Reporting data results #2

Matrices and lists

Matrices and lists

This week, we'll talk about the `apply` family of functions, which allow you to apply a function to all values in a vector, matrix, or list.

First, you need to know about two more object types in R:

- `matrix`
- `list`

MATRIX

A matrix is like a data frame, but all the values in all columns must be of the same class (e.g., numeric, character).

R uses matrices a lot for its underlying math (e.g., for the linear algebra operations required for fitting regression models). R can do matrix operations quite quickly.

MATRIX

You can create a matrix with the `matrix` function. Input a vector with the values to fill the matrix and `ncol` to set the number of columns:

```
foo <- matrix(1:10, ncol = 5)
foo
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
```

MATRIX

By default, the matrix will fill up by column. You can fill it by row with the `byrow` function:

```
foo <- matrix(1:10, ncol = 5, byrow = TRUE)
foo
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    2    3    4    5
## [2,]    6    7    8    9   10
```

MATRIX

If you want to convert a data frame to a matrix, you can use the `as.matrix` function:

```
foo <- data.frame(col_1 = 1:2, col_2 = 3:4,
                  col_3 = 5:6, col_4 = 7:8,
                  col_5 = 9:10)
(foo <- as.matrix(foo))
```

```
##      col_1 col_2 col_3 col_4 col_5
## [1,]     1     3     5     7     9
## [2,]     2     4     6     8    10
```

MATRIX

You can index matrices with square brackets, just like data frames:

```
foo[1, 1:2]
```

```
## col_1 col_2
##     1     3
```

You cannot, however, use "tidyverse" functions with matrices:

```
foo %>% filter(col_1 == 1)
```

```
Error in UseMethod("filter_") :
  no applicable method for 'filter_' applied to
  an object of class "c('matrix', 'integer',
  'numeric')"
```

MATRIX

All elements in a matrix must have the same class.

The matrix will default to make all values the most general class of any of the values, in any column. For example, if we replaced one numeric value with the character "a", everything would turn into a character:

```
foo[1, 1] <- "a"
foo
```

```
##      col_1 col_2 col_3 col_4 col_5
## [1,] "a"   "3"   "5"   "7"   "9"
## [2,] "2"   "4"   "6"   "8"   "10"
```

LIST

A list has different elements, just like a data frame has different columns. However, the different elements of a list can have different lengths (unlike the columns of a data frame). The different elements can also have different classes.

```
bar <- list(some_letters = letters[1:3],
            some_numbers = 1:5,
            some_logical_values = c(TRUE, FALSE))
bar
```

```
## $some_letters
## [1] "a" "b" "c"
##
## $some_numbers
## [1] 1 2 3 4 5
##
## $some_logical_values
## [1]  TRUE FALSE
```

LIST

To index an element from a list, use double square brackets. You can use bracket indexing either with numbers (which element in the list?) or with names. You can also index lists with the $ operator.

```
bar[[1]]
```

```
## [1] "a" "b" "c"
```

```
bar[["some_numbers"]]
```

```
## [1] 1 2 3 4 5
```

```
bar$some_logical_values
```

```
## [1]  TRUE FALSE
```

LIST

Lists can be used to contain data with an unusual structure and / or lots of different components. For example, the information from fitting a regression is often stored as a list:

```
my_mod <- glm(rnorm(10) ~ c(1:10))
is.list(my_mod)
```

```
## [1] TRUE
```

LIST

```
head(names(my_mod), 3)
```

```
## [1] "coefficients"  "residuals"     "fitted.values"
```

```
my_mod[["coefficients"]]
```

```
## (Intercept)      c(1:10)
##  -0.4815121    0.1367938
```

apply functions

`apply` family

There is a whole family of `apply` functions, as part of base R. These include:

- `apply`: Apply a function over all the rows (`MARGIN = 1`) or columns (`MARGIN = 2`) of a matrix
- `lapply`: Apply a function over elements of a list.
- `sapply`: Like `lapply`, but returns a vector instead of a list.

APPLY

Here is the syntax for apply:

```
## Generic code
apply([matrix], MARGIN = [margin (1: rows, 2: columns)],
      FUN =  [function])
```

I'll use the worldcup data as an example:

```
ex <- worldcup[ , c("Shots", "Passes", "Tackles", "Saves")]
head(ex)
```

```
##              Shots Passes Tackles Saves
## Abdoun           0      6       0     0
## Abe              0    101      14     0
## Abidal           0     91       6     0
## Abou Diaby       1    111       5     0
## Aboubakar        2     16       0     0
## Abreu            0     15       0     0
```

APPLY

Take the mean of all columns:

```
apply(ex, MARGIN = 2, mean)
```

```
##      Shots     Passes    Tackles      Saves
##  2.3042017 84.5210084  4.1915966  0.6672269
```

Take the sum of all rows:

```
head(apply(ex, MARGIN = 1, sum), 4)
```

```
##     Abdoun         Abe     Abidal Abou Diaby
##          6         115          97        117
```

APPLY

You can use your own function with any of the apply functions. For example, if you wanted to calculate a value for each player that is a weighted mean of some of their statistics, you could run:

```
weighted_mean <- function(soccer_stats,
                          weights = c(0.40, 0.01,
                                      0.25, 1.5)){
  out <- sum(weights * soccer_stats)
  return(out)
}

head(apply(ex, MARGIN = 1, weighted_mean), 4)
```

```
##      Abdoun         Abe      Abidal Abou Diaby
##        0.06        4.51        2.41        2.76
```

lapply

`lapply()` will apply a function across a list. The different elements of the list do not have to be the same length (unlike a data frame, where the columns all have to have the same length).

```
(ex <- list(a = c(1:5), b = rnorm(3), c = letters[1:4]))
```

```
## $a
## [1] 1 2 3 4 5
##
## $b
## [1] 0.2440823 0.2866286 0.1111254
##
## $c
## [1] "a" "b" "c" "d"
```

lapply

This call will calculate the mean of each function:

```
lapply(ex, FUN = mean)
```

```
## $a
## [1] 3
##
## $b
## [1] 0.2139454
##
## $c
## [1] NA
```

lapply

You can include arguments for the function that you specify with `FUN`, and they'll be passed to that function. For example, to get the first value of each element, you can run:

```
lapply(ex, FUN = head, n = 1)
```

```
## $a
## [1] 1
##
## $b
## [1] 0.2440823
##
## $c
## [1] "a"
```

sapply

`sapply()` also applies a function over a list, but it returns a vector rather than a list:

```
sapply(ex, FUN = head, n = 1)
```

```
##                   a                   b                   c
##                 "1" "0.24408234622259"                 "a"
```

`apply` family

In practice, I do use `apply()` some, but I can often find a way to do similar things to other `apply` family functions using the tools in `dplyr`.

You should know that `apply` family functions take advantage of the matrix structure in R. This can be one of the fastest way to run code in R. It is usually a lot faster than doing the same things with loops. However, unless you are working with large data sets, you may not notice a difference, and "tidyverse" functions are usually comparable in speed.

I would recommend using whichever method makes the most sense to you until you run into an analysis that takes a noticeable amount of time to run, and then you might want to work a bit more to optimize your code.

Point maps

Point maps

It is very easy to create point maps in R based on longitude and latitude values of specific locations.

To get a base map, you can use the `map_data` function from the `ggplot2` package to pull data for maps at different levels ("usa", "state", "world", "county").

Point maps

The maps you pull using `map_data` are just data frames. They include the data you need to plot polygon shapes for areas like states and counties.

```
library(ggplot2)
us_map <- map_data("state")
head(us_map, 3)
```

```
##        long      lat group order  region subregion
## 1 -87.46201 30.38968     1     1 alabama      <NA>
## 2 -87.48493 30.37249     1     2 alabama      <NA>
## 3 -87.52503 30.37249     1     3 alabama      <NA>
```

You can add points to these based on latitude and longitude.

Point maps

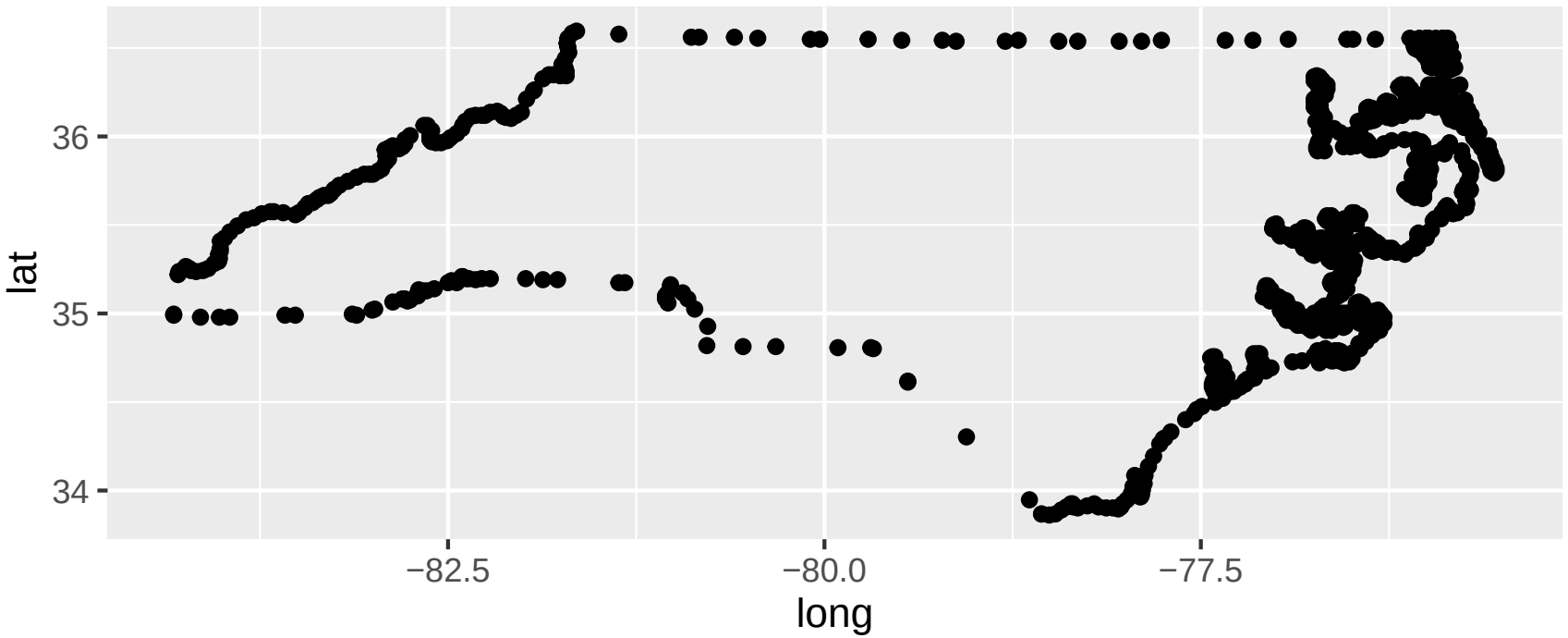
Map choices with `map_data` are currently limited to:

- `county`
- `state`
- `usa`
- `france`
- `italy`
- `nz`
- `world`
- `world2`

Point maps

Mapping uses the `long` and `lat` columns from this data for location:

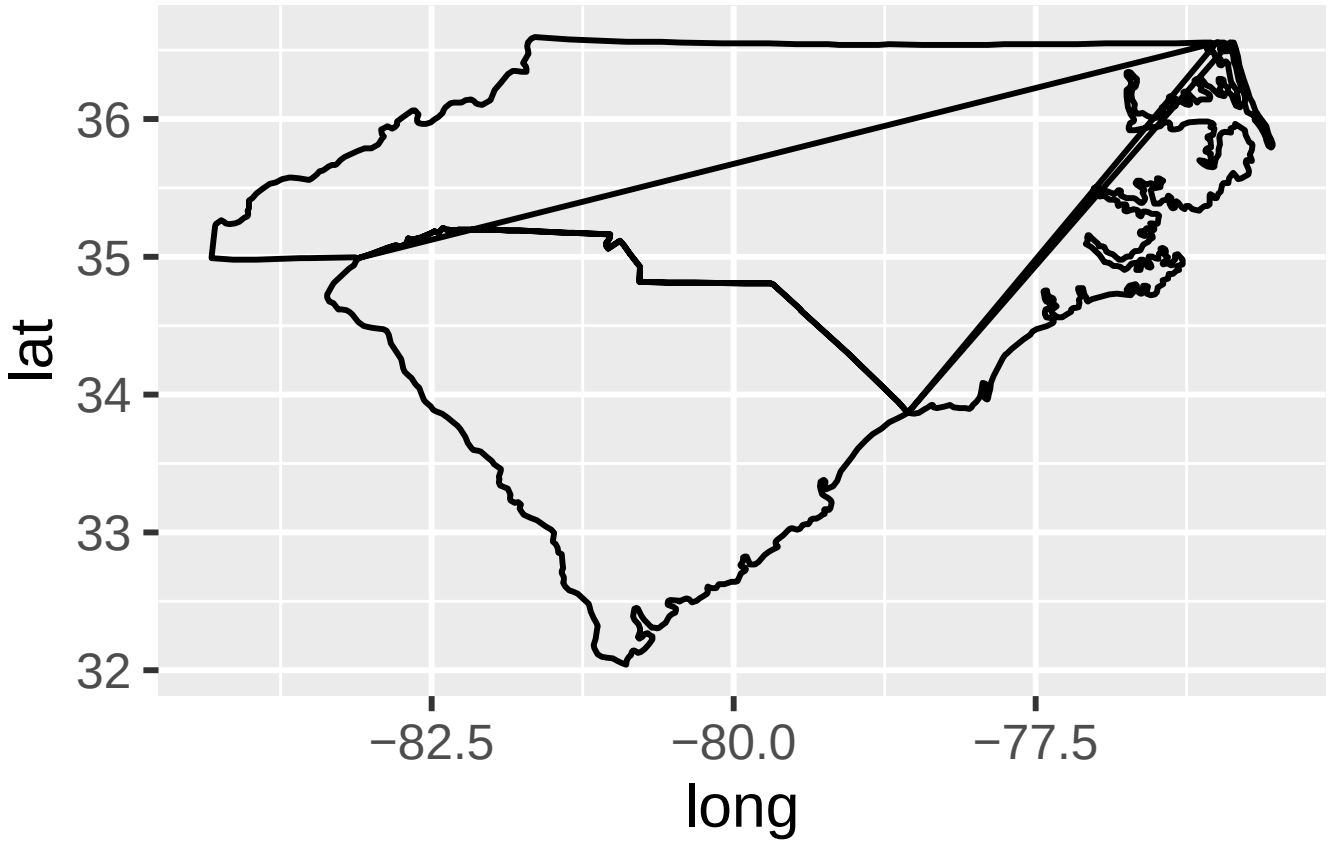
```
north_carolina <- us_map %>%
  filter(region == "north carolina")
ggplot(north_carolina, aes(x = long, y = lat)) +
  geom_point()
```



Point maps

If you try to plot lines, however, you'll have a problem:

```
carolinas <- us_map %>%
  filter(str_detect(region, "carolina"))
ggplot(carolinas, aes(x = long, y = lat)) +
  geom_path()
```



Point maps

The group column fixes this problem. It will plot a separate path or polygon for each separate group. For mapping, this gives separate groupings for mainland versus islands and for different states:

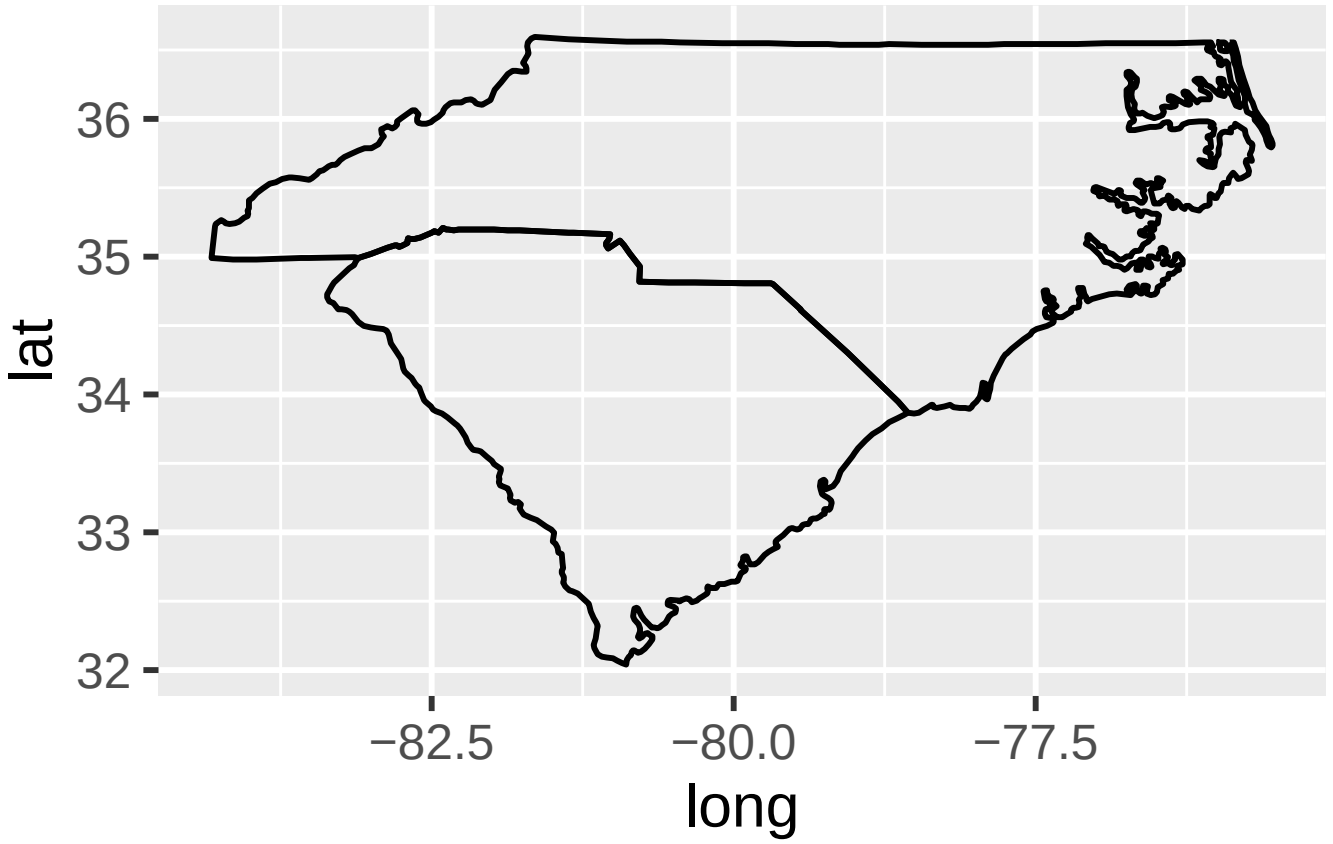
```
carolinas %>%
  group_by(group) %>%
  slice(1)
```

```
## Source: local data frame [4 x 6]
## Groups: group [4]
##
##        long      lat group order         region subregion
##       <dbl>    <dbl> <dbl> <int>          <chr>     <chr>
## 1 -75.89399 36.55471    38  9549 north carolina    knotts
## 2 -78.55824 33.86753    39  9587 north carolina      main
## 3 -76.00285 36.55471    40 10321 north carolina      spit
## 4 -83.10753 34.99053    47 11441 south carolina      <NA>
```

Point maps

Using `group = group` avoids the extra lines from the earlier map:

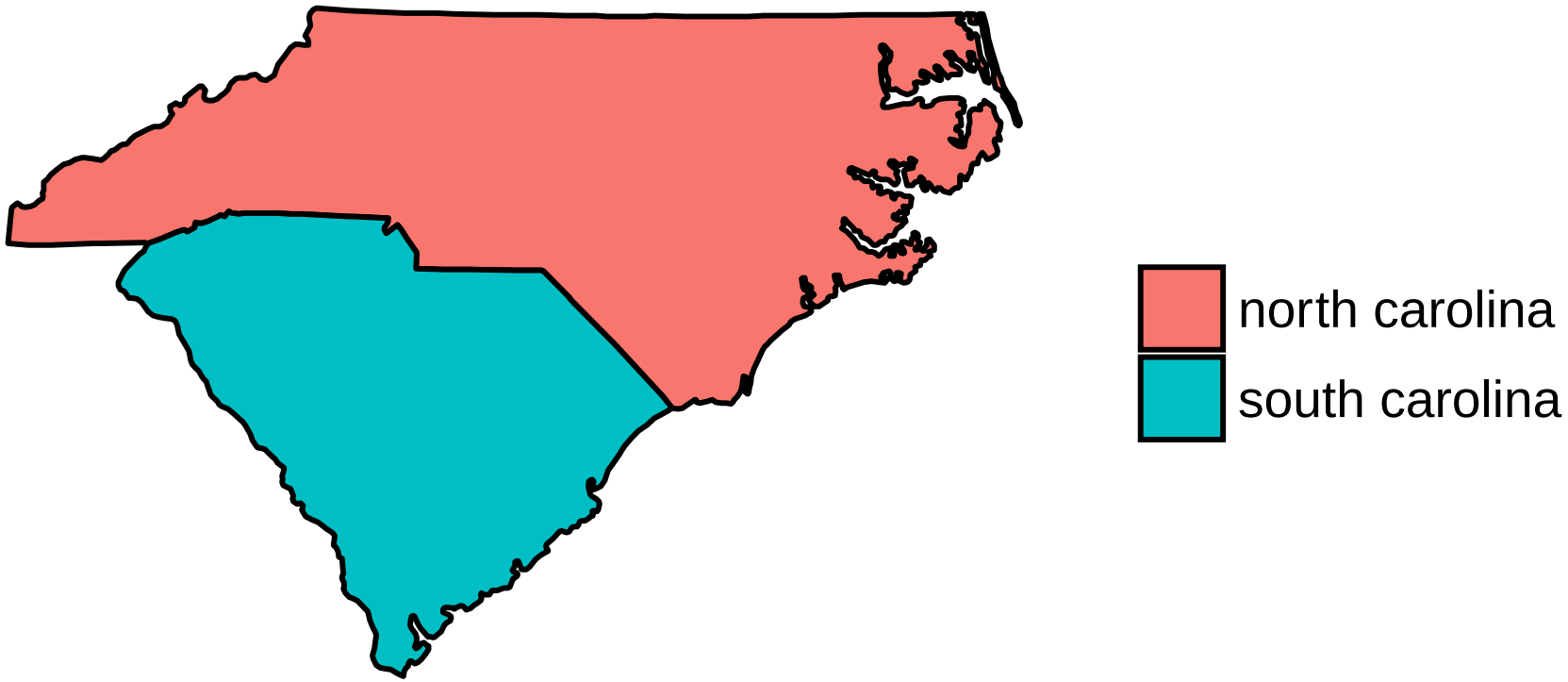
```
ggplot(carolinas, aes(x = long, y = lat,
                      group = group)) +
  geom_path()
```



Point maps

To plot filled regions, use `geom_polygon` with `fill = region`. Also, the "void" theme is often useful when mapping:

```
ggplot(carolinas, aes(x = long, y = lat,
                      group = group,
                      fill = region)) +
  geom_polygon(color = "black") +
  theme_void()
```



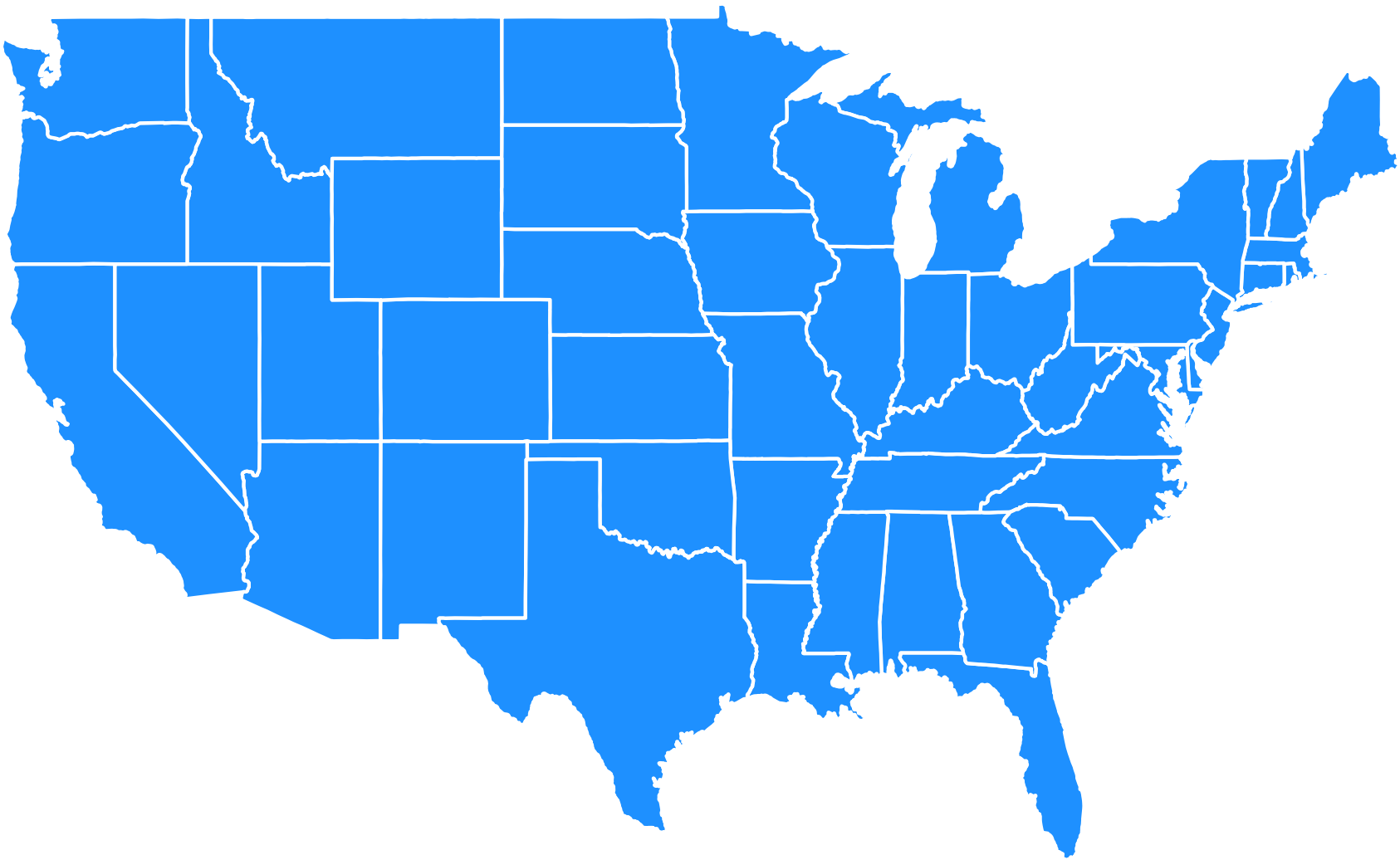
Point maps

Here is an example of plotting all of the US by state:

```
map_1 <- ggplot(us_map, aes(x = long, y = lat,
                            group = group)) +
        geom_polygon(fill = "dodgerblue",
                     color = "white") +
        theme_void()
```

Point maps

```
map_1
```



Point maps

To add points to these maps, you can use `geom_point`, again using longitude and latitude to define position.

Here I'll use an example of data points related to the story told in last year's "Serial" podcast.

```
serial <- read.csv("../data/serial_map_data.csv")
head(serial, 3)
```

```
##     x   y       Type Name Description
## 1 356 437 cell-site L688
## 2 740 360 cell-site L698
## 3 910 340 cell-site L654
```

Point maps

David Robinson figured out a way to convert the x and y coordinates in this data to latitude and longitude coordinates. I'm also adding a column for whether of not the point is a cell tower.

```
library(dplyr)
serial <- serial %>%
    mutate(long = -76.8854 + 0.00017022 * x,
           lat  = 39.23822 + 1.371014e-04 * y,
           tower = Type == "cell-site")
```

Point maps

```
serial[c(1:2, (nrow(serial) - 1):nrow(serial)),
       c("Type", "Name", "long", "lat", "tower")]
```

```
##              Type           Name      long      lat tower
## 1      cell-site           L688 -76.82480 39.29813  TRUE
## 2      cell-site           L698 -76.75944 39.28758  TRUE
## 24 base-location Adnan's house -76.76284 39.30622 FALSE
## 25 base-location  Jenn's house -76.72301 39.29443 FALSE
```

Point maps

Now I can map just Baltimore City and Baltimore County in Maryland and add these points.

I used `map_data` to pull the "county" map and specified "region" as "maryland", to limit the map just to Maryland counties.

```
baltimore <- map_data('county', region = 'maryland')
head(baltimore, 3)
```

```
##        long      lat group order   region subregion
## 1 -78.64992 39.53982     1     1 maryland  allegany
## 2 -78.70148 39.55128     1     2 maryland  allegany
## 3 -78.74159 39.57420     1     3 maryland  allegany
```

Point maps

From that, I filter to just rows where the subregion column was "baltimore city" or "baltimore".

```
baltimore <- baltimore %>%
  filter(subregion %in% c("baltimore city",
                          "baltimore"))
head(baltimore, 3)
```

```
##        long      lat group order   region subregion
## 1 -76.88521 39.35074     3   114 maryland baltimore
## 2 -76.89094 39.37939     3   115 maryland baltimore
## 3 -76.88521 39.40804     3   116 maryland baltimore
```

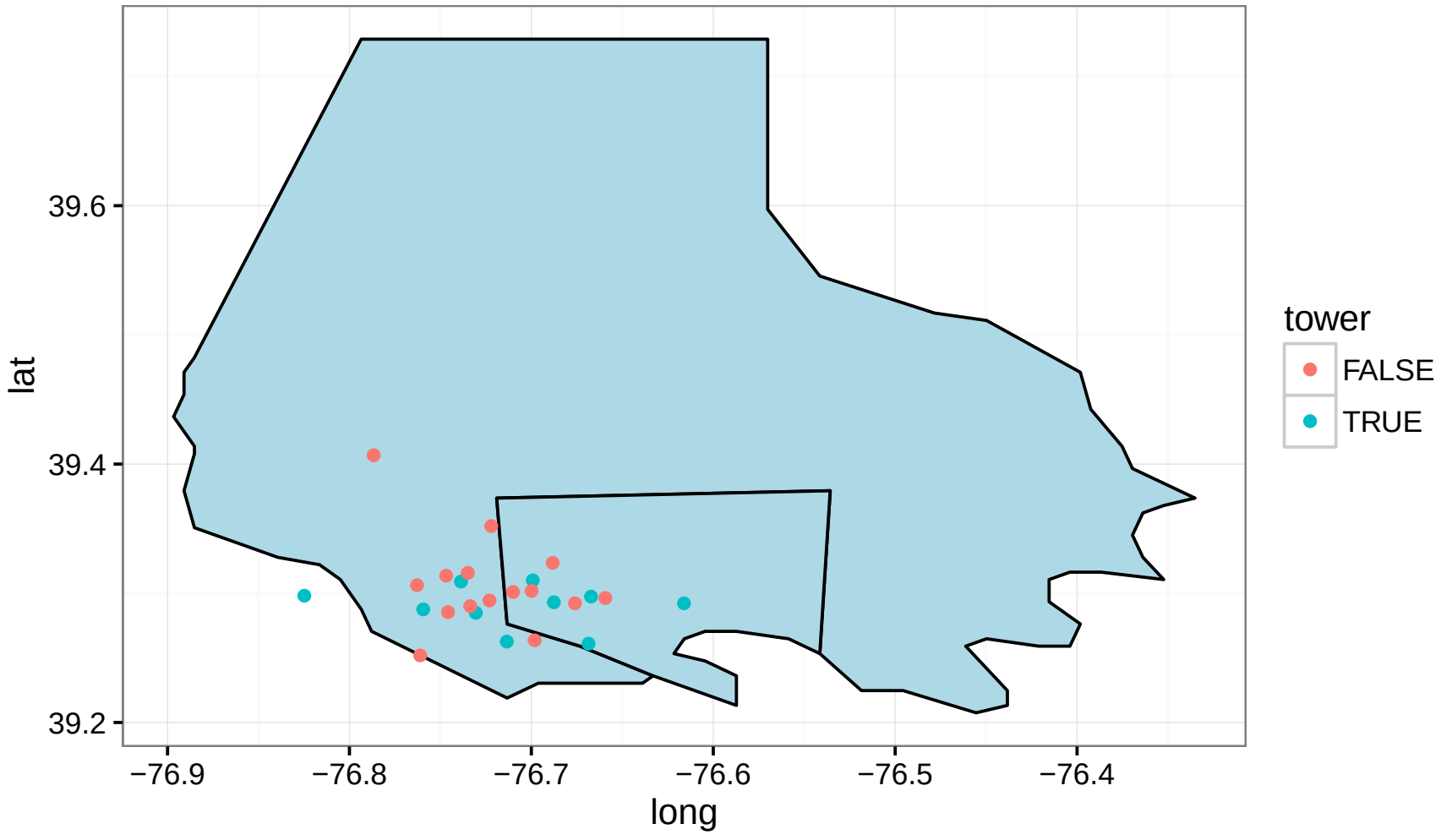
Point maps

I used `geom_point` to plot the points. When you plot points, you need to "ungroup" the `group` column you used to plot the polygons for counties. To do that, set `group = NA` in the `geom_point` statement.

```
balt_plot <- ggplot(baltimore,
                    aes(x = long, y = lat, group = group)) +
        geom_polygon(fill = "lightblue", color = "black") +
        geom_point(data = serial, aes(x = long, y = lat,
                                      group = NA,
                                      color = tower)) +
        theme_bw()
```

Point maps

```
balt_plot
```



Choropleths

Choropleths in R

There's a fantastic new(-ish) package in R to plot choropleth maps. You could also plot choropleths using `ggplot` and other mapping functions, but I would strongly recommend this new package if you're mapping the US.

You will need to install and load the `choroplethr` package in R to use the functions below.

```
# install.packages("choroplethr")
library(choroplethr)

# install.packages("choroplethrMaps")
library(choroplethrMaps)
```

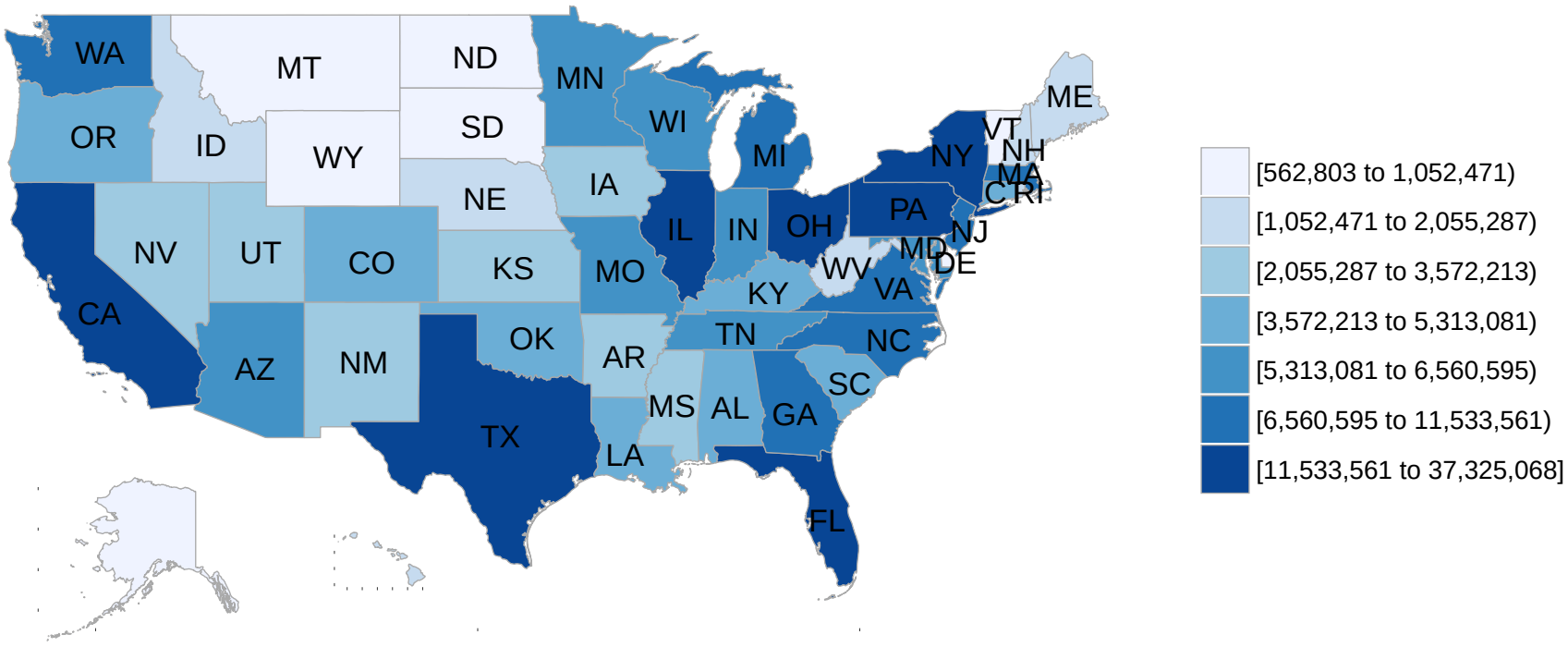
Choropleths in R

At the most basic level, you can use this package to plot some data that comes automatically with the package (you'll just need to load the data using the `data` function). For example, if you wanted to plot state-by-state populations as of 2012, you could use:

```
data(df_pop_state)
map_3 <- state_choropleth(df_pop_state)
```

Choropleths in R

```
map_3
```



Choropleths in R

You can find out more about the `df_pop_state` data if you type `?df_pop_state`. Notice that, for the data frame, the location is given in a column called `region` and the population size to plot is in a column called `value`.

```
head(df_pop_state, 3)
```

```
##    region   value
## 1 alabama 4777326
## 2  alaska  711139
## 3 arizona 6410979
```

Choropleths in R

You could use this function to create any state-level choropleth you wanted, as long as you could create a data frame with a column for states called `region` and a column with the value you want to show called `value`.

Choropleths in R

You can run similar functions at different spatial resolutions (for example, county or zip code):

```
data(df_pop_county)
head(df_pop_county, 3)
```

```
##   region  value
## 1   1001  54590
## 2   1003 183226
## 3   1005  27469
```

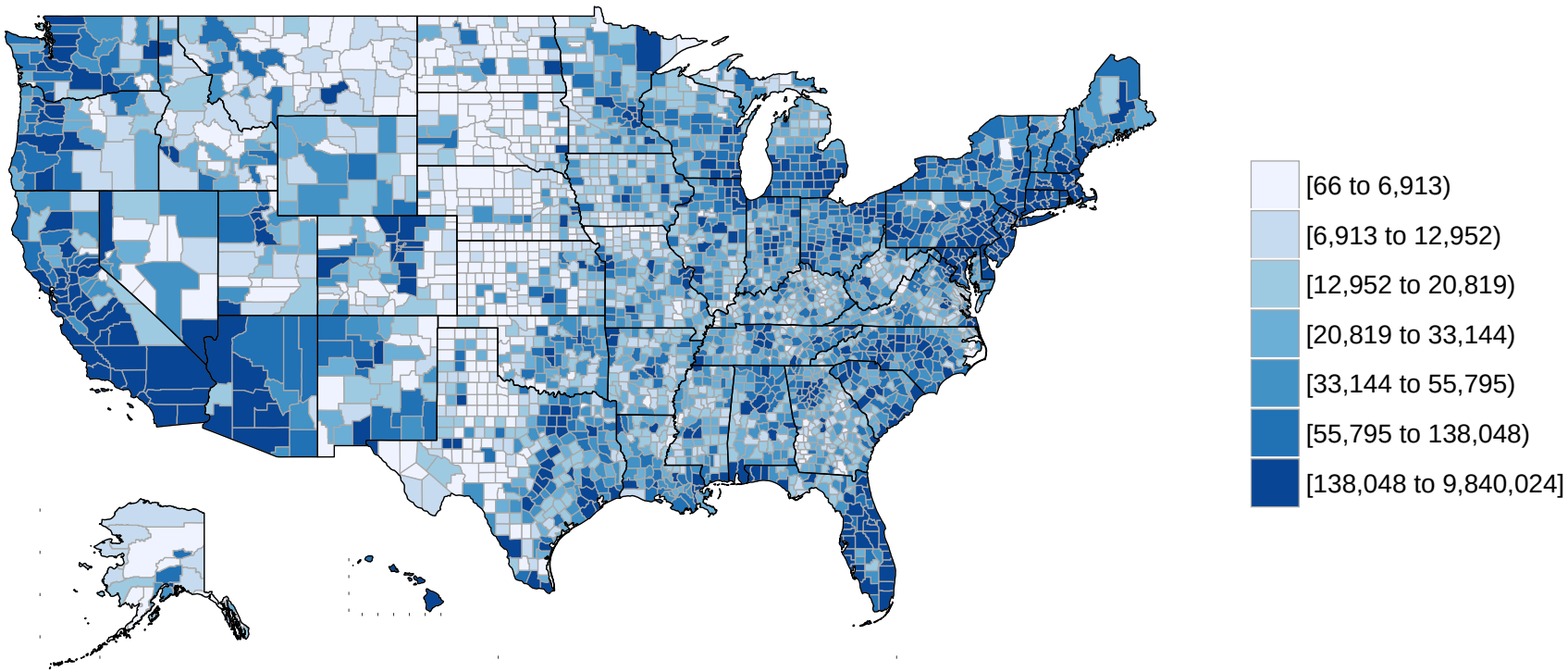
Choropleths in R

You can plot choropleths at this level, as well:

```
map_4 <- county_choropleth(df_pop_county)
```

Choropleths in R

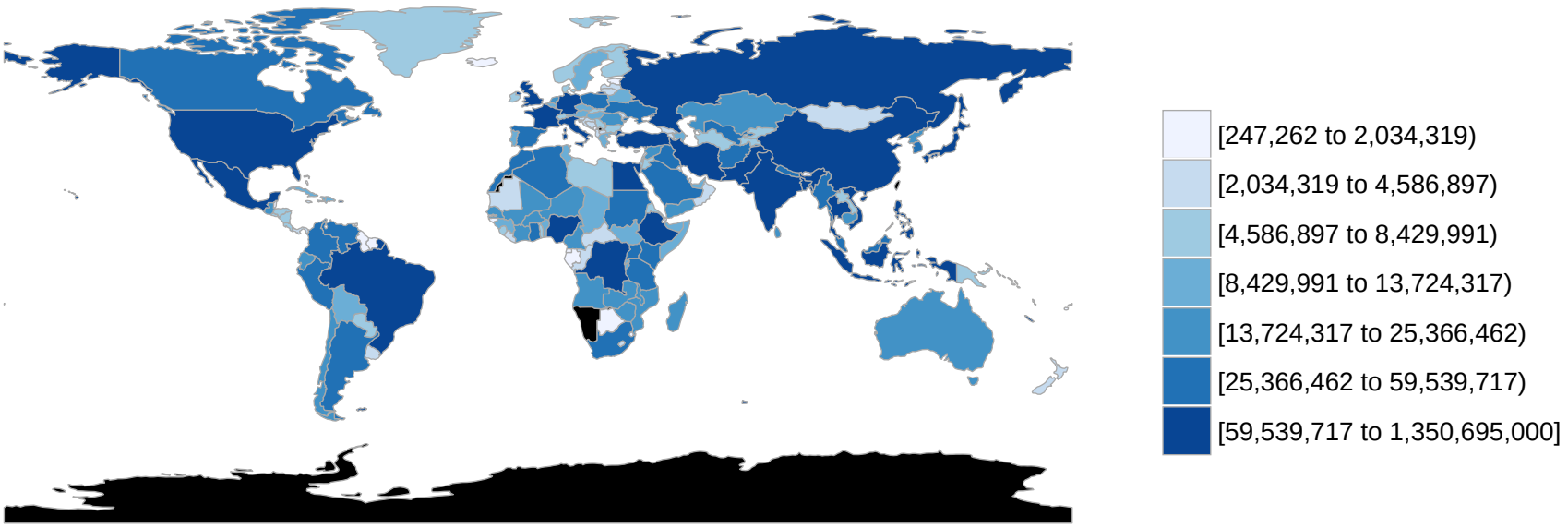
```
map_4
```



Choropleths in R

You can even do this for countries of the world:

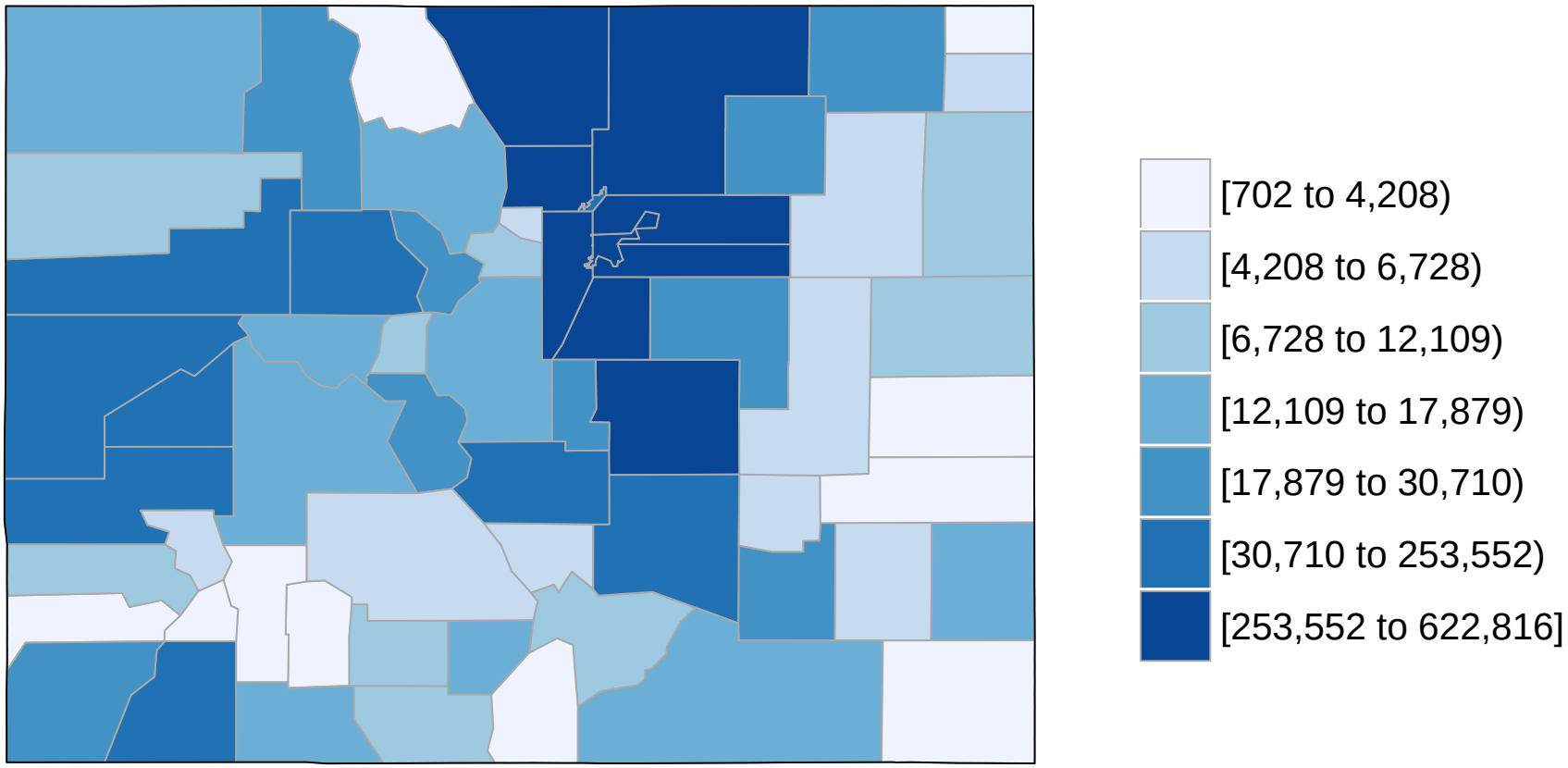
```
data(df_pop_country)
country_choropleth(df_pop_country)
```



Choropleths in R

You can zoom into states or counties. For example, to plot population by county in Colorado, you could run:

```
county_choropleth(df_pop_county, state_zoom = "colorado")
```



Choropleths in R

You can also use this package to map different tables from the US Census' American Community Survey.

The package includes the `choroplethr_acs()` function to do this, with an option for which level of map you want (`map =`, choices are "state", "country", and "zip"). If you want to map at the state level, for example, use `state_choroplethr_acs()` (other options are county level and zip code level).

Choropleths in R

These functions pull recent Census data directly from the US Census using its API, so they require you to get an API key, which you can get here.

Once you put in your request, they'll email you your key. Once they give you your API key, you'll need to install it on R:

```
library(acs)
api.key.install('[your census api key]');
```

Choropleths in R

You can pick from a large number of American Community Survey tables– see here for the list plus ID numbers. If the table has multiple columns, you will be prompted to select which one you want to plot.

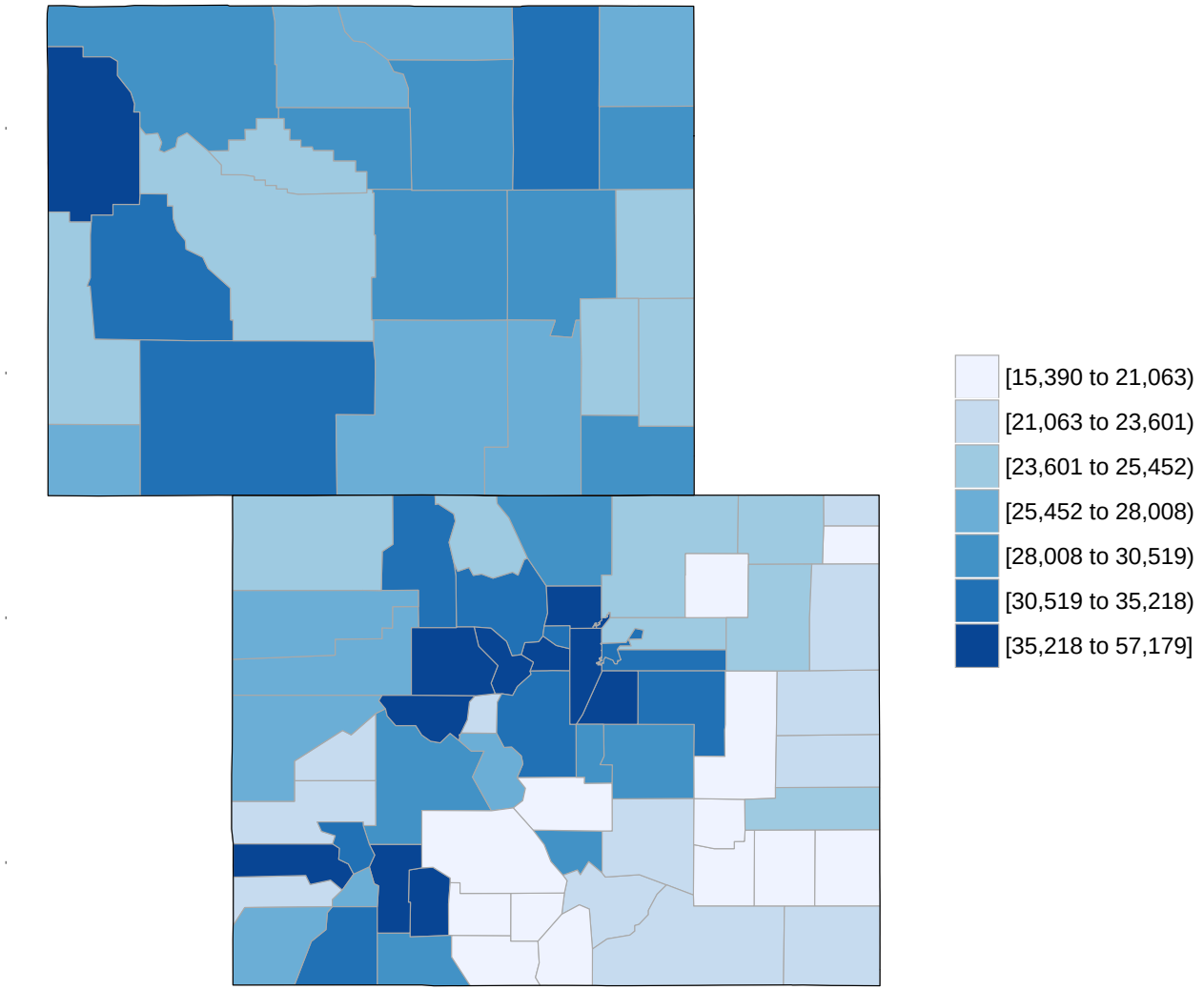
Choropleths in R

For example, table B19301 gives per-capita income, so if you wanted to plot that, you could run:

```
county_choropleth_acs(tableId = "B19301",
                      state_zoom = c("wyoming",
                                     "colorado"))
```

Choropleths in R

Per Capita Income: Per capita income in the past 12 months (in 2011 inflation−adjusted dollars)



Google Maps API

The ggmap package allows you to use tools from Google Maps directly from R.

```
## install.packages("ggmap")
library(ggmap)
```

This package uses the Google Maps API, so you should read their terms of service and make sure you follow them. In particular, you are limited to just a certain number of queries per time.

Google Maps API

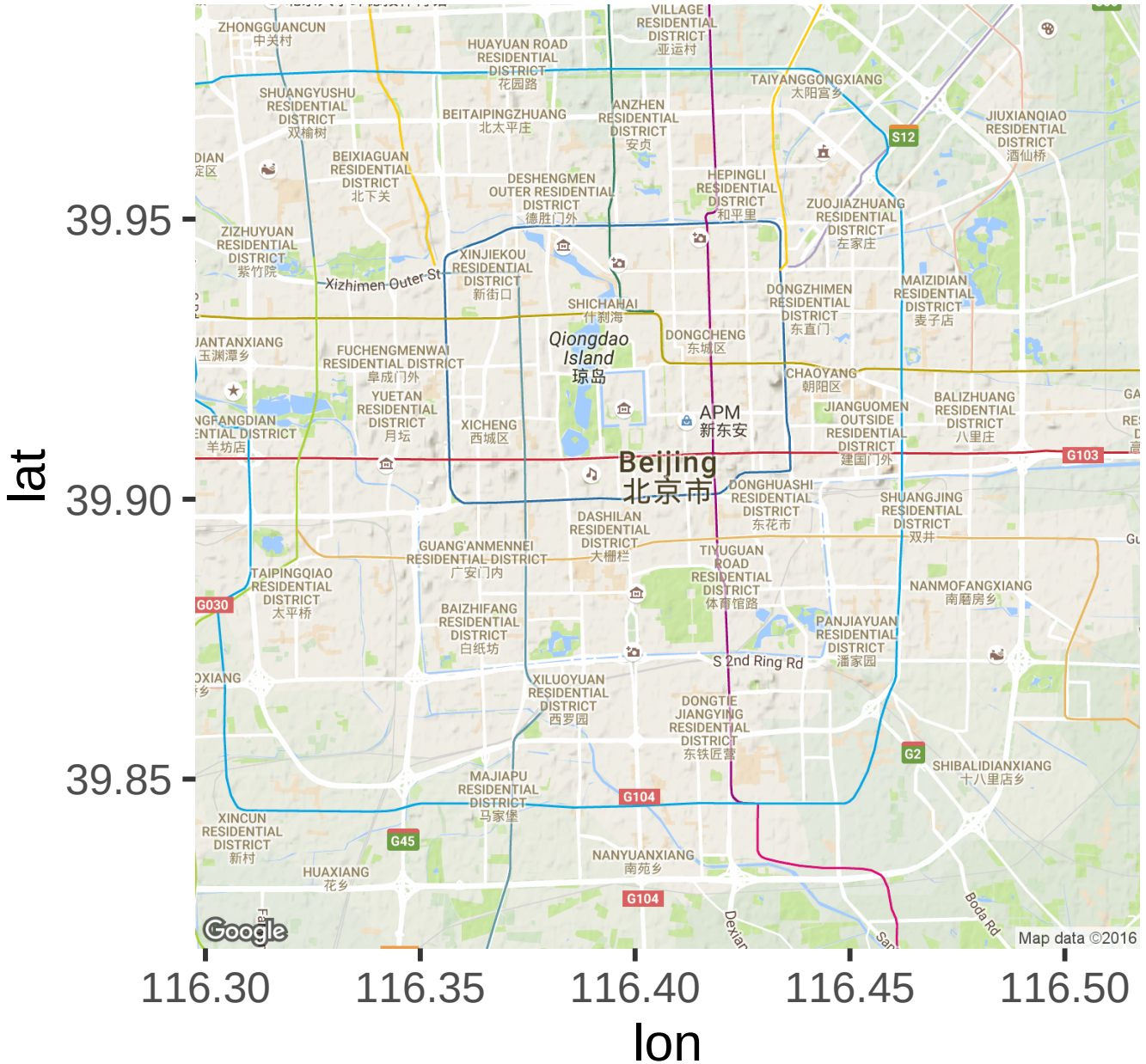
You can use the get_map function to get maps for different locations.

You can either use the longitude and latitude of the center point of the map, along with the zoom option to say how much to zoom in (3: continent to 20: building) or you can use a character string to specify a location.

If you do the second, get_map will actually use the Google Maps API to geocode the string to a latitude and longitude and then get the map (you can imagine that this is like searching in Google Maps in the search box for a location).

```
beijing <- get_map("Beijing", zoom = 12)
ggmap(beijing)
```

Google Maps API



Google Maps API

With this package, you can get maps from the following different sources:

- Google Maps
- OpenStreetMap
- Stamen Maps
- CloudMade Maps (You may need a separate API key for this)

Google Maps API

Here are different examples of Beijing using different map sources. (Also, note that I'm using the option `extent = "device"` to fill up the whole plot are with the map, instead of including axis labels and titles.)

```
beijing_a <- get_map("Beijing", zoom = 12,
                     source = "stamen", maptype = "toner")
a <- ggmap(beijing_a, extent = "device")

beijing_b <- get_map("Beijing", zoom = 12,
                     source = "stamen", maptype = "watercolor")
b <- ggmap(beijing_b, extent = "device")

beijing_c <- get_map("Beijing", zoom = 12,
                     source = "google", maptype = "hybrid")
c <- ggmap(beijing_c, extent = "device")
```

Google Maps API

```
grid.arrange(a, b, c, nrow = 1)
```

Google Maps API

As with the maps from ggplot2, you can add points to these maps:

```
serial_map <- get_map(c(-76.7, 39.3), zoom = 12,
                      source = "stamen",
                      maptype = "toner")
serial_map <- ggmap(serial_map, extent = "device") +
        geom_point(data = serial_phone,
                   aes(x = long, y = lat),
                   color = "red", size = 3,
                   alpha = 0.4) +
        geom_point(data = subset(serial,
                                 Type != "cell-site"),
                   aes(x = long, y = lat),
                   color = "darkgoldenrod1",
                   size = 2)
```

Google Maps API

Google Maps API

You can also use the Google Maps API, through the geocode function, to get the latitude and longitude of specific locations. Basically, if the string would give you the right location if you typed it in Google Maps, geocode should be able to geocode it.
For example, you can get the location of CSU:

```
geocode("Colorado State University")
```

```
##          lon      lat
## 1 -105.0807 40.57478
```

Google Maps API

You can also get a location by address through this:

```
geocode("1 First St NE, Washington, DC")
```

```
##         lon      lat
## 1 -77.00465 38.89051
```

Google Maps API

You can get distances, too, using the `mapdist` function with two locations. This will give you distance and also time.

```
mapdist("Fort Collins CO",
        "1 First St NE, Washington, DC") %>%
  select(from, miles, hours)
```

```
##                from    miles    hours
## 1 Fort Collins CO 1670.348 24.60167
```

String operations

String operations

The `str_trim` function from the `stringr` package allows you to trim leading and trailing whitespace:

```
with_spaces <- c("    a ", "  bob", " gamma")
with_spaces
```

```
## [1] "    a " "  bob"  " gamma"
```

```
str_trim(with_spaces)
```

```
## [1] "a"     "bob"   "gamma"
```

This is rarer, but if you ever want to, you can add leading and / or trailing whitespace to elements of a character vector with `str_pad` from the `stringr` package.

String operations

There are also functions to change a full character string to uppercase, lowercase, or title case:

```
titanic_train$Name[1]
```

```
## [1] "Braund, Mr. Owen Harris"
```

```
str_to_upper(titanic_train$Name[1])
```

```
## [1] "BRAUND, MR. OWEN HARRIS"
```

```
str_to_lower(titanic_train$Name[1])
```

```
## [1] "braund, mr. owen harris"
```

```
str_to_title(str_to_lower(titanic_train$Name[1]))
```

```
## [1] "Braund, Mr. Owen Harris"
```