A `maps` quirk and a `tmap` solution

Will Lowe & Asya Magazinnik

2/8/2017

For those who are interested, this tutorial walks through the buggy features of the `maps` package, then provides a better alternative. The important thing to know about using the `map()` function in `maps` is to loop over regions to color them one by one rather than providing a vector of colors in one function call. We'll first illlustrate why.

`maps`

Let's load a dataset containing the colors we wish to use for each county, identified by FIPS code.

```
load("data/county_data_full.RData")
head(final)
```

```
##   FIPS state          county     cols1
## 1 1001    AL Autauga County #FF000033
## 2 1003    AL Baldwin County #0000FF33
## 3 1005    AL Barbour County #FF000099
## 4 1007    AL    Bibb County #FF000066
## 5 1009    AL  Blount County #FF000066
## 6 1011    AL Bullock County #FF000033
```

We'll need to merge this dataset with the mapping data used by the `maps` library, by FIPS code.

```
library(maps)
library(ggmap)
```

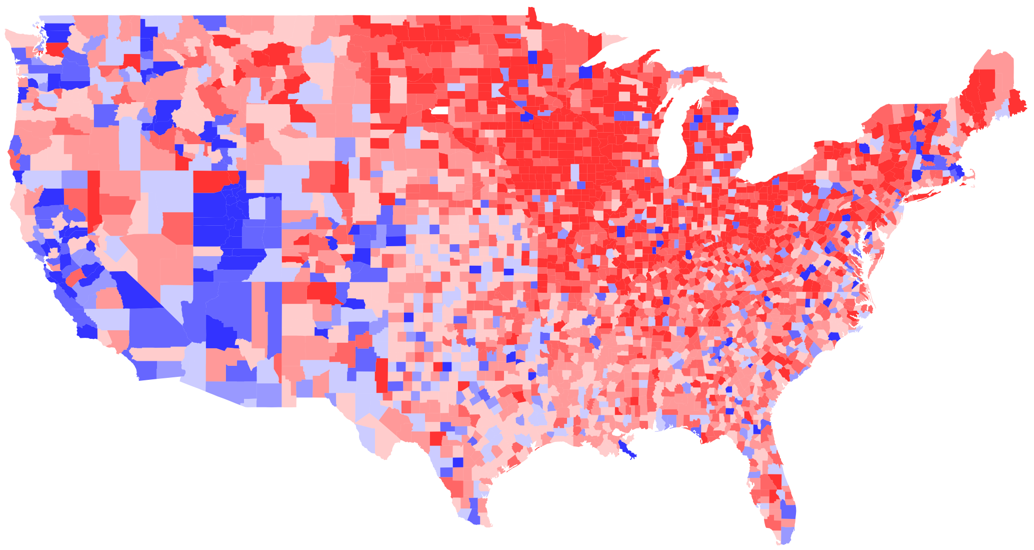
```
## Loading required package: ggplot2
```

```
cf <- county.fips # from maps library
names(cf) <- c("FIPS", "name")
toplot <- merge(cf, final, by = "FIPS", all.x = TRUE)
head(toplot)
```

```
##   FIPS            name state          county     cols1
## 1 1001 alabama,autauga    AL Autauga County #FF000033
## 2 1003 alabama,baldwin    AL Baldwin County #0000FF33
## 3 1005 alabama,barbour    AL Barbour County #FF000099
## 4 1007    alabama,bibb    AL    Bibb County #FF000066
## 5 1009  alabama,blount    AL  Blount County #FF000066
## 6 1011 alabama,bullock    AL Bullock County #FF000033
```

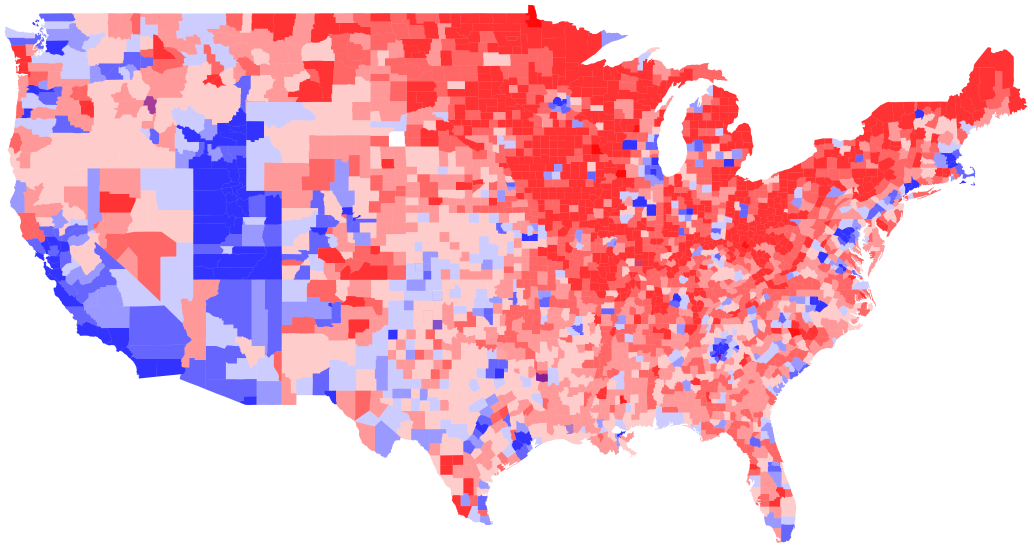
Now let's try making a map by providing a vector of colors to one map call.

```
map(database = "county", col = toplot$cols1, lty = 0, fill = TRUE)
```



And let's try an alternative approach: activating an empty map and looping over counties to color them one by one.

```r
map(database = "county", lty = 0) # activate empty map
for(i in 1:nrow(toplot)) {
  map(database = "county", regions = toplot$name[i], col = toplot$cols1[i],
      fill = TRUE, add = TRUE, lty = 0)
}
```



We see the same overall patterns, but if you look closely, there are subtle differences between the two maps (look at the Northwest and Northeast tips of the map). Which one is right?

The answer lies in the fact that we used the `county.fips` dataset as a base for the color vector, but the `map()` function call uses a different dataset for its geography. We can take a look at it using the `map_data()` function.

```r
geog <- map_data("county")
head(geog)
```

```
##        long      lat group order  region subregion
## 1 -86.50517 32.34920     1     1 alabama   autauga
## 2 -86.53382 32.35493     1     2 alabama   autauga
## 3 -86.54527 32.36639     1     3 alabama   autauga
## 4 -86.55673 32.37785     1     4 alabama   autauga
## 5 -86.57966 32.38357     1     5 alabama   autauga
## 6 -86.59111 32.37785     1     6 alabama   autauga
```

It turns out that the regions in this dataset don't align perfectly with the `county.fips` dataset.

```
length(unique(paste(geog$region, geog$subregion))) # county dataset
```

```
## [1] 3076
```

```
length(cf$name) # fips dataset
```

```
## [1] 3085
```

Whoops. It turns out these also aren't in the same order, which matters. Why didn't we get an error if we weren't using a color vector of the appropriate length? Because, as the documentation will tell you, it cycles through the vector given to `col`; if it stops short of the regions in the data, it goes back to the first position in the vector.

We can still use our first approach, but we have to use the `regions` argument to explicitly tell it which counties align with our color vector. To do so, we'll have to left-join our data onto the geography dataset.

```
geog <- unique(map_data("county")[,c("group", "region", "subregion")])
geog$name <- paste(geog$region, geog$subregion, sep=",")
merged <- dplyr::left_join(geog, toplot, by="name")
```

```
## Warning in left_join_impl(x, y, by$x, by$y, suffix$x, suffix$y): joining
## factor and character vector, coercing into character vector
```

Now let's see if we can get the same map as looping.

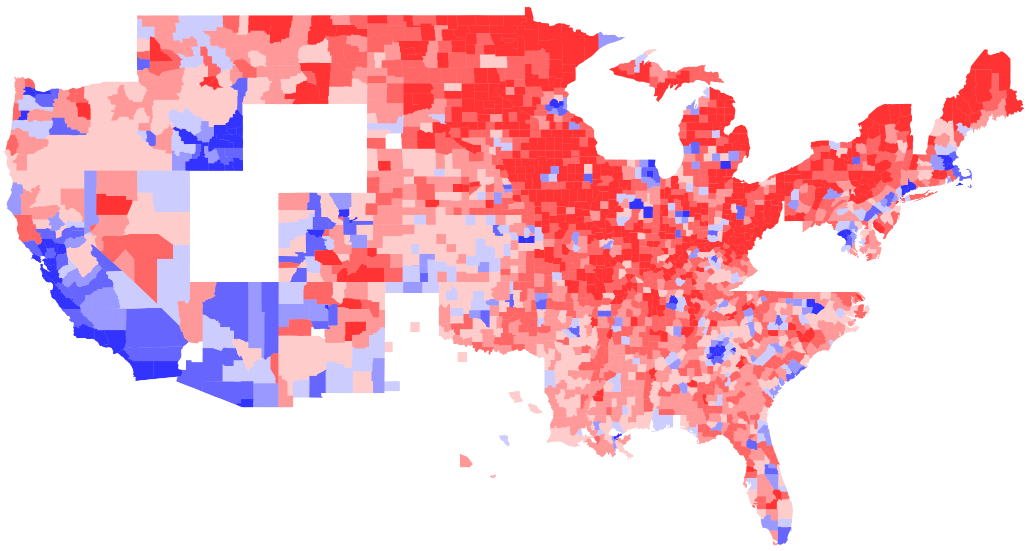
```
map(database = "county", regions = merged$name, col = merged$cols1,
    lty = 0, fill = TRUE)
```

Not so fast. It returns the error message:

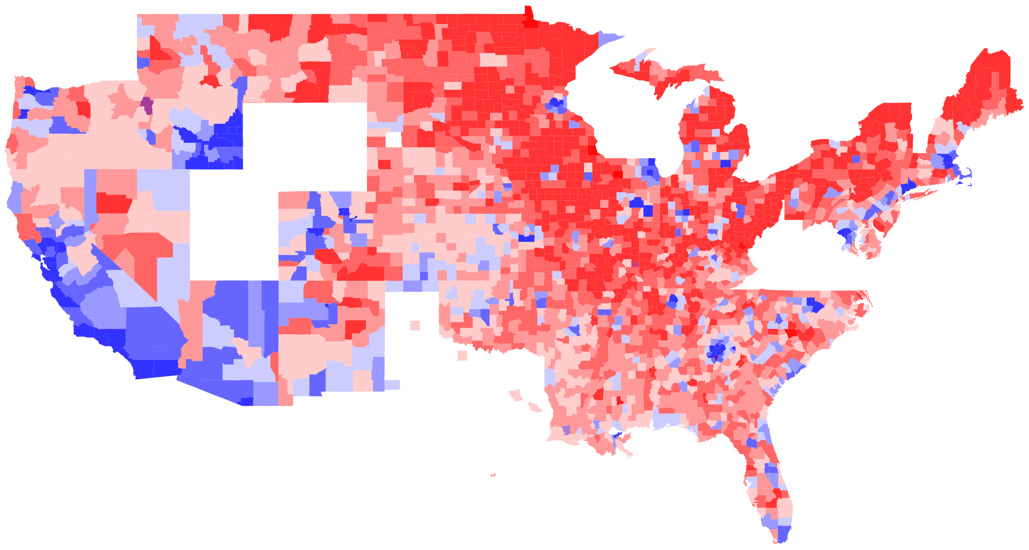
```
Error in grep(regexp, nam, ignore.case = TRUE, perl = (length(patterns) <  :
  invalid regular expression '(^alabama,autauga)|(^alabama,baldwin)|(^alabama,barbour)|
(^alabama,bibb)|(^alabama,blount)|(^alabama,bullock)|(^alabama,butler)|
(^alabama,calhoun)|(^alabama,chambers)|(^alabama,cherokee)|(^alabama,chilton)|
(^alabama,choctaw)|(^alabama,clarke)|(^alabama,clay)|(^alabama,cleburne)|
(^alabama,coffee)|(^alabama,colbert)|(^alabama,conecuh)|(^alabama,coosa)|
(^alabama,covington)|(^alabama,crenshaw)|(^alabama,cullman)|(^alabama,dale)|
(^alabama,dallas)|(^alabama,dekalb)|(^alabama,elmore)|(^alabama,escambia)|
(^alabama,etowah)|(^alabama,fayette)|(^alabama,franklin)|(^alabama,geneva)|
(^alabama,greene)|(^alabama,hale)|(^alabama,henry)|(^alabama,houston)|(^alabama,jackson)|
(^alabama,jefferson)|(^alabama,lamar)|(^alabama,lauderdale)|(^alabama,lawrence)|
(^alabama,lee)|(^alabama,limestone)|(^alabama,lowndes)|(^alabama,macon)|
(^alabama,madison)|(^alabama,marengo)|(^alabama,marion)|(^alabama,marshall)|
(^alabama,mobile)|(^alabama,monroe)|(^alabama,montgomery)|(^alabama'
```

What now? The issue is that the `map` function's source code uses the `grep` command, which can only take an argument up to a certain length; all the counties we want to plot turn out to be too many. If we do it with the first 2500 observations, we get the same results as looping.

```
# vector approach
map(database = "county", regions = merged$name[1:2500], col = merged$cols1[1:2500],
    lty = 0, fill = TRUE)
```



```r
# loop approach
map(database = "county", lty = 0) # activate empty map
for(i in 1:2500) {
  map(database = "county", regions = toplot$name[i], col = toplot$cols1[i],
      fill = TRUE, add = TRUE, lty = 0)
}
```



The bottom line? You should loop. But that's still slow and clunky, and it's difficult to provide your own geographies. A better approach is `tmap`; a quick illustration of how it works follows.

`tmap`

First, download and unpack some county shapefiles from the the census. These are 1:20M scale so the plots go relatively quickly.

```r
loc <- file.path('data', "counties1to20m.zip")
download.file("http://www2.census.gov/geo/tiger/GENZ2015/shp/cb_2015_us_county_20m.zip",
              loc)
unzip(loc, exdir='data')
```

Get the plotting library and its corresponding data manipulation library

```
library(tmap)
```

```
## Warning: package 'tmap' was built under R version 3.3.2
```

```
library(tmaptools) # for the shape file reader
```

```
## Warning: package 'tmaptools' was built under R version 3.3.2
```

Read the shapefile and turn it into a `SpatialPolygonsDataFrame` which behaves mostly like a `data.frame` but has spatial polygons associated with the rows.

```
cts <- read_shape(file.path('data', "cb_2015_us_county_20m.shp"))
```

Create a proper FIPS variable from the two components that the Census provides

```
cts$FIPS <- as.numeric(paste0(cts$STATEFP, cts$COUNTYFP))
```

Now to merge our metadata to the county shapes. This won't work first time...

```
append_data(cts, final, key.shp = "FIPS", key.data = "FIPS")
```

```
## Under coverage: 109 out of 3220 shape features did not get appended data. Run under_coverage() to ge
```

OK, so let's take a use the output of `under_coverage()` to remove some elements from the shape file. It tells us the ids and FIPS values of the offending polygons.

```
cts.smaller <- cts[-under_coverage()$id, ]
```

Now let's try that merge again.

```
cts.merged <- append_data(cts.smaller, final, key.shp="FIPS", key.data="FIPS")
```

```
## Keys match perfectly.
```

That's better.

Notice that `append_data` always *left joins* on the object holding the polygons. That's why we had to adjust that file a bit.

You might have been tempted to use `merge` for this, but that would have silently thrown away all the shape information that's secreted in `cts`, which would have sunk everything (with a confusing error about lacking projection information). So don't do that.

Now to make a nice plot. `tmap` has a ggplot style syntax using `+`. Here we just make a simple chloropleth:

```
tm_shape(cts.merged) + tm_fill("cols1")
```