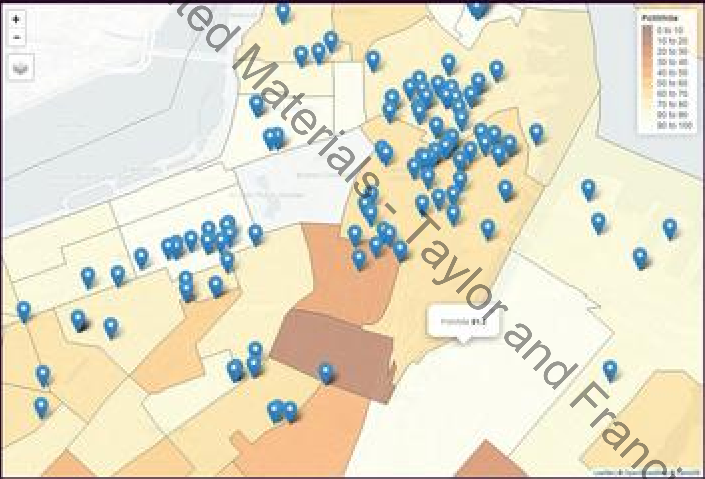
The R Series

Practical R for Mass Communication and Journalism



Sharon Machlis

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

Chapter 3

See How Much You Can Do in a Few Lines of Code

The sole purpose of this chapter is instant gratification. Fundamentals are important, but so is seeing a few of the cool things you can do in R *with very little code.* I do realize that adding 7 + 52 in a console, as I demonstrated last chapter, isn't all that compelling. I want to get you excited about R's potential! But if you're the type of person who gets frustrated doing things without fully understanding them, you may want to skip ahead to Chapter 4 and come back here later. This chapter is designed to show you some eye candy, not give detailed explanations.

3.1 Packages needed this chapter

(I'll include installation code when required)

- quantmod
- dygraphs
- htmlwidgets

3.2 What we'll cover

- Quick interactive graphs with the quantmod and dygraphs packages
- Interactive maps with a few lines of code.

You'll first need to install two packages, which you can do by running this code in your interactive R console (bottom left pane):

```
install.packages(c("quantmod", "dygraphs"))
```

quantmod is a library for financial analysis. dygraphs creates interactive Web graphics of data over time.

A note about installing packages: Usually, this is pretty seamless in R. Occasionally, you might get a message that the package you're trying to install *won't* install, because another package is missing. R *should* automatically install needed packages by default, but sometimes there's a glitch. If you get an error message that some other package you didn't know about is missing, try installing *that* one manually with `install.packages("missingPackageName")` and then run the install on the package you want.

In addition, sometimes you may be asked whether you want to "install from sources the package which needs compilation?" Most of the time, that just means, "Would you like the absolute latest version that doesn't have a handy, single download file yet?" For all packages we're using in this book, choosing n for no is easier and should work just fine.

Once installation is done, load both packages into your current R session with

```r
library("quantmod")
library("dygraphs")
```

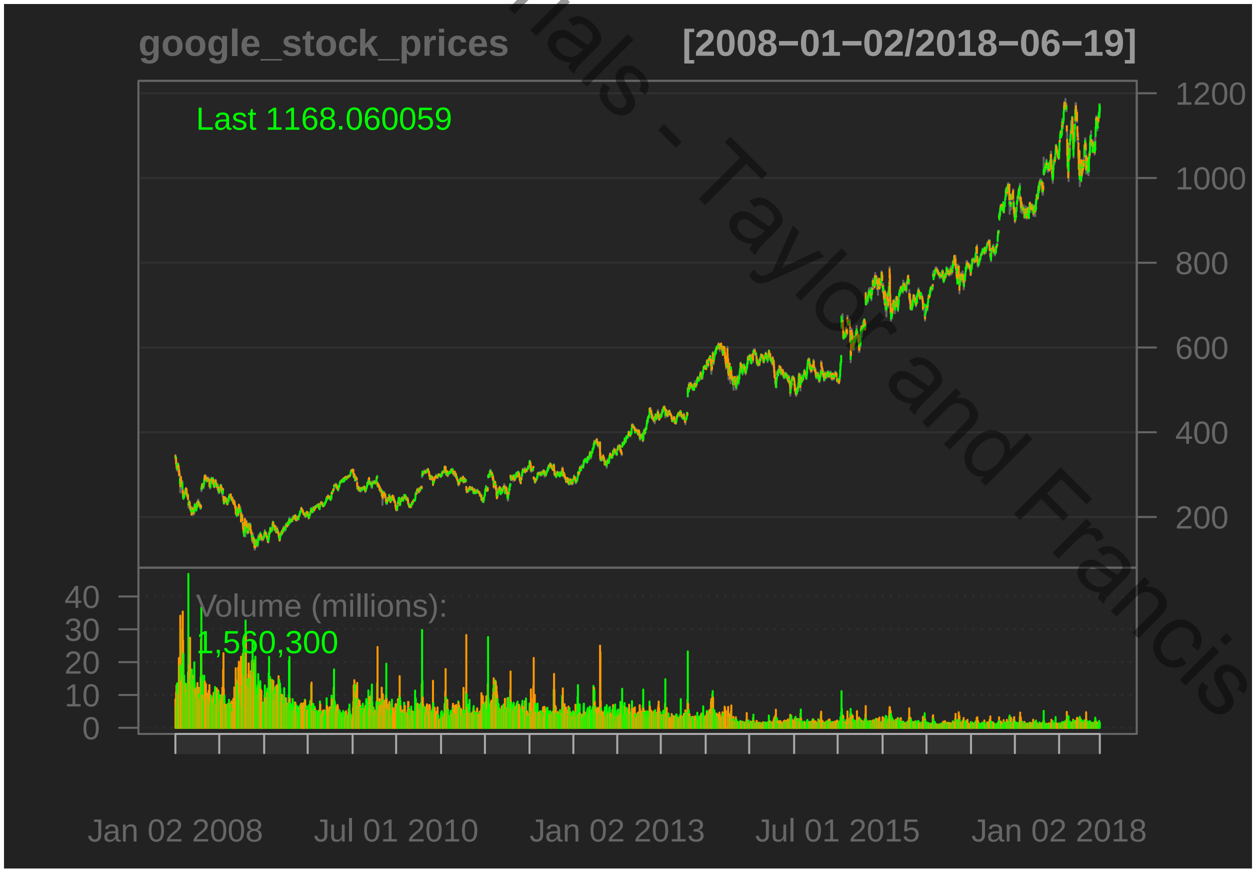
Finally, if you haven't yet set up an RStudio project for your code related to this book, as I described in the last chapter, create one for this work by going to File > New Project.

Now, let's give some those packages a try.

3.3 Simple stock market graphing

How did Google stock do since the 2008 stock market crash? Let's take a look ... with these two lines of code:

```r
google_stock_prices <- getSymbols("GOOG", src = "yahoo", from = "2008-01-01", auto.assign = FALSE)
chartSeries(google_stock_prices)
```



Setup notes: You can type each line of code into the console, but it will be more convenient in the long run to set up a script file for this chapter. Go to File > New File > R Script (or use the keyboard shortcut

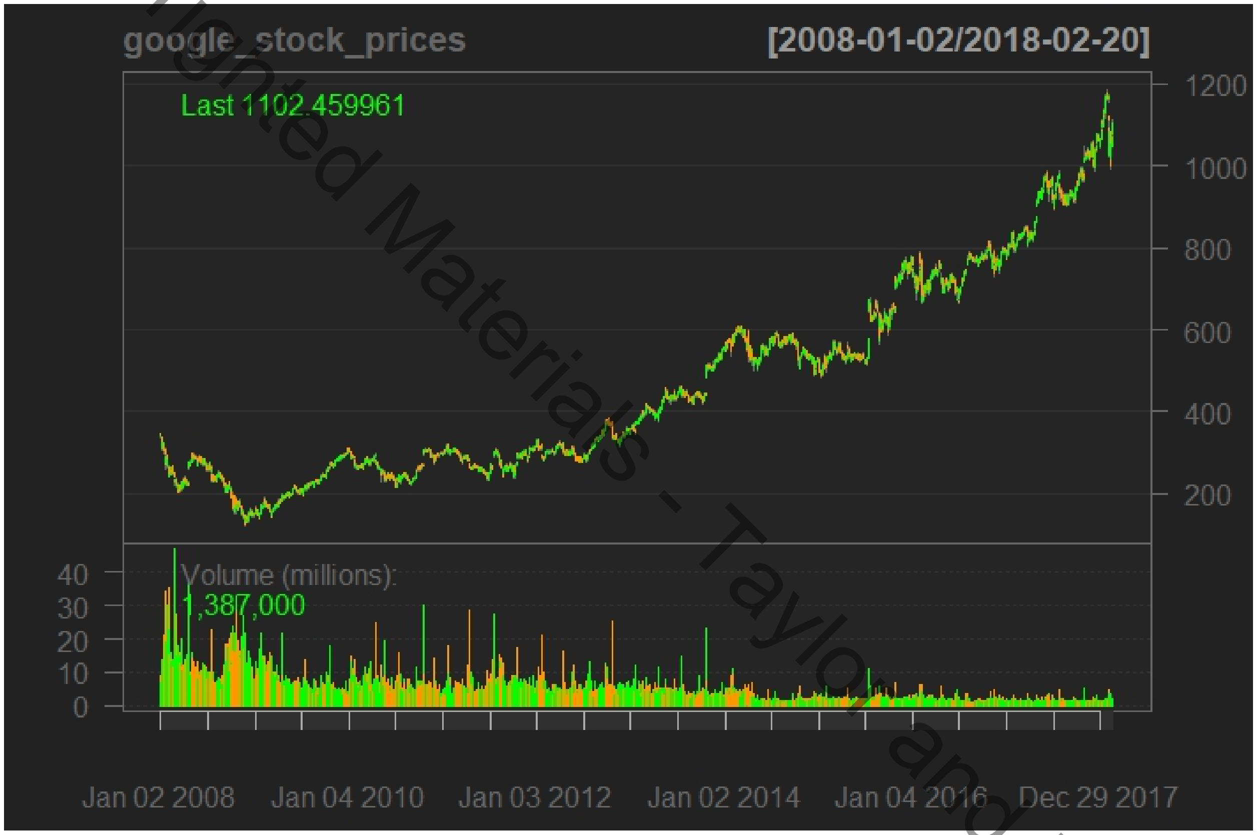

google_stock_prices
[2008-01-02/2018-02-20]
Last 1102.459961
1200
1000
800
600
400
200
Volume (millions):
1,387,000
40
30
20
10
0
Jan 02 2008
Jan 04 2010
Jan 03 2012
Jan 02 2014
Jan 04 2016
Dec 29 2017

Figure 3.1:

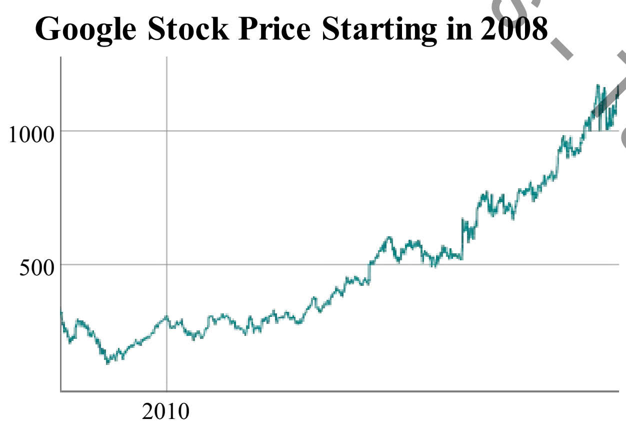
Ctrl-Shift-N on Windows / Cmnd-Shift-N on Mac). Type the commands into the script file, save it with the disk save icon, File > Save, or the ctrl/cmnd-S keyboard shortcut.

To run all *the code in the file, you can click the Source link above and to the right of the script panel (which is the top left panel). To run a few lines of code, select them the usual way (click and drag with your mouse) and hit control-Enter (command-Enter on Mac). To run one line of code, put your cursor anywhere on that line and hit control-Enter or command-Enter. When you do that, your cursor will jump to the next line of code and you can hit control/command-Enter again to quickly run the next line of code.*

getSymbols() is quantmod's powerful function for getting historical financial data from the Internet and importing it directly into R. `src = yahoo` tells quantmod that I want to pull the data from Yahoo; I could also have specified google to import from Google Finance. `from` tells quantmod when to start pulling the data. And auto.assign=FALSE deals with a quirk in older versions of quantmod, so it behaves like most R functions and can store functions in an R variable with the `<-` assignment operator.

The dygraphs package can make this graph interactive, so that you can mouse over the line and see underlying data, as well as click and drag to zoom in on a portion of the graph. Again, just one line of code, this one specifying I just want the GOOG.Close column (with closing prices) and a title of "Google Stock Price Starting in 2008"

```r
dygraph(google_stock_prices[,c("GOOG.Close")], main = "Google Stock Price Starting in 2008")
```



3.4 Download and graph a city's median income

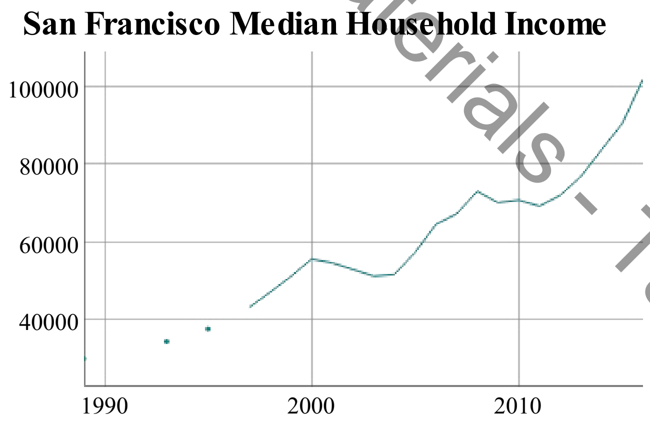
quantmod includes a function that lets you import data directly from the U.S. Federal Reserve – more specifically the St. Louis Federal Reserve's FRED database.

I went to FRED at https://fred.stlouisfed.org/ and searched for "median household income for San Francisco." This wasn't to get the data, but to find out which table contains the data. The URL for FRED's "Estimate of Median Household Income for San Francisco County/City, CA." was https://fred.stlouisfed.org/series/MHICA06075A052NCEN. The character/number string after fred.stlouisfed.org/series/ was what I needed, it's the St. Louis Fed's symbol for this data.

You may want to do the same search, so you can copy the MHICA06075A052NCEN portion of it into your clipboard from the FRED url instead of typing it manually.

Now try running the following code (I'll explain it in a bit) for some instant R gratification (gRatification?). Being able to paste MHICA06075A052NCEN from your clipboard should make this less onerous.

```r
sfincome <- getSymbols("MHICA06075A052NCEN", src="FRED", auto.assign = FALSE)
names(sfincome) <- "Income"
dygraph(sfincome, main = "San Francisco Median Household Income")
```



You should see a graph that looks like this in the Viewer tab of RStudio's lower right pane.

The zoom button above the graph on the left will open the pane into a larger window. The icon showing an arrow pointing to the upper right will open the graph in a browser window (in the image above, it's the icon directly over sco in San Francisco).

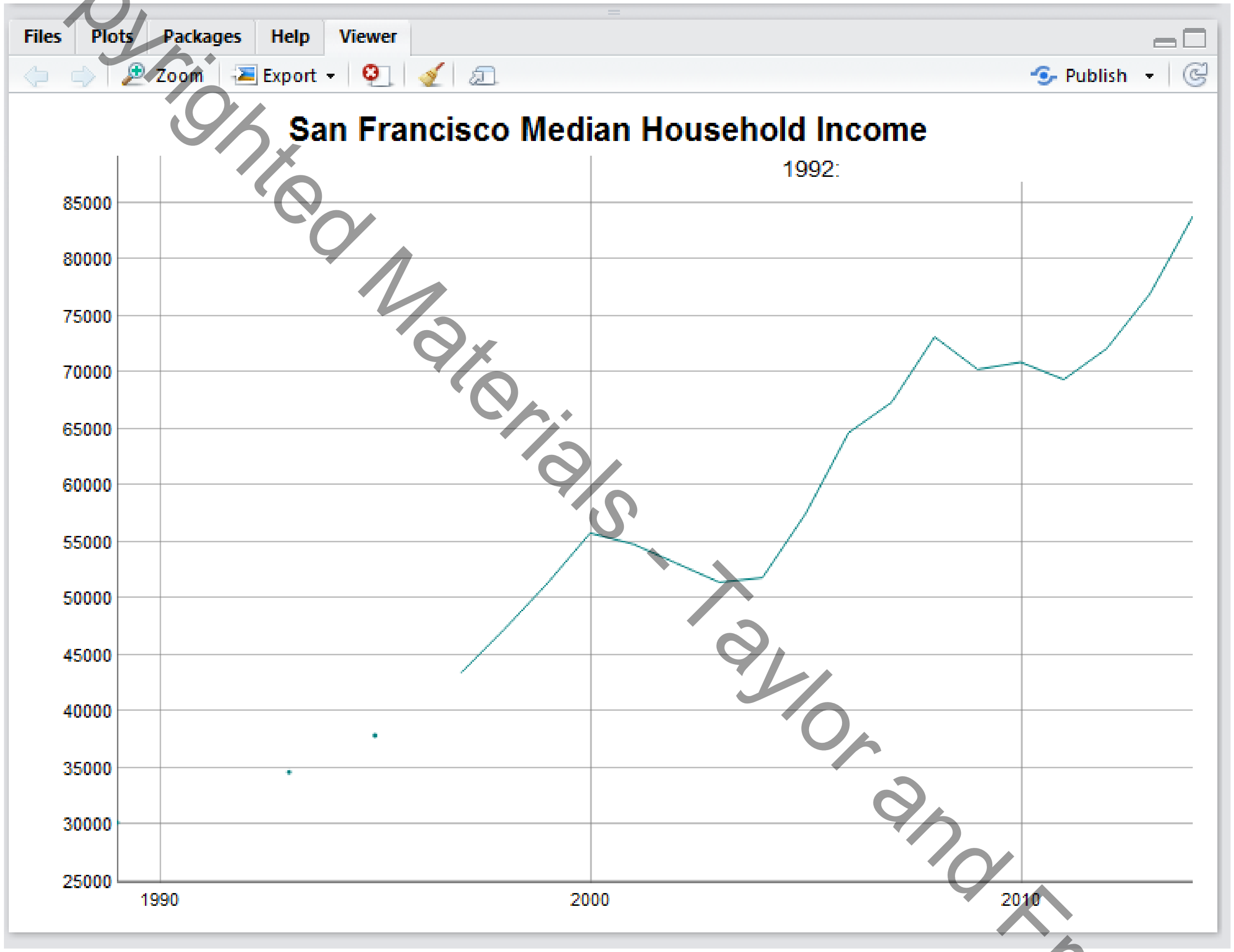

Files
Plots
Packages
Help
Viewer
Zoom
Export
Publish
San Francisco Median Household Income
1992:
85000
80000
75000
70000
65000
60000
55000
50000
45000
40000
35000
30000
25000
1990
2000
2010

Figure 3.2:

The graph is interactive. If you move your mouse from left to right anywhere on the graph, you'll see information about the closest underlying data point in the top right of the window. And if you click and drag from left to right inside the graph to define a portion of it - say from 2000 to 2010 - the graph will zoom in. (Double-clicking zooms out again, or you can always refresh the page.) This graph can be saved as an HTML "widget" or exported to a static JPG or PNG image using the RStudio Export menu item in this Viewer pane.

To save this graph as an HTML widget, first store it in an R variable, such as `sfmediangraph <- dygraph(MHICA06075A052NCEN, main = "San Francisco Median Household Income")`. Next, install the htmlwidget package with `install.packages("htmlwidgets")` and load it with `library("htmlwidgets")`. Finally, use the `saveWidget()` function to save your sfmediangraph object to a file, such as

```
saveWidget(sfmediangraph, file="sfmediangraph.html")
```

Find the sfmediangraph.html file on your computer and open it in a browser. You should see the same interactive HTML graph as you saw in RStudio, but now reusable on another Web site.

If by any chance you're thinking "This looks like a Web graphic I could make with JavaScript," you're right. The dygraphs package is an R wrapper for a JavaScript library, also called dygraphs. But with the R package, you can generate dygraphs JavaScript completely in R.

Now, the explanation of the code that I promised:

The first two lines load the quantmod and dygraphs packages into your current working session.

The third line uses quantmod's getSymbols() function to pull data from an external source. The function takes two "arguments" – options that the function needs to do its job. That first argument, "MHICA06075A052NCEN", is the symbol (the one we looked up at FRED) for retrieving data we want. The second argument, src="FRED", sets the data source to FRED, the St. Louis Federal Reserve's database. And you saw "autoassign = FALSE" in the previous section.

If you check the structure of this sfincome object with str(), you'll see that it's a special type of R object called `xts` (for extensible time series). Data starts in 1989, and those NA data listings you see stand for "Not Available," meaning that some points are missing (which you probably noticed on the graph).

```
str(sfincome)
```

```
## An 'xts' object on 1989-01-01/2016-01-01 containing:
##   Data: int [1:28, 1] 30166 NA NA NA 34623 NA 37854 NA 43405 47239 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : NULL
##   ..$ : chr "Income"
##   Indexed by objects of class: [Date] TZ: UTC
##   xts Attributes:
## List of 2
##  $ src    : chr "FRED"
##  $ updated: POSIXct[1:1], format: "2018-06-20 20:19:32"
```

The name of the lone data column with the median-income data is "MHICA06075A052NCEN". The data column title shows up on the graph, and I don't like that character string as my data label. So, in the fourth line, I change the name of that data column to "Income" with `names(sfincome) <- "Income"`. You can also run names() on an R object *without* the assignment operator, such as `names(sfincome)`, to see existing names.

(If you're wondering why no name for the *date* column shows up, that's how R time series are structured.)

The fifth line creates the graph, using the dygraph() function from the dygraphs package. The first argument, sfdata, tells dygraph what data set to graph. The second argument, main, is just the headline for the graph.

3.5 So many packages!

At this point, you might want to ask me: How did you know about these packages? How did you know what functions to use, and how they work? And, as you learn about different packages, how do you remember which ones to use when?

All good questions (and ones I'm often asked at workshops). I try to keep up with package developments by looking at top tweets with the #rstats Twitter hashtag, scanning a number of R blogs, and following posts on the Google Plus R group and Reddit R subreddit. This may be easier for me than a lot of other R users because 1) I cover developments in data analysis for my day job as a tech journalist, and 2) I'm somewhat R-obsessed. If you don't follow R developments for *your* job, a good shortcut to keep up with the latest R developments is the community-sourced R Weekly, which tries to round up the most interesting R news in a fairly easy-to-scan post at rweekly.org. You can also follow my tweets with the #rstats hashtag from my @sharon000 account.

When I learn about a very useful package, I add it to an interactive, searchable table published by Computerworld, which is available at http://bit.ly/Rpackages. You might want to keep your own spreadsheet of favorite packages and functions, whether or not it's published for others to read. I'd suggest keeping it somewhere in the cloud even if it's not public, like in a Google sheet or Excel spreadsheet on OneDrive, so you can access it from different computers.

There's another way to store some of your favorite functions right in RStudio, called code snippets. I'll be covering them in Chapter 6.

After you discover a package, reading the introductory vignette can help you figure out how to use it. Also, even if a package is on CRAN, the code may be on GitHub as well, and package authors will often add useful information there. In fact, there's an extremely helpful tutorial on the dygraph package at https://rstudio.github.io/dygraphs/. If you Google "R" and a package name you may come across other useful content (when I wrote this, the RStudio dygraphs tutorial on GitHub came up first when Googling R dygraphs).

3.6 Running functions without loading packages

Returning to the project at hand, I'd like to show you a slightly different way to generate the same graph:

```
sfincome <- quantmod::getSymbols("MHICA06075A052NCEN", src="FRED", auto.assign = FALSE)
names(sfincome) <- "Income"
dygraphs::dygraph(sfincome, main = "San Francisco Median Household Income")
```

In this code above, I use `quantmod::getSymbols()` instead of just `getSymbols()` and `dygraphs::dygraph()` instead of `dygraph()`. By adding `packagename::` before a function, I'm able to access a function from an external package _without having to load the package into memory first with library(packageName). This syntax `PackageName::FunctionName` works for any package that *exists on your computer's hard drive but isn't loaded into memory.* There can be a couple of advantages to this. First, if the package is large and you're only using one function once, you can save system memory by not loading all the package's functions into your session. (However, if you're using several of a package's functions multiple times throughout a script, it's often easier to just load it into memory.)

Second, when you look at a code snippet months later, it's clear which package each function belongs to.

Finally, if you've got multiple external packages loaded into memory, it's possible that the author of package1 named a function the same thing as the author of package2. Using package1::myfun() lets R know you want the myfun() function in package1, and not any other function named myfun() from some other package. We'll encounter that exact situation with two different functions called `describe()`.

3.7 Comparing one city's data to the US median

Whether you're a journalist or government staffer looking at a trend line like this, it's helpful to keep a key statistical question in mind: *compared to what?* A 6% increase in a city's median household income over several years might be impressive if overall national income stayed flat in the same period, but could be viewed as sluggish if the US household median rose 10% over that same period.

So let's add national median income to the graph. I looked up US income at FRED, and it's "MHIUS00000A052NCEN". Get the national data with

```
usincome <- getSymbols("MHIUS00000A052NCEN", src="FRED")
names(usincome) <- "US Income"
```

and then add the national data columns to the San Francisco data set with base R's `cbind()` function. cbind means "bind" two data sets together by adding one data column to another, side by side. You can also `rbind()` to add rows from one data set below another.

```
mygraphdata <- cbind(sfincome, usincome)
```

```
## Warning in merge.xts(..., all = all, fill = fill, suffixes = suffixes): NAs
## introduced by coercion
```

Now, use the `dygraph()` function to graph the mygraphdata data set the same way you created the first graph:

```
dygraph(mygraphdata, main = "Median Household Income")
```

One of the best things about scripting this: When new data is available from the Fed, you can just run the same code and you'll have an updated graph! Now, imagine how useful this is if you work with data that updates monthly or weekly.

Another advantage: Once you've got the basic code for pulling data from the Fed, it's easy to change to another data point such as unemployment. The code for US unemployment data on FRED is "UNRATE" and San Francisco is "SANF806UR", so you can just swap those in for "MHICA06075A052NCEN" and "MHIUS00000A052NCEN" and get your data set:

```
sfunemp <- getSymbols("SANF806UR", src="FRED", auto.assign = FALSE)
names(sfunemp) <- "SFRate"
usunemp <- getSymbols("UNRATE", src="FRED", auto.assign = FALSE)
names(usunemp) <- "USRate"
unemploymentdata <- cbind(sfunemp, usunemp)
dygraph(unemploymentdata, main = "Monthly Unemployment Rates, US and San Francisco")
```

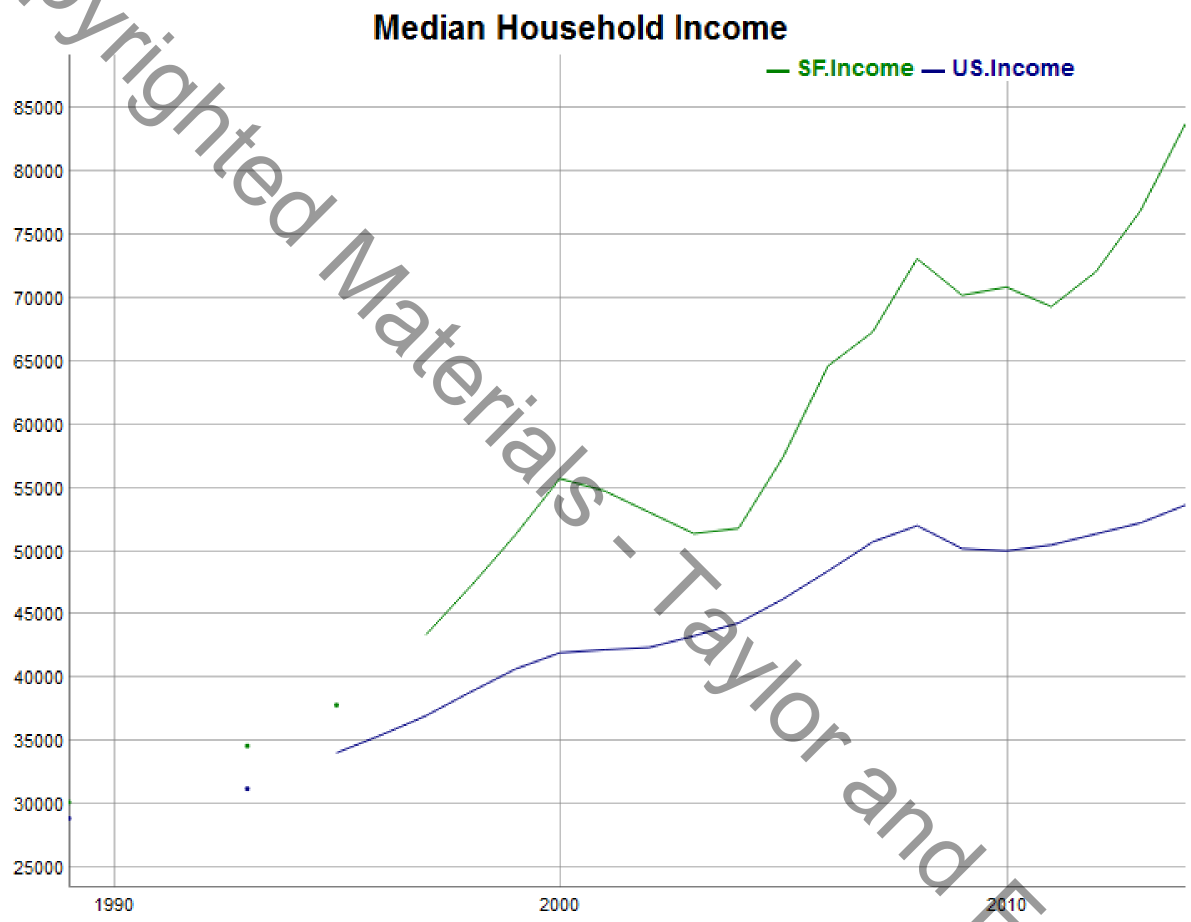
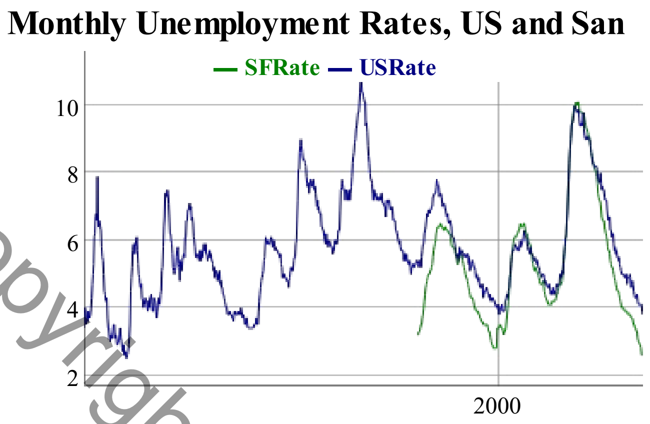


Figure 3.3: Graph of San Francisco and US median household income



Because San Francisco unemployment data is only available since 1990, it could be better to just show all data from 1990 onward instead of displaying earlier national data. Time series have their own unusual way of subsetting data; this code

```
unemploymentdata <- unemploymentdata["1990/"]
```

will update the unemploymentdata variable so it contains only information from 1990 and later.

Now you can draw the graph with:

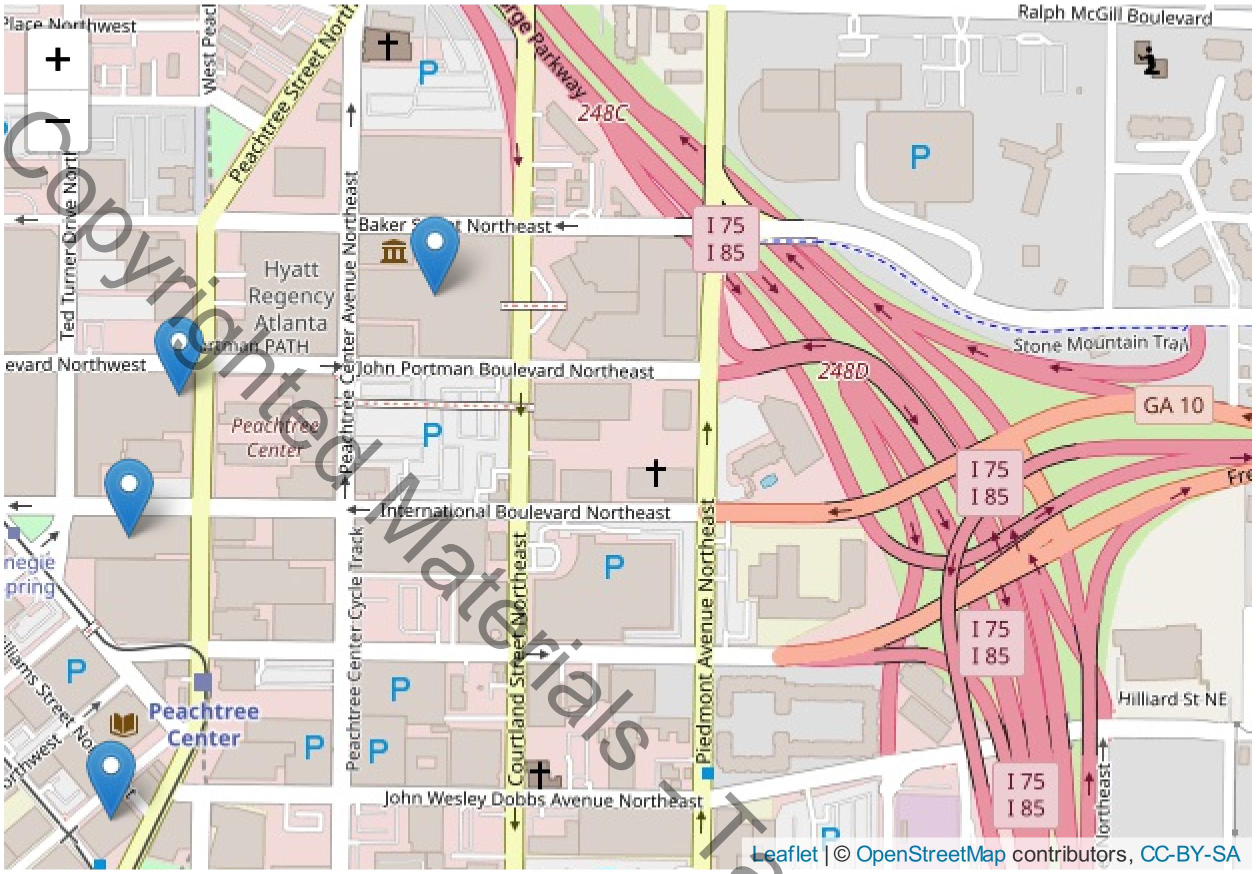
```
dygraph(unemploymentdata, main = "Monthly Unemployment Rates, US and San Francisco")
```

3.8 Run a remote script to make an interactive map

I've posted most of the data files from this book in a public repository on GitHub. Soon we'll create a new RStudio project for all these files. For now, though, I'd like you to run one file from that "repo" at https://raw.githubusercontent.com/smach/R4JournalismBook/master/03_map.R . If that's somewhat onerous to type, go to the bit.ly link http://bit.ly/R4JournalismLinks and find and copy the URL under the *Chapter 3* heading.

Then run

```
source("https://raw.githubusercontent.com/smach/R4JournalismBook/master/03_map.R")
make_mymap()
```


Baker Street Northeast
Hyatt Regency Atlanta
John Portman Boulevard Northeast
International Boulevard Northeast
Peachtree Center
Peachtree Street Northeast
Courtland Street Northeast
Piedmont Avenue Northeast
John Wesley Dobbs Avenue Northeast
Ralph McGill Boulevard
Stone Mountain Trail
GA 10
I 75 I 85
Hilliard St NE
Leaflet | © OpenStreetMap contributors, CC-BY-SA

and see what happens. Eventually, you should see an interactive map in your bottom-right RStudio Viewer pane, showing a few Starbucks in Atlanta. This is a national map, though – you can zoom out and see other areas of the U.S.

Before you use this to find a coffee shop near you, though, be warned that the latest available data file for Starbucks locations is from 2012. The point is to demonstrate making a cool interactive map with pop-ups, and I've found that the Starbucks data set tends to resonate in a crowd of journalists.

I also wanted to show you how to include code from an external file in your scripts. `source("myfile")` lets you run code from an external file in your current script, whether that file is somewhere else on your own system or a file that you've stored remotely.

3.9 Bonus map: Mapping income data

We'll be doing a lot more work with maps later. Meanwhile, though, if you'd like a sneak peek, try running the following remote code. Note that it will be loading several packages on your system the first time it executes as well as a data file, so it may take a little while to execute.

You can find and copy that lengthy URL by heading to http://bit.ly/R4JournalismLinks and copying the second URL under the Chapter 3 heading.

```
source("https://raw.githubusercontent.com/smach/R4JournalismBook/master/03_manhattan_income_map.R")
get_household_income_in_county()
```

Click on different portions of the map to get pop-ups with data details.

3.10 Wrap-Up

I hope that's gotten you enthused about some of the power of R. This chapter covered a lot of code somewhat quickly, but the point was to see that R can do some cool things, not to learn and understand every nuance of programming presented here.

Here's what's worth remembering from this chapter:

Load an entire R package into your working session's memory with `library(mypackage)`.

You can use a function from an external package *without* loading the entire package into memory, using the syntax `mypackage::myfunction()`.

Run R code from an external file with `source("myfile.R")`.

Combine data frames by column with `cbind()` (putting them together side by side) and by row with `rbind()` (putting them together one below the other). We'll cover more sophisticated merging by common columns in later chapters.

`names(myobject)` will display an object's existing names. You can also *change* the names of one or more items in an R object with names(myobject), such as names(mydataframe) <- c("Column1", "Column2").

NA stands for Not Available and indicates missing data in R.

There are some very elegant ways of importing and visualizing data with R.

Next up: How to import all sorts of data into R.

3.11 Additional resources

- **5 data visualizations in 5 minutes:** Each in 5 lines or less of R http://bit.ly/5LinesOrLess. This is a version of my 5-minute lightning talk at the 2015 National Institute for Computer Assisted Reporting conference. With video.
- **htmlwidgets for R** htmlwidgets.org. Find out more about interactive HTML for R including dozens of packages and a showcase of examples.