Importing External Data in Beancount

Martin Blais, March 2016

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

This document is about Beancount v2; Beancount v3 is in development and uses a completely different build and installation system. For instructions on importing v3, see this document (Beangulp).

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.

However, today, thanks to the open banking project, we have universal APIs that allow for the quick and reliable download of transactions from multiple bank accounts.

---

[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.

Manual download

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 files, 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 using 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.

Automatic download with open banking aggregator

[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'

The open-source accounting software Firefly III already integrates with some free open banking APIs. For more information, you can visit Firefly III Documentation. An example of an open banking aggregator that could be interesting is GoCardless. GoCardless supports many PSD2-compliant banks in the EU and the UK, and it is free to use.

In the Beancout ecosystem, the tarioch/beancounttools project contains an importer for Nordigen (which is the former name of the GoCardless banking api). This tool works with bean-extract.

1. Simply create and configure a Gocardless account
2. configure tarioch/beancounttools Nordigen importer: add nordigen importer in my-smart.import and put nordigen.yaml in Downloads/
3. run : `bean-extract -e database.beancount my-smart.import Download/ > new.beancount`.

This will automatically ingest the data for you.

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 refers 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 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:

[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.

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 pytest:

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

[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.

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:

---

[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.

```
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 abbreviated 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)