LEARN TO USE R

Your hands-on guide

2 Introduction

6 Getting your data into R

10 Easy ways to do basic data analysis

17 Painless data visualization

26 Syntax quirks you'll want to know

33 Useful resources

by Sharon Machlis

edited by Johanna Ambrosio

COMPUTERWORLD

Introduction

R is hot. Whether measured by more than 6,100 add-on packages, the 41,000+ members of LinkedIn's R group or the 170+ R Meetup groups currently in existence, there can be little doubt that interest in the R statistics language, especially for data analysis, is soaring.

Why R? It's free, open source, powerful and highly extensible. "You have a lot of prepackaged stuff that's already available, so you're standing on the shoulders of giants," Google's chief economist told *The New York Times* back in 2009.

Because it's a programmable environment that uses command-line scripting, you can store a series of complex data-analysis steps in R. That lets you re-use your analysis work on similar data more easily than if you were using a point-and-click interface, notes Hadley Wickham, author of several popular R packages and chief scientist with RStudio.

That also makes it easier for others to validate research results and check your work for errors — an issue that cropped up in the news recently after an Excel coding error was among several flaws found in an influential economics analysis report known as Reinhart/Rogoff.

The error itself wasn't a surprise, blogs Christopher Gandrud, who earned a doctorate in quantitative research methodology from the London School of Economics. "Despite our best efforts we always will" make errors, he notes. "The problem is that we often use tools and practices that make it difficult to find and correct our mistakes."

Sure, you can easily examine complex formulas on a spreadsheet. But it's not nearly as easy to run multiple data sets through spreadsheet formulas to check results as it is to put several data sets through a script, he explains.

Indeed, the mantra of "Make sure your work is reproducible!" is a common theme among R enthusiasts.

Who uses R?

Relatively high-profile users of R include:

Facebook: Used by some within the company for tasks such as analyzing user behavior.

Google: There are more than 500 R users at Google, according to David Smith at Revolution Analytics, doing tasks such as making online advertising more effective.

National Weather Service: Flood forecasts.

Orbitz: Statistical analysis to suggest best hotels to promote to its users.

Trulia: Statistical modeling.

Source: Revolution Analytics

Why *not* R? Well, R can appear daunting at first. That's often because R syntax is different from that of many other languages, not necessarily because it's any more difficult than others.

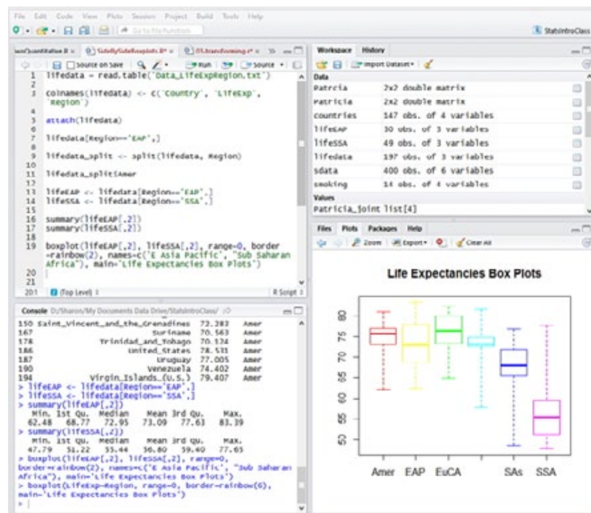
"I have written software professionally in perhaps a dozen programming languages, and the hardest language for me to learn has been R," writes consultant John D. Cook in a Web post about R programming for those coming from other languages. "The language is actually fairly simple, but it is unconventional."

And so, this guide. Our aim here isn't R mastery, but giving you a path to start using R for basic data work: Extracting key statistics out of a data set, exploring a data set with basic graphics and reshaping data to make it easier to analyze.

Your first step

To begin using R, head to r-project.org to download and install R for your desktop or laptop. It runs on Windows, OS X and "a wide variety of Unix platforms," but not yet on Android or iOS.

Installing R is actually all you need to get started. However, I'd suggest also installing the free R integrated development environment (IDE) RStudio. It's got useful features you'd expect from a coding platform, such as syntax highlighting and tab for suggested code auto-completion. I also like its four-pane workspace, which better manages multiple R windows for typing commands, storing scripts, viewing command histories, viewing visualizations and more.



■ **Although you don't need the free RStudio IDE to get started, it makes working with R much easier.**

The top left window is where you'll probably do most of your work. That's the R code editor allowing you to create a file with multiple lines of R code — or open an existing file — and then run the entire file or portions of it.

Bottom left is the interactive console where you can type in R statements one line at a time. Any lines of code that are run from the editor window also appear in the console.

The top right window shows your workspace, which includes a list of objects currently in memory. There's also a history tab with a list of your prior commands; what's handy there is that you can select one, some or all of those lines of code and one-click to send them either to the console or to whatever file is active in your code editor.

The window at bottom right shows a plot if you've created a data visualization with your R code. There's a history of previous plots and an option to export a plot to an image file or PDF. This window also shows external packages (R extensions) that are available on your system, files in your working directory and help files when called from the console.

Learning the shortcuts

Wickham, the RStudio chief scientist, says these are the three most important keyboard shortcuts in RStudio:

- **Tab** is a generic auto-complete function. If you start typing in the console or editor and hit the tab key, RStudio will suggest functions or file names; simply select the one you want and hit either tab or enter to accept it.

- **Control + the up arrow** (command + up arrow on a Mac) is a similar auto-complete tool. Start typing and hit that key combination, and it shows you a list of every command *you've* typed starting

with those keys. Select the one you want and hit return. This works only in the interactive console, not in the code editor window.

- **Control + enter** (command + enter on a Mac) takes the current line of code in the editor, sends it to the console and executes it. If you select multiple lines of code in the editor and then hit ctrl/cmd + enter, all of them will run.

For more about RStudio features, including a full list of keyboard shortcuts, head to the online documentation.

Setting your working directory

Change your working directory with the setwd() function, such as:

```
setwd("~/mydirectory")
```

Note that the slashes always have to be *forward* slashes, even if you're on a Windows system. For Windows, the command might look something like:

```
setwd("C:/Sharon/Documents/
RProjects")
```

If you are using RStudio, you can also use the menu to change your working directory under Session > Set Working Directory.

Installing and using packages

Chances are if you're going to be doing, well, pretty much *anything* in R, you're going to want to take advantage of some of the thousands of add-on packages available for R at CRAN, the Comprehensive R Archive Network. The command for installing a package is:

```
install.packages("thepackagename")
```

If you don't want to type the command, in RStudio there's a Packages tab in the lower right window; click that and you'll see a button to "Install Packages." (There's also a menu command; the location varies depending on your operating system.)

To see which packages are already installed on your system, type:

```
installed.packages()
```

Or, in RStudio, go to the Packages tab in the lower right window.

To use a package in your work once it's installed, load it with:

```
library("thepackagename")
```

If you'd like to make sure your packages stay up to date, you can run:

```
update.packages()
```

and get the latest versions for all your installed packages.

If you no longer need or want a package on your system, use the function:

```
remove.packages("thepackagename")
```

Help!

If you want to find out more about a function, you can type a question mark followed by the function name — one of the rare times parentheses are not required in R, like so:

```
?functionName
```

This is a shortcut to the help function, which *does* use parentheses:

```
help(functionName)
```

Although I'm not sure why you'd want to use this as opposed to the shorter ?functionName command.

If you already know what a function does and just want to see formats for using it properly, you can type:

```
example(functionName)
```

and you’ll get a list with examples of the function being used, if there’s one available. The arguments (args) function:

```
args(functionName)
```

just displays a list of a function’s arguments.

If you want to search through R’s help documentation for a specific term, you can use:

```
help.search(“your search term”)
```

That also has a shortcut:

```
??(“my search term”)
```

No parentheses are needed if the search term is a single word without spaces.

Get your data into R

Once you've installed and configured R to your liking, it's time to start using it to work with data. Yes, you can type your data directly into R's interactive console. But for any kind of serious work, you're a lot more likely to already have data in a file somewhere, either locally or on the Web. Here are several ways to get data into R for further work.

Sample data

If you just want to play with some test data to see how they load and what basic functions you can run, the default installation of R comes with several data sets. Type:

```
data()
```

into the R console and you'll get a listing of pre-loaded data sets. Not all of them are useful (body temperature series of two beavers?), but these do give you a chance to try analysis and plotting commands. And some online tutorials use these sample sets.

One of the less esoteric data sets is mtcars, data about various automobile models that come from *Motor Trends*. (I'm not sure from what year the data are from, but given that there are entries for the Valiant and Duster 360, I'm guessing they're not very recent; still, it's a bit more compelling than whether beavers have fevers.)

You'll get a printout of the entire data set if you type the name of the data set into the console, like so:

```
mtcars
```

There are better ways of examining a data set, which I'll get into later in this series. Also, R does have a print() function for printing with more options, but R beginners rarely seem to use it.

Existing local data

R has a function dedicated to reading comma-separated files. To import a local CSV file named filename.txt and store the data into one R variable named mydata, the syntax would be:

```
mydata <- read.csv("filename.txt")
```

(Aside: What's that <- where you expect to see an equals sign? It's the R assignment operator. I said R syntax was a bit quirky. More on this in the section on R syntax quirks.)

And if you're wondering what kind of object is created with this command, mydata is an extremely handy data type called a data frame — basically a table of data. A data frame is organized with rows and columns, similar to a spreadsheet or database table.

The read.csv function assumes that your file has a header row, so row 1 is the name of each column. If that's *not* the case, you can add header=FALSE to the command:

```
mydata <- read.csv("filename.txt",
header=FALSE)
```

In this case, R will read the first line as data, not column headers (and assigns default column header names you can change later).

If your data use another character to separate the fields, not a comma, R also has the more general read.table function. So if your separator is a tab, for instance, this would work:

```
mydata <- read.table("filename.txt",
sep="\t", header=TRUE)
```

The command above also indicates there's a header row in the file with header=TRUE.

If, say, your separator is a character such as | you would change the separator part of the command to sep="|"

Categories or values? Because of R's roots as a statistical tool, when you import non-numerical data, R may assume that character strings are statistical factors — things like "poor," "average" and "good" — or "success" and "failure."

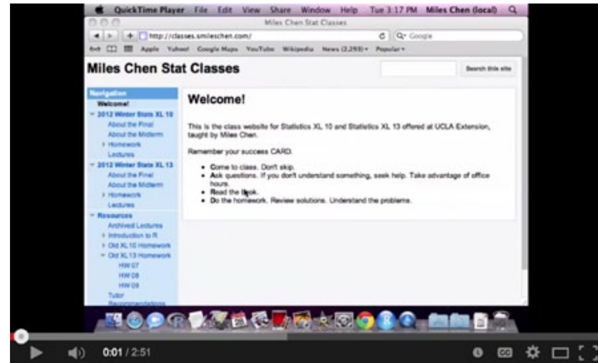
But your text columns may not be categories that you want to group and measure, just names of companies or employees. If you don't want your text data to be read in as factors, add stringsAsFactor=FALSE to read.table, like this:

```
mydata <- read.table("filename.
txt", sep="\t", header=TRUE,
stringsAsFactor=FALSE)
```

If you'd prefer, R allows you to use a series of menu clicks to load data instead of 'reading' data from the command line as just described. To do this, go to the Workspace tab of RStudio's upper-right window, find the menu option to "Import Dataset," then choose a local text file or URL.

As data are imported via menu clicks, the R command that RStudio generated from your menu clicks will appear in your console. You may want to save that data-reading command into a script file if you're using this for significant analysis work, so that others — or you — can reproduce that work.

This 3-minute YouTube video, recorded by UCLA statistics grad student Miles Chen, shows an RStudio point-and-click data import.



■ UCLA statistics grad student Miles Chen shows an RStudio point-and-click data import.

Copying data snippets

If you've got just a small section of data already in a table — a spreadsheet, say, or a Web HTML table — you can control-C copy those data to your Windows clipboard and import them into R.

The command below handles clipboard data with a header row that's separated by tabs, and stores the data in a data frame (x):

```
x <- read.table(file = "clipboard",
sep="\t", header=TRUE)
```

You can read more about using the Windows clipboard in R at the R For Dummies website.

On a Mac, the pipe ("pbpaste") function will access data you've copied with command-c, so this will do the equivalent of the previous Windows command:

```
x <- read.table(pipe("pbpaste"),
sep="\t")
```

Other formats

There are R packages that will read files from Excel, SPSS, SAS, Stata and various relational databases. I don't bother with the Excel package; it requires both Java

and Perl, and in general I'd rather export a spreadsheet to CSV in hopes of not running into Microsoft special-character problems. For more info on other formats, see UCLA's How to input data into R which discusses the foreign add-on package for importing several other statistical software file types.

If you'd like to try to connect R with a database, there are several dedicated packages such as RPostgreSQL, RMySQL, RMongo, RSQLite and RODBC.

(You can see the entire list of available R packages at the CRAN website.)

Remote data

read.csv() and read.table() work pretty much the same to access files from the Web as they do for local data.

Do you want Google Spreadsheets data in R? You don't have to download the spreadsheet to your local system as you do with a CSV. Instead, in your Google spreadsheet — properly formatted with just one row for headers and then one row of data per line — select File > Publish to the Web. (This will make the data public, although only to someone who has or stumbles upon the correct URL. Beware of this process, especially with sensitive data.)

Select the sheet with your data and click "Start publishing." You should see a box with the option to get a link to the published data. Change the format type from Web page to CSV and copy the link. Now you can read those data into R with a command such as:

```
mydata <- read.csv("http://bit.ly/10ER84j")
```

The command structure is the same for any file on the Web. For example, Pew Research Center data about mobile shopping are available as a CSV file for download. You can store the data in a variable called pew_data like this:

```
pew_data <- read.csv("http://bit.ly/11I3iuU")
```

It's important to make sure the file you're downloading is in an R-friendly format first: in other words, that it has a maximum of one header row, with each subsequent row having the equivalent of one data record. Even well-formed government data might include lots of blank rows followed by footnotes -- that's not what you want in an R data table if you plan on running statistical analysis functions on the file.

Help with external data

R enthusiasts have created add-on packages to help other users download data into R with a minimum of fuss.

For instance, the financial analysis package Quantmod, developed by quantitative software analyst Jeffrey Ryan, makes it easy to not only pull in and analyze stock prices but graph them as well.

All you need are four short lines of code to install the Quantmod package, load it, retrieve a company's stock prices and then chart them using the barChart function. Type in and run the following in your R editor window or console for Apple data:

```
install.packages('quantmod')

library('quantmod')

getSymbols("AAPL")

barChart(AAPL)
```

Want to see just the last couple of weeks? You can use a command like this:

```
barChart(AAPL, subset='last 14 days')
```

```
chartSeries(AAPL, subset='last 14
days')
```

Or grab a particular date range like this:

```
barChart(A
APL['2013-04-01::2013-04-12'])
```

Quantmod is a very powerful financial analysis package, and you can read more about it on the Quantmod website.

There are many other packages with R interfaces to data sources such as twitteR for analyzing Twitter data; Quandl and rdatamarket for access to millions of data sets at Quandl and Data Market, respectively; and several for Google Analytics, including rga, RGoogleAnalytics and ganalytics.

Looking for a specific type of data to pull into R but don't know where to find it? You can try searching Quandl and Datamarket, where data can be downloaded in R format even without needing to install the site-specific packages mentioned above.

Removing unneeded data

If you're finished with variable x and want to remove it from your workspace, use the rm() remove function:

```
rm(x)
```

Saving your data

Once you've read in your data and set up your objects just the way you want them, you can save your work in several ways. It's a good idea to store your commands in a script file, so you can repeat your work if needed.

How best to save your commands? You can type them first into the RStudio script editor (top left window) instead of directly into the interactive console, so you can save the script file when you're finished. If you haven't been doing that, you can find a history of all the commands you've typed in the history tab in the top right window; select the ones you want and click the "to source" menu option to copy them into a file in the script window for saving.

You can also save your entire workspace. While you're in R, use the function:

```
save.image()
```

That stores your workspace to a file named .RData by default. This will ensure you don't lose all your work in the event of a power glitch or system reboot while you've stepped away.

When you close R, it asks if you want to save your workspace. If you say yes, the next time you start R that workspace will be loaded. That saved file will be named .RData as well. If you have different projects in different directories, each can have its own .RData workspace file.

You can also save an individual R object for later loading with the save function:

```
save(variablename, file="filename.
rda")
```

Reload it at any time with:

```
load("filename.rda")
```

Easy ways to do basic data analysis

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

Examine your data object

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

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

If you type:

```
head(mydata)
```

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

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

Or just:

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

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

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

```
tail(mydata)
```

Or:

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

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

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

```
str(mydata)
```

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

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

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

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

Various other data types return slightly different results.

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

```
colnames(mydata)
```

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

```
rownames(mydata)
```

Pull basic stats from your data frame

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

```
summary(mydata)
```

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

       x                y                z
 Min.   : 0.000   Min.   : 0.000   Min.   : 0.000
 1st Qu.: 4.710   1st Qu.: 4.720   1st Qu.: 2.910
 Median : 5.700   Median : 5.710   Median : 3.530
 Mean   : 5.731   Mean   : 5.735   Mean   : 3.539
 3rd Qu.: 6.540   3rd Qu.: 6.540   3rd Qu.: 4.040
 Max.   :10.740   Max.   :58.900   Max.   :31.800
```

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

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

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

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

```
install.packages("psych")
```

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

```
library(psych)
```

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

Now try the command:

```
describe(mydata)
```

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

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

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

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

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

```
cor(mydata)
```

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

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

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

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

In most cases, adding the argument:

```
na.rm=TRUE
```

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

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

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

```
?median
```

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

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

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

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

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

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

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

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

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

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

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

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

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

But separating the two can be more readable for beginners.

Get slices or subsets of your data

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

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

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

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

```
names(mtcars)
```

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

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

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

```
mtcars$mpg
```

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

```
dataframename$columnname
```

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

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

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

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

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

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

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

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

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

remember whether you were slicing by columns or rows.

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

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

A couple of syntax notes here:

R indexes from 1, not 0. So your first column is at [1] and not [0].

R is case sensitive everywhere. mtcars$mpg is not the same as mtcars$MPG.

mtcars[,-1] will *not* get you the last column of a data frame, the way negative indexing works in many other languages. Instead, negative indexing in R means *exclude that item*. So, mtcars[,-1] will return every column except the first one.

To create a vector of items that are not contiguous, you need to use the combine function c(). Typing mtcars[,(2,4)] without the c will not work. You need that c in there:

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

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

```
mtcars$mpg>20
```

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

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

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

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

[28] TRUE FALSE FALSE FALSE TRUE

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

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

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

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

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

using column locations, or:

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

using the column names.

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

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

```
mpg20 <- mtcars$mpg > 20
cols <- c("mpg", "hp")
```

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

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

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

Getting tired of including the name of the data set multiple times per command? If you're using only one data set *and* you

are not making any changes to the data that need to be saved, you can attach and detach a copy of the data set temporarily.

The attach() function works like this:

```
attach(mtcars)
```

So, instead of having to type:

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

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

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

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

```
detach()
```

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

Alternative to bracket notation

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
dataframename %>%
firstfunction(argument
```

```
for first function) %>%
secondfunction(argument for second
function) %>%  thirdfunction(argument
for third function)
```

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

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

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

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

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

Counting factors

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

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

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

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

```
> table(diamonds$cut)

     Fair      Good Very Good   Premium     Ideal 
     1610      4906     12082     13791     21551 
> table(diamonds$cut, diamonds$color)

              D    E    F    G    H    I    J
  Fair      163  224  312  314  303  175  119
  Good      662  933  909  871  702  522  307
  Very Good 1513 2400 2164 2299 1824 1204  678
  Premium   1603 2337 2331 2924 2360 1428  808
  Ideal     2834 3903 3826 4884 3115 2093  896
```

■ **R’s table function returns a count of each factor in your data.**

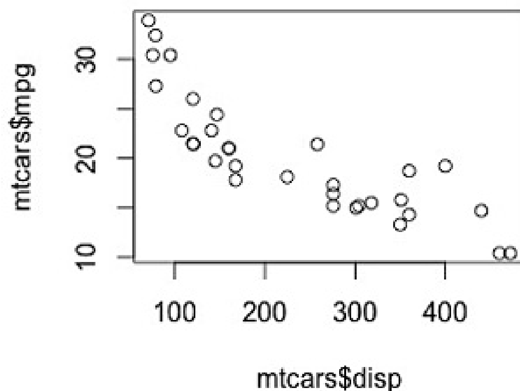
If you are interested in learning more about statistical functions in R and how to slice and dice your data, there are a number of free academic downloads with many more details. These include *Learning statistics with R* by Daniel Navarro at the University of Adelaide in Australia (500+ page PDF download,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

Painless data visualization

One of the most appealing things about R is its ability to create data visualizations with just a couple of lines of code.

For example, it takes just one line of code — and a short one at that — to plot two variables in a scatterplot. Let's use as an example the mtcars data set installed with R by default. To plot the engine displacement column *disp* on the x axis and *mpg* on y:

```
plot(mtcars$disp, mtcars$mpg)
```



■ **Default scatterplot in R.**

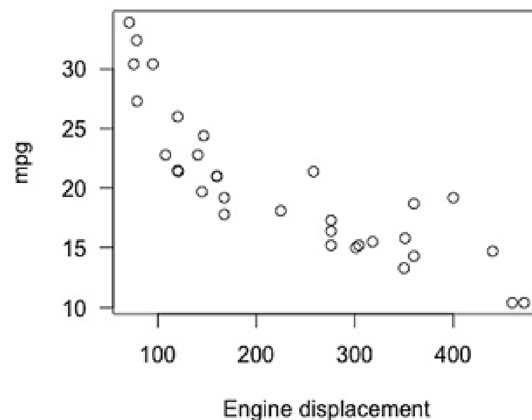
You really can't get much easier than that.

Of course that's a pretty no-frills graphic. If you'd like to label your x and y axes, use the parameters xlab and ylab. To add a main headline, such as "Page views by time of day," use the parameter main:

```
plot(mtcars$disp, mtcars$mpg,
xlab="Engine displacement",
ylab="mpg", main="MPG compared with
engine displacement")
```

If you find having the y-axis labels rotated 90 degrees annoying (as I do), you can position them for easier reading with the las=1 argument:

```
plot(mtcars$disp, mtcars$mpg,
xlab="Engine displacement",
ylab="mpg", main="MPG vs engine dis-
placement", las=1)
```



■ **Adding a main headline and axes labels to an R plot.**

What's las and why is it 1? las refers to **la**bel **s**tyle, and it's got four options. 0 is the default, with text always parallel to its axis. 1 is always horizontal, 2 is always perpendicular to the axis and 3 is always vertical. For much more on plot parameters, run the help command on par like so:

```
?par
```

In addition to the basic dataviz functionality included with standard R, there are numerous add-on packages to expand R's visualization capabilities. Some packages are for specific disciplines such as biosta-

tistics or finance; others add general visualization features.

Why use an add-on package if you don't need something discipline-specific? If you're doing more complex dataviz, or want to pretty up your graphics for presentations, some packages have more robust options. Another reason: The organization and syntax of an add-on package might appeal to you more than do the R defaults.

Using ggplot2

In particular, the ggplot2 package is quite popular and worth a look for robust visualizations. ggplot2 requires a bit of time to learn its "Grammar of Graphics" approach.

But once you've got that down, you have a tool to create *many* different types of visualizations using the same basic structure.

If ggplot2 isn't installed on your system yet, install it with the command:

```
install.packages("ggplot2")
```

You only need to do this once.

To use its functions, load the ggplot2 package into your current R session — you only need to do this once per R session — with the library() function:

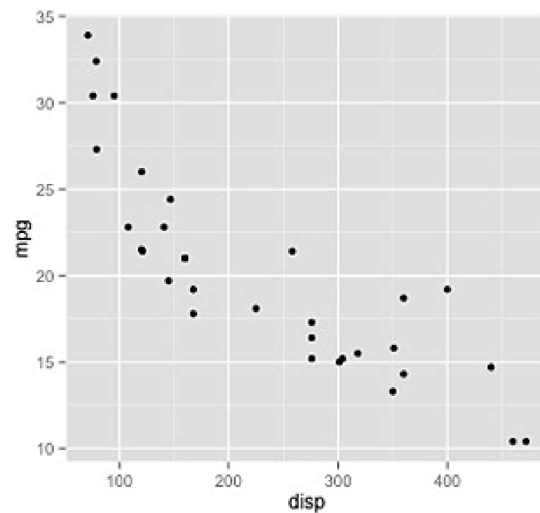
```
library(ggplot2)
```

Onto some ggplot2 examples.

ggplot2 has a "quick plot" function called qplot() that is similar to R's basic plot() function but adds some options. The basic quick plot code:

```
qplot(disp, mpg, data=mtcars)
```

generates a scatterplot.



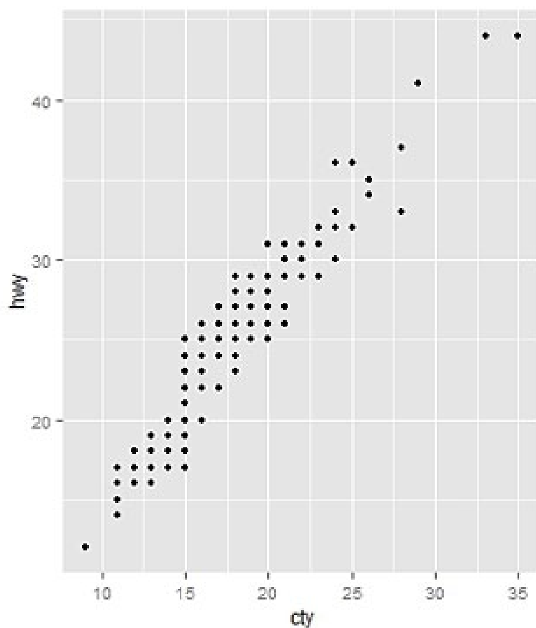
■ **A scatterplot from ggplot2 using the qplot() function.**

The qplot default starts the y axis at a value that makes sense to R. However, you might want your y axis to start at 0 so you can better see whether changes are truly meaningful (starting a graph's y axis at your first value instead of 0 can sometimes exaggerate changes).

Use the ylim argument to manually set your lower and upper y axis limits:

```
qplot(disp, mpg, ylim=c(0,35),
data=mtcars)
```

Bonus intermediate tip: Sometimes on a scatterplot you may not be sure if a point represents just one observation or multiple ones, especially if you've got data points that repeat — such as in this example that ggplot2 creator Hadley Wickham generated with the command:



■ **Some scatterplots such as this don't show the full picture because one point actually represents more than one entry in your data.**

```
qplot(cty, hwy, data=mpg)
```

The "jitter" geom parameter introduces just a little randomness in the point placement so you can better see multiple points:

```
qplot(cty, hwy, data=mpg,
geom="jitter")
```

As you might have guessed, if there's a "quick plot" function in ggplot2 there's also a more robust, full-featured plotting function. That's called ggplot() — yes, while the add-on *package* is called ggplot2, the *function* is ggplot() and not ggplot2().

The code structure for a basic graph with ggplot() is a bit more complicated than in either plot() or qplot(); it goes as follows:

```
ggplot(mtcars, aes(x=disp, y=mpg)) +
geom_point()
```

The first argument in the ggplot() function, mtcars, is fairly easy to understand — that's the data set you're plotting. But what's with "aes()" and "geom_point()"?

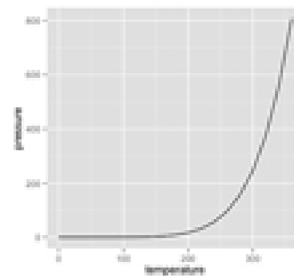
"aes" stands for aesthetics — what are considered *visual properties* of the graph. Those are things like position in space, color and shape.

"geom" is the graphing geometry you're using, such as lines, bars or the shapes of your points.

Now if "line" and "bar" also seem like aesthetic properties to you, similar to shape, well, you can either accept that's how it works or do some deep reading into the fundamentals behind the Grammar of Graphics. (Personally, I just take Wickham's word for it.)

Want a line graph instead? Simply swap out geom_point() and replace it with geom_line() , as in this example that plots temperature vs pressure in R's sample pressure data set:

```
ggplot(pressure, aes(x=temperature,
y=pressure)) + geom_line()
```



■ **Creating a line graph with ggplot2.**

It may be a little confusing here since both the data set and one of its columns are called the same thing: pressure. That first "pressure" represents the name of the data frame; the second, "y=pressure," represents the *column* named pressure.

In these examples, I set only x and y aesthetics. But there are lots more aesthetics we could add, such as color, axes and more.

You can also use the ylim argument with ggplot to change where the y axis starts. If *mydata* is the name of your data frame, *xcol* is the name of the column you want on the x axis and *ycol* is the name of the column you want on the y axis, use the ylim argument like this:

```
ggplot(mydata, aes(x=xcol, y=ycol), ylim=0) + geom_line()
```

Perhaps you'd like both lines and points on that temperature vs. pressure graph?

```
ggplot(pressure, aes(x=temperature, y=pressure)) + geom_line() + geom_point()
```

The point here (pun sort of intended) is that you can start off with a simple graphic and then add all sorts of customizations: Set the size, shape and color of the points, plot multiple lines with different colors, add labels and a ton more. See Bar and line graphs (ggplot2) for a few examples, or the The R Graphics Cookbook by Winston Chang for many more.

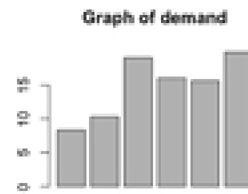
Bar graphs

To make a bar graph from the sample BOD data frame included with R, the basic R function is barplot(). So, to plot the demand column from the BOD data set on a bar graph, you can use the command:

```
barplot(BOD$demand)
```

Add main="Graph of demand" if you want a main headline on your graph:

```
barplot(BOD$demand, main="Graph of demand")
```



■ **Bar chart with R's bar plot() function.**

To label the bars on the x axis, use the names.arg argument and set it to the column you want to use for labels:

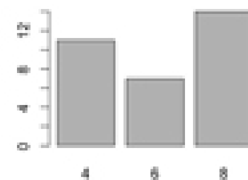
```
barplot(BOD$demand, main="Graph of demand", names.arg = BOD$Time)
```

Sometimes you'd like to graph the *counts* of a particular variable but you've got just raw data, not a table of frequencies. R's table() function is a quick way to generate counts for each factor in your data.

The R Graphics Cookbook uses an example of a bar graph for the number of 4-, 6- and 8-cylinder vehicles in the mtcars data set. Cylinders are listed in the cyl column, which you can access in R using mtcars$cyl.

Here's code to get the count of how many entries there are by cylinder with the table() function; it stores results in a variable called cylcount:

```
cylcount <- table(mtcars$cyl)
```



■ **Creating a bar plot.**

That creates a table called cylcount containing:

4 6 8

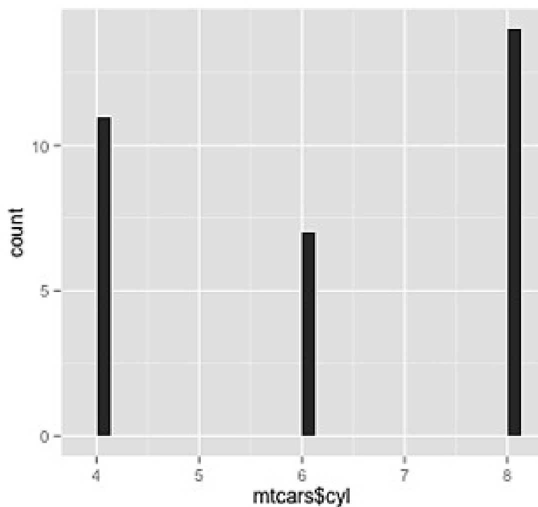
11 7 14

Now you can create a bar graph of the cylinder count:

```
barplot(cylcount)
```

ggplot2's qplot() quick plotting function can also create bar graphs:

```
qplot(mtcars$cyl)
```



■ **What happens to your bar chart when you don't instruct R not to plot continuous variables.**

However, this defaults to an assumption that 4, 6 and 8 are part of a variable set that could run from 4 *through* 8, so it shows blank entries for 5 and 7.

To treat cylinders as distinct groups — that is, you've got a group with 4 cylinders, a group with 6 and a group with 8, not the possibility of entries anywhere between 4 and 8 — you want cylinders to be treated as a statistical factor:

```
qplot(factor(mtcars$cyl))
```

To create a bar graph with the more robust ggplot() function, you can use syntax such as:

```
ggplot(mtcars, aes(factor(cyl))) +
geom_bar()
```

Histograms

Histograms work pretty much the same, except you want to specify how many buckets or bins you want your data to be separated into. For base R graphics, use:

```
hist(mydata$columnName, breaks = n)
```

where columnName is the name of your column in a mydata dataframe that you want to visualize, and n is the number of bins you want.

The ggplot2 commands are:

```
qplot(columnName, data=mydata,
binwidth=n)
```

For quick plots and, for the more robust ggplot():

```
ggplot(mydata, aes(x=columnName)) +
geom_histogram(binwidth=n)
```

You may be starting to see strong similarities in syntax for various ggplot() examples. While the ggplot() function is somewhat less intuitive, once you wrap your head around its general principles, you can do other types of graphics in a similar way.

Additional graphics options

There are many more graphics types in R than these few I've mentioned. Boxplots, a statistical staple showing minimum and maximum, first and third quartiles and median, have their own function called, intuitively, boxplot(). If you want to see a boxplot of the mpg column in the mtcars data frame it's as simple as:

```
boxplot(mtcars$mpg)
```

To see side-by-side boxplots in a single plot, such as the x, y and z measurements of all the diamonds in the diamonds sample data set included in ggplot2:

```
boxplot(diamonds$x, diamonds$y,
diamonds$z)
```

Creating a heat map in R is more complex but not ridiculously so. There's an easy-to-follow tutorial on Flowing Data.

You can do graphical correlation matrices with the corrplot add-on package and generate numerous probability distributions. See some of the links here or in the resources section to find out more.

Using color

Looking at nothing but black and white graphics can get tiresome after a while. Of course, there are numerous ways of using color in R.

Colors in R have both names and numbers as well as the usual RGB hex code, HSV (hue, saturation and value) specs and others. And when I say "names," I don't mean just the usual "red," "green," "blue," "black" and "white." R has *657 named colors*. The colors() or colours() function — R does not discriminate against either American or British English — gives you a list of all of them. If you want to see what they look like, not just their text names, you can get a full, multi-page PDF chart with color numbers, colors names and swatches, sorted in various ways. Or you can find just the names and color swatches for each.

There are also R functions that automatically generate a vector of n colors using a specific color palette such as "rainbow" or "heat":

```
rainbow(n)
```

```
heat.colors(n)
```

```
terrain.colors(n)
```

```
topo.colors(n)
```

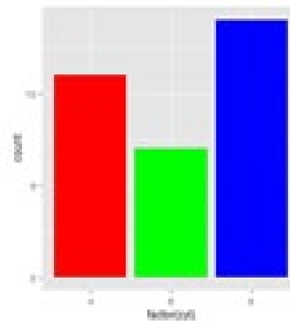
```
cm.colors(n)
```

So, if you want five colors from the rainbow palette, use:

```
rainbow(5)
```

For many more details, check the help command on a palette such as:

```
?rainbow
```



■ **Using three colors in the R rainbow palette.**

Now that you've got a list of colors, how do you get them in your graphic? Here's one way. Say you're drawing a 3-bar barchart using ggplot() and want to use 3 colors from the rainbow palette. You can create a 3-color vector like:

```
mycolors <- rainbow(3)
```

Or for the heat.colors pallette:

```
mycolors <- heat.colors(3)
```

Now instead of using the geom_bar() function without any arguments, add fill=mycolors to geombar() like this:

```
ggplot(mtcars, aes(x=factor(cyl))) +
geom_bar(fill=mycolors)
```

You don't need to put your list of colors in a separate variable, by the way; you can merge it all in a single line of code such as:

```
ggplot(mtcars, aes(x=factor(cyl))) +
geom_bar(fill=rainbow(3))
```

But it may be easier to separate the colors out if you want to create your own list of colors instead of using one of the defaults.

The basic R plotting functions can also accept a vector of colors, such as:

```
barplot(BOD$demand, col=rainbow(6))
```

You can use a single color if you want all the items to be one color (but not monochrome), such as

```
barplot(BOD$demand, col="royalblue3")
```

Chances are, you'll want to use color to show certain characteristics of your data, as opposed to simply assigning random colors in a graphic. That goes a bit beyond beginning R, but to give one example, say you've got a vector of test scores:

```
testscores <- c(96, 71, 85, 92, 82,
78, 72, 81, 68, 61, 78, 86, 90)
```

You can do a simple barplot of those scores like this:

```
barplot(testscores)
```

And you can make all the bars blue like this:

```
barplot(testscores, col="blue")
```

But what if you want the scores 80 and above to be blue and the lower scores to be red? To do this, create a vector of colors of the same length and in the same order as your data, adding a color to the vector based on the data. In other words, since the first test score is 96, the first color in your color vector should be blue; since the second score is 71, the second color in your color vector should be red; and so on.

Of course, you don't want to create that color vector manually! Here's a statement that will do so:

```
testcolors <- ifelse(testscores >=
80, "blue", "red")
```

If you've got any programming experience, you might guess that this creates a vector that loops through the testscores data and runs the conditional statement: 'If this entry in testscores is greater than or equal to 80, add "blue" to the testcolors vector; otherwise add "red" to the testcolors vector.'

Now that you've got the list of colors properly assigned to your list of scores, just add the testcolors vector as your desired color scheme:

```
barplot(testscores, col=testcolors)
```

Note that the *name of a color* must be in quotation marks, but a *variable name* that holds a list of colors should **not** be within quote marks.

Add a graph headline:

```
barplot(testscores, col=testcolors,
main="Test scores")
```

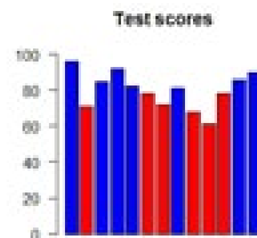
And have the y axis go from 0 to 100:

```
barplot(testscores, col=testcolors,
main="Test scores", ylim=c(0,100))
```

Then use las-1 to style the axis label to be horizontal and not turned 90 degrees vertical:

```
barplot(testscores, col=testcolors,
main="Test scores", ylim=c(0,100),
las=1)
```

And you've got a color-coded bar graph.



■ **A color-coded bar graph.**

By the way, if you wanted the scores sorted from highest to lowest, you could have set your original testscores variable to:

```
testscores <- sort(c(96, 71, 85, 92,
82, 78, 72, 81, 68, 61, 78, 86, 90),
decreasing = TRUE)
```

The sort() function defaults to ascending sort; for descending sort you need the additional argument: decreasing = TRUE.

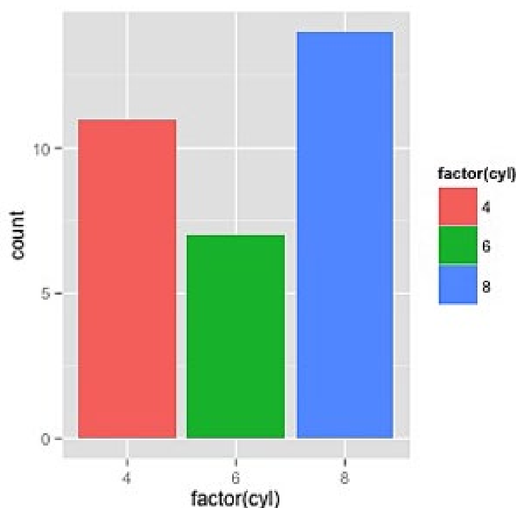
If that code above is starting to seem unwieldy to you as a beginner, break it into two lines for easier reading, and perhaps also set a new variable for the sorted version:

```
testscores <- c(96, 71, 85, 92, 82,
78, 72, 81, 68, 61, 78, 86, 90)

testscores_sorted <- sort(testscores,
decreasing = TRUE)
```

If you had scores in a data frame called results with one column of student names called students and another column of scores called testscores, you could use the ggplot2 package's ggplot() function as well:

```
ggplot(results, aes(x=students,
y=testscores)) + geom_
bar(fill=testcolors, stat =
"identity")
```



Coloring bars by factor.

Why stat = "identity"? That's needed here to show that the y axis represents a numerical value as opposed to an item count.

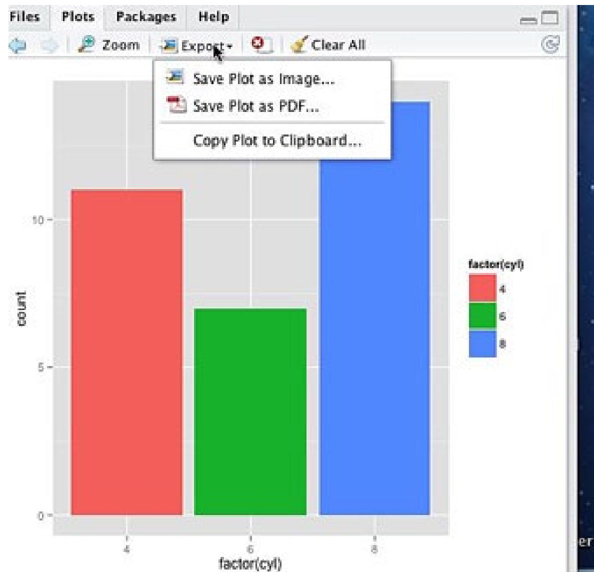
ggplot2's qplot() also has easy ways to color bars by a factor, such as number of cylinders, and then automatically generate a legend. Here's an example of graph counting the number of 4-, 6- and 8-cylinder cars in the mtcars data set:

```
qplot(factor(cyl), data=mtcars,
geom="bar", fill=factor(cyl))
```

But, as I said, we're getting somewhat beyond a beginner's overview of R when coloring by factor. For a few more examples and details for many of the themes covered here, you might want to see the online tutorial Producing Simple Graphs with R. For more on graphing with color, check out a source such as the R Graphics Cookbook. The ggplot2 documentation also has a lot of examples, such as this page for bar geometry.

Exporting your graphics

You can save your R graphics to a file for use outside the R environment. RStudio has an export option in the plots tab of the bottom right window.



■ **Exporting your graphics from RStudio.**

If you are using "plain" R in Windows, you can also right-click the graphics window to save the file.

To save a plot with R commands and not point-and-click, first create a container for your image using a function such as jpeg(), png(), svg() or pdf(). Those functions need to have a file name as one argument and optionally width and height, such as:

```
jpeg("myplot.jpg", width=350,
height=420)
```

Generate the plot with whatever graphics functions it requires, such as:

```
barplot(BOD$demand, col=rainbow(6))
```

And then issue the command:

```
dev.off()
```

That will save your graphic to its container.

If you are using ggplot2, you can also use the function ggsave(), which defaults to saving the last plot you created using ggplot2 at the size you displayed it. Based on the filename you pass to the ggsave() function, it will save in the appropriate format — myplot.jpg saves as a JPEG, myplot.png saves as a PNG and so on.

One final note: If you're working in RStudio and would like to see a larger version of your plot, click the Zoom button and a larger window with the current graphic will open. And, also in RStudio, to see previous plots from your session, click the back arrow.

Syntax quirks you'll want to know

I mentioned at the outset that R syntax is a bit quirky, especially if your frame of reference is, well, pretty much any other programming language. Here are some unusual traits of the language you may find useful to understand as you embark on your journey to learn R.

Assigning values to variables

In pretty much every other programming language I know, the equals sign assigns a certain value to a variable. You know, x = 3 means that x now holds the value of 3.

Not in R. At least, not necessarily.

In R, the primary assignment operator is `<-` as in:

```
x <- 3
```

But not:

```
x = 3
```

To add to the potential confusion, the equals sign actually *can* be used as an assignment operator in R — but not all the time. When can you use it and when can you not?

The best way for a beginner to deal with this is to use the preferred assignment operator <- and forget that equals is ever allowed. Hey, if it's good enough for [Google's R style guide]() — they advise not using equals to assign values to variables — it's good enough for me.

(If this isn't a good enough explanation for *you*, however, and you really really want to know the ins and outs of R's 5 — yes, count 'em, 5 — assignment options, check out the [R manual's Assignment Operators page]().)

One more note about variables: R is a case-sensitive language. So, variable x is not the same as X. That applies to pretty much everything in R; for example, the function subset() is not the same as Subset().

c is for combine (or concatenate, and sometimes convert/coerce.)

When you create an array in most programming languages, the syntax goes something like this:

```
myArray = array(1, 1, 2, 3, 5, 8);
```

Or:

```
int myArray = {1, 1, 2, 3, 5, 8};
```

Or maybe:

```
myArray = [1, 1, 2, 3, 5, 8]
```

In R, though, there's an extra piece: To put multiple values into a single variable, you need the c() function, such as:

```
my_vector <- c(1, 1, 2, 3, 5, 8)
```

If you forget that c, you'll get an error. When you're starting out in R, you'll probably see errors relating to leaving out that c() a *lot*. (At least I certainly did.)

And now that I've stressed the importance of that c() function, I (reluctantly) will tell you that there's a case when you can leave

it out — if you're referring to consecutive values in a range with a colon between minimum and maximum, like this:

```
my_vector <- (1:10)
```

I bring up this exception because I've run into that style quite a bit in R tutorials and texts, and it can be confusing to see the c required for *some* multiple values but not others. Note that it won't *hurt* anything to use the c with a colon-separated range, though, even if it's not required, such as:

```
my_vector <- c(1:10)
```

One more very important point about the c() function: It assumes that everything in your vector is of the same data type — that is, all numbers or all characters. If you create a vector such as:

```
my_vector <- c(1, 4, "hello", TRUE)
```

You will *not* have a vector with two integer objects, one character object and one logical object. Instead, c() will do what it can to convert them all into all the same object type, in this case all character objects. So my_vector will contain "1", "4", "hello" and "TRUE". In other words, c() is also for "convert" or "coerce."

To create a collection with multiple object types, you need a *list*, not a vector. You create a list with the list() function, not c(), such as:

```
My_list <- list(1,4,"hello", TRUE)
```

Now you've got a variable that holds the number 1, the number 4, the character object "hello" and the logical object TRUE.

Loopless loops

Iterating through a collection of data with loops like "for" and "while" is a cornerstone of many programming languages. That's not the R way, though. While R does have [for, while and repeat loops](), you'll more likely see operations applied to a data collection using *apply()* functions or by using the *plyr()* add-on package functions.

But first, some basics.

If you've got a vector of numbers such as:

```
my_vector <- c(7,9,23,5)
```

and, say, you want to multiply each by 0.01 to turn them into percentages, how would you do that? You don't need a for, foreach or while loop. Instead, you can create a new vector called my_pct_vectors like this:

```
my_pct_vector <- my_vector * 0.01
```

Performing a mathematical operation on a vector variable will automatically loop through each item in the vector.

Typically in data analysis, though, you want to apply functions to subsets of data: Finding the mean salary by job title or the standard deviation of property values by community. The apply() function group and plyr add-on package are designed for that.

There are more than half a dozen functions in the apply family, depending on what type of data object is being acted upon and what sort of data object is returned. "These functions can sometimes be frustratingly difficult to get working exactly as you intended, especially for newcomers to R," says a blog post at Revolution Analytics, which focuses on enterprise-class R.

Plain old apply() runs a function on either every row or every column of a 2-dimensional matrix where all columns are the same data type. For a 2-D matrix, you also need to tell the function whether you're applying by rows or by columns: Add the argument 1 to apply by row or 2 to apply by column. For example:

```
apply(my_matrix, 1, median)
```

returns the median of every row in my_matrix and

```
apply(my_matrix, 2, median)
```

calculates the median of every column.

Other functions in the apply() family such as lapply() or tapply() deal with different input/output data types. Australian statistical bioinformatician Neal F.W. Saunders has a nice brief introduction to apply in R in a blog post if you'd like to find out more and see some examples. (In case you're wondering, bioinformatics involves issues around storing, retrieving and organizing biological data, not just analyzing it.)

Many R users who dislike the the apply functions don't turn to for-loops, but instead install the plyr package created by Hadley Wickham. He uses what he calls the "split-apply-combine" model of dealing with data: Split up a collection of data the way you want to operate on it, apply whatever function you want to each of your data group(s) and then combine them all back together again.

The plyr package is probably a step beyond this basic beginner's guide; but if you'd like to find out more about plyr, you can head to Wickham's plyr website. There's also a useful slide presentation on plyr in PDF format from Cosma Shalizi, an associate professor of statistics at Carnegie Mellon University, and Vincent Vu. Another PDF presentation on plyr is from an introduction to R workshop at Iowa State University.

R data types in brief (very brief)

Should you learn about *all* of R's data types and how they behave right off the bat, as a beginner? If your goal is to be an R ninja then, yes, you've got to know the ins and outs of data types. But my assumption is that you're here to try generating quick plots and stats before diving in to create complex code.

So, to start off with the basics, here's what I'd suggest you keep in mind for now: R has multiple data types. Some of them are especially important when doing basic data work. And some functions that are quite useful for doing your basic data work require your data to be in a particular type and structure.

More specifically, R has the "Is it an integer or character or true/false?" data type, the basic building blocks. R has several of these including integer, numeric, character and logical. Missing values are represented by NaN (if a mathematical function won't work properly) or NA (missing or unavailable).

As mentioned in the prior section, you can have a vector with multiple elements of the same type, such as:

1, 5, 7

or

"Bill", "Bob", "Sue"

>

A single number or character string is also a vector — a vector of 1. When you access the value of a variable that's got just one value, such as 73 or "Learn more about R at Computerworld.com," you'll also see this in your console before the value:

```
[1]
```

That's telling you that your screen printout is starting at vector item number one. If you've got a vector with lots of values so the printout runs across multiple lines, each line will start with a number in brack-

ets, telling you which vector item number that particular line is starting with. (See the screen shot, below.)

```
  [1] 200.1 199.5 199.4 198.9 199.0 200.2 198.6 200.0 200.3 201.2
 [11] 201.6 201.5 201.5 203.5 204.9 207.1 210.5 210.5 209.8 208.8
 [21] 209.5 213.2 213.7 215.1 218.7 219.8 220.5 223.8 222.8 223.8
 [31] 221.7 222.3 220.8 219.4 220.1 220.6 218.9 217.8 217.7 215.0
 [41] 215.3 215.9 216.7 216.7 217.7 218.7 222.9 224.9 222.2 220.7
 [51] 220.0 218.7 217.0 215.9 215.8 214.1 212.3 213.9 214.6 213.6
 [61] 212.1 211.4 213.1 212.9 213.3 211.5 212.3 213.0 211.0 210.7
 [71] 210.1 211.4 210.0 209.7 208.8 208.8 208.8 210.6 211.9 212.8
 [81] 212.5 214.8 215.3 217.5 218.8 220.7 222.2 226.7 228.4 233.2
 [91] 235.7 237.1 240.6 243.8 245.3 246.0 246.3 247.7 247.6 247.8
[101] 249.4 249.0 249.9 250.5 251.5 249.0 247.6 248.8 250.4 250.7
[111] 253.0 253.7 255.0 256.2 256.0 257.4 260.4 260.0 261.3 260.4
[121] 261.6 260.8 259.8 259.0 258.9 257.4 257.7 257.9 257.4 257.3
[131] 257.6 258.9 257.8 257.7 257.2 257.5 256.8 257.5 257.0 257.6
[141] 257.3 257.5 259.6 261.1 262.9 263.3 262.8 261.8 262.2 262.7
```

■ **If you've got a vector with lots of values so the printout runs across multiple lines, each line will start with a number in brackets, telling you which vector item number that particular line is starting with.**

If you want to mix numbers and strings or numbers and TRUE/FALSE types, you need a list. (If you don't create a list, you may be unpleasantly surprised that your variable containing (3, 8, "small") was turned into a vector of characters ("3", "8", "small")).

And by the way, R assumes that 3 is the same class as 3.0 — numeric (i.e., with a decimal point). If you want the *integer* 3, you need to signify it as 3L or with the as.integer() function. In a situation where this matters to you, you can check what type of number you've got by using the class() function:

```
class(3)

class(3.0)

class(3L)

class(as.integer(3))
```

There are several as() functions for converting one data type to another, including as.character(), as.list() and as.data.frame().

R also has special vector and list types that are of special interest when analyzing data, such as matrices and data frames. A matrix has rows and columns; you can find a matrix dimension with dim() such as

```
dim(my_matrix)
```

A matrix needs to have all the same data type in every column, such as numbers everywhere.

Data frames are like matrices except one column can have a different data type from another column, and each column must have a name. If you've got data in a format that might work well as a database table (or well-formed spreadsheet table), it will also probably work well as an R data frame.

In a data frame, you can think of each row as similar to a database record and each column like a database field. There are lots of useful functions you can apply to data frames, some of which I've gone over in earlier sections, such as summary() and the psych package's describe().

And speaking of quirks: There are several ways to find an object's underlying data type, but not all of them return the same value. For example, class() and str() will return *data.frame* on a data frame object, but mode() returns the more generic *list*.

If you'd like to learn more details about data types in R, you can watch this video lecture by Roger Peng, associate professor of biostatistics at the Johns Hopkins Bloomberg School of Public Health:

■ **Roger Peng, associate professor of biostatistics at the Johns Hopkins Bloomberg School of Public Health, explains data types in R.**

One more useful concept to wrap up this section — hang in there, we're almost done: factors. These represent *categories* in your data. So, if you've got a data frame with employees, their department and their salaries, salaries would be numerical data and employees would be characters (strings in many other languages); but you'd likely want department to be a *factor* — in other words, a category you may want to group or model your data by. Factors can be unordered, such as department, or ordered, such as "poor", "fair", "good" and "excellent."

R command line differs from the Unix shell

When you start working in the R environment, it looks quite similar to a Unix shell. In fact, some R command-line actions behave as you'd expect if you come from a Unix environment, but others don't.

Want to cycle through your last few commands? The up arrow works in R just as it does in Unix — keep hitting it to see prior commands.

The list function, ls(), will give you a list, but not of files as in Unix. Rather, it will provide a list of objects in your current R session.

Want to see your current working directory? pwd just throws an error; what you want is getwd().

rm(my_variable) will delete a variable from your current session.

R does include a Unix-like grep() function. For more on using grep in R, see this brief writeup on [Regular Expressions with The R Language](#) at regular-expressions.info.

Terminating your R expressions

R doesn't need semicolons to end a line of code (although it's possible to put multiple commands on a single line separated by semicolons, you don't see that very often). Instead, R uses line breaks (i.e., new line characters) to determine when an expression has ended.

What if you want one expression to go across multiple lines? The R interpreter tries to guess if you mean for it to continue to the next line: If you obviously haven't finished a command on one line, it will assume you want to continue instead of throwing an error. Open some parentheses without closing them, use an open quote without a closing one or end a line with an operator like + or - and R will wait to execute your command until it comes across the expected closing character and the command otherwise looks finished.

Syntax cheating: Run SQL queries in R

If you've got SQL experience and R syntax starts giving you a headache — especially when you're trying to figure out how to

get a subset of data with proper R syntax — you might start longing for the ability to run a quick SQL SELECT command query your data set.

You can.

The add-on package sqldf lets you run SQL queries on an R data frame (there are separate packages allowing you to connect R with a local database). Install and load sqldf, and then you can issue commands such as:

```
sqldf("select * from mtcars where mpg > 20 order by mpg desc")
```

This will find all rows in the mtcars sample data frame that have an mpg greater than 20, ordered from highest to lowest mpg.

Most R experts will discourage newbies from "cheating" this way: Falling back on SQL makes it less likely you'll power through learning R syntax. However, it's there for you in a pinch — or as a useful way to double-check whether you're getting back the expected results from an R expression.

Examine and edit data with a GUI

And speaking of cheating, if you don't want to use the command line to examine and edit your data, R has a couple of options. The edit() function brings up an editor where you can look at and edit an R object, such as

```
edit(mtcars)
```

Data Editor

File Edit Help

	row.names	mpg	cyl	disp	hp	drat	wt	qsec	vs
1	Mazda RX4	21	6	160	110	3.9	2.62	16.46	0
2	Mazda RX4 Wag	21	6	160	110	3.9	2.875	17.02	0
3	Datsun 710	22.8	4	108	93	3.85	2.32	18.61	1
4	Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1
5	Hornet Sportabout	18.7	8	360	175	3.15	3.44	17.02	0
6	Valiant	18.1	6	225	105	2.76	3.46	20.22	1
7	Duster 360	14.3	8	360	245	3.21	3.57	15.84	0
8	Merc 240D	24.4	4	146.7	62	3.69	3.19	20	1
9	Merc 230	22.8	4	140.8	95	3.92	3.15	22.9	1
10	Merc 280	19.2	6	167.6	123	3.92	3.44	18.3	1
11	Merc 280C	17.8	6	167.6	123	3.92	3.44	18.9	1
12	Merc 450SE	16.4	8	275.8	180	3.07	4.07	17.4	0
13	Merc 450SL	17.3	8	275.8	180	3.07	3.73	17.6	0
14	Merc 450SLC	15.2	8	275.8	180	3.07	3.78	18	0
15	Cadillac Fleetwood	10.4	8	472	205	2.93	5.25	17.98	0
16	Lincoln Continental	10.4	8	460	215	3	5.424	17.82	0
17	Chrysler Imperial	14.7	8	440	230	3.23	5.345	17.42	0
18	Fiat 128	32.4	4	78.7	66	4.08	2.2	19.47	1
19	Honda Civic	30.4	4	75.7	52	4.93	1.615	18.52	1

■ **Invoking R's data editing window with the edit() function.**

This can be useful if you've got a data set with a lot of columns that are wrapping in the small command-line window. However, since there's no way to save your work as you go along — changes are saved only when you close the editing window — and there's no command-history record of what you've done, the edit window probably isn't your best choice for editing data in a project where it's important to repeat/reproduce your work.

In RStudio you can also examine a data object (although not edit it) by clicking on it in the workspace tab in the upper right window.

Saving and exporting your data

In addition to saving your entire R workspace with the save.image() function and various ways to save plots to image files, you can save individual objects for use in other software. For example, if you've got a data frame just so and would like to share it with colleagues as a tab- or comma-delimited file, say for importing into a spreadsheet, you can use the command:

```
write.table(myData, "testfile.txt",
sep="\t")
```

This will export all the data from an R object called myData to a tab-separated file called testfile.txt in the current working directory. Changing sep="\t" to sep="c" will generated a comma-separated file and so on.

60+ R resources to improve your data skills

This list was originally published as part of the Computerworld Beginner's Guide to R but has since been expanded to also include resources for advanced beginner and intermediate users.

These websites, videos, blogs, social media/ communities, software and books/ebooks can help you do more with R.; my favorites are listed in bold.

Books and e-books

R Cookbook. Like the rest of the O'Reilly Cookbook series, this one offers how-to "recipes" for doing lots of different tasks, from the basics of R installation and creating simple data objects to generating probabilities, graphics and linear regressions. It has the added bonus of being well written. If you like learning by example or are seeking a good R reference book, this is well worth adding to your reference library. By Paul Teetor, a quantitative developer working in the financial sector.

R Graphics Cookbook. If you want to do beyond-the-basics graphics in R, this is a useful resource both for its graphics recipes and brief introduction to ggplot2. While this goes way beyond the graphics capabilities that *I* need in R, I'd recommend this if you're looking to move beyond advanced-beginner plotting. By Winston Chang, a software engineer at RStudio.

R in Action: Data analysis and graphics with R. This book aims at all levels of users, with sections for beginning, intermediate and advanced R ranging from "Exploring R data structures" to running regressions and conducting factor analyses. The beginner's section may be a bit tough to follow if you haven't had *any* exposure to R, but it offers a good foundation in data types, imports and reshaping once you've had a bit of experience. There are some particularly useful explanations and examples for aggregating, restructuring and subsetting data, as well as a lot of applied statistics. Note that if your interest in graphics is learning ggplot2, there's relatively little on that here compared with base R graphics and the lattice package. You can see an excerpt from the book online: Aggregation and restructuring data. By Robert I. Kabacoff.

The Art of R Programming. For those who want to move beyond using R "in an ad hoc way ... to develop[ing] software in R." This is best if you're already at least moderately proficient in another programming language. It's a good resource for systematically learning fundamentals such as types of objects, control statements (unlike many R purists, the author doesn't actively discourage for loops), variable scope, classes and debugging — in fact, there's nearly as large a chapter on debugging as there is on graphics. With some robust examples of solving real-world statistical problems in R. By Norman Matloff.

R in a Nutshell. A reasonably readable guide to R that teaches the language's fundamentals — syntax, functions, data structures and so on — as well as how-to statistical and graphics tasks. Useful if you

want to start writing robust R programs, as it includes sections on functions, object-oriented programming and high-performance R. By Joseph Adler, a senior data scientist at LinkedIn.

Visualize This. Note; Most of this book is *not* about R, but there are several examples of visualizing data with R. And there's so much other interesting info here about how to tell stories with data that it's worth a read. By Nathan Yau, who runs the popular Flowing Data blog and whose doctoral dissertation was on "personal data collection and how we can use visualization to learn about ourselves."

R For Dummies. I haven't had a chance to read this one, but it's garnered some good reviews on Amazon.com. If you're familiar with the Dummies series and have found them helpful in the past, you might want to check this one out. You can get a taste of the authors' style in the Programming in R section of Dummies.com, which has more than a 100 short sections such as How to construct vectors in R and How to use the apply family of functions in R. By Joris Meys and Andrie de Vries.

Introduction to Data Science. It's highly readable, packed with useful examples and free — what more could you want? This e-book isn't technically an "R book," but it uses R for all of its examples as it teaches concepts of data analysis. If you're familiar with that topic you may find some of the explanations rather basic, but there's still a lot of R code for things like analyzing tweet rates (including a helpful section on how to get Twitter OAuth authorization working in R), simple map mashups and basic linear regression. Although Stanton calls this an "electronic textbook," Introduction to Data Science has a conversational style that's pleasantly non-textbook like. There used to be a downloadable PDF, but now the only versions are for OS X or iOS.

R for Everyone. Author Jared P. Lander promises to go over "20% of the functionality needed to accomplish 80% of the work." And in fact, topics that are actually covered, are covered pretty well; but be warned that some items appearing in the table of contents can be a little thin. This is still a well-organized reference, though, with information that beginning and intermediate users might want to know: importing data, generating graphs, grouping and reshaping data, working with basic stats and more.

Statistical Analysis With R: Beginner's Guide. This book has you "pretend" you're a strategist for an ancient Chinese kingdom analyzing military strategies with R. If you find that idea hokey, move along to see another resource; if not, you'll get a beginner-level introduction to various tasks in R, including tasks you don't always see in an intro text, such as multiple linear regressions and forecasting. Note: My early e-version had a considerable amount of bad spaces in my Kindle app, but it was still certainly readable and usable.

Reproducible Research with R and RStudio. Although categorized as a "bioinformatics" textbook (and priced that way — even the Kindle edition is more than $50), this is more general advice on steps to make sure you can document and present your work. This includes numerous sections on creating report documents using the knitr package, LaTeX and Markdown — tasks not often covered in-depth in general R books. The author has posted source code for generating the book on GitHub, though, if you want to create an electronic version of it yourself.

Exploring Everyday Things with R and Ruby. This book oddly goes from a couple of basic introductory chapters to some fairly robust, beyond-beginner programming examples; for those who are just starting to code, much of the book may be tough to follow at the outset. However, the intro to R is one of the better ones I've read, including lot of language fundamentals and basics of graphing with ggplot2. Plus experienced programmers can see how author Sau Sheong Chang splits up tasks between a general language like Ruby and the statistics-focused R.

Online references

4 data wrangling tasks in R for advanced beginners. This follow-up to our Beginner's Guide outlines how to do several specific data tasks in R: add columns to an existing data frame, get summaries, sort results and reshape data. With sample code and explanations.

Data manipulation tricks: Even better in R. From working with dates to reshaping data to if-then-else statements, see how to perform common data munging tasks. You can also download these R tips & tricks as a PDF (free Insider registration required).

Cookbook for R. Not to be confused with the R Cookbook *book* mentioned above, this website by software engineer Winston Chang (author of the R *Graphics* Cookbook) offers how-to's for tasks such as data input and output, statistical analysis and creating graphs. It's got a similar format to an O'Reilly Cookbook; and while not as complete, can be helpful for answering some "How do I do that?" questions.

Quick-R. This site has a fair amount of samples and brief explanations grouped by major category and then specific items. For example, you'd head to "Stats" and then "Frequencies and crosstabs" to get an explainer of the table() function. This ranges from basics (including useful how-to's for customizing R startup) through beyond-beginner statistics (matrix algebra, anyone?) and graphics. By Robert I. Kabacoff, author of R in Action.

R Reference Card. If you want help remembering function names and formats for various tasks, this 4-page PDF is quite useful despite its age (2004) and the fact that a link to what's supposed to be the latest version no longer works. By Tom Short, an engineer at the Electric Power Research Institute.

A short list of R the most useful commands. Commands grouped by function such as input, "moving around" and "statistics and transformations." This offers minimal explanations, but there's also a link to a longer guide to Using R for psychological research. HTML format makes it easy to cut and paste commands. Also somewhat old, from 2005. By William Revelle, psychology professor at Northwestern University.

R Graph Catalog. Lots of graph and other plot examples, easily searchable and each with downloadable code. All are made with ggplot2 based on visualization ideas in Creating More Effective Graphs. Maintained by Joanna Zhao and Jennifer Bryan.

Beautiful Plotting in R: A ggplot2 Cheatsheet. Easy to read with a lot of useful information, from starting with default plots to customizing title, axes, legends; creating multi-panel plots and more. By Zev Ross.

Frequently Asked Questions about R. Some basics about reading, writing, sorting and shaping data as well as a lineup of how to do various statistical operations and a few specialized graphics such as spaghetti plots.

From UCLA's Institute for Digital Research and Education.

R Reference Card for Data Mining. Includes examples and other documentation. including a substantial portion of his book R and Data Mining published by Elsevier in 2012. By Yanchang Zhao.

Spatial Cheat Sheet. For those doing GIS and spatial analysis work, this list offers some key functions and packages for working with spatial vector and raster data. By Barry Stephen Rowlingson at Lancaster University in the U.K.

Online tools

Web interface for ggplot2. This online tool by UCLA Ph.D. candidate Jeroen Ooms creates an interactive front end for ggplot2, allowing users to input tasks they want to do and get a plot plus R code in return. Useful for those who want to learn more about using ggplot2 for graphics without having to read through lengthy documentation.

Ten Things You Can Do in R That You Would've Done in Microsoft Excel. From the R for Dummies Web site, these code samples aim to help Excel users feel more comfortable with R.

Videos

Twotorials**.** You'll either enjoy these snappy 2-minute "twotorial" videos or find them, oh, corny or over the top. I think they're both informative and fun, a welcome antidote to the typically dry how-to's you often find in statistical programming. Analyst Anthony Damico takes on R in 2-minute chunks, from "how to create a variable with R" to "how to plot residuals from a regression in R;" he also tackles an occasional problem such as "how to calculate your ten, fifteen, or twenty thousandth day on earth with R." I'd strongly recommend giving this a look if textbook-style instruction leaves you cold.

■ **Sample "Twotorial" video.**

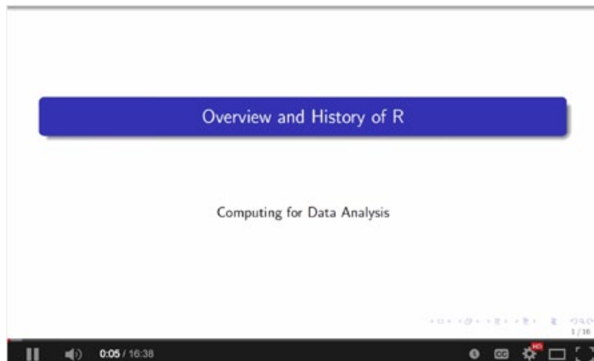
Google Developers' Intro to R. This series of 21 short YouTube videos includes some basic R concepts, a few lessons on reshaping data and some info on loops. In addition, six videos focus on a topic that's often missing in R intros: working with and writing your own functions. This YouTube playlist offers a good programmer's introduction to the language — just note that if you're looking to learn more about visualizations with R, that's not one of the topics covered.

■ **This video in the Google Developers' R series introduces functions in R.**

Up and Running with R. This lynda.com video class covers the basics of topics such as using the R environment, reading in data, creating charts and calculating

statistics. The curriculum is limited, but presenter Barton Poulson tries to explain what he's doing and why, not simply run commands. He also has a more in-depth 6-hour class, R Statistics Essential Training. Lynda.com is a subscription service that starts at $25/month, but several of the videos are available free for you to view and see if you like the instruction style, and there's a 7-day free trial available.

Coursera: Computing for Data Analysis. Coursera's free online classes are time-sensitive: You've got to enroll while they're taking place or you're out of luck. However, if there's no session starting soon, instructor Roger Peng, associate professor of biostatistics at Johns Hopkins University, posted his lectures on YouTube; Revolution Analytics then collected them on a handy single page. While I found some of these a bit difficult to follow at times, they are packed with information, and you may find them useful.



■ Intro video for the Coursera Computing for Data Analysis course

Coursera: Data Analysis. This was more of an applied statistics class that uses R as opposed to one that teaches R; but if you've got the R basics down and want to see it in action, this might be a good choice. There are no upcoming scheduled sessions for this at Coursera, but instructor Jeff Leek — an assistant professor of biostatistics at Johns Hopkins, posted his lecture videos on YouTube, and Revolution Analytics collected links to them all by week.

■ Intro video for Coursera Data Analysis online class

Coursera: Statistics One. If you don't mind going through a full, 12-week stats course along with learning R, Princeton University senior lecturer Andrew Conway's class includes an introduction to R. "All the examples and assignments will involve writing code in R and interpreting R output," says the course description. You can check the Coursera link to see if and when future sessions are scheduled.

Introduction to Data Science with R. At $160 this O'Reilly training course is somewhat pricey considering how many free and lower-cost video classes there are out there. However, if you're looking for a step-by-step intro to R, this is a useful course, starting with language and ggplot2 visualization basics through modeling. It's taught by RStudio Master Instructor Garrett Grolemund, who focuses on hands-on learning as well as explaining a few of the language's quirks. If cost is an issue and you're not in a hurry, sign up for O'Reilly's deal emails and you may eventually find a 50% off sale.

Data Analysis and Visualization Using R. Free course that uses both video and interactive R to teach language basics, ggplot2 visualization basics, some statistical tests

and exploratory data analysis including data.table. Videos by Princeton Ph.D. student David Robinson and Neo Christopher Chung, Ph.D, filmed and edited at the Princeton Broadcast Center.

Other online introductions and tutorials

Try R This beginner-level interactive online course will probably seem somewhat basic for anyone who has experience in another programming language. However, even if the focus on pirates and plunder doesn't appeal to you, it may be a good way to get some practice and get more comfortable using R syntax.

An Introduction to R. Let's not forget the R Project site itself, which has numerous resources on the language including this intro. The style here is somewhat dry, but you'll know you're getting accurate, up-to-date information from the R Core Team.

How to Visualize and Compare Distributions. This short and highly readable Flowing Data tutorial goes over traditional visualizations such as histograms and box plots. With downloadable code.

Handling and Processing Strings in R. This PDF download covers many things you're want to do with text, from string lengths and formatting to search and replace with regular expressions to basic text analysis. By statistician Gaston Sanchez.

Learning statistics with R: A tutorial for psychology students and other beginners by Daniel Navarro at the University of Adelaide (PDF). 500+ pages that go from "Why do we learn statistics" and "Statistics in every day life" to linear regression and ANOVA (ANalysis Of VAriance). If you don't need/want a primer in statistics, there are still many sections that focus specifically on R.

R Tutorial. A reasonably robust beginning guide that includes sections on data types, probability and plots as well as sections focused on statistical topics such as linear regression, confidence intervals and p-values. By Kelly Black, associate professor at Clarkson University.

r4stats.com. This site is probably best known in the R community for author Bob Muenchen's tracking of R's popularity vs. other statistical software. However, in the Examples section, he's got some R tutorials such as basic graphics and graphics with ggplots. He's also posted code for tasks such as data import and extracting portions of your data comparing R with alternatives such as SAS and SPSS.

Aggregating and restructuring data. This excerpt from R in Action goes over one of the most important subjects in using R: reshaping your data so it's in the format needed for analysis and then grouping and summarizing that data by factors. In addition to touching on base-R functions like the useful-but-not-always-well-known aggregate(), it also covers melt() and cast() with the reshape package. By Robert I. Kabacoff.

Getting started with charts in R. From the popular FlowingData visualization website run by Nathan Yau, this tutorial offers examples of basic plotting in R. Includes downloadable source code. (While many FlowingData tutorials now require a paid membership to the site, as of May 2013 this one did not.)

Using R for your basic statistical Needs LISA Short Course. Aimed at those who already know stats but want to learn R, this is a *file of R code with comments*, making it easy to run (and alter) the code yourself.

The programming is easy to follow, but if you haven't brushed up on your stats lately, be advised that comments such as

Suppose we'd like to produce a reduced set of independent variables. We could use the function # step() to perform stepwise model selection based on AIC which is -2log(Likelihood) + kp? Where k=2 # and p = number of model parameters (beta coefficients).

may be tough to follow. By Nels Johnson at Virginia Tech's Laboratory for Interdisciplinary Statistical Analysis.

Producing Simple Graphs with R. Although 6+ years old now, this gives a few more details and examples for several of the visualization concepts touched on in our beginner's guide. By Frank McCown at Harding University.

Short courses. Materials from various courses taught by Hadley Wickham, chief scientist at RStudio and author of several popular R packages including ggplot2. Features slides and code for topics beyond beginning R, such as R development master class.

Quick introduction to ggplot2. Very nice, readable and — as promised — quick introduction to the ggplot2 add-on graphic package in R, incuding lots of sample plots and code. By Google engineer Edwin Chen.

ggplot2 workshop presentation. This robust, single-but-very-long-page tutorial offers a detailed yet readable introduction to the ggplot2 graphing package. What sets this apart is its attention to its theoretical underpinnings while also offering useful, concrete examples. From a presentation at the Advances in Visual Methods for Linguistics conference. By Josef Fruehwald, then a PhD candidate at the University of Pennsylvania.

ggplot2_tutorial.R. This online page at RPubs.com, prepared for the Santa Barbara R User Group, includes a lot of commented R code and graph examples for creating data visualizations with ggplot2.

More and Fancier Graphics. This one-page primer features loads of examples, including explainers of a couple of functions that let you interact with R plots, locator() and identify() as well as a lot of core-R plotting customization. By William B. King, Coastal Carolina University.

ggplot2 Guide. This ggplot2 explainer skips the simpler qplot option and goes straight to the more powerful but complicated ggplot command, starting with basics of a simple plot and going through geoms (type of plot), faceting (plotting by subsets), statistics and more. By data analyst George Bull at Sharp Statistics.

Using R. In addition to covering basics, there are useful sections on data manipulation — an important topic not easily covered for beginners — as well as getting statistical summaries and generating basic graphics with base R, the Lattice package and ggplot2. Short explainers are interspersed with demo code, making this useful as both a tutorial and reference site. By analytics consultant Alastair Sanderson, formerly research fellow in the Astrophysics & Space Research (ASR) Group at the University of Birmingham in the U.K.

The Undergraduate Guide to R. This is a highly readable, painless introduction to R that starts with installation and the command environment and goes through data types, input and output, writing your own functions and programming tips. Viewable as a Google Doc or downloadable as a PDF, plus accompanying files. By Trevor Martin, then at Princeton University, funded in part by an NIH grant.

How to turn CSV data into interactive visualizations with R and rCharts. 9–page slideshow gives step-by-step instructions on various options for generating interactive graphics. The charts and graphs use jQuery libraries as the underlying technology but only a couple of line of R code are needed. By Sharon Machlis, Computerworld.

Higher Order Functions in R. If you're at the point where you want to apply functions on multiple vectors and data frames, you may start bumping up against the limits of R's apply family. This post goes over 6 extremely useful base R functions with readable explanations and helpful examples. By John Mules White, "soon-to-be scientist at Facebook."

Introduction to Linear Regression Using R and Quandl While this does indeed promote Quandl as your data source, that data *is* free, and for those interested in using R for regressions, you'll find several detailed walk-throughs from data import through statistical analysis.

Introduction to dplyr. The dplyr package (by ggplot2 creator Hadley Wickham) significantly speeds up operations like grouping and sorting of data frames. It also aims to rationalize such functions by using a common syntax. In this short introductory vignette, you'll learn about "five basic data manipulation" — filter(), arrange(), select(), mutate() and summarise() — including examples, as well as how to chain them together for more streamlined, readable code. Another useful package for manipulating data in R: doBy.

Applied Time Series Analysis. Text-based online class from Penn State "to learn and apply statistical methods for the analysis of data that have been observed over time." Access to the articles is free, although there is no community or instructor participation.

13 resources for time series analysis. A video and 12 slide presentations by Rob J. Hyndman, author of Forecasting time series using R. Also has links to exercises and answers to the exercises.

knitr in a knutshell. knitR is designed to easily create reports and other documents that can combine text, R code and the results of R code — in short, a way to share your R analyses with others. This "minimal tutorial" by Karl Broman goes over subjects such as creating Markdown documents and adding graphics and tables, along with links to resources for more info.

More free downloads and websites from academia:

Introducing R. Slide presentation from the UCLA Institute for Digital Research and Education, with downloadable data and code.

Introducing R. Although titled for beginners and including sections on getting started and reading data, this also shows how to use R for various types of linear models. By German Rodriguez at Princeton University's Office of Population Research.

R: A self-learn tutorial. Intro PDF from National Center for Ecological Analysis and Synthesis at UC Santa Barbara. While a bit dry, it goes over a lot of fundamentals and includes exercises.

Statistics with R Computing and Graphics. Unlike many PDF downloads from academia, this one is both short (15 pages) and basic, with some suggested informal exercises as well as explanations on things like getting data into R and statistical modeling (understanding statistical concepts like linear modeling is assumed). By Kjell Konis, then at the University of Oxford.

Little Book of R for Time Series. This is extremely useful if you want to use R for analyzing data collected over time, and also has some introductory sections for general R use even if you're not doing time series. By Avril Coghlan at the Wellcome Trust Sanger Instituie, Cambridge, U.K.

Introduction to ggplot2. 11-page PDF with some ggplot basics, by N. Matloff at UC Davis.

Communities

Pretty much every social media platform has an R group. I'd particularly recommend:

Statistics and R on Google+. Community members are knowledgeable and helpful, and various conversation threads engage both newbies and experts.

Twitter #rstats hashtag. Level of discourse here ranges from beginner to extremely advanced, with a lot of useful R resources and commentary getting posted.

You can also find R groups on LinkedIn, Reddit and Facebook, among other platforms.

Stackoverflow has a very active R community where people ask and answer coding questions. If you've got a specific coding challenge, it's definitely worth searching here to see if someone else has already asked about something similar.

There are dozens of R User Meetups worldwide. In addition, there are other user groups not connected with Meetup.com. Revolution Analytics has an R User Group Directory.

Blogs & blog posts

R-bloggers. This site aggregates posts and tutorials from more than 250 R blogs. While both skill level and quality can vary, this is a great place to find interesting posts about R — especially if you look at the "top articles of the week" box on the home page.

Revolutions. There's plenty here of interest to all levels of R users. Although author Revolution Analytics is in the business of selling enterprise-class R platforms, the blog is not focused exclusively on their products.

Post: 10 R packages I wish I knew about earlier. Not sure all of these would be in *my* top 10, but unless you've spent a fair amount of time exploring packages, you'll likely find at least a couple of interesting and useful R add-ons.

Post: R programming for those coming from other languages. If you're an experienced programmer trying to learn R, you'll probably find some useful tips here.

Post: A brief introduction to 'apply' in R. If you want to learn how the apply() function family works, this is a good primer.

Translating between R and SQL. If you're more experienced (and comfortable) with SQL than R, it can be frustrating and confusing at times to figure out how to do basic data tasks such as subsetting your data. Statistics consultant Patrick Burns shows how to do common data slicing in both SQL and R, making it easier for experienced database users to add R to their toolkit.

Graphs & Charts in base R, ggplot2 and rCharts. There are *lots* of sample charts with code here, showing how to do similar visualization tasks with basic R, the ggplot2

add-on package and rCharts for interactive HTML visualizations.

When to use Excel, when to use R? For spreadsheet users starting to learn R, this is a useful question to consider. Michael Milton, author of *Head First Data Analysis* (which discusses both Excel and R), offers practical (and short) advice on when to use each.

A First Step Towards R From Spreadsheets. Some advice on both when and how to start moving from Excel to R, with a link to a follow-up post, From spreadsheet thinking to R thinking.

Using dates and times in R. This post from a presentation by Bonnie Dixon at the Davis R Users' group goes over some of the intricacies of dates and times in R, including various date/time classes as well as different options for performing date/time calculations and other manipulations.

Scraping Pro-Football Data and Interactive Charts using rCharts, ggplot2, and shiny. This is a highly useful example of beginning-to-end data analysis with R. You'll see a sample of how to scrape data off a website, clean and restructure the data and then visualize it in several ways, including interactive Web graphics — all with downloadable code. By Vivek Patil, an associate professor at Gonzaga University.

Search

Searching for "R" on a general search engine like Google can be somewhat frustrating, given how many utterly unrelated English words include the letter r. Some search possibilities:

RSeek is a Web search engine that just returns results from certain R-focused websites.

R site search returns results just from R functions, package "vignettes" (documentation that helps explain how a function works) and task views (focusing on a particular field such as social science or econometrics).

Misc

Google's R Style Guide. Want to write neat code with a consistent style? You'll probably want a style guide; and Google has helpfully posted their internal R style for all to use. If that one doesn't work for you, Hadley Wickham has a fairly abbreviated R style guide based on Google's but "with a few tweaks."

RStudio documentation. If you're using RStudio, it's worth taking a look at parts of the documentation at some point so you can take advantage of all it has to offer.

History of R Financial Time Series Plotting. Although, as the name implies, this focuses on financial time-series graphics, it's also a useful look at various options for plotting any data over time. With lots of code samples along with graphics. By Timely Portfolio on GitHub.

Grouping & Summarizing Data in R. There are *so* many ways to do these tasks in R that it can be a little overwhelming even for those beyond the beginner stage to decide which to use when. This downloadable Slideshare presentation by analyst Jeffrey Breen from the Greater Boston useR Group is a useful overview.

Apps

R Instructor. This app is primarily a well-designed, very thorough index to R, offering snippets on how to import, summarize and plot data, as well as an introductory section. An "I want to..." section gives

short how-to's on a variety of tasks such as changing data classes or column/row names, ordering or subsetting data and more. Similar information is available free online; the value-add is if you want the info organized in an attractive mobile app. Extras include instructional videos and a "statistical tests" section explaining when to use various tests as well as R code for each. For iOS and Android, about $5.

Software

Comprehensive R Archive Network (CRAN). The most important of all: home of the R Project for Statistical Computing, including downloading the basic R platform, FAQs and tutorials as well as thousands of add-on packages. Also features detailed documentation and a number of links to more resources.

RStudio. You can download the free RStudio IDE as well as RStudio's Shiny project aimed at turning R analyses into interactive Web applications.

Revolution Analytics. In addition to its commercial Revolution R Enterprise, you can request a download of their free Revolution R Community (you'll need to provide an email address). Both are designed to improve R performance and reliability.

Tibco. This software company recently released a free Tibco Enterprise Runtime for R Developers Edition to go along with its commercial Tibco Enterprise Runtime for R engine aimed at helping to integrate R analysis into other enterprise platforms.

Shiny for interactive Web apps. This open-source project from RStudio is aimed at creating interactive Web applications from R analysis and graphics. There's a Shiny tutorial at the RStudio site; to see more examples, Show Me Shiny offers a gallery of apps with links to code.

Swirl. This R package for interactive learning teaches basic statistics and R together. See more info on version 2.0.