Tracking Personal Finances using Python

Learn how to build a developer-friendly workflow to keep track of your money

Siddhant Goel

Tracking Personal Finances using Python

Siddhant Goel

ISBN-13: 978-3-9824543-2-0

Beancount

In the previous chapter, we talked about Ledger - one of the very first tools driven by a CLI and plain-text files focusing on handling personal finances.

Even though Ledger has a ton of momentum behind it, there are quite a few other implementations of plain-text accounting to choose from. At the time of this writing, some of the most popular ones include GNUCash[1], Hledger[2] and Beancount[3].

GNUCash is an accounting program that is part of the GNU Project[4]. It's a GUI-based application designed to be easy to use and flexible. It supports many features, targeted mostly towards personal and small-business use cases. Hledger is another popular program in this category. It's a reimplementation of Ledger (in Haskell) with a particular focus on ease of use and robustness. Both the projects are in active development and have strong communities behind them.

In this book, we'll focus on Beancount. Beancount is another very popular implementation of the double-entry bookkeeping system. The project saw its first release back in 2008. Having been worked on since then, Beancount is a very mature project with a ton of support behind it. It's also, fortunately, the alternative written in Python, which I assume is the reason you bought this book.

Recall from the last chapter where I mentioned that most of the plain-text accounting tools assume that you (the user) take the responsibility of maintaining the list of your transactions in a plain-text file. Beancount is no different. Beancount defines a strict schema[5] which such transaction files have to follow. As long as your transaction journals follow this format, you can use all the command line utilities that Beancount provides for analyzing your financial data.

In the rest of this book, we'll have a look at what Beancount does, the different command-line tools that it provides you with, and overall how a Beancount-driven workflow to keep track of personal finances looks like.

[1] https://gnucash.org/
[2] https://hledger.org/
[3] https://beancount.github.io/
[4] http://www.gnu.org/
[5] https://beancount.github.io/docs/beancount_language_syntax.html

Workflow

Before getting into the different command-line tools that Beancount provides you with, let's have a quick look at the bigger picture. What does a Beancount-driven workflow look like?

There are three steps involved.

Downloading the transactions

The first step is downloading all the transactions in a machine-readable format. This format does not yet have to be Beancount-friendly. As long as we can use Python to get the contents out, we should be fine.

In the real world, we either pay or get paid. We pay others using cash or by card(s). We'll ignore the differences between debit and credit cards for now. Similarly, we get paid from others either using cash or via online bank transfers. All these count as transactions that we want to track.

Since most transactions are increasingly happening online, our bank account is where we can download all of them. Most banks worth their salt let you export transactions in some format or the other. Commonly used formats include CSV, OFX, and PDF files.

That is the first step of a Beancount-driven workflow. We need to find and download all the transactions for a given timeframe and put them somewhere on disk.

I do this monthly. Every month, usually over the first weekend, I set aside about an hour, download all the transactions from all my bank accounts for the previous month, and save them on disk. The frequency of how often you want to download your bank transactions is completely up to you. I suggest experimenting in the beginning with weekly, bi-weekly, and monthly schedules and seeing which one goes well with your time constraints.

At this point, it would be a good idea to open up your bank's website and check where and how you can download these exports. Often, the section in online banking where you can view your historical transactions is also the one where there's a button to export them to CSV (or some other format).

You don't already have to download and try to organize them on disk yet. We'll do that more systematically in the following sections. At the moment, we're focusing on the high-level workflow.

Convert the transactions

Next, we need to make sure that Beancount can understand our data.

In the previous step, we downloaded a few CSV exports from our bank's website. The content of these files is most likely custom to our bank and not something that Beancount can understand yet.

That is what we'll fix in this section. We'll take those CSV files and convert that data into something that Beancount can understand. That is where the concept of "Importers" comes in.

Importers are classes meant to convert external data to a Beancount-friendly format. These are regular Python classes that define a few specific entry points that Beancount expects. These entry points are nothing else but functions on the class that can read an input (CSV) file, parse it, and output the relevant Beancount data structures. The two main entry points that Importers must define include identifying if a given CSV file is something that it can work with and extracting transactions out of the raw CSV data.

In case you're interested in what these data structures look like, [here is a list] from the Beancount design doc.

Here is what an Importer roughly looks like.

```
class MyBankImporter:
    def identify(self, file):
        # 1. read the first few lines of the file
        # 2. check if that file is something we can work with
        # 3. return True/False accordingly
        pass

    def extract(self, file):
        # 1. read and parse the contents of the file
        # 2. for each piece of info from the file, initialize a
        #    corresponding Beancount data structure
        # 3. return the list
        pass
```

There is, of course, a bit more to it than this. But we'll get into the details later.

In practice, we'll have a distinct Importer for each bank we're working with since each bank will output the CSVs in a different format. We'll let Beancount know of the existence of all these importers using a configuration file. Once this bit of configuration is in place, we can use the command-line utilities that Beancount provides to work with the original CSV files. Based on the CSV file we're working with, Beancount will use our configuration to select an appropriate Importer class.

While this may sound like a lot is going on, I promise you that in practice it's much simpler than that!

For instance, let's say that the CSV file from our bank contains the following data:

```
"01.01.2021","AMAZON.DE","-37.83"
"01.02.2021","ALDI","-4.37"
```

Assume for a moment that we had an Importer class configured for this specific format. The final output from running the importer on this CSV would look roughly like the following:

```
2021-01-01 * "AMAZON.DE"
    Assets:MyBank  -37.83 EUR
```

```
2021-02-01 * "ALDI"
    Assets:MyBank   -4.37 EUR
```

These entries represent two transactions, directly mapping what was contained inside the CSV we obtained from our bank. In the following sections, we'll get into the details of the syntax we've just seen.

At this point, you might notice that the transactions in the final output contain only one leg, and the total does not sum to 0. That violates the "Rule of Zero", we talked about in the previous chapter.

That brings us to the next step.

Balance the transactions

Let's look again at the example from the previous step.

```
2021-01-01 * "AMAZON.DE"
    Assets:MyBank  -42.00 EUR
```

These two lines represent a transaction (specifically an online purchase on Amazon), which took place on the 1st of January 2021. The money that was deducted from `MyBank` is represented here using an `Asset` account.

This transaction is violating the Rule of Zero, which says that independent of how many legs the transaction has, the total amount **must** sum to zero.

So to balance this transaction, we'll insert a leg here manually.

```
2021-01-01 * "AMAZON.DE"
    Assets:MyBank           -42.00 EUR
    Expenses:Online:Amazon   42.00 EUR
```

Now this transaction has two legs, one that deducts 42 Euros from `Assets:MyBank` and another one that debits 42 Euros into `Expenses:Online:Amazon`.

As we talked about in the previous section, "Accounts" don't necessarily have to be physical. They can also be virtual. In this example, we have an account called `Expenses:Online:Amazon`, which tracks how much money has been spent on Amazon. That is not something that would exist in the real world. It's only for our benefit so we can accumulate all the Amazon transactions in one place.

Every time you purchase something online on Amazon, some money will be deposited into this account. And over time, you'll be able to see how much money you've spent on Amazon.

Adding the missing legs to make the transaction amounts sum to zero is called "balancing". Whenever we want to import a new set of transactions, we go through them one by one and balance them. That is often also called "bookkeeping" and forces you to look at all your financial activity.

Workflow: Recap

Those three steps broadly cover what a Beancount-driven workflow looks like.

As I mentioned earlier, this is something that needs to be done periodically. Therefore, I tend to do this every month. Usually, over the first weekend of each month, I set aside half an hour for myself. During that time, I download all my transactions from the different bank accounts, run my importers over them, and balance each transaction by hand. This way, I get a very good idea of my finances in the most recent timeframe.

In total, this entire activity does not take me more than half an hour. Your mileage may vary, but I think this is a fair price for the peace of mind it gives me.

Directory Structure

Now that we are familiar with what the overall workflow looks like let's have a look at what all this looks like "on paper".

For this illustration, I created an empty Beancount project in a directory called `finances/` on my laptop. Here is what the contents look like.

```
finances
|-- alice.beancount
|-- config.py
|-- requirements.txt
```

Let's go through the contents of this directory one by one.

alice.beancount

This is the main file that contains all our Beancount transactions.

Whenever we want to import any transactions from our bank exports or want to add a transaction by hand, this is where all those changes go.

This one file contains **all** our financial information. That includes the accounts we have, all our transactions occurring before today and any associated metadata.

config.py

This is the configuration file that Beancount reads when we're importing external data.