Quick answers to common problems

Clojure Data Analysis Cookbook

Over 110 recipes to help you dive into the world of practical data analysis using Clojure

Eric Rochester

[PACKT] open source* PUBLISHING
community experience distilled

Clojure Data Analysis Cookbook

Over 110 recipes to help you dive into the world of practical data analysis using Clojure

Eric Rochester

[PACKT] PUBLISHING
open source
community experience distilled

BIRMINGHAM - MUMBAI

Clojure Data Analysis Cookbook

First published: March 2013

Production Reference: 1130313

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-264-3

www.packtpub.com

Cover Image by J.Blaminsky (milak6@wp.pl)

Credits

Author

Eric Rochester

Reviewers

Jan Borgelin

Thomas A. Faulhaber, Jr.

Charles M. Norton

Miki Tebeka

Acquisition Editor

Erol Staveley

Lead Technical Editor

Dayan Hyames

Technical Editors

Nitee Shetty

Dennis John

Project Coordinator

Anugya Khurana

Proofreaders

Mario Cecere

Sandra Hopper

Indexer

Monica Ajmera Mehta

Graphics

Aditi Gajjar

Production Coordinator

Nilesh R. Mohite

Cover Work

Nilesh R. Mohite

About the Author

Eric Rochester enjoys reading, writing, and spending time with his wife and kids. When he's not doing those 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's exploring functional programming languages, including Clojure and Haskell. He works at the Scholars' Lab in the library at the University of Virginia, helping humanities professors and graduate students realize their digitally informed research agendas.

I'd like to thank everyone. My technical reviewers—Jan Borgelin, Tom Faulhaber, Charles Norton, and Miki Tebeka—proved invaluable. Also, thank you to the editorial staff at Packt Publishing. This book is much stronger for all of their feedbacks, and any remaining deficiencies are mine alone.

Thank you to Bethany Nowviskie and Wayne Graham. They've made the Scholars' Lab a great place to work, with interesting projects, as well as space to explore our own interests.

And especially I would like to thank Jackie and Melina. They've been exceptionally patient and supportive while I worked on this project. Without them, it wouldn't be worth it.

About the Reviewers

Jan Borgelin is a technology geek with over 10 years of professional software development experience. Having worked in diverse positions in the field of enterprise software, he currently works as a CEO and Senior Consultant for BA Group Ltd., an IT consultancy based in Finland. For the past 2 years, he has been more actively involved in functional programming and as part of that has become interested in Clojure among other things.

I would like to thank my family and our employees for tolerating my excitement about the book throughout the review process.

Thomas A. Faulhaber, Jr., is principal of Infolace (`www.infolace.com`), a San Francisco-based consultancy. Infolace helps clients from startups to global brands turn raw data into information and information into action. Throughout his career, he has developed systems for high-performance TCP/IP, large-scale scientific visualization, energy trading, and many more.

He has been a contributor to, and user of, Clojure and Incanter since their earliest days. The power of Clojure and its ecosystem (of both code and people) is an important "magic bullet" in Tom's practice.

Charles Norton has over 25 years of programming experience, ranging from factory automation applications and firmware to network middleware, and is currently a programmer and application specialist for a Greater Boston municipality. He maintains and develops a collection of software applications that support finances, health insurance, and water utility administration. These systems are implemented in several languages, including Clojure.

Miki Tebeka has been shipping software for more than 10 years. He has developed a wide variety of products from assemblers and linkers to news trading systems to cloud infrastructures. He currently works at Adconion where he shuffles through more than 6 billion monthly events. In his free time, he is active in several open source communities.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit `www.PacktPub.com` for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at `www.PacktPub.com` and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `service@packtpub.com` for more details.

At `www.PacktPub.com`, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

`http://PacktLib.PacktPub.com`

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Data's everywhere! And, as it has become more pervasive, our desire to use it has grown just as quickly. A lot hides in data: potential sales, users' browsing patterns, demographic information, and many, many more things. There are insights we could gain and decisions we could make better, if only we could find out what's in our data.

This book will help with that.

The programming language Clojure will help us. Clojure was first released in 2007 by Rich Hickey. It's a member of the lisp family of languages, and it has the strengths and flexibility that they provide. It's also functional, so Clojure programs are easy to reason with. And, it has amazing features for working concurrently and in parallel. All of these can help us as we analyze data while keeping things simple and fast.

Clojure's usefulness for data analysis is further improved by a number of strong libraries. Incanter provides a practical environment for working with data and performing statistical analysis. Cascalog is an easy-to-use wrapper over Hadoop and Cascading. Finally, when we're ready to publish our results, ClojureScript, an implementation of Clojure that generates JavaScript, can help us to visualize our 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*, will cover 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*, will present strategies and implementations for normalizing dates, fixing spelling, and working 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*, will cover Clojure's concurrency features and how we can use them to simplify our programs.

Chapter 4, *Improving Performance with Parallel Programming*, will cover using Clojure's parallel processing capabilities to speed up processing data.

Chapter 5, *Distributed Data Processing with Cascalog*, will cover using Cascalog as a wrapper over Hadoop and the Cascading library to process large amounts of data distributed over multiple computers. The final recipe in this chapter will use Pallet to run a simple analysis on Amazon's EC2 service.

Chapter 6, *Working with Incanter Datasets*, will cover the basics of working with Incanter datasets. Datasets are the core data structure used by Incanter, and understanding them is necessary to use Incanter effectively.

Chapter 7, *Preparing for and Performing Statistical Data Analysis with Incanter*, will cover 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*, will talk about setting up Clojure to talk to Mathematica or R. These are powerful data analysis systems, and sometimes we might want to use them. This chapter will show us how to get these systems to work together, as well as some tasks we can do once they are communicating.

Chapter 9, *Clustering, Classifying, and Working with Weka*, will cover more advanced machine learning techniques. In this chapter, we'll primarily use the Weka machine learning library, and 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, *Graphing in Incanter*, will show 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 11, *Creating Charts for the Web*, will show how to set up a simple web application to present findings from data analysis. It will include a number of recipes that leverage the powerful D3 visualization library.

What you need for this book

One piece of software required for this book is the **Java Development Kit** (**JDK**), which you can get from `http://www.oracle.com/technetwork/java/javase/downloads/index.html`. The JDK is necessary to run and develop on the Java platform.

The other major piece of software that you'll need is **Leiningen 2**, which you can download and install from `https://github.com/technomancy/leiningen`. Leiningen 2 is a tool for managing Clojure projects and their dependencies. It's quickly becoming the de facto standard project tool in the Clojure community.

Throughout this book, we'll use a number of other Clojure and Java libraries, including Clojure itself. Leiningen will take care of downloading these for us as we need them.

You'll also need a text editor or **integrated development environment** (**IDE**). If you already have a text editor that you like, you can probably use it. See `http://dev.clojure.org/display/doc/Getting+Started` for tips and plugins for using your particular favorite environment. If you don't have a preference, I'd suggest looking at using Eclipse with Counterclockwise. There are instructions for getting this set up at `http://dev.clojure.org/display/doc/Getting+Started+with+Eclipse+and+Counterclockwise`.

That is all that's required. However, at various places throughout the book, some recipes will access other software. The recipes in *Chapter 8*, *Working with Mathematica and R*, that relate to Mathematica will require Mathematica, obviously, and those that relate to R, will require that. However, these programs won't be used in the rest of the book, and whether you're interested in these recipes might depend on whether you already have this software available.

Who this book is for

This book is for programmers or data scientists who are familiar with Clojure and want to use it in their data analysis processes. This isn't a tutorial on Clojure—there are already a number of excellent introductory books out there—so you'll need to be familiar with the language; however, you don't need to be an expert at it.

Likewise, you don't need to be an expert on data analysis, although you should probably be familiar with its tasks, processes, and techniques. While you might be able to glean enough from these recipes to get started, to be truly effective, you'll want to get a more thorough introduction to this field.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: " We just need to make sure that the `clojure.string/upper-case` function is available."

A block of code is set as follows:

```
(defn fuzzy=
  "This returns a fuzzy match."
  [a b]
  (let [dist (fuzzy-dist a b)]
    (or (<= dist fuzzy-max-diff)
        (<= (/ dist (min (count a) (count b)))
            fuzzy-percent-diff))))
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
        [ring.middleware.file-info :only (wrap-file-info)]
        [ring.middleware.stacktrace :only (wrap-stacktrace)]
        [ring.util.response :only (redirect)]
        [hiccup core element page]
        [hiccup.middleware :only (wrap-base-url)]))
```

Any command-line input or output is written as follows:

```
$ lein cljsbuild auto
Compiling ClojureScript.
Compiling "resources/js/scripts.js" from "src-cljs"...
Successfully compiled "resources/js/script.js" in 4.707129 seconds.
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "errors are found in the page **Agents and Asynchronous Actions** in the Clojure documentation ".

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading 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.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Importing Data for Analysis

In this chapter, we will cover:

- 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
- Reading RDF data with SPARQL
- Aggregating data from different formats

Introduction

There's not a lot of data analysis that we can do without data, so the first step in any project is evaluating what data we have and what we need. And 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 for doing statistical analysis and graphics in Clojure, similar to R. Incanter may not be suitable for every task—later we'll use the Weka library for clustering and machine learning—but it is still an important part of our toolkit for doing data analysis in Clojure. This chapter has a collection of recipes for gathering data and making it accessible to Clojure. For the very first recipe, we'll look at how to start a new project. We'll start with very simple formats like comma-separated values (CSV) and move into reading data from relational databases using JDBC. Then 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 need a tool to download them and track them. We also need a tool to set up the environment and start a **read-eval-print-loop** (**REPL**, or interactive interpreter), which can access our code, or to execute our program.

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

Getting ready

Visit the Leiningen site (`http://leiningen.org/`) and download the `lein` script. This will download the Leiningen JAR file. 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 it the name of the project:

```
$ 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'.
```

Now, there will be a new subdirectory named `getting-data`. It will contain files with stubs for the `getting-data.core` namespace and for tests.

Downloading 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.

How it works...

The new project directory also contains a file named `project.clj`. This file contains metadata about the project: its name, version, and license. It also contains a list of dependencies that our code will use. The specifications it uses allows it to search Maven repositories and directories of Clojure libraries (Clojars, `https://clojars.org/`) to download the project's dependencies.

```
(defproject getting-data "0.1.0-SNAPSHOT"
  :description "FIXME: write description"
  :url "http://example.com/FIXME"
  :license {:name "Eclipse Public License"
            :url "http://www.eclipse.org/legal/epl-v10.html"}
  :dependencies [[org.clojure/clojure "1.4.0"]])
```

In the *Getting ready* section of each recipe, we'll see what libraries we need to list in the `:dependencies` section of this file.

Reading CSV data into Incanter datasets

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

Getting ready

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

```
:dependencies [[org.clojure/clojure "1.4.0"]
               [incanter/incanter-core "1.4.1"]
               [incanter/incanter-io "1.4.1"]]
```

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

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

Finally, I have a file named `data/small-sample.csv` that contains the following data:

```
Gomez,Addams,father
Morticia,Addams,mother
Pugsley,Addams,brother
Wednesday,Addams,sister
…
```

You can download this file from `http://www.ericrochester.com/clj-data-analysis/data/small-sample.csv`. There's a version with a header row at `http://www.ericrochester.com/clj-data-analysis/data/small-sample-header.csv`.

How to do it...

1. Use the `incanter.io/read-dataset` function:

```
user=> (read-dataset "data/small-sample.csv")
[:col0 :col1 :col2]
["Gomez" "Addams" "father"]
["Morticia" "Addams" "mother"]
["Pugsley" "Addams" "brother"]
["Wednesday" "Addams" "sister"]
…
```

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

```
user=> (read-dataset "data/small-sample-header.csv" :header true)
[:given-name :surname :relation]
["Gomez" "Addams" "father"]
["Morticia" "Addams" "mother"]
["Pugsley" "Addams" "brother"]
```

How it works...

Using Clojure and Incanter makes a lot of common tasks easy. This is a good example of that.

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, some numeric data. Incanter tries to detect automatically when a column contains numeric data and coverts it to a Java `int` or `double`. Incanter takes away a lot of the pain of importing data.

There's more...

If we don't want to involve Incanter—when you don't want the added dependency, for instance—`data.csv` is also simple (`https://github.com/clojure/data.csv`). We'll use this library in later chapters, for example, in the recipe *Lazily processing very large datasets* of *Chapter 2*, *Cleaning and Validating Data*.

See also

- *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 fuller data model than CSV, we may need to transform the data. In that case, we can pull out just 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, include these dependencies in the Leiningen `project.clj` file:

```
  :dependencies [[org.clojure/clojure "1.4.0"]
                 [incanter/incanter-core "1.4.1"]
                 [org.clojure/data.json "0.2.1"]]
```

Use these libraries in our REPL interpreter or in our program:

```
(use 'incanter.core
     'clojure.data.json)
```

And have some data. For this, I have a file named `data/small-sample.json` that looks like the following:

```
[{"given_name": "Gomez",
  "surname": "Addams",
  "relation": "father"},
 {"given_name": "Morticia",
  "surname": "Addams",
  "relation": "mother"}, …
]
```

You can download this data file from `http://www.ericrochester.com/clj-data-analysis/data/small-sample.json`.

How to do it...

Once everything's in place, this is just a one-liner, which we can execute at the REPL interpreter:

```
user=> (to-dataset (read-json (slurp "data/small-sample.json")))
[:given_name :surname :relation]
["Gomez" "Addams" "father"]
["Morticia" "Addams" "mother"]
["Pugsley" "Addams" "brother"]
…
```

How it works...

Like all Lisps, Clojure is usually read from inside out, from right to left. Let's break it down. `clojure.core/slurp` reads in the contents of the file and returns it as a string. This is obviously a bad idea for very large files, but for small ones it's handy. `clojure.data.json/read-json` takes the data from `slurp`, parses it as JSON, and returns native Clojure data structures. In this case, it returns a vector of maps. `maps.incanter.core/to-dataset` takes a sequence of maps and returns an Incanter dataset. This will use the keys in the maps as column names and will convert the data values into a matrix. Actually, `to-dataset` can accept many different data structures. Try `(doc to-dataset)` in the REPL interpreter or see the Incanter documentation at `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; reading an Excel spreadsheet is another example of this.

Getting ready

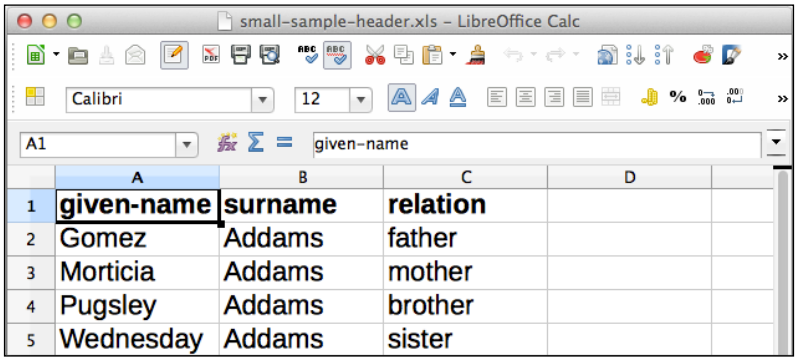
First, make sure that our Leiningen `project.clj` file contains the right dependencies:

```
:dependencies [[org.clojure/clojure "1.4.0"]
               [incanter/incanter-core "1.4.1"]
               [incanter/incanter-excel "1.4.1"]]
```

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

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

And find the Excel spreadsheet we want to work on. I've named mine `data/small-sample-header.xls`. You can download this from `http://www.ericrochester.com/clj-data-analysis/data/small-sample-header.xls`.



	A	B	C	D
1	given-name	surname	relation	
2	Gomez	Addams	father	
3	Morticia	Addams	mother	
4	Pugsley	Addams	brother	
5	Wednesday	Addams	sister	

How to do it...

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

```
user=> (read-xls "data/small-sample-header.xls")
["given-name" "surname" "relation"]
["Gomez" "Addams" "father"]
["Morticia" "Addams" "mother"]
["Pugsley" "Addams" "brother"]
…
```

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 is involved in connecting to the database.

Fortunately, there's a Clojure-contributed package that sits on top of JDBC and makes working with databases much easier. In this example, we'll load a table from an SQLite database (`http://www.sqlite.org/`).

Getting ready

First, list the dependencies in our Leiningen `project.clj` file. We also need to include the database driver library. For this example that's `org.xerial/sqlite-jdbc`.

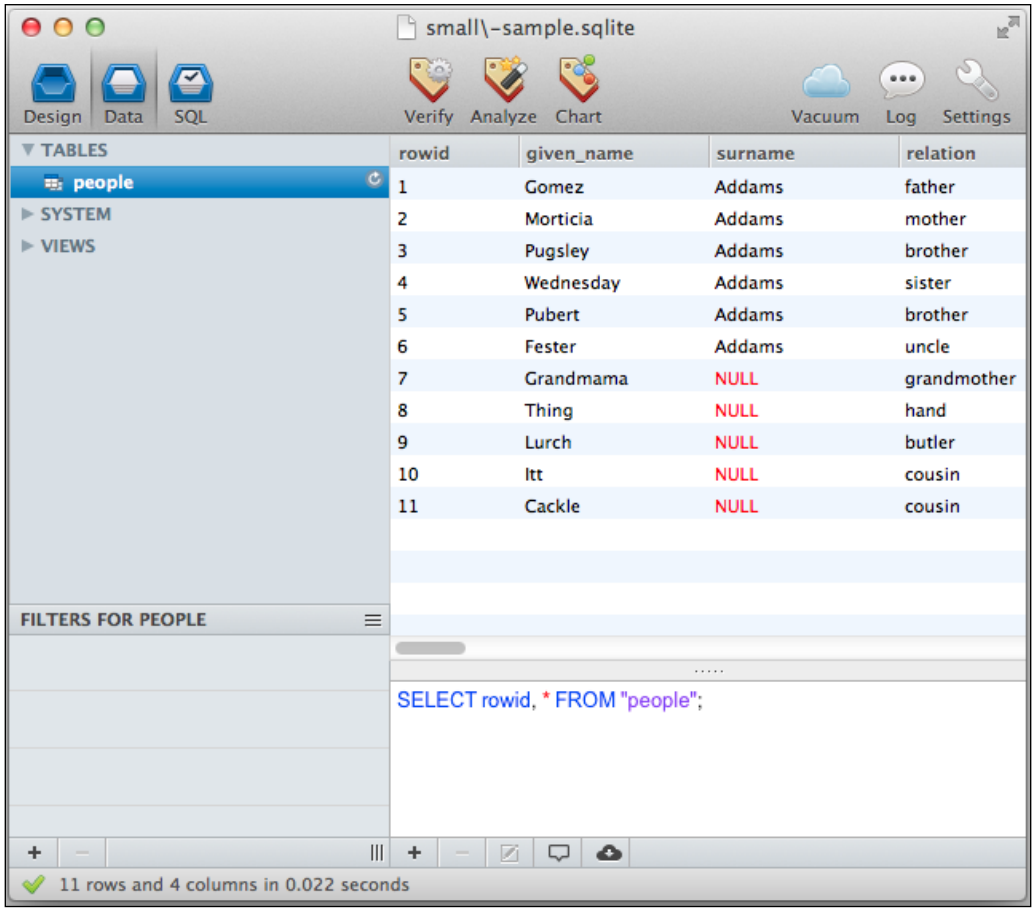
```
:dependencies [[org.clojure/clojure "1.4.0"]
               [incanter/incanter-core "1.4.1"]
```

```
                [org.clojure/java.jdbc "0.2.3"]
                [org.xerial/sqlite-jdbc "3.7.2"]]
```

Then load the modules into our REPL interpreter or script file:

```
(use '[clojure.java.jdbc :exclude (resultset-seq)]
     'incanter.core)
```

Finally, get the database connection information. I have my data in a SQLite database file named `data/small-sample.sqlite`. You can download this from `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 even easier with a wrapper function.

1. Create a function that takes a database connection map and a table name and returns a dataset created from that table:

```
(defn load-table-data
  "This loads the data from a database table."
  [db table-name]
  (let [sql (str "SELECT * FROM "
                 table-name ";")]
    (with-connection db
      (with-query-results rs [sql]
        (to-dataset (doall rs))))))
```

2. Next, define a database map with the connection parameters suitable for our database:

```
(def db {: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:

```
user=> (load-table-data db 'people)
[:relation :surname :given_name]
["father" "Addams" "Gomez"]
["mother" "Addams" "Morticia"]
["brother" "Addams" "Pugsley"]
…
```

How it works...

The `load-table-data` function sets up a database connection using `clojure.java.jdbc/with-connection`. It creates a SQL query that queries all the fields of the table passed in. It then retrieves the results using `clojure.java.jdbc/with-query-results`. Each result row is a sequence of maps of column names to 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 very dependent on what database we wish to connect to. Oracle has a tutorial for working with JDBC at `http://docs.oracle.com/javase/tutorial/jdbc/basics/`, and the documentation for the `clojure.java.jdbc` library has some good information also (`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 online. This one, `http://www.java2s.com/Tutorial/Java/0340__Database/AListofJDBCDriversconnectionstringdrivername.htm`, includes the major drivers.

Reading XML data into Incanter datasets

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

Getting ready

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

```
:dependencies [[org.clojure/clojure "1.4.0"]
               [incanter/incanter-core "1.4.1"]]
```

Use these libraries in our REPL interpreter or program:

```
(use 'incanter.core
     'clojure.xml
     '[clojure.zip :exclude [next replace remove]])
```

And find a data file. I have a file named `data/small-sample.xml` that looks like the following:

```
<?xml version="1.0" encoding="utf-8"?>
<data>
  <person>
    <given-name>Gomez</given-name>
    <surname>Addams</surname>
    <relation>father</relation>
  </person>
  …
```

You can download this data file from `http://www.ericrochester.com/clj-data-analysis/data/small-sample.xml`.

How to do it...

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

```
(defn load-xml-data [xml-file first-data next-data]
  (let [data-map (fn [node]
                   [(:tag node) (first (:content node))])]
    (->>
      ;; 1. Parse the XML data file;
      (parse xml-file)
      xml-zip
      ;; 2. Walk it to extract the data nodes;
      first-data
      (iterate next-data)
      (take-while #(not (nil? %)))
      (map children)
      ;; 3. Convert them into a sequence of maps; and
      (map #(mapcat data-map %))
      (map #(apply array-map %))
      ;; 4. Finally convert that into an Incanter dataset
      to-dataset)))
```

2. Which we call in the following manner:

```
user=> (load-xml-data "data/small-sample.xml" down right)
[:given-name :surname :relation]
["Gomez" "Addams" "father"]
["Morticia" "Addams" "mother"]
["Pugsley" "Addams" "brother"]
…
```

How it works...

This recipe follows a typical pipeline for working with XML:

1. It parses an XML data file.
2. It walks it to extract the data nodes.
3. It converts them into a sequence of maps representing the data.
4. And finally, it converts that into an Incanter dataset.

`load-xml-data` implements this process. It takes three parameters. The input file name, a function that takes the root node of the parsed XML and returns the first data node, and 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 discuss more about zippers in a later section). Then it uses the two functions passed in to extract all the data nodes as a sequence. For each data node, it gets its 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 has 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 file is it gets passed to `clojure.zip/xml-zip`. 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 prefers 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 how to work with immutable data structures. For more information on zippers, the Clojure-doc page for this is helpful (`http://clojure-doc.org/articles/tutorials/parsing_xml_with_zippers.html`). But if you rather like diving 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

Also, we've used the `->>` macro to express our process as a pipeline. For deeply nested function calls, this macro lets us read it from right to left, 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. `->>` simply rewrites the calls into Clojure's native, nested format, as the form is read. The first parameter to the macro is inserted into the next expression as the last parameter. That 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 `(->> x first (map length) (apply +))` expression. The following is a list of each intermediate step that occurs as Clojure builds the final expression (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. XML, meanwhile, 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 domain, `first_name` or `age`, for instance. However, the keys of the maps for XML will come from the data format, **tag**, **attribute**, or **children**, say, 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 get to. It's buried in tables, or articles, or deeply nested div tags. Web scraping is brittle and laborious, but it's often the only way to free this data so we can use it in our analyses. This recipe describes how to load a web page and dig down into its contents so you can pull the data out.

To do this, we're going to use the Enlive library (`https://github.com/cgrand/enlive/wiki`). This uses a **domain-specific language** (**DSL**) based on CSS selectors for locating 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

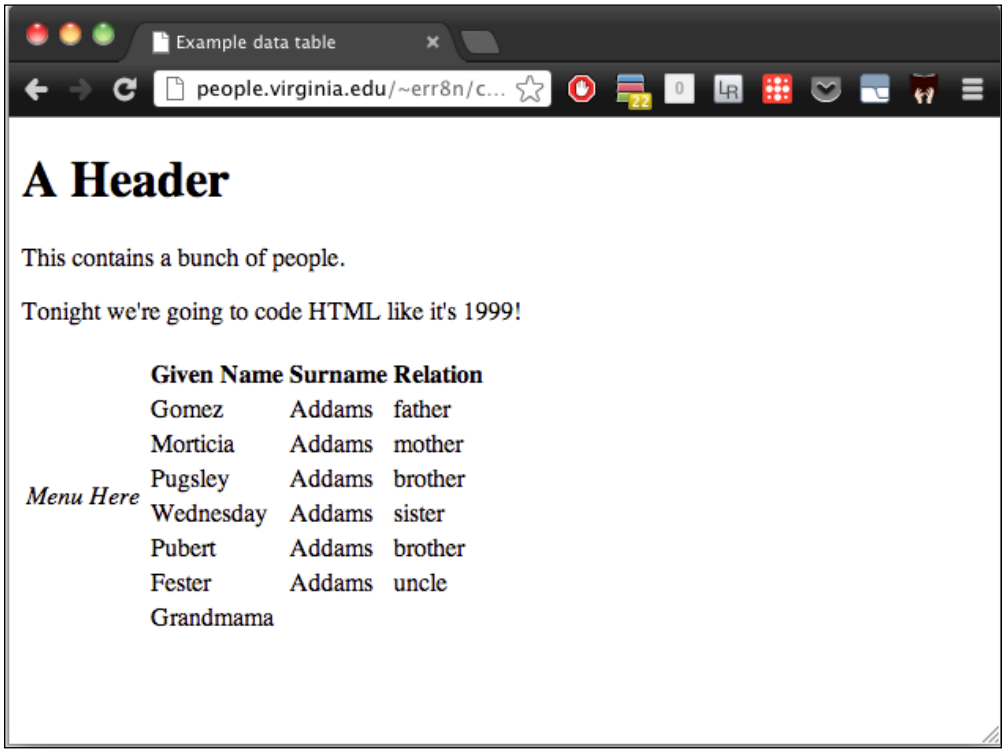
First we have to add Enlive to the dependencies of the project:

```
:dependencies [[org.clojure/clojure "1.4.0"]
               [incanter/incanter-core "1.4.1"]
               [enlive "1.0.1"]]
```

Next, we use those packages in our REPL interpreter or script:

```
(require '(clojure [string :as string]))
(require '(net.cgrand [enlive-html :as html]))
(use 'incanter.core)
(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 the following:



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 the steps out into several functions:

```
(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 [html (html/html-resource (URL. url))
        table (html/select html [: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 %))
                  (filter seq))]
    (dataset headers rows)))
```

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

```
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 take them one by one.

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

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

The next binding selects the table with the ID `data`:

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

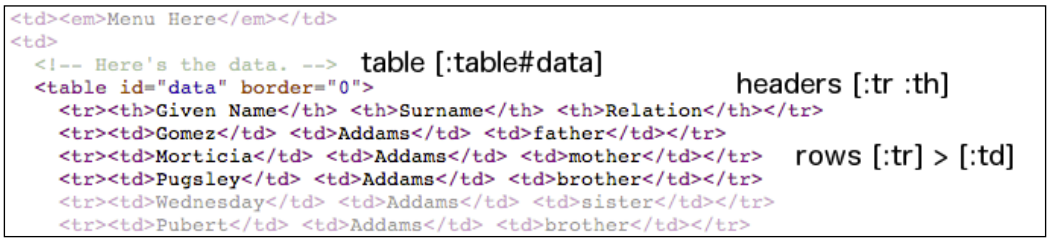
Now, we select all header cells from the table, extract the text from them, convert each to a keyword, and then the whole sequence into a vector. This gives us our headers for the dataset:

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

We first select each row individually. The next two steps are wrapped in `map` so that the cells in each row stay grouped together. In those steps, we select the data cells in each row and extract the text from each. And lastly, we filter using `seq`, which removes any rows with no data, such as the header row:

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

Here is another view of this data. In the following screenshot, we can see some of the code from this web page. The variable names and the 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.



Finally, we convert everything to a dataset. `incanter.core/dataset` is a lower-level constructor than `incanter.core/to-dataset`. It requires us to pass in the column names and data matrix as separate sequences:

```
        (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. Then I start REPL and parse the web page there. Then, I can look at the web page and HTML with the browser's "view source" functionality, and I can examine the data from the web page interactively in the REPL interpreter. While working, I copy and paste the code back and forth between the REPL interpreter and my text editor, as it's convenient. This workflow and environment makes screen scraping—a fiddly, difficult task even when all goes well—almost enjoyable.

See also

- The *Scraping textual data from web pages* recipe
- The *Aggregating data from different formats* recipe