Getting Started With Ledger

May 13, 2016
0d55539

Rolf Schröder & Others

Contents

Assumptions & Promises

This book is written for first timer users of Ledger (surprise!). Ledger is a command line tool and I therefore expect the reader to be familiar with the command line in general. You should know what a terminal is, how to execute arbitrary commands or install new software via the command line.

You do not need to know how to program but it probably helps if you have ever read some code.

The books aims to be concise. This implies some technical experience and a do-it-yourself attitude on the reader's side.

On the other hand, you will get a working environment to get started with Ledger out-of-the-box. Technical details are not hidden and the code is freely distributed. You may change every aspect of the work flow and code presented throughout the book.

You may be surprised how short the book is. This owes to the don't-repeat-yourself principle consequently followed in this book: Nothing that can be seen with minimum effort in code is repeated in the book. You will always get a hint on where to get the information. This simplifies the author's life, makes the book less error-prone and gives the reader plenty of opportunity to understand what's going on behind the curtain.

1 An introduction to Ledger

This chapter introduces the philosophy of double-entry accounting, Ledger as a command line tool and its basic usage.

1.1 Double-entry Accounting

Double-entry accounting is a standard bookkeeping approach. In accounting, every type of expense or income and every "place" which holds monetary value is called an "account" (think "category"). Example accounts may be "Groceries", "Bike", "Holidays", "Checking Account of Bank X", "Salary" or "Mortgage". In double-entry accounting, one tracks the flow of money from one account to another. An amount of money always figures twice ("double") in the books: At the place where it came from and at the place where it was moved to. That is, adding $1000 *here* means removing $1000 from *there* at the same time. In consequence, *the total balance of all accounts is always zero.* Money is never added to an account without stating where the exact same amount came from. However, more than two accounts may be involved in one transaction.

For example, buying a book online for $15 moves money from the account "Creditcard X" to the account "Books". Receiving a $2000 salary from your boss means moving $2000 from the account "Salary" to the account "Bank" (or whatever). Buying groceries and detergent at the supermarket may move money from "Creditcard X" to both "Groceries" and "Household".

In general, account names depend on the situation. But, one usually has the following main accounts:

- Expenses
- Income
- Assets
- Liabilities
- Receivables
- Equity

The level of detail needed for the subcategories ("Expenses" -> "Groceries" -> "Fruits" -> "Bananas") is up to the requirements.

1.2 Ledger

Ledger is a double-entry accounting command line tool originally authored by John Wiegley. It is an extremely potent tool and it takes some time and effort to be able to unleash its power. However, once mastered there is not much you may miss while doing personal or professional accounting.

Extensive documentation can be found at http://ledger-cli.org.

Working with Ledger boils down to two distinct types of action: Updating the list of transactions (the "journal") and using Ledger to view/interpret that data.

Ledger follows good old Unix traditions and stores data in plain text files. This data mainly includes the journal with the transactions and some meta information, too. A typical transaction in Ledger looks like this:

```
2042/02/21 Shopping
    Expenses:Food:Groceries                $42.00
    Assets:Checking                       -$42.00
```

Any transaction starts with a header line containing the date and some meta information (in the case above only a comment describing the transaction). The header is then followed by a list of accounts involved in the transaction (one "posting" per line; each line starting with a whitespace). Accounts have arbitrary names but Ledger uses the colon to distinguish between subcategories. The account name is followed by at least two whitespaces and the amount of money which was added (positive) or removed (negative) from that same account. Actually, Ledger is clever enough to calculate the appropriate amount so it would have been perfectly valid to only write:

```
2042/02/21 Shopping
    Expenses:Food:Groceries                $42.00
    Assets:Checking
```

The journal file is as simple as that and there is not much to know about it at this point. Note that Ledger never modifies your files.

The following transactions illustrate some basic concepts used in double accounting & Ledger:

```
; The opening balance sets up your initial financial state.
; This is needed as one rarely starts with no money at all.
; Your opening balance is the first "transaction" in your journal.
; The account name is not special. We only need something convenient here.
2041/12/31 * Opening Balance
    Assets:Checking                       $1000.00
    Equity:OpeningBalances

; The money comes from the employer and goes into the bank account.
2041/01/31 * Salary
    Income:Salary                          -$1337
    Assets:Checking                         $1337

; Groceries were paid using the bank account's electronic cash card
; so the money comes directly from the bank account.
2042/02/15 * Shopping
    Expenses:Food:Groceries                 $42.00
    Assets:Checking

; Although we know the cash sits in the wallet, everything in cash is
; considered as "lost" until recovered (see next transaction and later chapters).
2042/02/15 * ATM withdrawal
    Expenses:Unknown                       $150.00
    Assets:Checking

; Paying food with cash: Moving money from the Expenses:Unknown
; account to the food account.
2042/02/15 * Shopping
    Expenses:Food:Groceries                 $23.00
    Expenses:Unknown

; Ledger automatically reduces 'Expenses:Unknown' by $69.
2042/02/22 * Shopping
    Expenses:Food:Groceries                 $23.00
    Expenses:Clothing                       $46.00
    Expenses:Unknown

; You can use positive (add money to an account) or negative
; (remove money from an account) amounts interchangeably.
2042/02/22 * Shopping
    Expenses:Food:Groceries
    Expenses:Unknown                       -$42.00
```

The above example already introduced some nice concepts from Ledger. Still, reading the text file is a bit boring. Before we let Ledger parse that for us, you'll probably still need to install it first ...

1.3 Installing Ledger

Ledger's latest version can be obtained from its website. I recommend to have at least version 3.0.3 running.

Further dependencies for the ecosystem presented in this book are:

- Git
- Python

Optional but recommended:

- tmux

- tmuxinator
- gnuplot

1.3.1 Linux, Mac OS X & BSD

You'll find what you need at the download site.

When running Linux, it might just be a question of:

```
$ sudo apt-get install ledger
# or
$ sudo yum install ledger
# or ...
```

However, the distribution's package might be older than the one provided at the download site. Ledger comes with a very good installation documentation. Refer to the Github page for more details.

1.3.2 Windows

Ledger is hard to get running on Windows (you would probably need to compile it yourself and that's often a pain in the ass on Windows). Additionally, the setup presented in this book makes heavy use of the traditional Unix command line infrastructure. I therefore recommend to setup VirtualBox and install Ledger on a Linux machine. You could either use plain VirtualBox or VirtualBox with Vanguard on top. The latter is probably easier & faster. It will be totally fine to connect to your virtual machine via SSH in Windows afterwards so you won't need to actually "use" the Linux environment.

Step-by-step Instructions (without Vagrant):

- Download and install VirtualBox.
- Download an ISO of a Linux distribution of your choice (Ubuntu?).
- Install Linux on the virtual machine.
- Install the OpenSSH *server* ("`$ sudo apt-get install openssh-server`" for Ubuntu).
- Make sure you can access the vm via SSH.
- Run the machine in headless mode if you want.
- Install babun (built on top of Cygwin) on your Windows machine.
- Connect to the vm via SSH.
- Follow the instructions to install Ledger on Linux.

Step-by-step Instructions (with Vagrant):

- Download and install VirtualBox & Vagrant.
- Download this Vagrantfile from Github.
- Open a terminal, go to the Vagrantfile's location and run `vagrant up` (this will setup an Ubuntu machine with Ledger installed).
- To connect to the VM via SSH, use `vagrant up` followed by `vagrant ssh` from within the same folder.

1.4 A first teaser

With a working installation of Ledger on your machine, grab these sample transactions from Github (click the 'Raw' button) and copy them to a text file called `journal.txt`. Then run this:

```
$ # Usage: ledger -f <journal-file> [...]
$ ledger -f journal.txt balance
$ ledger -f journal.txt balance Groceries
$ ledger -f journal.txt register

# Start an interactive session
# and type "balance", then  press Enter
# (press ctrl+d to quit)
$ ledger -f journal.txt
```

This should give you a first feeling for Ledger. You will get to see more in the Reports chapter later. But first, we need to get our own Ledger ecosystem set up.

2 The Setup

Throughout this book, we will use a specific setup for Ledger. Two repositories are used to split up code and data. The "ecosystem" folder (see GSWL-ecosystem) folder contains the scripts & other stuff to manipulate the journal. On the other hand, the "private" folder will contain the actual financial data. I have provided a sample private folder (see GSWL-private) which we'll use as a reference.

Splitting up code & data allows to only encrypt what is necessary, share common code more easily and enable independent version control.

The Readme at GSWL-private explains how to clone the repos. Go ahead and get them now.

Let's look at the repo contents:

```
$ cd ~/src

$ ls ~/src/GSWL-ecosystem
alias  convert.py reports.py
# Omitting other files for now ...

$ ls ~/src/GSWL-private
alias.local       csv2journal.txt  main.txt  misc.tmp.txt
bankaccounts.yml  journal.txt      meta.txt  reports.txt
# Omitting other files for now ...
```

2.1 Common files

The ecosystem contains code to handle the actual data in a smart (automatic!) way. The scripts `convert.py` and `report.py` help to integrate external CSV data or interpret the data respectively. The `alias` file is a BASH script which defines common aliases and functions. You can check out these files now but we'll cover them later after having had look at the private repo.

2.2 Private data

There a lot of files in the private repo. Only the most important ones are covered for now.

Remember that Ledger expects the journal file to be provided by `-f`. The aliases defined in the ecosystem assume this file to be named `main.txt`. However, this file does not contain any transactional data. The file only contains a list of `include` statements. It may look like this:

```
; This is the main entry file for ledger.
; Nothing fancy happens here. All real data is in journal.txt, configuration
; stuff etc. is to be found in meta.txt.

; Include the config file.
include meta.txt

; Include the actual journal.
include journal.txt
```

Using `include` statements is a good way to keep stuff separated which does not belong together. It also helps you to try out new Ledger configurations or data without polluting your version-controlled files.

The actual transactions are all stored in `journal.txt`. Checkout the private folder's `journal.txt` to get a first overview of the sample data.

2.2.1 Going meta

In this setup, the file `meta.txt` should contain all Ledger configuration stuff & any other non-transactional data.

For example, a useful statement inside this file is `account`. This lets you predefine any accounts that shall be used by Ledger. Ledger does not require you to do so but it is good practice anyway. Furthermore we will use the `--pedantic` command line argument later on which causes Ledger to produce an error when unknown accounts are used. Account definitions may look like this:

```
account Assets:Checking
account Expenses:Dining
account Expenses:Groceries
account Income:Salary
```

Likewise, `commodity` defines the valid commodities in use:

```
commodity $
commodity €
commodity BTC
```

The sample `meta.txt` does indeed not include other configuration. Or does it?

2.2.2 Other files

Then, there is a Bash script called `alias.local` which contains local configurations. This script is automatically sourced by `ledger-ecoystem/alias`. Now may be a good moment to have a look at these scripts. Try to find out what the `led` command looks like, what the content of the environment variable `$LAST_AMN` is and how `alias.local` is sourced.

Finally, the files `bankaccounts.yml`, `csv2journal.txt` & `misc.tmp.txt` are used to update the journal in an automated fashion. `reports.txt` lists questions repeatedly asked about the financial situation. All of theses is explained in later chapters but feel free to inspect them right away.

2.3 Orchestrating ecoystem & private data

You should have the following mental model of the presented setup: Most code is in the ecosystem folder. All actual data is in the private folder. Working with Ledger means working in the private folder. To unleash the power of all scripts etc., one needs to source `ecosystem/alias` form within the private folder. This in turn sources `alias.local` from the current working directory. The local alias file allows to overwrite ecosystem functionality or add new features.

Having the ecosystem scripts & the sample data available allows to get a more precise feeling of the day to day work with Ledger. Run the following commands:

```
$ cd ~/src/GSWL-private && source ~/src/GSWL-ecosystem/alias # See GSWL-private/.bashrc for an alias!
$ which led
$ led bal
$ led reg
$ ledreports # explained later
```

To be absolutely clear: `led` is just an alias to `ledger` combined with some predefined arguments (see `ecosystem/alias`). You can of course run plain `ledger` on the same data. In this case, you'll need to tell Ledger at least where to find the journal file: `ledger -f main.txt`. Think of `led` as `ledger` with a predefined input file and sane defaults.

2.4 Tmux & Tmuxinator

I highly recommend the use of Tmux for whatever business you're doing on the command line. This tool speeds up your work flow so much it is actually ridiculous. It as a better version of `screen` and "lets you switch easily between several programs in one terminal, detach them (they keep running in the background) and reattach them to a different terminal". If you do not use it till now, you will wonder how you survived before. The sample `.tmux.conf` in the private repository and this HowTo gets you started if you need it. Have a look at the sample `.tmux.conf` and make sure to know at least how to jump between windows, jump between panes, create a new window and maximize (resize) a pane.

Tmuxinator sits on top of tmux and lets you predefine tmux sessions for specific tasks. I defined a specific tmux session for usage with the private folder. The tmuxinator session file `.tmuxinator.ledger.yml` can be found in the private repository (check it out now!).

Starting a tmux session with the private repo (assuming tmux & tmuxinator are installed):

```
$ cp ~/src/GSWL-private/.tmux.conf ~/ # Optional, only if you've never used tmux
$ mkdir -p ~/.tmuxinator
$ ln -s ~/src/GSWL-private/.tmuxinator.GSWL-private.yml ~/.tmuxinator/GSWL-private.yml
$ mux start GSWL-private # Starts a new Tmux session
```

Within each Tmux session window, `ecosystem/alias` is sourced.

The sample `GSLW-private/.bashrc` provides some aliases to start/stop the Tmux sessions. You should source this file in your `~/.bashrc` anyways.

With the setup up and running, we can now further play around with Ledger's actual features.

2.5 Your own setup

To get started with your personal setup, checkout (no pun intended) this.

3 Reports

Having a journal is all nice and fine but we actually only keep it to gain some insights on our financial situation. This is where reports fly in. A report displays the journal in some meaningful way. Ledger is able to report in a great variety of ways which makes it extremely powerful. The most standard commands for reporting are `balance` and `register`.

3.1 Balance reports

The balance report is quite intuitive. It creates a total balance from all transactions. The basic command is:

```
$ ledger -f file.txt bal[ance]
# or, in our case
$ led bal # alias defined by ecosystem/alias
```

The output is something like:

```
        $2145.00  Assets:Checking
       $-1000.00  Equity:OpeningBalances
         $192.00  Expenses
          $65.00    Food:Groceries
         $127.00    Unknown
       $-1337.00  Income:Salary
 ---------------
               0
```

Normally, you want to have a more precise balance by putting some restrictions on the account name, the time or whatever:

```
# Restrict by date.
$ led (--period|-p) "last 6 months" bal
$ led -p "2042" bal

# Restrict by account name.
$ led bal ^Expenses

# Restrict by account names.
$ led bal ^Expe and not Groceries

# Show all assets in the same currency (this assumes a prices database for conversion, see below).
led bal --exchange $ ^Assets

# Do not show sub accounts.
led --depth 2 bal
led --depth 3 bal # Note how the totals do not change.

# Do not indent accounts.
led --flat bal
```

3.2 Register reports

Register reports display the journal like an old-fashioned register. The example command line arguments as above apply, of course.

```
# Show the register report.
$ led reg
# (The second last column gives the actual amount, the last column the running sum.)

# Restrict time span.
$ led -p "2041" reg Assets
```

```
# Show the running average.
$ led reg -A Restaurant
# (Ignore the "<Adjustment> lines". The 2nd column gives the running average.)

# Group by month.
$ led reg -M Food

# Collapse transactions with multiple postings into one.
$ led reg -M -n Expenses # compare against 'led reg -M Expenses'
```

3.3 Advanced Report Filtering

Reported information (balance or register) can be filtered in a more sophisticated way. This is achieved by either `--limit` (`-l`) or `--display` (`-d`). The difference between both is that the first limits the postings to be considered for calculations whereas the latter limits the postings to be considered for display. This means that the arguments ("expression") used in conjunction with `--limit` is active while Ledger goes through the journal file. On the other hand, the expressions supplied to `--display` will only filter the final result *after* having read the journal completely.

As an example, consider one wants to get an overview of the usual amount spent every month. That is, we're interested in the average monthly expenses over the last x months. This can be easily achieved by `led -M -n -A -p "from 2041/11/01" register ^Expenses`. Go try it out in the private repo. The resulting report will report each month's total expenses starting from November 2041 and calculate the running average (last column). Now, to get back to our filtering: Imagine you're only interested in the average of all months combined. This information is only available after having taken into account the last month obviously. However, all previous monthly expenses are needed to calculate it. This is where the difference of `--limit` & `--display` can be easily seen:

```
# Show monthly expenses & average since Nov 2041
$ led -M -n -A --limit "date>=[2041/11/01]" reg ^Expenses
```

vs

```
# Show monthly expenses since Nov 2011 & average monthly expense since the dawn of time
$ led -M -n -A --display "date>=[2041/11/01]" reg ^Expenses
```

See how the last the value of the last column in the first row is different to the first command. This comes from the fact that there is journal data from before 2041/11/01 which is taken into account for the average calculation when only restricting with `--display`.

Combining both:

```
# Show monthly expenses for Mar 2042 & average monthly expenses since Nov 2011
$ led -M -n -A --limit "date>=[2041/11/01]" --display "date>=[2042/03/01]" reg ^Expenses
```

See how the average column changes? That's exactly the difference between filtering before calculation ("limit") or before presenting results ("display").

As another example, consider the following journal:

```
2042/01/15 * Random stuff 1
    ; Earn $100, spend $50 and keep the rest at the bank.
    Income                                  $-100
    Expenses                                $50
    Bank                                    $50

2042/01/25 * Random stuff 2
    ; Spend $150 taking the remaining $50 plus a $100 loan.
    Expenses                                $150
    Bank                                    $-150
```

Here, someone lives way beyond her means. This person only earned $100 but spent $200. The bank apparently gave a $100 loan:

```
$ led bal
               $-100  Bank
                $200  Expenses
               $-100  Income
--------------------
                   0
$ led reg
42-01-15 Random stuff 1         Income                    $-100        $-100
                                Expenses                    $50         $-50
                                Bank                        $50            0
42-01-25 Random stuff 2         Expenses                   $150         $150
                                Bank                      $-150            0
```

Let's say we want to have a look at all accounts which have a positive balance. The expressions employed would be `amount > 0`. However, depending on whether `--limit` or `--display` is used, the outcome is quite different:

```
$ ledger bal -d 'amount > 0'
                $200  Expenses
$ led bal -l 'amount > 0' # limit postings for calculation
                 $50  Bank
                $200  Expenses
--------------------
                $250
```

The output of `--display` seems quite intuitive. At the end of the day, the Expenses account has a balance of $200 whereas the others are negative. Using `--limit` only considers the postings ("lines") with a positive amount: For the first transaction, that means $50 in the Expenses account and $50 in the Bank account, for the second transaction we have another $150 in the Expenses account. Hence we end up with a total of $250.

Most of the time, `--display` is what you want. In fact, the result of `--limit` & `--display` are often the same. Not always though:

```
# Show the total amount of $ ever sent to the bank account (only possible with -l).
$ led bal -l 'account =~ /Assets:Checking/ and amount > 0'

# Get the amount of $ spent for books at RandomShop (-d is fine here, too).
$ led bal -l 'account =~ /Expenses:Books/ and payee =~ /RandomShop/'

# List all expenses higher than $100.
$ led reg  Expenses -l 'amount > 100'
```

More information on how to filter reports can be found at Ledger's online documentation.

3.4 Sample Questions

Try to get the following information using the private repo [additional question in brackets]. The answers are found on the next page.

(1) How much money was spent on groceries [since Jan 1st 2042]?
(2) How much money was spent for rent & electricity?
(3) How much money was spent in total each month [on average]?
(4) Which income was "earned" that is not salary?
(5) How much money was spent on gifts (account name) on Amazon (payee name)?

3.4.1 Answers

(1) How much money was spent on groceries [since Jan 1st 2042]?

```
led bal Groceries
# or
led bal -l 'account =~ /Groceries/'
# With date restriction:
led bal Groceries -p "since 2042/01/01"
# or
led bal -l 'account =~ /Groceries/ and date >= [2042/01/01]'
```

(2) How much money was spent for rent & electricity?

```
led bal Expenses:Rent Electr
# or
led bal -l 'account =~ /Expenses:Rent|Electr/'
```

(3) How much money was spent each month [on average]?

```
led reg -n -M [-A] Expenses
# or
led reg -n -M [-A] -l 'account =~ /^Expen/
```

(4) Which income was "earned" that is not salary?

```
led reg Income and not Salary
# or
led reg -l 'account =~ /Income/ and account !~ /Salary/'
```

(5) How much money was spent on gifts (account name) on Amazon (payee name)?

This can only be solved by `--limit`:

```
led bal -l 'account =~ /Gifts/ and payee =~ /AMAZON/'
```

Looking at the answers, one realizes that the `--limit` switch often looks cumbersome. Still, in some situations, you may have to resort to the more powerful expressions.

3.5 Recurring Reports

Most of the time, you're interested in the "usual suspects". I personally have 5-10 reports, which I always want to check after updating the journal. Obviously it's a waste of time to type them in by hand. One way to simplify things would be to define (probably cryptic) aliases for each of these reports. Another way is to use the `reports.py` script provided in the ecosystem repo. The script opens the file `reports.txt` in the current working directory and displays predefined reports one by one. The text file is expected to contain empty-line-separated sections consisting of comments (starting with `#`) and actual commands to execute. You may invoke the whole thing with the `ledreports` command (this is actually done automatically in the tmux session's "overview" window). The `reports.txt` in the current working directory may look like this:

```
# Each paragraph consists of explanations ('# ...') and the cmd itself (last line).
# The first section is the header.

# Show the current journal status.
led bal

# Show all transactions involving Food, then show transactions involving Transportation.
led reg Food
led reg Transportation

# Show expenses in percentage & sort by amount.
led bal --percent --sort "(total)" --depth 2 Expenses
```

The `reports.py` script will show the above sections as 3 distinct pages each with one or more reports. You may cycle through the listed reports using `j` and `k`. In the terminal window, each report is prefixed by the comment and the actual command to help you understand what's going on. Note that the script sources the `alias` file from the ecosystem in the current working directory in order to allow the usual commands. Remember that the `alias` file itself tries to source the file `alias.local` in the current working directory which allows you to easily append your own stuff.

Try out `ledreports` in the private repo:

```
$ sl # "start ledger" as defined the .bashrc
$ ledreports
```

3.6 Other Reports

You want to try out these:

```
$ led stats
$ led accounts
$ led payees
$ led print
$ led xml
```

3.7 Visualization

Ledger comes with two handy switches (`-j/-J`) to allow to feed it's output to gnuplot (or other tools) to visualize the data. The register report can be modified to only output the date and either the current the amount (`-j`) or the running total (`-J`).

```
# Output monthly expenses
$ led reg -n --monthly -j Expenses
2041-09-01 509
2041-10-01 484
2041-11-01 955.49
2041-12-01 809.49
2042-01-01 455.5
2042-02-01 285.5
2042-03-01 882.47
# Output cumulative monthly expenses
$ led reg -n --monthly -J Expenses
2041-09-01 509
2041-10-01 993
2041-11-01 1948.49
2041-12-01 2757.98
2042-01-01 3213.48
2042-02-01 3498.98
2042-03-01 4381.45
```

The `ecosystem/alias` defines the function `ledplot` which wraps around gnuplot to visualize some data:

```
ledplot -j -M -n reg Expenses # assuming gnuplot is installed
```

Some of the sample reports do have a couple of predefined plots you can checkout. See `private/reports.txt` for more.

4 Updating the journal

This chapter explains how the journal is updated. The work flow described here assumes that the journal update happens on a monthly basis. However, nothing prevents you from doing it differently.

Remember that the journal keeps track of all financial transactions. Updating the journal happens in two steps: By manually adding transactions (everything without electronic record, i.e. cash transactions) and by using Ledger's in-built conversion method to automatically add CSV data. New data is then merged into the existing journal.

4.1 Cash transactions

The `journal.txt` is never manipulated directly. Instead, it gets updated automatically using a combination of scripts and aliases (see later). However, cash transactions need to get into the ledger *somehow*. Cash transactions should be added to the file `misc.tmp.txt`. (The filename ends with `.tmp.txt` to denote that it shall never be put into version control. See the file `.gitignore`; obviously the private repo contains it for the sake of the example.)

So, how do you deal with cash expenses in a double accounting system? You don't want to track every single dime and you don't want to ignore bigger cash expenses neither. The most convenient way is to assume that every dollar withdrawn is a dollar spent. In Ledger's double accounting speak, this means moving money from the account "Assets:YourBank:Checking" to "Expenses:Unknown". Basically, you assume the money is gone. However, when you know how the money was spent, you just move it from "Expenses:Unknown" to whatever account you want. Although this not correct technically, the approach simplifies dealing with cash quite a lot. The cash was either spent on a specific account or it's an unknown expense.

So tracking cash in `misc.tmp.txt` may look like this.

```
; This file lists all cash transactions that happened *after* the last
; journal update. Once this data has been added to journal, this file
; is emptied. Note how the scheme is always the same: move money from
; Expenses:Unknown to a specific account.

2042/04/10 * Swimming
    Expenses:Sports:Swimming                $10.00
    Expenses:Unknown

2042/04/13 * Cinema
    Expenses:Cinema                         $15.00
    Expenses:Unknown

2042/04/18 * Tails of the City
    Expenses:Books                          $5.00
    Expenses:Unknown
```

We'll see later how the data in `misc.tmp.txt` is appended to the journal.

4.2 Electronic transactions

Spending cash money is just fine but most our transactions are electronic nowadays. Keeping track of theses transactions is actually quite easy. Virtually every financial institution (banks, credit unions, payment service provider, etc.) provides you with a CSV file that lists your transactions. You should probably change your bank if they don't provide this service.

Ledger has the built-in command `convert` which automatically converts CSV files into Ledger's transaction format. The main feature of interest for us is the account recognition based on the transaction's payee. See `meta.txt` and look for "payee" to get a first feeling how that might work. We'll use this command in combination with an utility script to convert CSV files in a quite efficient way.

For completeness, I would like you to check out reckon and/or csv2ledger. Both are yet other approaches to convert CSV data. They did not suit my needs (see below) and it's obviously more fun to hack something together for your own work flow.

4.2.1 The general work flow for electronic transactions

The work flow goes like this:

- Download the CSV file from the bank.
- Call the utility script to convert the input data.
- Check for "unknown" (= not yet recognized) transactions; modify `meta.txt` to match these bank transactions with your Ledger accounts.
- Repeat until done.

Let's go through theses steps in greater detail. Getting the CSV data depends obviously on the financial institution. It's handy to always save it to the same location in a "machine readable name" (ex: `CSV/bankname_<month><year>.csv` or `CSV/bankname_latest.csv`) because this allows for easier scripting.

The utility script (`ecosystem/convert.py`) manipulates the CSV data to make Ledger's `convert` understand it. This is mainly replacing the header lines and providing some more info for Ledger like the bank account's currency. Ledger's `convert` command expects the first line in the CSV file to describe the transaction columns of the remaining lines. One has to tell Ledger which columns represent the payee, the amount, the date and so on. For a CSV line like so ...

```
04/08/2042,05/08/2042,xx,xx-xx-xx,123456789,JOHN DOE,MONEY FOR LAST NIGHT,200.0
```

... the new header line may look like this:

```
,date,,,,payee,note,amount
```

That is, the second column codes for the date, the sixth for the payee and so on.

The converter script needs this predefined header and other information for all your bank accounts (read: all your different CSV files). They are configured in `private/bankaccounts.yml`. This file is read in by the utility script. One example entry of that file may be:

```
Assets___BankName___CurrentAccount:
    convert_header: ',date,,payee,note,,,amount'
    ignored_header_lines: 7
    date_format: '%d.%m.%Y'
    currency: 'EUR'
    ledger_args: '--invert'
    expenses_unknown: 'Expenses:Unknown'
    ignored_transactions:
        - ['.*EXAMPLEPAYEENAME.*', '.*PayPal.*']
    modify_transactions:
        - ['Name Surname";"Unique Desc', 'Name Surname UniqueIdentifier";"Unique Desc']
```

The name of the root node equals the bank account's account name in Ledger where colons are replaced by 3 underscores. In the above case, that would mean the configured account is `Assets:BankName:CurrentAccount`. The sub-sections are:

- `convert_header`: The header line mentioned above.
- `ignored_header_lines`: The number of header lines in the original CSV file (which should be ignored).
- `date_format`: The date format of the transaction entries (more here).
- `currency`: The currency used in this account.
- `ledger_args`: Further ledger arguments (`--invert` inverts the input amounts).
- `expenses_unknown` (optional): Ledger assigns money from unknown sources to the in-built account "Expenses:Unknown". You may change that account if needed.
- `ignored_transactions` (optional): A list of regular expressions for transactions to be ignored. For example, I always use this when moving money from one bank account to another. In this case, both CSV files contain the transaction. One shall be ignored.
- `modify_transactions` (optional): A 2D list containing [old_regexp, replacement] to modify transactions if needed. This operates on the original input data without modifying it. For example, you may want to replace semicolons by commas: `[';', ',']`. The order of modifications is obviously important.

The utility script also removes non-ASCII characters from the input file.

You may wonder how Ledger's `convert` command actually matches transactions to Leger accounts? When defining accounts in Ledger, one may also provide a regular expression to identify an account by it's payee. For example:

```
account Expenses:Food:Groceries
 payee ^MegaSuperMarket
```

This not only defines the "Expense:Food:Groceries" account but also states that any transaction with the payee "MegaSupermarket" is associated with that account. This is where the `modify_transactions` variable from the `bankaccounts.yml` comes in handy: For example, when the same payee occurs in multiple transactions for different reasons you may want to modify a transaction to associate it with the correct account. You could for example modify the payee to include the transaction's note or description and then match the correct Ledger account by that combination (see the example above). Checkout `private/bankaccounts.yml` for some more ideas.

When starting to fill your journal, you will likely need to modify your account matching quite often and then just rerun the converter script. Later on, this is rarely needed.

After converting the CSV file to Ledger's format, it is saved in a temporary file in `./tmp/`. It might happen that you want to modify it then for whatever reasons.

4.3 Putting it all together with an example

We now have all the pieces to update the journal in a consistent and efficient way. The overall procedure is:

- Update `misc.tmp.txt` whenever you think of it or when you empty you wallet.
- For each electronic account (bank account/credit card/payment service/etc.):
  - Download the CSV file from the service provider
  - Convert the CSV file into a Ledger-formated file.
- Merge manually added & automatically generated data. (will be shown in the example below)
- Append it to the actual `journal.txt`. (will be shown in the example below)

There are aliases for most of the above steps. Check out the journal update section in `ecosystem/alias` and `private/alias.local` for details. Let's go over one example using the private repo.

4.3.1 Update `misc.tmp.txt`

```
$ mux start GSWL-private # if not done yet
# jump to the window 'edit' and open misc.tmp.txt
```

You will see that some transaction have already been added to that file. Feel free to add more postings. Just make sure you stay within the "current" month for the sake of the example (we're in May 2042 by the way . . .).

4.3.2 Get data from NorthBank

In the sample repo, two banks a used as placeholders: NorthBank & SouthBank. Imagine you would now go to your NorthBank online banking site and download the CSV data for the last month (the sample repo already contains this file):

```
$ mux start GSWL-private # if not done yet
# not really using wget obviously!
$ wget https://banking.northbank.com/myaccount/latest.csv CSV/apr2042_northbank.csv
```

Note how I renamed the CSV file to a machine readable name (i.e. added the date in a consistent manner). This now enables us to parse the ledger file with a simple alias:

```
$ mux start GSWL-private # if not done yet
# jump to the window 'make'
$ lmnorthbank # lm =~ last month, see private/alias.local
```

You should see the data from `CSV/apr2042_northbank.csv` converted into Ledger's format. The data is stored in `./tmp/northbank.tmp.txt` and shown with `less`; press `j/k/q` to move up/move down/quit. Some things of interest here:

- Sometimes, one needs to fix account matching or other stuff, so you may call `lmnorthbank` multiple times.

- In case you modified the output file by hand, a backup is stored in `./tmp/<bank>.tmp.txt.bak` to not overwrite your changes when running `lmnorthbank` again.
- See how Ledger automatically assigned the correct accounts. This is due to the `payee` directives in `GSWL-private/meta.txt`. In this example, you may want to try to match the book shop expenses correctly, too.
- One transaction has been ignored because money was moved to the SouthBank's account and also figures in that CSV file (see `bankaccounts.yml` for details).

Furthermore, you will have noticed that some transactions include more than 2 postings, namely the electricity & rent payments. This is due to Ledger's feature called "Automated Transaction". The Advanced chapter explains this in greater detail. For now, we'll keep the explanation short: The file `private/csv2journal.txt` is taken into account when converting CSV files. It contains automated transactions that should be applied to the actual transactions from the CSV file. This results in more than two Ledger account involved into a single bank transaction. When and how this can be used completely dependents on you. Again, the rent & electricity example is explained further below and we can safely skip this for now.

4.3.3 Get data from SouthBank

This is much like NorthBank:

```
$ mux start GSWL-private # if not done yet
# jump to the window 'make'
$ wget https://banking.southbank.com/account/data.csv CSV/apr2042_southbank.csv
$ lmsoutbhank
```

4.3.4 Merging everything

At this point, we can merge the different sources (`misc.tmp.txt` & bank account files in `private/tmp/*.txt`) and append them to `journal.txt`. The folder should now look like this:

```
$ ls CSV # check whether we have "downloaded" the CSV files
apr2042_northbank.csv  apr2042_southbank.csv
$ ls tmp # check whether we converted the CSV files
apr2042_northbank.csv.tmp  apr2042_northbank.csv.tmp.bak
apr2042_southbank.csv.tmp  apr2042_southbank.csv.tmp.bak
```

Merging is achieved by the `lmmake` command (see `ecosystem/alias`). What this basically does is:

- Concatenate the different sources into one ledger file.
- Sort the merged file's transactions by date.
- Checkout a clean version of `journal.txt`
- Append the new transactions to `journal.txt`
- Output the new transactions with `less`.

Let's go:

```
$ mux start GSWL-private # if not done yet
# jump to the window 'make'
$ lmmake
2042/04/08 * Mr. Scrooge
    ; CSV data:
    ; from : 08/04/2042,xxx,xx-xx-xx,xxxxxxxx,Mr. Scrooge,520.00,,
    ; (raw): 08/04/2042,xxx,xx-xx-xx,xxxxxxxx,Mr. Scrooge,520.00,,
    Expenses:Rent                          $520.00
    Assets:NorthBank:Checking             $-520.00
    Expenses:Rent                         $-260.00
    Receivables:Flatmates                  $260.00


2042/04/10 * Swimming
    Expenses:Sports:Swimming                $10.00
```

```
    Expenses:Unknown

2042/04/13 * Cinema
    Expenses:Cinema                          $15.00
    Expenses:Unknown
...

$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   journal.txt

no changes added to commit (use "git add" and/or "git commit -a")
$ git add journal.txt
$ git commit -m "Updated journal for April 2042"
```

That's it! Use `lmclean` to wipe out everything in `./tmp` though you don't really need to do this. For the next update, remember to clean up `mist.tmp.txt`.

A journal update is the right moment to fully appreciate the `ledreports` command because some of the reports defined in `private/reports.txt` refer to the "last month" (april 2042 in our example case). See above.

5 Advanced

This chapter introduces some advanced uses of Ledger.

5.1 Formatting

Although the default output of Ledger's report is sufficient in most cases, you sometimes want to change aspects of the output. Here are a couple of examples to get you started:

```
# make big expenses bold
$ led reg --bold-if "amount>100" ^Expenses

# Cut account names
$ led reg --account-width 10

# Assume bigger terminal size
$ led reg -w

# Double the amount of each posting
$ led bal --amount '2*a'

# Invert all amounts
$ led bal --invert
$ led bal --amount '-a'

# Show subtotals as percentage
$ led bal --percent
$ led bal --%
# If you use multiple currencies, you may need to specify --exchange (-X) to calculate percentages
$ led bal --exchange EUR --percent

# Omit the last line in the balance report
$ led bal --no-total
```

The output format of any Ledger report can be customized even more. In general, there is a format flag for each type of report, say `--balance-format` or `--register-format`. However, you can always use `--format` (`-F`). This flag allows you to completely change any aspect of the report.

The default format of the balance report is:

```
led bal -F '%(ansify_if(justify(scrub(display_total), 20, 20 + int(prepend_width), true, color), \
    bold if should_bold)) %(!options.flat ? depth_spacer : "") \
    %-(ansify_if(ansify_if(partial_account(options.flat), blue if color), bold if should_bold)) \
    \n%/%$1\n%/%(prepend_width ? " " * int(prepend_width) : "")--------------------\n'
```

This looks a bit scary at first, so let's go over this one by one:

```
%(ansify_if(
  justify(scrub(display_total), 20,
          20 + int(prepend_width), true, color),
            bold if should_bold))
```

The above function roughly says: "Output the total amount (`display_total`), justify it and prepend 20 characters. And make it bold if needed". This and the latter strings are executed for each posting of the transaction. For the balance report, this means for each account. The next line tells Ledger to put some whitespace for subcatogorical accounts. (Try out `--flat` to see the difference):

```
%(!options.flat ? depth_spacer : "")
```

What follows is the account name; written in blue and bold if needed:

```
%-(ansify_if(
   ansify_if(partial_account(options.flat), blue if color),
             bold if should_bold))\n
```

The remaining part is special. The sequence '%/' separates the format string into stuff which should be printed for the first posting of each transactions and what should be printed for all postings. In the balance report, there is only "one" transaction. '$1' refers to the first element of the previous lines, in this case the total amount:

```
%/%$1\n
%/%(prepend_width ? " " * int(prepend_width) : "") --------------------\n")
```

Using the above pieces, you could create your own report format. As an example, the `ecoystem/alias` defines the function `ledbalper` which make use of this feature: The defined report format resembles the usual balance format but adds a percentage column. Try out `ledbalper Expenses` in the private repo to get an impression of how that looks like. Often, you will want to have the output sorted by total. This is achieved with `ledbalper --sort "T" Expenses:Transportation` or `ledbalper --sort "T" --flat Expenses:Transportation Expenses:MobileCommunication` (use `--flat` when combining subcategories on the same hierarchy level).

5.2 Virtual postings

A virtual posting is not a real posting (sic!). Hence, virtual postings do not count when balancing out the transaction's postings to zero. A normal transaction ...

```
2042/01/25 * Pizza
    Expenses:Holidays            $20.00
    Assets:Cash
```

... becomes:

```
$ ledger bal
             $-20.00  Assets:Cash
              $20.00  Expenses:Holidays
--------------------
                   0
```

Whereas a transaction with virtual postings doesn't need to balance to zero:

```
2042/01/25 * Pizza
    ; Spent the money during holidays. But actually for food.
    Expenses:Holidays           $20.00
    Assets:Cash
    (Expenses:Food)             $20.00
```

Balance:

```
$ ledger bal
             $-20.00  Assets:Cash
              $40.00  Expenses
              $20.00    Food
              $20.00    Holidays
--------------------
             $40.00
```

Or:

```
$ ledger bal --real
             $-20.00  Assets:Cash
              $20.00  Expenses:Holidays
--------------------
                   0
```

That is, any virtual posting may be omitted by providing the `--real` argument:

```
$ ledger bal Food
             $-20.00  Expenses:Food

$ ledger bal Food --real
# empty
```

You may use brackets (they look “more strict”) instead of parentheses to force virtual postings to balance out:

```
2042/01/25 * Pizza
    Expenses:Holidays            $20.00
    Assets:Cash
    [Expenses:Food]             $-20.00
    [Equity:Food]                $20.00
```

You’ll ask what’s the big deal about this? Well, virtual postings are very handy in combination with automated transactions . . .

5.3 Automated Transactions

An automated transaction is like a normal transaction except that it’s header line does not contain the date but rather specifies under which circumstances the automated transaction should amend its postings to another transaction. Automated transaction need to be specified before any transaction they should apply to. An automated transaction is introduced with a “=”. The posting’s amount may either be a total amount (in a commodity) or a percentage value.

Examples:

```
; Whenever the posting's account matches 'food', add 100% of the value
; to it's corresponding account in the budget.
= food
    (Budget:$account)                 1

2042/01/25 * Pizza
    Expenses:Food                $20.00
    Assets:Cash
```

When running through Ledger, the above entry becomes:

```
    2042/01/25 * Pizza
        Expenses:Food                $20.00
        (Budget:Expenses:Food)       $20.00
        Assets:Cash                 $-20.00
```

Or:

```
; When encountering Income:Sales, add 19% of the posting's
; value to Liabilities:Taxes.
= /^Income:Sales$/
    (Liabiliites:Taxes)             0.19

2042/01/25 * Gotchas
    ; sold 43 gotchas the other day
    Income:Sales         (43 * -$39.99)
    Equity
```

Becomes:

```
$ ledger reg
42-01-25 Gotchas               Income:Sales             $-1719.57    $-1719.57
                               Equity                    $1719.57            0
                               (Liabilites:Taxes)         $-326.72     $-326.72
```

The following example (get Gist online) refers to the automated transaction employed during the journal update:

```
; I live together with a flatmate. He transfers me money every month to cover
; for the rent & utilities. I pay the bills for all flatmates. Hence, the total
; amount of money I transfer to say the electricity company is not what I spend
; myself on electricity. The automated transactions below splits up the money I
; receive from my flatmate into the different accounts and reduce the money I
; actually pay.

= expr account =~ /Expenses:Utilities:Phone/
    ; Make it look like paying $15 less when paying for the phone bill
    ; and expect that amount from the flatmates.
    Expenses:Utilities:Phone               $-15
    Receivables:Flatmates                  $15
= expr account =~ /Expenses:Utilities:Electricity/
    ; Make it look like paying 50% less.
    Expenses:Utilities:Electricity         -0.5
    Receivables:Flatmates                  0.5
= expr account =~ /Expenses:Rent/ and payee =~ /Scrooge/
    ; Only deduct when paying money to that specific landlord.
    ; Use '$account' as a placeholder to not repeat the account's name.
    $account                               $-150
    Receivables:Flatmates                  $150

; Here come the real transactions ...

2042/01/15 * John Doe
    ; Here I received the money from my flatmate.
    Receivables:Flatmates               $-205
    Assets:Checking

2042/01/23 * Mr. Scrooge
    ; Paying the rent to the landlord.
    Expenses:Rent                       $300
    Assets:Checking

2042/01/25 * TalkTalkTalk Inc.
    ; Paying the phone bill.
    Expenses:Utilities:Phone            $30
    Assets:Checking

2042/01/31 * HamsterWheel Ltd.
    ; Paying for electricity.
    Expenses:Utilities:Electricity      $80
    Assets:Checking
```

From the recorded transactions above, we would expect to pay \$300 + \$30 + \$80 = \$410 to the various parties. However, due to the automated transaction, the money received from the flatmate is used to reduce this amount by 50%:

```
$ ledger -f sample.txt bal ^Expenses
               $205  Expenses
               $150    Rent
                $55    Utilities
                $40      Electricity
                $15      Phone
--------------------
               $205

$ ledger -f sample.txt reg ^Expenses
42-01-23 Mr. Scrooge            Expenses:Rent                 $300         $300
```

```
                            Expenses:Rent                $-150         $150
42-01-25 TalkTalkTalk Inc.  Expens:Utilities:Phone         $30         $180
                            Expens:Utilities:Phone         $-15         $165
42-01-31 HamsterWheel Ltd.  Ex:Utiliti:Electricity         $80         $245
                            Ex:Utiliti:Electricity         $-40         $205
```

Grab the sample journal here. You may use `--actual` to ignore the automated transactions. On the other hand, `--generated` will explicitly include auto-generated postings in the resulting journal. Go give both command line switches a try. (By the way: `--generated` is used in `ecosystem/convert.py`.)

Here's another automated transaction example: I have a liability insurance and a household insurance both provided by the same insurance company. Whenever they withdraw money from my bank account, I want that money to be split up among the different insurance accounts.

```
; Note: An automated transaction applies to all matching postings.
; Matching by payee would apply the auto. trans. to all postings of a
; transaction. But we only want to apply it once. Hence, we will take
; the posting with the positive amount.
= expr payee =~ /Insurance Company X/ and amount > 0
    Expenses:Insurance:Liability         $5.31
    Expenses:Insurance:Household         $3.87
    Expenses:Unknown                    $-9.18
```

Given the above, the following transaction:

```
2042/04/01 * Insurance Company X
    Expenses:Unknown                     $9.18
    Assets:Checking                     $-9.18
```

Becomes:

```
2042/04/01 * Insurance Company X
    Expenses:Unknown                     $9.18
    Assets:Checking                     $-9.18
    Expenses:Insurance:Liability         $5.31
    Expenses:Insurance:Household         $3.87
    Expenses:Unknown                    $-9.18
```

5.4 Resetting a balance

It may be possible that one of your Ledger account does not match it's value in real life. In this situation, Ledger allows you to set the account's value to a specific amount. This is achieved by using the "=" operator in front of the posting's amount.

```
2042/04/01 Adjusting Checking account
   Assets:Checking                  = $1190.63
   Equity:Adjustments
```

6 Investing with Ledger

Like any other bookkeeper, Ledger allows you to track your investments. In the following, an "investment" is any asset that is not in dollars (or your local currency) but convertible to it. These assets will mainly be stocks but could potentially be anything. This chapter deals with investments in more detail but the general knowledge on how to deal with other commodities can be applied for other situations.

6.1 Dealing with commodities & market values

Ledger doesn't make a difference between commodities (apples, stocks, ...) and currencies (USD, EUR, ...). Although it does not matter, it is common practice to put a currency symbol before the amount whereas a commodity symbol will be put behind. Sample currencies/commodities:

```
$42.00          ; USD (currency)
€42.00          ; Euro (currency)
42 Apple        ; Apples (commodity)
42 AAPL         ; Shares (commodity)
```

Note that Ledger will pay attention to the format used for any commodity/currency and stick it accordingly. This is not only true foe the symbol's position but also for white spaces or thousand marks ("$5,000").

In the following, "commodity" and "currency" will be used interchangeably.

To allow Ledger to deal with different commodities, it has to know how to convert them. This is done by defining one commodity's value in terms of the other. Two approaches can be used: Either define the exchange rate in a specific price database (a text file, of course!). Or specifically mention the rate when adding a journal entry where a commodity needs to be converted. The latter is quite intuitive:

```
2042/05/01 * Opening Balance
    ; $5,000 in the bank.
    Assets:Checking                    $5,000.00
    Equity:Opening Balances

2042/05/18 * Buying Stock
    ; "Converting" $1500 into 50 AAPL. Exchange rate is $30 per share.
    Assets:Broker                      50 AAPL @ $30.00
    Assets:Checking

2042/05/28 * Selling Stock
    ; Selling 10 shares which have doubled their value.
    Assets:Broker                      -10 AAPL @ $60.00
    Assets:Checking
```

Now, looking at some reports:

```
$ led --flat bal Assets
             40 AAPL  Assets:Broker
          $4,100.00  Assets:Checking
--------------------
          $4,100.00
             40 AAPL

$ led reg Assets
42-05-01 Opening Balance        Assets:Checking          $5,000.00    $5,000.00
42-05-18 Buying Stock           Assets:Broker              50 AAPL    $5,000.00
                                                                        50 AAPL
                                Assets:Checking         $-1,500.00    $3,500.00
                                                                        50 AAPL
42-05-28 Selling Stock          Assets:Broker              -10 AAPL    $3,500.00
                                                                        40 AAPL
```

```
                            Assets:Checking            $600.00    $4,100.00
                                                                    40 AAPL
```

Forcing Ledger to display everything in a specific currency is achieved using `--exchange` or `-X`:

```
$ led --flat -X $ bal Assets
           $2,400.00  Assets:Broker
           $4,100.00  Assets:Checking
--------------------
           $6,500.00

$ led -X $ reg Assets
42-05-01 Opening Balance        Assets:Checking          $5,000.00    $5,000.00
42-05-18 Buying Stock           Assets:Broker            $1,500.00    $6,500.00
                                Assets:Checking         $-1,500.00    $5,000.00
42-05-28 Commodities reval ued  <Revalued>               $1,500.00    $6,500.00
42-05-28 Selling Stock          Assets:Broker             $-600.00    $5,900.00
                                Assets:Checking            $600.00    $6,500.00
```

While defining exchange rates on a per transaction base is handy for the daily work, it does not provide the possibility to reflect current market valuations. For example, if one bought some shares a year ago, their value has most probably changed. How could Ledger know? A simple text file can be used to associate specific dates to exchange rates. The file's content may look like this:

```
; On that particular day, 1 bitcoin was worth 4242 Ether.
P 2042/02/29 10:00:00 BTC 4242 ETH
; On that particular day, 1 bitcoin was worth $1337.
P 2042/02/29 10:00:00 BTC 1337 $
; On that particular day, 1 share of AAPL was worth $3.14
P 2042/02/29 10:00:00 AAPL 3.14 $
```

Having defined such a database, one can get the current market values by:

```
$ ledger --price-db <filename> --market balance
```

Every once in a while, one can append current prices to the database. This allows the balance report to reflect the "real" values of any asset.

The `led` command defined in `ecosystem/alias` expects the price database to be the file `prices.txt` and always reports current (=latest) market values.

6.2 Reporting gain & loss

Let's have a look again at the last balance report from above:

```
$ led -X $ bal Assets
           $2,400.00  Assets:Broker
           $4,100.00  Assets:Checking
--------------------
           $6,500.00
```

Remember that there were $5,000 in the checking account initially. Buying shares did not change the total amount of assets: $3,500 + 50 AAPL (valued $1,500). It is only on sell day that this figure changes. After having sold 10 shares for $60 each, a total of $600 is add to the checking account and removed from the broker account. And: The remaining 40 shares now value $60 each, too. Hence, The checking account values $3,500 + $600 = $4,100 while the broker account values (50 - 10) * $60 = $2,400.

A total gain of $1,500 was achieved due to the shares doubling in valuation. This can be seen using `--gain`:

```
$ led -X $ --gain bal Assets
$ ledger --gain bal
           $1,500.00  Assets:Broker
```

```
$ led --gain reg
42-05-28 Commodities reval ued  <Revalued>               $1,500.00    $1,500.00
```

6.3 Asset Allocation

You may want to know how your money is distributed among different asset classes. This can be easily achieved by having distinct "allocation" accounts which will serve as placeholders whenever money is put into any asset class. Using automated transactions & the virtual allocation accounts allow to get an easy overview. Consider the following:

```
account AssetAllocation:P2PLending
account AssetAllocation:Bonds
account AssetAllocation:Stocks

= /Receivables:P2PCompanyX/ or /Assets:P2PCompanyY/
    (AssetAllocation:P2PLending)              1

= expr (commodity == 'AAPL')
    (AssetAllocation:Stocks)                  1

= expr (commodity == 'FundsWithStocksAndBonds')
    (AssetAllocation:Stocks)                0.3
    (AssetAllocation:Bonds)                 0.7
```

(This could be appended to the `meta.txt`.) Having the above automated transactions setup will keep track of all the investments in the virtual `AssetAllocation` accounts, too. One can then easily run:

```
$ led bal [--percent] AssetAllocation
```

to get a nice overview of the asset distribution. The advantage of this approach is that different accounts can be merged into one asset class for example. Or the other way around: Split up money from one account into different asset classes. The output will be something like

```
$ led bal --percent AssetAllocation
             100.00%  AssetAllocation
              17.10%    Stocks
              43.95%    P2PLending
              13.51%    Bonds
              25.43%    Cash
```

7 The End

This is the end! I hope you enjoyed the ride.

If you have any questions, suggestions or general feedback, reach out for me via rolf dot schr at gmail dot com.

7.1 Contributions

The following people helped getting this book finished:

- Ben Finney (general feedback)
- Daniele Pitrolo (proof reading)
- Christian Sillaber (general feedback, proof reading & Vagrant setup)
- Simon Michael (general feedback & proof reading)
- Trannie Carter (epub version)

Many thanks to them!