Clojure Data Analysis Cookbook  Second Edition

As data invades more and more of life and business, the need to analyze it effectively has never been greater. With Clojure and this book, you’ll soon be getting to grips with every aspect of data analysis. You’ll start with practical recipes that show you how to load and clean your data, then get concise instructions to perform all the essential analysis tasks from basic statistics to sophisticated machine learning and data clustering algorithms. Get a more intuitive handle on your data through hands-on visualization techniques that allow you to provide interesting, informative, and compelling reports, and use Clojure to publish your findings to the Web.

What this book will do for you...

- Read data from a variety of data formats
- Transform data to make it more useful and easier to analyze
- Process data concurrently and in parallel for faster performance
- Harness multiple computers to analyze big data
- Use powerful data analysis libraries such as Incanter, Hadoop, and Weka to get things done quickly
- Apply powerful clustering and data mining techniques to better understand your data

Inside the Cookbook...

- A straightforward and easy-to-follow format
- A selection of the most important tasks and problems
- Carefully organized instructions for solving the problem efficiently
- Clear explanations of what you did
- Apply the solution to other situations

What this book will do for you...

- Read data from a variety of data formats
- Transform data to make it more useful and easier to analyze
- Process data concurrently and in parallel for faster performance
- Harness multiple computers to analyze big data
- Use powerful data analysis libraries such as Incanter, Hadoop, and Weka to get things done quickly
- Apply powerful clustering and data mining techniques to better understand your data

Prices do not include local sales tax or VAT where applicable

Dive into data analysis with Clojure through over 100 practical recipes for every stage of the analysis and collection process


Eric Rochester

Dive into data analysis with Clojure through over 100 practical recipes for every stage of the analysis and collection process
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Importing Data for Analysis'
- A synopsis of the book’s content
- More information on Clojure Data Analysis Cookbook Second Edition

About the Author

Eric Rochester enjoys reading, writing, and spending time with his wife and kids. When he’s not doing these things, he programs in a variety of languages and platforms, including websites and systems in Python, and libraries for linguistics and statistics in C#. Currently, he is exploring functional programming languages, including Clojure and Haskell. He works at Scholars’ Lab in the library at the University of Virginia, helping humanities professors and graduate students realize their digitally informed research agendas. He is also the author of Mastering Clojure Data Analysis, Packt Publishing.

I’d like to thank everyone. My technical reviewers proved invaluable. Also, thank you to the editorial staff at Packt Publishing. This book is much stronger because of all of their feedback, and any remaining deficiencies are mine alone.

A special thanks to Jackie, Melina, and Micah. They’ve been patient and supportive while I worked on this project. It is, in every way, for them.
Welcome to the second edition of Clojure Data Analysis Cookbook! It seems that books become obsolete almost as quickly as software does, so here we have the opportunity to keep things up-to-date and useful.

Moreover, the state of the art of data analysis is also still evolving and changing. The techniques and technologies are being refined and improved. Hopefully, this book will capture some of that. I've also added a new chapter on how to work with unstructured textual data.

In spite of these changes, some things have stayed the same. Clojure has further proven itself to be an excellent environment to work with data. As a member of the lisp family of languages, it inherits a flexibility and power that is hard to match. The concurrency and parallelization features have further proven themselves as great tools for developing software and analyzing data.

Clojure's usefulness for data analysis is further improved by a number of strong libraries. Incanter provides a practical environment to work with data and perform statistical analysis. Cascalog is an easy-to-use wrapper over Hadoop and Cascading. Finally, when you're ready to publish your results, ClojureScript, an implementation of Clojure that generates JavaScript, can help you to visualize your data in an effective and persuasive way.

Moreover, Clojure runs on the Java Virtual Machine (JVM), so any libraries written for Java are available too. This gives Clojure an incredible amount of breadth and power.

I hope that this book will give you the tools and techniques you need to get answers from your data.

What This Book Covers

Chapter 1, Importing Data for Analysis, covers how to read data from a variety of sources, including CSV files, web pages, and linked semantic web data.

Chapter 2, Cleaning and Validating Data, presents strategies and implementations to normalize dates, fix spelling, and work with large datasets. Getting data into a useable shape is an important, but often overlooked, stage of data analysis.

Chapter 3, Managing Complexity with Concurrent Programming, covers Clojure's concurrency features and how you can use them to simplify your programs.

Chapter 4, Improving Performance with Parallel Programming, covers how to use Clojure's parallel processing capabilities to speed up the processing of data.
Chapter 5, *Distributed Data Processing with Cascalog*, covers how to use Cascalog as a wrapper over Hadoop and the Cascading library to process large amounts of data distributed over multiple computers.

Chapter 6, *Working with Incanter Datasets*, covers the basics of working with Incanter datasets. Datasets are the core data structures used by Incanter, and understanding them is necessary in order to use Incanter effectively.

Chapter 7, *Statistical Data Analysis with Incanter*, covers a variety of statistical processes and tests used in data analysis. Some of these are quite simple, such as generating summary statistics. Others are more complex, such as performing linear regressions and auditing data with Benford's Law.

Chapter 8, *Working with Mathematica and R*, talks about how to set up Clojure in order to talk to Mathematica or R. These are powerful data analysis systems, and we might want to use them sometimes. This chapter will show you how to get these systems to work together, as well as some tasks that you can perform once they are communicating.

Chapter 9, *Clustering, Classifying, and Working with Weka*, covers more advanced machine learning techniques. In this chapter, we'll primarily use the Weka machine learning library. Some recipes will discuss how to use it and the data structures its built on, while other recipes will demonstrate machine learning algorithms.

Chapter 10, *Working with Unstructured and Textual Data*, looks at tools and techniques used to extract information from the reams of unstructured, textual data.

Chapter 11, *Graphing in Incanter*, shows you how to generate graphs and other visualizations in Incanter. These can be important for exploring and learning about your data and also for publishing and presenting your results.

Chapter 12, *Creating Charts for the Web*, shows you how to set up a simple web application in order to present findings from data analysis. It will include a number of recipes that leverage the powerful D3 visualization library.
1 Importing Data for Analysis

In this chapter, we will cover the following recipes:

- Creating a new project
- Reading CSV data into Incanter datasets
- Reading JSON data into Incanter datasets
- Reading data from Excel with Incanter
- Reading data from JDBC databases
- Reading XML data into Incanter datasets
- Scraping data from tables in web pages
- Scraping textual data from web pages
- Reading RDF data
- Querying RDF data with SPARQL
- Aggregating data from different formats

Introduction

There's not much data analysis that can be done without data, so the first step in any project is to evaluate the data we have and the data that we need. Once we have some idea of what we'll need, we have to figure out how to get it.
Many of the recipes in this chapter and in this book use Incanter (http://incanter.org/) to import the data and target Incanter datasets. Incanter is a library that is used for statistical analysis and graphics in Clojure (similar to R) an open source language for statistical computing (http://www.r-project.org/). Incanter might not be suitable for every task (for example, we'll use the Weka library for machine learning later) but it is still an important part of our toolkit for doing data analysis in Clojure. This chapter has a collection of recipes that can be used to gather data and make it accessible to Clojure.

For the very first recipe, we'll take a look at how to start a new project. We'll start with very simple formats such as comma-separated values (CSV) and move into reading data from relational databases using JDBC. We'll examine more complicated data sources, such as web scraping and linked data (RDF).

### Creating a new project

Over the course of this book, we're going to use a number of third-party libraries and external dependencies. We will need a tool to download them and track them. We also need a tool to set up the environment and start a REPL (read-eval-print-loop or interactive interpreter) that can access our code or to execute our program. REPLs allow you to program interactively. It's a great environment for exploratory programming, irrespective of whether that means exploring library APIs or exploring data.

We'll use Leiningen for this (http://leiningen.org/). This has become a standard package automation and management system.

#### Getting ready

Visit the Leiningen site and download the lein script. This will download the Leiningen JAR file when it's needed. The instructions are clear, and it's a simple process.

#### How to do it...

To generate a new project, use the lein new command, passing the name of the project to it:

```
$ lein new getting-data
```

Generating a project called getting-data based on the default template. To see other templates (app, lein plugin, etc), try lein help new.

There will be a new subdirectory named getting-data. It will contain files with stubs for the getting-data.core namespace and for tests.
How it works...

The new project directory also contains a file named `project.clj`. This file contains metadata about the project, such as its name, version, license, and more. It also contains a list of the dependencies that our code will use, as shown in the following snippet. The specifications that this file uses allow it to search Maven repositories and directories of Clojure libraries (Clojars, https://clojars.org/) in order to download the project's dependencies. Thus, it integrates well with Java's own packaging system as developed with Maven (http://maven.apache.org/).

```clj
(defproject getting-data "0.1.0-SNAPSHOT"
  :description "FIXME: write description"
  :url "http://example.com/FIXME"
  :license {:name "Eclipse Public License"
  :dependencies [[org.clojure/clojure "1.6.0"]])
```

In the Getting ready section of each recipe, we'll see the libraries that we need to list in the :dependencies section of this file. Then, when you run any lein command, it will download the dependencies first.

Reading CSV data into Incanter datasets

One of the simplest data formats is comma-separated values (CSV), and you'll find that it's everywhere. Excel reads and writes CSV directly, as do most databases. Also, because it's really just plain text, it's easy to generate CSV files or to access them from any programming language.

Getting ready

First, let's make sure that we have the correct libraries loaded. Here's how the project Leiningen (https://github.com/technomancy/leiningen) `project.clj` file should look (although you might be able to use more up-to-date versions of the dependencies):

```clj
(defproject getting-data "0.1.0-SNAPSHOT"
  :dependencies [[org.clojure/clojure "1.6.0"
                   [incanter "1.5.5"]])
```

Download the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.
Importing Data for Analysis

Also, in your REPL or your file, include these lines:

```
(use 'incanter.core 'incanter.io)
```

Finally, downloaded a list of rest area locations from POI Factory at http://www.poi-factory.com/node/6643. The data is in a file named data/RestAreasCombined(Ver.BN).csv. The version designation might be different though, as the file is updated. You'll also need to register on the site in order to download the data. The file contains this data, which is the location and description of the rest stops along the highway:

-67.834062,46.141129,"REST AREA-FOLLOW SIGNS SB I-95 MM305","RR, PT, Pets, HF"
-67.845906,46.138084,"REST AREA-FOLLOW SIGNS NB I-95 MM305","RR, PT, Pets, HF"
-68.498471,45.659781,"TURNOUT NB I-95 MM249","Scenic Vista-NO FACILITIES"
-68.534061,45.598464,"REST AREA SB I-95 MM240","RR, PT, Pets, HF"

In the project directory, we have to create a subdirectory named data and place the file in this subdirectory.

I also created a copy of this file with a row listing the names of the columns and named it RestAreasCombined(Ver.BN)-headers.csv.

How to do it...

1. Now, use the incanter.io/read-dataset function in your REPL:

```
user=> (read-dataset "data/RestAreasCombined(Ver.BN).csv")
```

```
<table>
<thead>
<tr>
<th>:col0</th>
<th>:col1</th>
<th>:col2</th>
<th>:col3</th>
</tr>
</thead>
<tbody>
<tr>
<td>-67.834062</td>
<td>46.141129</td>
<td>REST AREA-FOLLOW SIGNS SB I-95 MM305</td>
<td>&quot;RR, PT, Pets, HF&quot;</td>
</tr>
<tr>
<td>-67.845906</td>
<td>46.138084</td>
<td>REST AREA-FOLLOW SIGNS NB I-95 MM305</td>
<td>&quot;RR, PT, Pets, HF&quot;</td>
</tr>
<tr>
<td>-68.498471</td>
<td>45.659781</td>
<td>TURNOUT NB I-95 MM249</td>
<td>&quot;Scenic Vista-NO FACILITIES&quot;</td>
</tr>
<tr>
<td>-68.534061</td>
<td>45.598464</td>
<td>REST AREA SB I-95 MM240</td>
<td>&quot;RR, PT, Pets, HF&quot;</td>
</tr>
<tr>
<td>-68.539034</td>
<td>45.594001</td>
<td>REST AREA NB I-95 MM240</td>
<td>&quot;RR, PT, Pets, HF&quot;</td>
</tr>
</tbody>
</table>
```

...
2. If we have a header row in the CSV file, then we include :header true in the call to read-dataset:

```clojure
user=> (read-dataset "data/RestAreasCombined(Ver.BJ)-headers.csv" :header true)
```

| :longitude | :latitude |                                :name                                |
|------------|-----------|---------------------------------|---------------------------------|
| -67.834062 | 46.141129 | REST AREA-FOLLOW SIGNS SB I-95 MM305 | RR, PT, Pets, HF |
| -67.845906 | 46.138084 | REST AREA-FOLLOW SIGNS NB I-95 MM305 | RR, PT, Pets, HF |
| -68.498471 | 45.659781 |                TURNOUT NB I-95 MM249 | Scenic Vista-NO FACILITIES |
| -68.534061 | 45.598464 |              REST AREA SB I-95 MM240 | RR, PT, Pets, HF |
| -68.539034 | 45.594001 |              REST AREA NB I-95 MM240 | RR, PT, Pets, HF |

How it works...

Together, Clojure and Incanter make a lot of common tasks easy, which is shown in the How to do it section of this recipe.

We've taken some external data, in this case from a CSV file, and loaded it into an Incanter dataset. In Incanter, a dataset is a table, similar to a sheet in a spreadsheet or a database table. Each column has one field of data, and each row has an observation of data. Some columns will contain string data (all of the columns in this example did), some will contain dates, and some will contain numeric data. Incanter tries to automatically detect when a column contains numeric data and converts it to a Java `int` or `double`. Incanter takes away a lot of the effort involved with importing data.

There's more...

For more information about Incanter datasets, see Chapter 6, Working with Incanter Datasets.
Reading JSON data into Incanter datasets

Another data format that's becoming increasingly popular is JavaScript Object Notation (JSON, http://json.org/). Like CSV, this is a plain text format, so it's easy for programs to work with. It provides more information about the data than CSV does, but at the cost of being more verbose. It also allows the data to be structured in more complicated ways, such as hierarchies or sequences of hierarchies.

Because JSON is a much richer data model than CSV, we might need to transform the data. In that case, we can just pull out the information we're interested in and flatten the nested maps before we pass it to Incanter. In this recipe, however, we'll just work with fairly simple data structures.

Getting ready

First, here are the contents of the Leiningen project.clj file:

```clojure
(defproject getting-data "0.1.0-SNAPSHOT"
  :dependencies [[org.clojure/clojure "1.6.0"]
                 [incanter "1.5.5"]
                 [org.clojure/data.json "0.2.5"]])
```

Use these libraries in your REPL or program (inside an ns form):

```clojure
(require '[incanter.core :as i]
         [clojure.data.json :as json]
         [clojure.java.io :as io])

(import '[java.io EOFException])
```

Moreover, you need some data. For this, I have a file named delicious-rss-214k.json and placed it in the folder named data. It contains a number of top-level JSON objects. For example, the first one starts like this:

```json
{
    "guidislink": false,
    "link": "http://designreviver.com/tips/a-collection-of-wordpress-tutorials-tips-and-themes/",
    "title_detail": {
        "base": "http://feeds.delicious.com/v2/rss/recent?min=1&count=100",
        "value": "A Collection of Wordpress Tutorials, Tips and Themes | Design Reviver",
        "language": null,
        "type": "text/plain"
    },
    "author": "mccarrd4",
    ...
}
```
You can download this data file from Infochimps at http://www.ericrochester.com/clj-data-analysis/data/delicious-rss-214k.json.xz. You'll need to decompress it into the data directory.

**How to do it...**

Once everything's in place, we'll need a couple of functions to make it easier to handle the multiple JSON objects at the top level of the file:

1. We'll need a function that attempts to call a function on an instance of `java.io.Reader` and returns nil if there's an EOFException, in case there's a problem reading the file:
   ```clojure
   (defn test-eof [reader f]
     (try
       (f reader)
       (catch EOFException e
         nil)))
   ``

2. Now, we'll build on this to repeatedly parse a JSON document from an instance of `java.io.Reader`. We do this by repeatedly calling `test-eof` until `eof` or until it returns `nil`, accumulating the returned values as we go:
   ```clojure
   (defn read-all-json [reader]
     (loop [accum []]
       (if-let [record (test-eof reader json/read)]
         (recur (conj accum record))
         accum)))
   ``

3. Finally, we'll perform the previously mentioned two steps to read the data from the file:
   ```clojure
   (def d (i/to-dataset
     (with-open
       [r (io/reader
           "data/delicious-rss-214k.json")
        (read-all-json r)]))
   ```

   This binds `d` to a new dataset that contains the information read in from the JSON documents.
How it works...

Similar to all Lisp's (List Processing), Clojure is usually read from the inside out and from right to left. Let's break it down. `clojure.java.io/reader` opens the file for reading. `read-all-json` parses all of the JSON documents in the file into a sequence. In this case, it returns a vector of the maps. `incanter.core/to-dataset` takes a sequence of maps and returns an Incanter dataset. This dataset will use the keys in the maps as column names, and it will convert the data values into a matrix. Actually, `to-dataset` can accept many different data structures. Try `doc to-dataset` in the REPL (`doc` shows the documentation string attached to the function), or see the Incanter documentation at [http://data-sorcery.org/contents/](http://data-sorcery.org/contents/) for more information.

Reading data from Excel with Incanter

We've seen how Incanter makes a lot of common data-processing tasks very simple, and reading an Excel spreadsheet is another example of this.

Getting ready

First, make sure that your Leiningen project.clj file contains the right dependencies:

```clojure
(defproject getting-data "0.1.0-SNAPSHOT"
 :dependencies [[org.clojure/clojure "1.6.0"]
                [incanter "1.5.5"]])
```

Also, make sure that you've loaded those packages into the REPL or script:

```clojure
(use 'incanter.core
     'incanter.excel)
```

Find the Excel spreadsheet you want to work on. The file name of my spreadsheet is `data/small-sample-header.xls`, as shown in the following screenshot. You can download this from [http://www.ericrochester.com/clj-data-analysis/data/small-sample-header.xls](http://www.ericrochester.com/clj-data-analysis/data/small-sample-header.xls).
How to do it...

Now, all you need to do is call incanter.excel/read-xls:

```
user=> (read-xls "data/small-sample-header.xls")
```

<table>
<thead>
<tr>
<th>given-name</th>
<th>surname</th>
<th>relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gomez</td>
<td>Addams</td>
<td>father</td>
</tr>
<tr>
<td>Morticia</td>
<td>Addams</td>
<td>mother</td>
</tr>
<tr>
<td>Pugsley</td>
<td>Addams</td>
<td>brother</td>
</tr>
</tbody>
</table>

How it works...

This can read standard Excel files (.xls) and the XML-based file format introduced in Excel 2003 (.xlsx).

Reading data from JDBC databases

Reading data from a relational database is only slightly more complicated than reading from Excel, and much of the extra complication involves connecting to the database.

Fortunately, there's a Clojure-contributed package that sits on top of JDBC (the Java database connector API, http://www.oracle.com/technetwork/java/javase/jdbc/index.html) and makes working with databases much easier. In this example, we'll load a table from an SQLite database (http://www.sqlite.org/), which stores the database in a single file.

Getting ready

First, list the dependencies in your Leiningen project.clj file. We will also need to include the database driver library. For this example, it is org.xerial/sqlite-jdbc:

```
(defproject getting-data "0.1.0-SNAPSHOT"
 :dependencies [[org.clojure/clojure "1.6.0"]
 [incanter "1.5.5"]
 [org.clojure/java.jdbc "0.3.3"]
 [org.xerial/sqlite-jdbc "3.7.15-M1"]])
```

Then, load the modules into your REPL or script file:

```
(require '[incanter.core :as i]
 '[clojure.java.jdbc :as j])
```
Finally, get the database connection information. I have my data in an SQLite database file named `data/small-sample.sqlite`, as shown in the following screenshot. You can download this from [http://www.ericrochester.com/clj-data-analysis/data/small-sample.sqlite](http://www.ericrochester.com/clj-data-analysis/data/small-sample.sqlite).

### How to do it...

Loading the data is not complicated, but we'll make it easier with a wrapper function:

1. We'll create a function that takes a database connection map and a table name and returns a dataset created from this table:

   ```clojure
   (defn load-table-data
     "This loads the data from a database table."
     [db table-name]
     (i/to-dataset
      (j/query db (str "SELECT * FROM " table-name ";"))))
   ```
2. Next, we define a database map with the connection parameters suitable for our database:

```clojure
(defdb {:subprotocol "sqlite"
        :subname "data/small-sample.sqlite"
        :classname "org.sqlite.JDBC"})
```

3. Finally, call `load-table-data` with `db` and a table name as a symbol or string:

```clojure
user=> (load-table-data db 'people)
```

<table>
<thead>
<tr>
<th>:relation</th>
<th>:surname</th>
<th>:given_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>father</td>
<td>Addams</td>
<td>Gomez</td>
</tr>
<tr>
<td>mother</td>
<td>Addams</td>
<td>Morticia</td>
</tr>
<tr>
<td>brother</td>
<td>Addams</td>
<td>Pugsley</td>
</tr>
</tbody>
</table>

How it works...

The `load-table-data` function passes the database connection information directly through to `clojure.java.jdbc/query`. It creates an SQL query that returns all of the fields in the table that is passed in. Each row of the result is a sequence of hashes mapping column names to data values. This sequence is wrapped in a dataset by `incanter.core/to-dataset`.

See also

Connecting to different database systems using JDBC isn't necessarily a difficult task, but it's dependent on which database you wish to connect to. Oracle has a tutorial for how to work with JDBC at http://docs.oracle.com/javase/tutorial/jdbc/basics, and the documentation for the `clojure.java.jdbc` library has some good information too (http://clojure.github.com/java.jdbc/). If you're trying to find out what the connection string looks like for a database system, there are lists available online. The list at http://www.java2s.com/Tutorial/Java/0340__Database/AListofJDBCDriver/Drivers/connectionstringdrivername.htm includes the major drivers.
Importing Data for Analysis

Reading XML data into Incanter datasets

One of the most popular formats for data is XML. Some people love it, while some hate it. However, almost everyone has to deal with it at some point. While Clojure can use Java's XML libraries, it also has its own package which provides a more natural way to work with XML in Clojure.

Getting ready

First, include these dependencies in your Leiningen project.clj file:

```clojure
(defproject getting-data "0.1.0-SNAPSHOT"
  :dependencies [[org.clojure/clojure "1.6.0"]
                 [incanter "1.5.5"]])
```

Use these libraries in your REPL or program:

```clojure
(require '[incanter.core :as i]
         [clojure.xml :as xml]
         [clojure.zip :as zip])
```

Then, find a data file. I visited the website for the Open Data Catalog for Washington, D.C. (http://data.octo.dc.gov/), and downloaded the data for the 2013 crime incidents. I moved this file to data/crime_incidents_2013_plain.xml. This is how the contents of the file look:

```xml
<?xml version="1.0" encoding="iso-8859-1"?>
<dcst:ReportedCrimes
  xmlns:dcst="http://dc.gov/dcstat/types/1.0/">
  <dcst:ReportedCrime
    xmlns:dcst="http://dc.gov/dcstat/types/1.0/">
    <dcst:dcn><![CDATA[04104147]]></dcst:dcn>
    <dcst:reportdatetime>2013-04-16T00:00:00-04:00</dcst:reportdatetime>
  </dcst:ReportedCrime>
</dcst:ReportedCrimes>
```

...
How to do it...

Now, let’s see how to load this file into an Incanter dataset:

1. The solution for this recipe is a little more complicated, so we'll wrap it into a function:

   ```clojure
   (defn load-xml-data [xml-file first-data next-data]
     (let [data-map (fn [node]
                       [(:tag node) (first (:content node))])
           data (-> (xml/parse xml-file)
                     zip/xml-zip
                     first-data
                     (iterate next-data)
                     (take-while #(not (nil? %))
                                 (map zip/children)
                                 (map #(mapcat data-map %))
                                 (map #(apply array-map %))
                                 i/to-dataset))]
     data)
   ```

2. We can call the function like this. Because there are so many columns, we’ll just verify the data that is loaded by looking at the column names and the row count:

   ```clojure
   user=> (def d (load-xml-data "data/crime_incidents_2013_plain.xml" zip/down zip/right))
   user=> (i/col-names d)
   user=> (i/nrow d)
   35826
   ```

   This looks good. This gives you the number of crimes reported in the dataset.

How it works...

This recipe follows a typical pipeline for working with XML:

1. Parsing an XML data file
2. Extracting the data nodes
3. Converting the data nodes into a sequence of maps representing the data
4. Converting the data into an Incanter dataset
load-xml-data implements this process. This takes three parameters:

- The input filename
- A function that takes the root node of the parsed XML and returns the first data node
- A function that takes a data node and returns the next data node or nil, if there are no more nodes

First, the function parses the XML file and wraps it in a zipper (we'll talk more about zippers in the next section). Then, it uses the two functions that are passed in to extract all of the data nodes as a sequence. For each data node, the function retrieves that node's child nodes and converts them into a series of tag name / content pairs. The pairs for each data node are converted into a map, and the sequence of maps is converted into an Incanter dataset.

There's more...

We used a couple of interesting data structures or constructs in this recipe. Both are common in functional programming or Lisp, but neither have made their way into more mainstream programming. We should spend a minute with them.

Navigating structures with zippers

The first thing that happens to the parsed XML is that it gets passed to clojure.zip/xml-zip. Zippers are standard data structures that encapsulate the data at a position in a tree structure, as well as the information necessary to navigate back out. This takes Clojure's native XML data structure and turns it into something that can be navigated quickly using commands such as clojure.zip/down and clojure.zip/right. Being a functional programming language, Clojure encourages you to use immutable data structures, and zippers provide an efficient, natural way to navigate and modify a tree-like structure, such as an XML document.

Zippers are very useful and interesting, and understanding them can help you understand and work better with immutable data structures. For more information on zippers, the Clojure-doc page is helpful (http://clojure-doc.org/articles/tutorials/parsing_xml_with_zippers.html). However, if you would rather dive into the deep end, see Gerard Huet's paper, The Zipper (http://www.st.cs.uni-saarland.de/edu/seminare/2005/advanced-fp/docs/huet-zipper.pdf).

Processing in a pipeline

We used the ->> macro to express our process as a pipeline. For deeply nested function calls, this macro lets you read it from the left-hand side to the right-hand side, and this makes the process's data flow and series of transformations much more clear.
We can do this in Clojure because of its macro system. -&gt; simply rewrites the calls into Clojure's native, nested format as the form is read. The first parameter of the macro is inserted into the next expression as the last parameter. This structure is inserted into the third expression as the last parameter, and so on, until the end of the form. Let's trace this through a few steps. Say, we start off with the expression: 

```
(->> x first (map length) (apply +))
```

As Clojure builds the final expression, here's each intermediate step (the elements to be combined are highlighted at each stage):

1. `(->> x first (map length) (apply +))`
2. `(->> (first x) (map length) (apply +))`
3. `(->> (map length (first x)) (apply +))`
4. `(apply + (map length (first x)))`

**Comparing XML and JSON**

XML and JSON (from the Reading JSON data into Incanter datasets recipe) are very similar. Arguably, much of the popularity of JSON is driven by disillusionment with XML's verboseness. When we're dealing with these formats in Clojure, the biggest difference is that JSON is converted directly to native Clojure data structures that mirror the data, such as maps and vectors. Meanwhile, XML is read into record types that reflect the structure of XML, not the structure of the data.

In other words, the keys of the maps for JSON will come from the domains, first name or age, for instance. However, the keys of the maps for XML will come from the data format, such as tag, attribute, or children, and the tag and attribute names will come from the domain. This extra level of abstraction makes XML more unwieldy.

**Scraping data from tables in web pages**

There's data everywhere on the Internet. Unfortunately, a lot of it is difficult to reach. It's buried in tables, articles, or deeply nested div tags. Web scraping (writing a program that walks over a web page and extracts data from it) is brittle and laborious, but it's often the only way to free this data so it can be used in our analyses. This recipe describes how to load a web page and dig down into its contents so that you can pull the data out.

To do this, we're going to use the Enlive (https://github.com/cgrand/enlive/wiki) library. This uses a domain specific language (DSL, a set of commands that make a small set of tasks very easy and natural) based on CSS selectors to locate elements within a web page. This library can also be used for templating. In this case, we'll just use it to get data back out of a web page.
Getting ready

First, you have to add Enlive to the dependencies in the project.clj file:

```clojure
(defproject getting-data "0.1.0-SNAPSHOT"
  :dependencies [[org.clojure/clojure "1.6.0"]
                 [incanter "1.5.5"]
                 [enlive "1.1.5"]])
```

Next, use these packages in your REPL or script:

```clojure
(require '[clojure.string :as string]
         [net.cgrand.enlive-html :as html]
         [incanter.core :as i])
(import [java.net URL])
```

Finally, identify the file to scrape the data from. I've put up a file at http://www.ericrochester.com/clj-data-analysis/data/small-sample-table.html, which looks like this:

![Table Example](https://www.ericrochester.com/clj-data-analysis/data/small-sample-table.html)

**A Header**

This contains a bunch of people.

Tonight we're going to code HTML like it's 1999!

<table>
<thead>
<tr>
<th>Given Name</th>
<th>Surname</th>
<th>Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gomez</td>
<td>Addams</td>
<td>father</td>
</tr>
<tr>
<td>Moricia</td>
<td>Addams</td>
<td>mother</td>
</tr>
<tr>
<td>Pugsley</td>
<td>Addams</td>
<td>brother</td>
</tr>
<tr>
<td>Wednesday</td>
<td>Addams</td>
<td>sister</td>
</tr>
<tr>
<td>Pubert</td>
<td>Addams</td>
<td>brother</td>
</tr>
<tr>
<td>Fester</td>
<td>Addams</td>
<td>uncle</td>
</tr>
<tr>
<td>Grandmama</td>
<td></td>
<td>grandmother</td>
</tr>
<tr>
<td>Thing</td>
<td></td>
<td>hand</td>
</tr>
<tr>
<td>Lunch</td>
<td></td>
<td>butler</td>
</tr>
<tr>
<td>Itt</td>
<td></td>
<td>cousin</td>
</tr>
<tr>
<td>Cackle</td>
<td></td>
<td>cousin</td>
</tr>
</tbody>
</table>


It's intentionally stripped down, and it makes use of tables for layout (hence the comment about 1999).
How to do it...

1. Since this task is a little complicated, let's pull out the steps into several functions:

```clojure
(defn to-keyword
  "This takes a string and returns a normalized keyword."
  [input]
  (->input
   string/lower-case
   (string/replace \space \-)
   keyword))

(defn load-data
  "This loads the data from a table at a URL."
  [url]
  (let [page (html/html-resource (URL. url))
    table (html/select page [:table#data])
    headers (->>
      (html/select table [:tr :th])
      (map html/text)
      (map to-keyword)
      vec)
    rows (->> (html/select table [:tr])
      (map #(html/select % [:td]))
      (map #(map html/text %))
      (filterseq))
    (i/dataset headers rows)))
```

2. Now, call `load-data` with the URL you want to load data from:

```clojure
user=> (load-data (str "http://www.ericrochester.com/
  "clj-data-analysis/data/small-sample-table.html"))
| :given-name | :surname | :relation |
|------------------------|
| Gomez | Addams | father |
| Morticia | Addams | mother |
| Pugsley | Addams | brother |
| Wednesday | Addams | sister |
```
How it works...

The let bindings in load-data tell the story here. Let's talk about them one by one.

The first binding has Enlive download the resource and parse it into Enlive's internal representation:

```enlive
(let [page (html/html-resource (URL. url))]
```

The next binding selects the table with the data ID:

```enlive
table (html/select page [:table#data])
```

Now, select all the header cells from the table, extract the text from them, convert each to a keyword, and then convert the entire sequence into a vector. This gives headers for the dataset:

```enlive
headers (->>
  (html/select table [:tr :th])
  (map html/text)
  (map to-keyword)
  vec)
```

First, select each row individually. The next two steps are wrapped in map so that the cells in each row stay grouped together. In these steps, select the data cells in each row and extract the text from each. Last, use filterseq, which removes any rows with no data, such as the header row:

```enlive
rows (->> (html/select table [:tr])
  (map #(html/select % [:td]))
  (map #(map html/text %))
  (filterseq))]
```

Here's another view of this data. In this image, you can see some of the code from this web page. The variable names and select expressions are placed beside the HTML structures that they match. Hopefully, this makes it more clear how the select expressions correspond to the HTML elements:

```
<table id="data" border="0">
  <tr><th>Given Name</th> <th>Surname</th> <th>Relation</th></tr>
  <tr><td> Gomez </td> <td> Addams </td> <td> father </td></tr>
  <tr><td> Morticia </td> <td> Addams </td> <td> mother </td></tr>
  <tr><td> Pugsley </td> <td> Addams </td> <td> brother </td></tr>
  <tr><td> Wednesday </td> <td> Addams </td> <td> sister </td></tr>
  <tr><td> Pubert </td> <td> Addams </td> <td> brother </td></tr>
```
Finally, convert everything to a dataset. `incanter.core/dataset` is a lower level constructor than `incanter.core/to-dataset`. It requires you to pass in the column names and data matrix as separate sequences:

```clojure
(i/dataset headers rows))
```

It's important to realize that the code, as presented here, is the result of a lot of trial and error. Screen scraping usually is. Generally, I download the page and save it, so I don't have to keep requesting it from the web server. Next, I start the REPL and parse the web page there. Then, I can take a look at the web page and HTML with the browser's view source function, and I can examine the data from the web page interactively in the REPL. While working, I copy and paste the code back and forth between the REPL and my text editor, as it's convenient. This workflow and environment (sometimes called REPL-driven-development) makes screen scraping (a fiddly, difficult task at the best of times) almost enjoyable.

See also

- The next recipe, Scraping textual data from web pages, has a more involved example of data scraping on an HTML page
- The Aggregating data from different formats recipe has a practical, real-life example of data scraping in a table

Scraping textual data from web pages

Not all of the data on the Web is in tables, as in our last recipe. In general, the process to access this nontabular data might be more complicated, depending on how the page is structured.

Getting ready

First, we'll use the same dependencies and the `require` statements as we did in the last recipe, Scraping data from tables in web pages.

Next, we'll identify the file to scrape the data from. I've put up a file at http://www.ericrochester.com/clj-data-analysis/data/small-sample-list.html.

This is a much more modern example of a web page. Instead of using tables, it marks up the text with the `section` and `article` tags and other features from HTML5, which help convey what the text means, not just how it should look.
Importing Data for Analysis

As the screenshot shows, this page contains a list of sections, and each section contains a list of characters:

```
Small Sample List

- Section 1
- Section 2
- Section 3

Addam's Family

Here's some information about the Addam's Family.

- Gomez Addams — father
- Morticia Addams — mother
- Pugsley Addams — brother
- Wednesday Addams — sister
- Pubert Addams — brother
- Fester Addams — uncle
- Grandmama — grandmother
- Thing — hand
- Lurch — butler
- Cousin Itt — cousin
- Cousin Cackle — cousin
```

**How to do it...**

1. Since this is more complicated, we'll break the task down into a set of smaller functions:
   ```clojure
   (defn get-family
     "This takes an article element and returns the family name."
     [article]
     (string/join
      (map html/text (html/select article [:header :h2]))))
   ```
(defn get-person
 "This takes a list item and returns a map of the person's name and relationship."
 [li]
 (let [{\{pnames :content\} rel} (:content li)]
  \{:name (apply str pnames)
  \:relationship (string/trim rel)}))

(defn get-rows
 "This takes an article and returns the person mappings, with the family name added."
 [article]
 (let [family (get-family article)]
  (map #(assoc % :family family)
    (map get-person
     (html/select article [:ul :li])))
  )

(defn load-data
 "This downloads the HTML page and pulls the data out of it."
 [html-url]
 (let [html (html/html-resource (URL. html-url))
      articles (html/select html [:article])
    ]
    (i/to-dataset (mapcat get-rows articles)))

2. Now that these functions are defined, we just call load-data with the URL that we want to scrape:

user=> (load-data (str "http://www.ericrochester.com/
    "clj-data-analysis/data/
    "small-sample-list.html"))

<table>
<thead>
<tr>
<th>:family</th>
<th>:name</th>
<th>:relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addam's Family</td>
<td>Gomez Addams</td>
<td>— father</td>
</tr>
<tr>
<td>Addam's Family</td>
<td>Morticia Addams</td>
<td>— mother</td>
</tr>
<tr>
<td>Addam's Family</td>
<td>Pugsley Addams</td>
<td>— brother</td>
</tr>
</tbody>
</table>
| ...
How it works...

After examining the web page, each family is wrapped in an article tag that contains a header with an h2 tag. get-family pulls that tag out and returns its text.

get-person processes each person. The people in each family are in an unordered list (ul), and each person is in an li tag. The person's name itself is in an em tag. let gets the contents of the li tag and decomposes it in order to pull out the name and relationship strings. get-person puts both pieces of information into a map and returns it.

get-rows processes each article tag. It calls get-family to get that information from the header, gets the list item for each person, calls get-person on the list item, and adds the family to each person's mapping.

Here’s how the HTML structures correspond to the functions that process them. Each function name is mentioned beside the elements it parses:

```
<article>
  <header>
    <h2 id='addams'>Addam's Family</h2>
  </header>
  <p>Here's some information about the Addam's Family.</p>
  <ul>
    <li><em>Gomez Addams</em> &mdash; father</li>
    <li><em>Morticia Addams</em> &mdash; mother</li>
    <li><em>Pugsley Addams</em> &mdash; brother</li>
    <li><em>Wednesday Addams</em> &mdash; sister</li>
    <li><em>Pubert Addams</em> &mdash; brother</li>
    <li><em>Fester Addams</em> &mdash; uncle</li>
    <li><em>Grandmama</em> &mdash; grandmother</li>
    <li><em>Thing</em> &mdash; hand</li>
    <li><em>Lurch</em> &mdash; butler</li>
    <li><em>Cousin Itt</em> &mdash; cousin</li>
    <li><em>Cousin Cackle</em> &mdash; cousin</li>
  </ul>
</article>
```

Finally, load-data ties the process together by downloading and parsing the HTML file and pulling the article tags from it. It then calls get-rows to create the data mappings and converts the output to a dataset.
More and more data is going up on the Internet using linked data in a variety of formats such as microformats, RDFa, and RDF/XML.

Linked data represents entities as consistent URLs and includes links to other databases of the linked data. In a sense, it's the computer equivalent of human-readable web pages. Often, these formats are used for open data, such as the data published by some governments, like in the UK and elsewhere.

Linked data adds a lot of flexibility and power, but it also introduces more complexity. Often, to work effectively with linked data, we need to start a triple store of some kind. In this recipe and the next three, we'll use Sesame (http://rdf4j.org/) and the kr Clojure library (https://github.com/drlivingston/kr).

Getting ready

First, we need to make sure that the dependencies are listed in our Leiningen project.clj file:

```
(defproject getting-data "0.1.0-SNAPSHOT"
  :dependencies [[org.clojure/clojure "1.6.0"]
                 [incanter "1.5.5"]
                 [edu.ucdenver.ccp/kr-sesame-core "1.4.17"]
                 [org.clojure/tools.logging "0.3.0"]
                 [org.slf4j/slf4j-simple "1.7.7"]])
```

We'll execute these packages to have these loaded into our script or REPL:

```
(use 'incanter.core
 'edu.ucdenver.ccp.kr.kb
 'edu.ucdenver.ccp.kr.rdf
 'edu.ucdenver.ccp.kr.sparql
 'edu.ucdenver.ccp.kr.sesame.kb
 'clojure.set)
(import [java.io File])
```

For this example, we'll get data from the Telegraphis Linked Data assets. We'll pull down the database of currencies at http://telegraphis.net/data/currencies/currencies.ttl. Just to be safe, I've downloaded that file and saved it as data/currencies.ttl, and we'll access it from there.

We'll store the data, at least temporarily, in a Sesame data store (http://notes.3kbo.com/sesame) that allows us to easily store and query linked data.
How to do it...

The longest part of this process will be to define the data. The libraries we're using do all of the heavy lifting, as shown in the steps given below:

1. First, we will create the triple store and register the namespaces that the data uses. We'll bind this triple store to the name tstore:

   ```
   (defn kb-memstore
     "This creates a Sesame triple store in memory."
     []
     (kb :sesame-mem))
   (defn init-kb [kb-store]
     (register-namespaces
      kb-store
      '("geographis" "http://telegraphis.net/ontology/geography/geography#"
       "code" "http://telegraphis.net/ontology/measurement/code#"
       "money" "http://telegraphis.net/ontology/money/money#"
       "owl" "http://www.w3.org/2002/07/owl#"
       "rdf" "http://www.w3.org/1999/02/22-rdf-syntax-ns#"
       "xsd" "http://www.w3.org/2001/XMLSchema#"
       "currency" "http://telegraphis.net/data/currencies/"
       "dbpedia" "http://dbpedia.org/resource/"
       "dbpedia-ont" "http://dbpedia.org/ontology/"
       "dbpedia-prop" "http://dbpedia.org/property/"
       "err" "http://ericrochester.com/")))

   (def t-store (init-kb (kb-memstore)))
   ```
2. After taking a look at the data some more, we can identify what data we want to pull out and start to formulate a query. We’ll use the kr library’s (https://github.com/drlivingston/kr) query DSL and bind it to the name q:

   (def q '((?/c rdf/type money/Currency)
                 (?/c money/name ?/full_name)
                 (?/c money/shortName ?/name)
                 (?/c money/symbol ?/symbol)
                 (?/c money/minorName ?/minor_name)
                 (?/c money/minorExponent ?/minor_exp)
                 (?/c money/isoAlpha ?/iso)
                 (?/c money/currencyOf ?/country))

3. Now, we need a function that takes a result map and converts the variable names in the query into column names in the output dataset. The header-keyword and fix-headers functions will do this:

   (defn header-keyword
     "This converts a query symbol to a keyword."
     [header-symbol]
     (keyword (.replace (name header-symbol) \\ \-))
   )

   (defn fix-headers
     "This changes all of the keys in the map to make them valid header keywords."
     [coll]
     (into {}
       (map (fn [[k v]] [(header-keyword k) v])
            coll)))

4. As usual, once all of the pieces are in place, the function that ties everything together is short:

   (defn load-data
     [krdf-file q]
     (load-rdf-file k rdf-file)
     (to-dataset (map fix-headers (query k q))))
5. Also, using this function is just as simple:

```
user=> (sel d :rows (range 3)
    :cols [[:full-name :name :iso :symbol]])
```

|                  :full-name |   :name | :iso | :symbol |
|-----------------------------+---------+------|---------|
| United Arab Emirates dirham | dirham  | AED  | $2 |
| Afghan afghani afghani     | afghani | AFN  | $ |
| Albanian lek                | lek     | ALL  | L |

How it works...

First, here's some background information. **Resource Description Format (RDF)** isn't an XML format, although it's often written using XML. (There are other formats as well, such as N3 and Turtle.) RDF sees the world as a set of statements. Each statement has at least three parts (a triple): a subject, predicate, and object. The subject and predicate must be URIs. (URIs are like URLs, only more general. For example, `uri:7890` is a valid URI.) Objects can be a literal or a URI. The URIs form a graph. They are linked to each other and make statements about each other. This is where the *linked in linked data* comes from.

If you want more information about linked data, [http://linkeddata.org/guides-and-tutorials](http://linkeddata.org/guides-and-tutorials) has some good recommendations.

Now, about our recipe. From a high level, the process we used here is pretty simple, given as follows:

1. Create a triple store (`kb-memstore` and `init-kb`)
2. Load the data (`load-data`)
3. Query the data to pull out only what you want (`q` and `load-data`)
4. Transform it into a format that Incanter can ingest easily (`rekey` and `col-map`)
5. Finally, create the Incanter dataset (`load-data`)

The newest thing here is the query format. `kb` uses a nice SPARQL-like DSL to express the queries. In fact, it's so easy to use that we'll deal with it instead of working with raw RDF. The items starting with `?/` are variables which will be used as keys for the result maps. The other items look like `rdf-namespace/value`. The namespace is taken from the registered namespaces defined in `init-kb`. These are different from Clojure's namespaces, although they serve a similar function for your data: to partition and provide context.

See also

The next few recipes, *Querying RDF data with SPARQL* and *Aggregating data from different formats*, build on this recipe and will use much of the same set up and techniques.
Querying RDF data with SPARQL

For the last recipe, Reading RDF data, the embedded domain-specific language (EDSL) used for the query gets converted to SPARQL, which is the query language for many linked data systems. If you squint just right at the query, it looks kind of like a SPARQL WHERE clause. For example, you can query DBPedia to get information about a city, such as its population, location, and other data. It's a simple query, but a query nevertheless.

This worked great when we had access to the raw data in our own triple store. However, if we need to access a remote SPARQL endpoint directly, it's more complicated.

For this recipe, we'll query DBPedia (http://dbpedia.org) for information on the United Arab Emirates currency, which is the Dirham. DBPedia extracts structured information from Wikipedia (the summary boxes) and republishes it as RDF. Just as Wikipedia is a useful first-stop for humans to get information about something, DBPedia is a good starting point for computer programs that want to gather data about a domain.

Getting ready

First, we need to make sure that the dependencies are listed in our Leiningen project.clj file:

```
(defproject getting-data "0.1.0-SNAPSHOT"
  :dependencies [[org.clojure/clojure "1.6.0"]
                 [incanter "1.5.5"]
                 [edu.ucdenver.ccp/kr-sesame-core "1.4.17"]
                 [org.clojure/tools.logging "0.3.0"]
                 [org.slf4j/slf4j-simple "1.7.7"]])
```

Then, load the Clojure and Java libraries we'll use:

```
(require '[clojure.java.io :as io]
         '[clojure.xml :as xml]
         '[clojure.pprint :as pp]
         '[clojure.zip :as zip]
         'incanter.core
         'edu.ucdenver.ccp.kr.kb
         'edu.ucdenver.ccp.kr.rdf
         'edu.ucdenver.ccp.kr.sparql
         'edu.ucdenver.ccp.kr.sesame.kb
         'clojure.set)
(import [java.io File]
        [java.net URL URLDecoder])
```
How to do it...

As we work through this, we'll define a series of functions. Finally, we'll create one function, `load-data`, to orchestrate everything, and we'll finish by doing the following:

1. We have to create a Sesame triple store and initialize it with the namespaces we'll use. For both of these, we'll use the `kb-memstore` and `init-kb` functions from *Reading RDF data*. We define a function that takes a URI for a subject in the triple store and constructs a SPARQL query that returns at most 200 statements about this subject. The function then filters out any statements with non-English strings for objects, but it allows everything else:

   ```clojure
   (defn make-query
     "This creates a query that returns all of the triples related to a subject URI. It filters out non-English strings."
     ([subject kb]
      (binding [\*kb* kb
                 \*select-limit* 200]
        (sparql-select-query
         (list '(~subject ?/p ?/o)
               '(:or (:not (:isLiteral ?/o))
                  (= (:lang ?/o) "en"))))))
   )
   
2. Now that we have the query, we'll need to encode it into a URL in order to retrieve the results:

   ```clojure
   (defn make-query-uri
     "This constructs a URI for the query."
     ([base-uri query]
      (URL. (str base-uri
             "?format=
             (URLEncoder/encode "text/xml")
             
             "&query=" (URLEncoder/encode query))))
   )
   
3. Once we get a result, we'll parse the XML file, wrap it in a zipper, and navigate to the first result. All of this will be in a function that we'll write in a minute. Right now, the next function will take this first result node and return a list of all the results:

   ```clojure
   (defn result-seq
     "This takes the first result and returns a sequence of this node, plus all of the nodes to the right of it."
     ([first-result]
      (cons (zip/node first-result)
            (zip/rights first-result))))
4. The following set of functions takes each result node and returns a key-value pair (result-to-kv). It uses binding-str to pull the results out of the XML. Then, accum-hash pushes the key-value pairs into a map. Keys that occur more than once have their values accumulated in a vector:

```clojure
(defn binding-str
  "This takes a binding, pulls out the first tag's content, and concatenates it into a string."
  ([b]
   (apply str (:content (first (:content b))))))

(defn result-to-kv
  "This takes a result node and creates a key-value vector pair from it."
  ([r]
   (let [[p o] (:content r)]
     [(binding-str p) (binding-str o)])))

(defn accum-hash
  ([m [k v]]
   (if-let [current (m k)]
     (assoc m k (str current \ space v))
     (assoc m k v))))
```

5. For the last utility function, we'll define rekey. This will convert the keys of a map based on another map:

```clojure
(defn rekey
  "This just flips the arguments for clojure.set/rename-keys to make it more convenient."
  ([k-map map]
   (rename-keys
     (select-keys map (keys k-map)) k-map)))
```

6. Let's now add a function that takes a SPARQL endpoint and subject and returns a sequence of result nodes. This will use several of the functions we've just defined:

```clojure
(defn query-sparql-results
  "This queries a SPARQL endpoint and returns a sequence of result nodes."
  ([sparql-uri subject kb]
   (->> kb
    ;; Build the URI query string.
    (make-query subject)
    (make-query-uri sparql-uri)
    ;; Perform the query.
    ;; Return the results.
    (map #(binding-str %))
    ;; Accumulate results.
    (reduce accum-hash []))
```

---

Chapter 1
7. Finally, we can pull everything together. Here's `load-data`:

```clojure
(defn load-data
  "This loads the data about a currency for the given URI."
  [sparql-uri subject col-map]
  (->>
    ;; Initialize the triple store.
    (kb-memstore) init-kb
    ;; Get the results.
    (query-sparql-results sparql-uri subject)
    ;; Generate a mapping.
    (map result-to-kv)
    (reduce accum-hash {})
    ;; Translate the keys in the map.
    (rekey col-map)
    ;; And create a dataset.
    to-dataset))
```

8. Now, let's use this data. We can define a set of variables to make it easier to reference the namespaces we'll use. We'll use these to create the mapping to column names:

```clojure
(def rdfs "http://www.w3.org/2000/01/rdf-schema#")
(def dbpedia "http://dbpedia.org/resource/")
(def dbpedia-ont "http://dbpedia.org/ontology/")
(def dbpedia-prop "http://dbpedia.org/property/")

(def col-map {
  (str rdfs 'label) :name,
  (str dbpedia-prop 'usingCountries) :country
  (str dbpedia-prop 'peggedWith) :pegged-with
  (str dbpedia-prop 'symbol) :symbol
  (str dbpedia-prop 'usedBanknotes) :used-banknotes
  (str dbpedia-prop 'usedCoins) :used-coins
  (str dbpedia-prop 'inflationRate) :inflation})
```
9. We call load-data with the DBPedia SPARQL endpoint, the resource we want information about (as a symbol), and the column map:

```clojure
user=> (def d (load-data "http://dbpedia.org/sparql"
(symbol (str dbpedia dbpedia "United_Arab_Emirates_dirham")))
col-map))
user=> (sel d :cols [:country :name :symbol])

|             :country |                       :name | :symbol |
|----------------------+-----------------------------+---------|
| United Arab Emirates | United Arab Emirates dirham | إ.د |
```

### How it works...

The only part of this recipe that has to do with SPARQL, really, is the make-query function. It uses the sparql-select-query function to generate a SPARQL query string from the query pattern. This pattern has to be interpreted in the context of the triple store that has the namespaces defined. This context is set using the binding command. We can see how this function works by calling it from the REPL by itself:

```clojure
user=> (println
(make-query
(symbol (str dbpedia "/United_Arab_Emirates_dirham")))
(init-kb (kb-memstore)))
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT ?p ?o
WHERE {
  FILTER ( ( ! isLiteral(?o)
    || ( datatype(?o) !=<http://www.w3.org/1999/02/22-rdf-syntax-ns#langString> )
    || ( lang(?o) = "en" ) ) )
}
LIMIT 200
```

The rest of the recipe is concerned with parsing the XML format of the results, and in many ways, it's similar to the last recipe.

### There's more...

For more information on RDF and linked data, see the previous recipe, Reading RDF data.
Aggregating data from different formats

Being able to aggregate data from many linked data sources is good, but most data isn’t already formatted for the semantic Web. Fortunately, linked data’s flexible and dynamic data model facilitates the integration of data from multiple sources.

For this recipe, we’ll combine several previous recipes. We’ll load currency data from RDF, as we did in the Reading RDF data recipe. We’ll also scrape the exchange rate data from X-Rates (http://www.x-rates.com) to get information out of a table, just as we did in the Scraping data from tables in web pages recipe. Finally, we’ll dump everything into a triple store and pull it back out, as we did in the last recipe.

Getting ready

First, make sure your Leiningen project.clj file has the right dependencies:

```clojure
(defproject getting-data "0.1.0-SNAPSHOT"
  :dependencies [[org.clojure/clojure "1.6.0"]
                 [incanter "1.5.5"]
                 [enlive "1.1.5"]
                 [edu.ucdenver.ccp/kr-sesame-core "1.4.17"]
                 [org.clojure/tools.logging "0.3.0"]
                 [org.slf4j/slf4j-simple "1.7.7"]
                 [clj-time "0.7.0"]])
```

We need to declare that we’ll use these libraries in our script or REPL:

```clojure
(require '(clojure.java [io :as io]
       [net.cgrand [enlive-html :as html]]
       (use 'incanter.core
            'clj-time.coerce
            '(clj-time.format :only [formatter formatters parse unparse])
            'edu.ucdenver.ccp.kr.kb
            'edu.ucdenver.ccp.kr.rdf
            'edu.ucdenver.ccp.kr.sparql
            'edu.ucdenver.ccp.kr.sesame.kb)

(import [java.io File]
        [java.net URL URLEncoder])
```

Finally, make sure that you have the file, data/currencies.ttl, which we’ve been using since Reading RDF data.
How to do it...

Since this is a longer recipe, we'll build it up in segments. At the end, we'll tie everything together.

Creating the triple store

To begin with, we'll create the triple store. This has become pretty standard. In fact, we'll use the same version of kb-memstore and init-kb that we've been using from the Reading RDF data recipe.

Scraping exchange rates

The first data that we'll pull into the triple store is the current exchange rates:

1. This is where things get interesting. We'll pull out the timestamp. The first function finds it, and the second function normalizes it into a standard format:
   
   ```
   (defn find-time-stamp
     ([module-content]
      (second
       (map html/text
         (html/select module-content
                       [:span.ratesTimestamp])))))
   
   (def time-stamp-format
     (formatter "MMM dd, yyyy HH:mm 'UTC'"))
   
   (defn normalize-date
     ([date-time]
      (unparse (formatters :date-time)
               (parse time-stamp-format date-time))))
   ```

2. We'll drill down to get the countries and their exchange rates:
   
   ```
   (defn find-data
     ([module-content]
      (html/select module-content
                    [:table.tablesorter.ratesTable :tbody :tr])))
   
   (defn td->code
     ([td]
      (let [code (-> td
                     html/text
                     (map html/select
                           module-content
                           [:span.ratesTsTimestamp]))]
       code)))
   ```
(html/select [:a])
first
:attrs :href
(string/split #"=")
last]
(symbol "currency" (str code "#" code)))

(defn get-td-a
  ([td]
   (->> td
    :content
    (mapcat :content)
    string/join
    read-string)))

(defn get-data
  ([row]
   (let [[[td-header td-to td-from]
           (filter map? (:content row))]
         {:currency (td->code td-to)
          :exchange-to (get-td-a td-to)
          :exchange-from (get-td-a td-from)})

3. This function takes the data extracted from the HTML page and generates a list of RDF triples:

(defn data->statements
  ([{time-stamp data}
    (let [{:keys [currency exchange-to]} data]
      (list [currency 'err/exchangeRate exchange-to]
            [currency 'err/exchangeWith 'currency/USD#USD]
            [currency 'err/exchangeRateDate [time-stamp 'xsd/dateTime]]))))

4. This function ties all of the processes that we just defined together by pulling the data out of the web page, converting it to triples, and adding them to the database:

(defn load-exchange-data
  "This downloads the HTML page and pulls the data out of it."
  [kb html-url]
That's a mouthful, but now that we can get all of the data into a triple store, we just need to pull everything back out and into Incanter.

**Loading currency data and tying it all together**

Bringing the two data sources together and exporting it to Incanter is fairly easy at this point:

```clojure
(defn aggregate-data
  "This controls the process and returns the aggregated data."
  [kb data-file data-url q col-map]
  (load-rdf-file kb (File. data-file))
  (load-exchange-data kb (URL. data-url))
  (to-dataset (map (partial rekey col-map) (query kb q))))

We'll need to do a lot of the set up we've done before. Here, we'll bind the triple store, the query, and the column map to names so that we can refer to them easily:

```clojure
(def t-store (init-kb (kb-memstore)))

(def q
  '((?/c rdf/type money/Currency)
    (?/c money/name ?/name)
    (?/c money/shortName ?/shortName)
    (?/c money/isoAlpha ?/iso)
    (?/c money/minorName ?/minorName)
    (?/c money/minorExponent ?/minorExponent)
      :optional
        ((?/c err/exchangeRate ?/exchangeRate)
        (?/c err/exchangeWith ?/exchangeWith)
        (?/c err/exchangeRateDate ?/exchangeRateDate))))

(def col-map {'/name :fullname
          '/iso :iso
          '/shortName :name
          '/minorName :minor-name}
Importing Data for Analysis

```
'?/minorExponent :minor-exp
'?/exchangeRate :exchange-rate
'?/exchangeWith :exchange-with
'?/exchangeRateDate :exchange-date}
```

The specific URL that we're going to scrape is http://www.x-rates.com/table/?from=USD&amount=1.00. Let's go ahead and put everything together:

```clojure
user=> (def d
  (aggregate-data t-store "data/currencies.ttl"
    "http://www.x-rates.com/table/?from=USD&amount=1.00"
    q col-map))
```

As you will see, some of the data from currencies.ttl doesn't have exchange data (the ones that start with nil). We can look in other sources for that, or decide that some of those currencies don't matter for our project.

**How it works...**

A lot of this is just a slightly more complicated version of what we've seen before, pulled together into one recipe. The complicated part is scraping the web page, which is driven by the structure of the page itself.

After taking a look at the source for the page and playing with it on the REPL, the page's structure was clear. First, we needed to pull the timestamp off the top of the table that lists the exchange rates. Then, we walked over the table and pulled the data from each row. Both the data tables (the short and long ones) are in a div element with a moduleContent class, so everything begins there.

Next, we drilled down from the module's content into the rows of the rates table. Inside each row, we pulled out the currency code and returned it as a symbol in the currency namespace. We also drilled down to the exchange rates and returned them as floats. Then, we put everything into a map and converted it to triple vectors, which we added to the triple store.
See also

- For more information on how we pulled in the main currency data and worked with the triple store, see the Reading RDF data recipe.
- For more information on how we scraped the data from the web page, see Scraping data from tables in web pages.
- For more information on the SPARQL query, see Reading RDF data with SPARQL.
Where to buy this book

You can buy Clojure Data Analysis Cookbook Second Edition from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.