Prise en main de Ledger

1er août 2019
f168418

Rolf Schröder & Contributeurs

Table des matières

Suppositions & Promesses

Ce livre est écrit pour les nouveaux utilisateurs de Ledger (surprise !). Ledger est un outil en ligne de commande et je m'attends donc à ce que le lecteur soit familier avec la ligne de commande en général. Vous devez savoir ce qu'est un terminal, comment exécuter des commandes quelconques ou installer un nouveau logiciel via la ligne de commande.

Vous n'avez pas besoin de savoir comment programmer mais ça aide probablement si vous avez déjà lu du code.

Le livre se veut concis. Cela implique une certaine expérience technique et une attitude de bricoleur de la part du lecteur.

D'autre part, vous disposerez d'un environnement de travail prêt à l'emploi pour commencer à utiliser Ledger. Les détails techniques ne sont pas cachés et le code est distribué gratuitement. Vous pouvez modifier tous les aspects du flux de travail et du code présentés dans le livre.

Vous serez peut-être surpris de voir à quel point le livre est court. Cela est dû au principe de ne pas se répéter, qui a été suivi dans ce livre : Rien de ce qui peut être vu avec un minimum d'effort en code n'est répété dans le livre. Vous obtiendrez toujours une indication sur l'endroit où obtenir l'information. Cela simplifie la vie de l'auteur, rend le livre moins sujet aux erreurs et donne au lecteur de nombreuses occasions de comprendre ce qui se passe derrière le rideau.

1 Une introduction à Ledger

Ce chapitre présente la philosophie de la comptabilité en partie double, Ledger comme outil en ligne de commande et son utilisation de base.

1.1 Comptabilité en partie double

La comptabilité en partie double est une approche comptable standard. En comptabilité, chaque type de dépenses ou de revenus et chaque "emplacement" qui détient une valeur monétaire est appelé un "compte" (pensez "catégorie"). Des exemples de comptes peuvent être "Epicerie", "Vélo", "Vacances", "Compte chèque de la Banque X", "Salaire" ou "Hypothèque". Dans la comptabilité en partie double, on suit le flux d'argent d'un compte à l'autre. Un montant d'argent figure toujours deux fois ("double") dans les registres : A l'endroit d'où il vient et à l'endroit où il a été déplacé. C'est-à-dire, ajouter 1000€ *ici* signifie retirer 1000€ de *là* en même temps. En conséquence, *le solde total de tous les comptes est toujours nul*. L'argent n'est jamais ajouté à un compte sans indiquer d'où vient exactement le même montant. Toutefois, plus de deux comptes peuvent être impliqués dans une même transaction.

Par exemple, l'achat d'un livre en ligne pour 15€ déplace l'argent du compte "Carte de crédit X" vers le compte "Livres". Recevoir un salaire de 2000€ de votre patron signifie transférer 2000€ du compte "Salaire" au compte "Banque" (ou autre). L'achat de produits d'épicerie et de détergents au supermarché peut faire passer de l'argent de la "Carte de crédit X" à l'"Épicerie" et au "Ménage".

En général, les noms de compte dépendent de la situation. Mais, on a habituellement les comptes principaux suivants :

- Dépenses
- Revenus
- Actifs
- Passifs
- Créances
- Fonds propres

Le niveau de détail requis pour les sous-catégories ("Dépenses" -> "Epicerie" -> "Fruits" -> "Bananes") est à la hauteur des exigences.

1.2 Ledger

Ledger est un outil en ligne de commande de comptabilité en partie double créé par John Wiegley avec une communauté de collaborateurs actifs. C'est un outil extrêmement puissant et il faut du temps et des efforts pour être en mesure de libérer sa puissance. Cependant, une fois maîtrisé, il n'y a pas grand-chose qui peut vous manquer lorsque vous faites de la comptabilité personnelle ou professionnelle.

Une documentation détaillée est disponible à l'adresse suivante http://ledger-cli.org.

L'utilisation de Ledger se résume à deux types d'action distincts : Mise à jour de la liste des transactions (le "journal") et utilisation de Ledger pour visualiser/interpréter ces données.

Ledger suit les bonnes vieilles traditions Unix et stocke les données dans des fichiers texte en clair. Ces données comprennent principalement le journal avec les transactions et quelques méta-informations. Une transaction typique dans Ledger ressemble à ceci :

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

Toute transaction commence par une ligne d'en-tête contenant la date et quelques méta-informations (dans le cas ci-dessus seulement un commentaire décrivant la transaction). L'en-tête est suivi d'une liste des comptes impliqués dans la transaction (un "enregistrement" par ligne, chaque ligne commençant par un espace blanc). Les comptes ont des noms arbitraires, mais Ledger utilise les deux points pour distinguer les sous-catégories. Le nom du compte est suivi d'au moins deux espaces blancs et du montant d'argent qui a été ajouté (positif) ou supprimé (négatif) de ce même compte. En fait, Ledger est assez intelligent pour calculer le montant approprié aussi il aurait été parfaitement valide de n'écrire que :

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

Le fichier journal est aussi simple que cela et il n'y a pas grand-chose à en savoir pour le moment. Notez que Ledger ne modifie jamais vos fichiers.

Les transactions suivantes illustrent quelques concepts de base utilisés dans la double comptabilité et 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
```

L'exemple ci-dessus a déjà introduit quelques concepts sympathiques de Ledger. Cependant, la lecture du fichier texte est un peu ennuyeuse. Avant de laisser Ledger l'analyser pour nous, vous devrez probablement en premier lieu l'installer ...

1.3 Installation de Ledger

La dernière version de Ledger peut être obtenue sur son site Web. Je recommande d'avoir au moins la version 3.1 fonctionnelle.

D'autres dépendances pour l'écosystème présenté dans ce livre sont :

— Git
— Python

Facultatif mais recommandé :

— gnuplot
— tig
— tmux
— tmuxinator

1.3.1 Linux & BSD

Vous trouverez ce dont vous avez besoin sur le site de téléchargement.

Lorsque vous utilisez Linux, ce pourrait être juste une question de :

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

Cependant, le paquet de la distribution peut être plus ancien que celui fourni sur le site de téléchargement. Ledger est livré avec une très bonne documentation d'installation. Reportez-vous à la page Github pour plus de détails.

1.3.2 macOS / OS X / Mac OS X

La façon la plus simple d'installer Ledger sur un Mac est avec Homebrew. Installez Homebrew en utilisant la méthode actuellement recommandée, puis installez Ledger avec une simple commande :

```
$ brew install ledger
```

1.3.3 Windows

Ledger est difficile à exécuter sous Windows (vous auriez probablement besoin de le compiler vous-même et c'est souvent un casse-pieds sous Windows). De plus, l'installation présentée dans ce livre fait un usage intensif de l'infrastructure traditionnelle de la ligne de commande Unix. Je recommande donc d'installer VirtualBox et d'installer Ledger sur une machine Linux. Vous pouvez utiliser VirtualBox simple ou VirtualBox avec Vagrant par dessus. Ce dernier est probablement plus facile et plus rapide. Il vous sera tout à fait possible de vous connecter à votre machine virtuelle via SSH dans Windows par la suite, vous n'aurez donc pas besoin "d'utiliser" l'environnement Linux.

Instructions étape par étape (sans Vagrant) :

— Télécharger et installer VirtualBox.
— Téléchargez une distribution ISO d'une distribution Linux de votre choix (Ubuntu?).
— Installez Linux sur la machine virtuelle.
— Installez le serveur OpenSSH *server* ("`$ sudo apt-get install openssh-server`" pour Ubuntu).
— Assurez-vous que vous pouvez accéder à la vm via SSH.
— Exécutez la machine en mode headless si vous le souhaitez.
— Installez babun (construit sur Cygwin) sur votre machine Windows.
— Connectez-vous à la vm via SSH.
— Suivez les instructions pour installer Ledger sous Linux.

Instructions étape par étape (avec Vagrant) :

— Télécharger et installer VirtualBox & Vagrant.
— Téléchargez ce Vagrantfile depuis Github.
— Ouvrez un terminal, allez à l'emplacement du Vagrantfile et lancez `vagrant up` (ceci configurera une machine Ubuntu avec Ledger installé).
— Pour se connecter à la VM via SSH, utilisez `vagrant up` suivi de `vagrant ssh` depuis le même dossier.

1.4 Un premier avant-goût

Avec une installation fonctionnelle de Ledger sur votre machine, récupérez ces exemples de transactions depuis Github (cliquez sur le bouton 'Raw') et copiez-les dans un fichier texte appelé `journal.txt`. Alors, lancez ceci :

```
$ # 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
```

Cela devrait vous donner une première impression sur Ledger. Vous en verrez plus dans le chapitre Rapports plus loin. Mais d'abord, nous devons mettre en place notre propre écosystème Ledger.

2 L'installation

Tout au long de ce livre, nous utiliserons une configuration de Ledger spécifique. Deux dépôts sont utilisés pour séparer le code et les données. Le dossier "ecosystem" (voir GSWL-ecosystem) contient les scripts et d'autres choses pour manipuler le journal. D'autre part, le dossier "privé" contiendra les données financières réelles. J'ai fourni un exemple de dossier privé (voir GSWL-private) que nous utiliserons comme référence.

La division du code et des données permet de ne chiffrer que ce qui est nécessaire, de partager plus facilement le code commun et de permettre un contrôle de version indépendant.

Le Readme de GSWL-private explique comment cloner les dépôts. Allez-y, allez les chercher.

Regardons le contenu des dépôts :

```
$ 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 Fichiers communs

L'écosystème contient du code pour traiter les données actuelles de manière intelligente (automatique !). Les scripts `convert.py` et `report.py` aident à intégrer des données CSV externes ou à interpréter les données respectivement. Le fichier `alias` est un script BASH qui définit les alias et fonctions courants. Vous pouvez consulter ces fichiers maintenant, mais nous les traiterons plus tard après avoir consulté le dépôt privé.

2.2 Données privées

Il y a beaucoup de dossiers dans le dépôt privé. Seuls les plus importants sont couverts pour l'instant.

Rappelez-vous que Ledger s'attend à ce que le fichier journal soit fourni par `-f`. Les alias définis dans l'écosystème supposent que ce fichier s'appelle `main.txt`. Cependant, ce fichier ne contient aucune donnée transactionnelle. Le fichier ne contient qu'une liste d'instructions `include`. Cela peut ressembler à ceci :

```
; 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
```

L'utilisation d'instructions `include` est un bon moyen de séparer les choses qui ne sont pas faites l'une pour l'autre. Il vous permet également d'essayer de nouvelles configurations ou données pour Ledger sans polluer vos fichiers dont la version est contrôlée.

Les transactions réelles sont toutes enregistrées dans `journal.txt`. Consultez le fichier `journal.txt` du dossier privé pour obtenir un premier aperçu d'un exemple de données.

2.2.1 Méta de départ

Dans cette configuration, le fichier `meta.txt` doit contenir toutes les données de configuration de Ledger et toute autre donnée non transactionnelle.

Par exemple, une instruction utile dans ce fichier est `account`. Ceci vous permet de prédéfinir tous les comptes qui doivent être utilisés par Ledger. Ledger n'exige pas que vous le fassiez, mais c'est une bonne pratique de toute façon. De plus, nous utiliserons plus tard l'argument de ligne de commande `--pedantic` qui provoque une erreur dans Ledger lorsque des comptes inconnus sont utilisés. Les définitions de compte peuvent ressembler à ceci :

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

De même, `commodity` définit les devises valides en usage :

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

L'exemple `meta.txt` n'inclut en effet pas d'autre configuration. Ou est-ce le cas ?

2.2.2 Autres fichiers

Ensuite, il y a un script Bash appelé `alias.local` qui contient les configurations locales. Ce script est automatiquement source par `ledger-ecoystem/alias`. C'est peut-être le moment de jeter un coup d'oeil à ces scripts. Essayez de découvrir à quoi ressemble la commande `led`, quel est le contenu de la variable d'environnement `$LAST_AMN` et comment `alias.local` est source.

Enfin, les fichiers `bankaccounts.yml`, `csv2journal.txt` et `misc.tmp.txt` sont utilisés pour mettre à jour le journal de manière automatisée. `reports.txt` liste les questions posées à plusieurs reprises sur la situation financière. Tout cela est expliqué dans les chapitres suivants, mais n'hésitez pas à les inspecter immédiatement.

2.3 Orchestrer l'écoystème et les données privées

Vous devriez avoir le modèle mental suivant de la configuration présentée : La plupart du code se trouve dans le dossier ecosystem. Toutes les données actuelles se trouvent dans le dossier privé. Travailler avec Ledger signifie travailler dans le dossier privé. Pour libérer la puissance de tous les scripts etc., il est nécessaire de source `ecosystem/alias` dans le dossier privé. Ceci source `alias.local` à partir du répertoire de travail courant. Le fichier d'alias local permet d'écraser les fonctionnalités de l'écosystème ou d'ajouter de nouvelles fonctionnalités.

Le fait d'avoir l'écosystème de scripts et les données d'exemples disponibles permet d'avoir une idée plus précise du travail quotidien avec Ledger. Exécutez les commandes suivantes :

```
$ 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
```

Pour être tout à fait clair : `led` n'est qu'un alias à `ledger` combiné avec quelques arguments prédéfinis (voir `ecosystem/alias`). Vous pouvez bien sûr exécuter `ledger` sur les mêmes données. Dans ce cas, vous devrez au moins indiquer à Ledger où trouver le fichier journal : `ledger -f main.txt`. Pensez à `led` comme à `ledger` avec un fichier d'entrée prédéfini et des valeurs par défaut saines.

2.4 Tmux & Tmuxinator

Je recommande fortement l'utilisation de Tmux pour toutes les affaires que vous faites sur la ligne de commande. Cet outil accélère tellement votre flux de travail qu'il est en fait ridicule. Il s'agit d'une meilleure version de `screen` et "permet de basculer

facilement entre plusieurs programmes dans un terminal, de les détacher (ils continuent à fonctionner en arrière-plan) et de les rattacher à un autre terminal". Si vous ne l'utilisez pas jusqu'à présent, vous vous demanderez comment vous avez survécu avant. L'exemple de `.tmux.conf` dans le dépôt privé et ce HowTo vous permet de démarrer si vous en avez besoin. Jetez un coup d'oeil au fichier example `.tmux.conf` et assurez-vous de savoir au moins comment passer d'une fenêtre à l'autre, sauter entre les fenêtres, créer une nouvelle fenêtre et maximiser (redimensionner) un volet.

Tmuxinator s'appuis sur tmux et vous permet de prédéfinir des sessions tmux pour des tâches spécifiques. J'ai défini une session tmux spécifique à utiliser avec le dossier privé. Le fichier de session tmuxinator `.tmuxinator.ledger.yml` peut être trouvé dans le dépôt privé (consultez le maintenant !).

Démarrer une session tmux avec le dépôt privé (en supposant que tmux & tmuxinator sont installés) :

```
$ 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
```

Dans chaque fenêtre de session Tmux, `ecosystem/alias` est source.

L'exemple `GSLW-private/.bashrc` fournit quelques alias pour démarrer/arrêter les sessions Tmux. Vous devriez de toute façon trouver ce fichier dans votre `~/.bashrc`.

Avec la mise en place et le fonctionnement, nous pouvons maintenant continuer à jouer avec les fonctions actuelles de Ledger.

2.5 Votre propre configuration

Pour commencer avec votre configuration personnelle, passez à la caisse (sans jeu de mots) ceci.

3 Rapports

C'est bien beau d'avoir un journal, mais nous ne le gardons en fait que pour avoir un aperçu de notre situation financière. C'est là que les rapports arrivent. Un état affiche le journal d'une manière significative. Ledger peut produire des rapports de diverses façons, ce qui le rend extrêmement puissant. Les commandes les plus standard pour le reporting sont `balance` et `register`.

3.1 Rapports de solde

Le rapport de bilan est très intuitif. Il crée un solde total à partir de toutes les transactions. La commande de base est :

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

La sortie est quelque chose comme :

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

Normalement, vous voulez avoir un solde plus précis en mettant quelques restrictions sur le nom du compte, l'heure ou autre :

```
# 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 Rapports de registre

Les rapports de registre affichent le journal comme un registre à l'ancienne. Les exemples d'arguments de ligne de commande comme ci-dessus s'appliquent, bien sûr.

```
# 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 Filtrage avancé des rapports

Les informations rapportées (balance ou register) peuvent être filtrées de manière plus sophistiquée. Ceci est réalisé soit par `--limit` (`-l`) soit par `--display` (`-d`). La différence entre les deux est que la première limite les écritures à prendre en compte pour les calculs tandis que la seconde limite les écritures à prendre en compte pour l'affichage. Cela signifie que les arguments ("expression") utilisés conjointement avec `--limit` sont actifs pendant que Ledger parcourt le fichier journal. D'autre part, les expressions fournies à `--display` ne filtreront le résultat final *qu'après* avoir lu le journal complètement.

À titre d'exemple, considérons que l'on veut avoir un aperçu du montant habituellement dépensé chaque mois. Autrement dit, nous sommes intéressés par la moyenne des dépenses mensuelles au cours des x derniers mois. Ceci peut être facilement réalisé par `led -M -n -A -p "from 2041/11/01" register ^Expenses`. Allez l'essayer dans le dépôt privé. Le rapport qui en résultera fera état du total des dépenses mensuelles à partir de novembre 2041 et calculera la moyenne mobile (dernière colonne). Maintenant, revenons à notre filtrage : Imaginez que vous n'êtes intéressé que par la moyenne de tous les mois combinés. Cette information n'est disponible qu'après avoir pris en compte le dernier mois évidemment. Cependant, toutes les dépenses mensuelles antérieures sont nécessaires pour la calculer. C'est là que la différence entre `--limit` et `--display` peut être facilement vue :

```
# 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
```

Voyez en quoi la dernière valeur de la dernière colonne de la première ligne est différente de celle de la première commande. Ceci est dû au fait qu'il existe des données de journal antérieures au 2041/11/01 qui sont prises en compte pour le calcul de la moyenne lorsque l'on limite uniquement avec `--display`.

Combiner les deux :

```
# 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
```

Voyez comment la colonne moyenne change ? C'est exactement la différence entre le filtrage avant calcul ("limit") ou avant présentation des résultats ("display").

Prenons un autre exemple : le journal suivant :

```
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
```

Ici, quelqu'un vit au-dessus de ses moyens. Cette personne ne gagnait que 100 $, mais dépensait 200 $. La banque a apparemment accordé un prêt de 100 $:

```
$ 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
```

Disons que nous voulons examiner tous les comptes qui ont un solde positif. Les expressions employées seraient `amount > 0`. Mais selon que l'on utilise `--limit` ou `--display`, le résultat est très différent :

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

La sortie de `--display` semble assez intuitive. En fin de compte, le compte de dépenses a un solde de 200€ alors que les autres sont négatifs. L'utilisation de `--limit` ne considère que les écritures ("lignes") avec un montant positif : Pour la première transaction, cela signifie 50€ dans le compte des dépenses et 50€ dans le compte bancaire, pour la deuxième transaction nous avons 150€ de plus dans le compte des dépenses. On se retrouve donc avec un total de 250€.

La plupart du temps, `--display` est ce que vous voulez. En fait, le résultat de `--limit` & `--display` est souvent le même. Mais pas toujours :

```
# 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'
```

Vous trouverez de plus amples renseignements sur la façon de filtrer les rapports dans la documentation en ligne de Ledger.

3.4 Exemples de questions

Essayez d'obtenir les informations suivantes en utilisant le dépôt privé [question supplémentaire entre crochets]. Les réponses se trouvent à la page suivante.

(1) Combien d'argent a été dépensé en épicerie [depuis le 1er janvier 2042] ?
(2) Combien d'argent a été dépensé pour le loyer et l'électricité ?
(3) Combien d'argent a été dépensé au total chaque mois [en moyenne] ?
(4) Quel est le revenu "gagné" qui n'est pas un salaire ?
(5) Combien d'argent a été dépensé en cadeaux (nom du compte) sur Amazon (nom du bénéficiaire) ?

3.4.1 Réponses

(1) Combien d'argent a été dépensé en épicerie [depuis le 1er janvier 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) Combien d'argent a été dépensé pour le loyer et l'électricité ?

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

(3) Combien d'argent a été dépensé au total chaque mois [en moyenne] ?

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

(4) Quel est le revenu "gagné" qui n'est pas un salaire ?

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

(5) Combien d'argent a été dépensé en cadeaux (nom du compte) sur Amazon (nom du bénéficiaire) ?

Ceci ne peut être résolu que par `--limit` :

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

En regardant les réponses, on se rend compte que l'interrupteur `--limit` semble souvent encombrant. Néanmoins, dans certaines situations, il se peut que vous deviez recourir à des expressions plus puissantes.

3.5 Rapports récurrents

La plupart du temps, vous vous intéressez aux "suspects habituels". Personnellement, j'ai de 5 à 10 rapports, que je veux toujours vérifier après avoir mis à jour le journal. De toute évidence, c'est une perte de temps de les taper à la main. Une façon de simplifier les choses serait de définir des alias (probablement énigmatique) pour chacun de ces rapports. Une autre façon est d'utiliser le script `reports.py` fourni dans le dépôt écosystème. Le script ouvre le fichier `reports.txt` dans le répertoire de travail courant et affiche les rapports prédéfinis un par un. Le fichier texte doit contenir des sections séparées par des lignes vides composées de commentaires (commençant par #) et de commandes réelles à exécuter. Vous pouvez invoquer le tout avec la commande `ledreports` (cela se fait en fait automatiquement dans la fenêtre "overview" de la session tmux). Le fichier `reports.txt` du répertoire de travail actuel peut ressembler à ceci :

```
# 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
```

Le script `reports.py` montrera les sections ci-dessus comme 3 pages distinctes avec chacune un ou plusieurs rapports. Vous pouvez parcourir les rapports listés en utilisant `j` et `k`. Dans la fenêtre du terminal, chaque rapport est préfixé par le commentaire et la commande réelle pour vous aider à comprendre ce qui se passe. Notez que le script source le fichier `alias` à partir de l'écosystème dans le répertoire de travail courant afin d'autoriser les commandes habituelles. Rappelez-vous que le fichier `alias` lui-même essaie de trouver le fichier `alias.local` dans le répertoire de travail courant, ce qui vous permet d'ajouter facilement vos propres fichiers.

Essayez `ledreports` dans le repo privé :

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

3.6 Autres rapports

Tu veux essayer ça :

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

3.7 Visualisation

Ledger est livré avec deux commutateurs pratiques (`-j/-J`) pour permettre d'alimenter le gnuplot (ou d'autres outils) avec ses sorties pour visualiser les données. Le rapport de registre peut être modifié pour n'éditer que la date et le montant actuel (`-j`) ou le total courant (`-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
```

`ecosystem/alias` définit la fonction `ledplot`qui entoure gnuplot pour visualiser certaines données :

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

Certains des exemples de rapports contiennent quelques graphes prédéfinis que vous pouvez utiliser. Voir `private/reports.txt` pour plus d'informations.

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 these 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
```

Another good use for automated transactions is grouping of accounts. As an example, all fixed cost that have to be paid every month can be linked together like so:

```
account FixedCost

= /Rent$/ or /Electricity$/ or /Insurance$/ or /Telephone$/
    (FixedCost)                           1
```

The above statement allows to query for (any change in) the fixed cost with `led reg FixedCost`. You'll have to add the appropriate accounts, of course.

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 for 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 La fin

C'est la fin ! J'espère que vous avez aimé la balade.

Pour toutes questions, suggestions ou commentaires généraux, contactez-moi via github.

Si vous avez aimé ce livre, veuillez envisager de soutenir Ledger :

— ajouter, corriger ou surveiller bugs
— apporter sa contribution code

ou directement :

— améliorer la qualité de ce livre contenu EN contenu FR
— acheter une bière à rolfschr `bitcoin:1JWADV8azQDUqUS3HXEABiLGW65qzwCJi4`

7.1 Contributions

Les personnes suivantes ont aidé à terminer ou à mettre à jour ce livre :

— Ben Finney (general feedback)
— Christian Sillaber (general feedback, proof reading & Vagrant setup)
— Colin Dean (macOS installation instructions)
— Daniele Pitrolo (proof reading)
— Georg Lutz (ecosystem patches)
— Klaus Thoden (proof reading)
— Nikita Tchayka (improved html version)
— Max Beutelspacher (ecosystem patches)
— Samuel Marcaille (improved html version)
— Simon Michael (general feedback & proof reading)
— Trannie Carter (epub version)

Un grand merci à eux !