**Introduction**

MDArray-jCSV (jCSV for short) is the first and only (as far as I know) multidimensional CSV reader. Multidimensional? Yes... jCSV can read multidimensional data, also known sometimes as “panel” data.

From Wikipedia: “In [statistics](https://en.wikipedia.org/wiki/Statistics) and [econometrics](https://en.wikipedia.org/wiki/Econometrics), the term panel data refers to multi-dimensional [data](https://en.wikipedia.org/wiki/Data_set) frequently involving measurements over time. Panel data contain observations of multiple phenomena obtained over multiple time periods for the same firms or individuals. In [biostatistics](https://en.wikipedia.org/wiki/Biostatistics), the term longitudinal data is often used instead, wherein a subject or cluster constitutes a panel member or individual in a [longitudinal study](https://en.wikipedia.org/wiki/Longitudinal_study).” jCSV makes this definition a bit less strict as it can read observations of multiple phenomena obtained over multiple time periods for multiple firms or individuals.

Other than reading panel data, jCSV is also a very powerful and feature packed CSV reader. The CSV file format is a common format for data exchange between diverse applications. Although it is widely used, surprisingly, there aren't that many good libraries for CSV reading and writing. In Ruby there are a couple of well-known libraries to accomplish this task. First, there is the standard Ruby CSV library that comes with any Ruby implementation. This library according to smarter\_csv (<https://github.com/tilo/smarter_csv>) has the following limitations:

"Ruby's CSV library's API is pretty old, and it's processing of CSV-files returning Arrays of Arrays feels 'very close to the metal'. The output is not easy to use - especially not if you want to create database records from it. Another shortcoming is that Ruby's CSV library does not have good support for huge CSV-files, e.g. there is no support for 'chunking' and/or parallel processing of the CSV-content (e.g. with Resque or Sidekiq).

In order to eliminate those restrictions, smarter\_csv was developed. Although it does remove those restrictions it removes support for Arrays of Arrays. Although such format is really 'very close to metal' in some cases this is actually what is needed. This format is less memory intensive than the 'hash' approach from smarter\_csv and it might make it easier to put the date in a simple table. When reading scientific data, such as a matrix or multidimensional array (panel data), it might also be better to remove headers and informational columns and read the actual data as just a plain array.

jCSV was developed to be the "ultimate" CSV reader (and soon writer). It tries to merge all the good features of standard Ruby CSV library, smarter\_csv, and other CSV libraries from other languages. jCSV is based on Super CSV (<http://super-csv.github.io/super-csv/index.html>), a java CSV library. According to Super CSV web page its motivation is "for Super CSV is to be the foremost, fastest, and most programmer-friendly, free CSV package for Java". jCSV motivation is to bring this view to the Ruby world, and since we are in Ruby, make it even easier and more programmer-friendly.

jCSV reading features are:

* Reads data as lists (Array of Arrays);
* Reads data as maps (Array of hashes);
* Reads multidimensional (panel) data to lists or hashes;
* Reads multidimensional data to vectors, i.e., a multidimensional array (MDArray);
* When reading panel data, use dimensions as keys, allowing random access to any row in the data by use of the key. For instance, if first\_name, last\_name are dimensions, then one can access data by doing data[“John.Smith”];
* Read panel data with the ‘critbit’ reader which *automagically* sorts keys and allows for prefix retrieval of data, i.e., doing data.each(“D”) { } will retrieve all names starting with “D” and give it to the block;
* When reading panel data, organize data as maps of maps (deep\_map);
* Able to read files with headers or no-headers;
* When the file has no-headers, allow the user to provide headers so that reading can be done either as array of arrays, array of hashes, or multidimensional with keys;
* Able to process large CSV-files;
* Able to chunk the input from the CSV file to avoid loading the whole CSV file into memory;
* Able to treat the file as an enumerator, so that reading more data can be done at any time during the script execution, it can be stopped and restarted at any time;
* Able to pass a block to the read method, so data from the CSV file can be directly processed (e.g. Resque.enqueue )
* Allows to have a bit more flexible input format, where comments are possible, and col\_sep, row\_sep can be set to any character sequence, including control characters;
* Able to re-map CSV "column names" to Hash-keys of your choice (normalization);
* Able to ignore "columns" in the input (delete columns);
* Able to change columns´ order, when reading to an Array of Arrays;
* Provide dozens of filters/validators for the data;
* Filters can be chained allowing for complex data manipulation. For instance, suppose one column can have empty values or dollar values. If it is a dollar value, then it should be a float. Consider that the data is stored using a Brazilian locale format, i.e., decimal separator is ‘,’ and grouping is ‘.’ (the reverse of US locale). Suppose also that the value should be in the range of US$ 1.000,00 and US$ 2.000,00 and finally suppose that we actually want to see this data not as dollar amounts but as Brazilian Reais, converted with the day´s current rate. Then this sequence of filters should do it:
  + Jcsv.optional >> Jcsv.float(locale: Brazil) >>
  + Jcsv.in\_range(1000, 2000) >> Jcsv.dynamic { |value| rate \* value }
* Date can be parsed by any of Ruby DateTime formats: httpdate, iso8601, jd, etc.;
* Can filter data by any of the Ruby String methods: [], reverse, gsub, prepend, etc.

**Reading as Lists**

In this section we will read the following 'customer.csv' file. Some things should be observed in the records of this data:

* It has a header;
* 4 rows of data, all with 10 columns;
* Records can have line break in them;
* The mailingAddress column contains data that spans multiple lines
* The favouriteQuote column contains data with escaped quotes.

customerNo,firstName,lastName,birthDate,mailingAddress,married,numberOfKids,favouriteQuote,email,loyaltyPoints

1,John,Dunbar,13/06/1945,"1600 Amphitheatre Parkway Mountain View, CA 94043 United States",,,"""May the Force be with you."" - Star Wars",[jdunbar@gmail.com](https://mail.google.com/mail/?view=cm&fs=1&tf=1&to=jdunbar@gmail.com" \t "_blank),0

2,Bob,Down,25/02/1919,"1601 Willow Rd. Menlo Park, CA 94025 United States",Y,0,"""Frankly, my dear, I don't give a damn."" - Gone With The Wind",[bobdown@hotmail.com](https://mail.google.com/mail/?view=cm&fs=1&tf=1&to=bobdown@hotmail.com),123456

3,Alice,Wunderland,08/08/1985,"One Microsoft Way Redmond, WA 98052-6399 United States",Y,0,"""Play it, Sam. Play ""As Time Goes By."""" - Casablanca",[throughthelookingglass@yahoo.com](https://mail.google.com/mail/?view=cm&fs=1&tf=1&to=throughthelookingglass@yahoo.com" \t "_blank),2255887799

4,Bill,Jobs,10/07/1973,"2701 San Tomas Expressway Santa Clara, CA 95050 United States",Y,3,"""You've got to ask yourself one question: ""Do I feel lucky?"" Well, do ya, punk?"" - Dirty Harry",[billy34@hotmail.com](https://mail.google.com/mail/?view=cm&fs=1&tf=1&to=billy34@hotmail.com" \t "_blank),36

**Simple Interface**

The simplest way of reading a CSV file is as a list or an array of arrays. Reading in this way is a simple call to Jcsv.reader with the filename, in this case the file is called 'customer.csv'. In the next examples we will parse a CSV file and show all the features and options that can be applied for changing the parsing. The file we are reading has headers. Headers are converted from string to symbol.

require 'jcsv'

require 'pp'

# Create a new reader by passing the filename to be parsed

reader = Jcsv.reader("../data/customer.csv")

# now read the whole csv file and stores it in the 'content' variable

content = reader.read

When reading a file with headers, the 'headers' instance variable from reader has the headers read from the file. Headers are converted to symbols:

> p reader.headers

[:customer\_no, :first\_name, :last\_name, :birth\_date, :mailing\_address, :married, :number\_of\_kids, :favourite\_quote, :email, :loyalty\_points]

We now take a look at the content of the file and should note that line breaks were read as \n and quotes are properly escaped with ". The 'content' variable has an array of arrays and each line is an array.

> content.each do |row|

+ p row

+ end

["1", "John", "Dunbar", "13/06/1945", "1600 Amphitheatre Parkway\nMountain View, CA 94043\nUnited States", nil, nil, "\"May the Force be with you.\" - Star Wars", "jdunbar@gmail.com", "0"]

["2", "Bob", "Down", "25/02/1919", "1601 Willow Rd.\nMenlo Park, CA 94025\nUnited States", "Y", "0", "\"Frankly, my dear, I don't give a damn.\" - Gone With The Wind", "bobdown@hotmail.com", "123456"]

["3", "Alice", "Wunderland", "08/08/1985", "One Microsoft Way\nRedmond, WA 98052-6399\nUnited States", "Y", "0", "\"Play it, Sam. Play \"As Time Goes By.\"\" - Casablanca", "throughthelookingglass@yahoo.com", "2255887799"]

["4", "Bill", "Jobs", "10/07/1973", "2701 San Tomas Expressway\nSanta Clara, CA 95050\nUnited States", "Y", "3", "\"You've got to ask yourself one question: \"Do I feel lucky?\" Well, do ya, punk?\" - Dirty Harry", "billy34@hotmail.com", "36"]

**Strings as Keys**

We will now see many of the options for reading files. Options are passed to method reader. First 'strings\_as\_key' when true will not convert headers to symbol. Note also, that the reader method immediately reads the file headers without reading the other lines:

> reader = Jcsv.reader("../data/customer.csv", strings\_as\_keys: true)

+ p reader.headers

["customerNo", "firstName", "lastName", "birthDate", "mailingAddress", "married", "numberOfKids", "favouriteQuote", "email", "loyaltyPoints"]

**Processing with a Block**

One very interesting feature of Ruby CSV libraries is the ability to give a block to the parser and process the data as it is being read. In jCSV this can also be accomplished. Notice also that the block receives line\_no: the last line of the record, row\_no: the row number of the data, row: the actual data read and headers: the headers. With this, we can somehow simulate the array of hash output (but as we will see later, jCSV can read maps):

# read lines and pass them to a block for processing. The block receives line\_no (last line

# of the record),

# row\_no, row and the headers.

# Read file 'customer.csv'. File has headers (this is the default) and we keep the keys as string

reader = Jcsv.reader("../data/customer.csv", headers: true, strings\_as\_keys: true)

reader.read do |line\_no, row\_no, row, headers|

puts "line number: #{line\_no}, row number: #{row\_no}"

headers.each\_with\_index do |head, i|

puts "#{head}: #{row[i]}"

end

puts

end

line number: 4, row number: 2 customerNo: 1 firstName: John lastName: Dunbar birthDate: 13/06/1945 mailingAddress: 1600 Amphitheatre Parkway Mountain View, CA 94043 United States married: numberOfKids: favouriteQuote: "May the Force be with you." - Star Wars email:[jdunbar@gmail.com](https://mail.google.com/mail/?view=cm&fs=1&tf=1&to=jdunbar@gmail.com" \t "_blank) loyaltyPoints: 0

line number: 7, row number: 3 customerNo: 2 firstName: Bob lastName: Down birthDate: 25/02/1919 mailingAddress: 1601 Willow Rd. Menlo Park, CA 94025 United States married: Y numberOfKids: 0 favouriteQuote: "Frankly, my dear, I don't give a damn." - Gone With The Wind email: [bobdown@hotmail.com](https://mail.google.com/mail/?view=cm&fs=1&tf=1&to=bobdown@hotmail.com) loyaltyPoints: 123456

line number: 10, row number: 4 customerNo: 3 firstName: Alice lastName: Wunderland birthDate: 08/08/1985 mailingAddress: One Microsoft Way Redmond, WA 98052-6399 United States married: Y numberOfKids: 0 favouriteQuote: "Play it, Sam. Play "As Time Goes By."" - Casablanca email:[throughthelookingglass@yahoo.com](https://mail.google.com/mail/?view=cm&fs=1&tf=1&to=throughthelookingglass@yahoo.com" \t "_blank) loyaltyPoints: 2255887799

line number: 13, row number: 5 customerNo: 4 firstName: Bill lastName: Jobs birthDate: 10/07/1973 mailingAddress: 2701 San Tomas Expressway Santa Clara, CA 95050 United States married: Y numberOfKids: 3 favouriteQuote: "You've got to ask yourself one question: "Do I feel lucky?" Well, do ya, punk?" - Dirty Harry email: [billy34@hotmail.com](https://mail.google.com/mail/?view=cm&fs=1&tf=1&to=billy34@hotmail.com) loyaltyPoints: 36

**Default Filter and Filters**

A powerful feature of jCSV is the ability to filter and transform data cells. In the next example we define a default\_filter for every cell in the dataset: 'default\_filter: Jcsv.not\_nil'. With this filter, no cell can be empty (nil). Looking back at our data, we can see that row number 2 has an empty field for 'married' and 'number of kids". So, we should expect this reading to fail.

> parser = Jcsv.reader("../data/customer.csv", default\_filter: Jcsv.not\_nil)

+ parser.read

Value is nil

As we can see, this parsing dies on record number 2 with a Constraint violation, since we have a default filter of not\_nil and in this record two fields are empty. In order to properly handle this issue, we will add filters to our parser. First we make :numberofkids and :married as optional; however, if this field is filled, then :numberofkids should be and integer and :married should be a boolean. In order to add this filters, we chain them with: Jcsv.optional >> Jcsv.int and Jcsv.optional >> Jcsv.bool:

parser = Jcsv.reader("../data/customer.csv", default\_filter: Jcsv.not\_nil)

# Add filters, so that we get 'objects' instead of strings for filtered fields

parser.filters = {:number\_of\_kids => Jcsv.optional >> Jcsv.int,

:married => Jcsv.optional >> Jcsv.bool,

:customer\_no => Jcsv.int,

:birth\_date => Jcsv.date("dd/MM/yyyy")}

content = parser.read

Let's take a look at the second record. We can see that :customer\_no is an integer, in this case 2, since we have selected the second record, :married is now true and the number of kids is the integer 0.

> p content[1]

[2, "Bob", "Down", #<DateTime: 1919-02-25T00:00:00+00:00 ((2422015j,0s,0n),+0s,2299161j)>, "1601 Willow Rd.\nMenlo Park, CA 94025\nUnited States", true, 0, "\"Frankly, my dear, I don't give a damn.\" - Gone With The Wind", "bobdown@hotmail.com", "123456"]

**Chunking**

As with smarter\_csv, jCSV also supports data chunking, by passing as argument the chunk\_size.

# Read chunks of the file. In this case, we are breaking the file in chunks of 2

reader = Jcsv.reader("../data/customer.csv", chunk\_size: 2)

# Add filters, so that we get 'objects' instead of strings for filtered fields

reader.filters = {:number\_of\_kids => Jcsv.optional >> Jcsv.int,

:married => Jcsv.optional >> Jcsv.bool,

:customer\_no => Jcsv.int}

content = reader.read

> pp content[0]

[[1,

"John",

"Dunbar",

"13/06/1945",

"1600 Amphitheatre Parkway\nMountain View, CA 94043\nUnited States",

nil,

nil,

"\"May the Force be with you.\" - Star Wars",

"jdunbar@gmail.com",

"0"],

[2,

"Bob",

"Down",

"25/02/1919",

"1601 Willow Rd.\nMenlo Park, CA 94025\nUnited States",

true,

0,

"\"Frankly, my dear, I don't give a damn.\" - Gone With The Wind",

"bobdown@hotmail.com",

"123456"]]

Note now that content[0] is an array of array. The external array is of size 2, the chunk size and each sub array is a row. Note also, that the filters have properly converted strings into objects. Remember that method read can receive a block.

Let's show an example with chunk\_size of 3. In this example, the first chunk will be of size 3 and the second of size 1, since there are no more records after that. When reading chunks, in the given blocks, 'line\_no' and 'row\_no' are the last line and row read in the chunk, respectively.

# Read chunks of the file. In this case, we are breaking the file in chunks of 2

reader = Jcsv.reader("../data/customer.csv", chunk\_size: 3)

# Add filters, so that we get 'objects' instead of strings for filtered fields

reader.filters = {:number\_of\_kids => Jcsv.optional >> Jcsv.int,

:married => Jcsv.optional >> Jcsv.bool,

:customer\_no => Jcsv.int}

> reader.read do |line\_no, row\_no, chunk, headers|

+ puts "line number: #{line\_no}, row number: #{row\_no}"

+ pp chunk

+ puts

+ end

line number: 10, row number: 4

[[1,

"John",

"Dunbar",

"13/06/1945",

"1600 Amphitheatre Parkway\nMountain View, CA 94043\nUnited States",

nil,

nil,

"\"May the Force be with you.\" - Star Wars",

"jdunbar@gmail.com",

"0"],

[2,

"Bob",

"Down",

"25/02/1919",

"1601 Willow Rd.\nMenlo Park, CA 94025\nUnited States",

true,

0,

"\"Frankly, my dear, I don't give a damn.\" - Gone With The Wind",

"bobdown@hotmail.com",

"123456"],

[3,

"Alice",

"Wunderland",

"08/08/1985",

"One Microsoft Way\nRedmond, WA 98052-6399\nUnited States",

true,

0,

"\"Play it, Sam. Play \"As Time Goes By.\"\" - Casablanca",

"throughthelookingglass@yahoo.com",

"2255887799"]]

line number: 13, row number: 5

[[4,

"Bill",

"Jobs",

"10/07/1973",

"2701 San Tomas Expressway\nSanta Clara, CA 95050\nUnited States",

true,

3,

"\"You've got to ask yourself one question: \"Do I feel lucky?\" Well, do ya, punk?\" - Dirty Harry",

"billy34@hotmail.com",

"36"]]

**Chunks as Enumerators**

Ruby has a very interesting feature called Enumerator. jCSV supports the use of enumerators, allowing for partial file read. Let's first give an example of using enumerators, and then we will show an example of partially reading a CSV file. In order to get an enumerator on the reader we call method each without any blocks:

reader = Jcsv.reader("../data/customer.csv", chunk\_size: 2)

# Add filters, so that we get 'objects' instead of strings for filtered fields

reader.filters = {:number\_of\_kids => Jcsv.optional >> Jcsv.int,

:married => Jcsv.optional >> Jcsv.bool,

:customer\_no => Jcsv.int}

# Method each without a block returns an enumerator

enum = reader.each

# read the first chunk. Chunk is of size 2

chunk = enum.next

The 'chunk' variable above is an array with the following elements: line\_no, row\_no, chunk data, and headers. In this example, at this point we have read only two records.

> pp chunk

[7,

3,

[[1,

"John",

"Dunbar",

"13/06/1945",

"1600 Amphitheatre Parkway\nMountain View, CA 94043\nUnited States",

nil,

nil,

"\"May the Force be with you.\" - Star Wars",

"jdunbar@gmail.com",

"0"],

[2,

"Bob",

"Down",

"25/02/1919",

"1601 Willow Rd.\nMenlo Park, CA 94025\nUnited States",

true,

0,

"\"Frankly, my dear, I don't give a damn.\" - Gone With The Wind",

"bobdown@hotmail.com",

"123456"]],

[:customer\_no,

:first\_name,

:last\_name,

:birth\_date,

:mailing\_address,

:married,

:number\_of\_kids,

:favourite\_quote,

:email,

:loyalty\_points]]

In order to read the other two records we need to call method 'next' again. Here we are using ‘p’ to print instead of ‘pp’ to save some trees:

> p enum.next

[13, 5, [[3, "Alice", "Wunderland", "08/08/1985", "One Microsoft Way\nRedmond, WA 98052-6399\nUnited States", true, 0, "\"Play it, Sam. Play \"As Time Goes By.\"\" - Casablanca", "throughthelookingglass@yahoo.com", "2255887799"], [4, "Bill", "Jobs", "10/07/1973", "2701 San Tomas Expressway\nSanta Clara, CA 95050\nUnited States", true, 3, "\"You've got to ask yourself one question: \"Do I feel lucky?\" Well, do ya, punk?\" - Dirty Harry", "billy34@hotmail.com", "36"]], [:customer\_no, :first\_name, :last\_name, :birth\_date, :mailing\_address, :married, :number\_of\_kids, :favourite\_quote, :email, :loyalty\_points]]

We now write a small script that will look for a record that has "Bob" on the :firstname. When this happens, the script terminates and no more reading needs to be done. For large CSV files, breaking reading when the required data is read is a very useful feature. Remember that row is an array with line\_no, row\_no, row data, and headers. So, to get :first\_name we need to read row[2][1].

reader = Jcsv.reader("../data/customer.csv")

# Add filters, so that we get 'objects' instead of strings for filtered fields

reader.filters = {:number\_of\_kids => Jcsv.optional >> Jcsv.int,

:married => Jcsv.optional >> Jcsv.bool,

:customer\_no => Jcsv.int}

# Method each without a block returns an enumerator

enum = reader.each

begin

row = enum.next

end while row[2][1] != "Bob"

p row

[7, 3, [2, "Bob", "Down", "25/02/1919", "1601 Willow Rd.\nMenlo Park, CA 94025\nUnited States", true, 0, "\"Frankly, my dear, I don't give a damn.\" - Gone With The Wind", "bobdown@hotmail.com", "123456"], [:customer\_no, :first\_name, :last\_name, :birth\_date, :mailing\_address, :married, :number\_of\_kids, :favourite\_quote, :email, :loyalty\_points]]

**Skipping Columns**

Sometimes, a CSV file contains columns that are of no interest, and thus, reading them just consumes memory without any benefit. jCSV allows skipping such columns, by defining a mapping. Bellow an example where the columns :customer\_no, :mailing\_address and :favourite\_quote are not read:

reader = Jcsv.reader("../data/customer.csv")

# Add mapping. When column is mapped to false, it will not be retrieved from the

# file, improving time and speed efficiency

reader.mapping = {:customerno => false, :mailingaddress => false, :favouritequote => false}

Note that the headers in the block do not show any of the removed columns, although the reader.headers still has all headers.

> p reader.headers

[:customer\_no, :first\_name, :last\_name, :birth\_date, :mailing\_address, :married, :number\_of\_kids, :favourite\_quote, :email, :loyalty\_points]

> reader.read do |line\_no, row\_no, row, headers|

+ p headers

+ p row

+ end

[:customer\_no, :first\_name, :last\_name, :birth\_date, :mailing\_address, :married, :number\_of\_kids, :favourite\_quote, :email, :loyalty\_points]

["1", "John", "Dunbar", "13/06/1945", "1600 Amphitheatre Parkway\nMountain View, CA 94043\nUnited States", nil, nil, "\"May the Force be with you.\" - Star Wars", "jdunbar@gmail.com", "0"]

[:customer\_no, :first\_name, :last\_name, :birth\_date, :mailing\_address, :married, :number\_of\_kids, :favourite\_quote, :email, :loyalty\_points]

["2", "Bob", "Down", "25/02/1919", "1601 Willow Rd.\nMenlo Park, CA 94025\nUnited States", "Y", "0", "\"Frankly, my dear, I don't give a damn.\" - Gone With The Wind", "bobdown@hotmail.com", "123456"]

[:customer\_no, :first\_name, :last\_name, :birth\_date, :mailing\_address, :married, :number\_of\_kids, :favourite\_quote, :email, :loyalty\_points]

["3", "Alice", "Wunderland", "08/08/1985", "One Microsoft Way\nRedmond, WA 98052-6399\nUnited States", "Y", "0", "\"Play it, Sam. Play \"As Time Goes By.\"\" - Casablanca", "throughthelookingglass@yahoo.com", "2255887799"]

[:customer\_no, :first\_name, :last\_name, :birth\_date, :mailing\_address, :married, :number\_of\_kids, :favourite\_quote, :email, :loyalty\_points]

["4", "Bill", "Jobs", "10/07/1973", "2701 San Tomas Expressway\nSanta Clara, CA 95050\nUnited States", "Y", "3", "\"You've got to ask yourself one question: \"Do I feel lucky?\" Well, do ya, punk?\" - Dirty Harry", "billy34@hotmail.com", "36"]

**Column Reordering**

jCSV also allows for reordering the columns of the CSV file. This is also done through a mapping:

reader = Jcsv.reader("../data/customer.csv")

# Add filters...

reader.filters = {:number\_of\_kids => Jcsv.optional >> Jcsv.int,

:married => Jcsv.optional >> Jcsv.bool,

:customer\_no => Jcsv.int}

# Mapping allows reordering of columns. In this example, column 0 (:customerno)

# in the csv file will be loaded in position 2 (3rd column); column 1 (:firstname)

# in the csv file will be loaded in position 0 (1st column); column 2 on the csv file

# will not be loaded (false); column 4 (:birthdate) will be loaded on position 3,

# and so on.

# When reordering columns, care should be taken to get the mapping right or unexpected

# behaviour could result.

reader.mapping = {:customer\_no => 2, :first\_name => 0, :last\_name => false,

:birth\_date => 3, :mailing\_address => false, :married => false,

:number\_of\_kids => false, :favourite\_quote => false, :email => 1,

:loyalty\_points => 4}

> reader.read do |line\_no, row\_no, row, headers|

+ p headers

+ p row

+ end

[:first\_name, :email, :customer\_no, :birth\_date, :loyalty\_points]

["John", "jdunbar@gmail.com", 1, "13/06/1945", "0"]

[:first\_name, :email, :customer\_no, :birth\_date, :loyalty\_points]

["Bob", "bobdown@hotmail.com", 2, "25/02/1919", "123456"]

[:first\_name, :email, :customer\_no, :birth\_date, :loyalty\_points]

["Alice", "throughthelookingglass@yahoo.com", 3, "08/08/1985", "2255887799"]

[:first\_name, :email, :customer\_no, :birth\_date, :loyalty\_points]

["Bill", "billy34@hotmail.com", 4, "10/07/1973", "36"]

**Read to Map**

In this section we show how to read data into an array of maps (hashes) instead as into an array of arrays. Reading to map is very easy and only requires passing one argument: 'format: :map'.

reader = Jcsv.reader("../data/customer.csv", format: :map)

# map is an array of hashes

map = reader.read

In order to get the :loyalty\_points for the second customer we do:

> p map[1][:loyalty\_points]

"123456"

Reading to maps support most of the same arguments as reading to lists. Bellow we read with chunk\_size 2, and strings as key:

reader = Jcsv.reader("../data/customer.csv", format: :map, chunk\_size: 2,

strings\_as\_keys: true)

map = reader.read

With chunk\_size 2, we have in variable map two chunks, each of size 2. Let's take a look at the first element of the second chunk, i.e., the third row of data:

> pp map[1][0]

{"customerNo"=>"3",

"firstName"=>"Alice",

"lastName"=>"Wunderland",

"birthDate"=>"08/08/1985",

"mailingAddress"=>"One Microsoft Way\nRedmond, WA 98052-6399\nUnited States",

"married"=>"Y",

"numberOfKids"=>"0",

"favouriteQuote"=>"\"Play it, Sam. Play \"As Time Goes By.\"\" - Casablanca",

"email"=>"throughthelookingglass@yahoo.com",

"loyaltyPoints"=>"2255887799"}

Filters and mappings are also supported for maps. Note that we introduce some new filters: Jcsv.long and Jcsv.date. We also show how to rename columns by using a mapping, for instance, we want column :number\_of\_kids to be mapped to :numero\_criancas which is the same label but in Portuguese. Note also that we map :loyalty\_points to the string with white spaces "pontos fidelidade". Finally, columns :customer\_no, :mailing\_address and :favourite\_quote are also droped.

# type is :map. Rows are hashes. Set the default filter to not\_nil. That is, all

# fields are required unless explicitly set to optional.

reader = Jcsv.reader("../data/customer.csv", format: :map, default\_filter: Jcsv.not\_nil)

# Set numberOfKids and married as optional, otherwise an exception will be raised

reader.filters = {:number\_of\_kids => Jcsv.optional >> Jcsv.int,

:married => Jcsv.optional >> Jcsv.bool,

:loyalty\_points => Jcsv.long,

:customerno => Jcsv.int,

:birth\_date => Jcsv.date("dd/MM/yyyy")}

# When parsing to map, it is possible to make a mapping. If column name is :false

# the column will be removed from the returned row

reader.mapping = {:number\_of\_kids => :numero\_criancas,

:married => "casado",

:loyalty\_points => "pontos fidelidade",

:customer\_no => false,

:mailing\_address => false,

:favourite\_quote => false}

reader.read do |line\_no, row\_no, row|

pp row

end

{:first\_name=>"John",

:last\_name=>"Dunbar",

:birth\_date=>

#<DateTime: 1945-06-13T00:00:00+00:00 ((2431620j,0s,0n),+0s,2299161j)>,

"casado"=>nil,

:numero\_criancas=>nil,

:email=>"jdunbar@gmail.com",

"pontos fidelidade"=>0}

{:first\_name=>"Bob",

:last\_name=>"Down",

:birth\_date=>

#<DateTime: 1919-02-25T00:00:00+00:00 ((2422015j,0s,0n),+0s,2299161j)>,

"casado"=>true,

:numero\_criancas=>0,

:email=>"bobdown@hotmail.com",

"pontos fidelidade"=>123456}

{:first\_name=>"Alice",

:last\_name=>"Wunderland",

:birth\_date=>

#<DateTime: 1985-08-08T00:00:00+00:00 ((2446286j,0s,0n),+0s,2299161j)>,

"casado"=>true,

:numero\_criancas=>0,

:email=>"throughthelookingglass@yahoo.com",

"pontos fidelidade"=>2255887799}

{:first\_name=>"Bill",

:last\_name=>"Jobs",

:birth\_date=>

#<DateTime: 1973-07-10T00:00:00+00:00 ((2441874j,0s,0n),+0s,2299161j)>,

"casado"=>true,

:numero\_criancas=>3,

:email=>"billy34@hotmail.com",

"pontos fidelidade"=>36}

Reading as map also supports reading as enumerator:

reader = Jcsv.reader("../data/customer.csv", chunk\_size: 2, format: :map)

# Add filters, so that we get 'objects' instead of strings for filtered fields

reader.filters = {"numberOfKids" => Jcsv.optional >> Jcsv.int,

"married" => Jcsv.optional >> Jcsv.bool,

"customerNo" => Jcsv.int}

enum = reader.each

chunk = enum.next

> pp chunk[2][1]

{:customer\_no=>"2",

:first\_name=>"Bob",

:last\_name=>"Down",

:birth\_date=>"25/02/1919",

:mailing\_address=>"1601 Willow Rd.\nMenlo Park, CA 94025\nUnited States",

:married=>true,

:number\_of\_kids=>"0",

:favourite\_quote=>

"\"Frankly, my dear, I don't give a damn.\" - Gone With The Wind",

:email=>"bobdown@hotmail.com",

:loyalty\_points=>"123456"}

**Dimensions**

From Wikipedia:

"A dimension is a structure that categorizes facts and measures in order to enable users to answer business questions. Commonly used dimensions are people, products, place and time.

In a data warehouse, dimensions provide structured labeling information to otherwise unordered numeric measures. The dimension is a data set composed of individual, non-overlapping data elements. The primary functions of dimensions are threefold: to provide filtering, grouping and labelling."

Data often has dimensions, but they are just treated as labels for the data in a column of the CSV file.

The following excerpt shows data from an experiment in which patients with epilepsy were given either a placebo or Progabide to check the effect of this medicament in their seizure rate during a four week treatment period (data from R).

Clearly, treatment is a dimension in this data, as a patient is either given a placebo or Progabide. The patient id (first column) can also be considered a dimension. In this experiment there were 59 patients, during a 4 week period, thus this dataset has 236 rows.

"patient","treatment","base","age","seizure.rate","period","subject"

"1","placebo",11,31,5,"1","1"

"110","placebo",11,31,3,"2","1"

"112","placebo",11,31,3,"3","1"

"114","placebo",11,31,3,"4","1"

"29","Progabide",76,18,11,"1","29"

"291","Progabide",76,18,14,"2","29"

"292","Progabide",76,18,9,"3","29"

"293","Progabide",76,18,8,"4","29"

"30","Progabide",38,32,8,"1","30"

Let's now read this dataset and see how dimensions can help understand this data and organize it. Four dimensions are set for this dataset: patient, subject, treatment and period and it will be read as an array of maps:

reader = Jcsv.reader("../data/epilepsy.csv", format: :map,

dimensions: [:patient, :subject, :treatment, :period])

treatment = reader.read

> treatment.first(5).each do |row|

+ pp row

+ end

["1.1.placebo.1", {:base=>"11", :age=>"31", :"seizure.rate"=>"5"}]

["110.1.placebo.2", {:base=>"11", :age=>"31", :"seizure.rate"=>"3"}]

["112.1.placebo.3", {:base=>"11", :age=>"31", :"seizure.rate"=>"3"}]

["114.1.placebo.4", {:base=>"11", :age=>"31", :"seizure.rate"=>"3"}]

["2.2.placebo.1", {:base=>"11", :age=>"30", :"seizure.rate"=>"3"}]

Observe that the :key hashkey has an array with all the element's dimensions and the other columns are still processed as in a regular map reader. At this point it might not be clear what the benefit of dimensions are, but let's move a little bit forward.

**Deep Map**

In the next example we will add some new directives to the reader: chunk\_size :all and deep\_map true. The patient field will be ignored, since this is redundant information that can be obtained from the subject:

reader = Jcsv.reader("../data/epilepsy.csv", format: :map, chunk\_size: :all,

dimensions: [:treatment, :subject, :period], deep\_map: true)

# remove the :patient field from the data, as this field is already given by the

# :subject field.

reader.mapping = {:patient => false}

treatment = reader.read[0]

First let's understand the directive chunk\_size :all: this indicates that the file should be read in one large chunk. When reading chunks, each chunk is an array, of arrays of the given chunk size. When reading chunk\_size all, the returned data is in an array that has an array with all row, this is why we have treatment above to be reader.read[0].

The attentive reader might ask: "why do we need chunck\_size :all, since when no chunk size is given the whole file is read anyway?". This has to do with the directive deep\_map true. When deep\_map is used, the first dimension's elements are keys to the second dimensions element's and so on. In order to be able to make deep maps chunks of data need to be read.

The treatment variable above is a hash that has two entries: 'placebo' and 'Progabide'. The placebo entry has as entries the elements from the second dimension, which is subject. So treatment["placebo"]["1"] shows the data for all four periods os treatment for subject "1".

> pp treatment["placebo"]["1"]

{"1"=>{:base=>"11", :age=>"31", :"seizure.rate"=>"5"},

"2"=>{:base=>"11", :age=>"31", :"seizure.rate"=>"3"},

"3"=>{:base=>"11", :age=>"31", :"seizure.rate"=>"3"},

"4"=>{:base=>"11", :age=>"31", :"seizure.rate"=>"3"}}

As we can see, on the first period, subject "1" had base = "11", age = "31", seizure.rate = "5". On the second period it's seizure.rate was "3". We can get this from our treatment variable with:

> p treatment["placebo"]["1"]["2"][:"seizure.rate"]

"3"

Let's take a look at another entry: a subject 38 that uses Progabide. One aspect to be noted in the subject's dimensions on is that it would be better for subjects to be numbered 1, 2, 3, etc. for treatment with 'placebo' and also, 1, 2, 3, etc. for treatment with 'Progabide'. If this were the case then the first patient treated with placebo would be accessed with 'treatment["placebo"]["1"]' and the first patient treated with Progabide would be accessed with 'treatment["Progabide"]["1"]'. In the current forms, we need to know that patient "38" was treated with Progabide.

> pp treatment["Progabide"]["38"]

{"1"=>{:base=>"67", :age=>"20", :"seizure.rate"=>"3"},

"2"=>{:base=>"67", :age=>"20", :"seizure.rate"=>"7"},

"3"=>{:base=>"67", :age=>"20", :"seizure.rate"=>"7"},

"4"=>{:base=>"67", :age=>"20", :"seizure.rate"=>"7"}}

Dimensions' elements can be accessed by accessing reader's 'dimensions' instance variable and getting the labels:

> pp reader.dimensions[:treatment].labels

+ pp reader.dimensions[:period].labels

{"placebo"=>0, "Progabide"=>1}

{"1"=>0, "2"=>1, "3"=>2, "4"=>3}

**Dimensions Ordering**

Dimensions should be read from slowest to fast changing in the file. The example bellow shows a CSV file and the proper way of organizing dimensions:

# File GoodOrder.csv

Dim 1 Dim 2 Dim 3 Data

A X K 1

A X J 2

A Y H 3

A Y G 4

B X K 5

B X J 6

B Y H 7

B Y G 8

C X K 9

C X J 10

C Y H 11

C Y G 12

reader = Jcsv.reader("../data/GoodOrder.csv", format: :map, chunk\_size: :all, col\_sep: ";",

dimensions: [:dim\_1, :dim\_2, :dim\_3], deep\_map: true)

table = reader.read[0]

> pp table

{"A"=>

{"X"=>{"K"=>{:data=>"1"}, "J"=>{:data=>"2"}},

"Y"=>{"H"=>{:data=>"3"}, "G"=>{:data=>"4"}}},

"B"=>

{"X"=>{"K"=>{:data=>"5"}, "J"=>{:data=>"6"}},

"Y"=>{"H"=>{:data=>"7"}, "G"=>{:data=>"8"}}},

"C"=>

{"X"=>{"K"=>{:data=>"9"}, "J"=>{:data=>"10"}},

"Y"=>{"H"=>{:data=>"11"}, "G"=>{:data=>"12"}}}}

Changing the order of reading dimensions will generate errors:

reader = Jcsv.reader("../data/GoodOrder.csv", format: :map, chunk\_size: :all, col\_sep: ";",

dimensions: [:dim\_2, :dim\_1, :dim\_3], deep\_map: true)

table = reader.read[0]

Warning reading row: [C, X, K, 9] in field 'dim\_1'. Dimension 'dim\_1' is frozen. Cannot add label 'C'.

Warning reading row: [C, X, J, 10] in field 'dim\_1'. Dimension 'dim\_1' is frozen. Cannot add label 'C'.

Warning reading row: [C, Y, H, 11] in field 'dim\_1'. Dimension 'dim\_1' is frozen. Cannot add label 'C'.

Warning reading row: [C, Y, G, 12] in field 'dim\_1'. Dimension 'dim\_1' is frozen. Cannot add label 'C'.

In this example we get a message saying that dimension 'dim\_1' is frozen. What does that mean? As we explained above, the CSV reader expects slower changing dimensions to be read first. The proper order of reading dimensions is thus, dim\_1, dim\_2, dim\_3. A dimension becomes frozen whenever it cycles back to its first element. In this example, dim\_2 is X, X, Y, Y when it cycles back to X it becomes frozen indicating that X and Y are the only two elements in this dimension. When a dimension if frozen all dimensions after it are also frozen. In this case, dim\_1 and dim\_3 also become frozen.

When dim\_2 cycles back to X the values of dim\_1 that were read are A and B. When it becomes frozen, no other element can be added to this dimension. When label C is read form dim\_1, it generates the error saying that label C cannot be added to dim\_1.

We will now read the file bellow, BadOrder.csv. It contains the same data as above but dimension dim\_2 is the first column:

# File BadOrder.csv

Dim\_2 Dim\_1 Dim\_3 Data

X A K 1

X A J 2

Y A H 3

Y A G 4

X B K 5

X B J 6

Y B H 7

Y B G 8

X C K 9

X C J 10

Y C H 11

Y C G 12

reader = Jcsv.reader("../data/BadOrder.csv", format: :map, chunk\_size: :all, col\_sep: ";",

dimensions: [:dim\_2, :dim\_1, :dim\_3], deep\_map: true)

table = reader.read[0]

Warning reading row: [X, B, K, 5] in field 'dim\_1'. Dimension 'dim\_1' is frozen. Cannot add label 'B'.

Warning reading row: [X, B, J, 6] in field 'dim\_1'. Dimension 'dim\_1' is frozen. Cannot add label 'B'.

Warning reading row: [Y, B, H, 7] in field 'dim\_1'. Dimension 'dim\_1' is frozen. Cannot add label 'B'.

Warning reading row: [Y, B, G, 8] in field 'dim\_1'. Dimension 'dim\_1' is frozen. Cannot add label 'B'.

Warning reading row: [X, C, K, 9] in field 'dim\_1'. Dimension 'dim\_1' is frozen. Cannot add label 'C'.

Warning reading row: [X, C, J, 10] in field 'dim\_1'. Dimension 'dim\_1' is frozen. Cannot add label 'C'.

Warning reading row: [Y, C, H, 11] in field 'dim\_1'. Dimension 'dim\_1' is frozen. Cannot add label 'C'.

Warning reading row: [Y, C, G, 12] in field 'dim\_1'. Dimension 'dim\_1' is frozen. Cannot add label 'C'.

Note now that the error happens earlier in the file. Again, as we read dim\_2 we get X, X, Y, Y. When we cycle back to X, dim\_2 is frozen, freezing dim\_1 and dim\_3 in the sequence. Now when the first B is read, dim\_1 is already frozen and an error is issued.

Even though an error is issued, reading continues normally and the table can be printed:

> pp table

{"X"=>

{"A"=>{"K"=>{:data=>"1"}, "J"=>{:data=>"2"}},

"B"=>{"K"=>{:data=>"5"}, "J"=>{:data=>"6"}},

"C"=>{"K"=>{:data=>"9"}, "J"=>{:data=>"10"}}},

"Y"=>

{"A"=>{"H"=>{:data=>"3"}, "G"=>{:data=>"4"}},

"B"=>{"H"=>{:data=>"7"}, "G"=>{:data=>"8"}},

"C"=>{"H"=>{:data=>"11"}, "G"=>{:data=>"12"}}}}

If reading continues normally, why is an error issued? For large datasets, when data is organized with the slowest changing dimension first, it becomes easier to identify missing or duplicated data. It is also a necessary condition for reading data into a vector, as we will show in the next section ("Reading into a Vector").

We now show a data file in which there is missing data. Note that we removed the fourth line from the file, and note also that this is not easily identified. In a larger dataset, seeing this would be very hard.

# File missing\_data.csv

Dim\_1 Dim\_2 Dim\_3 Data

A X K 1

A X J 2

A Y H 3

B X K 5

B X J 6

B Y H 7

B Y G 8

C X K 9

C X J 10

C Y H 11

C Y G 12

reader = Jcsv.reader("../data/missing\_data.csv", format: :map, chunk\_size: :all,

col\_sep: ";", dimensions: [:dim\_1, :dim\_2, :dim\_3],

deep\_map: true)

table = reader.read[0]

Warning reading row: [B, Y, G, 8] in field 'dim\_3'. Dimension 'dim\_3' is frozen. Cannot add label 'G'.

Warning reading row: [C, Y, G, 12] in field 'dim\_3'. Dimension 'dim\_3' is frozen. Cannot add label 'G'.

Again, we get an error with dimension frozen. This happens when reading the 8th row. Dimension 3 was frozen after reading element K, since this dimension cycled from H back to K. When reaching the 8th row a new element G is seen and indicates that something is wrong in the file.

Let's again take a look at what was read:

> pp table

{"A"=>{"X"=>{"K"=>{:data=>"1"}, "J"=>{:data=>"2"}}, "Y"=>{"H"=>{:data=>"3"}}},

"B"=>

{"X"=>{"K"=>{:data=>"5"}, "J"=>{:data=>"6"}},

"Y"=>{"H"=>{:data=>"7"}, "G"=>{:data=>"8"}}},

"C"=>

{"X"=>{"K"=>{:data=>"9"}, "J"=>{:data=>"10"}},

"Y"=>{"H"=>{:data=>"11"}, "G"=>{:data=>"12"}}}}

Bellow we show another CSV file. Note that there is a missing row in this data. Can you quickly see it? Which row is it?

# File missing\_data2.csv

Dim\_1 Dim\_2 Dim\_3 Data

A X K 1

A X J 2

A Y H 3

A Y G 4

A Z F 5

A Z D 6

B X K 7

B X J 8

B Y H 9

B Z F 10

B Z D 11

reader = Jcsv.reader("../data/missing\_data2.csv", format: :map, chunk\_size: :all,

col\_sep: ";", dimensions: [:dim\_1, :dim\_2, :dim\_3],

deep\_map: true)

table = reader.read[0]

Warning reading row: [B, Z, F, 10] in field 'dim\_3'. Missing data: next expected label was 'G' but read 'F'.

This last example shows how dimensions can help identify duplicate data. A set of dimensions should be unique, as a key in a database. If the key is duplicate, an error is issued. The same missing\_data2.csv file is read, but passing only two dimensions, dim\_1 and dim2:

reader = Jcsv.reader("../data/missing\_data2.csv", format: :map,

chunk\_size: :all, col\_sep: ";",

dimensions: [:dim\_1, :dim\_2], deep\_map: true)

table = reader.read[0]

Key not unique for this dataset. {:dim\_3=>"J", :data=>"2"}

**Hidding Errors**

Since errors are shown but data is still read, if the user knows she doesn't want to be notified of errors, she could add the suppress\_warnings directive:

reader = Jcsv.reader("../data/missing\_data2.csv", format: :map, chunk\_size: :all,

col\_sep: ";", dimensions: [:dim\_1, :dim\_2, :dim\_3],

deep\_map: true, suppress\_warnings: true)

table = reader.read[0]

As can be seen, the code above does not generate any errors.

**Dimensions to Lists**

Reading data with dimensions to lists is also possible, and will generate arrays of arrays:

reader = Jcsv.reader("../data/GoodOrder.csv", chunk\_size: :all, col\_sep: ";",

dimensions: [:dim\_1, :dim\_2, :dim\_3])

table = reader.read

> pp table

[[[["A", "X", "K"], "1"],

[["A", "X", "J"], "2"],

[["A", "Y", "H"], "3"],

[["A", "Y", "G"], "4"],

[["B", "X", "K"], "5"],

[["B", "X", "J"], "6"],

[["B", "Y", "H"], "7"],

[["B", "Y", "G"], "8"],

[["C", "X", "K"], "9"],

[["C", "X", "J"], "10"],

[["C", "Y", "H"], "11"],

[["C", "Y", "G"], "12"]]]

This text was Markup formated with CodeWriter:

* gem install CodeWriter
* jruby -S gem install CodeWriter
* <https://github.com/rbotafogo/CodeWriter>