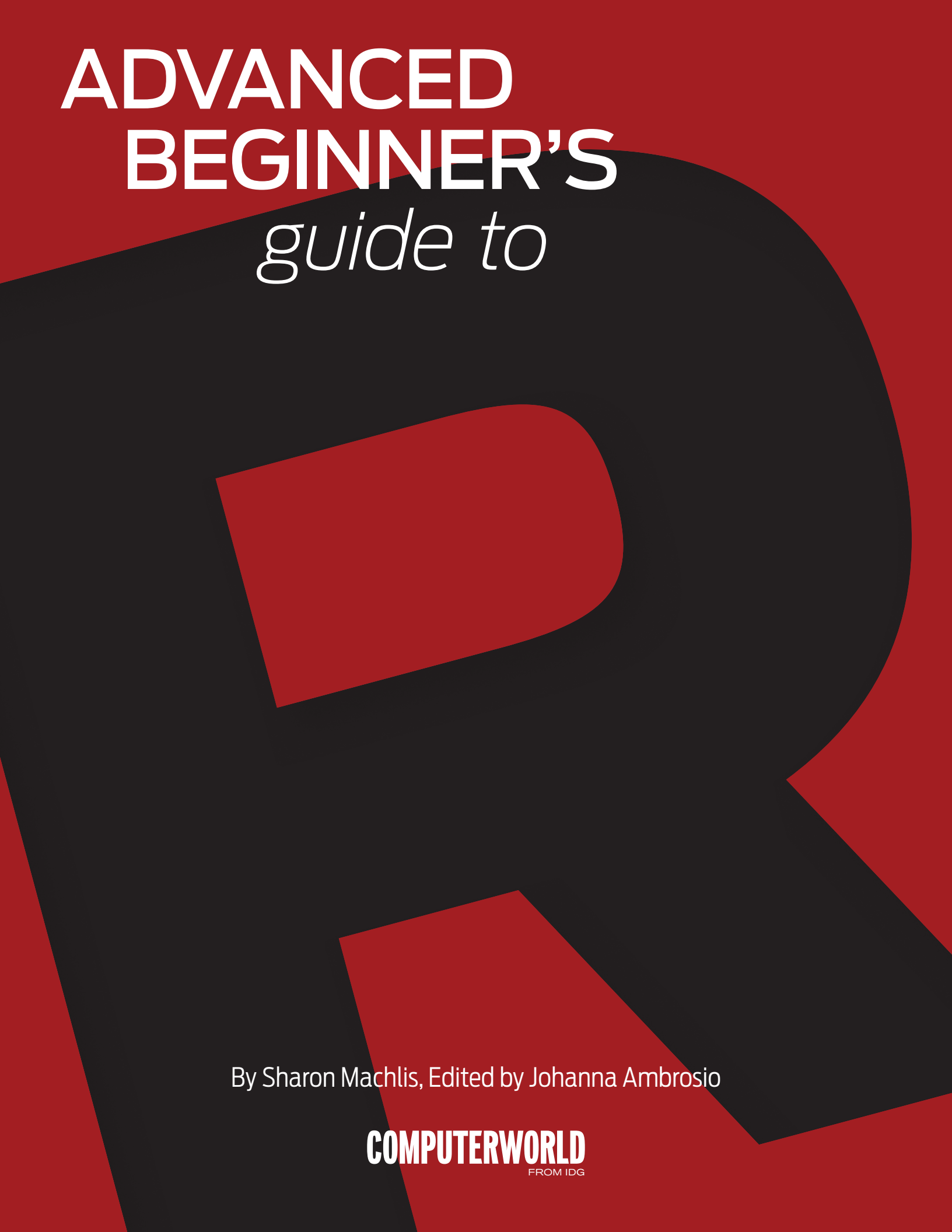
ADVANCED BEGINNER'S *guide to*



By Sharon Machlis, Edited by Johanna Ambrosio

COMPUTERWORLD
FROM IDG

So you've gone through the Computerworld *Beginner's Guide to R* and want to take some next steps in your R journey? In this advanced beginner's guide, you'll learn data wrangling, best packages to use for different tasks, how to make maps with R and more.

Table of Contents

Wrangle data with R

I've created a sample data set with three years of revenue and profit data from Apple, Google and Microsoft. (The source of the data was the companies themselves; fy means fiscal year. And while the data is a bit old now, the wrangling will be the same regardless of fiscal year.) If you'd like to follow along, you can type (or cut and paste) this into your R terminal window:

```
fy <- c(2010,2011,2012,2010,2011,2012,2010,2011,2012)
company <- c("Apple","Apple","Apple","Google","Google",
"Google","Microsoft","Microsoft","Microsoft")
revenue <- c(65225,108249,156508,29321,37905,50175,62484,
69943,73723)
profit <- c(14013,25922,41733,8505,9737,10737,18760,23150,
16978)
companiesData <- data.frame(fy, company, revenue, profit)
```

The code above will create a data frame like the one below, stored in a variable named "companiesData":

	fy	company	revenue	profit
1	2010	Apple	65225	14013
2	2011	Apple	108249	25922
3	2012	Apple	156508	41733
4	2010	Google	29321	8505
5	2011	Google	37905	9737
6	2012	Google	50175	10737
7	2010	Microsoft	62484	18760
8	2011	Microsoft	69943	23150
9	2012	Microsoft	73723	16978

(R adds its own row numbers if you don't include row names.)

If you run the str() function on the data frame to see its structure, you'll see that the year is being treated as a number and not as a year or factor:

```
str(companiesData)
'data.frame': 9 obs. of 4 variables:
$ fy : num 2010 2011 2012 2010 2011 ...
$ company: Factor w/ 3 levels "Apple","Google",..: 1 1 1 2
2 2 3 3 3
$ revenue: num 65225 108249 156508 29321 37905 ...
$ profit : num 14013 25922 41733 8505 9737 ...
```

I may want to group my data by year, but don't think I'm going to be doing specific time-based analysis, so I'll turn the fy column of numbers into a column that contains R categories (called factors) instead of dates with the following command:

```
companiesData$fy <- factor(companiesData$fy,
ordered = TRUE)
```

Now we're ready to get to work.

Add a column to an existing data frame

One of the easiest tasks to perform in R is adding a new column to a data frame based on one or more other columns. You might want to add up several of your existing columns, find an average or otherwise calculate some "result" from existing data in each row.

There are many ways to do this in R. Some will seem overly complicated for this easy task at hand, but for now you'll have to take my word for it that some more complex options can come in handy for advanced users with more robust needs.

Syntax 1: By equation

Simply create a variable name for the new column and pass in a calculation formula as its value if, for example, you want a new column that's the sum of two existing columns:

dataFrame$newColumn <- dataFrame$oldColumn1 + dataFrame$oldColumn2

As you can probably guess, this creates a new column called "newColumn" with the sum of oldColumn1 + oldColumn2 in each row.

For our sample data frame called data, we could add a column for profit margin by dividing profit by revenue and then multiplying by 100:

```
companiesData$margin <- (companiesData$profit /
companiesData$revenue) * 100
```

That gives us:

	fy	company	revenue	profit	margin
1	2010	Apple	65225	14013	21.48409
2	2011	Apple	108248	25922	23.94664
3	2012	Apple	156508	41733	26.66509
4	2010	Google	29321	8505	29.00651
5	2011	Google	37905	9737	25.68790
6	2012	Google	50175	10737	21.39910
7	2010	Microsoft	62484	18760	30.02369
8	2011	Microsoft	69943	23150	33.09838
9	2012	Microsoft	73723	16978	23.02945

Whoa — that's a lot of decimal places in the new margin column.

We can round that off to just one decimal place with the round() function; round() takes the format:

round(number(s) to be rounded, how many decimal places you want)

So, to round the margin column to one decimal place:

```
companiesData$margin <- round(companiesData$margin, 1)
```

And you'll get this result:

	fy	company	revenue	profit	margin
1	2010	Apple	65225	14013	21.5
2	2011	Apple	108248	25922	23.9
3	2012	Apple	156508	41733	26.7
4	2010	Google	29321	8505	29.0
5	2011	Google	37905	9737	25.7
6	2012	Google	50175	10737	21.4
7	2010	Microsoft	62484	18760	30.0
8	2011	Microsoft	69943	23150	33.1
9	2012	Microsoft	73723	16978	23.0

Syntax 2: R's transform() function

This is another way to accomplish what we did above. Here's the basic transform() syntax:

dataFrame <- transform(dataFrame, newColumnName = some equation)

So, to get the sum of two columns and store that into a new column with transform(), you would use code such as:

dataFrame <- transform(dataFrame, newColumn = oldColumn1 + oldColumn2)

To add a profit margin column to our data frame with transform() we'd use:

```
companiesData <- transform(companiesData, margin =
(profit/revenue) * 100)
```

We can then use the round() function to round the column results to one decimal place. Or, in one step, we can create a new column that's already rounded to one decimal place:

```
companiesData <- transform(companiesData, margin =
round((profit/revenue) * 100, 1))
```

One brief aside about round(): You can use negative numbers for the second, "number of decimal places" argument. While round(73842.421, 1) will round to one decimal, in this case 73842.42, round(73842.421, -3) will round to the nearest thousand, in this case 74000.

Syntax 3: R's apply() function

As the name helpfully suggests, this will *apply* a function to a data frame (or several other R data structures, but we'll stick with data frames for now). This syntax is more complicated than the first two but can be useful for some more complex calculations.

The basic format for apply() is:

dataFrame$newColumn <- apply(dataFrame, 1, function(x) { . . . })

The line of code above will create a new column called "newColumn" in the data frame; the contents will be whatever the code in { . . . } does.

Here's what each of those apply() arguments above is doing. The first argument for apply() is the existing data frame. The second argument — 1 in this example — means "apply

a function *by row*." If that argument was 2, it would mean "apply a function by column" — for example, if you wanted to get a sum or average by columns instead of for each row.

The third argument, function(x), should appear as written. More specifically the function( ) part needs to be written as just that; the "x" can be any variable name. This means "What follows after this is an ad-hoc function that I haven't named. I'll call its input argument x." What's x in this case? It's each item (row or column) being iterated over by apply().

Finally, { . . . } is whatever you want to be doing with each item you're iterating over.

Keep in mind that apply() will seek to apply the function on *every* item in each row or column. That can be a problem if you're applying a function that works only on numbers if some of your data frame columns aren't numbers.

That's exactly the case with our sample data of financial results. For the data variable, this won't work:

```
apply(companiesData, 1, function(x) sum(x))
```

Why? Because (apply) will try to sum every item per row, and company names can't be summed.

To use the apply() function on only *some* columns in the data frame, such as adding the revenue and profit columns together (which, I'll admit, is an unlikely need in the real world of financial analysis), we'd need to use *a subset of the data frame as our first argument*. That is, instead of using apply() on the entire data frame, we just want apply() on the revenue and profit columns, like so:

```
apply(companiesData[,c('revenue', 'profit')], 1,
function(x) sum(x))
```

Where it says:

[c('revenue', 'profit')]

after the name of the data frame, it means "only use columns revenue and profit" in the sum.

You then might want to store the results of apply in a new column, such as:

```
companiesData$sums <- apply(companiesData[,
c('revenue', 'profit')], 1, function(x) sum(x))
```

That's fine for a function like sum, where you take each number and do the same thing to it. But let's go back to our earlier example of calculating a profit margin for each row. In that case, we need to pass profit and revenue in a certain order — it's profit divided by revenue, not the other way around — and then multiply by 100.

How can we pass multiple items to apply() in a certain order for use in an anonymous function(x)? By referring to the items in our anonymous function as x[1] for the first one, x[2] for the second, etc., such as:

```
companiesData$margin <- apply(companiesData[,
c('revenue', 'profit')], 1,
function(x) { (x[2]/x[1]) * 100 } )
```

That line of code above creates an anonymous function that uses the second item — in this case profit, since it's listed second in companiesData[,c('revenue', 'profit')] — and divides it by the first item in each row, revenue. This will work because there are only two items here, revenue and profit — remember, we told apply() to use only those columns.

Syntax 4: mapply()

This, and the simpler sapply(), also can apply a function to some — but not necessarily all — columns in a data frame, without having to worry about numbering each item like x[1] and x[2] above. The mapply() format to create a new column in a data frame is:

dataFrame$newColumn <- mapply(someFunction, dataFrame$column1, dataFrame$column2, dataFrame$column3)

The code above would apply the function someFunction() to the data in column1, column2 and column3 of each row of the data frame.

Note that the first argument of mapply() here is the *name of a function*, not an equation or formula. So if we want (profit/revenue) * 100 as our result, we could first write our own function to do this calculation and then use it with mapply().

Here's how to create a named function, profitMargin(), that takes two variables — in this case we're calling them netIncome and revenue just within the function — and return the first variable divided by the second variable times 100, rounded to one decimal place:

```
profitMargin <- function(netIncome, revenue) {
    mar <- (netIncome/revenue) * 100
    mar <- round(mar, 1)
    }
```

Now we can use that user-created named function with mapply():

```
companiesData$margin <- mapply(profitMargin,
companiesData$profit, companiesData$revenue)
```

Or we could create an anonymous function within mapply():

```
companiesData$margin <- mapply(function(x, y)
round((x/y) * 100, 1), companiesData$profit,
companiesData$revenue)
```

One advantage mapply() has over transform() is that you can use columns from different data frames (note that this may not always work if the columns are different lengths). Another is that it's got an elegant syntax for applying functions to vectors of data when a function takes more than one argument, such as:

mapply(someFunction, vector1, vector2, vector3)

sapply() has a somewhat different syntax from mapply, and there are yet more functions in R's apply family. I won't go into them further here, but this may give you a sense of why R maestro Hadley Wickham created his own package called plyr with functions *all having the same syntax* in order to try to rationalize applying functions in R. (We'll get to plyr in the next section.)

For a more detailed look at base R's various apply options, *A brief introduction to 'apply' in R* by bioinformatician Neil Saunders is a useful starting point.

Syntax 5: dplyr

Hadley Wickham's dplyr package, released in early 2014 to rationalize and speed up operations on data frames, is an extremely useful addition to anyone's R arsenal and well worth learning. To add a column to an existing data frame with dplyr, first install the package with `install.packages("dplyr")` — you only need to do this once — and then load it with `library("dplyr")`. To add a column using dplyr:

```
companiesData <- mutate(companiesData, margin =
round((profit/revenue) * 100, 1))
```

See more about dplyr in the dplyr basics section.

Getting summaries by subgroups of your data

It's easy to find, say, the highest profit margin in our data with max(companiesData$margin). To assign the *value* of the highest profit margin to a variable named highestMargin, this simple code does the trick.

```
highestMargin <- max(companiesData$margin)
```

That just returns:

[1] 33.09838

but you don't know anything more about the other variables in the row, such as year and company.

To see the *entire row* with the highest profit margin, not only the value, this is one option:

```
highestMargin <- companiesData[companiesData$
margin == max(companiesData$margin),]
```

and here's another:

```
highestMargin <- subset(companiesData,
margin==max(margin))
```

(For an explanation on these two techniques for extracting subsets of your data, see Get slices or subsets of your data from the Beginner's guide to R: Easy ways to do basic data analysis.)

But what if you want to find rows with the highest profit margin *for each company*? That involves applying a function by groups — what R calls factors.

Both the older plyr and newer dplyr packages created by Hadley Wickham consider this type of task "split-apply-combine": Split up your data set by one or more factors, apply some function, then combine the results back into a data set.

plyr's ddply() function performs a "split-apply-combine" on a data frame and then produces a new separate data frame with your results. That's what the first two letters, dd, stand for in ddply(), by the way: Input a **d**ata frame and get a **d**ata

frame back. There's a whole group of "ply" functions in the plyr package: alply to input an array and get back a list, ldply to input a list and get back a data frame, and so on. dplyr only handles data frames.

To use the ddply() function, first you need to install the plyr package if you never have, with:

```
install.packages("plyr")
```

Then, if you haven't yet for your current R session, load the plyr package with:

```
library("plyr")
```

The format for splitting a data frame by multiple factors and applying a function with ddply would be:

ddply(mydata, c('column name of a factor to group by', 'column name of the second factor to group by'), summarize OR transform, newcolumn = myfunction(column name(s) I want the function to act upon))

Let's take a more detailed look at that. The ddply() first argument is the name of the original data frame and the second argument is the name of the column or columns you want to subset your data by. The third tells ddply() whether to return just the resulting data points (summarize) or the entire data frame with a new column giving the desired data point per factor in every row. Finally, the fourth argument names the new column and then lists the function you want ddply() to use.

If you don't want to have to put the column names in quotes, an alternate syntax you'll likely see frequently uses a dot before the column names:

myresult <- ddply(mydata, .(column name of factor I'm splitting by, column name second factor I'm splitting by), summarize OR transform, newcolumn = myfunction(column name I want the function to act upon))

To get the highest profit margins for each company, we're splitting the data frame by only one factor — company. To get *just* the highest value and company name for each company, use summarize as the third argument:

```
highestProfitMargins <- ddply(companiesData,
.(company), summarize, bestMargin = max(margin))
```

(Here we've assigned the results to the variable highestProfitMargins.)

Syntax note: Even if you've only got one factor, it needs to be in parentheses after that dot if you're using the dot to avoid putting the column name in quotes. No parentheses are needed for just one factor if you're using quotation marks:

```
highestProfitMargins <- ddply(companiesData,
'company', summarize, bestMargin = max(margin))
```

Either way, you'll end up with a brand new data frame with the highest profit margin for each company:

	company	bestMargin
1	Apple	26.7
2	Google	29.0
3	Microsoft	33.1

Summarize doesn't give any information from other columns in the original data frame. In what year did each of the highest margins occur? We can't tell by using summarize.

If you want all the other column data, too, change "summarize" to "transform." That will return your existing data frame with a new column that repeats the maximum margin for each company:

```
highestProfitMargins <- ddply(companiesData,
'company', transform, bestMargin = max(margin))
```

	fy	company	revenue	profit	margin	bestMargin
1	2010	Apple	65225	14013	21.5	26.7
2	2011	Apple	108248	25922	23.9	26.7
3	2012	Apple	156508	41733	26.7	26.7
4	2010	Google	29321	8505	29.0	29.0
5	2011	Google	37905	9737	25.7	29.0
6	2012	Google	50175	10737	21.4	29.0
7	2010	Microsoft	62484	18760	30.0	33.1
8	2011	Microsoft	69943	23150	33.1	33.1
9	2012	Microsoft	73723	16978	23.0	33.1

Note that this result shows the profit margin for each company and year in the margin column along with the bestMargin repeated for each company and year. The only way to tell which year has the best margin is to compare the two columns to see where they're equal.

ddply() lets you apply more than one function at a time, for example:

```
myResults <- ddply(companiesData, 'company',
transform, highestMargin = max(margin),
lowestMargin = min(margin))
```

This gets you:

	fy	company	revenue	profit	margin	highest-Margin	lowest-Margin
1	2010	Apple	65225	14013	21.5	26.7	21.5
2	2011	Apple	108248	25922	23.9	26.7	21.5
3	2012	Apple	156508	41733	26.7	26.7	21.5
4	2010	Google	29321	8505	29.0	29.0	21.4
5	2011	Google	37905	9737	25.7	29.0	21.4
6	2012	Google	50175	10737	21.4	29.0	21.4
7	2010	Microsoft	62484	18760	30.0	33.1	23.0
8	2011	Microsoft	69943	23150	33.1	33.1	23.0
9	2012	Microsoft	73723	16978	23.0	33.1	23.0

In some cases, though, what you want is a new data frame with just the (entire) rows that have the highest profit margins. One way to do that is with the somewhat more complex syntax below:

```
highestProfitMargins <- ddply(companiesData,
'company', function(x) x[x$margin==max(x$margin),])
```

	fy	company	revenue	profit	margin
1	2012	Apple	156508	41733	26.7
2	2010	Google	29321	8505	29.0
3	2011	Microsoft	69943	23150	33.1

That may look a bit daunting, but really it's not so bad once you break it down. Let's take it step by step.

The ddply(companiesData, company, function(x)) portion should look familiar by now: companiesData is the original data frame and function(x) says that an anonymous

(unnamed, ad-hoc) function is coming next. So the only new part is:

x[x$margin==max(x$margin),]

That code is extracting a subset of x. In this case, x refers to the data frame that was passed into the anonymous function. The equation inside the bracket says: I want to match every row where x$margin equals the maximum of x$margin. The comma after x$margin==max(x$margin) tells R to return every column of those matching rows, since no columns were specified. As an alternative, we could seek to return only one or several of the columns instead of all of them.

Note that:

companiesData[companiesData$margin==max(companiesData$margin),]

alone, without ddply(),gives the highest *overall* margin, not the highest margin for *each company*. But since the anonymous function is being passed into a ddply() statement that's splitting the data frame by company, what's returned is the matching row(s) for each company.

One more note about ddply(): While it's designed for "split, apply, combine" — that is, applying a function to different categories of your data — you can still use it to apply a function to your entire data frame at once. So, once again here's the ddply() statement we used to get a summary of highest profit margin for each company:

```
highestProfitMargins <- ddply(companiesData,
'company', summarize, bestMargin = max(margin))
```

To use ddply() to see the highest margin *in the entire data set*, not just segmented by company, I'd enter NULL as the second argument for factors to split by:

```
highestProfitMargin <- ddply(companiesData, NULL,
summarize, bestMargin = max(margin))
```

That's obviously a much more complicated way of doing this than max(companiesData$margin). But you nevertheless may find plyr's "ply" family useful at times if you want to apply multiple functions on an entire data structure and like the idea of consistent syntax.

Why learn plyr as well as dplyr? Because your data might not always be in a data frame. If you do have a data frame,

though, dplyr is usually an excellent choice. Performing these operations with dplyr is considerably faster than with plyr — not an issue for a tiny data frame like this, but important if you've got data with thousands of rows. In addition, I find dplyr syntax to be more readable and intuitive — once you get used to it.

To add the two columns for highest and lowest margins by company:

```
 myresults <- companiesData %>%
    group_by(company) %>%
    mutate(highestMargin = max(margin), lowestMargin =
min(margin))
```

and to create a new data frame with maximum margin by company:

```
highestProfitMargins <- companiesData %>%
    group_by(company) %>%
    summarise(bestMargin = max(margin))
```

The %>% is a "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)
```

and so on for as many functions as you want to chain. Why? This lets you group, sort, filter, summarize and more — all in one block of readable code. In the highestProfitMargins example above, we're first grouping the companiesData data frame by the company column, then getting the maximum margin for each one of those groups and putting it in a new column called bestMargin. Finally, those results will be stored in a variable called highestProfitMargins.

In the myresults example, we're taking the companiesData data frame, grouping it by company and then using mutate() to add two columns: highestMargin and lowestMargin. Those results are being stored in the variable myresults.

Note that highestProfitMargins and myresults are a special type of data frame created by dplyr. If you have problems running more conventional non-dplry operations on a dplyr result, convert it to a "regular" data frame with as.data.frame(), such as

```
highestProfitMargins <- as.data.
frame(highestProfitMargins)
```

Bonus special case: Grouping by date range

If you've got a series of dates and associated values, there's an extremely easy way to group them by date range such as week, month, quarter or year: R's cut() function.

Here are some sample data in a vector:

```
vDates <- as.Date(c("2013-06-01", "2013-07-08",
"2013-09-01", "2013-09-15"))
```

Which creates:

[1] "2013-06-01" "2013-07-08" "2013-09-01" "2013-09-15"

The as.Date() function is important here; otherwise R will view each item as a string object and not a date object.

If you want a second vector that sorts those by month, you can use the cut() function using the basic syntax:

```
vDates.bymonth <- cut(vDates, breaks = "month")
```

That produces:

[1] 2013-06-01 2013-07-01 2013-09-01 2013-09-01

Levels: 2013-06-01 2013-07-01 2013-08-01 2013-09-01

It might be easier to see what's happening if we combine these into a data frame:

```
dfDates <- data.frame(vDates, vDates.bymonth)
```

Which creates:

	vDates	vDates.bymonth
1	2013-06-01	2013-06-01
2	2013-07-08	2013-07-01
3	2013-09-01	2013-09-01
4	2013-09-15	2013-09-01

The new column gives the starting date for each month, making it easy to then slice by month.

Ph.D. student Mollie Taylor's blog post Plot Weekly or Monthly Totals in R introduced me to this shortcut, which isn't apparent if you simply read the cut() help file. If you ever work with analyzing and plotting date-based data, this short and extremely useful post is definitely worth a read. Her downloadable code is available as a GitHub gist.

Sorting your results

For a simple sort by one column, you can get the order you want with the order() function, such as:

```
companyOrder <- order(companiesData$margin)
```

This tells you *how your rows would be reordered*, producing a list of line numbers such as:

6 1 9 2 5 3 4 7 8

Chances are, you're not interested in the new order by line number but instead actually want to see the data reordered. You can use that order to reorder rows in your data frame with this code:

```
companiesOrdered <- companiesData[companyOrder,]
```

where companyOrder is the order you created earlier. Or, you can do this in a single (but perhaps less human-readable) line of code:

```
companiesOrdered <- companiesData[order(
companiesData$margin),]
```

If you forget that comma after the new order for your rows you'll get an error, because R needs to know what columns to return. Once again, a comma followed by nothing defaults to "all columns" but you can also specify just certain columns like:

```
companiesOrdered <- companiesData[order(
companiesData$margin),c("fy", "company")]
```

To sort in descending order, you'd want companyOrder to have a minus sign before the ordering column:

```
companyOrder <- order(-companiesData$margin)
```

And then:

```
companiesOrdered <- companiesData[companyOrder,]
```

You can put that together in a single statement as:

```
companiesOrdered <-
companiesData[order(-companiesData$margin),]
```

	fy	company	revenue	profit	margin
8	2011	Microsoft	69943	23150	33.1
7	2010	Microsoft	62484	18760	30.0
4	2010	Google	29321	8505	29.0
3	2012	Apple	156508	41733	26.7
5	2011	Google	37905	9737	25.7
2	2011	Apple	108249	25922	23.9
9	2012	Microsoft	73723	16978	23.0
1	2010	Apple	65225	14013	21.5
6	2012	Google	50175	10737	21.4

Note how you can see the original row numbers reordered at the far left.

If you'd like to sort one column ascending and another column descending, just put a minus sign before the one that's descending. This is one way to sort this data first by year (ascending) and then by profit margin (descending) to see which company had the top profit margin by year:

```
companiesData[order(companiesData$fy,
-companiesData$margin),]
```

If you don't want to keep typing the name of the data frame followed by the dollar sign for each of the column names, R's with() function takes the name of a data frame as the first argument and then lets you leave it off in subsequent arguments in one command:

```
companiesOrdered <- companiesData[with(companiesData,
order(fy, -margin)),]
```

While this does save typing, it can make your code somewhat less readable, especially for less experienced R users.

Packages offer some more elegant sorting options. The doBy package features orderBy() using the syntax

```
orderBy(~columnName + secondColumnName,
data=dataFrameName)
```

The ~ at the beginning just means "by" (as in "order by this"). If you want to order by descending, just put a minus sign after the tilde and before the column name. This also orders the data frame:

```
companiesOrdered <- orderBy(~-margin, companiesData)
```

Both plyr and dplyr have an arrange() function with the syntax

```
arrange(dataFrameName, columnName, secondColumnName)
```

To sort descending, use desc(columnName))

```
companiesOrdered <- arrange(companiesData,
desc(margin))
```

Reshaping: Wide to long (and back)

Different analysis tools in R — including some graphing packages — require data in specific formats. One of the most common — and important — tasks in R data manipulation is switching between "wide" and "long" formats in order to use a desired analysis or graphics function. For example, it is usually easier to visualize data using the popular ggplot2() graphing package if it's in long format. Wide means that you've got multiple *measurement columns* across each row, like we've got here:

	fy	company	revenue	profit	margin
1	2010	Apple	65225	14013	21.5
2	2011	Apple	108249	25922	23.9
3	2012	Apple	156508	41733	26.7
4	2010	Google	29321	8505	29.0
5	2011	Google	37905	9737	25.7
6	2012	Google	50175	10737	21.4
7	2010	Microsoft	62484	18760	30.0
8	2011	Microsoft	69943	23150	33.1
9	2012	Microsoft	73723	16978	23.0

Each row includes a column for revenue, for profit and, after some calculations above, profit margin.

Long means that *there's only one measurement per row.* but likely multiple *categories*, as you see below:

	fy	company	variable	value
1	2010	Apple	revenue	65225.0
2	2011	Apple	revenue	108249.0
3	2012	Apple	revenue	156508.0
4	2010	Google	revenue	29321.0
5	2011	Google	revenue	37905.0
6	2012	Google	revenue	50175.0
7	2010	Microsoft	revenue	62484.0
8	2011	Microsoft	revenue	69943.0
9	2012	Microsoft	revenue	73723.0
10	2010	Apple	profit	14013.0
11	2011	Apple	profit	25922.0
12	2012	Apple	profit	41733.0
13	2010	Google	profit	8505.0
14	2011	Google	profit	9737.0
15	2012	Google	profit	10737.0
16	2010	Microsoft	profit	18760.0
17	2011	Microsoft	profit	23150.0
18	2012	Microsoft	profit	16978.0
19	2010	Apple	margin	21.5
20	2011	Apple	margin	23.9
21	2012	Apple	margin	26.7
22	2010	Google	margin	29.0
23	2011	Google	margin	25.7
24	2012	Google	margin	21.4
25	2010	Microsoft	margin	30.0
26	2011	Microsoft	margin	33.1
27	2012	Microsoft	margin	23.0

Please trust me on this (I discovered it the hard way): Once you thoroughly understand the *concept* of wide to long, actually *doing* it in R becomes much easier.

If you find it confusing to figure out what's a category and what's a measurement, here's some advice: Don't pay too much attention to definitions that say long data frames should contain only one "value" in each row. Why? For people with experience programming in other languages, pretty much everything seems like a "value." If the year equals 2011 and the company equals Google, isn't 2011 your value for year and Google your value for company?

For data reshaping, though, the term "value" is being used a bit differently.

I like to think of a "long" data frame as having only one *"measurement that would make sense to plot on its own"* per row. In the case of these financial results, would it make sense to plot that the year changed from 2010 to 2011 to 2012? No, because the year is *a category I set up in advance to decide what measurements I want to look at.*

Even if I'd broken down the financial results by quarter — and quarters 1, 2, 3 and 4 certainly look like numbers and thus "values" — it wouldn't make sense to plot the quarter changing from 1 to 2 to 3 to 4 and back again as a "value" on its own. Quarter is a *category* — a factor in R — that you might want to *group data by*. However, it's not a measurement you would want to *plot by itself.*

This may be more apparent in the world of scientific experimentation. If you're testing a new cholesterol drug, for example, the categories you set up in advance might look at patients by age, gender and whether they're given the drug or a placebo. The measurements (or calculations resulting from those measurements) are your results: Changes in overall cholesterol level, LDL and HDL, for example. But whatever your data, you should have at least one category and one measurement if you want to create a long data frame.

In the example data we've been using here, my categories are fy and company, while my measurements are revenue, profit and margin.

And now here's the next concept you need to understand about reshaping from wide to long: Because you want only one measurement in each row, *you need to add a column that says which type of measurement each value is.*

In my existing wide format, the column headers tell me the measurement type: revenue, profit or margin. But since

I'm rearranging this to only have *one* of those numbers in each row, not three, I'll add a column to show which measurement it is.

I think an example will make this a lot clearer. Here's one "wide" row:

fy	company	revenue	profit	margin
2010	Apple	65225	14013	21.48409

And here's how to have only one measurement per row — by creating three "long" rows:

fy	company	financialCategory	value
2010	Apple	revenue	65225
2010	Apple	profit	14013
2010	Apple	margin	21.5

The column financialCategory now tells me what type of measurement each value is. And now, the term "value" should make more sense.

At last we're ready for some code to reshape a data frame from wide to long! As with pretty much everything in R, there are multiple ways to perform this task. To use reshape2, first you need to install the package if you never have, with:

```
install.packages("reshape2")
```

Load it with:

```
library(reshape2)
```

And then use reshape2's melt() function. melt() uses the following format to assign results to a variable named longData:

longData <- melt(your original data frame, a vector of your category variables)

That's all melt() requires: The name of your data frame and the names of your category variables. However, you can optionally add several other variables, including a vector of your measurement variables (if you don't, melt() assumes that all the rest of the columns are measurement columns) and the name you want your new category column to have.

So, again using the data frame of sample data, wide-to-long code can simply be:

```
companiesLong <- melt(companiesData, c("fy",
"company"))
```

This produces:

	fy	company	variable	value
1	2010	Apple	revenue	65225.0
2	2011	Apple	revenue	108249.0
3	2012	Apple	revenue	156508.0
4	2010	Google	revenue	29321.0
5	2011	Google	revenue	37905.0
6	2012	Google	revenue	50175.0
7	2010	Microsoft	revenue	62484.0
8	2011	Microsoft	revenue	69943.0
9	2012	Microsoft	revenue	73723.0
10	2010	Apple	profit	14013.0
11	2011	Apple	profit	25922.0
12	2012	Apple	profit	41733.0
13	2010	Google	profit	8505.0
14	2011	Google	profit	9737.0
15	2012	Google	profit	10737.0
16	2010	Microsoft	profit	18760.0
17	2011	Microsoft	profit	23150.0
18	2012	Microsoft	profit	16978.0
19	2010	Apple	margin	21.5
20	2011	Apple	margin	23.9
21	2012	Apple	margin	26.7
22	2010	Google	margin	29.0
23	2011	Google	margin	25.7
24	2012	Google	margin	21.4
25	2010	Microsoft	margin	30.0
26	2011	Microsoft	margin	33.1
27	2012	Microsoft	margin	23.0

It's actually fairly simple after you understand the basic concept. Here, the code assumes that all the *other* columns except fy and company are measurements — items you might want to plot.

You can be lengthier in your code if you prefer, especially if you think that will help you remember what you did down the road. The statement below lists *all* the columns in the data frame, assigning them to either id.vars or measure.vars, and also changes the new column names from the default "variable" and "value."

I find it a bit confusing that reshape2 calls category variables "id.vars" (short for ID variables) and not categories or factors, but after a while you'll likely get used to that. Measurement variables in reshape2 are somewhat more intuitively called measure.vars.

```
companiesLong <- melt(companiesData,
id.vars=c("fy", "company"),
measure.vars=c("revenue", "profit", "margin"),
variable.name="financialCategory", value.name="amount")
```

This produces:

	fy	company	financialCategory	amount
1	2010	Apple	revenue	65225.0
2	2011	Apple	revenue	108249.0
3	2012	Apple	revenue	156508.0
4	2010	Google	revenue	29321.0
5	2011	Google	revenue	37905.0
6	2012	Google	revenue	50175.0
7	2010	Microsoft	revenue	62484.0
8	2011	Microsoft	revenue	69943.0
9	2012	Microsoft	revenue	73723.0
10	2010	Apple	profit	14013.0
11	2011	Apple	profit	25922.0
12	2012	Apple	profit	41733.0
13	2010	Google	profit	8505.0
14	2011	Google	profit	9737.0
15	2012	Google	profit	10737.0
16	2010	Microsoft	profit	18760.0
17	2011	Microsoft	profit	23150.0
18	2012	Microsoft	profit	16978.0
19	2010	Apple	margin	21.5
20	2011	Apple	margin	23.9

21	2012	Apple	margin	26.7
22	2010	Google	margin	29.0
23	2011	Google	margin	25.7
24	2012	Google	margin	21.4
25	2010	Microsoft	margin	30.0
26	2011	Microsoft	margin	33.1
27	2012	Microsoft	margin	23.0

Reshaping: Long to wide

Once your data frame is "melted," it can be "cast" into any shape you want. reshape2's dcast() function takes a "long" data frame as input and allows you to create a reshaped data frame in return. (The somewhat similar acast() function can return an array, vector or matrix.) One of the best explanations I've seen on going from long to wide with dcast() is from the *R Graphics Cookbook* by Winston Chang:

> "[S]pecify the ID variables (those that remain in columns) and the *variable* variables (those that get 'moved to the top'). This is done with a formula where the ID variables are before the tilde (~) and the variable variables are after it."

In other words, think briefly about the structure you want to create. The variables you want repeating in each row are your "ID variables." Those that should become column headers are your "variable variables."

Look at this row from the original, "wide" version of our table:

fy	company	revenue	profit	margin
2010	Apple	65225	14013	21.5

Everything following fiscal year and company is *a measurement relating to that specific year and company*. That's why fy and company are the ID variables; revenue, profit and margin are the "variable variables" that have been "moved to the top" as column headers.

How to re-create a wide data frame from the long version of the data? Here's code, if you've got two columns with ID variables and one column with variable variables:

wideDataFrame <- dcast(longDataFrame, idVariableColumn1 + idVariableColumn2 ~ variableColumn, value.var="Name of column with the measurement values")

dcast() takes the name of a long data frame as the first argument. You need to create a formula of sorts as the second argument with the syntax:

```
id variables ~ variable variables
```

The id and measurement variables are separated by a tilde, and if there are more than one on either side of the tilde they are listed with a plus sign between them.

The third argument for dcast() assigns the name of the column that holds your measurement values to value.var.

So, to produce the original, wide data frame from companiesLong using dcast():

```
companiesWide <- dcast(companiesLong, fy + company
~ financialCategory, value.var="amount")
```

To break that down piece by piece: companiesLong is the name of my long data frame; fy and company are the columns I want to *remain* as items in each row of my new, wide data frame; I want to create a *new column* for each of the different categories in the financialCategory column — move them up to the top to become column headers, as Chang said; and I want the actual measurements for each of those financial categories to come from the amount column.

Note: Hadley Wickham created the tidyr package to perform a subset of reshape2's capabilities with two main functions: gather() to take multiple values and turn them into key-value pairs and spread() to go from long to wide. I still use reshape2 for these tasks, but you may find tidyr better fits your needs.

dplyr basics

The goal of dplyr is to offer a fairly easy, rational data manipulation. Creator Hadley Wickham talks about just a handful of basic, core things you want to do when manipulating data:

- To choose only certain observations or rows by 1 or more criteria: filter()
- To choose only certain variables or columns: select()
- To sort: arrange()

- To add new columns: mutate()
- To summarize or otherwise analyze by subgroups: group_by() and summarise()
- To apply a function to data by subgroups: group_by() and do()

There are other useful functions, such as ranking functions top_n() for the top n items in a group, min_rank() and dense_rank(), lead() and lag().

dplyr creates class of data frame called tbl_df that behaves largely like a data frame but has some convenience functionality, such as not accidentally printing out hundreds of rows if you type its name.

Wickham has a sample data package called nycflights13 for learning dplyr, but I'll use a smaller data file for these examples, a CSV file of domestic flights in and out of Georgia airports, GAontime.csv, available for download on GitHub.

Note that this just contains data for January through November 2014.

```
library(dplyr)
ga <- read.csv("GAontime.csv", stringsAsFactors = FALSE,
header = TRUE)
```

NOTE: read.csv can take a while to process large data files. In a hurry?

Use the data.table package's fread function. data.table has its own object classes and own ecosystem of functions. If you're not planning to use those (I don't), just convert the object back to a data frame or dplyr tbl_df object:

```
ga <- data.table::fread("GAontime.csv")
# You can turn this into a dplyr class tbl_df object with
ga <- tbl_df(ga)
# Now see what happens if you just type the variable name
ga
# Look at the structure:
str(ga)
# There's also a dplyr-specific function glimpse() with a
slightly better format
glimpse(ga)
# Let's just get Hartfield data. We want to filter for
either ORIGIN or DEST being Hartsfield with code ATL
atlanta <- filter(ga, ORIGIN == "ATL" | DEST == "ATL")
```

Now there are all sorts of questions we can answer with this data.

What's the average, median and longest delay for flights to a specific place *by carrier*? I'll use Boston's Logan Airport:

```
bosdelays1 <- atlanta %>%
  filter(DEST == "BOS") %>%
  group_by(CARRIER) %>%
  summarise(
    avgdelay = mean(DEP_DELAY, na.rm = TRUE),
    mediandelay = median(DEP_DELAY, na.rm = TRUE),
    maxdelay = max(DEP_DELAY, na.rm = TRUE)
    )

bosdelays1
```

```
# Or just the average delay by airline to Boston?

avg_delays <- atlanta %>%
  filter(DEST == "BOS") %>%
  group_by(CARRIER) %>%
  summarise(avgdelay = mean(DEP_DELAY, na.rm=TRUE))

avg_delays
```

```
# What's the average delay by airline for each month to a
specific destination?

avg_delays_by_month <- atlanta %>%
  filter(DEST == "BOS") %>%
  group_by(CARRIER, MONTH) %>%
  summarise(avgdelay = round(mean(DEP_DELAY,
na.rm=TRUE),1))

avg_delays_by_month


# Not as easy to see those, let's make a datatable:

data.table(avg_delays_by_month)
```

What were the top 5 longest delays per airline?

```
delays <- atlanta %>%
  select(CARRIER, DEP_DELAY, DEST, FL_NUM, FL_DATE) %>%
# columns I want
  group_by(CARRIER) %>%
  top_n(5, DEP_DELAY) %>%
  arrange(CARRIER, desc(DEP_DELAY))

View(delays)

# Which are the unlucky destinations in those top 5?

table(delays$DEST)

# What were the top 5 longest  delays per destination?

delays2 <- atlanta %>%
  select(CARRIER, DEP_DELAY, DEST, FL_NUM, FL_DATE) %>%
# columns I want
  group_by(DEST) %>%
  top_n(5, DEP_DELAY) %>%
  arrange(CARRIER, desc(DEP_DELAY))

View(delays2)
```

ggplot2 101

There's a reason ggplot2 is one of the most popular add-on packages for R: It's a powerful, flexible and well-thought-out platform to create data visualizations you can customize to your heart's content.

But it also can be a bit overwhelming. While I find the logic of plot *layers* to be intuitive, some of the *syntax* can be a bit of a challenge. Unless you do a lot of work in ggplot2, I'm not sure how easy it is to remember that, for example, the simple task of "make my graph title bold" requires the rather wordy `theme(plot.title = element_text(face = "bold"))`.

What follows is a short, highly simplified guide to visualizing data with ggplot2 along with a table of commands for a lot of basic, useful tasks.

There's a visualization philosophy behind ggplot2 called the "Grammar of Graphics" (that's where the gg in ggplot2 comes from) to describe various components of a graphic. Here I'll focus primarily on what code you need to build a few basic visualizations layer by layer.

Layer 1 defines which variables are going to do what. And *that's all*. It's *mapping* things like what data frame variable holds your data and which column will be on your x and y axes.

Here's an important point about the first layer: When you use a property like color or size as an "aesthetic property" (aes) in this first layer, *you are not setting a specific color or a specific size*. You are saying something like "I want the color of my points to *change* based on the values of this column" and NOT "Make the colors of my points *the specific color* light blue." Picking your color(s) comes later.

A first layer might look something like this:

```
myplot <- ggplot(mydf, aes(x="colname1",
y="colname2", color="colname3")
```

That says: Create a plot using data in mydf and use the following "aesthetics": Set the x axis to colname1 values in mydf, set the y axis to colname2 values in mydf and use different colors depending on the values in mydf colname3.

What layer 1 *doesn't* do is say what kind of visualization you want: scatterplot, bar graph, histogram, etc. For that

you need layer 2: a geometry, or geom in gpplot-speak. You need these first two layers before R will actually show anything (you can add lots more layers for customizing the graph, but two is your minimum). You add a layer with, intuitively, the + symbol. Since we already stored layer 1 in myplot, we can add layer 2 for a scatterplot with:

```
myplot <- myplot + geom_point()
```

Make sure your plus sign is on the same line as the new layer. Your layers can either be all on one line, or code after the plus can be on a new line, such as:

```
ggplot(mydf, aes(x="colname1")) +
geom_histogram()
```

Don't put the plus sign and new layer on a new line like this

```
ggplot(mydf, aes(x="colname1"))
+ geom_histogram()
```

because R will think that first line is complete and not understand that line 2 goes with line 1.

There are a whole host of customizations you can do beyond this. For more on ggplot2 layers, see ggplot2 creator Hadley Wickham's Build a plot layer by layer.

Command cheat sheet

Below is a cheat sheet, easily searchable by task (using the search function of your PDF reader of choice), to see how to create and modify plots — everything from generating basic bar charts and line graphs to customizing colors and automatically adding annotations.

I've also created RStudio code snippets for several dozen of these tasks, so you don't even have to copy and paste — or re-type — these commands if you use RStudio. Instead, download my free ggplot2 code snippets as part of your Computerworld Insider registration.

Cheat sheet for useful ggplot2 tasks

TASK	PLOT TYPE	FORMAT	NOTE
Create basic plot object that will display something	Any	ggplot(data=mydf, aes(x=myxcolname, y=myycolname))	data=mydf sets the overall source of your data; it must be a data frame. aes(x=colname1, y=colname2) sets which variables are mapped to the x and y axes. A geom layer must be added to this object in order for anything to display, such as + geom_point() or geom_line().
Create basic scatterplot	Scatterplot	+ geom_point()	This is added to the basic ggplot object. Need (continuous) numerical data on both axes. aes properties of ggplot you can assign include x data, y data, and mapping color, shape or size to the value of a variable column. To set the specific color of points, use the color property of geom_point, not aes. Aesthetics are mappings.
Set size of points	Scatterplot, points on line graph and others	+ geom_point(size=mynumber)	Larger numbers make larger points.
Solve scatterplot issue of too many points exactly on top of each other	Scatterplot	+ geom_point(position = "jitter")	Change the amount of jitter with geom_jitter(position = position_jitter(width = mynumber)).
Set shape of points to be all one shape	Scatterplot, points on line graph and others	+ geom_point(shape=mynumber)	See chart of available shapes.
Set shape of points based on category	Scatterplot, points on line graph and others	+ geom_point(aes(shape=mycategory)) + scale_shape_manual(values=myshapevector)	mycategory needs to be a categorical variable. See chart of available shapes.
Create basic line graph	Line graph	+ geom_line()	This is added to the basic ggplot object.
Create line graph with lines of different colors by category	Line graph	+ geom_line(aes(color=mycategory))	
Set color of points or lines to be one color	Scatterplot, line graph and others	+ geom_mychoice(color="mycolor")	Unlike with bars, here the color property sets the main color of the item.
Set color of points based on a specific category	Any	ggplot(mydf, aes(x=myxcolname, y=myycolname, color=mygroupingcol)) + geom_mychoice()	Default colors will be selected.

TASK	PLOT TYPE	FORMAT	NOTE
Set color of scatterplot points by numeric data values - define your own palette	Scatterplot	+ geom_point(aes(color=mygroupingvariable)) + scale_color_gradient(low="mylowcolor", high="myhighcolor")	Continuous numeric variable needed for grouping-by-color variable when using scale_color_gradient. There are other variations with a midpoint color, specific numbers of colors and more. See docs for scale_color_gradient and scale_fill_gradient.
Set color of scatterplot points by categorical data values - use RColorBrewer	Scatterplot	+ geom_point(aes(color =mygroupingvariable)) + scale_color_brewer(type="seq", palette="mypalettechoice")	Color grouping variable needs to be categorical/discrete, not continuous. Type can be sequential or diverging; palettes can be names or numbers. See documentation.
Set type of line	Line graph and others with lines	+ geom_line(linetype="mylinetype")	Available line types include solid, dashed, dotted, dotdash, longdash and twodash.
Set width of line	Line graph and others with lines	+ geom_line(size=mysizenumber)	
Set color of line	Line graph and others with lines	+ geom_line(color="mycolor")	Color can be a color name available in R like "lightblue" or a hex value like "#0072B2". Run colors() in base R to see all available color names.
Create basic bar graph	Bar	+ geom_bar(stat="identity")	This is added to the basic ggplot object. Need categorial data for x axis. stat="identity" uses values in a y column for the y axis. Without this, the graph will show counts of each value on the x axis.
Create basic bar graph with y axis showing count of items in x axis	Bar	+ geom_bar()	This is added to the basic ggplot object. Only an x value is needed because this default counts number of records for each x category.
Reorder x axis based on y column values in descending order	Bar, boxplots and others	ggplot(data = mydf, aes(x=reorder(myxcolname, -myycolname), y=myycolname)) + geom_mychoice()	Needs categorical data on x axis and numerical data on y axis. Remove the - before the y column name if you want ascending order. A geom such as geom_bar() or geom_boxplot() must be added.
Create bar graph grouped by category (grouped bar)	Bar	ggplot(mydf, aes(x=myxcolname, y=myycolname, fill=mygroupcolname))+ geom_bar(stat="identity", position="dodge")	Without position="dodge", a stacked barchart is created

TASK	PLOT TYPE	FORMAT	NOTE
Set fill color of bars (or other 2D items in graphs) to be all one specific color	Bar, histogram and others	+ geom_mychoice(fill="mycolor")	
for bar graph: + geom_ bar(fill="mycolor, stat="identity")	Color can be a color name available in R like "lightblue" or a hex value like "#0072B2". Run colors() in base R to see all available color names. There's a PDF showing R colors here; demo(colors) shows some in your R session.		
Set outline color of 2D graph items such as bars	Bar, histogram and others	+ geom_mychoice(color="mycolor")	This can be confusing since "color" is not the main item color but its outline. As with fill, the color can be a color name available in R like "lightblue" or a hex value like "#0072B2".
Create a bar graph that will color each bar a different color	Bar	ggplot(mydf, aes(x=myxcolname, y=myycolname, fill=myxcolname)) + geom_bar(stat="identity")	
Customize colors for bar graph with different color for each bar - define your own palette	Bar	+ scale_fill_manual(values=c("mycolor1", "mycolor2", "mycolor3"))	
Customize colors in a bar graph where colors have been defined to change by a category - use RColorBrewer	Bar	+ scale_fill_brewer(palette="mycolorbrewerpalettename")	See available RColorBrewer palettes with display.brewer.all(n=10, exact.n=FALSE). RColorBrewer package must be loaded with library(RColorBrewer).
Create basic histogram	Histogram	ggplot(data=mydf, aes(x=myxcolname)) + geom_histogram()	
Change bin width of histogram	Histogram	+ geom_histogram(binwidth=mynumber)	This sets the width of the bin, not the number of bins.
Set color of histogram bars to one color	Histogram	+ geom_histogram(fill="mycolor")	

TASK	PLOT TYPE	FORMAT	NOTE
Add horizontal line to any type of graph at a specific position	Any	+ geom_hline(yintercept=mynumber)	Set color with color argument, width with size arg and type with linetype, such as geom_hline(yintercept=100, color="red", size=2, linetype="dashed").
Add vertical line to any type of graph at a specific position	Any	+ geom_vline(xintercept=mynumber)	With categories on x axis, intercept 3 means the 3rd item on the axis. Set color with color arg, width with size arg and type with linetype, such as geom_hline(yintercept=100, color="red", size=2, linetype="dashed").
Add regression line (line of best fit) to scatterplot	Scatterplot	+ stat_smooth(method=lm, level=FALSE)	lm stands for linear model. Change default color by adding color property in stat_smooth
Add regression line (line of best fit) with 95% confidence interval to scatterplot	Scatterplot	+ stat_smooth(method=lm, level=0.95)	lm stands for linear model.
Use an already-made alternate theme for graph	Any	+ theme_mychoice()	Available themes include theme_gray, theme_bw, theme_classic and theme_minimal. If you are customizing a pre-made theme, make sure to add that code after calling the initial theme_mychoice() function.
Add title (headline)	Any	+ ggtitle("My headline text")	
Change headline size	Any	+ theme(plot.title = element_text(size = myinteger))	+ theme(plot.title = element_text(size = rel(myinteger))) sets the headline size relative to the plot's base font.
Change headline color	Any	+ theme(plot.title = element_text(color = "mycolor"))	
Make plot headline bold	Any	+ theme(plot.title = element_text(face = "bold"))	Also works for face = "italic" or "bold.italic"
Change x-axis title	Any	+ xlab("My x-axis title text")	
Change y-axis title	Any	+ ylab("My y-axis title text")	
Change value labels along the x axis for categorical variables	Any	+ scale_x_discrete(labels=myvectoroflabels)	

TASK	PLOT TYPE	FORMAT	NOTE
Change value labels along the y axis for continuous numerical variable	Any	+ scale_y_continuous(breaks=myvectorofbreaks)	scale_x_continuous works similarly for the x axis. A vector of breaks could look something like c(0,25,50,75,100) or seq(0,100,25).
Set y-axis minimum and maximum values	Any	+ ylim(mymin, mymax)	xlim works the same for the x axis. If there are values outside your defined limits, they won't display, so you can use this to statically zoom in on a portion of your dataviz.
Rotate x-axis value labels	Any	+ theme(axis.text.x= element_text(angle=myrotationAngle, hjust=myOptionalTweak, vjust=myOptionalTweak2))	rotation angle should be between 1 and 359, such as theme(axis.text.x= element_text(angle=45, hjust=1)). hjust and vjust can be needed to position the text properly with the axis. I often use + theme(axis.text.x= element_text(angle=45, hjust = 1.3, vjust = 1.2)) as settings.
Rotate y-axis title to be horizontal (parallel to x axis)	Any	+ theme(axis.title.y = element_text(angle = 0))	angle can take different values to rotate y-axis text in other ways.
Turn off automatic legend	Any	+ theme(legend.position = "none")	
Change order of legend items	Any	mydf$mylegendcolumnNew <- factor(mydf$mylegendcolumn, levels=c(myOrderedVectorOfItems), ordered = TRUE)	While there are ways to do this in ggplot2, if order matters to you, create a variable ordered as you want in R.
Change legend title font size	Any	+ theme(legend.title = element_text(size=mypointsize))	
Change legend labels size	Any	+ theme(legend.text = element_text(size=mypointsize))	

TASK	PLOT TYPE	FORMAT	NOTE
Create multiple plots based on one or two variables in your data	Any	+ facet_grid(mycolname1 ~ mycolname2)	Once you've set up an initial plot using one or more variables, this facet_grid "formula" plots a grid of all possible permutations of additional variables mycolname1 by mycolname2, with mycolname1 in the rows and mycolname2 in the columns. Example: You set up a basic plot of online sales transactions by hour of day, and then make a facet_grid of all such transactions subsetted by category of merchandise and whether customers were new or returning. To use facet_grid for only 1 variable, use a dot for the other one, such as facet_grid(. ~ mycolname1).
Create multiple plots based on one or two variables in your data	Any	+ facet_wrap(mycolname1 ~ mycolname2, ncol=myinteger)	Similar to facet_grid above but you can manually set number of columns or number of rows in your grid with ncol or nrow, and only those permutations with available values will be plotted. + facet_wrap(~ mycolname1) to facet by one variable, then set nrow or ncol.
Put multiple plots from different data on one page - gridExtra package	Any	grid.arrange(plot1, plot2, plot3..., ncol=mynumberofcolumns)	Any number of plots can be entered, separated by a comma. ncol defaults to 1. gridExtra package must be installed and loaded.
Add text annotations to a plot by x,y position on plot	Any	+ annotate("text", x=myxposition, y=myyposition, label="My text")	There are other options for annotate besides "text" such as "rect" for rectangle with properties xmin, xmax, ymin, ymax and alpha (transparency) and optional color (border) and fill (fill color).
Create and auto annotate scatterplot grouped by color - directlabels package	Scatterplot	myplot <- ggplot(mydf, aes(x=myxcolname, y=myycolname, color=mygroupingcol)) + geom_point()	
direct.label(myplot, "smart.grid")	directlabels package must be installed and loaded.		
Create and auto annotate line graph where lines are different colors by category	Line graph	myplot <- ggplot(mydf, aes(x=myxcolname, y=myycolname, color=mygroupingcol)) + geom_line()	

TASK	PLOT TYPE	FORMAT	NOTE
direct.label(myplot, list(last.points, hjust = 0.7, vjust = 1))	directlabels package must be installed and loaded. first.points is another option to label at start of line instead of end.		
Save plot	Any	ggsave(filename="myname.ext")	ggsave defaults to the most recent plot, but you can set a different plot with ggsave (filename="myname.ext", plot=myplot). File extension determines type of file created — .pdf, .png and so on. Set width and height in inches with width and height arguments.

Great R packages for data import, wrangling and visualization

One of the best things about R is the thousands of packages users have written to solve specific problems in various disciplines — analyzing everything from weather or financial data to the human genome — not to mention analyzing computer security-breach data.

Some tasks are common to almost all users, though, regardless of subject area: Data import, data wrangling and data visualization. The table below shows my favorite go-to packages for one of these three tasks (plus a few miscellaneous ones tossed in). The package names in the table are clickable if you want more information. To find out more about a package once you've installed it, type `help(package = "packagename")` in your R console (of course substituting the actual package name).

Useful R packages for data visualization and munging

PACKAGE	CATEGORY	DESCRIPTION	SAMPLE USE	AUTHOR
devtools	package development, package installation	While devtools is aimed at helping you create your own R packages, it's also essential if you want to easily install other packages from GitHub. Install it! Requires Rtools on Windows and XCode on a Mac. On CRAN.	install_github("rstudio/leaflet")	Hadley Wickham & others

PACKAGE	CATEGORY	DESCRIPTION	SAMPLE USE	AUTHOR
installr	misc	Windows only: Update your installed version of R from within R. On CRAN.	updateR()	Tal Galili & others
readxl	data import	Fast way to read Excel files in R, without dependencies such as Java. CRAN.	read_excel("my-spreadsheet.xls", sheet = 1)	Hadley Wickham
googlesheets	data import, data export	Easily read data into R from Google Sheets. CRAN.	mysheet <- gs_ title("Google Spread-sheet Title")	
mydata <- mydata <- gs_ read(mysheet, ws = "WorksheetTitle")	Jennifer Bryan			
RMySQL	data import	Read data from a MySQL database into R. There are similar packages for other databases. CRAN.	con <- dbConnect (RMySQL::MySQL (), group = "my-db")	
myresults <- dbSendQuery(con, "SELECT * FROM mytable")	Jeroen Ooms & others			
readr	data import	Base R handles most of these functions; but if you have huge files, this is a speedy and standardized way to read tabular files such as CSVs into R data frames, as well as plain text files into character strings with read_file. CRAN.	read_csv(myfile.csv)	Hadley Wickham
rio	data import, data export	rio has a good idea: Pull a lot of separate data-reading packages into one, so you just need to remember 2 functions: import and export. CRAN.	import("myfile")	Thomas J. Leeper & others
psych	data analysis	No, I'm not using the functions that analyze personalitydata; but I do regularly use the describe and describeBy functions to summarize data sets, as well as read.clipboard to get data I've copied into R. CRAN.	describe(mydf)	William Revelle
sqldf	data wrangling, data analysis	Do you know a great SQL query you'd use if your R data frame were in a SQL database? Run SQL queries on your data frame with sqldf. CRAN.	sqldf("select * from mydf where mycol > 4")	G. Grothendieck

PACKAGE	CATEGORY	DESCRIPTION	SAMPLE USE	AUTHOR
jsonlite	data import, data wrangling	Parse json within R or turn R data frames into json. CRAN.	myjson <- toJSON(mydf, pretty=TRUE)	
mydf2 <- fromJSON(myjson)	Jeroen Ooms & others			
XML	data import, data wrangling	Many functions for elegantly dealing with XML and HTML, such as readHTMLTable. CRAN.	mytables <- readHTMLTable (myurl)	Duncan Temple Lang
quantmod	data import, data visualization, data analysis	Even if you're not interested in analyzing and graphing financial investment data, quantmod has easy-to-use functions for importing economic as well as financial data from sources like the Federal Reserve. CRAN.	getSymbols("AITINO", src="FRED")	Jeffrey A. Ryan
rvest	data import, web scraping	Web scraping: Extract data from HTML pages. Inspired by Python's Beautiful Soup. Works well with Selectorgadget. CRAN.	See the package vignette	Hadley Wickham
dplyr	data wrangling, data analysis	The essential data-munging R package when working with data frames. Especially useful for operating on data by categories. CRAN.	See the intro vignette	Hadley Wickham
plyr	data wrangling	While dplyr is my go-to package for wrangling data frames, the older plyr package still comes in handy when working with other types of R data such as lists. CRAN.	llply(mylist, myfunction)	Hadley Wickham
reshape2	data wrangling	Change data row and column formats from "wide" to "long"; turn variables into column names or column names into variables and more. The tidyr package is a newer, more focused option, but I still use reshape2. CRAN.	See my tutorial	Hadley Wickham
tidyr	data wrangling	While I still prefer reshape2 for general re-arranging, tidy won me over with specialized functions like fill (fill in missing columns from data above) and replace_na. CRAN	See examples in this blog post.	Hadley Wickham

PACKAGE	CATEGORY	DESCRIPTION	SAMPLE USE	AUTHOR
data.table	data wrangling, data analysis	Popular package for heavy-duty data wrangling. While I typically prefer dplyr, data.table has many fans for its speed with large data sets. CRAN.	Useful tutorial	Matt Dowle & others
stringr	data wrangling	Numerous functions for text manipulation. Some are similar to existing base R functions but in a more standard format, including working with regular expressions. Some of my favorites: str_pad and str_trim. CRAN.	str_pad (myzipcodevector, 5, "left", "0")	Hadley Wickham
lubridate	data wrangling	Everything you ever wanted to do with date arithmetic, although understanding & using available functionality can be somewhat complex. CRAN.	mdy("05/06/2015") + months(1)	
More examples in the package vignette	Garrett Grolemund, Hadley Wickham & others			
zoo	data wrangling, data analysis	Robust package with a slew of functions for dealing with time series data; I like the handy rollmean function with its align=right and fill=NA options for calculating moving averages. CRAN.	rollmean(mydf, 7)	Achim Zeileis & others
editR	data display	Interactive editor for R Markdown documents. On GitHub at swarm-lab/editR.	editR("path/to/myfile.Rmd")	Simon Garnier
knitr	data display	Add R to a markdown document and easily generate reports in HTML, Word and other formats. A must-have if you're interested in reproducible research and automating the journey from data analysis to report creation. CRAN.	This tutorial is a few years old but goes over some basics	Yihui Xie & others
listviewer	data display, data wrangling	Elegant way to view complex nested lists within R. GitHub timelyportfolio/listviewer.	jsonedit(mylist)	Kent Russell
DT	data display	Create a sortable, searchable table in one line of code with this R interface to the jQuery DataTables plug-in. GitHub rstudio/DT.	datatable(mydf)	RStudio

PACKAGE	CATEGORY	DESCRIPTION	SAMPLE USE	AUTHOR
ggplot2	data visualization	Powerful, flexible and well-thought-out dataviz package following 'grammar of graphics' syntax to create static graphics, but be prepared for a steep learning curve. CRAN.	qplot(factor (myfactor), data =mydf, geom="bar", fill=factor(myfactor))	
See my searchable ggplot2 cheat sheet and				
time-saving code snippets.	Hadley Wickham			
dygraphs	data visualization	Create HTML/JavaScript graphs of time series - one-line command if your data is an xts object. CRAN.	dygraph (myxtsobject)	JJ Allaire & RStudio
googleVis	data visualization	Tap into the Google Charts API using R. CRAN.	mychart <- gvis ColumnChart (mydata)	
plot(Column)				
Numerous examples here	Markus Gesmann & others			
metricsgraphics	data visualization	R interface to the metrics-graphics JavaScript library for bare-bones line, scatter-plot and bar charts. GitHub hrbrmstr/metricsgraphics.	See package intro	Bob Rudis
RColorBrewer	data visualization	Not a designer? RColorBrewer helps you select color palettes for your visualizations. CRAN.	See Jennifer Bryan's tutorial	Erich Neuwirth
plotly	data visualization	This allows you to create interactive JavaScript graphs at the Plotly service, which you can link to or embed in a Web page. Free plotly account required. GitHub ropensci/plotly.	See the documentation examples	rOpenSci project
leaflet	mapping	Map data using the Leaflet JavaScript library within R. GitHub rstudio/leaflet.	See my tutorial	RStudio
choroplethr	mapping	Easy ways to map data with built-in state, county, zip code and country geographic info; you can also import your own shape files. Recent update improved earlier issues with projections. CRAN.	data(df_pop_state)	

PACKAGE	CATEGORY	DESCRIPTION	SAMPLE USE	AUTHOR
state_choropleth(df_pop_state)				
Free email course by pkg author	Ari Lamstein			
tmap	mapping	Not the most polished-looking maps for publication or presentation, but this new package offers a very easy way to read in shape files and join data files with geographic info, as well as do some exploratory mapping. CRAN.	See the package vignette	Martijn Tennekes
fitbitScraper	misc	Import Fitbit data from your account into R. CRAN.	cookie <- login(email="", password="")	
df <- get_daily_data(cookie, what="steps", "2015-01-01", "2015-05-18")	Cory Nisson			
rga	Web analytics	Use Google Analytics with R. GitHub skardhamar/rga.	See package README file and my tutorial	Bror Skardhamar
RSiteCatalyst	Web analytics	Use Adobe Analytics with R. GitHub randyzwitch/RSiteCatalyst.	See intro video	Randy Zwitch
roxygen2	package development	Useful tools for documenting functions within R packages. CRAN.	See this short, easy-to-read blog post	
on writing R packages	Hadley Wickham & others			
shiny	data visualization	Turn R data into interactive Web applications. I haven't used this much yet, but I've seen some nice (if sometimes sluggish) apps and it's got many enthusiasts. CRAN.	See the tutorial	RStudio
openxlsx	misc	If you need to write to an Excel file as well as read, this package is easy to use. CRAN.	write.xlsx(mydf, "myfile.xlsx")	Alexander Walker
gmodels	data wrangling, data analysis	There are several functions for modeling data here, but the one I use, CrossTable, simply creates cross-tabs with loads of options -- totals, propotions and several statistical tests. CRAN.	CrossTable (myxvector, myyvector, prop.t= FALSE, prop.chisq = FALSE)	Gregory R. Warnes

PACKAGE	CATEGORY	DESCRIPTION	SAMPLE USE	AUTHOR
car	data wrangling	car's recode function makes it easy to bin continuous numerical data into categories or factors. While base R's cut accomplishes the same task, I find recode's syntax to be more intuitive - just remember to put the entire recoding formula within double quotation marks. CRAN.	recode(x, "1:3='Low'; 4:7='Mid'; 8:hi='High'")	John Fox & others
rcdimple	data visualization	R interface to the dimple JavaScript library with numerous customization options. Good choice for JavaScript bar charts, among others. GitHub timelyportfolio/rcdimple.	dimple(mtcars, mpg ~ cyl, type = "bar")	Kent Russell
foreach	data wrangling	Efficient - and intuitive if you come from another programming language - for loops in R. CRAN.	foreach(i=1:3) %do% sqrt(i)	
Also see The Wonders of foreach	Revolution Analytics, Steve Weston			
downloader	data acquisition	Wrapper for base R download function that eases dealing with files over https (although R 3.2.2 solves some of these issues as well). CRAN.	download("https://url.com/filename", "myfilename.zip", mode = "wb")	NA
scales	data wrangling	While this package has many more sophisticated ways to help you format data for graphing, it's worth a download just for the comma(), percent() and dollar() functions. CRAN.	comma(mynumvec)	Hadley Wickham
plotly	data visualization	R interface to the open-source Plotly JavaScript library that was open-sourced in late 2015. Graphs have a distinctive look and a promo for the Plotly site, which may not be for everyone, but it's full-featured, relatively easy to learn (especially if you know ggplot2) and includes an ggplotly() function for graphs created with ggplot2. CRAN.	d <- diamonds [sample(nrow (diamonds), 1000),]	

PACKAGE	CATEGORY	DESCRIPTION	SAMPLE USE	AUTHOR
plot_ly(d, x = carat, y = price, text = paste("Clarity: ", clarity), mode = "markers", color = carat, size = carat)	Carson Sievert & others			

A few important points for newbies:

To install a package from CRAN, use the command `install.packages("packagename")` — of course substituting the actual package name for packagename and putting it in quotation marks. Package names, like pretty much everything else in R, are case sensitive.

To install from GitHub, it's easiest to use the install-github function from the devtools package, using the format `devtools::install_github("githubaccountname/packagename")`. That means you first want to install the devtools package on your system with `install.packages("devtools")`. Note that devtools sometimes needs some extra non-R software on your system — more specifically, an Rtools download for Windows or Xcode for OS X. There's more information about devtools here.

In order to use a package's function during your R session, you need to do one of two things. One option is to load it into your R session with the `library("packagename")` or `require("packagename")`. The other is to call the function including the package name, like this: `packagename::functioname()`. Package names, like pretty much everything else in R, are case sensitive.

Create choropleth maps in R

There are many options for mapping data besides R, of course. If you do this kind of thing often or want to create a map with lots of slick bells and whistles, it could make more sense to learn GIS software like Esri's ArcGIS or open-source QGIS. If you care only about well-used geographic areas such as cities, counties or zip codes, software like Tableau and Microsoft Power BI may have easier interfaces. If you don't mind drag-and-drop tools and having your data in the cloud, there are still more options such as Google Fusion Tables.

But there are also advantages to using R — a language designed for data analysis and visualization. It's open source, which means you don't have to worry about ever losing access to (or paying for) your tools. All your data stays local if you want it to. It's fully command-line scripted end-to-end, making an easily repeatable process in a single platform from data input and re-formatting through final visualization. And, R's mapping options are surprisingly robust.

Ready to code your own election results maps — or any other kind of color-coded choropleth map? Here's how to handle a straightforward two-person race and a more complex race with three or more candidates in R.

We'll be using two mapping packages in this tutorial: tmap for quick static maps and leaflet for interactive maps. You can install and load them now with

```
install.packages("tmap")

install.packages("leaflet")

library("tmap")

library("leaflet")
```

(Skip the install.packages lines for any R packages that are already on your system.)

Step 1: Get election results data

I'll start with the New Hampshire Democratic primary results, which are available from the NH secretary of state's office as a downloadable Excel spreadsheet.

Getting election data into the proper format for mapping is one of this project's biggest challenges — more so than actually creating the map. For simplicity, let's stick to results by county instead of drilling down to individual towns and precincts.

One common problem: **Results data need to have one column with all election district names** — whether counties, precincts or states — and candidate names as column headers. Many election returns, though, are reported with each election district *in its own column* and candidate results by row.

That's the case with the official NH results. I transposed the data to fix that and otherwise cleaned up the spreadsheet a bit before importing it into R (such as removing ", d" after each candidate's name). The first column now has county names, while every additional column is a candidate name; each *row* is a county result. I also got rid of the' total' row at the bottom, which can interfere with data sorting.

You can do the same — or, if you'd like to download the data file and all the other files I'm using, including R code, head to the "Mapping with R" file download page. (Free Insider registration needed. Bonus: You'll be helping me convince my boss that I ought to write more of these types of tutorials). If you download and unzip the mapping with R file, look for NHD2016.xlsx in the zip file.

To make your R mapping script as re-usable as possible, I suggest putting data file names at the top of the script — that makes it easy to swap in different data files without having to hunt through code to find where a file name appears. You can put this toward the top of your R script:

```
datafile <- "data/NHD2016.xlsx"
```

Note: My data file isn't in the same working directory as my R script; I have it in a data subdirectory. Make sure to include the appropriate file path for your system, using forward slashes even on Windows.

There are several packages for importing Excel files into R; but for ease of use, you can't beat rio. Install it with:

`install.packages("rio")` if it's not already on your system, and then run:

```
nhdata <- rio::import(datafile)
```

to store data from the election results spreadsheet into a variable called nhdata.

There were actually *28* candidates in the results; but to focus on mapping instead of data wrangling, let's not worry about the many minor candidates and pretend there were just two: Hillary Clinton and Bernie Sanders. Select just the County, Clinton and Sanders columns with:

```
nhdata <- nhdata[,c("County", "Clinton",
"Sanders")]
```

Step 2: Decide what data to map

Now we need to think about **what exactly we'd like to color-code on the map**. We need to pick one column of data for the map's county colors, but all we have so far is raw vote totals. We probably want to calculate either the winner's overall percent of the vote, the winner's percentage-point margin of victory or, less common, the winner's margin expressed by number of votes (after all, winning by 5 points in a heavily populated county might be more useful than winning by 10 points in a place with way fewer people if the goal is to win the entire state).

It turns out that Sanders won every county; but if he didn't, we could still map the Sanders "margin of victory" and use negative values for counties he lost.

Let's add columns for candidates' margins of victory (or loss) and percent of the vote, again for now pretending there were votes cast only for the two main candidates:

```
nhdata$SandersMarginVotes <- nhdata$Sanders -
nhdata$Clinton
nhdata$SandersPct <- (nhdata$Sanders - nhdata$Clinton) /
(nhdata$Sanders + nhdata$Clinton)
# Will use formatting later to multiply by 100
  nhdata$ClintonPct <- (nhdata$Clinton - nhdata$Sanders) /
(nhdata$Sanders + nhdata$Clinton)
nhdata$SandersMarginPctgPoints <- nhdata$SandersPct -
nhdata$ClintonPct
```

Step 3: Get your geographic data

Whether you're mapping results for your city, your state or the nation, you need geographic data for the area you'll be mapping in addition to election results. There are several common formats for such geospatial data; but for this tutorial, we'll focus on just one: shapefiles, a widely used format developed by Esri.

If you want to map results down to your city or town's precinct level, you'll probably need to get files from a local or state GIS office. For mapping by larger areas like cities, counties or states, the Census Bureau is a good place to find shapefiles.

For this New Hampshire mapping project by county, I downloaded files from the Cartographic Boundary shapefiles page — these are smaller, simplified files designed for mapping projects where extraordinarily precise boundaries aren't needed. (Files for engineering projects or redistricting tend to be considerably larger).

I chose the national county file at >http://www2.census.gov/geo/tiger/GENZ2014/shp/cb_2014_us_county_5m.zip and unzipped it within my data subdirectory. With R, it's easy to create a subset for just one state, or more; and now I've got a file I can re-use for other state maps by county as well.

There are a lot of files in that newly unzipped subdirectory; the one you want has the .shp extension. I'll store the name of this file in a variable called usshapefile:

```
usshapefile <- "data/cb_2014_us_county_5m/cb_2014_
us_county_5m.shp"
```

Several R packages have functions for importing shapefiles into R. I'll use tmap's read_shape(), which I find quite intuitive:

```
usgeo <- read_shape(file=usshapefile)
```

If you want to check to see if the usgeo object looks like geography of the U.S., run tmap's quick thematic map command: `qtm(usgeo)`. This may take a while to load and appear small and rather boring, but if you've got a map of the U.S. with divisions, you're probably on the right track.

If you run `str(usgeo)` to see the data structure within usgeo, it will look pretty unusual if you haven't done GIS in R before. usgeo contains a LOT of data, including columns starting with @ as well as more familiar entries starting with $. If you're interested in the ins and outs of this type of geospatial object, known as a SpatialPolygonsDataFrame, see Robin Lovelace's excellent *Creating maps in R* tutorial, especially the section on "The structure of spatial data in R."

For this tutorial, we're interested in what's in usgeo@data — the object's "data slot." (Mapping software will need spatial data in the @Polygons slot, but that's nothing we'll manipulate directly). Run `str(usgeo@data)` and that structure should look familiar - much more like a typical R data frame, including columns for STATEFP (state FIPS code), COUNTYFP (county FIPS codes), NAME (in this

case county names, making it easy to match up with county names in election results).

Extracting geodata just for New Hampshire is similar to subsetting any other type of data in R, we just need the state FIPS code for New Hampshire, which turns out to be 33 — or in this case "33," since the codes aren't stored as integers in usgeo.

Here's the command to extract New Hampshire data using FIPS code 33:

```
nhgeo <- usgeo[usgeo@data$STATEFP=="33",]
```

If you want to do a quick check to see if nhgeo looks correct, run the quick thematic map function again `qtm(nhgeo)` and you should see something like this:

Still somewhat boring, but it looks like the Granite State with county-sized divisions, so it appears we've got the correct file subset.

Step 4: Merge spatial and results data

Like any database join or merge, this has two requirements: 1) a column shared by each data set, and 2) records stored exactly the same way in both data sets. (Having a county listed as "Hillsborough" in one file and FIPS code "011" in another wouldn't give R any idea how to match them up without some sort of translation table.)

Trust me: You will save yourself a lot of time if you run a few R commands to see whether the nhgeo@data$NAME vector of county names is the same as the nhdata$County vector of county names.

Do they have the same structure?

```
str(nhgeo@data$NAME)
```

Factor w/ 1921 levels "Abbeville","Acadia",..: 1470 684 416 1653 138 282 1131 1657 334 791

```
str(nhdata$County)
```

chr [1:11] "Belknap" "Carroll" "Cheshire" "Coos" "Grafton"

Whoops, problem number one: The geospatial file lists counties as R factors, while they're plain character text in the data. Change the factors to character strings with:

```
nhgeo@data$NAME <- as.character(nhgeo@data$NAME)
```

Next, it is helpful to sort both data sets by county name and then compare.

```
nhgeo <- nhgeo[order(nhgeo@data$NAME),]

nhdata <- nhdata[order(nhdata$County),]
```

Are the two county columns identical now? They should be; let's check:

```
identical(nhgeo@data$NAME,nhdata$County )

[1] TRUE
```

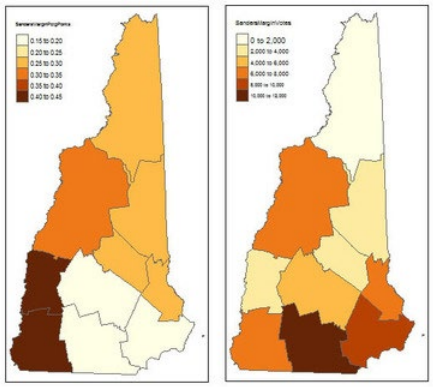
Now we can join the two files. The sp package's merge function is pretty common for this type of task, but I like tmap's append_data() because of its intuitive syntax and allowing names of the two join columns to be different.

```
nhmap <- append_data(nhgeo, nhdata, key.shp =
"NAME", key.data="County")
```

You can see the new data structure with:

```
str(nhmap@data)
```

Step 5: Create a static map



The hard part is done: Finding data, getting it into the right format and merging it with geospatial data. Now, creating a simple static map of Sanders' margins by county in number of votes is as easy as:

```
qtm(nhmap,
"SandersMarginVotes")
```

and mapping margins by percentage:

```
qtm(nhmap, "SandersMarginPctgPoints")
```

We can see that there's some difference between which areas gave Sanders the highest percent win versus which ones were most valuable for largest number-of-votes advantage.

For more control over the map's colors, borders and such, use the tm_shape() function, which uses a ggplot2-like syntax to set fill, border and other attributes:

```
tm_shape(nhmap) +
tm_fill("SandersMarginVotes", title="Sanders Margin, Total
Votes", palette = "PRGn") +
tm_borders(alpha=.5) +
tm_text("County", size=0.8)
```

The first line above sets the geodata file to be mapped, while tm_fill() sets the data column to use for mapping color values. The PRGn" palette argument is a ColorBrewer palette of purples and greens — if you're not familiar with ColorBrewer, you can see the various palettes available at colorbrewer2.org. Don't like the ColorBrewer choices? You can use built-in R palettes or set your own color HEX values manually instead of using a named ColorBrewer option.

There are also a few built-in tmap themes, such as tm_style_classic:

```
tm_shape(nhmap) +
  tm_fill("SandersMarginVotes", title="Sanders Margin,
Total Votes", palette = "PRGn") +
  tm_borders(alpha=.5) +
  tm_text("County", size=0.8) +
tm_style_classic()
```

You can save static maps created by tmap by using the save_ tmap() function:

```
nhstaticmap <- tm_shape(nhmap) +
  tm_fill("SandersMarginVotes", title="Sanders Margin,
Total Votes", palette = "PRGn") +
  tm_borders(alpha=.5) +
tm_text("County", size=0.8)
save_tmap(nhstaticmap, filename="nhdemprimary.jpg")
```

The filename extension can be .jpg, .svg, .pdf, .png and several others; tmap will then produce the appropriate file, defaulting to the size of your current plotting window. There are also arguments for width, height, dpi and more; run `?("save_tmap")` for more info.

If you'd like to learn more about available tmap options, package creator Martijn Tennekes posted a PDF presentation on creating maps with tmap as well as tmap in a nutshell.

Step 6: Create palette and pop-ups for interactive map

The next map we'll create will let users click to see underlying data as well as switch between maps, thanks to RStudio's Leaflet package that gives an R front-end to the open-source JavaScript Leaflet mapping library.

For a Leaflet map, there are two extra things we'll want to create in addition to the data we already have: A color palette and pop-up window contents.

For palette, we specify the data range we're mapping and what kind of color palette we want — both the particular colors and the *type* of color scale. There are four built-in types:

- colorNumeric is for a continuous range of colors from low to high, so you might go from a very pale blue all the way to a deep dark blue, with many gradations in between.
- colorBin maps a set of numerical data to a set of discreet bins, either defined by exact breaks or specific number of bins — things like "low," "medium" and "high".
- colorQuantile maps numerical data into groups where each group (quantile) has the same number of records — often used for income levels, such as bottom 20%, next-lowest 20% and so on.
- colorFactor is for non-numerical categories where no numerical value makes sense, such as countries in Europe that are part of the Eurozone and those that aren't.

Create a Leaflet palette with this syntax:

```
mypalette <- colorFunction(palette = "colors I
want", domain = mydataframe$dataColumnToMap)
```

where colorFunction is one of the four scale types above, such as colorNumeric() or colorFactor and "colors I want" is a vector of colors.

Just to change things up a bit, I'll map where *Hillary Clinton* was strongest, the inverse of the Sanders maps. To map Clinton's vote percentage, we could use this palette:

```
clintonPalette <- colorNumeric(palette = "Blues",
domain=nhmap$ClintonPct)
```

where "Blues" is a range of blues from ColorBrewer and domain is the *data range* of the color scale. This can be the same as the data we're actually plotting but doesn't have to be. colorNumeric means we want a continuous range of colors, not specific categories.

We'll also want to add a pop-up window — what good is an interactive map without being able to click or tap and see underlying data?

Aside: For the pop-up window text display, we'll want to turn the decimal numbers for votes such as 0.7865 into percentages like 78.7%. We could do it by writing a short formula, but the scales package has a percent() function to make this easier. Install (if you need to) and load the scales package:

```
install.packages("scales")
```

```
library("scales")
```

Content for a pop-up window is just a simple combination of HTML and R variables, such as:

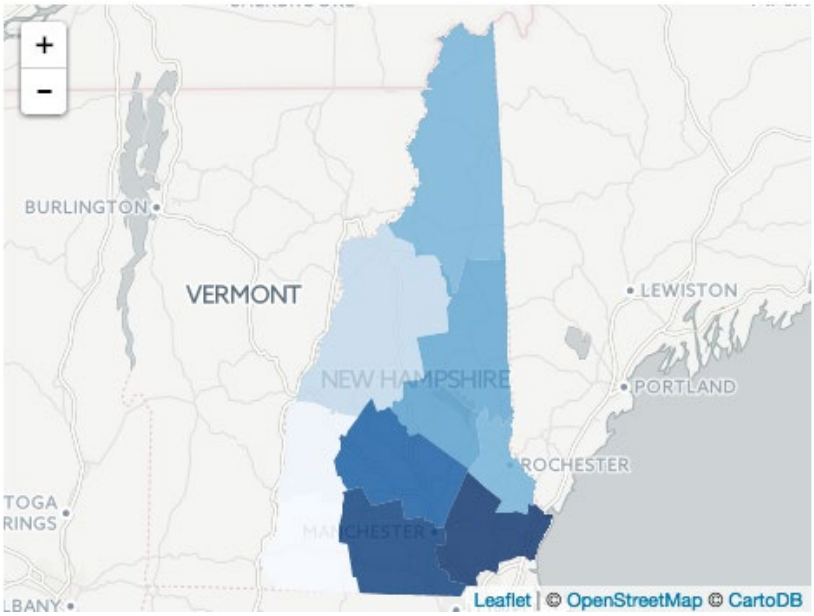
```
nhpopup <- paste0("County: ", nhmap@data$County,
"Sanders ", percent(nhmap@data$SandersPct), " - Clinton ",
percent(nhmap@data$ClintonPct))
```

(If you're not familiar with paste0, it's a concatenate function to join text and text within variables.)

Step 7: Generate an interactive map

Now, the map code:

```
leaflet(nhmap) %>%
  addProviderTiles("CartoDB.Positron") %>%
  addPolygons(stroke=FALSE,
              smoothFactor = 0.2,
              fillOpacity = .8,
              popup=nhpopup,
              color= ~clintonPalette(nhmap@data$ClintonPct)
)
```



■ **Basic interactive map created in R and RStudio's Leaflet package.**

Let's go over the code. `leaflet(nhmap)` creates a leaflet map object and sets nhmap as the data source. `addProviderTiles("CartoDB.Positron" )` sets the background map tiles to CartoDB's attractive Positron design. There's a list of free background tiles and what they look like on GitHub if you'd like to choose something else.

The `addPolygons()` function does the rest — putting the county shapes on the map and coloring them accordingly. `stroke=FALSE` says no border around the counties, `fillOpacity` sets the opacity of the colors, `popup` sets the contents of the popup window and `color` sets the palette — I'm not sure why the tilde is needed before the palette

name, but that's the function format — and what data should be mapped to the color.

The Leaflet package has a number of other features we haven't used yet, including adding legends and the ability to turn layers on and off. Both will be very useful when mapping a race with three or more candidates, such as the current Republican primary.

Step 8: Add palettes for a multi-layer map

Let's look at the GOP results in South Carolina among the top three candidates. I won't go over the data wrangling on this, except to say that I downloaded results from the South Carolina State Election Commission as well as Census Bureau data for education levels by county. If you download the project files, you'll see the initial data as well as the R code I used to add candidate vote percentages and join all that data to the South Carolina shapefile. That creates a geospatial object scmap to map.

There's so much data for a multi-candidate race that it's a little more complicated to choose what to color beyond "who won." I decided to go with one map layer to show the winner in each county, one layer each for the top three candidates (Trump, Rubio and Cruz) and a final layer showing percent of adult population with at least a bachelor's degree. (Why education? Some news reports out of South Carolina said that seemed to correlate with levels of Trump's support; mapping that will help show such a trend.)

In making my color palettes, I decided to use *the same numerical scale for all three candidates*. If I scaled color intensity for each candidate's minimum and maximum, a candidate with 10% to 18% would have a map with the same color intensities as one who had 45% to 52%, which gives a wrong impression of the losing candidate's strength. So, first I calculated the minimum and maximum for the combined Trump/Rubio/Cruz county results:

```
minpct <- min(c(scmap$`Donald J TrumpPct`, scmap$`Marco
RubioPct` , scmap$`Ted CruzPct`))
maxpct <- max(c(scmap$`Donald J TrumpPct`, scmap$`Marco
RubioPct` , scmap$`Ted CruzPct`))
```

Now I can create a palette for each candidate using *different colors* but *the same intensity range*.

```
trumpPalette <- colorNumeric(palette = "Purples",
domain=c(minpct, maxpct))
rubioPalette <- colorNumeric(palette = "Reds", domain =
c(minpct, maxpct))
cruzPalette <- colorNumeric(palette = "Oranges", domain =
c(minpct, maxpct))
```

I'll also add palettes for the winner and education layers:

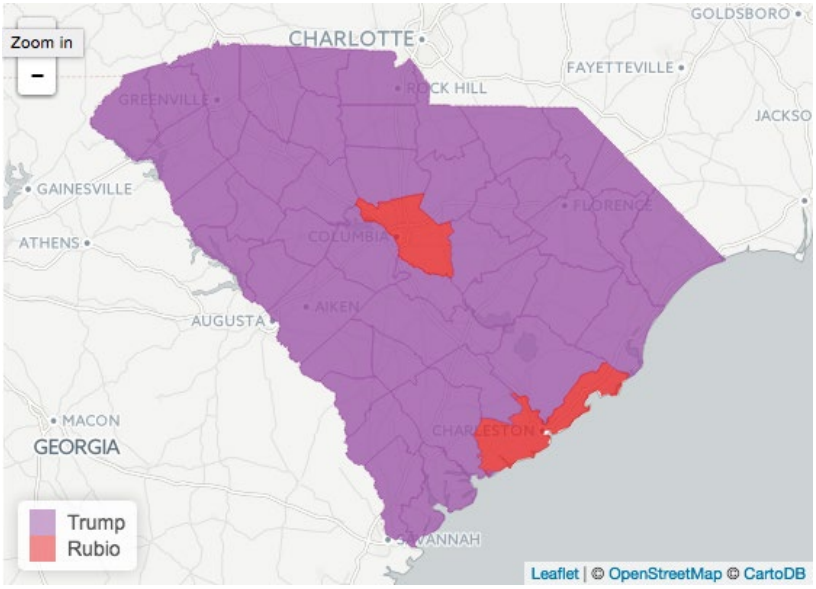
```
winnerPalette <- colorFactor(palette=c("#984ea3",
"#e41a1c"), domain = scmap$winner)
edPalette <- colorNumeric(palette = "Blues", domain=scmap@
data$PctCollegeDegree)
```

Finally, I'll create a basic pop-up showing the county name, who won, the percentage for each candidate and percent of population with a college degree:

```
scpopup <- paste0("County: ", scmap@data$County,
"Winner: ", scmap@data$winner,
"Trump: ", percent(scmap@data$`Donald J TrumpPct`),
"Rubio: ", percent(scmap@data$`Marco RubioPct`),
"Cruz: ", percent(scmap@data$`Ted CruzPct`),
"Pct w college ed: ", scmap@data$PctCollegeDegree, "% vs
state-wide avg of 25%")
```

This shows a basic map of winners by county:

```
leaflet(scmap) %>%
  addProviderTiles("CartoDB.Positron") %>%
  addPolygons(stroke=TRUE,
              weight=1,
              smoothFactor = 0.2,
              fillOpacity = .75,
              popup=scpopup,
              color= ~winnerPalette(scmap@data$winner),
              group="Winners"
 ) %>%
    addLegend(position="bottomleft", colors=c("#984ea3",
"#e41a1c"), labels=c("Trump", "Rubio"))
```



■ **Another basic interactive map, this one with data on more than two candidates.**

Step 9: Add map layers and controls

A multi-layer map with layer controls starts off the same as our previous map, with one addition: A group name. In this case, each layer will be its own group, but it's also possible to turn multiple layers on and off together.

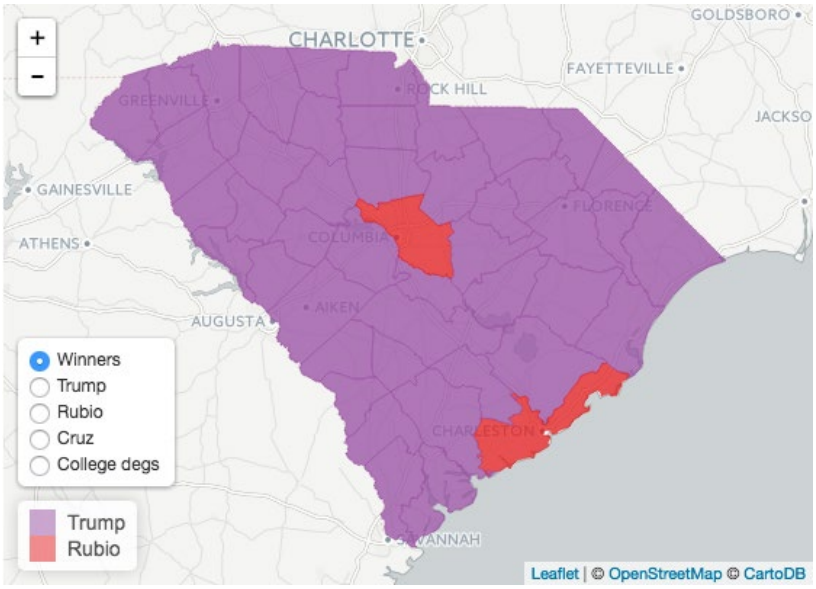
The next step is to add additional polygon layers for each candidate and a final layer for college education, along with a layer control to wrap up the code. This time, we'll store the map in a variable and then display it:

```
scGOPmap <- leaflet(scmap) %>%
  addProviderTiles("CartoDB.Positron") %>%
  addPolygons(stroke=TRUE,
              weight=1,
              smoothFactor = 0.2,
              fillOpacity = .75,
              popup=scpopup,
              color= ~winnerPalette(scmap@data$winner),
              group="Winners"
  ) %>%
    addLegend(position="bottomleft", colors=c("#984ea3",
"#e41a1c"), labels=c("Trump", "Rubio"))  %>%
```

```
  addPolygons(stroke=TRUE,
     weight=1,
     smoothFactor = 0.2,
     fillOpacity = .75,
     popup=scpopup,
     color= ~trumpPalette(scmap@data$`Donald J TrumpPct`),
     group="Trump"
    ) %>%

  addPolygons(stroke=TRUE,
              weight=1,
              smoothFactor = 0.2,
              fillOpacity = .75,
              popup=scpopup,
              color= ~rubioPalette(scmap@data$`Marco
RubioPct`),
              group="Rubio"
  ) %>%

  addPolygons(stroke=TRUE,
              weight=1,
              smoothFactor = 0.2,
              fillOpacity = .75,
              popup=scpopup,
              color= ~cruzPalette(scmap@data$`Ted
CruzPct`),
              group="Cruz"
  ) %>%
  addPolygons(stroke=TRUE,
              weight=1,
              smoothFactor = 0.2,
              fillOpacity = .75,
              popup=scpopup,
              color= ~edPalette(scmap@
data$PctCollegeDegree),
              group="College degs"
  ) %>%

  addLayersControl(
      baseGroups=c("Winners", "Trump", "Rubio", "Cruz",
"College degs"),
      position = "bottomleft",
      options = layersControlOptions(collapsed = FALSE)
      )
```

And now display the map with:

```
scGOPmap
```



■ **Interactive map with multiple layers. Click on the radio buttons at the bottom left to change which layer displays.**

`addLayersControl` can have two types of groups: baseGroups, like used above, which allow only one layer to be viewed at a time; and overlayGroups, where multiple layers can be viewed at once and each turned off individually.

Step 10: Save your interactive map

If you're familiar with RMarkdown or Shiny, a Leaflet map can be embedded in an RMarkdown document or Shiny Web application. If you'd like to use this map as an HTML page on a website or elsewhere, save a Leaflet map with the htmlwidget package's saveWidget() function:

```
# install.packages("htmlwidgets")
library("htmlwidgets")
saveWidget(widget=scGOPmap, file="scGOPprimary.html")
```

You can also save the map with external resources such as jQuery and the Leaflet JavaScript code in a separate directory by using the `selfcontained=FALSE` argument and choosing the subdirectory for the dependency files:

```
# install.packages("htmlwidgets")
save(widget=scGOPmap2, file="scGOPprimary_withdependencies.
html", selfcontained=FALSE, libdir = "js")
```

This should get you started on creating your own choropleth maps with R.

Create a Leaflet map with markers

If you're not familiar with Leaflet, it's a JavaScript mapping package. To install it, you need to use the devtools package and get it from GitHub (if you don't already have devtools installed on your system, download and install it with install.packages("devtools").

```
devtools::install_github("rstudio/leaflet")
```

Load the library:

```
library("leaflet")
```

Step 1: Create a basic map object and add tiles

```
mymap <- leaflet()
mymap <- addTiles(mymap)
```

View the empty map by typing the object name:

```
mymap
```

Step 2: Set where you want the map to be centered and its zoom level:

```
mymap <- setView(mymap, -84.3847, 33.7613, zoom = 17)
mymap
```

Add a pop-up:

```
addPopups(-84.3847, 33.7616, 'Data journalists at work,
<b>NICAR 2015</b>')
```

A reminder that the chaining function in R - %>% - takes the results of one function and sends it to the next one, so you don't have to keep repeating the variable name you're storing things, similar to the one-character Unix pipe command. We could compact the code above to:

```
mymap <- leaflet() %>%
  addTiles() %>%
  setView(-84.3847, 33.7613, zoom = 17) %>%
  addPopups(-84.3847, 33.7616, "Data journalists at work,
<b>NICAR 2015</b>")
View
```

View the finished product:

```
mymap
```

Or if you didn't want to store the results in a variable for now but just work interactively:

```
leaflet() %>%
  addTiles() %>%
  setView(-84.3847, 33.7613, zoom = 16) %>%
  addPopups(-84.3847, 33.7616, 'Data journalists at work,
<b>NICAR 2015</b>')
```

Now let's do something a little more interesting — map nearby Starbucks locations. Load the starbucks.csv data set; see data source at: https://opendata.socrata.com/Business/All-Starbucks-Locations-in-the-US-Map/ddym-zvjk

Data files for these exercises are available on my NICAR15data repository on GitHub. You can also download the Starbucks data file directly from Socrata's OpenData site in R with the code:

```
download.file("https://opendata.socrata.com/api/views/ddym-
zvjk/rows.csv?accessType=DOWNLOAD", destfile="starbucks.
csv", method="curl")
```

Here's code to read in the data and make the map:

```
starbucks <- read.csv("starbucks.csv",
stringsAsFactors = FALSE)
str(starbucks)
atlanta <- subset(starbucks, City == "Atlanta" & State ==
"GA")
leaflet() %>% addTiles() %>% setView(-84.3847, 33.7613,
zoom = 16) %>%
  addMarkers(data = atlanta, lat = ~ Latitude, lng = ~
Longitude,popup = atlanta$Name) %>%
  addPopups(-84.3847, 33.7616, "Data journalists at work,
<b>NICAR 2015</b>")
```

A script created by a TCU prof lets you create choropleth maps of World Bank data with a single line of code! More info here: http://rpubs.com/walkerke/wdi_leaflet

More info on the Leaflet project page: http://rstudio.github.io/leaflet/

A little more fun with Starbucks data: How many people are there per Starbucks in each state? Let's load in a file of state populations:

```
statepops <- read.csv("acs2013_1yr_statepop.csv",
stringsAsFactors = FALSE)
# A little glimpse at the dplyr library; lots more on that
soon
library(dplyr)
```

There's a very easy way to count Starbucks by state with dplyr's count function format: count(mydataframe, mycolumnname)

```
starbucks_by_state <- count(starbucks, State)
```

We'll need to add state population here. You can do that with base R's merge or dplyr's left_join. left_join is faster but I find merge more intuitive:

```
starbucks_by_state <- merge(starbucks_by_state, statepops,
all.x = TRUE, by.x="State", by.y="State") # No need to do
by.x and by.y if columns have the same name

# better names

names(starbucks_by_state) <- c("State", "NumberStarbucks",
"StatePopulation")
```

Add new column to starbucks_by_state with dplyr mutate function, which just means alter the data frame by adding one or more columns. Then we'll store in a new dataframe, starbucks_data, so as not to mess with the original.

```
starbucks_data <- starbucks_by_state %>%

  mutate(
    PeoplePerStarbucks = round(StatePopulation /
NumberStarbucks)
    ) %>%
  select(State, NumberStarbucks, PeoplePerStarbucks) %>%
  arrange(desc(PeoplePerStarbucks))
```

Again the %>% character, so we don't have to keep writing things like:

```
starbucks_data <- mutate(starbucks_by_state,
PeoplePerStarbucks = round(StatePopulation /
NumberStarbucks))
starbucks_data <- select(starbucks_data, State,
NumberStarbucks, PeoplePerStarbucks)
starbucks_data <- arrange(starbucks_data,
desc(PeoplePerStarbucks))
```

More mapping resources

A script created by a TCU prof lets you create choropleth maps of World Bank data with Leaflet a single line of code! More info here: http://rpubs.com/walkerke/wdi_leaflet

You can do considerably more sophisticated GIS work with Leaflet and R.

Draw circles with a 2km radius around each marker, for example. Tutorial by TCU assistant prof Kyle Walker http://rpubs.com/walkerke/rstudio_gis

More info about Leaflet on the Leaflet project page http://rstudio.github.io/leaflet/

Extract custom data from the Google Analytics API

Google Analytics provides a robust API that enables you to tap into your data programmatically, meaning you can conveniently pull and package data in ways that might not be as easy to do on the Web. Google has tutorials that cover how to use this feature with Java, Python, PHP and JavaScript, but I prefer to tap into Google Analytics with R, a language that's specifically designed for data visualization and graphical analysis.

There are several R packages available that have functions specifically designed for Google Analytics, including ganalytics, RGoogleAnalytics and rga ("R Google Analytics"). I'll be using rga for this tutorial, but any of them would work.

Step 1: Install packages

Like ganalytics, rga resides on GitHub. To easily install any of the Google Analytics packages from GitHub, first install and load the R package devtools by typing the following commands into the R console window:

```
install.packages("devtools")

library(devtools)
```

Then install and load rga from package author Bror Skardhamar's account:

```
install_github("skardhamar/rga")

library(rga)
```

(You only have to run the first three commands once per machine, but you need to load `library(rga)` each time you open R.)

Step 2: Allow rga to access your Google Analytics account

On a Mac, authentication is as easy: Create an instance of the Google Analytics API authentication object by typing the following in your R console window:

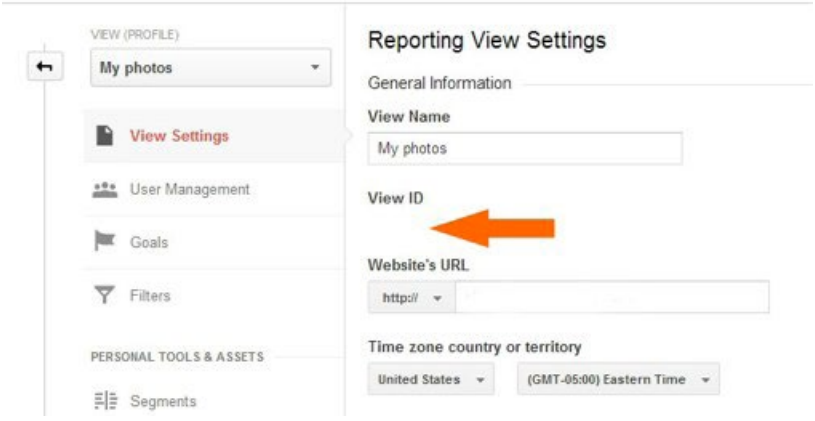
```
rga.open(instance="ga")
```

That will open a browser window that asks you to give rga permission to access your Google data. When you accept, you'll be given a code to cut and paste back into your R console window where it says, "Please enter code here."

In Windows, I find that adding a line of code before opening an rga instance helps with any authentication errors:

```
options(RCurlOptions = list(cainfo = system.
file("CurlSSL", "cacert.pem", package = "RCurl")))

rga.open(instance="ga")
```

Next, you need to find the profile ID for your Google account, which is *not* found in the tracking code that you add to a website to allow Google Analytics to monitor your site. Instead, on your Google Analytics Admin page, go to View Settings and you'll see the ID under "View ID."



- You'll find your profile ID for your Google account by going to View Settings on your Google Analytics Admin page.

Or, run the command:

```
ga$getProfiles()
```

in your R terminal window to get a list of all available profiles in your account; the profile ID will be listed in the first column.

Whichever way you find it, save that value in a variable so you don't have to keep typing it. You can use a command like:

```
id <- "1234567"
```

(Replace the number with your actual ID, and make sure to put it between quote marks.) This stores your profile ID as the variable "id."

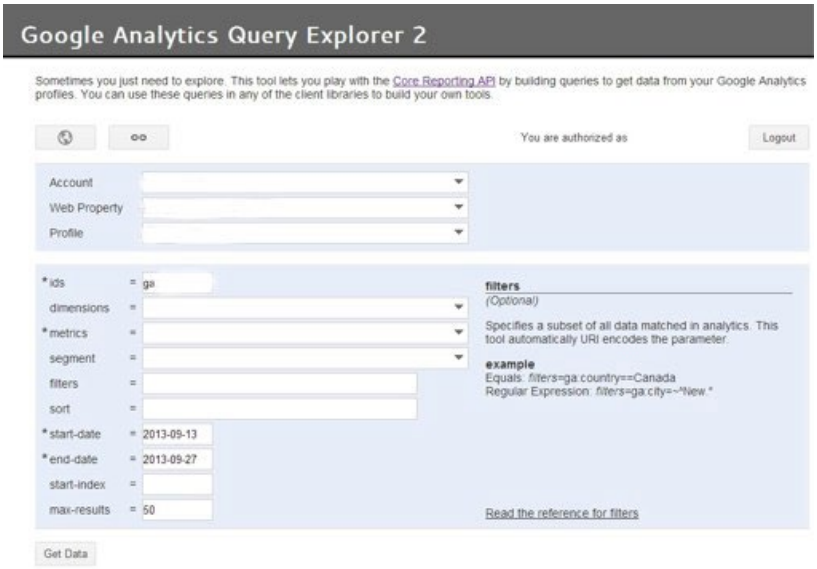
Step 3: Extract data

Now we're ready to start pulling some data using the ga instance we just created. The getData method will actually extract data from your Google Analytics account that you can then store in another new R variable. If you want to see all available methods for your ga object, run:

```
ga$getRefClass()
```

You can query the Google API for metrics and dimensions. Metrics are things like page views, visits and organic searches; dimensions include information like traffic sources and visitor type. (See Google's Dimensions Metrics Reference for full details.)

In addition, you can focus your query by criteria like visits from search, visits with conversions (assuming you've set that up in Google Analytics beforehand) and even visits just from tablets, by including *segments* in a query. Finally, you can also create your own filters to narrow your results.



■ **Google's Query Explorer helps you figure out what data is available and how to structure a query.**

Google has created a Query Explorer for the Google Analytics API. It's a great resource to help you figure out what data is available and how to structure a query. If you're new to the Google Analytics API, play around with Query Explorer for a bit to see what data you can extract and the variables you need to pull the data you want. Further information on the terms to use for various queries is available in the API documentation.

Once you decide on what you'd like to include in your query, here's the syntax for using R to get the data:

```
myresults <- ga$getData(id, start.date="", end.
date="",

metrics = "",

dimensions = "",

sort = "",

filters = "",

segment = "",

start = 1,

max = 1000)
```

You fill in information for your specific query between the various quotation marks, of course. Note that dates are in the format yyyy-mm-dd, such as "2013-10-30."

Here's a specific example: Say I want to see the top ten referrers for visits to my site in September. My start date is September 1 and my end date is September 30. My metric is visits — called "ga:visits" by the API — and my dimension is their sources — called "ga:source."

I'll further refine the query to get just my top 10 referrers:

```
myresults <- ga$getData(id, start.date="2015-09-01",
end.date="2015-09-30",

metrics = "ga:visits",

dimensions = "ga:source",

sort = "-ga:visits",

start = 1, max = 10)
```

Here's a breakdown of that query:

- `ga$getData` is using the getData method of my ga Google Analytics API-accessing object.
- The first argument, `id`, is the profile number for my account, which I already stored in a variable called id.
- Next are the start and end dates for my query, followed by the metric I want (`"ga:visits"`).
- Since I want to know the visits by source, I specify the dimension as `"ga.source"`.
- I only want the top 10 referrers, so I need to sort the `ga:visits` results in descending order. I do that on the next line by putting a minus sign in front of `ga:visits` when setting the sort criteria.
- Finally I specifically ask to start at the first result with a maximum of 10 to return at most 10 listings.

The results are stored in the variable `myresults`. Type

```
myresults
```

at the R prompt in your R terminal window to see what data has been returned.

```
> myresults <- ga$getData(id, start.date="2013-09-01", end.date="2013-09-30",
+                         metrics = "ga:visits",
+                         dimensions = "ga:source",
+                         sort="-ga:visits",
+                         start=1, max=10)
> myresults
                  source visits
1                (direct)     24
2                  google     14
3             smugmug.com      4
4                   yahoo      2
5 us-mg204.mail.yahoo.com      1
6      webmaila.juno.com       1
```

■ **The results from a query searching for a site's top 10 referrers.**

If I wanted to see the *overall* number of visits without breaking it down by source, I wouldn't include the dimensions, sort, start or max in the query. Instead, I'd just use a simple:

```
myresults <- ga$getData(id, start.date="2015-09-01",
end.date="2015-09-30", metrics = "ga:visits")
```

Note that if you are trying to copy and paste, in R you can't break a variable or other name at the . onto separate lines, or have a date break in mid-date at a hyphen.

That returns a listing of number of visits per day. I can have it return results by different time periods by adding the time dimension of my choice — for example by week:

```
myresultsPVsByWeek <- ga$getData(id, start.
date="2013-09-01", end.date="2013-09-30",

metrics = "ga:visits",

dimensions = "ga:week")
```

Or, I can get page views for the entire year by month:

```
myresultsPVsByMonth <- ga$getData(id, start.
date="2013-01-01", end.date="2013-12-31",

metrics = "ga:pageviews",

dimensions = "ga:month")
```

You can seek more than one metric at a time:

```
myresultsPVsVisits <- ga$getData(id, start.
date="2013-01-01", end.date="2013-12-31",

metrics = "ga:visits, ga:pageviews",

dimensions = "ga:month")
```

(For those who know R and are used to combining items using R's concatenate `c()` function, you *don't* use that when combining items within a `ga$getData` query.)

Want to just see visits that came from, say, Google News each month this year? Add a filter, such as:

```
myresultsGNvisits <- ga$getData(id, start.date =
"2013-01-01", end.date = "2013-12-31",

metrics = "ga:visits",

filters = "ga:source=~news.google.com",

dimensions = "ga:month")
```

I used =~ rather than == because the latter would set the filter to only those referrals that *exactly equal* news.google.com. By using the =~ operator instead, it uses more powerful regular expression searching, which in this case would match anything containing news.google.com. (Regular expressions allow much more robust pattern searching.)

As before, for each of these queries, type:

```
myresults
```

(or the appropriate results variable) at the prompt in your R window to see what's returned.

```
> myresults <- ga$getData(id, start.date="2013-01-01", end.date="2013-12-31",
+                         metrics = "ga:visits, ga:pageviews",
+                         dimensions = "ga:month")
> myresults
   month visits pageviews
1     01     86      1023
2     02    113      1330
3     03    102       865
4     04    287      2306
5     05     79      1022
6     06    159      1478
7     07    148      1146
8     08     36       400
9     09     46       767
10    10      0         0
11    11      0         0
12    12      0         0
>
```

■ **The query has been refined to show the visits that came from Google News each month for a year.**

Step 4: Manipulate your data

Now that you've got your data, what can you do with it?

If you're not an R enthusiast, the easiest thing is to save the results to a CSV file. R's `write.csv()` function first lists what you want to save and then the file name. To save the myresults variable to a file called data.csv, type:

```
write.csv(myresults, file="data.csv", row.
names=FALSE)
```

The optional `row.names=FALSE` eliminates an extra column with the row numbers, just to keep the file uncluttered. The resulting file looks something like this (but hopefully with many more visits):

"month","visits"

"01",625

"02",790

"03",395

"04",219

"05",927

"06",151

"07",231

"08",244

"09",231

You can then use that data in the spreadsheet or graphing program of your choice.

You can also analyze your data right within R, of course, without exporting to a spreadsheet. Let me first pull some real data — visits and page views — from a personal site I set up years ago that I no longer tend to but that still gets occasional visitors:

```
mydata <- ga$getData(id, start.date="2013-01-01",
end.date="2013-12-31",

metrics = "ga:visits, ga:pageviews",

dimensions = "ga:month")
```

```
> mydata <- ga$getData(id, start.date="2013-01-01", end.date="2013-12-31",
+                      metrics = "ga:visits, ga:pageviews",
+                      dimensions = "ga:month")
> mydata
   month visits pageviews
1     01     70       123
2     02     40        44
3     03     54        62
4     04     54        69
5     05    126       147
6     06     46        60
7     07     39        54
8     08     43        55
9     09     54        59
10    10      0         0
11    11      0         0
12    12      0         0
>
```

■ **Data on monthly visits to a site.**

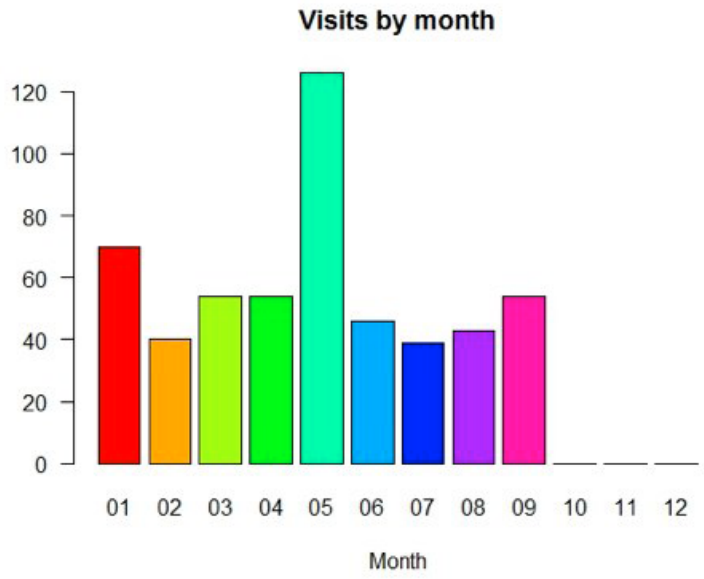
You can use R's `str()` function to find out how the `mydata` object is structured.

```
> str(mydata)
'data.frame': 12 obs. of  3 variables:
 $ month    : chr  "01" "02" "03" "04" ...
 $ visits   : num  70 40 54 54 126 46 39 43 54 0 ...
 $ pageviews: num  123 44 62 69 147 60 54 55 59 0 ...
```

■ **This shows how the mydata object is structured.**

Like the other results above, it's an R data frame with character strings as the month number and numbers for the data. That makes it easy to run simple analyses and generate basic graphs within R, such as:

```
barplot(mydata$visits, main="Visits by month",
xlab="Month", names.arg=mydata$month, las=1,
col=rainbow(9))
```



■ **You can generate basic graphs within R, such as this one, which shows the number of visits to a site for each month.**

The `R barplot()` command above uses the number of visits for the graph's y axis values (you can refer to a specific column in a data frame with the syntax `dataframename$columnname)` and `names.arg` as names on the x axis. The command `main` specifies the graph title, `xlab` is the x-axis label and `col=rainbow(9)` tells R to choose nine colors from its rainbow palette to color the bars. The nonintuitive command `las=1` tells R to set both the x- and y-axis labels horizontally (0 makes them parallel to the axis, 2 perpendicular to the axis, and 3 vertical).

More R Resources

For more resources to help improve your R skills, see Computerworld's 60+ R resources.