in the equivalent of:

```
2014-03-15 * "Adjust cost basis from 500 USD to 510 USD"
  Assets:US:Invest:HOOL      -10.00 HOOL {500.00 USD}
  Assets:US:Invest:HOOL       10.00 HOOL {534.51 USD}
  Income:US:Invest:Gains    -340.51 USD
```

Of course, preserving the original date of the lot should work too, so this should work as well:

```
2014-03-15 * "Adjust cost basis from 500 USD to 510 USD"
  Assets:US:Invest:HOOL      -10.00 HOOL {500.00 USD}
  Assets:US:Invest:HOOL       10.00 HOOL {2014-02-04}
  Income:US:Invest:Gains    -340.51 USD
```

Average Cost Booking

Augmenting lots with the average cost syntax should fail:

```
2014-03-15 * "Buying at average cost, what does this mean?"
  Assets:US:Invest:Stock      10.00 HOOL {*}
  Income:US:Invest:Gains   -5000.00 USD
```

Reducing should work, this is the main use case:

```
2014-03-15 * "Buying a first lot"
  Assets:US:Invest:Stock      10.00 HOOL {500.00 USD}
  Assert:US:Invest:Cash    -5000.00 USD

2014-04-15 * "Buying a second lot"
  Assets:US:Invest:Stock      10.00 HOOL {510.00 USD}
  Assets:US:Invest:Cash    -5100.00 USD

2014-04-28 * "Obtaining a dividend in stock"
  Assets:US:Invest:Stock         1.00 HOOL {520.00 USD}
  Income:US:Invest:Dividends  -520.00 USD

2014-05-20 * "Sell some stock at average cost"
  Assets:US:Invest:Stock      -8.00 HOOL {*}
  Assets:US:Invest:Cash     4240.00 USD
  Income:US:Invest:Gains    ; Automatically calculated: -194.29 USD
```

The effect would be to aggregate the 10 HOOL {500.00 USD} lot, the 10 HOOL {510.00 USD} lot, and the 1.00 HOOL {520.00 USD} lot, together into a single lot of 21 HOOL {505.714285 USD}, and to remove 8 HOOL from this lot, which is 8 HOOL x 505.714285 USD/HOOL = 4045.71 USD. We receive 4240.00 USD, which allows us to automatically compute the capital gain: 4240.00 - 4045.71 = 194.29 USD. After the reduction, a single lot of 13 HOOL {505.714285 USD} remains.

It should also work if we have multiple currencies in the account, that is adding the following transaction to the above should not make it faile:

```
2014-04-15 * "Buying another stock"
  Assets:US:Invest:Stock      15.00 AAPL {300.00 USD}
  Assets:US:Invest:Cash    -4500.00 USD
```

However it should fail if we have the same currency priced in distinct cost currencies in the same account:

```
2014-03-15 * "Buying a first lot"
  Assets:US:Invest:Stock      10.00 HOOL {500.00 USD}
  Assert:US:Invest:Cash    -5000.00 USD

2014-04-15 * "Buying a second lot"
  Assets:US:Invest:Stock      10.00 HOOL {623.00 CAD}
  Assets:US:Invest:Cash    -62300.00 CAD

2014-05-20 * "Sell some stock at average cost"
  Assets:US:Invest:Stock      -8.00 HOOL {*}  ; Which HOOL, USD or CAD?
  Assets:US:Invest:Cash    4240.00 USD
  Income:US:Invest:Gains
```

But of course, this *never* happens in practice, so I'm not too concerned. We

could potentially support specifying the cost-currency to resolve this case, that is, replacing the sell leg with this:

```
2014-05-20 * "Sell some stock at average cost"
  Assets:US:Invest:Stock      -8.00 HOOL {* USD}
  Assets:US:Invest:Cash    4240.00 USD
  Income:US:Invest:Gains
```

I'm quite sure it wouldn't be useful, but we could go the extra mile and be as general as we can possibly be.

Future Work

This proposal does not yet deal with cost basis re-adjustments! We need to way to be able to add or remove a fixed *dollar amount* to a position's cost basis, that is, from (1) a lot identifier and (2) a cost-currency amount, we should be able to update the cost of that lot. The transaction still has to balance.

Here are some ideas what this might look like:

```
2014-05-20 * "Adjust cost basis of a specific lot"
  Assets:US:Invest:Stock       10.00 HOOL {500.00 USD + 230.00 USD}
  Income:US:Invest:Gains     -230.00 USD
```

The resulting inventory would be 10 HOOL at 523.00 USD per share, and the rest of the original lots, less 10 HOOL at 500.00 USD per share (there might be some of that remaining, that's fine). This is the equivalent of

```
2014-05-20 * "Adjust cost basis of a specific lot"
  Assets:US:Invest:Stock      -10.00 HOOL {500.00 USD}
  Assets:US:Invest:Stock       10.00 HOOL {523.00 USD}
  Income:US:Invest:Gains     -230.00 USD
```

except you did not have to carry out the calculation. This should therefore be implemented as a transformation. However, this is not super useful, because this is already supported... adjustments are usually performed on lots at average cost, and this is where the problem lie: you'd have to make the calculation yourself! Here's a relevant use case:

```
2014-05-20 * "Adjust cost basis of a specific lot"
  Assets:US:Invest:Stock       10.00 HOOL {* + 230.00 USD}
  Income:US:Invest:Gains     -230.00 USD
```

This would first average all the lots of HOOL together and recompute the cost basis with the new amount. This should work:

```
2014-03-15 * "Buying a first lot"
  Assets:US:Invest:Stock       5.00 HOOL {500.00 USD}
  Assert:US:Invest:Cash    -2500.00 USD


2014-04-15 * "Buying a second lot"
```

```
  Assets:US:Invest:Stock        5.00 HOOL {520.00 USD}
  Assets:US:Invest:Cash     -2600.00 USD

2014-05-20 * "Adjust cost basis of the Hooli units"
  Assets:US:Invest:Stock       10.00 HOOL {* + 230.00 USD}
  Income:US:Invest:Gains     -230.00 USD
```

Notice how we stll have to specify an amount of HOOL shares to be repriced. I like this, but I'm wondering if we should allow specifying "all the units" like this (this would be a more radical change to the syntax):

```
2014-05-20 * "Adjust cost basis of the Hooli units"
  Assets:US:Invest:Stock           * HOOL {* + 230.00 USD}
  Income:US:Invest:Gains     -230.00 USD
```

Perhaps the best way to auto-compute cost basis adjustments would be via powerful interpolation, like this:

```
2014-05-20 * "Adjust cost basis of a specific lot"
  Assets:US:Invest:Stock      -10.00 HOOL {500 USD}  ; reduce
  Assets:US:Invest:Stock       10.00 HOOL {}         ; augment, interpolated
  Income:US:Invest:Gains     -230.00 USD
```

See the Smarter Elision document for more details on a proposal.

Inter-Account Booking

Some tax laws require a user to book according to a specific method, and this might apply between all accounts. This means that some sort of transfer needs to be applied in order to handle this correctly. See the separate document for detail.

TODO - complete this with more tests for average cost booking!

Implementation Notes

Separate Parsing & Interpolation

In order to implement a syntax for reduction that does not specify the cost, we need to make changes to the parser. Because there may be no costs on the posting, it becomes impossible to perform balance checks at parsing time. Therefore, we will need to postpone balance checks to a stage *after* parsing. This is reasonable and in a way nice: it is best if the Beancount parser not output many complex errors at that stage. A new "post-parse" stage will be added, right before running the plugins.

Conclusion

This document is work-in-progress. I'd really love to get some feedback in order to improve the suggested method, which is why I put this down in words instead

of just writing the code. I think a better semantic can be reached by iterating on this design to produce something that works better in all systems, and this can be used as documentation later on for developers to understand why it is designed this way.

Your feedback is much appreciated.

Settlement Dates & Transfer Accounts in Beancount

Martin Blais, July 2014

http://furius.ca/beancount/doc/proposal-dates

Motivation

When a trade executes in an investment account, there is most often a delay between the date that the transaction is carried out (the "transaction date") and the date that the funds are deposited in an associated cash account (the "settlement date"). This makes balance imported balance assertions sometimes requiring the fudging of their dates, and sometimes they can even be impossible. This document proposes the addition of an optional "settlement date" to be attached to a transaction or a posting, and associated semantics for how to deal with the problem.

Proposal Description

Settlement Dates

In the first implementation of Beancount I used to have two dates attached to a transaction, but I never did anything with them. The alternate date would get attached but was ignored thereafter. The meaning of it is that it should have split the transaction into two, with some sort of transfer account, that might have been useful semantics, I never developed it.

Something like this as input:

```
2014-06-23=2014-06-28 * "Sale"
   Assets:ScottTrade:GOOG    -10 GOOG {589.00 USD}
   S Assets:ScottTrade:Cash
```

where the "S" posting flag marks the leg "to be postponed to the settlement date."

Alternatively, you could attach the date to a posting:

```
2014-06-23 * "Sale"
   Assets:ScottTrade:GOOG    -10 GOOG {589.00 USD}
   2014-06-28 Assets:ScottTrade:Cash
```

Both of the above syntax proposals allow you to specify which postings are meant to be postponed to settlement. The second one is more flexible, as each posting could potentially have a different date, but the more constrained syntax of the first would create less complications.

Either of these could get translated to multiple transactions with a transfer account to absorb the pending amount:

```
2014-06-23 * "Sale" ^settlement-3463873948
   Assets:ScottTrade:GOOG    -10 GOOG {589.00 USD}
   Assets:ScottTrade:Transfer

2014-06-28 * "Sale" ^settlement-3463873948
   Assets:ScottTrade:Transfer
   Assets:ScottTrade:Cash        5890.00 USD
```

So far, I've been getting away with fudging the dates on balance assertions where necessary. I have relatively few sales, so this hasn't been a big problem so far. I'm not convinced it needs a solution yet, maybe the best thing to do is just to document how to deal with the issue when it occurs.

Maybe someone can convince me otherwise.

Transfer Accounts

In the previous section, we discuss a style whereby a single entry moving money between two accounts contains two dates and results in two separate entries. An auxiliary problem, which is related in its solution, is how to carry out the reverse operation, that is, how to *merge* two separate entries posting to a common transfer account (sometimes called a "suspense account").

For example, a user may want to input the two sides of a transaction separately, e.g. by running import scripts on separate input files, and instead of having to reconcile and merge those by hand, we would want to explicitly support this by identifying matching transactions to these transfer accounts and creating a common link between them.

Most importantly, we want to be able to easily identify which of the transactions is not matched on the other side, which indicates missing data.

- There is a prototype of this under beancount.plugins.tag_pending.
- Also see redstreet0's "zerosum" plugin from this thread.

Remaining Questions

How do we determine a proper transfer account name? Is a subaccount a reasonable approach? What if a user would like to have a single global limbo account?

`TODO`

Does this property solve the problem of making balance assertions between trade and settlement?

`TODO [Write out a detailed example]`

Any drawbacks?

`TODO`

How does this affect the balance sheet and income statement, if any? Is it going to be obvious to users what the amounts in these limbo/transfer accounts are?

`TODO`

Unrooting Transactions

A wilder idea would be to add an extra level in the transaction-posting hierarchy, adding the capability to group multiple partial transactions, and move the balancing rule to that level. Basically, two transactions input separately and then grouped - by some rule, or trivially by themselves - could form a new unit of balance rule.

That would be a much more demanding change on the schema and on the Beancount design but would allow to natively support partial transactions, keeping their individual dates, descriptions, etc. Maybe that's a better model? Consider the advantages.

Previous Work

Ledger Effective and Auxiliary Dates

Ledger has the concept of "auxiliary dates". The way these work is straightforward: any transaction may have a second date, and the user can select at runtime (with --aux-date) whether the main date or the auxiliary dates are meant to be used.

It is unclear to me how this is meant to be used in practice, in the presence of balance assertions. Without balance assertions, I can see how it would just

work: you'd render everything with settlement dates only. This would probably only make sense for specific reports.

I would much rather keep a single semantic for the set of transactions that gets parsed in; the idea that the meaning of the transactions varies depending on the invocation conditions would set a precedent in Beancount, I'd prefer not to break this nice property, so by default I'd prefer to avoid implementing this solution.

Auxiliary dates are also known as "effective dates" and can be associated with each individual posting. Auxiliary dates are secondary to the the "primary date" or the "actual date", being the posting date of the record):

```
2008/10/16 * (2090) Bountiful Blessings Farm
    Expenses:Food:Groceries                  $ 37.50  ; [=2008/10/01]
    Expenses:Food:Groceries                  $ 37.50  ; [=2008/11/01]
    Expenses:Food:Groceries                  $ 37.50  ; [=2008/12/01]
    Expenses:Food:Groceries                  $ 37.50  ; [=2009/01/01]
    Expenses:Food:Groceries                  $ 37.50  ; [=2009/02/01]
    Expenses:Food:Groceries                  $ 37.50  ; [=2009/03/01]
    Assets:Checking
```

The original motivation for this was for budgeting, allow one to move accounting of expenses to neighboring budget periods in order to carry over actual paid amounts to those periods. Bank amounts in one month could be set against a budget from the immediately preceding or following month, as needed. (Note: John Wiegley)

This is similar to one of the syntaxes I'm suggesting above—letting the user specify a date for each posting—but the other postings are not split as independent transactions. The usage of those dates is similarly triggered by a command-line option (--effective). I'm assuming that the posting on the checking account above occurs at once at 2008/10/16, regardless of reporting date. Let's verify this:

```
$ ledger -f settlement1.lgr reg checking --effective
08-Oct-16 Bountiful Blessings.. Assets:Checking          $ -225.00    $ -225.00
```

That's what I thought. This works, but a problem with this approach is that any balance sheet drawn between 2008/10/01 (the earliest effective date) and 2009/03/01 (the latest effective date) would not balance. Between those dates, some amounts are "in limbo" and drawing up a balance sheet at one of those dates would not balance.

This would break an invariant in Beancount: we require that you should always be able to draw a balance sheet at any point in time, and any subset of transactions should balance. I would rather implement this by splitting this example transaction into many other ones, as in the proposal above, moving those temporary amounts living in limbo in an explicit "limbo" or "transfer"

account, where each transaction balances. Moreover, this step can be implemented as a transformation stage, replacing the transaction with effective dates by one transaction for each posting where the effective date differs from the transaction's date (this could be enabled on demand via a plugin).

GnuCash

TODO(blais) - How is this handled in GnuCash and other GUI systems? Is there a standard account method?

References

The IRS requires you to use the trade date and NOT the settlement date for tax reporting; from the IRS Publication 17:

```
Securities traded on established market. For securities traded on an established securities
```

Threads

- An interesting "feature by coincidence"
- First Opinions, Coming from Ledger

Balance Assertions in Beancount

Martin Blais, July 2014

http://furius.ca/beancount/doc/proposal-balance

This document summarizes the different kinds of semantics for balance assertions in all command-line accounting systems and proposes new syntax for total and file-order running assertions in Beancount.

Motivation

Both Beancount and Ledger implement *balance assertions.* These provide the system with checkpoints it can use to verify the integrity of the data entry[1].

Traditional accounting principles state that a user may never change the past—correcting the past involves inserting new entries to undo past mistakes[2]—but we in the command-line accounting community take issue with that: we want to remain able to reconstruct the past and more importantly, to correct past mistakes at the site of their original data entry. Yes, we are dilettantes but we are bent on simplifying the process of bookkeeping by challenging existing concepts and think that this is perfectly reasonable, as long as it does not break known balances. These known balances are what we provide via balance assertions.

Another way to look at these balance assertions, is that they are simply the bottom line amounts reported on various account statements, as exemplified in the figure below.

Following this thread, we established that there were differing interpretations of balance assertions in the current versions of Ledger (3.0) and Beancount (2.0b).

Partial vs. Complete Assertions

An assertion in Beancount currently looks like this:

```
2012-02-04 balance Assets:CA:Bank:Checking    417.61 CAD
```

The meaning of this directive is: "please assert that the inventory of account 'Assets:CA:Bank:Checking' contains exactly 417.61 units of CAD at the end of February 3, 2012" (so it's dated on the 4th because in Beancount balance assertions are specified at the beginning of the date). It says nothing about other commodities in the account's inventory. For example, if the account contains units of "USD", those are unchecked. We will call this interpretation a **partial balance assertion** or **single-commodity assertion**.

An alternative assertion would be exhaustive: "please assert that the inventory of account 'Assets:CA:Bank:Checking' contains only 417.61 units of CAD at the end of February 3, 2012." We call this a **complete balance assertion**. In order to work, this second type of assertion would have to support the specification of the full contents of an inventory. This is currently not supported.

Further note that we are not asserting the cost basis of an inventory, just the number of units.

File vs. Date Assertions

There are two differing interpretations and implementations of the running balances for assertions:

- Beancount first sorts all the directives and verifies the balance at the *beginning* of the date of the directive. In the previous example, that is "*the balance before any transactions on 2012-02-04 are applied.*" We will call this **date assertions** or **date-based assertions**.
- Ledger keeps a running balance of each account's inventory during its parsing phase and performs the check *at the site of the assertion in the file.* We will call this **file assertions** or **file-order,** or **file-based assertions**. This kind of assertion has no date associated with it (this is slightly misleading in Ledger because of the way assertions are specified, as attached to a transaction's posting, which appears to imply that they occur on the transaction date, but they don't, they strictly apply to the file location).

Ordering & Ambiguity

An important difference between those two types of assertions is that file-based assertions are not order-independent. For example, take the following input file:

```
;; Credit card account
2014/05/01 opening
  Liabilities:CreditCard    $-1000.00
  Expenses:Opening-Balances

2014/05/12 dinner
  Liabilities:CreditCard    $-74.20
  Expenses:Restaurant

;; Checking account
2014/06/05 salary
  Assets:Checking            $4082.87
  Income:Salary

2014/06/05 cc payment
  Assets:Checking            $-1074.20 = $3008.67
  Liabilities:CreditCard     = $0
```

If you move the credit card payment up to the credit card account section, the same set of transactions fails:

```
;; Credit card account
2014/05/01 opening
  Liabilities:CreditCard    $-1000.00
  Expenses:Opening-Balances

2014/05/12 dinner
  Liabilities:CreditCard    $-74.20
  Expenses:Restaurant
```

```
2014/06/05 cc payment
  Assets:Checking            $-1074.20 = $3008.67
  Liabilities:CreditCard     = $0

;; Checking account
2014/06/05 salary
  Assets:Checking            $4082.87
  Income:Salary
```

This subtle problem could be difficult for beginners to understand. Moving the file assertion to its own, undated directive might be more indicative syntax of its semantics, something that would look like this:

```
balance Assets:Checking = $3008.67
```

The absence of a date indicates that the check is not applied at a particular point in time.

Intra-Day Assertions

On the other hand, date-based assertions, because of their order-independence, preclude intra-day assertions, that is, a balance assertion that occurs *between* two postings on the same account during the same day.

Beancount does not support intra-day assertions at the moment.

Note despite this shortcoming, this has not been much of a problem, because it is often possible to fudge the date where necessary by adding or removing a day, or even just skipping an assertion where it would be impossible (skipping assertions is not a big deal, as they are optional and their purpose is just to provide certainty. As long as you have one that appears some date after the skipped one, it isn't much of an issue).

It would be nice to find a solution to intra-day assertions for date-based assertions, however. One interesting idea would be to extend the semantics to apply the balance in file order within the set of all transactions directly before and after the balance check that occur on the same date as the balance check, for example, this would balance:

```
2013-05-05 balance Assets:Checking    100 USD

2013-05-20 * "Interest payment"
  Assets:Checking           12.01 USD
  Income:Interest

2013-05-20 balance Assets:Checking    121.01 USD

2013-05-20 * "Check deposit"
  Assets:Checking           731.73 USD
```

```
Assets:Receivable
```

The spell would be broken as soon as a directive would appear at a different date.

Another idea would be to always sort the balance assertions in file-order as the second sort key (after the date) and apply them as such. I'm not sure this would be easy to understand though.

Beginning vs. End of Day

Finally, just for completeness, it is worth mentioning that date assertions have to have well-defined semantics regarding *when* they apply during the day. In Beancount, they currently apply at the beginning of the day.

It might be worthwhile to provide an alternate version of date-based assertions that applies at the end of the day, e.g. "balance_end". Beancount v1 used to have this ("check_end") but it was removed in the v2 rewrite, as it wasn't clear it would be really needed. The simplicity of a single meaning for balance assertions is nice too.

Status

Ledger 3.0 currently supports only partial file-order assertions, on transactions.

Beancount 2.0 currently supports only partial date-based assertions at the beginning of the day.

Proposal

I propose the following improvements to Beancount's balance assertions.

File Assertions

File assertions should be provided as a plugin. They would look like this:

```
2012-02-03 file_balance Assets:CA:Bank:Checking    417.61 CAD
```

Ideally, in order to make it clear that they apply strictly in file order, they would not have a date, something like this:

```
file_balance Assets:CA:Bank:Checking    417.61 CAD
```

But this breaks the regularity of the syntax for all other directives. It also complicates an otherwise very regular and simple parser just that much more… *all* other directives begin with a date and a word, and all other lines are pretty much ignored. It would be a bit of a departure from this. Finally, it would still be nice to have a date just to insert those directives somewhere in the rendered journal. So I'm considering keeping a date for it. If you decide to use those odd assertions, you should know what they mean.

I also don't like the idea of attaching assertions to transactions; transaction syntax is already busy enough, it is calling to remain simple. This should be a standalone directive that few people use.

In order to implement this, the plugin would simply resort all the directives according to their file location only (using the fileloc attribute of entries), disregarding the date, and recompute the running balances top to bottom while applying the checks. This can entirely be done via post-processing, like date-based assertions, without disturbing any of the other processing.

Moreover, another advantage of doing this in a plugin is that people who don't use this directive won't have to pay the cost of calculating these inventories.

Complete Assertions

Complete assertions should be supported in Beancount by the current balance assertion directive. They aren't very important but are potentially useful.

Possible syntax ideas:

```
2012-02-03 balance* Assets:CA:Bank:Checking        417.61 CAD, 162 USD
```

I'm still undecided which is best. So far it seems a matter of taste.

[1] As far as we know, the notion of inputting an explicit expected amount is unique to command-line accounting systems. Other systems "reconcile" by freezing changes in the past.

[2] There are multiple reasons for this. First, in pre-computer times, accounting was done using books, and recomputing running balances manually would have involved making multiple annoying corrections to a book. This must have been incredibly inconvenient, and inserting correcting entries at the current time is a lot easier. Secondly, if your accounting balances are used to file taxes, changing some of the balances retroactively makes it difficult to go back and check the detail of reported amounts in case of an audit. This problem also applies to our context, but whether a past correction should be allowed is a choice that depends on the context and the particular account, and we leave it up to the user to decide whether it should be allowed.

Fund Accounting with Beancount

Martin Blais, Carl Hauser, August 2014

http://furius.ca/beancount/doc/proposal-funds

A discussion about how to carry out fund accounting within Beancount, various approaches, solutions and possible extensions.

Motivation

Multiple users are attempting to solve the problem of fund accounting using command-line accounting systems, partially because this type of accounting occurs in the context of non-profit organizations that have small budgets and would prefer to use free software, and partially because the flexibility and customization required appear to be a good fit for command-line bookkeeping systems.

What is Fund Accounting?

For example, see this thread:

```
"Another religious duty I compute is effectively tithing (we call it Huqúqu'lláh, and it's c
```

Here's another description, as a comment from another user:

```
[...] basically the idea that you split your financial life into separate pots called "funds
```

From the PowerChurchPlus 11.5 Manual (PowerChurch, Inc. 2013):

```
"In accounting, the term fund has a very specific meaning. An Accounting Fund is a self-bala
```

This is an interesting and apparently common problem. We will describe use cases in the rest of this section.

Joint Account Management

I have personally used this "fund accounting" idea to manage a joint account that I had with my ex-wife, where we would both hold individual accounts—we were both working professionals— and chip in to the joint account as needed. This section describes how I did this[1].

The accounting for the joint account was held in a separate file. Two subaccounts were created to hold each of our "portions":

```
2010-01-01 open Assets:BofA:Joint
2010-01-01 open Assets:BofA:Joint:Martin
2010-01-01 open Assets:BofA:Joint:Judie
```

Transfers to the joint account were directly booked into one of the two subaccount:

```
2012-09-07 * "Online Xfer Transfer from CK 345"
  Assets:BofA:Joint:Judie          1000.00 USD
  Income:Contributions:Judie
```

When we would incur expenses, we would reduce the asset account with two legs, one for each subaccount. We often booked them 2:1 to account for difference in income, or I just booked many of the transactions to myself (the fact that it was precisely accounted for does not imply that we weren't being generous to each other in that way):

```
2013-04-27 * "Urban Vets for Grumpy"
  Expenses:Medical:Cat           100.00 USD
  Assets:BofA:Joint:Martin          -50 USD
  Assets:BofA:Joint:Judie           -50 USD

2013-05-30 * "Takahachi"  "Dinner"
  Expenses:Food:Restaurant        65.80 USD
  Assets:BofA:Joint:Judie        -25.00 USD
  Assets:BofA:Joint:Martin
```

It was convenient to elide one of the two amounts, as we weren't being very precise about this.

Handling Multiple Funds

(Contributed from Carl Hauser)

Here's the model used in the PowerChurchPlus system that is mentioned above (replacing the account numbers it uses with Beancount-style names). "Fund" names are *prefixed* to the account names.

```
Operations:Assets:Bank:...
Endowment:Assets:Bank:...
Operations:Liabilities:CreditCard:...
Endowment:Liabilities:CreditCard:...
Operations:Income:Pledges:2014
Operations:Expenses:Salaries:...
Operations:Expenses:BuildingImprovement:...
Endowment:Income:Investments:...
Endowment:Expenses:BuildingImprovement:...
…
```

It is required that any transaction be balanced in every fund that it uses. For example, our Endowment fund often helps pay for things that are beyond the reach of current donations income.

```
2014-07-25 * "Bill's Audio" "Sound system upgrade"
   Endowment:Assets:Bank1:Checking        800.00 USD
   Operations:Assets:Bank1:Checking       200.00 USD
   Endowment:Expenses:BuildingImprovement:Sound -800.00 USD
   Operations:Expenses:BuildingImprovement:Sound -200.00 USD
```

This represents a single check to Bill's Audio paid from assets of both the Endowment and Operations funds that are kept in the single external assets account Assets:Bank1:Checking.

Note 1: An empty fund name could be allowed and the following ":" omitted, and in fact could be the default for people who don't want to use these features.

(i.e., nothing changes if you don't use these features.) The Fund with the empty string for its name is, of course, distinct from all other Funds.

Note 2: balance and pad directives are not very useful with accounts that participate in more than one Fund. Their use would require knowing the allocation of the account between the different funds and account statements from external holders (banks, e.g.) will not have this information. It might be useful to allow something like

```
2014-07-31 balance *:Assets:Bank1:Checking     579.39 USD
```

as a check that things were right, but automatically correcting it with pad entries seems impossible.

A balance sheet report can be run on any Fund *or any combination of Funds* and it will balance. You can keep track of what is owned for each different purpose easily. Transfers between funds are booked as expenses and decreases in the assets of one fund and income and increases in assets of the other. The income and expense accounts used for transfers may be generic (Operations:Income:Transfer) or you can use accounts set up for a particular kind of income or expense (Endowment:Expense:BuildingImprovement:Sound) would be fine as one leg of a transfer transaction.

The account name syntax here is just one way it might work and relies on Beancount's use of five specifically-named top-level accounts. Anything to the left of the first of those could be treated as a fund name, or a different separator could be used between the fund part and the account name part. Similarly, I've only shown single-level fund names but they might be hierarchical as well. I'm not sure of the value of that, but observe that if transactions balance at the leaf-level funds they will also balance for funds higher in the hierarchy and there might be some mileage there.

For John W.'s *Huqúqu'lláh* example one might set up a Fund whose liabilities were "moral obligations" rather than legal ones (that seems to be the objection to simply tracking the tithes in an ordinary liability account). As income comes in (say, direct deposited in a real bank checking account), book 19% of it to the "moral obligation" fund's checking account with a matching liability. When needful expenses are made, take back 19% from the "moral obligation" fund's checking account and reduce the liability. No virtual postings or transactions -- everything must balance. This would work well if for example we were to have a HisRetirement fund and a HerRetirement fund -- useful to have separate for estate planning purposes -- but more commonly we want to know about our combined retirement which could be defined to be a *virtual* fund OurRetirement equal to the sum of HisRetirement and HerRetirement. Note that this only matters when creating reports: there is no need to do anything except normal, double-entry booking with balanced transactions in each *real* fund. When I say the "sum" of two funds I mean a combination of taking the union of the contained account names, after stripping off the fund names, then summing the balances of the common accounts and keeping the balances of the others.

(Balance Sheet) For reporting, one wants the capability for balance by fund and balance summed over a set of funds. I also use a report that shows a subset of funds, one per column, with corresponding account names lined up horizontally and a column at the right that is the "sum". When all funds are included in this latter report you get a complete picture of what you own and owe and what is set aside for different purposes, or restricted in different ways. Here's a small sample of a balance sheet for a subset of the church Funds. The terminology is a little different: what Beancount calls Equity is here Net Assets. And because using large numbers of Funds in PowerChurchPlus is not so easy, the NetAssets are actually categorized for different purposes -- this is where I think the ideas we're exploring here can really shine: if Funds are really easy to create and combine for reporting then some of the mechanisms that PCP has for divvying up assets within a fund become unnecessary.

(Income Statement) For the Income and Expense report, usually I only care about expenses in one Fund at a time, but adding two funds together makes perfect sense if that's what you need to do.

For me, this approach to fund accounting is appealing because it relies on and preserves the fundamental principles of double-entry bookkeeping: when transactions sum to 0 the balance sheet equation is always true. Out of this we automatically get the ability to combine *any set of funds* (we don't have to do anything special when entering transactions or organizing the deep structure of the accounts) and have it make at least arithmetical sense, and we don't rely on any "magic" associated with renaming or tagging. I don't see how this can be so easily or neatly achieved by pushing the idea of the "funds" down into the account hierarchy: funds belong *above* the five root accounts (Assets, Liabilities, Equity, Income and Expenses), not below them.

Ideas for Implementation

Some random ideas for now. This needs a bit more work.

- If multiple redundant postings are required, the generation of these can be **automated using a plugin**. For instance, if a technique similar to mirror accounting is used in order to "send the same dollars to multiple accounts", at least the user should not have to do this manually, which would be both tedious and prone to errors.
- A procedure to **rename accounts** upon parsing could be used, in order to merge multiple files into one. (Allowing the user to install such a mapping is an idea I've had for a while but never implemented, though it could be implemented by a plugin filter.)
- We can rely on the fact that the transactions of **subaccounts may be joined and summed in a parent account** (despite the fact that reporting is lagging behind in that matter at the moment. It will be implemented eventually).

- Building off the earlier remark about doing something similar to the tag stack for Funds. What if the current tag architecture were extended to allow tags to have a value, #fund=Operations, or #fund=Endowment. Call them value-tags. You would also need to allow postings to have tags. Semantics of value-tags would be that they could only occur once for any posting, that a tag explicitly on a posting overrides the value-tag inherited from the transaction, and that an explicit tag on a transaction overrides a value from the tag-stack, and that only the last value-tag (with a particular key) in the tag-stack is applied to a transaction. This makes Funds a little less first-class than the earlier proposal to stick them in front of account names, but gets around the minor parsing difficulty previously mentioned. It suggests that opening of accounts within funds is not necessary where the previous syntax suggests that it is. The strict balancing rule for each fund in a transaction can still be implemented as a plugin. And reporting for a fund (or sum of funds) looks like:
  - select transactions with any posting matching the desired fund (or funds)
  - collect (and sum if necessary in the obvious way) postings associated with the fund (or funds) being reported on (A)
  - collect (and sum in the obvious way) postings from the selected transactions not associated with the desired fund (B)
  - Format a report with (A) as the information for the desired fund or funds and (B) as OTHER. OTHER is needed to make sure that the report balances, but could be omitted by choice.

`From an implementation perspective this seems more orthogonal to the current status quo, rec`

Examples

(Carl) Here is an example of how I might try to handle things associated with my paycheck, which involves deferred compensation (403(b) -- extremely similar to a 401(k)) and a Flexible Spending Account (somewhat similar to an HSA which has been discussed previously on the ledger-cli group).

Without Funds

(Carl) First, without Funds (this passes a bean-check):

This is how I might set up my pay stub with health insurance, an FSA and 403b contributions in the absence of Funds. One problem is that it distributes Gross Income directly into the 403b and FSA accounts even though it recognizes that the health insurance contribution is a reduction in salary which the 403b and FSA ought to be as well. So tracking contributions to both of those is made more difficult as well as tracking taxable income.

By thinking hard we could fix this -- we would come up with Income and Expense type accounts to represent the contributions, but they would end up looking kind of silly (in my opinion) because they are entirely internal to the accounts system.

See the next example for how it would look using Funds. If you stick out your tongue, rub your tummy and stand on your head you will see that the Funds-based solution is equivalent to what we would have come up with in the paragraph above in terms of the complexity of its transactions -- just as many lines are required. The advantage is primarily a mental one -- it is much easier to see what to do to be both consistent and useful.

```
option "title" "Paystub - no funds"

2014-07-15 open Assets:Bank:Checking
2014-07-15 open Assets:CreditUnion:Saving
2014-07-15 open Assets:FedIncTaxDeposits
2014-07-15 open Assets:Deferred:R-403b
2014-07-15 open Expenses:OASI
2014-07-15 open Expenses:Medicare
2014-07-15 open Expenses:MedicalAid
2014-07-15 open Expenses:SalReduction:HealthInsurance
2014-07-15 open Income:Gross:Emp1
2014-07-15 open Income:EmplContrib:Emp1:Retirement
2014-01-01 open Assets:FSA

; This way of setting up an FSA looks pretty good. It recognizes the
; rule that the designated amount for the year is immediately available
; (in the Asset account), and that we are obliged to make contributions
; to fund it over the course of the year (the Liability account).
2014-01-01 open Liabilities:FSA
2014-01-01 ! "Set up FSA for 2014"
  Assets:FSA                                                    2000 USD
  Liabilities:FSA                                              -2000 USD

2014-07-15 ! "Emp1 Paystub"
  Income:Gross:Emp1                                            -6000 USD
  Assets:Bank:Checking                                          3000 USD
  Assets:CreditUnion:Saving                                     1000 USD
  Assets:FedIncTaxDeposits                                       750 USD
  Expenses:OASI                                                  375 USD
  Expenses:Medicare                                              100 USD
  Expenses:MedicalAid                                             10 USD
  Assets:Deferred:R-403b                                         600 USD
  Liabilities:FSA                                                 75 USD
  Expenses:SalReduction:HealthInsurance                           90 USD
  Income:EmplContrib:Emp1:Retirement                            -600 USD
```

```
  Assets:Deferred:R-403b

2014-01-01 open Expenses:Medical
2014-07-20 ! "Direct expense from FSA"
  Expenses:Medical                                                  25 USD
  Assets:FSA

2014-07-20 ! "Medical expense from checking"
  Expenses:Medical                                                  25 USD
  Assets:Bank:Checking

2014-07-20 ! "Medical expense reimbursed from FSA"
  Assets:Bank:Checking                                              25 USD
  Assets:FSA
```

Using Funds

(Carl) And now using Funds (uses proposed features and hence can't be checked by bean-check):

This is how I might set up my pay stub with health insurance, an FSA and 403b contributions using Funds. I can straightforwardly arrange things so that contributions to the FSA and 403b are recognized as salary reductions for income tax purposes. And I can easily see how much I have contributed to the 403b and how much my employer has contributed.

See the previous example for how it would look without using Funds and which is not as accurate.

What this does NOT do is track taxes ultimately owed on the 403b money. I think that is a matter of one's decision about whether to do cash-basis or accrual-basis accounting. If cash basis those taxes are not a current liability and cannot be reported as such. If accrual basis, they are a current liability and need to be recorded as such when the income is booked.

For cash-basis books, we'd want the ability to report the state of affairs as if taxes were owed, but that is a matter for reporting rather than booking. We need to make sure we have enough identifiable information to automate creating those reports. I believe that taking a Fund-based accounting perspective easily does this.

A problem not solved: what if your basis is different for Federal and State purposes, or even for Federal and multiple different states. Yikes!

I've used the convention that the Fund name precedes the root account name. Note that with appropriate standing on one's head along with pivoting rules you can put the Fund name anywhere. Putting it first emphasizes that it identifies a set of accounts that must balance, and it makes it easy for the txn processor

to guarantee this property. Accounts without a fund name in front belong to the Fund whose name is the empty string.

```
option "title" "Paystub - no Funds"

2014-07-15 open Assets:Bank:Checking
2014-07-15 open Assets:CreditUnion:Saving
2014-07-15 open Assets:FedIncTaxDeposits
2014-07-15 open Expenses:OASI
2014-07-15 open Expenses:Medicare
2014-07-15 open Expenses:MedicalAid
2014-07-15 open Expenses:SalReduction:HealthInsurance
2014-07-15 open Expenses:SalReduction:FSA
2014-07-15 open Expenses:SalReduction:R-403b
2014-07-15 open Income:Gross:Emp1
2014-07-15 open Income:EmplContrib:Emp1:Retirement
2014-01-01 open FSA:Assets                      ; FSA fund accounts
2014-01-01 open FSA:Income:Contributions
2014-01-01 open FSA:Expenses:Medical
2014-01-01 open FSA:Expenses:ReimburseMedical
2014-01-01 open FSA:Liabilities
2014-07-15 open Retirement403b:Assets:CREF      ; Retirement fund accounts
2014-07-15 open Retirement403b:Income:EmployeeContrib
2014-07-15 open Retirement403b:Income:EmployerContrib
2014-07-15 open Retirement403b:Income:EarningsGainsAndLosses

; This implements the same idea as above for the FSA, of balancing
; Assets and Liabilities at the opening, but now does it using a
; separate Fund.
2014-01-01 ! "Set up FSA for 2014"
  FSA:Assets                                                    2000 USD
  FSA:Liabilities                                              -2000 USD

2014-07-15 ! "Emp1 Paystub"
  Income:Gross:Emp1                                            -6000 USD
  Assets:Bank:Checking                                          3000 USD
  Assets:CreditUnion:Saving                                     1000 USD
  Assets:FedIncTaxDeposits                                       750 USD
  Expenses:OASI                                                  375 USD
  Expenses:Medicare                                              100 USD
  Expenses:MedicalAid                                             10 USD
  Expenses:SalReduction:R-403b                                   600 USD
  Retirement403b:Income:EmployeeContrib                         -600 USD
  Retirement403b:Assets:CREF                                     600 USD
  Expenses:SalReduction:FSA                                       75 USD
  FSA:Income:Contributions                                       -75 USD
```

```
  FSA:Liabilities
  Expenses:SalReduction:HealthInsurance
  Retirement403b:Income:EmployerContrib                          -600 USD
  Retirement403b:Assets:CREF

2014-01-01 open Expenses:Medical
2014-07-20 ! "Direct expense from FSA"
  FSA:Expenses:Medical                                            25 USD
  FSA:Assets

2014-07-20 ! "Medical expense from checking"
  Expenses:Medical                                                25 USD
  Assets:Bank:Checking

2014-07-20 ! "Medical expense reimbursed from FSA"
  Assets:Bank:Checking                                            25 USD
  Income:ReimburseMedical
  FSA:Assets                                                     -25 USD
  FSA:Expenses:ReimburseMedical
```

Transfer Accounts Proposal

One problem that I've experienced using the Fund approach is that it's a bit too easy to make mistakes when transferring money between funds, such as in the very last transaction above. Formalizing the idea of Transfer accounts can help with this. The most common mistake is to end up with something that moves assets in both accounts in the same direction -- both up or both down as in this mistaken version of the transaction in question:

```
2014-07-20 ! "Medical expense reimbursed from FSA - with mistake"
 Assets:Bank:Checking                          25 USD
 Income:ReimburseMedical
 FSA:Assets                                    25 USD
 FSA:Expenses:ReimburseMedical
```

This balances but isn't what we intended. Suppose we add the idea of Transfer accounts. They live at the same place in the hierarchy as Income and Expenses and like those are non-balance-sheet accounts. But they really come into play only for transactions that involve multiple funds. There is an additional rule for transactions containing Transfer accounts: the sum of the transfers must also be zero (additional in the sense that the rule about transactions balancing within each fund is still there). So to use this we set things up a little differently:

```
2014-07-15 open FSA:Transfer:Incoming:Contribution
2014-07-15 open FSA:Transfer:Outgoing:ReimburseMedical
2014-07-15 open Transfer:Outgoing:FSAContribution
2014-07-15 open Transfer:Incoming:ReimburseMedical
```

The incorrect transaction is now flagged because sum of the transfers is -50 USD, not zero.

```
2014-07-20 ! "Medical expense reimbursed from FSA - with mistake"
  Assets:Bank:Checking                                    25 USD
  Transfer:Incoming:ReimburseMedical
  FSA:Assets                                              25 USD
  FSA:Transfer:Outgoing:ReimburseMedical
```

The paycheck transaction using transfer accounts for the FSA and the retirement account amounts might look like this (after appropriate opens of course):

```
2014-07-15 ! "Emp1 Paystub - using transfer accounts"
  Income:Gross:Emp1                                     -6000 USD
  Assets:Bank:Checking                                   3000 USD
  Assets:CreditUnion:Saving                              1000 USD
  Assets:FedIncTaxDeposits                                750 USD
  Expenses:OASI                                           375 USD
  Expenses:Medicare                                       100 USD
  Expenses:MedicalAid                                      10 USD
  Transfer:Outgoing:SalReduction          600 USD
  Retirement403b:Transfer:Incoming:EmployeeContrib
  Retirement403b:Assets:CREF                              600 USD
  Transfer:Outgoing:SalReduction                       75 USD
  FSA:Transfer:Incoming:Contributions
  FSA:Liabilities
  Expenses:SalReduction:HealthInsurance
  Retirement403b:Income:EmployerContrib                  -600 USD
  Retirement403b:Assets:CREF
```

Some might think that this is too complicated. Without changing the Transfer accounts idea or rule, you can simplify booking to just a single account per fund, Fund:Transfer, losing some ability for precision in reporting but without losing the ability to check correctness of transfer transactions.

Account Aliases

Simon Michael mentions that this is related to HLedger account aliases:

> "I think this is related to the situation where you want to view entities' finances both separately and merged. Account aliases can be another way to approximate this, as inhttp://hledger.org/how-to-use-account-aliases."

[1] If you find yourself culturally challenged by our modern lifestyle, perhaps you can imagine the case of roommates, although I don't like the reductionist view this association brings to my mind.

Proposal: Rounding & Precision in Beancount

Martin Blais, October 2014

This document describes the problem of rounding errors on Beancount transactions and how they are handled. It also includes a proposal for better handling precision issues in Beancount.

Motivation

Balancing Precision

Balancing transactions cannot be done precisely. This has been discussed on the Ledger mailing-list before. It is necessary to allow for some tolerance on the amounts used to balance a transaction.

This need is clear when you consider that inputting numbers in a text file implies a limited decimal representation. For example, if you're going to multiply a number of units and a cost, say both written down with 2 fractional digits, you might end up with a number that has 4 fractional digits, and then you need to compare that result with a cash amount that would typically be entered with only 2 fractional digits, something like this example:

```
2014-05-06 * "Buy mutual fund"
  Assets:Investments:RGXGX       4.27 RGAGX {53.21 USD}
  Assets:Investments:Cash     -227.21 USD
```

If you calculate it, the first posting's precise balance amount is 227.2067 USD, not 227.21 USD. However, the broker company managing the investment account will apply rounding to the closest cent for the cash withdrawal, and the rounded amount is the correct one to be used. This transaction has to balance; we need to allow for some looseness somehow.

The great majority of the cases where mathematical operations occur involve the conversion from a number of units and a price or a cost to a corresponding cash value (e.g., units x cost = total cost). Our task in representing transactional information is the replication of operations that take place mostly in institutions. These operations always involve the rounding of numbers for units and currencies (banks do apply stochastic rounding), and the *correct* numbers to be used from the perspective of these institutions, and from the perspective of the government, are indeed the *rounded* numbers themselves. It is a not a question of mathematical purity, but one of practicality, and our system should do the same that banks do. Therefore, I think that we should always post the rounded numbers to accounts. Using rational numbers is not a limitation in that sense, but we must be careful to store rounded numbers where it matters.

Automatic Rounding

Another related issue is that of automatically rounding amounts for interpolated numbers. Let's take our original problematic example again:

```
2014-05-06 * "Buy mutual fund"
  Assets:Investments:RGXGX       4.27 RGAGX {53.21 USD}
  Assets:Investments:Cash     ;; Interpolated posting = -227.2067 USD
```

Here the amount from the second posting is interpolated from the balance amount for the first posting. Ideally, we should find a way to specify how it should round to 2 fractional digits of precision.

Note that this affects interpolated prices and costs too:

```
2014-05-06 * "Buy mutual fund"
  Assets:Investments:RGXGX       4.27 RGAGX {USD}
  Assets:Investments:Cash     -227.21 USD
```

Here, the cost is intended to be automatically calculated from the cash legs: 227.21 / 4.27 = 53.2107728337… USD. The correct cost to be inferred is also the rounded amount of 53.21 USD. We would like a mechanism to allow us to infer the desired precision.

This mechanism cannot unfortunately be solely based on commodity: different accounts may track currencies with different precisions. As a real-world example, I have a retail FOREX trading account that really uses 4 digits of precision for its prices and deposits.

Precision of Balance Assertions

The precision of a balance assertions is also subject to this problem, assertions like this one:

```
2014-04-01 balance Assets:Investments:Cash   4526.77 USD
```

The user does not intend for this balance check to precisely sum up to 4526.77000000… USD. However, it this cash account previously received a deposit with a greater precision as in the previous section's example, then we have a problem. Now the cash amount contains some of the crumbs deposited from the interpolation (0.0067 USD). If we were able to find a good solution for the automatic rounding of postings in the previous section, this would not be a problem. But in the meantime, we must find a solution.

Beancount's current approach is a kludge: it uses a user-configurable tolerance of 0.0150 (in any unit). We'd like to change this so that the tolerance used is able to depend on the commodity, the account, or even the particular directive in use.

Other Systems

Other command-line accounting systems differ in how they choose that tolerance:

- Ledger attempts to automatically derive the precision to use for its balance checks by using recently parsed context (in file order). The precision to be used is that of the last value parsed for the particular commodity under consideration. This can be problematic: it can lead to unnecessary side-effects between transactions which can be difficult to debug.
- HLedger, on the other hand, uses global precision settings. The whole file is processed first, then the precisions are derived from the most precise numbers seen in the entire input file.
- At the moment, Beancount uses a constant value for the tolerance used in its balance checking algorithm (0.005 of any unit). This is weak and should, at the very least, be commodity-dependent, if not also dependent on the particular account in which the commodity is used.

Proposal

Automatically Inferring Tolerance

Beancount should derive its precision using a method entirely *local* to each transaction, perhaps with a global value for defaults. That is, for each transaction, it will inspect postings with simple amounts (no cost, no price) and infer the precision to be used for tolerance as half of that of the most precise amount entered by the user on this transaction. For example:

```
2014-05-06 * "Buy mutual fund"
  Assets:Investments:RGXGX       4.278  RGAGX {53.21 USD}
  Assets:Investments:Cash     -227.6324 USD
  Expenses:Commissions           9.95   USD
```

The number of digits for the precision to be used here is the maximum of the 2nd and 3rd postings, that is, $\max(4,\ 2) = 4$. The first postings is ignored because its amount is the result of a mathematical operation. The tolerance value should be *half* of the most precise digit, that is 0.00005 USD. This should allow the user to use an arbitrary precision, simply by inserting more digits in the input.

Only fractional digits will be used to derive precision… an integer should imply exact matching. The user could specify a single trailing period to imply a sub-dollar precision. For example, the following transaction should fail to balance because the calculated amount is 999.999455 USD:

```
2014-05-06 * "Buy mutual fund"
  Assets:Investments:RGXGX       23.45 RGAGX {42.6439 USD}
  Assets:Investments:Cash        -1000 USD
```

Instead, the user should explicitly allow for some tolerance to be used:

```
2014-05-06 * "Buy mutual fund"
  Assets:Investments:RGXGX       23.45 RGAGX {42.6439 USD}
  Assets:Investments:Cash       -1000. USD
```

Or better, use 1000.00 USD. This has the disadvantage that is prevents the user from specifying the simpler integer amount. I'm not sure if this is a big deal.

Finally, no global effect implied by transactions will be applied. No transaction should ever affect any other transaction's balancing context.

Inference on Amounts Held at Cost

An idea from by Matthew Harris (here) is that we could also use the value of the to the smallest decimal of the number of units times the cost as a number to use in establishing the tolerance for balancing transactions. For example, in the following transaction:

```
2014-05-06 * "Buy mutual fund"
  Assets:Investments:RGXGX       23.45 RGAGX {42.6439 USD}
  ...
```

The tolerance value that could be derived is

0.01 RGAGX x 42.6439 USD = **0.426439 USD**

The original use case presented by Matthew was of a transaction that did not contain a simple amount, just a conversion with both legs held at cost:

```
2011-01-25 * "Transfer of Assets, 3467.90 USD"
  * Assets:RothIRA:Vanguard:VTIVX  250.752 VTIVX {18.35 USD} @ 13.83 USD
  * Assets:RothIRA:DodgeCox:DODGX  -30.892 DODGX {148.93 USD} @ 112.26 USD
```

I've actually tried to implement this and the resulting tolerances are either unacceptably wide or unacceptably small. **It does not work well in practice so I've abandoned the idea.**

Automated Rounding

For values that are automatically calculated, for example, on auto-postings where the remaining value is derived automatically, we should consider rounding the values. No work has been done on this yet; these values are currently not rounded.

Fixing Balance Assertions

To fix balance assertions, we will derive the required precision by the number of digits used in the balance amount itself, by looking at the most precision fractional digit and using half of that digit's value to compute the tolerance:

```
2014-04-01 balance Assets:Investments:Cash   4526.7702 USD
```

This balance check implies a precision of 0.00005 USD.

If you use an integer number of units, no tolerance is allowed. The precise number should match:

```
2014-04-01 balance Assets:Investments:Cash   4526 USD
```

If you want to allow for sub-dollar variance, use a single comma:

```
2014-04-01 balance Assets:Investments:Cash   4526. USD
```

This balance check implies a precision of 0.50 USD.

Approximate Assertions

Another idea, proposed in this ticket on Ledger, proposes an explicitly approximate assertion.

We could implement it this way (just an idea):

```
2014-04-01 balance Assets:Investments:Cash   4526.00 +/- 0.05 USD
```

Accumulating & Reporting Residuals

In order to explicitly render and monitor the amount of rounding errors that occur in a Ledger, we should accumulate it to an Equity account, such as "Equity:Rounding". This should be turned on optionally. It should be possible for the user to specify an account to be used to accumulate the error. Whenever a transaction does not balance exactly, the residual, or rounding error, will be inserted as a posting of the transaction to the equity account.

By default, this accumulation should be turned off. It's not clear whether the extra postings will be disruptive yet (if they're not, maybe this should be turned on by default; practice will inform us).

Implementation

The implementation of this proposal is documented here.