© iStockphoto.com/Tuned_In

6

DATA MUNGING

LEARNING OBJECTIVES

- Describe what data munging is.
- Demonstrate how to read a CSV data file.
- Explain how to select, remove, and rename rows and columns.
- Assess why data scientists need to be able to munge data.
- Demonstrate how to munge data in R while using the following functions: read.csv, url, gsub, rownames, colnames, order.

Data munging is the process of turning a data set with a bunch of junk in it into a nice clean data set. Why is data munging required and why is it important? Well, often R does not guess correctly the structure of the data set, or perhaps R reads a number or a date and thinks it is a simple string. Another issue might be that the data file might have additional information that is useful for humans, but not for R. If you think about it, so far we have only explored simple data sets that we created within R. Clearly, the larger the data set, the more difficult it becomes to just type the data into R. Working through these issues, so that R can process the data in a dataframe is often a lot of work. It's a big part of data science, but perhaps not the most glamorous.

READING A CSV TEXT FILE

So, in this chapter, we will explore how to read in a data set that is stored as a comma-delimited text file (known as a CSV file—which stands for comma separated values) that needs to be cleaned up. As we will see in future chapters, there are many formats that we might have to be able to process to get data into R, but for now we will focus on a very common human readable file format. Our first real data set will be U.S. census data. The U.S. Census Bureau has stored population data in many locations on its website, with many interesting data sets to explore. We will use one of the simpler data sets available at www2.census.gov/programs-surveys/popest/tables/2010-2011/state/totals/

Click on the CSV link for nst-est2011-01.csv; you will either download a CSV (comma separated value file) or your browser will show a bunch of text information, with the first few lines likes like:

```
table with row headers in column A and column headers in rows
3 through 4. (leading dots indicate sub-parts),,,,,,,,,
"Table 1. Annual Estimates of the Population for the
United States, Regions, States, and Puerto Rico: April 1,
2010 to July 1, 2011",,,,,,,,, Geographic Area,"April 1,
2010",,Population Estimates (as of July 1),,,,,,,
Census,Estimates Base,2010,2011,,,,,United States,"308,
745,538","308,745,538","309,330,219","311,591,
917",,,,,Northeast,"55,317,240","55,317,244","55,366,
108","55,521,598",,,,,  Midwest,"66,927,001","66,926,
987","66,976,458","67,158,835",,,,,   South,"114,555,
744","114,555,757","114,857,529","116,046,736",,,,,
West,"71,945,553","71,945,550","72,130,124","72,864,
748",,,,,,.Alabama,"4,779,736","4,779,735","4,785,401",
"4,802,740",,,,,
```

Now, having the data in the browser isn't useful, so let's write some R code to read in this data set.

```
> urlToRead <-
+ "http://www2.census.gov/programs-surveys/
+    popest/tables/2010-2011/state/totals/
+    nst-est2011-01.csv"
> testFrame <- read.csv(url(urlToRead))
```

The first line of code just defines the location (on the web) of the file to load (note that the URL is so long, it actually takes four lines to define the assignment). As we noted before, since the CSV file is human readable, you can actually cut and paste the URL into a web browser, and the page will show up as a list of rows of data. The next row of code reads the file, using the read.csv command. Note we also use the url() function so R knows that the filename is a URL (as opposed to a local file on the computer).

Next, let's take a look at what we got back. We can use the str() function to create a summary of the structure of testFrame:

```
> str(testFrame)
'data.frame':    66 obs. of 10 variables:
$ table.with.row.headers.in.column.A.and.column.
headers.in.rows.3.through.4...leading.dots.indicate.sub.
parts.: Factor w/ 65 levels "",".Alabama",..: 62 53 1
64 55 54 60 65 2 3 ...
$ X: Factor w/ 60 levels "","1,052,567",..: 1 59 60
27 38 47 10 49 32 50 ...
$ X.1: Factor w/ 59 levels "","1,052,567",..: 1 1 59
27 38 47 10 49 32 50 ...
$ X.2: Factor w/ 60 levels "","1,052,528",..: 1 60 21
28 39 48 10 51 33 50 ...
$ X.3: Factor w/ 59 levels "","1,051,302",..: 1 1 21
28 38 48 10 50 33 51 ...
$ X.4: logi NA NA NA NA NA NA ...
$ X.5: logi NA NA NA NA NA NA ...
$ X.6: logi NA NA NA NA NA NA ...
$ X.7: logi NA NA NA NA NA NA ...
$ X.8: logi NA NA NA NA NA NA ...
```

The last few lines are reminiscent of that late 1960s song entitled, "Na Na Hey Hey Kiss Him Goodbye." Setting aside all the NA NA NA NAs, however, the overall structure is 66 observations of 10 variables, signifying that the spreadsheet contained 66 rows and 10 columns of data. The variable names that follow are pretty bizarre. Now you understand what data scientists mean by junk in their data. The first variable name is

table.with.row.headers.in.column.A.and.column.headers.in.rows.3.through.4...
leading.dots.indicate.sub.parts.

REMOVING ROWS AND COLUMNS

What a mess! It is clear that read.csv() treated the upper-left-most cell as a variable label, but was flummoxed by the fact that this was really just a note to human users of the spreadsheet (the variable labels, such as they are, came on lower rows of the spreadsheet). Subsequent variable names include X, X.1, and X.2: clearly the read.csv() function did not have an easy time getting the variable names out of this file.

The other worrisome finding from str() is that all of our data are factors. This indicates that R did not see the incoming data as numbers, but rather as character strings that it interpreted as factor data. Again, this is a side effect of the fact that some of the first cells that read.csv() encountered were text rather than numeric. The numbers came much later in the sheet. Clearly, we have some work to do if we are to make use of these data as numeric population values. This is common for data scientists, in that sometimes the data are available, but need to be cleaned up before they can be used. In fact, data scientists often use the phrase "data munging" as the verb to describe the act of cleaning up data sets. So, let's get data munging!

First, let's review one way to access a list, a vector or a dataframe. As mentioned briefly in a previous chapter, in R, square brackets allow indexing into a list, vector, or dataframe. For example, myList[3] would give us the third element of myList. Keeping in mind that a dataframe is a rectangular structure, really a two-dimensional structure, we can address any element of a dataframe with both a row and column designator: myFrame[4,1] would give the fourth row and the first column. A shorthand for taking the whole column of a dataframe is to leave the row index empty: myFrame[, 6] would give every row in the sixth column. Likewise, a shorthand for taking a whole row of a dataframe is to leave the column index empty: myFrame[10,] would give every column in the tenth row. We can also supply a list of rows instead of just one row, like this: myFrame[c(1,3,5),] would return rows 1, 3, 5 (including the data for all columns, because we left the column index blank).

Using this knowledge, we will use an easy trick to get rid of stuff we don't need. The Census Bureau put in three header rows that we can eliminate like this:

```
> testFrame <- testFrame[-1:-8,]
```

The minus sign used inside the square brackets refers to the index of rows that should be eliminated from the dataframe. So the notation -1:-8 gets rid of the first eight rows. We also leave the column designator empty so that we can keep all columns for now. So the interpretation of the notation within the square brackets is that rows 1 through 8 should be dropped, all other rows should be included, and all columns should be included. We assign the result back to the same data object, thereby replacing the original with our new, smaller, cleaner version.

Next, we can see that of the 10 variables we got from read.csv(), only the first five are useful to us (the last five seem to be blank). How can we know that the last columns are not useful? Well, we can use the summary command we saw last chapter to explore testFrame, but only look at the summary for the last five columns:

```
> summary(testFrame[,6:10])
X.4             X.5             X.6             X.7             X.8
Mode:logical    Mode:logical    Mode:logical    Mode:logical    Mode:logical
NA's:58         NA's:58         NA's:58         NA's:58         NA's:58
```

So, with the summary command, we can see those five columns are all just NA, and so can be removed without removing any data from testFrame. We can use the following command keeps the first five columns of the dataframe:

```
> testFrame <- testFrame[,1:5]
```

In the same vein, the tail() function shows us that the last few rows just contained some Census Bureau notes:

```
> tail(testFrame,5)
```

So we can safely eliminate those like this:

```
> testFrame <- testFrame[-52:-58,]
```

If you're alert you will notice that we could have combined some of these commands, but for the sake of clarity we have done each operation individually. The result is a dataframe with 51 rows and five observations.

RENAMING ROWS AND COLUMNS

Now we are ready to perform a couple of data transformations. But before we start these transformations, let's give our first column a more reasonable name:

```
> testFrame$stateName <- testFrame[,1]
```

We've used a little hack here to avoid typing out the ridiculously long name of that first variable/column. We've used the column notation in the square brackets on the right-hand side of the expression to refer to the first column (the one with the ridiculous name) and simply copied the data into a new column entitled stateName.

Rather than create a new column, we could have renamed the column. So, let's also do this renaming, using the colnames() function. If this function is just called with a dataframe as a parameter, then the function returns the column names in the dataframe, as shown below:

```
> colnames(testFrame)
[1]
"table.with.row.headers.in.column.A.and.column.headers
.in.rows.3.through.4...leading.dots.indicate.sub.parts
."
[2] "X"
[3] "X.1"
[4] "X.2"
[5] "X.3"
[6] "stateName"
```

We also can use colnames() to update the column names in the dataframe. We do this by having the colnames() function on the left side of the assignment statement. Putting this together, we first use colnames() to store the current column names, then update the first element to a new name, and finally use colnames() to update the column names in the dataframe:

```
> cnames <- colnames(testFrame)
> cnames[1] <- "newName"
> cnames
[1] "newName" "X"       "X.1"      "X.2"      "X.3"
"stateName"
> colnames(testFrame) <- cnames
> colnames(testFrame)
[1] "newName" "X"       "X.1"      "X.2"      "X.3"
"stateName"
```

This points out one of the good (and bad) aspects of using R—there is often more than one way to get something done. Sometimes there is a better way, but sometimes just an alternative way. In this situation, for very large data sets, renaming columns would typically be slightly better than creating a new column. In any event, since we have created the new column, let's remove the first column (since we already have the column name we want with the last column in the data set).

```
> testFrame <- testFrame[,-1]
```

CLEANING UP THE ELEMENTS

Next, we can change formats and data types as needed. We can remove the dots from in front of the state names very easily with the gsub() command, which replaces all occurrence of a pattern and returns the new string. The g means replace all (it actually stands for global substitute). There is also a sub function, but we want all the dots to be removed, so we will use the gsub() function.

```
> testFrame$stateName <- gsub("\\.","",
+       testFrame$stateName)
```

The two backslashes in the string expression above are called escape characters and they force the dot that follows to be treated as a literal dot rather than as a wildcard character. The dot on its own is a wildcard that matches one instance of any character.

Next, we can use gsub() and as.numeric() to convert the data contained in the population columns to usable numbers. Remember that those columns are now represented as R factors and what we are doing is taking apart the factor labels (which are basically character strings that look like this: 308,745,538) and making them into numbers. First, let's get rid of the commas.

```
> testFrame$april10census <-gsub(",", "", testFrame$X)
> testFrame$april10base <-gsub(",", "", testFrame$X.1)
> testFrame$july10pop <- gsub(",", "", testFrame$X.2)
> testFrame$july11pop <- gsub(",", "", testFrame$X.3)
```

Next, let's get rid of spaces and convert to a number:

```
> testFrame$april10census <- as.numeric(gsub(" ", "",
+     testFrame$april10census))
> testFrame$april10base <- as.numeric(gsub(" ", "",
+     testFrame$april10base))
> testFrame$july10pop <- as.numeric(gsub(" ", "",
+     testFrame$july10pop))
> testFrame$july11pop <- as.numeric(gsub(" ", "",
+     testFrame$july11pop))
```

This code is flexible in that it will deal with both unwanted commas and spaces, and will convert strings into numbers whether they are integers or not (i.e., possibly with digits after the decimal point).

Finally, let's remove the columns with the X names:

```
> testFrame <- testFrame[,-1:-4]
```

By the way, the choice of variable names for the new columns in the dataframe was based on an examination of the original data set that was imported by read.csv(). We can confirm that the new columns on the dataframe are numeric by using str() to accomplish this.

```
> str(testFrame)
'data.frame':    51 obs. of  5 variables:
 $ stateName      : chr  "Alabama" "Alaska" "Arizona"
"Arkansas" ...
 $ april10census:  num  4779736 710231 6392017 2915918
37253956 ...
 $ april10base   :  num  4779735 710231 6392013 2915921
37253956 ...
 $ july10pop     :  num  4785401 714146 6413158 2921588
37338198 ...
 $ july11pop     :  num  4802740 722718 6482505 2937979
37691912 ...
```

Perfect! Let's take a look at the first five rows

```
> head(testFrame,5)
stateName april10census april10base july10pop july11pop
9      Alabama    4779736    4779735    4785401    4802740
10      Alaska     710231     710231     714146     722718
11     Arizona    6392017    6392013    6413158    6482505
12    Arkansas    2915918    2915921    2921588    2937979
13  California   37253956   37253956   37338198   37691912
```

Well, the data look good, but what are the 9, 10, 11, 12, and 13? They are row names—which the read.csv function defined. At the time, those numbers were the same as the row number in the file. But now, these make no sense (if you remember, we deleted the first eight rows in this data set). So, we have to do one more command to remove the confusing row names with the following one line of R code:

```
> rownames(testFrame) <- NULL
```

This line basically tells R that we do not want to have row names and is similar to colnames(), but works on the row names, not the column names.

```
> head(testFrame,5)
stateName april10census april10base july10pop july11pop
1     Alabama    4779736    4779735    4785401    4802740
2      Alaska     710231     710231     714146     722718
3     Arizona    6392017    6392013    6413158    6482505
4    Arkansas    2915918    2915921    2921588    2937979
5  California   37253956   37253956   37338198   37691912
```

That's much better. Notice that we've spent a lot of time just conditioning the data we got in order to make it usable for later analysis. Herein lies a very important lesson. An important, and sometimes time-consuming, aspect of what data scientists do is to make sure that data are fit for the purpose to which they are going to be put. We had the convenience of importing a nice data set directly from the web with one simple command, and yet getting those data actually ready to analyze took several additional steps.

SORTING DATAFRAMES

Now that we have a real data set, let's do something with it! How about showing the five states with the highest populations? One way to do this is to sort the data set by the july11pop. But, while we can sort a vector with the sort command, sorting the dataframe is somewhat more challenging. So, let's explore how to sort a column in a dataframe, and basically reorder the dataframe. To accomplish this, we will use the order() function together with R's built-in square bracket notation.

As a reminder, we can supply a list of rows to access the dataframe: myFrame[c(1,3,5),] would return rows 1, 3, 5 (including the data for all columns, because we left the column index blank). We can use this feature to reorder the rows, using the order() function. We tell order() which variable we want to sort on, and it will give back a list of row indices in the order we requested.

Putting it all together yields this command:

```
> sortedStates <-
+    testFrame[order(testFrame$july11pop), ]
```

Working our way from the inside to the outside of the expression above, we want to sort in the order of the population, as defined by the july11pop column. We wrap this inside the order() function. The order() function will provide a list of row indices that reflects the

population of the states. We use the square brackets notation to address the rows in the testFrame, taking all of the columns by leaving the index after the comma empty. Finally, we stored the new dataframe in variable sortedStates. Let's take a look at our results:

```
> head(sortedStates,5)
                  stateName april10census april10base
59                  Wyoming        563626       563626
17 District of Columbia        601723       601723
54                  Vermont        625741       625741
43             North Dakota        672591       672591
10                   Alaska        710231       710231

          july10pop     july11pop
51               564554        568158
9                604912        617996
46               625909        626431
35               674629        683932
2                714146        722718
```

Well, that is close, but it's the states with the lowest populations. We wanted the states with the largest (greatest) populations. We can either use the tail command to see the states with the largest population, or do the sort, but tell R to sort largest to smallest. We tell R we want the largest populations first by putting a minus sign (–) next to the vector we want sorted. What this actually does is that it makes the large numbers large negative numbers (so they are smaller), and the small numbers small negative numbers (so they are larger relative to the negative larger numbers). Wow, that's confusing, but it is easy to do in R, and is done as follows:

```
> sortedStates <-
+    testFrame[order(-testFrame$july11pop), ]
> head(sortedStates,5)
stateName april10census april10base july10pop july11pop
5  California   37253956   37253956   37338198   37691912
44      Texas   25145561   25145561   25253466   25674681
33   New York   19378102   19378104   19395206   19465197
10    Florida   18801310   18801311   18838613   19057542
14   Illinois   12830632   12830632   12841980   12869257
```

That's it! We can see California has the most people, followed by Texas, and then New York.

In summary, as you have seen, data munging requires lots of knowledge of how to work with dataframes, combined with persistence to get the data into a format that is useful. While we have explored some common challenges related to data munging, there are other challenges we did not get to in this chapter. One classic challenge is working with dates, in that there are many formats such as a year with two or four digits, and dates with the month or day is listed first. Another challenge often seen is when we want to combine two data sets. Combining them can be useful, for example when you have a data set with a person's name (or id) and her purchase history. A related data set might have that person's name (or id) and the state where she lives.

Chapter Challenge

Practice reading in a data set; this time the data set is about loans. Go to the lendingClub website (http://www.lendingclub.com/info/download-data.action), download a CSV file and then read in the file (using read.csv). Then, clean up the data set, making sure all the columns have useful information. This means you must explore the data set to understand what needs to be done! One trick to get you started is that you might need to skip one or more lines (before the header line in the CSV file). There is a skip parameter that you can use in your read.csv() command.

Sources

http://www2.census.gov/programs-surveys/popest/

R Commands Used in This Chapter

read.csv()	read in a CSV file
url()	make sure R knows that the file is a URL (not a local file)
gsub()	substitute one string for another
rownames()	get/set the row names for the data
colnames()	get/set the column names for the dataframe
order()	return the indices in the order of the vector supplied