Lecture 7 Tutorial: Maps 1 of 3

Leah Brooks

March 8, 2018

A. Agenda for Today

Our goals today are two-fold. We will gain some programming experience

- loops
- merging
- string functions

We will also learn a lot about making maps in R. We will do with by using `plot()` from Base R, and also a very new variant of `ggplot` which allows for the simple plotting of spatial objects, based on a package called `sf`.

B. Packages

This class requires many new packages. Right now, you should install

- rgdal
- raster
- sp
- sf
- devtools
- digest
- scales
- tibble

But that's not it. We are using the new `geom_sf()` command from ggplot. It's so new, it's not yet available in standard packages from CRAN (Comprehensive R Action Network). Instead, we have to load it from github, which is website for all kinds of programming stuff. To do this, you'll need to do run the code below. R may ask you about installing a .exe file or files, and you should say yes. R may also ask you about restarting and you should say yes.

```
library(devtools)
devtools::install_github("tidyverse/ggplot2")
```

Finally, you also need to install the `devtools` version of the `sf` package. To do this, try

```
devtools::install_github("r-spatial/sf")
```

If you've successfully installed all those packages, you're ready to go.

C. Load a first shapefile

We are going to load the shapefile that corresponds with the county data that we've been using. We read shapefiles differently than we read dataframes. Below I use the `writeOGR` command to load a shapefile (later in this tutorial, we'll explore an alternate method).

You can download this shapefile from here. You will need to unzip after downloading. A complete shapefile has 4 to 6 separate files – and you need them all. This file has 5 parts with extensions as listed below

- .dbf : a spreadsheet file with information about the polygons
- .prj : the coordinate reference system for the polygons
- .shp : the actual polygon information in coordinate terms
- .shx : not sure what this adds
- .xml : not sure what this adds

For all the libraries below, you may need to install the associated package for it to work. For example, if you don't find the package `rgdal`, you should do `install.packages("rgdal")`.

```
library(rgdal)
```

```
## Loading required package: sp
```

```
## rgdal: version: 1.2-16, (SVN revision 701)
##  Geospatial Data Abstraction Library extensions to R successfully loaded
##  Loaded GDAL runtime: GDAL 2.2.0, released 2017/04/28
##  Path to GDAL shared files: C:/Users/lbrooks/Documents/R/win-library/3.4/rgdal/gdal
##  GDAL binary built with GEOS: TRUE
##  Loaded PROJ.4 runtime: Rel. 4.9.3, 15 August 2016, [PJ_VERSION: 493]
##  Path to PROJ.4 shared files: C:/Users/lbrooks/Documents/R/win-library/3.4/rgdal/proj
##  Linking to sp version: 1.2-7
```

```
library(raster)
cnty.map <- readOGR(dsn = "h:/pppa_data_viz/2018/tutorials/lecture07/data",
                    layer ="gz_2010_us_050_00_500k")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: "h:/pppa_data_viz/2018/tutorials/lecture07/data", layer: "gz_2010_us_050_00_500k"
## with 3221 features
## It has 6 fields
```

We have just created a Spatial Polygon Data Frame. This is an object of the "S4" class in R. To date we have worked with objects of the "S3" class. I tell you this not because you need to know the details of these separate classes, but to say that all the tools we've learned to date work on S3 objects. They may or may not work on S4 objects.

R has many tools to help you figure out what's in this spatial polygon dataframe.

```
# what kind of file is it?
class(cnty.map)
```

```
## [1] "SpatialPolygonsDataFrame"
## attr(,"package")
## [1] "sp"
```

```
# how many features does this file have?
# feature: point, polygon, line
length(cnty.map)
```

```
## [1] 3221
```

```
# how far does this thing go out?
extent(cnty.map)
```

```
## class       : Extent
## xmin        : -179.1473
## xmax        : 179.7785
```

```
## ymin        : 17.88481
## ymax        : 71.35256
```

```
# metadata summary
cnty.map
```

```
## class       : SpatialPolygonsDataFrame
## features    : 3221
## extent      : -179.1473, 179.7785, 17.88481, 71.35256  (xmin, xmax, ymin, ymax)
## coord. ref. : +proj=longlat +datum=NAD83 +no_defs +ellps=GRS80 +towgs84=0,0,0
## variables   : 6
## names       :         GEO_ID, STATE, COUNTY,      NAME,    LSAD, CENSUSAREA
## min values  : 0500000US01001,    01,    001, Abbeville, Borough,      1.999
## max values  : 0500000US72153,    72,    840,   Ziebach,  Parish, 145504.789
```

```
# look at the attributes
head(cnty.map)
```

```
##           GEO_ID STATE COUNTY      NAME   LSAD CENSUSAREA
## 0 0500000US01029    01    029  Cleburne County    560.100
## 1 0500000US01031    01    031    Coffee County    678.972
## 2 0500000US01037    01    037     Coosa County    650.926
## 3 0500000US01039    01    039 Covington County   1030.456
## 4 0500000US01041    01    041  Crenshaw County    608.840
## 5 0500000US01045    01    045      Dale County    561.150
```

We also care about how the file is projected, which is a way of saying we may be interested in how we're telling R to lay our shapes out across space. You can see the "coordinate reference system" directly using the `crs()` command below. You can also make the coordinate reference system an object itself, in case you want to give this CRS to another map.

```
# how is this map projected?
crs(cnty.map)
```

```
## CRS arguments:
##  +proj=longlat +datum=NAD83 +no_defs +ellps=GRS80 +towgs84=0,0,0
```

```
# note that you can store this in something
cproj <- crs(cnty.map)
cproj
```

```
## CRS arguments:
##  +proj=longlat +datum=NAD83 +no_defs +ellps=GRS80 +towgs84=0,0,0
```

Spatial polygon data frames (or their parallel in lines or points) have a "dataframe" component – the equivalent of the .dbf file that is part of a shapefile. You can use specific language when you want to get to the data, which is called a data "slot." In particular, you get to the data by writing `dataframe@data$varname`, where `dataframe` is the name of the data frame, `varname` is the name of the variable, and `@data` is always the same, letting R know that you're getting to the attributes part of this file.

As you see below, you can do regular dataframe things like `head()` and `table()` with this slot. You can also subset the spatial polygon dataframe like you would a regular dataframe, using standard subset language. Below I subset the map to be only the continental US, dropping Alaska ("02"), Hawaii ("15") and Puerto Rico ("72"). See lecture 6's tutorial for more explanation on this command.

```
# shapefiles have a -data- slot
# @data is common to all files
head(cnty.map@data)
```

```
##           GEO_ID STATE COUNTY      NAME   LSAD CENSUSAREA
## 0 0500000US01029    01    029  Cleburne County    560.100
## 1 0500000US01031    01    031    Coffee County    678.972
## 2 0500000US01037    01    037     Coosa County    650.926
## 3 0500000US01039    01    039 Covington County   1030.456
## 4 0500000US01041    01    041  Crenshaw County    608.840
## 5 0500000US01045    01    045      Dale County    561.150
```

```
# you can do regular R things with the @data part
table(cnty.map@data$LSAD)
```

```
##
## Borough      CA    city  County Cty&Bor    Muno    Muny  Parish
##      12      11      41    3007       4      78       2      64
```

```
table(cnty.map@data$STATE)
```

```
##
##  01  02  04  05  06  08  09  10  11  12  13  15  16  17  18  19  20  21
##  67  29  15  75  58  64   8   3   1  67 159   5  44 102  92  99 105 120
##  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39
##  64  16  24  14  83  87  82 115  56  93  17  10  21  33  62 100  53  88
##  40  41  42  44  45  46  47  48  49  50  51  53  54  55  56  72
##  77  36  67   5  46  66  95 254  29  14 134  39  55  72  23  78
```

```
# lets get rid of alaska and hi! too bothersome for plotting
ccnties <- cnty.map[ !(cnty.map@data$STATE %in% c("02","15","72")),]
length(ccnties)
```

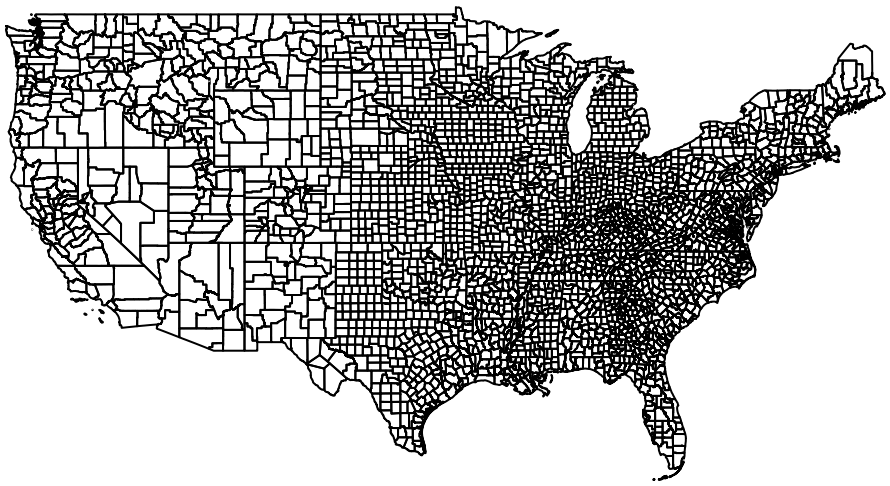
```
## [1] 3109
```

D. Make a map

It's surprisingly easy to make a no-fills map in R. We'll start by using R's very basic `plot()` command. We'll make the whole US, and then the new continental US file we created.

```
# make the whole US
plot(cnty.map)
```

```
# make just the continental US
plot(ccnties)
```



Of course, there are about a 1000 different options for improving the look of these plots. In the interest of time, I am going to leave them to your exploration.

E. Color by Attribute

You'll very frequently see maps with attributes colored in. This kind of map is called a choropleth. We'll now do a very simple version of such a map, coloring in the census divisions that we have used before.

We begin by assigning a division to each state. We did this before in tutorial 4, so you can copy and modify your code from that tutorial if you'd like. I use the table command at the end to make sure that every county is assigned (as it should be) to a division.

```
# assign a division to each state
# we did this before in tutorial 4 -- just go copy your old code
# i used search and replace to change the dataframe name
# and watch out -- numbers w/ leading zeros in map, so your numbers need to comply
ccnties@data$division <-
  ifelse(ccnties@data$STATE == "09" | ccnties@data$STATE == 23 |
         ccnties@data$STATE == 25 | ccnties@data$STATE == 33 |
         ccnties@data$STATE == 44 | ccnties@data$STATE == 50, 1,
    ifelse(ccnties@data$STATE == 34 | ccnties@data$STATE == 36 |
           ccnties@data$STATE == 42, 2,
```

```
      ifelse(ccnties@data$STATE == 18 | ccnties@data$STATE == 17 |
             ccnties@data$STATE == 26 | ccnties@data$STATE == 39 |
             ccnties@data$STATE == 55, 3,
        ifelse(ccnties@data$STATE == 19 | ccnties@data$STATE == 20 |
               ccnties@data$STATE == 27 | ccnties@data$STATE == 29 |
               ccnties@data$STATE == 31 | ccnties@data$STATE == 38 |
               ccnties@data$STATE == 46, 4,
          ifelse(ccnties@data$STATE == 10 | ccnties@data$STATE == 11 |
                 ccnties@data$STATE == 12 | ccnties@data$STATE == 13 |
                 ccnties@data$STATE == 24 | ccnties@data$STATE == 37 |
                 ccnties@data$STATE == 45 | ccnties@data$STATE == 51 |
                 ccnties@data$STATE == 54, 5,
            ifelse(ccnties@data$STATE == "01" | ccnties@data$STATE == 21 |
                   ccnties@data$STATE == 28 | ccnties@data$STATE == 47,6,
              ifelse(ccnties@data$STATE == "05" | ccnties@data$STATE == 22 |
                     ccnties@data$STATE == 40 | ccnties@data$STATE == 48, 7,
                ifelse(ccnties@data$STATE == "04" | ccnties@data$STATE == "08" |
                       ccnties@data$STATE == 16 | ccnties@data$STATE == 35 |
                       ccnties@data$STATE == 30 | ccnties@data$STATE == 49 |
                       ccnties@data$STATE == 32 | ccnties@data$STATE == 56, 8,
                  ifelse(ccnties@data$STATE == "02" | ccnties@data$STATE == "06" |
                         ccnties@data$STATE == 15 | ccnties@data$STATE == 41 |
                         ccnties@data$STATE == 53,9,0)))))))))
table(ccnties@data$division)
```

```
##
##   1   2   3   4   5   6   7   8   9
##  67 150 437 618 589 364 470 281 133
```

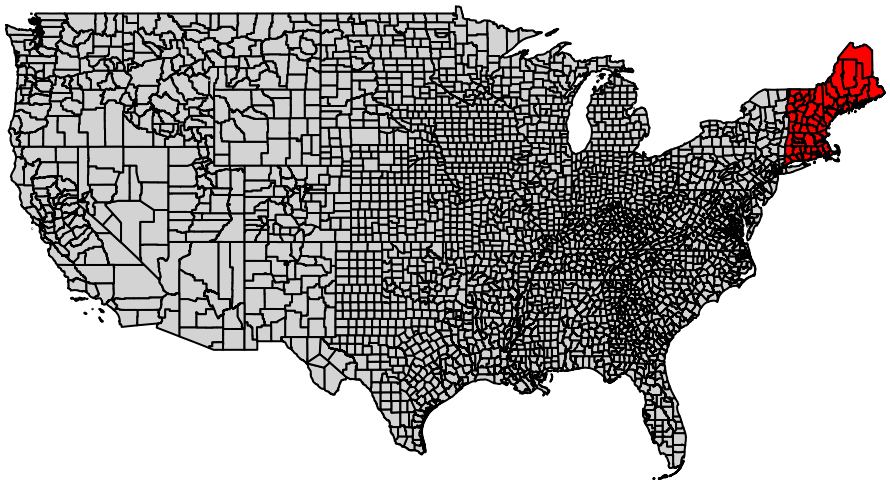
We should see that there are no counties with a division code of 0, the final residual category of the `ifelse()` statements.

Rosa points out that you can do this same thing with an easier-to-read command as below. These two commands are equivalent.

```
# assign a division to each state
ccnties@data$division <- ifelse(ccnties@data$STATE %in% c("09", "23", "25", "33", "44",
                                                          "50"), 1,
                         ifelse(ccnties@data$STATE %in% c("34", "36", "42"), 2,
                         ifelse(ccnties@data$STATE %in% c("18", "17", "26", "39", "55"), 3,
                         ifelse(ccnties@data$STATE %in% c("19", "20", "27", "29", "31",
                                                          "38", "46"), 4,
                         ifelse(ccnties@data$STATE %in% c("10", "11", "12", "13", "24",
                                                          "37", "45", "51", "54"), 5,
                         ifelse(ccnties@data$STATE %in% c("01", "21", "28", "47"), 6,
                         ifelse(ccnties@data$STATE %in% c("05", "22", "40", "48"), 7,
                         ifelse(ccnties@data$STATE %in% c("04", "08", "16", "35", "30",
                                                          "49", "32", "56"), 8,
                         ifelse(ccnties@data$STATE %in% c("02", "06", "15", "41", "53")
                                , 9, 0)))))))))
```

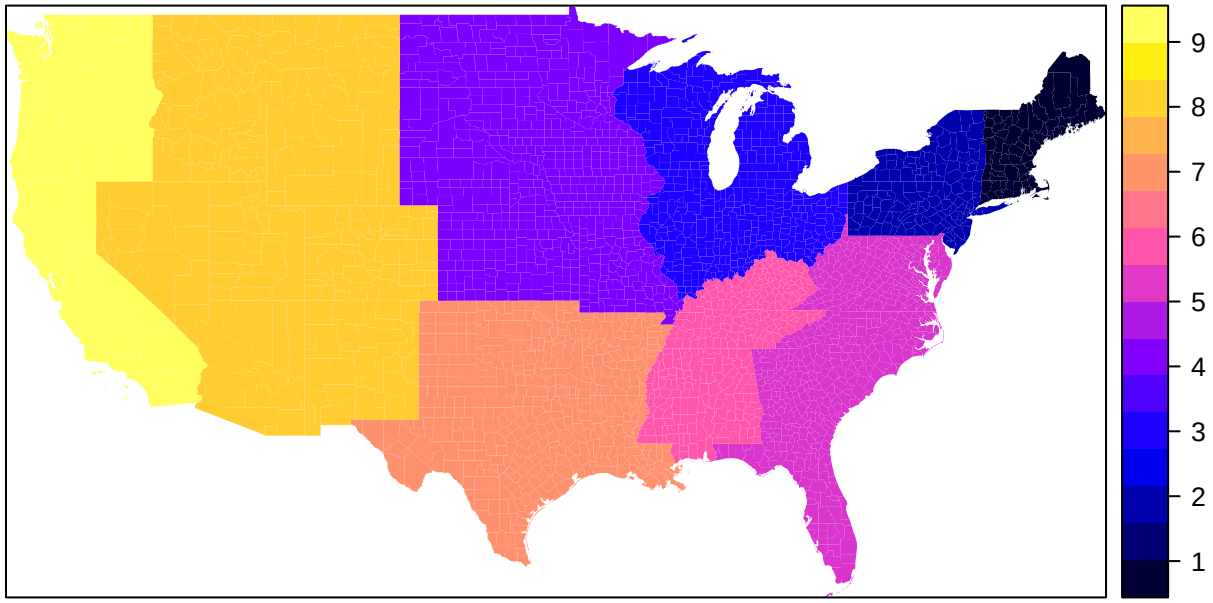
First, we'll show you the power of stacking plots in R. We begin by plotting all counties with a light grey background (`col="lightgrey"`). On top of that, we add (`add = TRUE`) another plot, which is a subset of our map to division one (`ccnties[ccnties@data$division %in% c("1"),]`), colored in red.

```
# now color in one division
plot(ccnties, col="lightgrey")
plot(ccnties[ccnties@data$division %in% c("1"),],
     col = "red",
     add = TRUE)
```



This is great if you want to highlight one division. If you want to show where all the divisions are, you need something slightly more complicated. We'll use the `spplot()` command, which is designed for making maps by attributes.

```
spplot(ccnties, "division", col = "transparent")
```



You might also need to make a bunch of maps – perhaps one by division. To do this, you can type the same code nine times – or you can use a loop.

I've gone over loops in class, but the basic idea is that you have sometime you want to repeat by an index value. Below we plot each division separately with a title. First I create a vector called `divisions` which is sequence of numbers 1 to 9 (by 1). Then, for each of those numbers – indexed by `d`, the loop does the commands below. Note that the loop begins with `{` and ends with `}`. The loop

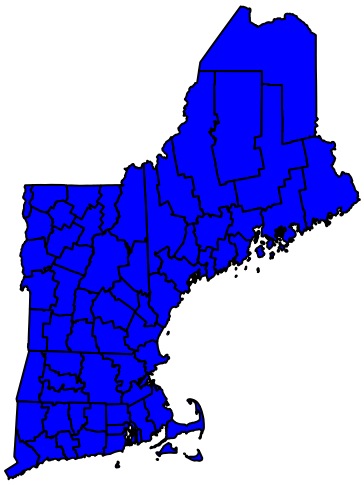
- creates a character string that says "this is division [whatever the division number is]"
  - this line uses the `paste` command, which squishes strings of characters together, separated (or not) by whatever is after the `sep=` option. Here we separate by nothing.
- if you'd like to print this character string to the screen, add `print(tito)`
  - this isn't necessary – but it can be helpful for de-bugging problems
  - I don't do it here for space reasons
- makes a plot of just the counties in this specific division, colored in blue, and with the title of the text string we just created

```
# do a loop and plot each division
divisions <- seq(1,9,1)
divisions
```

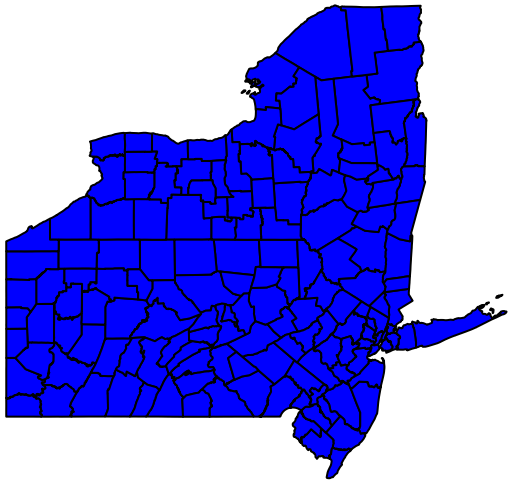
```
## [1] 1 2 3 4 5 6 7 8 9
```

```
for(d in divisions){
  tito <- paste("this is division ",d,sep="")
  plot(ccnties[ccnties@data$division == c(d),],
```

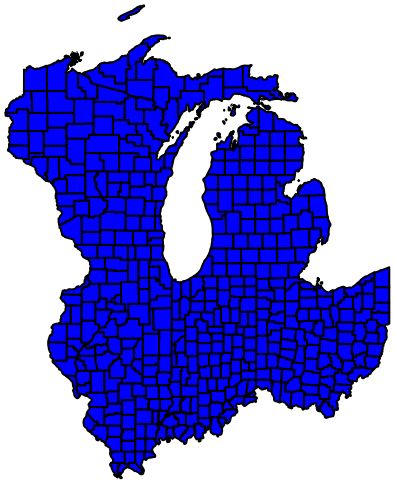
```
        col = "blue",
        main = tito)
}
```



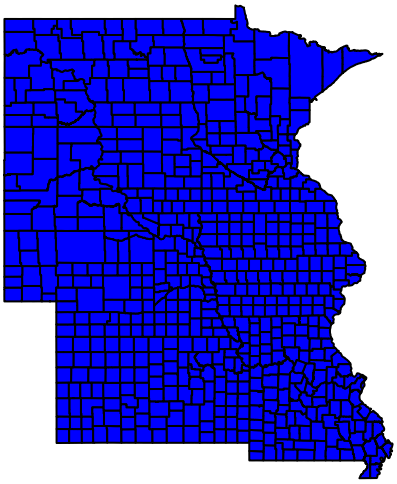
this is division 2



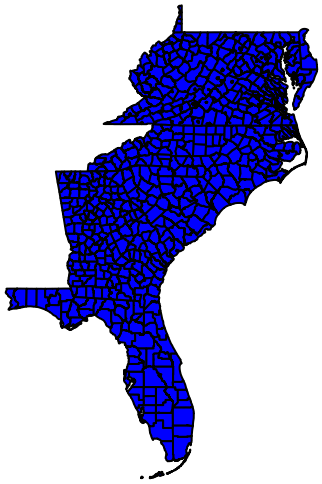
this is division 3



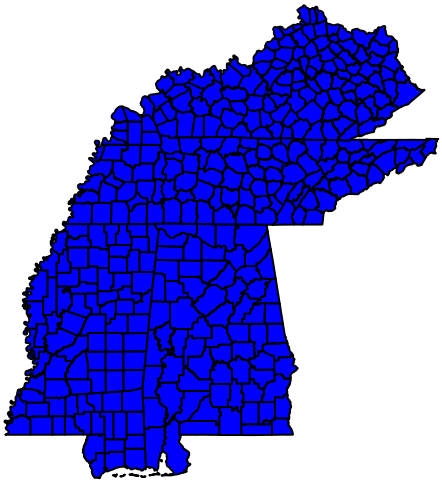
this is division 4



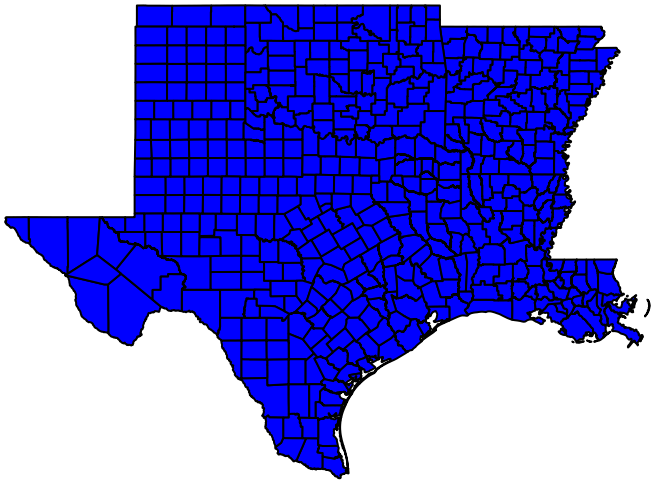
this is division 5



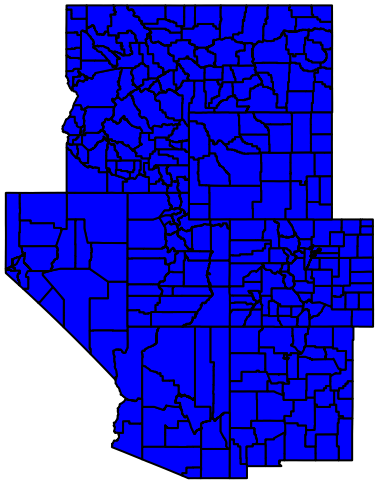
this is division 6

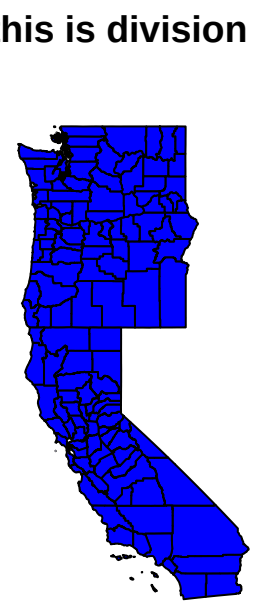


this is division 7



this is division 8





You can use this technique for anything you want to replicate by a list.

F. Preparing attribute data

Of course, what we can show with just this map file is pretty limited. There are no demographic or environmental variables. However, we do have a dataset with county information. So the next step in this tutorial is to link the county dataset with the map. We'll begin by preparing the county dataframe.

The commands below do things we've already practiced. They load the data, keep just 2010, limit to a few variables so that things are easier, get rid of Alaska, Hawaii and Puerto Rico to make the data match the map, and then look at the data that are left.

```
# load the county data
counties <- read.csv("h:/pppa_data_viz/2018/tutorials/lecture01/counties_1910to2010_20180115.csv")

# just keep 2010
counties.2010 <- counties[which(counties$year == 2010),]
dim(counties.2010)
```

```
## [1] 3143   68
```

```
# just keep a few variables to make this a little easier
dim(counties.2010)
```

```
## [1] 3143   68
```

```
counties.2010 <- counties.2010[,c("statefips","countyfips","cv1")]
dim(counties.2010)
```

```
## [1] 3143    3
```

```
# get rid of alaska, hawaii and puerto rico for easier matching later
dim(counties.2010)
```

```
## [1] 3143    3
```

```
table(counties.2010$statefips)
```

```
##
##   1   2   4   5   6   8   9  10  11  12  13  15  16  17  18  19  20  21
##  67  29  15  75  58  64   8   3   1  67 159   5  44 102  92  99 105 120
##  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39
##  64  16  24  14  83  87  82 115  56  93  17  10  21  33  62 100  53  88
##  40  41  42  44  45  46  47  48  49  50  51  53  54  55  56
##  77  36  67   5  46  66  95 254  29  14 134  39  55  72  23
```

```
counties.2010 <- counties.2010[!(counties.2010$statefips %in% c("2","15","72")),]
dim(counties.2010)
```

```
## [1] 3109    3
```

```
# look at the variables we need to merge on
head(counties.2010)
```

```
##       statefips countyfips    cv1
## 31007         1          1  54571
## 31008         1          3 182265
## 31009         1          5  27457
## 31010         1          7  22915
## 31011         1          9  57322
## 31012         1         11  10914
```

```
head(ccnties)
```

```
##           GEO_ID STATE COUNTY      NAME   LSAD CENSUSAREA division
## 0 0500000US01029    01    029  Cleburne County    560.100        6
## 1 0500000US01031    01    031    Coffee County    678.972        6
## 2 0500000US01037    01    037     Coosa County    650.926        6
## 3 0500000US01039    01    039 Covington County   1030.456        6
## 4 0500000US01041    01    041  Crenshaw County    608.840        6
## 5 0500000US01045    01    045      Dale County    561.150        6
```

If you've been paying careful attention, you may have realized that while this dataframe and our spatial polygons dataframe both have state and county IDs, their formats are rather different. The state and county in the map are character variables, where each state ID is always two characters, and each county ID is always three. To make this consistent number of characters, when the value is less than the number of characters, strings are filled with leading zeros. For example, California is state code 6, or 06 in the map; Los Angeles county is code 37, and is 037 in the map.

In contrast, county numbers in `counties.2010` are regular old numbers.

In order for the merge between the two datasets to work properly, both variables need to be expressed in the same way. The variables don't have to be named the say, but the do have to be expressed the same way, since the computer will not merge 037 with 37.

To make these consistent, I change the data into character strings (rather than the map into numeric). The

code for this is below. The only new command here is `nchar()`, which counts the number of characters in a string. If the state code is less than two characters, add a leading zero. If the county code is 1 character, add two zeros; if the county code is 2 characters, add one zero. Otherwise, keep it the same.

I check the results using the `table()` command.

```
# need to fix counties.2010 to have character state and county variable
# state values
counties.2010$cstatefips <- as.character(counties.2010$statefips)
counties.2010$cstatefips <- ifelse(nchar(counties.2010$cstatefips)==1,
                                   paste("0",counties.2010$cstatefips,sep=""),counties.2010$cstatefips)
table(counties.2010$cstatefips)
```

```
##
##  01  04  05  06  08  09  10  11  12  13  16  17  18  19  20  21  22  23
##  67  15  75  58  64   8   3   1  67 159  44 102  92  99 105 120  64  16
##  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41
##  24  14  83  87  82 115  56  93  17  10  21  33  62 100  53  88  77  36
##  42  44  45  46  47  48  49  50  51  53  54  55  56
##  67   5  46  66  95 254  29  14 134  39  55  72  23
```

```
# county values
counties.2010$ccountyfips <- as.character(counties.2010$countyfips)
counties.2010$ccountyfips <- ifelse(nchar(counties.2010$ccountyfips)==1,
                                   paste("00",counties.2010$ccountyfips,sep=""),
                                   ifelse(nchar(counties.2010$ccountyfips)==2,
                                          paste("0",counties.2010$ccountyfips,sep=""),
                                          counties.2010$ccountyfips))
table(counties.2010$ccountyfips)
```

```
##
## 001 003 005 006 007 009 011 012 013 014 015 017 019 021 023 025 027 028
##  48  48  48   1  46  47  46   1  46   1  46  45  45  44  44  42  44   1
## 029 031 033 035 036 037 039 041 043 045 047 049 051 053 055 057 059 061
##  41  41  40  39   1  39  38  38  38  38  37  36  36  36  35  36  35  35
## 063 065 067 069 071 073 075 077 078 079 081 083 085 086 087 089 091 093
##  34  34  34  34  34  33  33  33   1  32  32  32  32   1  32  31  31  30
## 095 097 099 101 103 105 107 109 111 113 115 117 119 121 123 125 127 129
##  30  30  30  30  30  30  29  29  28  27  27  26  26  26  25  25  24  22
## 131 133 135 137 139 141 143 145 147 149 151 153 155 157 159 161 163 165
##  22  22  20  20  19  19  18  18  18  18  16  17  16  16  16  16  16  15
## 167 169 171 173 175 177 179 181 183 185 186 187 189 191 193 195 197 199
##  14  14  14  14  13  12  12  12  12  11   1  10   9   8   8   9   9   8
## 201 203 205 207 209 211 213 215 217 219 221 223 225 227 229 231 233 235
##   6   5   5   5   5   4   4   4   4   4   4   4   4   4   4   3   3   3
## 237 239 241 243 245 247 249 251 253 255 257 259 261 263 265 267 269 271
##   3   3   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2
## 273 275 277 279 281 283 285 287 289 291 293 295 297 299 301 303 305 307
##   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2   2
## 309 311 313 315 317 319 321 323 325 327 329 331 333 335 337 339 341 343
##   2   2   2   2   2   2   2   1   1   1   1   1   1   1   1   1   1   1
## 345 347 349 351 353 355 357 359 361 363 365 367 369 371 373 375 377 379
##   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
## 381 383 385 387 389 391 393 395 397 399 401 403 405 407 409 411 413 415
##   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
## 417 419 421 423 425 427 429 431 433 435 437 439 441 443 445 447 449 451
```

```
##   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
## 453 455 457 459 461 463 465 467 469 471 473 475 477 479 481 483 485 487
##   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
## 489 491 493 495 497 499 501 503 505 507 510 515 520 530 540 550 570 580
##   1   1   1   1   1   1   1   1   1   1   4   1   1   1   1   1   1   1
## 590 595 600 610 620 630 640 650 660 670 678 680 683 685 690 700 710 720
##   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1   1
## 730 735 740 750 760 770 775 790 800 810 820 830 840
##   1   1   1   1   1   1   1   1   1   1   1   1   1
```

The only thing left to do before merging (you could do it after merging, but this seemed simpler to me, since we don't have to work with the spatial data frame) is to make deciles of population size for the maps we want to make.

As we have done in previous classes, we make a vector that holds the quantiles of population, `pdec`. We then "cut" the `counties.2010` dataset by this decile vector, so that each county has a marker for the decile into which it falls. I use the table function to check that counties are evenly divided by decile as they should be, and they are.

```
# lets make deciles of population
# similar to what we did last class
pdec <- quantile(counties.2010$cv1, probs=seq(0,1,0.1), na.rm = TRUE)
pdec
```

```
##        0%       10%       20%       30%       40%       50%       60%
##      82.0    5297.8    9254.6   13891.6   19105.0   26008.0   36837.8
##       70%       80%       90%      100%
##   52512.0   90918.4  198081.0 9818605.0
```

```
counties.2010$pop.decile <- cut(counties.2010$cv1, pdec,
                            include.lowest = TRUE,
                            right = FALSE)
table(counties.2010$pop.decile)
```

```
##
##        [82,5.3e+03)  [5.3e+03,9.25e+03) [9.25e+03,1.39e+04)
##                 311                 311                 311
## [1.39e+04,1.91e+04)  [1.91e+04,2.6e+04)  [2.6e+04,3.68e+04)
##                 311                 310                 311
## [3.68e+04,5.25e+04) [5.25e+04,9.09e+04) [9.09e+04,1.98e+05)
##                 311                 311                 311
## [1.98e+05,9.82e+06]
##                 311
```

G. Load simple shapefile

Now we are ready to merge the map file and the data file.

However, we are going to use the hot-off-the-presses `sf` function to do this. This function is so new that you had to use the version set up for software developers to use it (that's why you had to use `devtools::` to load it). I recommend the overview file here.

To get the map into a "simple polygon" version, you need to reload it using `sf`'s `st_read()` command. The syntax for this is very similar to `readOGR`. Once you load this file, you can inspect it like a regular dataframe.

```
library(sf)
```

```
## Linking to GEOS 3.6.1, GDAL 2.2.3, proj.4 4.9.3
```

```
sfcounties <- st_read(dsn = "h:/pppa_data_viz/2018/tutorials/lecture07/data",
                      layer ="gz_2010_us_050_00_500k")
```

```
## Reading layer `gz_2010_us_050_00_500k' from data source `h:\pppa_data_viz\2018\tutorials\lecture07\da
## Simple feature collection with 3221 features and 6 fields
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: -179.1473 ymin: 17.88481 xmax: 179.7785 ymax: 71.35256
## epsg (SRID):    4269
## proj4string:    +proj=longlat +datum=NAD83 +no_defs
```

```
sfcounties$geometry
```

```
## Geometry set for 3221 features
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: -179.1473 ymin: 17.88481 xmax: 179.7785 ymax: 71.35256
## epsg (SRID):    4269
## proj4string:    +proj=longlat +datum=NAD83 +no_defs
## First 5 geometries:
```

```
## MULTIPOLYGON (((-85.38872 33.91304, -85.38088 3...
```

```
## MULTIPOLYGON (((-86.03044 31.61894, -86.00408 3...
```

```
## MULTIPOLYGON (((-86.00928 33.10164, -86.00917 3...
```

```
## MULTIPOLYGON (((-86.34851 30.99434, -86.35023 3...
```

```
## MULTIPOLYGON (((-86.14699 31.68045, -86.14711 3...
```

```
names(sfcounties)
```

```
## [1] "GEO_ID"     "STATE"      "COUNTY"     "NAME"       "LSAD"
## [6] "CENSUSAREA" "geometry"
```

You can use these `sf` files to make maps using the `geom_sf()` command in `ggplot`.

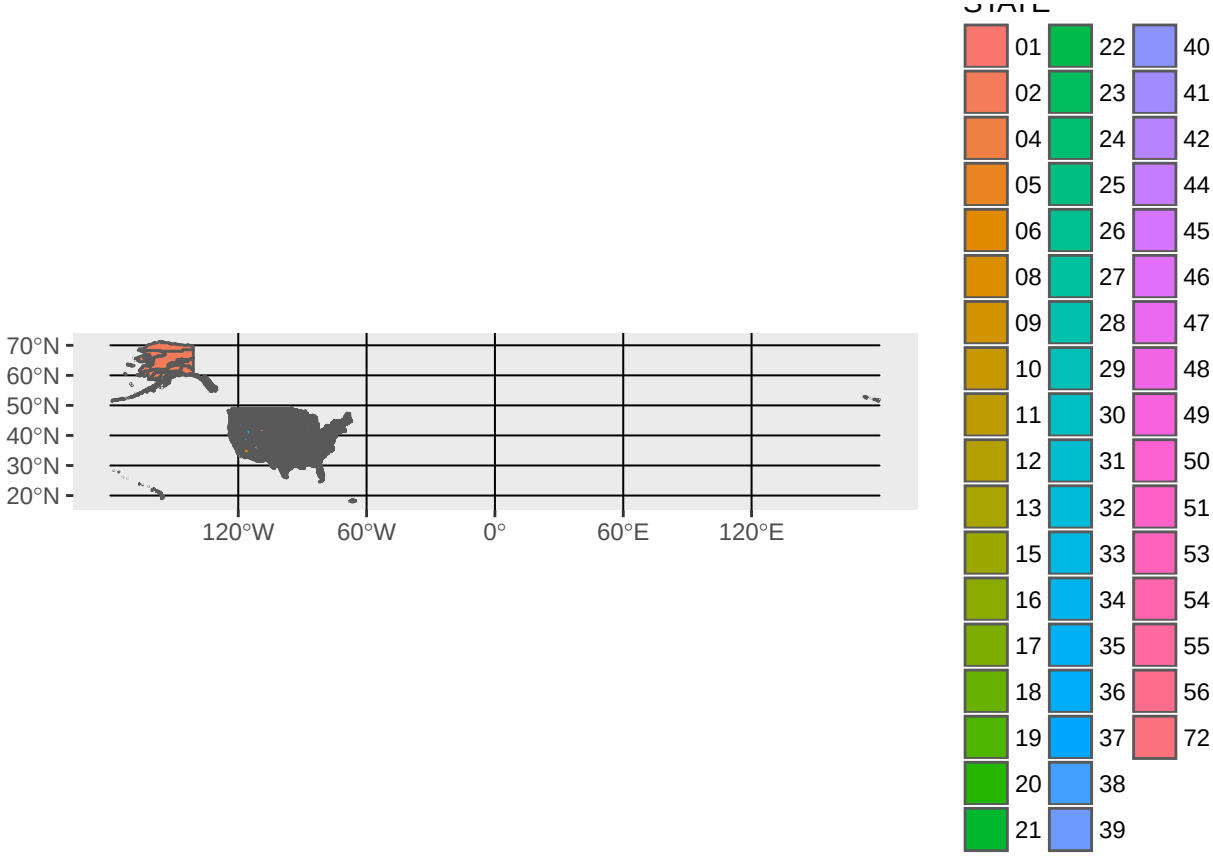
In the examples below, I first chart all states, coloring by color. In the second set of lines I keep only the continental US via subsetting. I then plot the continental US, coloring by state (`aes(fill=STATE)`).

```
library(ggplot2)
```

```
##
## Attaching package: 'ggplot2'
```

```
## The following object is masked from 'package:raster':
##
##     calc
```

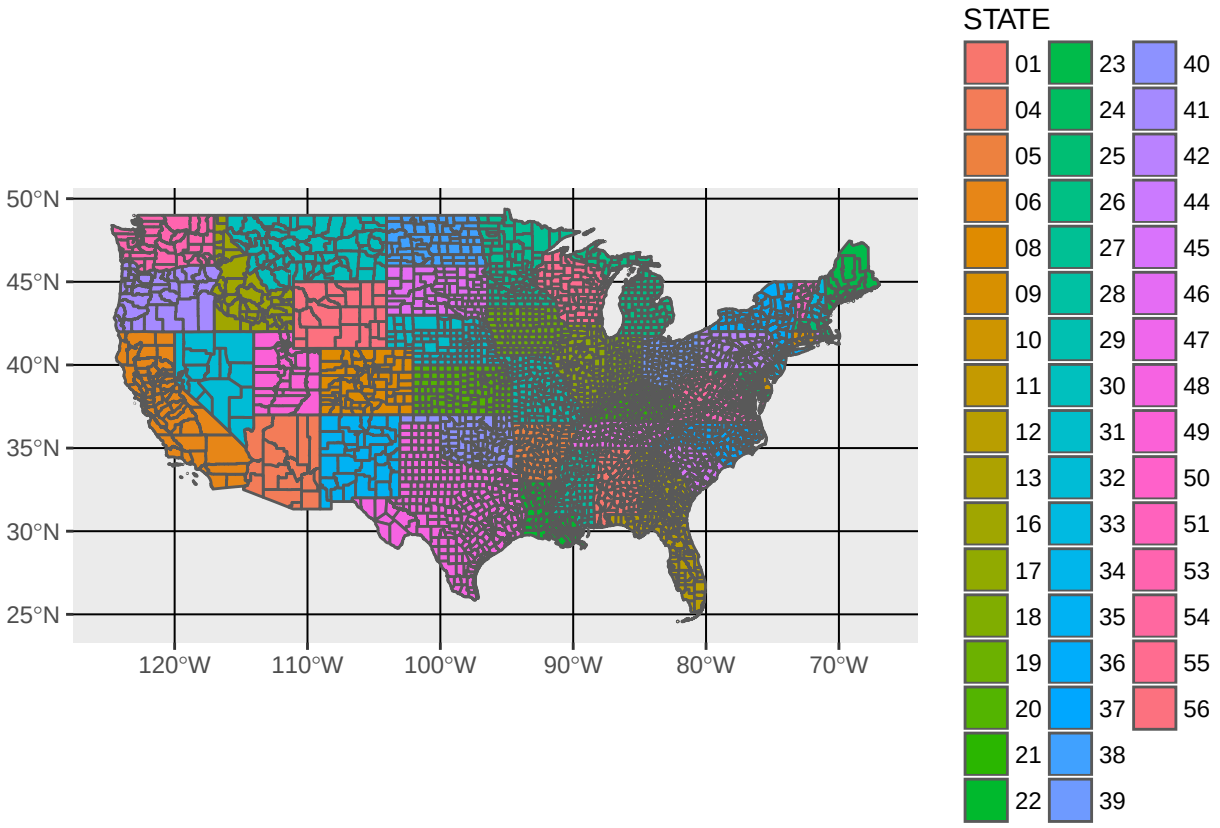
```
ggplot(sfcounties) +
  geom_sf(aes(fill=STATE))
```


70°N
60°N
50°N
40°N
30°N
20°N
120°W
60°W
0°
60°E
120°E
STATE
01
02
04
05
06
08
09
10
11
12
13
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
44
45
46
47
48
49
50
51
53
54
55
56
72

```
# keep only continental us
sfcountiesc <- sfcounties[!(sfcounties$STATE %in% c("02","15","72")),]
dim(sfcountiesc)
```

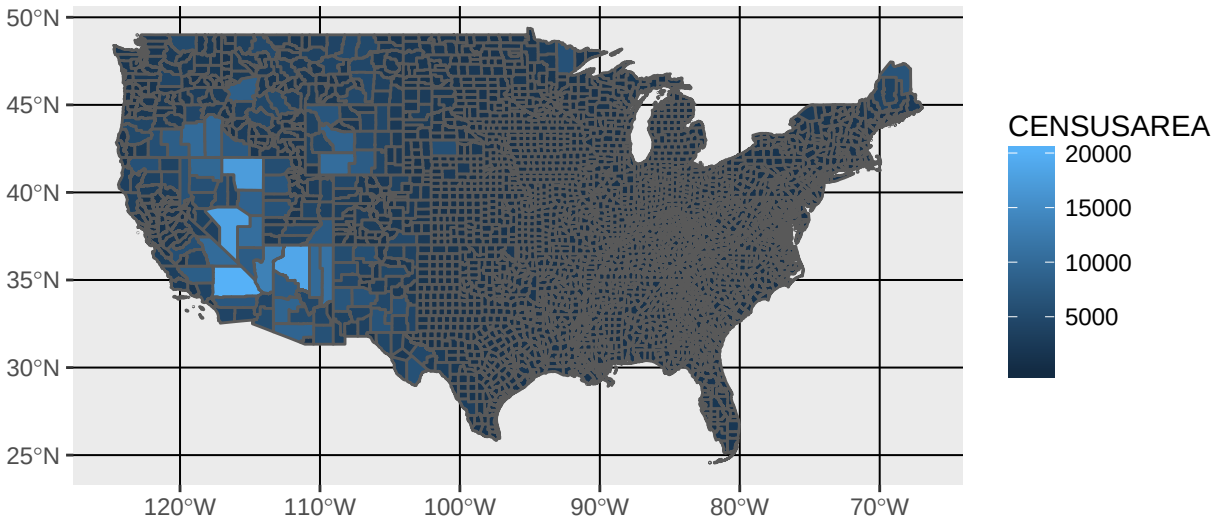
```
## [1] 3109    7
```

```
ggplot(sfcountiesc) +
  geom_sf(aes(fill=STATE))
```



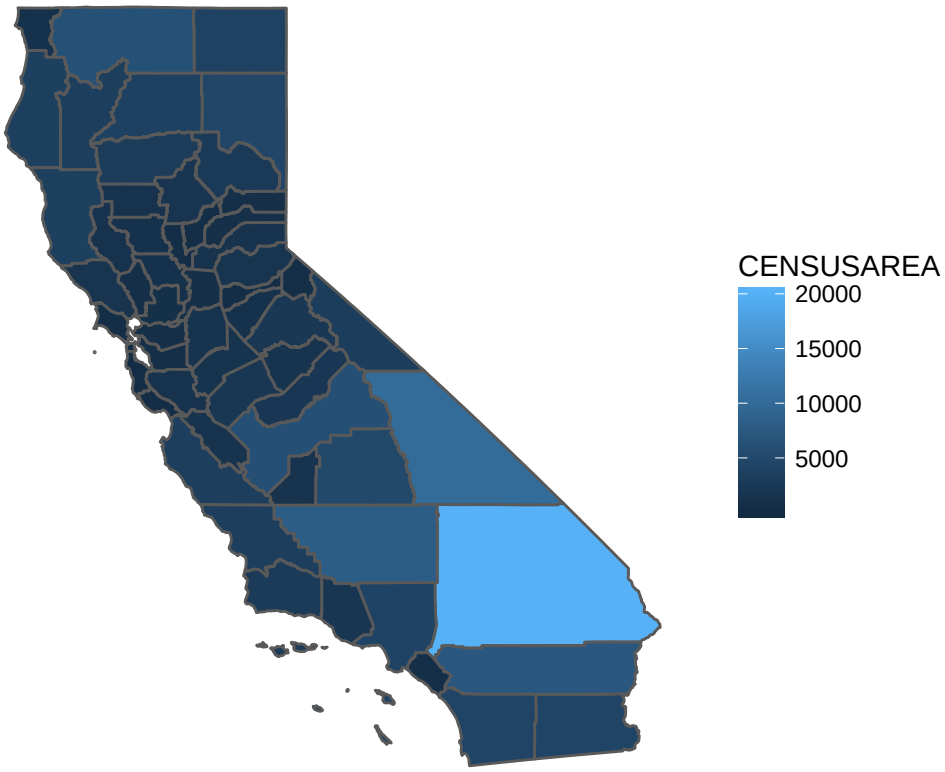
Here are two more examples of ggplot's reach. The first is the same map, but filled by area of the polygon.

```
# make a map, color coding by area (silly)
library(ggplot2)
ggplot(sfcountiesc) +
  geom_sf(data = sfcountiesc, aes(fill=CENSUSAREA))
```



The second subsets to California and gets rid of most of the background junk.

```
# lets do just Ca and see if I can get rid of ugly background
ggplot(sfcountiesc[sfcountiesc$STATE == "06",]) +
  geom_sf(aes(fill=CENSUSAREA)) +
  coord_sf(crs = st_crs(sfcountiesc), datum = NA) +
  theme(panel.background = element_blank())
```



H. Merge in County Data

Finally, we're ready to merge in the county data. Check the size of the map file and the data file beforehand. Your merged file should have this same number of observations. The command `merge()` puts the datasets together, specifying the variables from the first dataset (`by.x=c("STATE","COUNTY")`) and the second dataset (`by.y=c("cstatefips","ccountyfips")`). The option `all = TRUE` keeps all observations from both datasets, regardless of whether there is a match. This is key for quality control.

Luckily, the datasets merge perfectly – that is each has 3109 observations to start with, and the final merged dataset also have 3109 observations. I know the latter using the `dim()` command, and I also check to see that the new dataframe looks ok with the `head()` command.

```r
# merge the county dataframe with the shapefile
# have to put spatial file first
# then data.frame second
dim(sfcountiesc)
```

```
## [1] 3109    7
```

```r
dim(counties.2010)
```

```
## [1] 3109    6
```

```r
cntystuff <- merge(sfcountiesc,counties.2010,
                       by.x=c("STATE","COUNTY"),
                       by.y=c("cstatefips","ccountyfips"),
                   all = TRUE)
head(cntystuff)
```

```
## Simple feature collection with 6 features and 10 fields
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: -88.02927 ymin: 30.22113 xmax: -85.05307 ymax: 34.26048
## epsg (SRID):    4269
## proj4string:    +proj=longlat +datum=NAD83 +no_defs
##   STATE COUNTY         GEO_ID    NAME   LSAD CENSUSAREA statefips
## 1    01    001 0500000US01001 Autauga County    594.436         1
## 2    01    003 0500000US01003 Baldwin County   1589.784         1
## 3    01    005 0500000US01005 Barbour County    884.876         1
## 4    01    007 0500000US01007    Bibb County    622.582         1
## 5    01    009 0500000US01009  Blount County    644.776         1
## 6    01    011 0500000US01011 Bullock County    622.805         1
##   countyfips    cv1          pop.decile                       geometry
## 1          1  54571 [5.25e+04,9.09e+04) MULTIPOLYGON (((-86.52469 3...
## 2          3 182265 [9.09e+04,1.98e+05) MULTIPOLYGON (((-87.41247 3...
## 3          5  27457  [2.6e+04,3.68e+04) MULTIPOLYGON (((-85.13285 3...
## 4          7  22915  [1.91e+04,2.6e+04) MULTIPOLYGON (((-87.11632 3...
## 5          9  57322 [5.25e+04,9.09e+04) MULTIPOLYGON (((-86.73121 3...
## 6         11  10914 [9.25e+03,1.39e+04) MULTIPOLYGON (((-85.74209 3...
```

```r
dim(cntystuff)
```

```
## [1] 3109   11
```

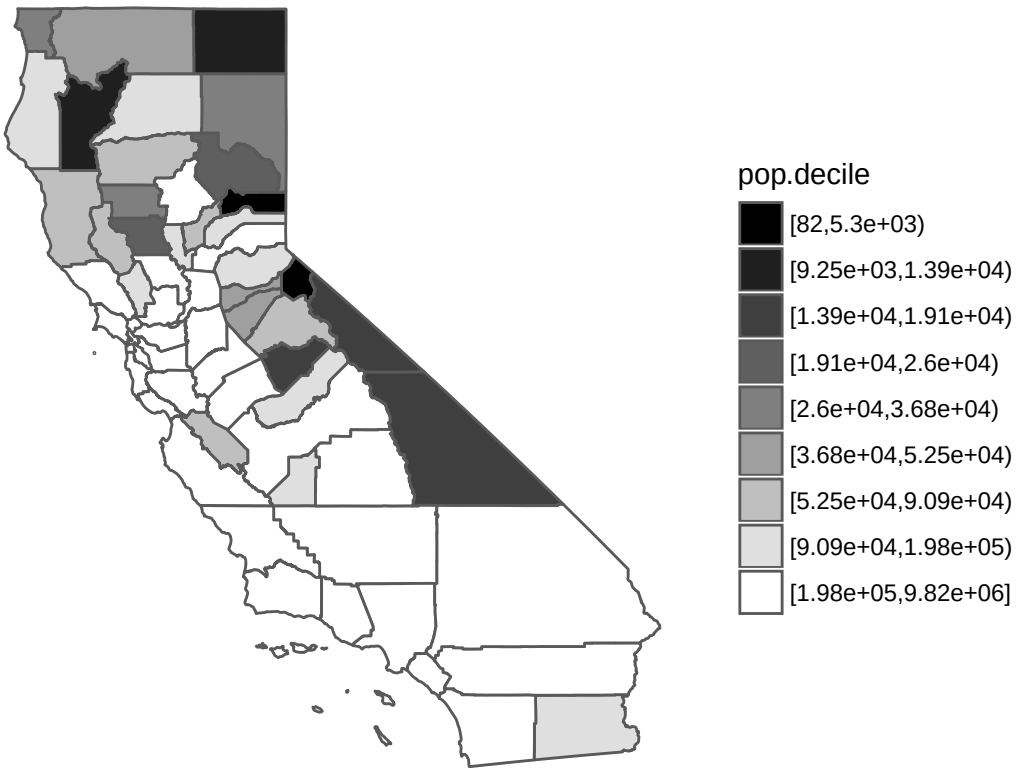
Finally, I create population density for mapping.

```
# make population density
cntystuff$pop.density <- cntystuff$cv1/cntystuff$CENSUSAREA/1000
```

I. Make maps from these merged data

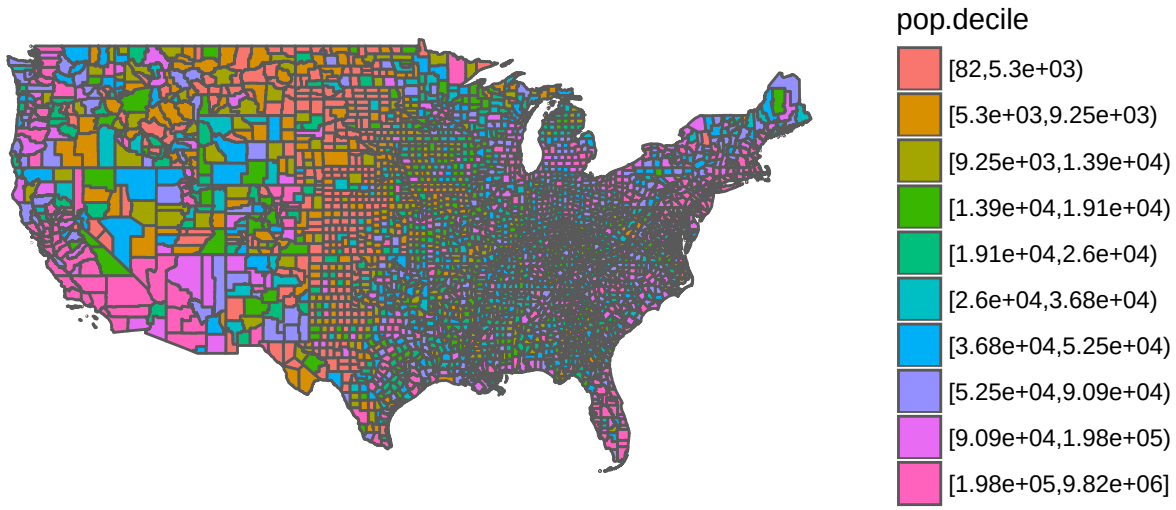
Here is a black and white map of California plotting population density (recall we defined this on a national scale).

```
# population decile
ggplot(cntystuff[cntystuff$STATE == "06",]) +
  scale_fill_manual(values = colorRampPalette(c("black","white"))(9)) +
  geom_sf(aes(fill=pop.decile)) +
  coord_sf(crs = st_crs(cntystuff), datum = NA) +
  theme(panel.background = element_blank())
```



Here is a similar map for the entire US.

```
# population decile
ggplot(cntystuff) +
  geom_sf(aes(fill=pop.decile)) +
  coord_sf(crs = st_crs(cntystuff), datum = NA) +
  scale_color_gradient() +
  theme(panel.background = element_blank())
```



J. Homework

1. Repeat the final charts with another county variable that has a different range from what we plotted in the tutorial.
2. Make a choropleth map from another dataset! It is ok to bring in a shapefile that already has values attached so you can skip the bothersome merging bit.