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

Plain Text Accounting

In the previous chapter, I listed requirements from a personal finance system that I could imagine using myself. Broadly speaking, this included three main things. First, I was looking for software that would ideally be simple, could be self-hosted, and would ideally be open-source.

When I started looking for existing apps that would satisfy these criteria, I didn't find many. Most of the applications I found were either Software-as-a-Service (SaaS) apps or mobile apps. In addition, these were either proprietary, cloud-based, or both.

After a bit of searching, I ran into something called "Plain Text Accounting". The software developer in me loves plain text, so I decided to look further into it. And it turned out that this was **exactly** what I had been looking for.

Accounting

Before getting into Plain Text Accounting, let's first agree on a definition of Accounting.

Accounting is an umbrella term that can have hundreds of different definitions for different people under different contexts. For the purpose of this book, though, we are going to limit our definition to "personal accounting". Thus, over the next chapter, when we say "Accounting", we will mean keeping track of money flow for **individuals**.

Now that a definition of Accounting has been established and agreed upon let's get back to Plain Text Accounting.

Plain Text Accounting

As might be obvious from the name, Plain Text Accounting is "Accounting" performed using "Plain Text".

In other words, when an application is built around the principles of Plain Text Accounting, it usually means that it stores user data in plain-text files and provides a suite of tools, often command line, to

work with such files. Another implication is that the end-user is often responsible, at least partially, for maintaining their data in such files.

At this point, one might think that this suite of tools potentially includes UNIX utilities like `grep` or `sed` to do some sort of pattern matching or file modifications in place. At least that's the first reaction I get when I tell people I maintain my finances in plain text files.

That is not at all the case. When you're working with such an application, you must work within the boundaries of a specification. For instance, if you want to import all bank transactions into a plain-text file, there's a strict schema that you need to stick to. If there are any deviations, the application you're using would loudly complain and ask you to go back and fix things. Having the format strictly defined makes it easier for computers to parse such data without resorting to `grep`-like utilities. Consider this functionally equivalent to a "compiler".

The idea behind plain-text based accounting started with the invention of Ledger[1] back in 2003. There might certainly have been similar ideas and concepts (and maybe even implementations) floating around before that. But Ledger seems to have given the whole ecosystem the momentum it has today.

At its core, Ledger is a set of commands written in C++ that help you manage money flow. The core idea is that there's a plain-text file on your disk somewhere that **you** are responsible for maintaining. This file contains transactions in a specified format. Transactions, in this case, mean the flow of money from one account to another. We'll talk in detail later about what an "account" means. As long as the contents of this file follow the schema that Ledger requires, you can use the commands that Ledger provides to do a variety of tasks. That includes things like checking account balances, generating monthly, weekly, or yearly reports, and much more.

Double Entry Bookkeeping

One of the most important ideas behind Ledger is letting **the user maintain** a journal of transactions. I consider this idea so important that I'll keep coming back to it now and then over the next chapters.

But before that, let's talk about this user-maintained transactions journal.

I mentioned earlier that we are not resorting to parsing files ad-hoc. Instead, the plain-text files we're interested in have a strictly defined format. So, what exactly does this format look like?

[1] https://www.ledger-cli.org/

Transactions

Consider for a moment the interaction you went through when you bought this book. Ignoring the web side of things for a minute, what essentially happened was that you paid me money through your bank and downloaded the book in exchange.

That was one transaction.

Money flowed from one bank account to another. And the amount of money subtracted from one account was the same as the amount added to the other one. Tracking such transactions is what we're generally interested in. This concept of representing transactions as the flow of money from one account to another is a slightly simplistic view of what we call Double Entry Bookkeeping.

Double Entry Bookkeeping is a way of keeping accounting records, or "books", where individual records represent money going from one account to another. Every record contains the source and destination accounts as well as the amount of money that was exchanged.

This transaction is represented using two "legs" - a **debit** leg representing money *added* and a **credit** leg representing money *deducted*. If we were to try to specify this in plain text, we might develop something as follows.

```
01-01-2021 Account-A -25 EUR
           Account-B +25 EUR
```

In this short snippet, we are trying to say that on the 1st of January 2021, we deducted 25 EUR from `Account-A` and added them to `Account-B`. The first line represents the **credit** leg, and the second is the **debit**. Even though this is a fictitious format that I just made up, it can represent a transaction quite well because of three reasons:

1. It's easy for humans to parse.
2. It's easy for machines to parse.
3. It describes very concisely what happened (in terms of the flow of money).

Rule of Zero

One very important thing about transactions is that the total amount inside a transaction **must** sum to zero, independent of how many legs the transaction has.

The reasoning is as follows. When we talk about transactions, we're talking about money flow from one account to another. So if we're taking away an amount X from one account, an equal amount must be debited to some other account, such that the sum of the amounts inside a transaction is zero.

Note that we've talked of transactions representing an exchange of money between *two* accounts so far. That does not *always* have to be the case. A transaction can also involve money flowing between *multiple* accounts. In any case, the sum of the amounts in a transaction must be zero, regardless of the number of accounts involved.

Accounts

In the previous sections, I used the word accounts without a proper definition.

Accounts represent "entities" between which money flows. They do not necessarily have to refer to real-life bank accounts. They can also represent things like supermarket expenses. In the plain-text account world, one can have hundreds (or even thousands) of accounts. For instance, you probably have one or more "physical" accounts with banks. But you're also most likely going to the supermarket every few days or taking public transport every day. These two things qualify as accounts.

Generally speaking, accounts can be both physical and virtual, and you can have as many accounts as you want.

Let's go back to the previous snippet to make this concept a bit clearer. We'll replace `Account-A` and `Account-B` with something that resembles the real world a bit more.

```
01-01-2021 Credit-Card -25 EUR
           Supermarket +25 EUR
```

Now, this transaction represents a visit to a supermarket where we bought 25 EUR worth of groceries and paid with our credit card.

The signs that you see next to the amounts tend to be the same for all accounts of a given type. In this example, we can see that the `Supermarket` account was debited with 25 EUR, and as such, has a positive account balance after this transaction. Over time, as more and more transactions come in, the value associated with the `Supermarket` account will become more and more positive.

That might be slightly unintuitive in the beginning. We aren't expecting the supermarket to **give** us money. Why should it then have a positive balance? The reason goes back to the "Rule of Zero" from earlier. Since the amounts in a transaction must sum to zero and the `Credit-Card` account is being *charged* an amount, we must assign an equal and opposite amount of money to the `Supermarket` account. Over time, as our transaction journal represents more transactional activity, this positive amount associated with the `Supermarket` account will show us how much we've spent over time buying groceries.

Generally speaking, the sign of an amount is a property inherent to the type of account we're handling. Accounts are of four types, each with a corresponding sign (positive or negative).

Account #1. Assets

Asset accounts generally represent things you have or things you own. These accounts generally tend to have a positive value because this is something you own.

For instance, the physical account you have with your bank would qualify as an asset. In my transaction journal, the asset accounts are almost exclusively bank accounts.

Account #2. Income

Income accounts represent money flowing into one of your Asset accounts. Generally, the transactions representing Income accounts are the ones where you're receiving your salary from your employer or getting paid from some consulting work you might have taken up or some other income source.

These accounts tend to have negative values. That might be slightly counter-intuitive as the income one earns is always positive amounts of money. But if we go back to the "Rule of Zero", we can see that since an income gets deposited directly into a bank account, an Asset, we need an equal and opposite amount for the complete transaction to balance out. Hence, transaction legs representing Income accounts generally take negative values.

Account #3. Liabilities

Liability accounts represent money you owe others. For instance, if you're buying a house and having it financed through a bank, the bank lends you money and expects you to pay it back in monthly installments. That is a Liability.

Until the point when you've paid all the money back, this is money that you owe the bank. Liability accounts track such amounts because of which they generally have negative values. Over time, as you pay off the periodic installments from one of your Asset accounts, the value will become less negative. At some point, when all the money has been paid off, the value will become zero.

Account #4. Expenses

Expense accounts are very common and represent your everyday transactions. Things like supermarkets, vacations, restaurants, etc., can all be represented using Expense accounts.

These accounts generally tend to have positive values. The reason can again be traced back to the "Rule of Zero". We generally spend money **on** these things. For instance, when we go to the supermarket, we're giving them money in exchange for some groceries. This money generally comes from the

Asset accounts. So Expense accounts are usually involved as the debit leg, their value-adding up to something positive over time.

Conclusion

At this point, we have covered enough ground to get a general understanding of the plain-text accounting ecosystem. We're also familiar with the two concepts we'll use most often (accounts and transactions) in the rest of the book.

We should now be ready to take the next step in this journey and move on to the implementation side of things. Let's go to the next chapter, where we'll do exactly that!