

6

Working with Incanter Datasets

In this chapter, we will cover:

- ▶ Loading Incanter's sample datasets
- ▶ Loading Clojure data structures into datasets
- ▶ Viewing datasets interactively with view
- ▶ Converting datasets to matrices
- ▶ Using infix formulas in Incanter
- ▶ Selecting columns with \$
- ▶ Selecting rows with \$
- ▶ Filtering datasets with \$where
- ▶ Grouping data with \$group-by
- ▶ Saving datasets to CSV and JSON
- ▶ Projecting from multiple datasets with \$join

Introduction

We've seen Incanter (<http://incanter.org/>) earlier in this book, but we'll spend a lot more time with that library over the next few chapters. Incanter combines the power of doing statistics using a fully-featured statistical language such as R with the ease and joy of Clojure.

Incanter's core data structure is the dataset, so we'll be spending some time in this chapter looking at how to use them effectively. Learning basic tools like this is often not the most exciting way to spend our time, but it can still be incredibly useful. At its most fundamental level, an Incanter dataset is a table of rows. Each row has the same set of columns, much like a spreadsheet. The data in each cell of an Incanter dataset can be a string or numeric.

First, we'll learn how to populate and view datasets, and then we'll learn different ways to query and project the parts of the dataset that we're interested in onto a new dataset. Finally, we'll look at saving datasets and merging multiple datasets together.

Loading Incanter's sample datasets

Incanter comes with a set of default datasets that are useful for exploring Incanter's functions. I haven't made use of them in this book, since there is so much data available at other places, but they're a great way to get a feel for what we can do with Incanter. Some of these datasets, for instance, the Iris dataset, are widely used for teaching. That's the dataset we'll access today.

In this recipe, we'll load a dataset and see what it contains.

Getting ready

We'll need to include Incanter in our Leiningen `project.clj` file.

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

We'll also need to include the right Incanter namespaces into our script or REPL.

```
(use '(incanter core datasets))
```

How to do it...

Once the namespaces are available, we can access the datasets easily.

```
user=> (def iris (get-dataset :iris))
#'user/iris
user=> (col-names iris)
[:Sepal.Length :Sepal.Width :Petal.Length :Petal.Width :Species]
user=> (nrow iris)
150
user=> (set ($ :Species iris))
#{"versicolor" "virginica" "setosa"}
```

How it works...

We use the `get-dataset` function to access the built-in datasets. In this case, we're loading the Fisher's or Anderson's Iris data. This is a multivariate dataset for discriminant analysis. It gives petal and sepal measurements for fifty each of three different species of Iris.

Incanter's sample datasets cover a wide variety of topics—from US arrests, to plant growth, to ultrasonic calibration. They can be used for testing different algorithms and analyses and for working with different types of data.

There's more...

Incanter's API documentation for `get-dataset` (<http://liebke.github.com/incanter/datasets-api.html#incanter.datasets/get-dataset>) lists more sample datasets, and you should refer to that for the latest information about the data that Incanter bundles.

Loading Clojure data structures into datasets

While good for learning, Incanter's built-in datasets probably won't be that useful for your work (unless you work with Irises). Other recipes cover ways to get data from CSV files and other sources into Incanter (refer to *Chapter 1, Importing Data for Analysis*). Incanter also accepts native Clojure data structures in a number of formats. We'll look at a couple of those in this recipe.

Getting ready

We'll just need Incanter listed in our `project.clj` file.

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

We'll need to include it in our script or REPL.

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

How to do it...

The primary function for converting data into a dataset is `to-dataset`. While it can convert single, scalar values into a dataset, we'll start with slightly more complicated inputs:

1. Generally, we'll be working with at least one matrix. If we pass that to `to-dataset`, what do we get?

```
user=> (def matrix-set (to-dataset [[1 2 3] [4 5 6]]))
#'user/matrix-set
user=> (nrow matrix-set)
2
user=> (col-names matrix-set)
[:col-0 :col-1 :col-2]
```

2. All the data is there, but it could be labeled better. Does `to-dataset` handle maps?

```
user=> (def map-set (to-dataset {:a 1, :b 2, :c 3}))
#'user/map-set
user=> (nrow map-set)
1
user=> (col-names map-set)
[:a :c :b]
```

3. So map keys become the column labels. That's much more intuitive. Let's throw a sequence of maps at it.

```
user=> (def maps-set (to-dataset [{:a 1, :b 2, :c 3},
                                  {:a 4, :b 5, :c 6}]))
#'user/maps-set
user=> (nrow maps-set)
2
user=> (col-names maps-set)
[:a :c :b]
```

4. That's much more useful. We can also create a dataset by passing the column vector and the row matrix separately to `dataset`.

```
user=> (def matrix-set-2
        (dataset [:a :b :c]
                  [[1 2 3] [4 5 6]]))
#'user/matrix-set-2
user=> (nrow matrix-set-2)
2
user=> (col-names matrix-set-2)
[:c :b :a]
```

How it works...

The function `to-dataset` looks at the input and tries to process it intelligently. If given a sequence of maps, the column names are taken from the keys of the first map in the sequence.

Ultimately, it uses the `dataset` constructor to create the dataset. It requires the dataset to be passed in as a column vector and a row matrix. When the data is in this format, or when we need to most control—to rename the columns, for instance—we can use `dataset`.

See also

Several recipes in *Chapter 1, Importing Data for Analysis*, look at how to load data from different external sources into Incanter datasets.

Viewing datasets interactively with view

Being able to interact with our data programmatically is important, but sometimes it's also helpful to be able to look at it. This can be especially useful while doing data exploration.

Getting ready

We'll need to have Incanter in our `project.clj` file and script or REPL, so we'll use the same set up as we did for the *Loading Incanter's sample datasets* recipe. We'll also use the Iris dataset from that recipe.

How to do it...

Incanter makes this very easy. Let's look at just how simple it is:

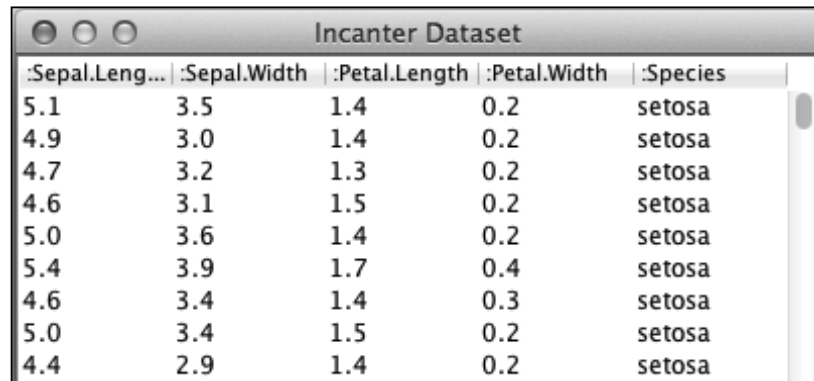
1. First, we need to load the dataset.

```
user=> (def iris (get-dataset :iris))
#'user/iris
```

2. Then we just call `view` on the dataset:

```
user=> (view iris)
#<JFrame javax.swing.JFrame[frame0,0,22,400x600,invalid,layout=
=java.awt.BorderLayout,title=Incanter
Dataset,resizable,normal,defaultCloseOperation=
HIDE_ON_CLOSE,rootPane=javax.swing.JRootPane
[,0,22,400x578,invalid,layout=javax.swing.
JRootPane$RootLayout,alignmentX=0.0,alignmentY=
0.0,border=,flags=16777673,maximumSize=,minimumSize=
,preferredSize=],rootPaneCheckingEnabled=true]>
```

This function returns the Swing window frame, which contains our data. That window should also be open on your desktop, although for me it's usually hiding behind another window. It is shown in the following screenshot:



The screenshot shows a window titled "Incanter Dataset" with a table containing 10 rows of data. The columns are labeled :Sepal.Length, :Sepal.Width, :Petal.Length, :Petal.Width, and :Species. All species listed are setosa.

:Sepal.Length	:Sepal.Width	:Petal.Length	:Petal.Width	:Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa
4.6	3.4	1.4	0.3	setosa
5.0	3.4	1.5	0.2	setosa
4.4	2.9	1.4	0.2	setosa

How it works...

Incanter's `view` function takes any object and tries to display it graphically. In this case, it simply displays the raw data as a table. We'll use this function a lot in *Chapter 10, Graphing in Incanter* when we talk about Incanter's graphing functionality.

See also

In *Chapter 10, Graphing in Incanter*, we'll see more sophisticated, exciting ways to visualize Incanter datasets.

Converting datasets to matrices

Although datasets are often convenient, many times we'll want something a bit faster. Incanter matrices store a table of doubles. This provides good performance in a compact data structure. We'll also need matrices many times because some of Incanter's functions, `trans`, for example, only operate on a single matrix.

Also, it implements Clojure's `ISEq` interface, so interacting with matrices is also convenient.

Getting ready

For this recipe, we'll need the Incanter libraries, so we'll use this `project.clj` file:

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

We'll use the `core` and `io` namespaces, so we'll load those into our script or REPL:

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

We'll use the Virginia census data that we've used periodically throughout the book. Refer to the *Managing program complexity with STM* recipe from *Chapter 3, Managing Complexity with Concurrent Programming*, for information on how to get this dataset. You can also download it from http://www.ericrochester.com/clj-data-analysis/data/all_160_in_51.P35.csv.

```
(def data-file "data/all_160_in_51.P35.csv")
```

How to do it...

For this recipe, we'll create a dataset, convert it to a matrix, and then perform some operations on it using the following steps:

1. First, we need to read the data into a dataset.

```
(def va-data (read-dataset data-file :header true))
```

2. Then, to convert it to a matrix, we just pass it to the `to-matrix` function. Before we do that, we'll pull out a few of the columns, since matrices can only contain floating point numbers.

```
(def va-matrix
  (to-matrix ($ [:POP100 :HU100 :P035001] va-data)))
```

3. Now that it's a matrix, we can treat it like a sequence. Here we pass it to `first` to get the first row, to `take` to get a subset of the matrix, and to `count` to get the number of rows in the matrix:

```
user=> (first va-matrix)
[8191.0000 4271.0000 2056.0000]

user=> (take 5 va-matrix)
([8191.0000 4271.0000 2056.0000]
 [519.0000 229.0000 117.0000]
 [298.0000 163.0000 77.0000]
 [139966.0000 72376.0000 30978.0000]
 [117.0000 107.0000 32.0000]
)
user=> (count va-matrix)
591
```

4. We can also use Incanter's matrix operators to get the sum of each column, for instance:

```
user=> (reduce plus va-matrix)
[5433225.0000 2262695.0000 1332421.0000]
```

How it works...

The `to-matrix` function takes a dataset of floating-point values and returns a compact matrix. Matrices are used by many of Incanter's more sophisticated analysis functions, as they're easy to work with.

There's more...

In this recipe, we saw the `plus` matrix operator. Incanter defines a full suite of these. You can learn more about matrices and see what operators are available at <https://github.com/liebke/incanter/wiki/matrices>.

See also

- ▶ The *Selecting columns with \$* recipe

Using infix formulas in Incanter

There's a lot to like about lisp: macros, the simple syntax, and the rapid development cycle. Most of the time, it is fine that we treat math operators like functions and use prefix notation, which is a consistent, function-first syntax. This allows us to treat math operators the same as everything else so that we can pass them to `reduce`, or anything else we want to do.

But we're not taught to read math expressions using prefix notation (with the operator first). Especially when formulas get even a little complicated, tracing out exactly what's happening can get hairy.

Getting ready

For this, we'll just need Incanter in our `project.clj` file, so we'll use the dependencies statement—as well as the `use` statement—from the *Loading Clojure data structures into datasets* recipe.

For data, we'll use the matrix that we created in the *Converting datasets to matrices* recipe.

How to do it...

Incanter has a macro that converts standard math notation to lisp notation. We'll explore that in this recipe:

1. The `$=` macro changes its contents to use infix notation, which is what we're used to from math class.

```
user=> ($= 7 * 4)
```



```
28
user=> ($= 7 * 4 + 3)
31
```

2. We can also work on whole matrices or just parts of matrices. In this example, we perform scalar multiplication of the matrix.

```
user=> ($= va-matrix * 4)
[ 32764.0000 17084.0000 8224.0000
  2076.0000  916.0000  468.0000
  1192.0000  652.0000  308.0000
...
user=> ($= (first va-matrix) * 4)
[32764.0000 17084.0000 8224.0000]
```

3. Using this we can build complex expressions, such as the following one that takes the mean of the values in the first row of the matrix:

```
user=> ($= (sum (first va-matrix)) /
(count (first va-matrix)))
4839.333333333333
```

4. Or the following expression, which takes the mean of each column:

```
user=> ($= (reduce plus va-matrix) / (count va-matrix))
[9193.2741 3828.5871 2254.5195]
```

How it works...

Whenever we're working with macros and we wonder how they work, we can always get at their output expressions easily, so that we can see what the computer is actually executing. The tool to do this is `macroexpand-1`. This expands the macro one step and returns the result. Its sibling function, `macroexpand`, expands the expression until there is no longer a macro expression. Usually, this is more than we want, so we just use `macroexpand-1`.

Let's see what the following macros expand into:

```
user=> (macroexpand-1 '($= 7 * 4))
(incanter.core/mult 7 4)
user=> (macroexpand-1 '($= 7 * 4 + 3))
(incanter.core/plus (incanter.core/mult 7 4) 3)
user=> (macroexpand-1 '($= 3 + 7 * 4))
(incanter.core/plus 3 (incanter.core/mult 7 4))
```

Here we can see that it doesn't expand into Clojure's `*` or `+` functions, but instead it uses Incanter's matrix functions, `mult` and `plus`. This allows it to handle a variety of input types, including matrices, intelligently.

Otherwise, it switches the expressions around the way we'd expect. We can also see by comparing the last two that it even handles operator precedence correctly.

Selecting columns with \$

Often we need to cut down the data to make it more useful. One common transformation is to pull out all the values from one or more columns into a new dataset. This can be useful for generating summary statistics or aggregating the values of some columns.

The Incanter macro `$` slices out parts of a dataset. In this recipe, we'll see this in action.

Getting ready

For this recipe, we'll need to have Incanter listed in our `project.clj` file:

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

We'll also need to include `incanter.core` and `incanter.io` in our script or REPL.

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

We'll also need some data. This time we'll use the race data from the US census data available at <http://censusdata.ire.org/>. However, instead of using the data for one state we'll use all states' data. These have to be downloaded separately and joined together. I've already done this, and the file is available for download at http://www.ericrochester.com/clj-data-analysis/data/all_160.P3.csv.

To make this data easy to access, we can bind the file name for that data to `data-file`. We'll then load the dataset and bind it to the name `race-data`.

```
(def data-file "data/all_160.P3.csv")
(def race-data (read-dataset data-file :header true))
```

How to do it...

We'll use the `$` macro several different ways to get different results as seen in the following steps:

1. We can select columns to pull out from the dataset by passing the column names or numbers to the `$` macro. It returns a sequence of the values in the column.

```
user=> ($ :POP100 race-data)
(192 2688 4522 758 356 30352 21160 14875 3917 2486 ...)
```

2. We can select more than one column by listing all of them in a vector. This time, the results are in a dataset.

```
user=> ($ [:STATE :POP100 :POP100.2000] race-data)
[:STATE :POP100 :POP100.2000]
[1 192 ""]
[1 2688 2987]
[1 4522 4965]
[1 758 723]
[1 356 521]
...
```

3. We can list as many columns as we want.

```
user=> ($ [:STATE :POP100 :P003002 :P003003
          :P003004 :P003005 :P003006 :P003007
          :P003008]
        race-data)
[:STATE :POP100 :P003002 :P003003 :P003004 :P003005 :P003006
:P003007 :P003008]
[1 192 129 58 0 0 0 2 3]
[1 2688 1463 1113 2 26 0 53 31]
[1 4522 2366 2030 23 14 1 50 38]
[1 758 751 1 0 1 0 0 5]
[1 356 47 308 0 0 0 0 1]
...
```

How it works...

The `$` function is just a wrapper over the Incanter's `sel` function. It provides a nice way of slicing columns out of the dataset so we can focus only on the data that actually pertain to our analysis.

There's more...

The column headers for this dataset are a little cryptic. The IRE download page for the census data (<http://census.ire.org/data/bulkdata.html>) has a link for the column header information, or you can access that data in CSV format directly at https://raw.githubusercontent.com/ireapps/census/master/tools/metadata/sf1_labels.csv.

In this recipe, I pulled out a number of columns:

- ▶ The total population (`:POP100`)
- ▶ The population for whites (`:P003002`)

- ▶ African-Americans (:P003003)
- ▶ American Indians or Alaska natives (:P003004)
- ▶ Asians (:P003005)
- ▶ Native Hawaiians or Pacific islanders (:P003006)
- ▶ Some other race (:P003007)
- ▶ Two or more races (:P003008)

See also

- ▶ The *Selecting rows with \$* recipe

Selecting rows with \$

The Incanter macro `$` also pulls rows out of a dataset. In this recipe, we'll see this in action.

Getting ready

For this recipe, we'll use the same dependencies, imports, and data that we did in the *Selecting columns with \$* recipe.

How to do it...

Like using `$` to select columns, there are several ways we can use it to select rows. Refer to the following steps:

1. We can create a sequence of the values of one row by using `$` and passing it the index of the row we want and `:all` for the columns.

```
user=> ($ 0 :all race-data)
(100100 160 1 "" "" "" "" "" "Abanda CDP" 192 79 "" "
" 192 "" 129 "" 58 "" 0 "" 0 "" 0 "" 2 "" 3 "")
```

2. We can also pull out a dataset containing multiple rows by passing more than one index into `$` with a vector.

```
user=> ($ [0 1 2 3 4] :all race-data)
[:GEOID :SUMLEV :STATE :COUNTY :CBSA :CSA :NECTA :CNECTA :NAME
:POP100 :HU100 :POP100.2000 :HU100.2000 :P003001 :P003001.2000
:P003002 :P003002.2000 :P003003 :P003003.2000 :P003004
:P003004.2000 :P003005 :P003005.2000 :P003006 :P003006.2000
:P003007 :P003007.2000 :P003008 :P003008.2000]
```

```
[100100 160 1 "" "" "" "" "" "Abanda CDP" 192 79 "" "
" 192 "" 129 "" 58 "" 0 "" 0 "" 0 "" 2 "" 3 ""]
[100124 160 1 "" "" "" "" "" "Abbeville city" 2688 1255 2987 1353
2688 2987 1463 1692 1113 1193 2 0 26 2 0 0 53 85 31 15]
[100460 160 1 "" "" "" "" "" "Adamsville city" 4522 1990 4965 2042
4522 4965 2366 3763 2030 1133 23 20 14 7 1 1 50 8 38 33]
[100484 160 1 "" "" "" "" "" "Addison town" 758 351 723 339 758
723 751 719 1 1 0 1 1 1 0 0 0 0 5 1]
[100676 160 1 "" "" "" "" "" "Akron town" 356 205 521 239 356 521
47 93 308 422 0 0 0 0 0 0 0 0 1 6]
```

3. We can also combine the two ways to slice data to pull specific columns and rows. We can either pull out a single row or multiple rows.

```
user=> ($ 0 [:STATE :POP100 :P003002 :P003003 :P003004 :P003005
:P003006 :P003007 :P003008]
race-data)
(1 192 129 58 0 0 0 2 3)
user=> ($ [0 1 2 3 4]
[:STATE :POP100 :P003002 :P003003 :P003004 :P003005
:P003006 :P003007 :P003008]
race-data)
[:STATE :POP100 :P003002 :P003003 :P003004 :P003005 :P003006
:P003007 :P003008]
[1 192 129 58 0 0 0 2 3]
[1 2688 1463 1113 2 26 0 53 31]
[1 4522 2366 2030 23 14 1 50 38]
[1 758 751 1 0 1 0 0 5]
[1 356 47 308 0 0 0 0 1]
...
```

How it works...

The `$` macro is the workhorse for slicing rows and projecting (or selecting) columns from datasets. When it's called with two indexing parameters, the first is the row or rows, and the second is the column or columns.

Filtering datasets with `$where`

While we can filter datasets before we import them into Incanter, Incanter makes it easy to filter and create new datasets from existing ones. We'll take a look at its query language in this recipe.

Getting ready

We'll use the same dependencies, imports, and data that we did in the *Using infix formulas in Incanter* recipe.

How to do it...

Once we have the data, we query it using the `$where` function.

1. For example, the following creates a dataset with the row for Richmond:

```
user=> (def richmond ($where {:NAME "Richmond city"}
  va-data))
#'user/richmond
user=> richmond
[:GEOID :SUMLEV :STATE :COUNTY :CBSA :CSA :NECTA :CNECTA :NAME
:POP100 :HU100 :POP100.2000 :HU100.2000 :P035001 :P035001.2000]
[5167000 160 51 "" "" "" "" "" "Richmond city"
204214 98349 197790 92282 41304 43649]
```

2. The queries can be more complicated, too. The following one picks out the small towns, ones with population less than 1,000:

```
user=> (def small ($where {:POP100 {:lte 1000}} va-data))
#'user/small
user=> (nrow small)
232
user=> ($ [0 1 2 3] :all small)
[:GEOID :SUMLEV :STATE :COUNTY :CBSA :CSA :NECTA :CNECTA :NAME
:POP100 :HU100 :POP100.2000 :HU100.2000 :P035001 :P035001.2000]
[5100180 160 51 "" "" "" "" "" "Accomac town" 519 229 547 235 117
126]
[5100724 160 51 "" "" "" "" "" "Alberta town" 298 163 306 158 77
86]
[5101256 160 51 "" "" "" "" "" "Allisonia CDP" 117 107 "" "" 32
""]
[5102248 160 51 "" "" "" "" "" "Arcola CDP" 233 96 "" ""
59 ""]
```

3. This one picks out the medium-sized towns, ones with populations between 1,000 and 40,000:

```
user=> (def medium ($where {:POP100 {:gt 1000 :lt 40000}}
  va-data))
#'user/medium
user=> (nrow medium)
```

```

333
user=> ($ [0 1 2 3] :all medium)
[:GEOID :SUMLEV :STATE :COUNTY :CBSA :CSA :NECTA :CNECTA
:NAME :POP100 :HU100 :POP100.2000 :HU100.2000 :P035001
:P035001.2000]
[5100148 160 51 "" "" "" "" "" "Abingdon town" 8191
4271 7780 3788 2056 2091]
[5101528 160 51 "" "" "" "" "" "Altavista town" 3450
1669 3425 1650 928 940]
[5101640 160 51 "" "" "" "" "" "Amelia Court House CDP"
1099 419 "" "" 273 ""]
[5101672 160 51 "" "" "" "" "" "Amherst town" 2231 1032
2251 1000 550 569]

```

Incanter's query language is more powerful than this, but these examples should show us the basic structure and give us an idea of the possibilities.

How it works...

To a better understand how to use `$where`, let's pick apart the last example:

```
($where { :POP100 { :gt 1000 :lt 40000 } } va-data)
```

The query is expressed as a hash-map from fields (highlighted) to values. As we saw in the first example, the value can be a raw value, either a literal or an expression. This tests for equality.

```
($where { :POP100 { :gt 1000 :lt 40000 } } va-data)
```

It can also be another map as it is here (highlighted). The keys of this map are tests, and the values are parameters to those tests. All of the tests in this map are *anded* together, so that the field's values have to pass all predicates.

Incanter supports a number of test operators. Basic Boolean tests are `:$gt` (greater than), `:$lt` (less than), `:$gte` (greater than or equal to), `:$lte` (less than or equal to), `:$eq` (equal to), and `:$ne` (not equal to). There are also some operators that take sets as parameters: `:$in` (in) and `:$nin` (not in).

The last operator—`:$fn`—is interesting. It allows us to use any predicate function. For example, this would randomly select approximately half of the dataset.

```

(def random-half
  ($where { :GEOID { :$fn (fn [_] (< (rand) 0.5)) } } va-data))

```

All of these tests are *anded* together to produce the final result set.

There's more...

For full details of the query language, see the documentation for `incanter.core/query-dataset` (<http://liebke.github.com/incanter/core-api.html#incanter.core/query-dataset>).

Grouping data with `$group-by`

Datasets often come with inherent structure. Two or more rows may have the same value in one column, and we may want to leverage that by grouping those rows together in our analysis.

Getting ready

First, we'll need to declare a dependency on Incanter in the `project.clj` file:

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

Next, we'll include Incanter `core` and `io` in our script or REPL.

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

For data, we'll use the census race data for all states. We first saw this in the *Selecting columns with \$* recipe, and we can download it from http://www.ericrochester.com/clj-data-analysis/data/all_160.P3.csv.

```
(def data-file "data/all_160.P3.csv")
(def race-data (read-dataset data-file :header true))
```

How to do it...

Incanter lets us group rows for further analysis or summarizing with the `$group-by` function. All we need to do is pass the data to `$group-by` with the column or function to group on.

```
(def by-state ($group-by :STATE race-data))
```

How it works...

This function returns a map where each key is a map of the fields and values represented by that grouping. For example, the keys look like this:

```
user=> (take 5 (keys by-state))
```



```
({:STATE 29} {:STATE 28} {:STATE 31} {:STATE 30} {:STATE 25})
We can get the data for Virginia back out by querying the
group map for state 51.
user=> ($ [0 1 2 3] :all (by-state {:STATE 51}))
[:P003005 :SUMLEV :P003008.2000 :HU100.2000 :P003002.2000
:HU100 :P003007.2000 :NAME :GEOID :NECTA :P003006.2000
:P003001.2000 :CBSA :P003001 :P003002 :CSA :P003005.2000
:POP100.2000 :CNECTA :POP100 :COUNTY :P003007 :P003008
:P003004.2000 :P003003.2000 :STATE :P003004 :P003003 :P003006]
[86 160 49 3788 7390 4271 15 "Abingdon town" 5100148 ""
1 7780 "" 8191 7681 "" 50 7780 "" 8191 ""
64 82 10 265 51 15 257 6]
[3 160 4 235 389 229 5 "Accomac town" 5100180 "" 0 547 "
" 519 389 "" 14 547 "" 519 "" 17 3 0 135 51 0 106 1]
[0 160 0 158 183 163 1 "Alberta town" 5100724 "
" 0 306 "" 298 177 "" 1 306 "" 298 "" 2 3 0 121 51 4 112 0]
[8432 160 5483 64251 76702 72376 9467 "Alexandria city"
5101000 "" 112 128283 "" 139966 85186 "" 7249 128283 "
" 139966 "" 9902 5225 355 28915 51 589 30491 141]
```

Saving datasets to CSV and JSON

Once we've gone to the work of slicing, dicing, cleaning, and aggregating our datasets, we might want to save them. Incanter by itself doesn't have a good way to do that. However, with the help of some Clojure libraries, it's not difficult at all.

Getting ready

We'll need to include a number of dependencies in our `project.clj` file.

```
:dependencies [[org.clojure/clojure "1.4.0"]
               [incanter "1.4.1"]
               [org.clojure/data.json "0.2.1"]
               [org.clojure/data.csv "0.1.2"]]
```

We'll also need to include those libraries in our script or REPL.

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

We'll be using the same data file that we introduced in the *Selecting columns with \$* recipe.

How to do it...

This process is really as simple as getting the data and saving it. We'll pull out the state, name of the location, and the population and race data from the larger dataset. We'll use this subset of the data in both formats.

```
(def census2010 ($ [:STATE :NAME :POP100 :P003002 :P003003
:P003004 :P003005 :P003006 :P003007
:P003008]
race-data))
```

Saving data as CSV

To save a dataset as CSV, all in one statement, we open a file and use `clojure.data.csv/write-csv` to write the column names and data to it.

```
(with-open [f-out (io/writer "data/census-2010.csv")]
  (csv/write-csv f-out [(map name (col-names census2010))])
  (csv/write-csv f-out (to-list census2010)))
```

Saving data as JSON

To save a dataset as JSON, we open a file and use `clojure.data.json/write` to serialize the file.

```
(with-open [f-out (io/writer "data/census-2010.json")]
  (json/write (:rows census2010) f-out))
```

How it works...

For CSV and JSON, as well as many other data formats, the process is very similar. Get the data, open the file, and serialize data into it. There will be differences in how the output function wants the data (`to-list` or `:rows`), and there will be differences in how the output function is called. (For instance, is the file handle the first argument or the second?) But generally, outputting datasets will be very similar and relatively simple.

See also

- ▶ The *Reading CSV data into Incanter datasets* recipe in *Chapter 1, Importing Data for Analysis*
- ▶ The *Reading JSON data into Incanter datasets* recipe in *Chapter 1, Importing Data for Analysis*

Projecting from multiple datasets with \$join

So far we've been focusing on splitting datasets up, on dividing them into groups of rows or groups of columns with functions and macros such as `$` or `$where`. However, sometimes we'd like to move in the other direction. We may have two, related datasets, and we'd like to join them together to make a larger one.

Getting ready

First, we'll need to include these dependencies in our `project.clj` file.

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

We'll use the following statements for includes:

```
(use '(incanter core io charts)
      '[clojure.set :only (union)])
```

For our data file, we'll use the census data that we used in the *Converting datasets to matrices* recipe. You can download this from http://www.ericrochester.com/clj-data-analysis/data/all_160_in_51.P35.csv. Save it to `data/all_160_in_51.P35.csv`.

We'll also use a new data file, `data/all_160_in_51.P3.csv`. This contains the race questions from the census for Virginia. I downloaded this also from <http://census.ire.org/>. You can query it from there or download it directly at http://censusdata.ire.org/51/all_160_in_51.P3.csv or http://www.ericrochester.com/clj-data-analysis/data/all_160_in_51.P3.csv.

How to do it...

In this recipe, we'll look at how to join two datasets using Incanter.

1. Once all the data is in place, we first need to load both files into separate datasets.

```
(def family-data (read-dataset
                  "data/all_160_in_51.P35.csv" :header
                  true))
(def racial-data (read-dataset
                 "data/all_160_in_51.P3.csv" :header
                 true))
```

2. Looking at the columns, we can see that there's a fair amount of overlap between them.

```
user=> (set/intersection (set (col-names family-data))
  #_=> (set (col-names racial-data)))
#{:SUMLEV :HU100.2000 :HU100 :NAME :GEOID :NECTA :CBSA
:CSA :POP100.2000 :CNECTA :POP100 :COUNTY :STATE}
```

3. We can project from the `racial-data` dataset to get rid of all of the duplicate columns from it. Any duplicates not listed in the join will get silently dropped. The values kept will be those from the last dataset listed. To make this easier, we'll create a function that returns the columns of the second dataset, minus those found in the first dataset, plus the index column.

```
(defn dedup-second
  [a b id-col]
  (let [a-cols (set (col-names a))]
    (conj (filter #(not (contains? a-cols %))
      (col-names b))
      id-col)))
```

4. We apply that to the `racial-data` dataset to get a copy of it without the duplicate fields.

```
(def racial-short ($ (vec (dedup-second family-data
  racial-data :GEOID))
  racial-data))
```

5. Once it's in place, we merge the full family dataset with the `race` data subset using `$join`.

```
user=> (def all-data
  #_=> ($join [:GEOID :GEOID] family-data racial-short))
#'user/all-data
user=> (col-names all-data)
[:P003005 :SUMLEV :P003008.2000 :P035001 :HU100.2000 :P003002.2000
:HU100 :P003007.2000 :NAME :GEOID :NECTA :P003006.2000
:P003001.2000 :CBSA :P003001 :P003002 :CSA :P003005.2000
:P035001.2000 :POP100.2000 :CNECTA :POP100 :COUNTY :P003007
:P003008 :P003004.2000 :P003003.2000 :STATE :P003004 :P003003
:P003006]
```

6. We can also see that all rows from both input datasets were merged into the final dataset.

```
user=> (= (nrow family-data) (nrow racial-short) (nrow all-data))
true
```

From this point on, we can use `all-data` just as we would use any other Incanter dataset.

How it works...

Let's look at this in more detail.

```
($join [:GEOID :GEOID] family-data racial-short)
```

The pair of column keywords in a vector (`[:GEOID :GEOID]`) are the keys that the datasets will be joined on. In this case, the `:GEOID` column from both datasets are used, but the keys could be different for the two datasets. The first column listed will be from the first dataset (`family-data`). The second column listed will be from the second dataset (`racial-short`).

This returns a new dataset. As I mentioned, in the output, duplicate columns contain only the values from the second dataset. But otherwise, each row is the superset of the corresponding rows from the two input datasets.