COMPUTERWORLD

Print Article ☒ Close Window

Beginner's guide to R: Easy ways to do basic data analysis

Part 3 of our hands-on series covers pulling stats from your data frame, and related topics.

Sharon Machlis

June 6, 2013 (Computerworld)

So you've read your data into an R object. Now what?

Examine your data object

Before you start analyzing, you might want to take a look at your data object's structure and a few row entries. If it's a 2-dimensional table of data stored in an R data frame object with rows and columns -- one of the more common structures you're likely to encounter -- here are some ideas. Many of these also work on 1-dimensional vectors as well.

Many of the commands below assume that your data are stored in a variable called *mydata* (and not that *mydata* is somehow part of these functions' names).

[This story is part of Computerworld's "Beginner's guide to R." To read from the beginning, check out the introduction; there are links on that page to the other pieces in the series.]

If you type:

```
head(mydata)
```

R will display mydata's column headers and first 6 rows by default. Want to see, oh, the first 10 rows instead of 6? That's:

```
head(mydata, n=10)
```

Or just:

```
head(mydata, 10)
```

Note: If your object is just a 1-dimensional vector of numbers, such as (1, 1, 2, 3, 5, 8, 13, 21, 34), head(mydata) will give you the first 6 items in the vector.

To see the *last* few rows of your data, use the tail() function:

```
tail(mydata)
```

Or:

```
tail(mydata, 10)
```

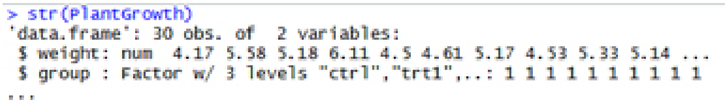
Tail can be useful when you've read in data from an external source, helping to see if anything got garbled (or there was some footnote row at the end you didn't notice).

To quickly see how your R object is structured, you can use the str() function:

`str(mydata)`

This will tell you the type of object you have; in the case of a data frame, it will also tell you how many rows (observations in statistical R-speak) and columns (variables to R) it contains, along with the type of data in each column and the first few entries in each column.

For a vector, str() tells you how many items there are -- for 8 items, it'll display as [1:8] -- along with the type of item (number, character, etc.) and the first few entries.



```
> str(PlantGrowth)
'data.frame':	30 obs. of  2 variables:
 $ weight: num  4.17 5.58 5.18 6.11 4.5 4.61 5.17 4.53 5.33 5.14 ...
 $ group : Factor w/ 3 levels "ctrl","trt1",..: 1 1 1 1 1 1 1 1 1 1
...
```

Results of the str() function on the sample data set PlantGrowth.

Various other data types return slightly different results.

If you want to see just the column names in the data frame called mydata, you can use the command:

`colnames(mydata)`

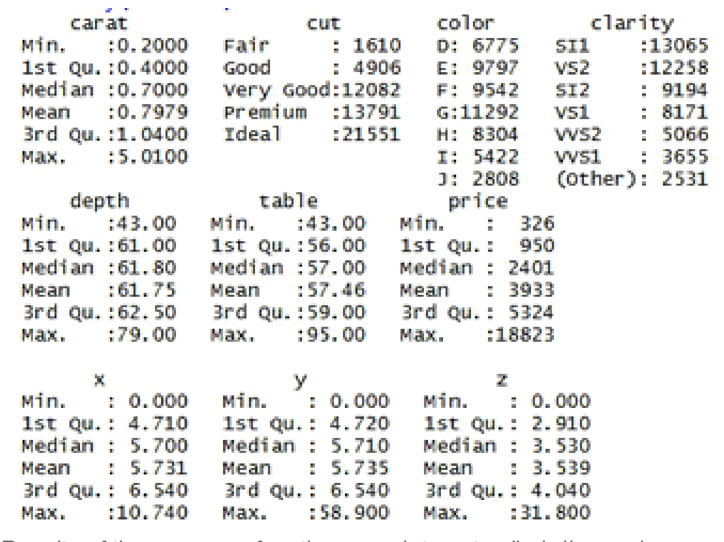
Likewise, if you're interested in the row names -- in essence, all the values in the first column of your data frame -- use:

`rownames(mydata)`

Pull basic stats from your data frame

Because R is a statistical programming platform, it's got some pretty elegant ways to extract statistical summaries from data. To extract a few basic stats from a data frame, use the summary() function:

`summary(mydata)`



```
     carat              cut        color        clarity     
 Min.   :0.2000   Fair     : 1610   D: 6775   SI1    :13065  
 1st Qu.:0.4000   Good     : 4906   E: 9797   VS2    :12258  
 Median :0.7000   Very Good:12082   F: 9542   SI2    : 9194  
 Mean   :0.7979   Premium  :13791   G:11292   VS1    : 8171  
 3rd Qu.:1.0400   Ideal    :21551   H: 8304   VVS2   : 5066  
 Max.   :5.0100                     I: 5422   VVS1   : 3655  
                                    J: 2808   (Other): 2531  
     depth           table           price      
 Min.   :43.00   Min.   :43.00   Min.   :  326  
 1st Qu.:61.00   1st Qu.:56.00   1st Qu.:  950  
 Median :61.80   Median :57.00   Median : 2401  
 Mean   :61.75   Mean   :57.46   Mean   : 3933  
 3rd Qu.:62.50   3rd Qu.:59.00   3rd Qu.: 5324  
 Max.   :79.00   Max.   :95.00   Max.   :18823  
                                                
       x                y                z         
 Min.   : 0.000   Min.   : 0.000   Min.   : 0.000  
 1st Qu.: 4.710   1st Qu.: 4.720   1st Qu.: 2.910  
 Median : 5.700   Median : 5.710   Median : 3.530  
 Mean   : 5.731   Mean   : 5.735   Mean   : 3.539  
 3rd Qu.: 6.540   3rd Qu.: 6.540   3rd Qu.: 4.040  
 Max.   :10.740   Max.   :58.900   Max.   :31.800  
```

Results of the summary function on a data set called diamonds, which is included in the ggplot2 add-on package.

That returns some basic calculations for each column. If the column has numbers, you'll see the minimum and maximum values along with median, mean, 1st quartile and 3rd quartile. If it's got factors such as fair, good, very good and excellent, you'll get the number of each factor listed in the column.

The summary() function also returns stats for a 1-dimensional vector.

If you'd like even more statistical summaries from a single command, install and load the psych package. Install it with this command:

`install.packages("psych")`

You need to run this install only once on a system. Then load it with:

`library(psych)`

You need to run the library command each time you start a new R session if you want to use the psych package.

Now try the command:

```
describe(mydata)
```

and you'll get several more statistics from the data including standard deviation, "mad" (mean absolute deviation), skew (measuring whether or not the data distribution is symmetrical) and kurtosis (whether the data have a sharp or flatter peak near its mean).

R has the statistical functions you'd expect, including mean(), median(), min(), max(), sd() [standard deviation], var() [variance] and range()which you can run on a 1-dimensional vector of numbers. (Several of these functions -- such as mean() and median() -- will not work on a 2-dimensional data frame).

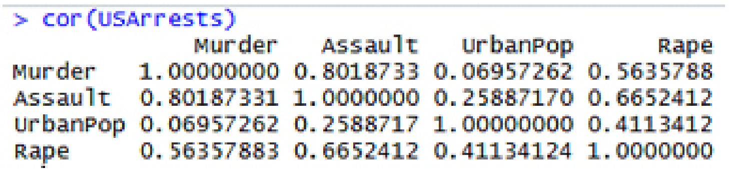
Oddly, the mode() function returns information about data type instead of the statistical mode; there's an add-on package, modeest, that adds a mfv() function (most frequent value) to find the statistical mode.

R also contains a load of more sophisticated functions that let you do analyses with one or two commands: probability distributions, correlations, significance tests, regressions, ANOVA (analysis of variance between groups) and more.

As just one example, running the correlation function cor() on a dataframe such as:

```
cor(mydata)
```

will give you a matrix of correlations for each column of numerical data compared with every other column of numerical data.



```
> cor(USArrests)
             Murder  Assault   UrbanPop      Rape
Murder   1.00000000 0.8018733 0.06957262 0.5635788
Assault  0.80187331 1.0000000 0.25887170 0.6652412
UrbanPop 0.06957262 0.2588717 1.00000000 0.4113412
Rape     0.56357883 0.6652412 0.41134124 1.0000000
```

Results of the correlation function on the sample data set of U.S.arrests.

Note: Be aware that you can run into problems when trying to run some functions on data where there are missing values. In *some* cases, R's default is to return NA even if just a single value is missing. For example, while the summary() function returns column statistics excluding missing values (and also tells you how many NAs are in the data), the mean() function will return NA if even only one value is missing in a vector.

In most cases, adding the argument:

```
na.rm=TRUE
```

to NA-sensitive functions will tell that function to remove any NAs when performing calculations, such as:

```
mean(myvector, na.rm=TRUE)
```

If you've got data with some missing values, read a function's help file by typing a question mark followed by the name of the function, such as:

```
?median
```

The function description should say whether the na.rm argument is needed to exclude missing values.

Checking a function's help files -- even for simple functions -- can also uncover additional useful options, such as an optional trim argument for mean() that lets you exclude some outliers.

Not all R functions need a robust data set to be useful for statistical work. For example, how many ways can you select a committee of 4 people from a group of 15? You can pull out your calculator and find 15! divided by 4! times 11! ... or you can use the R choose() function:

```
choose(15,4)
```

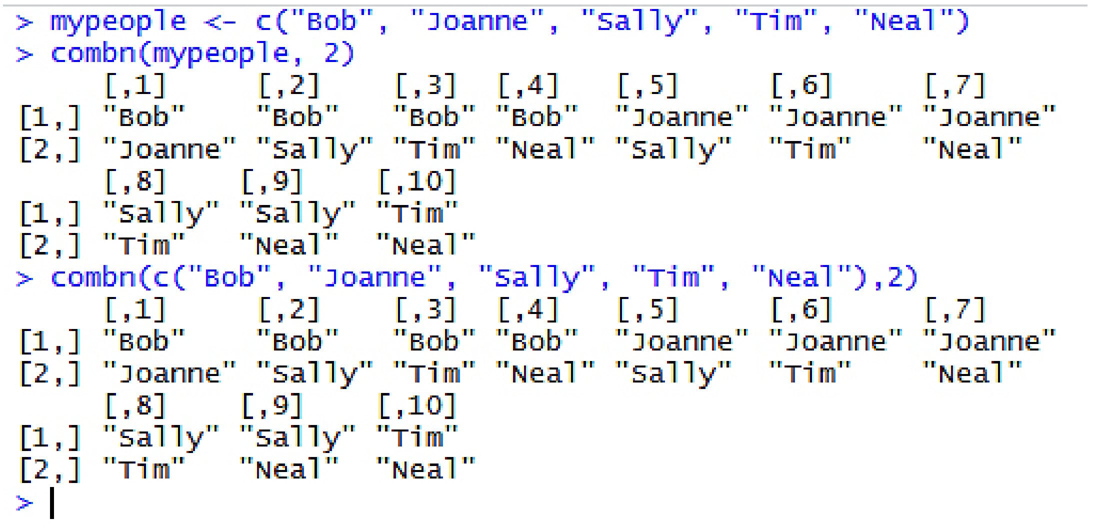
Or, perhaps you want to see all of the possible pair combinations of a group of 5 people, not simply count them. You can create a vector with the people's names and store it in a variable called mypeople:

```
mypeople <- c("Bob", "Joanne", "Sally", "Tim", "Neal")
```

In the example above, c() is the combine function.

Then run the combn() function, which takes two arguments -- your entire set first and then the number you want to have in each group:

```
combn(mypeople, 2)
```



```
> mypeople <- c("Bob", "Joanne", "Sally", "Tim", "Neal")
> combn(mypeople, 2)
     [,1]     [,2]    [,3]  [,4]   [,5]     [,6]     [,7]
[1,] "Bob"    "Bob"   "Bob" "Bob"  "Joanne" "Joanne" "Joanne"
[2,] "Joanne" "Sally" "Tim" "Neal" "Sally"  "Tim"    "Neal"
     [,8]    [,9]    [,10]
[1,] "Sally" "Sally" "Tim"
[2,] "Tim"   "Neal"  "Neal"
> combn(c("Bob", "Joanne", "Sally", "Tim", "Neal"),2)
     [,1]     [,2]    [,3]  [,4]   [,5]     [,6]     [,7]
[1,] "Bob"    "Bob"   "Bob" "Bob"  "Joanne" "Joanne" "Joanne"
[2,] "Joanne" "Sally" "Tim" "Neal" "Sally"  "Tim"    "Neal"
     [,8]    [,9]    [,10]
[1,] "Sally" "Sally" "Tim"
[2,] "Tim"   "Neal"  "Neal"
> |
```

Use the combine function to see all possible combinations from a group.

Probably most experienced R users would combine these two steps into one like this:

```
combn(c("Bob", "Joanne", "Sally", "Tim", "Neal"),2)
```

But separating the two can be more readable for beginners.

Get slices or subsets of your data

Maybe you don't need correlations for every column in your data frame and you just want to work with a couple of columns, not 15. Perhaps you want to see data that meets a certain condition, such as within 3 standard deviations. R lets you slice your data sets in various ways, depending on the data type.

To select just certain columns from a data frame, you can either refer to the columns by *name* or by their *location* (i.e., column 1, 2, 3, etc.).

For example, the mtcars sample data frame has these column names: mpg, cyl, disp, hp, drat, wt, qsec, vs, am, gear and carb.

Can't remember the names of all the columns in your data frame? If you just want to see the column names and nothing else, instead of functions such as str(mtcars) and head(mtcars) you can type:

```
names(mtcars)
```

That's handy if you want to store the names in a variable, perhaps called mtcars.colnames (or anything else you'd like to call it):

```
mtcars.colnames <- names(mtcars)
```

But back to the task at hand. To access *only the data* in the mpg column in mtcars, you can use R's dollar sign notation:

```
mtcars$mpg
```

More broadly, then, the format for accessing a column by name would be:

```
dataframename$columnname
```

That will give you a 1-dimensional vector of numbers like this:

```
[1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8

[12] 16.4 17.3 15.2 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5

[23] 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7 15.0 21.4
```

The numbers in brackets are not part of your data, by the way. They indicate what item number each line is starting with. If you've only got one line of data, you'll just see [1]. If there's more than one line of data and only the first 11 entries can fit on the first line, your second line will start with [12], and so on.

Sometimes a vector of numbers is exactly what you want -- if, for example, you want to quickly plot mtcars$mpg and don't need item labels, or you're looking for statistical info such as variance and mean.

Chances are, though, you'll want to subset your data by more than one column at a time. That's when you'll want to use bracket notation, what I think of as rows-comma-columns. Basically, you take the name of your data frame and follow it by [rows,columns]. The rows you want come first, followed by a comma, followed by the columns you want. So, if you want all rows but just columns 2 through 4 of mtcars, you can use:

```
mtcars[,2:4]
```

Do you see that comma before the 2:4? That's leaving a blank space where the "which rows do you want?" portion of the bracket notation goes, and it means "I'm not asking for any subset, so return all." Although it's not always required, it's not a bad practice to get into the habit of using a comma in bracket notation so that you remember whether you were slicing by columns or rows.

If you want multiple columns that aren't contiguous, such as columns 2 AND 4 but not 3, you can use the notation:

```
mtcars[,c(2,4)]
```

A couple of syntax notes here:

- R indexes from 1, not 0. So your first column is at [1] and not [0].
- R is case sensitive everywhere. mtcars$mpg is not the same as mtcars$MPG.
- mtcars[,-1] will *not* get you the last column of a data frame, the way negative indexing works in many other languages. Instead, negative indexing in R means *exclude that item*. So, mtcars[,-1] will return every column except the first one.
- To create a vector of items that are not contiguous, you need to use the combine function c(). Typing mtcars[,(2,4)] without the c will not work. You need that c in there:

```
mtcars[,c(2,4)]
```

What if want to select your data by data *characteristic*, such as "all cars with mpg > 20", and not column or row location? If you use the column name notation and add a condition like:

```
mtcars$mpg>20
```

you don't end up with a list of all rows where mpg is greater than 20. Instead, you get a vector showing whether each row meets the condition, such as:

```
[1] TRUE TRUE TRUE TRUE FALSE FALSE FALSE TRUE TRUE

[10] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE TRUE

[19] TRUE TRUE TRUE FALSE FALSE FALSE FALSE TRUE TRUE

[28] TRUE FALSE FALSE FALSE TRUE
```

To turn that into a listing of the data you want, use that logical test condition *and* row-comma-column bracket notation. Remember that this time you want to select *rows* by condition, not columns. This:

```
mtcars[mtcars$mpg>20,]
```

tells R to get all rows from mtcars where mpg > 20, and then to return all the columns.

If you don't want to see *all* the column data for the selected rows but are just interested in displaying, say, mpg and horsepower for cars with an mpg greater than 20, you could use the notation:

```
mtcars[mtcars$mpg>20,c(1,4)]
```

using column locations, or:

```
mtcars[mtcars$mpg>20,c("mpg","hp")]
```

using the column names.

Why do you need to specify mtcars$mpg in the row spot but "mpg" in the column spot? Just another R syntax quirk is the best answer I can give you.

If you're finding that your selection statement is starting to get unwieldy, you can put your row and column selections into variables first, such as:

```
mpg20 <- mtcars$mpg > 20

cols <- c("mpg", "hp")
```

Then you can select the rows and columns with those variables:

```
mtcars[mpg20, cols]
```

making for a more compact select statement but more lines of code.

Getting tired of including the name of the data set multiple times per command? If you're using only one data set *and* you are not making any changes to the data that need to be saved, you can attach and detach a copy of the data set temporarily.

The attach() function works like this:

```
attach(mtcars)
```

So, instead of having to type:

```
mpg20 <- mtcars$mpg > 20
```

You can leave out the data set reference and type this instead:

```
mpg20 <- mpg > 20
```

After using attach() remember to use the detach function when you're finished:

```
detach()
```

Some R users advise avoiding attach() because it can be easy to forget to detach(). If you don't detach() the copy, your variables could end up referencing the wrong data set.

Alternative to bracket notation

Bracket syntax is pretty common in R code, but it's not your only option. If you dislike that format, you might prefer the subset() function instead, which works with vectors and matrices as well as data frames. The format is:

```
subset(your data object, logical condition for the rows you want to return,
select statement for the columns you want to return)
```

So, in the mtcars example, to find all rows where mpg is greater than 20 and return only those rows with their mpg and hp data, the subset() statement would look like:

```
subset(mtcars, mpg>20, c("mpg", "hp"))
```

What if you wanted to find the row with the highest mpg?

```
subset(mtcars, mpg==max(mpg))
```

If you just wanted to see the mpg information for the highest mpg:

```
subset(mtcars, mpg==max(mpg), mpg)
```

If you just want to use subset to extract some columns and display all rows, you can either leave the row conditional spot blank with a comma, similar to bracket notation:

```
subset(mtcars, , c("mpg", "hp"))
```

Or, indicate your second argument is for columns with select= like this:

```
subset(mtcars, select=c("mpg", "hp"))
```

Update: The dplyr package, released in early 2014, is aimed at making manipulation of data frames faster and more rational, with similar syntax for a variety of tasks. To select certain rows based on specific logical criteria, you'd use the filter() function with the syntax `filter(dataframename, logical expression)`. As with subset(), column names stand alone after the data frame name, so mpg>20 and not mtcars$mpg > 20.

```
filter(mtcars, mpg>20)
```

To choose only certain columns, you use the select() function with syntax such as `select(dataframename, columnName1, columnName2)`. No quotation marks are needed with the column names:

```
select(mtcars, mpg, hp)
```

You can also combine filter and subset with the dplyr %.% "chaining" operation that allows you to string together multiple commands on a data frame. The chaining syntax in general is:

```
dataframename %.%
firstfunction(argument for first function) %.%
secondfunction(argument for second function) %.%
thirdfunction(argument for third function)
```

So viewing just mpg and hp for rows where mpg is greater than 20:

```
mtcars %.%
filter(mpg > 20) %.%
select(mpg, hp)
```

No need to keep repeating the data frame name. To order those results from highest to lowest mpg, add the arrange() function to the chain with desc(columnName) for descending order:

```
mtcars %.%
filter(mpg > 20) %.%
select(mpg, hp) %.%
arrange(desc(mpg))
```

You can find out more about dplyr in the dplyr package's introduction vignette.

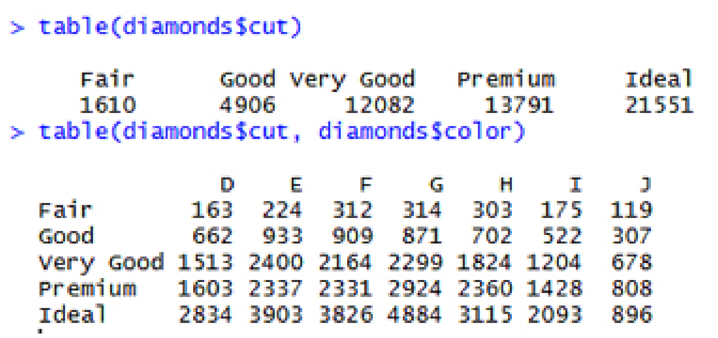
Counting factors

To tally up counts by factor, try the table command. For the diamonds data set, to see how many diamonds of each category of cut are in the data, you can use:

```
table(diamonds$cut)
```

This will return how many diamonds of each factor -- fair, good, very good, premium and ideal -- exist in the data. Want to see a cross-tab by cut *and* color?

```
table(diamonds$cut, diamonds$color)
```



```
> table(diamonds$cut)

     Fair      Good Very Good   Premium     Ideal 
     1610      4906     12082     13791     21551 
> table(diamonds$cut, diamonds$color)
           
               D    E    F    G    H    I    J
  Fair       163  224  312  314  303  175  119
  Good       662  933  909  871  702  522  307
  Very Good 1513 2400 2164 2299 1824 1204  678
  Premium   1603 2337 2331 2924 2360 1428  808
  Ideal     2834 3903 3826 4884 3115 2093  896
```

R's table function returns a count of each factor in your data.

If you are interested in learning more about statistical functions in R and how to slice and dice your data, there are a number of free academic downloads with many more details. These include *Learning statistics with R* by Daniel Navarro at the University of Adelaide in Australia (500+ page PDF download, may take a little while). And although not free, books such as *The R Cookbook* and *R in a Nutshell* have a lot of good examples and well-written explanations.

***Next: Painless data visualization*.**

Learn to use R: Your hands-on guide

Don't miss the rest:

- Part 1: Introduction - get started with this popular programming language.
- Part 2: Getting your data into R - tips on how to import data in various formats, both local and on the Web.
- Part 3: Easy ways to do basic data analysis - extract some simple stats.
- Part 4: Painless data visualization - simple graphics, bar graphs and a few more complex charts.
- Part 5: Syntax quirks you'll want to know - some R idiosyncrasies.
- Part 6: Useful resources - not for beginners only!

This article, Beginner's guide to R: Easy ways to do basic data analysis, was originally published at Computerworld.com.

***Sharon Machlis** is online managing editor at Computerworld. Her e-mail address is smachlis@computerworld.com. You can follow her on Twitter @sharon000, on Facebook, on Google+ or by subscribing to her RSS feeds: articles; and blogs.*