Reporting data results #2

Functional programming

Applying a function repeatedly

One way that functions are really useful is that you can use the `map` family of functions from the `purrr` package to apply that function to all elements in a vector of a list (remember, a list could hold lots of similar dataframes).

For example, you could use `map` to apply the `add_one` function separately to 1, 2, and 3 by using `map` on a vector with those values:

```
library(purrr)
my_list <- list(a = 1:2, b = 3:5)
map(my_list, add_one)
```

```
## $a
## [1] 2 3
##
## $b
## [1] 4 5 6
```

Applying a function repeatedly

This can also be very useful if you have a dataframe for which you would like to apply the same function to subsets of the data. For example, for the `nepali` data you may want to apply the model of weight regressed on height and sex separately for children 12 months and younger versus older children.

First, we can add a factor variable that specifies whether the child is younger or older than 12 months:

```
nepali <- nepali %>%
  mutate(young = age < 12,
         young = factor(young, levels = c(TRUE, FALSE),
                        labels = c("younger", "older")))
```

Applying a function repeatedly

```
nepali %>%
  slice(1:3)
```

```
## # A tibble: 3 x 6
##       id    sex    wt    ht   age   young
##   <fctr> <fctr> <dbl> <dbl> <int>  <fctr>
## 1 120011   Male  12.8  91.2    41   older
## 2 120012 Female  14.9 103.9    57   older
## 3 120021 Female   7.7  70.1     8 younger
```

Applying a function repeatedly

Then, you can use the `nest` function to "nest" a dataframe by a factor variable. This function will create a column that actually stores its own dataframe:

```
nested_nepali <- nepali %>%
  group_by(young) %>%
  nest()
nested_nepali
```

```
## # A tibble: 2 x 2
##     young              data
##    <fctr>            <list>
## 1   older <tibble [162 x 5]>
## 2 younger  <tibble [38 x 5]>
```

Applying a function repeatedly

Each element of the `data` column in the nested dataset is actually a full dataframe:

```
nested_nepali$data[[1]] %>% slice(1:3)
```

```
## # A tibble: 3 x 5
##       id    sex    wt    ht   age
##   <fctr> <fctr> <dbl> <dbl> <int>
## 1 120011   Male  12.8  91.2    41
## 2 120012 Female  14.9 103.9    57
## 3 120022 Female  12.1  86.4    35
```

Applying a function repeatedly

Now, you can use `map` to apply the modeling function to each of these subsets of the dataframe. Use `mutate` to add a column with the results. We can also add a column with the results of applying `augment` to each of the model results:

```
modeled_nepali <- nepali %>%
  group_by(young) %>%
  nest() %>%
  mutate(mod_results = map(data, fit_ht_wt_mod),
         augmented_data = map(mod_results, augment))
modeled_nepali
```

```
## # A tibble: 2 x 4
##     young               data mod_results          augmented_data
##    <fctr>             <list>      <list>                  <list>
## 1   older <tibble [162 x 5]>    <S3: lm> <data.frame [151 x 11]>
## 2 younger  <tibble [38 x 5]>    <S3: lm>  <data.frame [34 x 11]>
```

Applying a function repeatedly

Each element of the `mod_results` column is the output from an `lm` model:

```
modeled_nepali$mod_results[[1]]
```

```
##
## Call:
## lm(formula = wt ~ ht + sex, data = df)
##
## Coefficients:
## (Intercept)           ht    sexFemale
##     -9.4638       0.2452      -0.3095
```

Applying a function repeatedly

We can apply `tidy` and `glance` to this output, just like we did with the output from fitting a single model:

```
tidy(modeled_nepali$mod_results[[1]])
```

```
##          term   estimate   std.error  statistic      p.value
## 1 (Intercept) -9.4637940 0.662450802 -14.286033 1.200446e-29
## 2          ht  0.2452427 0.007707689  31.817924 4.615549e-68
## 3   sexFemale -0.3095451 0.152599469  -2.028481 4.430458e-02
```

Applying a function repeatedly

To get back to a regular dataframe, you can "unnest". Before you do this, you should limit the data to only columns that will "unnest" to the same number of rows (if you have multiple columns with dataframes or lists in them). For example, we could unnest the `augmented_data` column, so we can plot observed data versus the model fit:

```
modeled_nepali %>%
  select(young, augmented_data) %>%
  unnest() %>%
  slice(1:3) %>% select(1:7)
```

```
## # A tibble: 3 x 7
##    young .rownames    wt    ht    sex  .fitted   .se.fit
##   <fctr>     <chr> <dbl> <dbl> <fctr>    <dbl>     <dbl>
## 1  older         1  12.8  91.2   Male 12.90234 0.1131341
## 2  older         2  14.9 103.9 Female 15.70737 0.1916600
## 3  older         3  12.1  86.4 Female 11.41563 0.1147084
```

Applying a function repeatedly

```
modeled_nepali %>%
  select(young, augmented_data) %>%
  unnest() %>%
  ggplot(aes(x = ht, y = wt, group = sex, color = sex)) +
  geom_point(size = 0.8) +
  geom_line(aes(y = .fitted), size = 1.2) +
  facet_wrap(~ young, ncol = 2)
```

Applying a function repeatedly

The full pipe for creating this figure is:

```
nepali %>%
  group_by(young) %>%     # Group by young and nest
  nest() %>%              # to model each group of "young"
  mutate(mod_results = map(data, fit_ht_wt_mod),
         augmented_data = map(mod_results, augment)) %>%
  select(young, augmented_data) %>%
  unnest %>%              # Unnest `augmented_data` (nested)
  ggplot(aes(x = ht, y = wt, group = sex, color = sex)) +
  geom_point(size = 0.8) +
  geom_line(aes(y = .fitted), size = 1.2) +
  facet_wrap(~ young, ncol = 2)
```

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

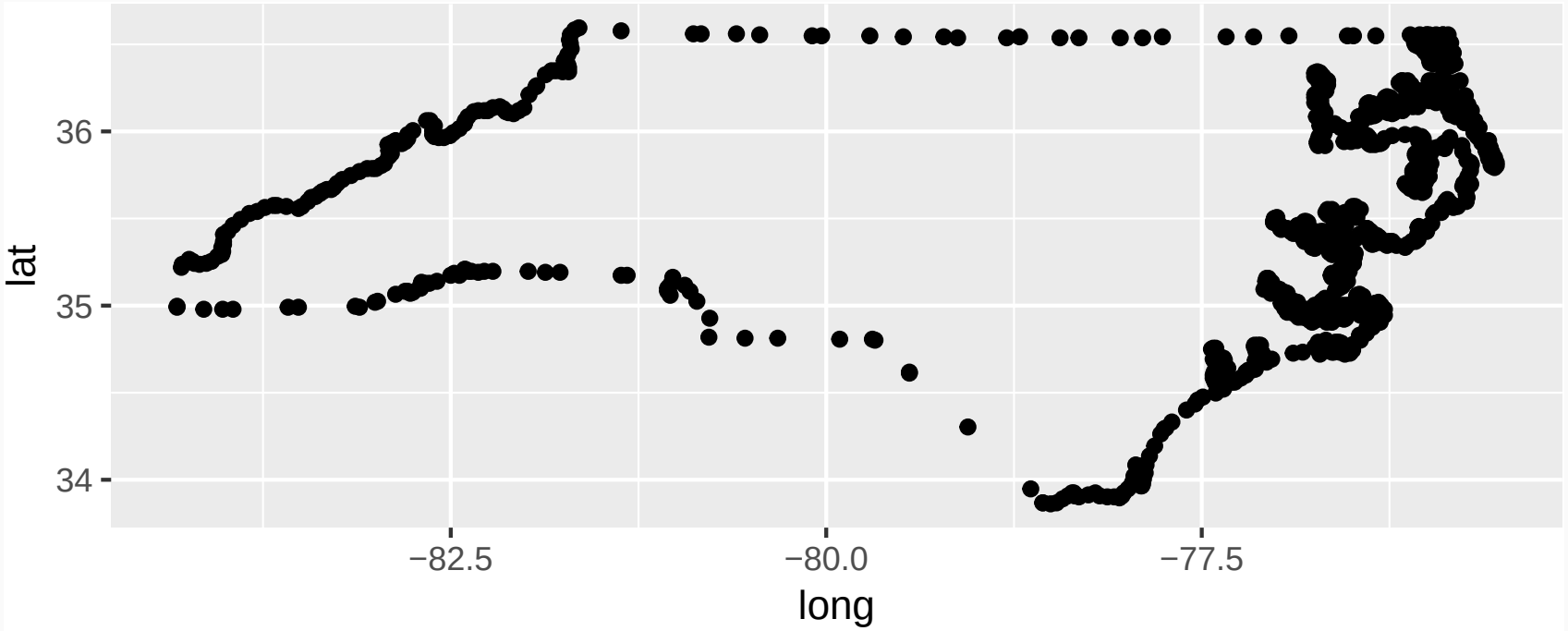
Map choices with `map_data` are currently limited to:

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

Point maps

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

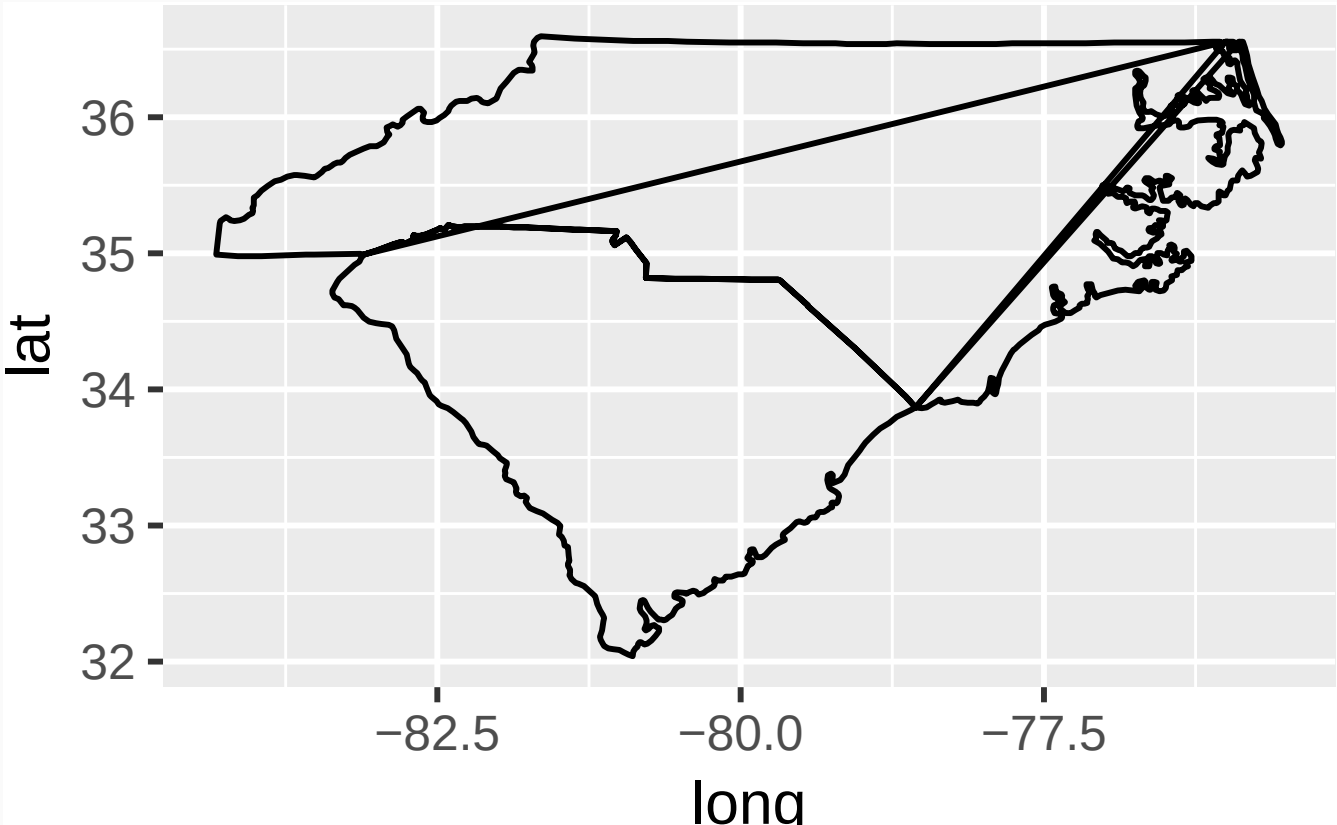
```
north_carolina <- us_map %>%
  filter(region == "north carolina")
ggplot(north_carolina, aes(x = long, y = lat)) +
  geom_point()
```



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)
```

```
## # A tibble: 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

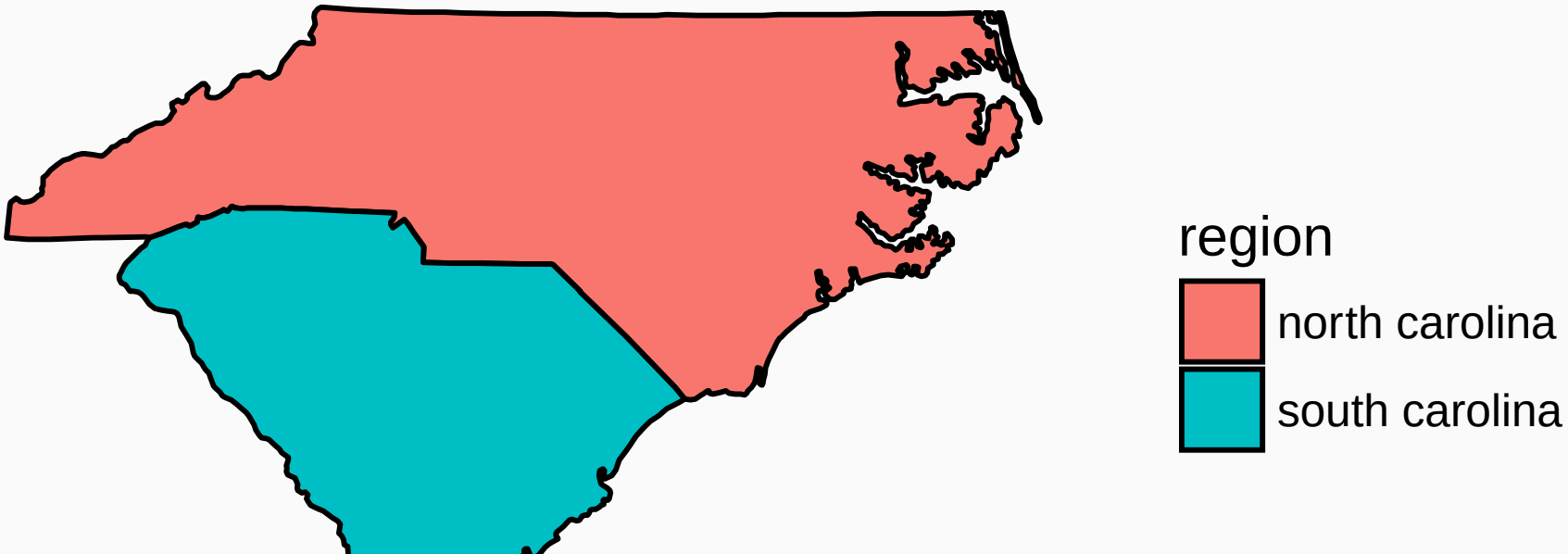
Using `group = group` avoids the extra lines from the earlier map:

```
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()
```



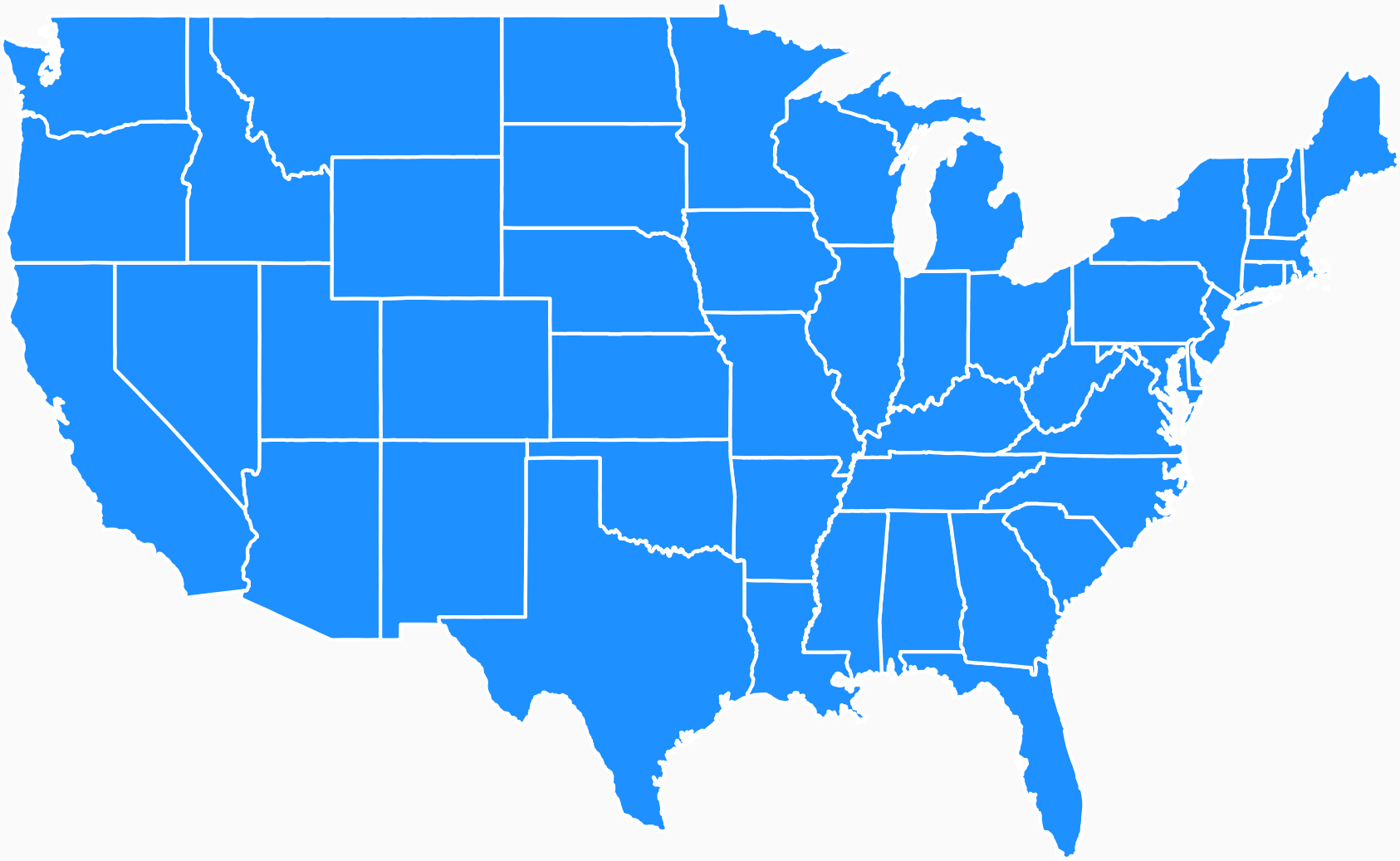
Point maps

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
```

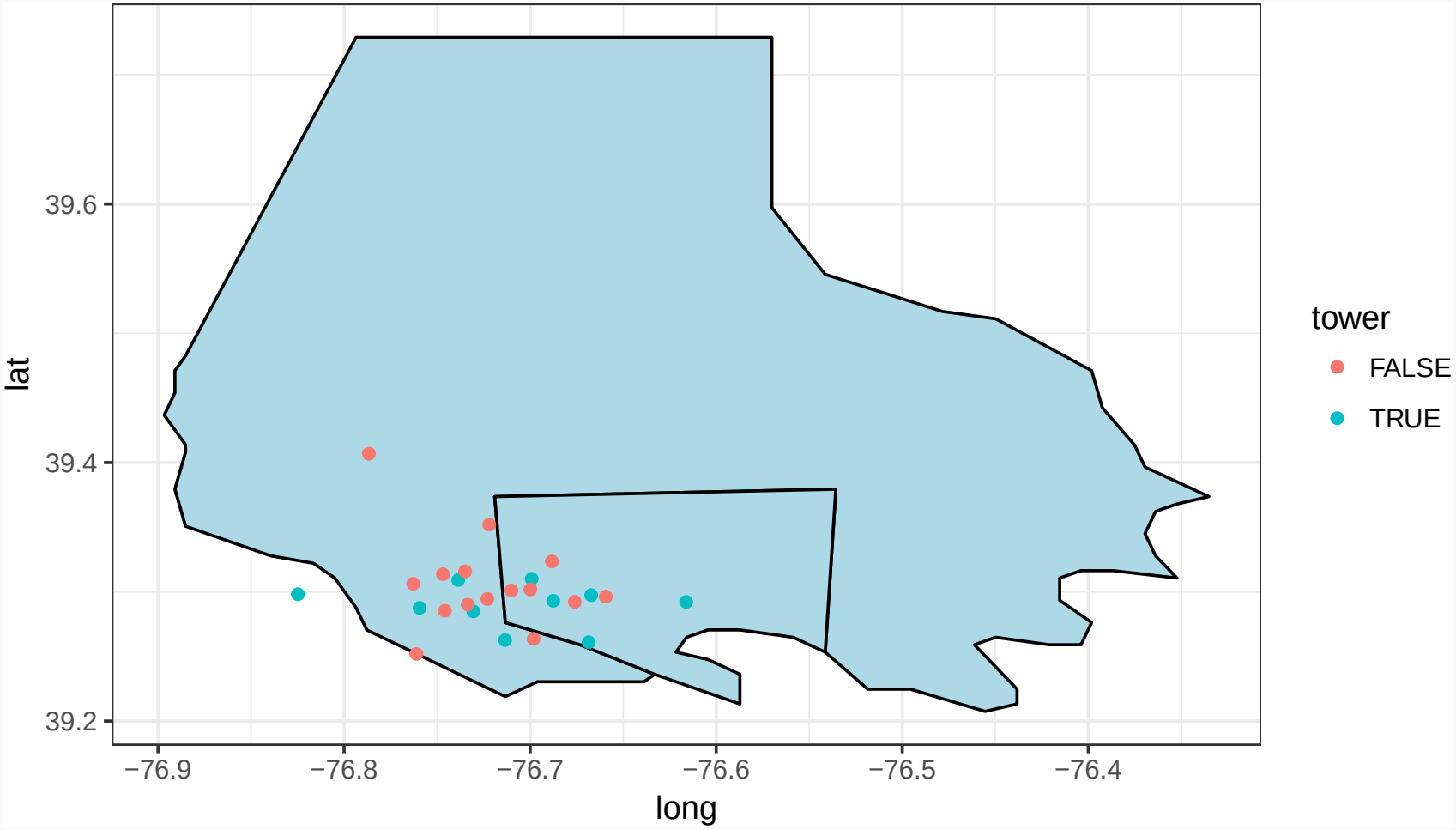
Point maps

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)
```

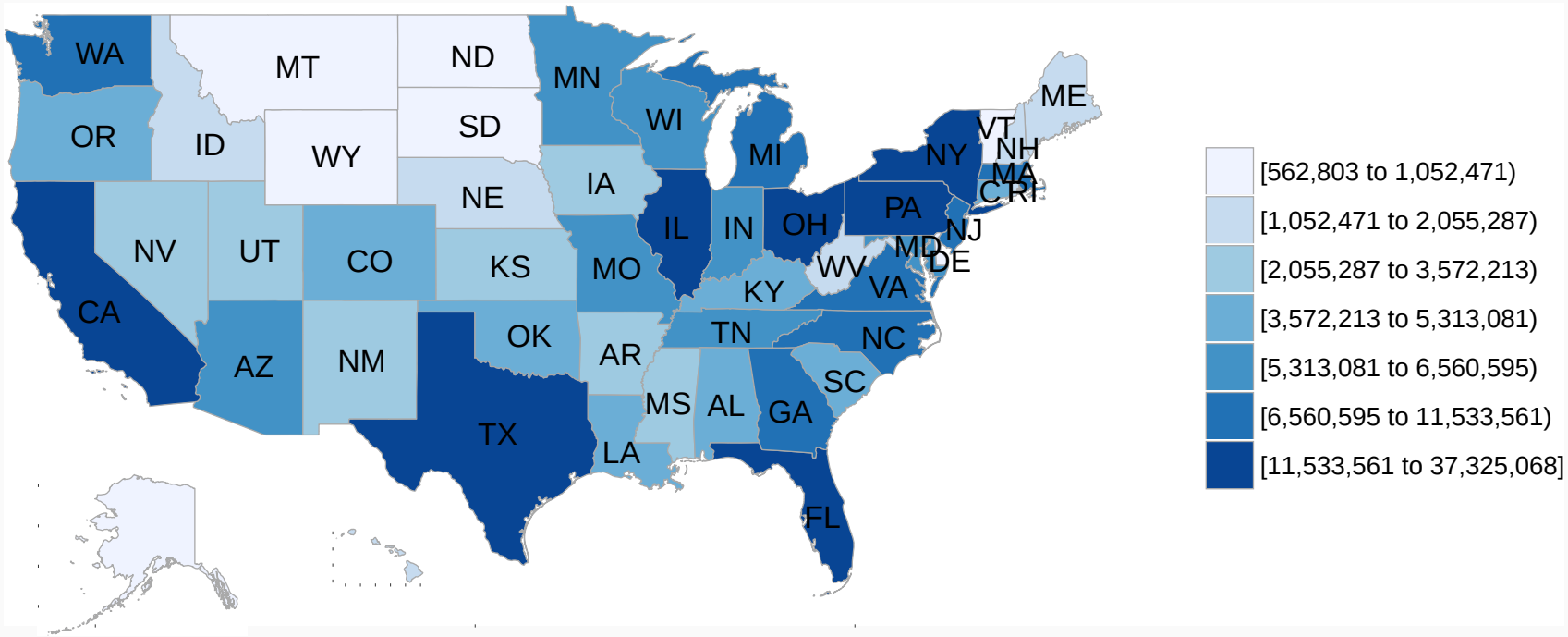
Choropleths in R

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
```

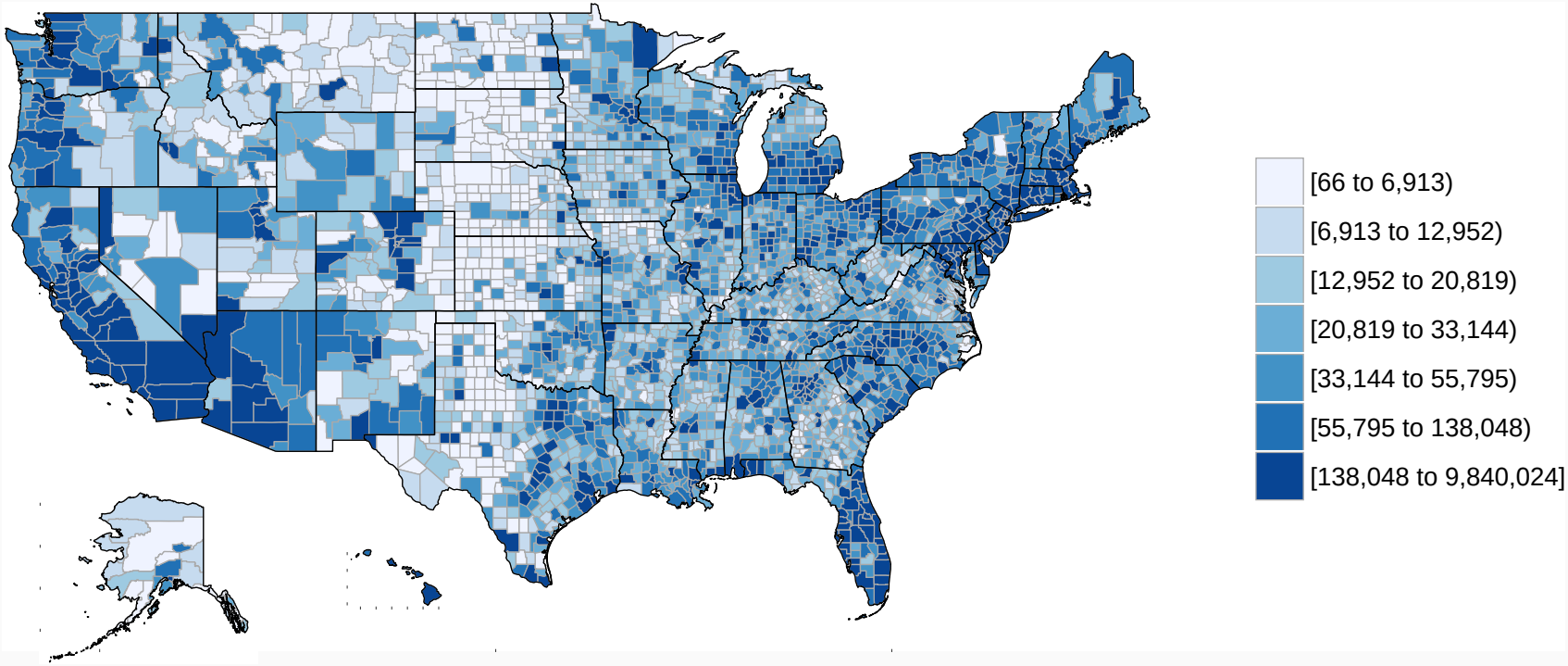
Choropleths in R

You can plot choropleths at this level, as well:

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

Choropleths in R

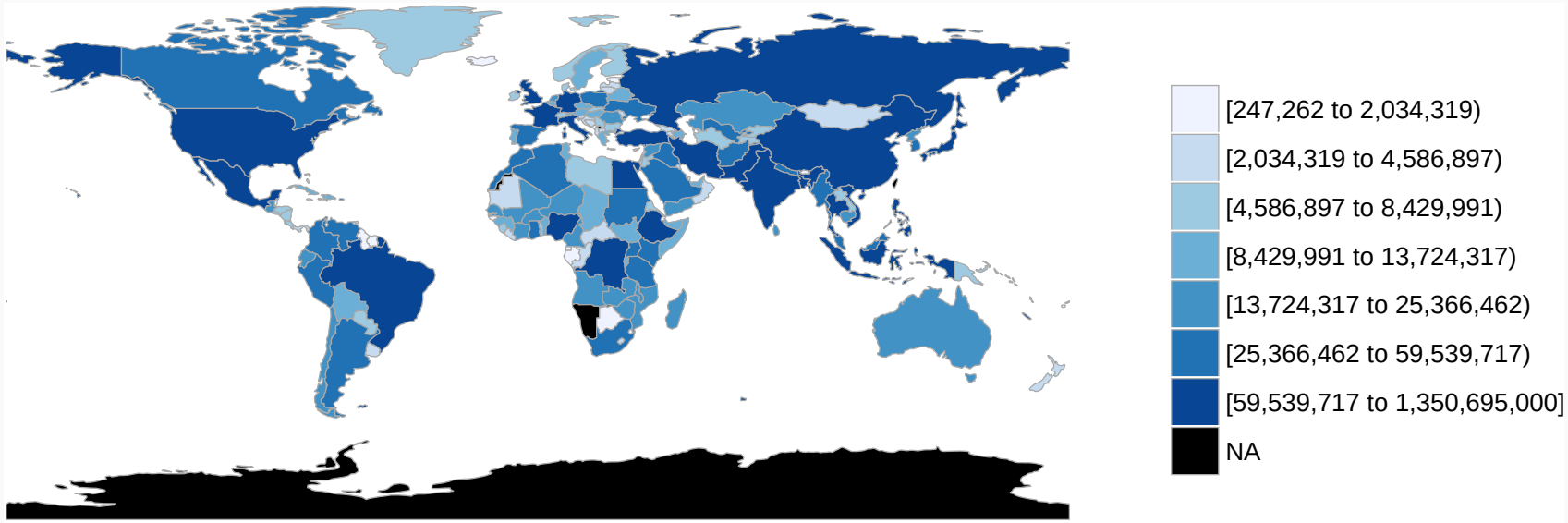
```
map_4
```



Choropleths in R

You can even do this for countries of the world:

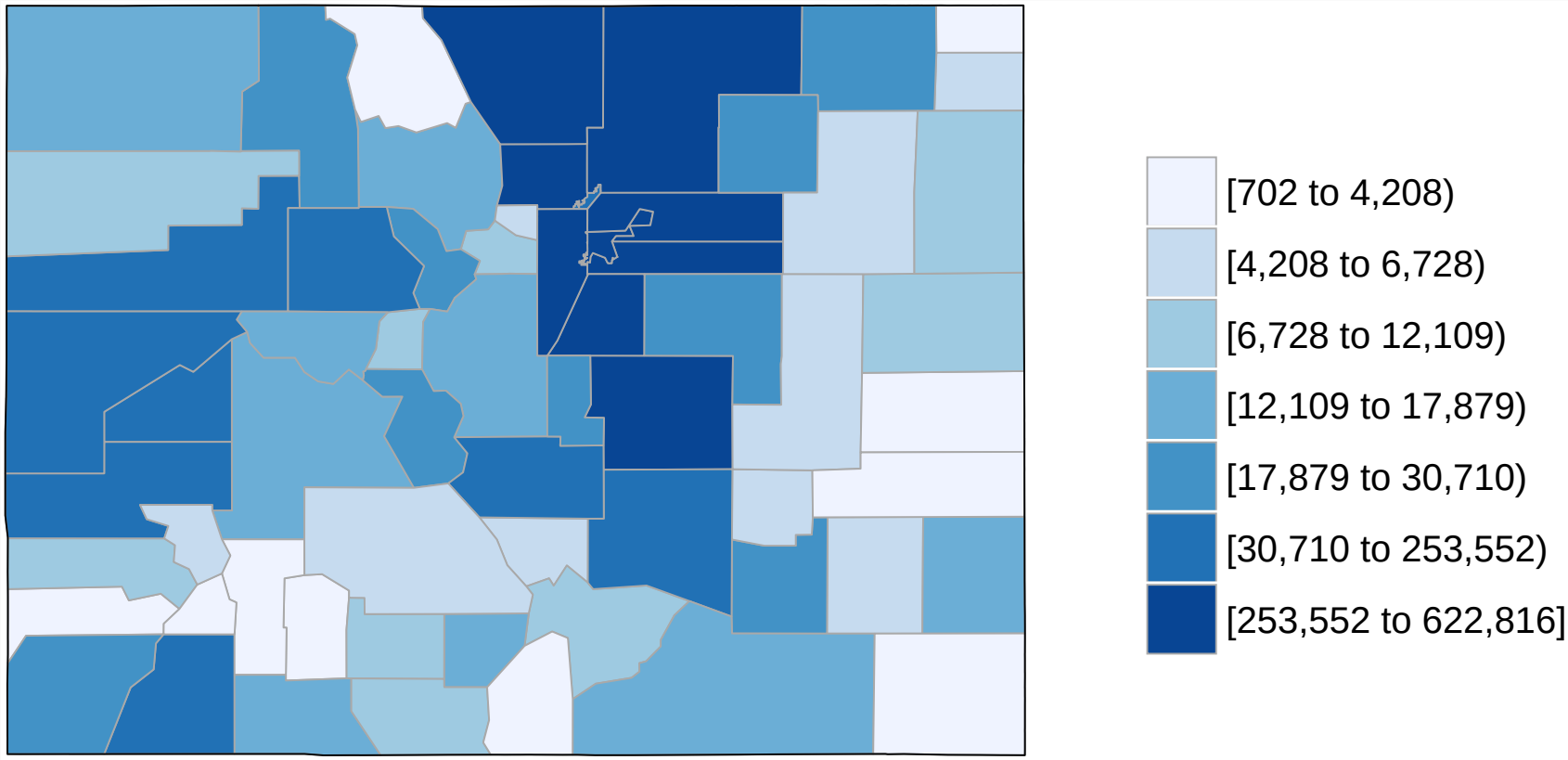
```
data(df_pop_country)
country_choropleth(df_pop_country)
```



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.

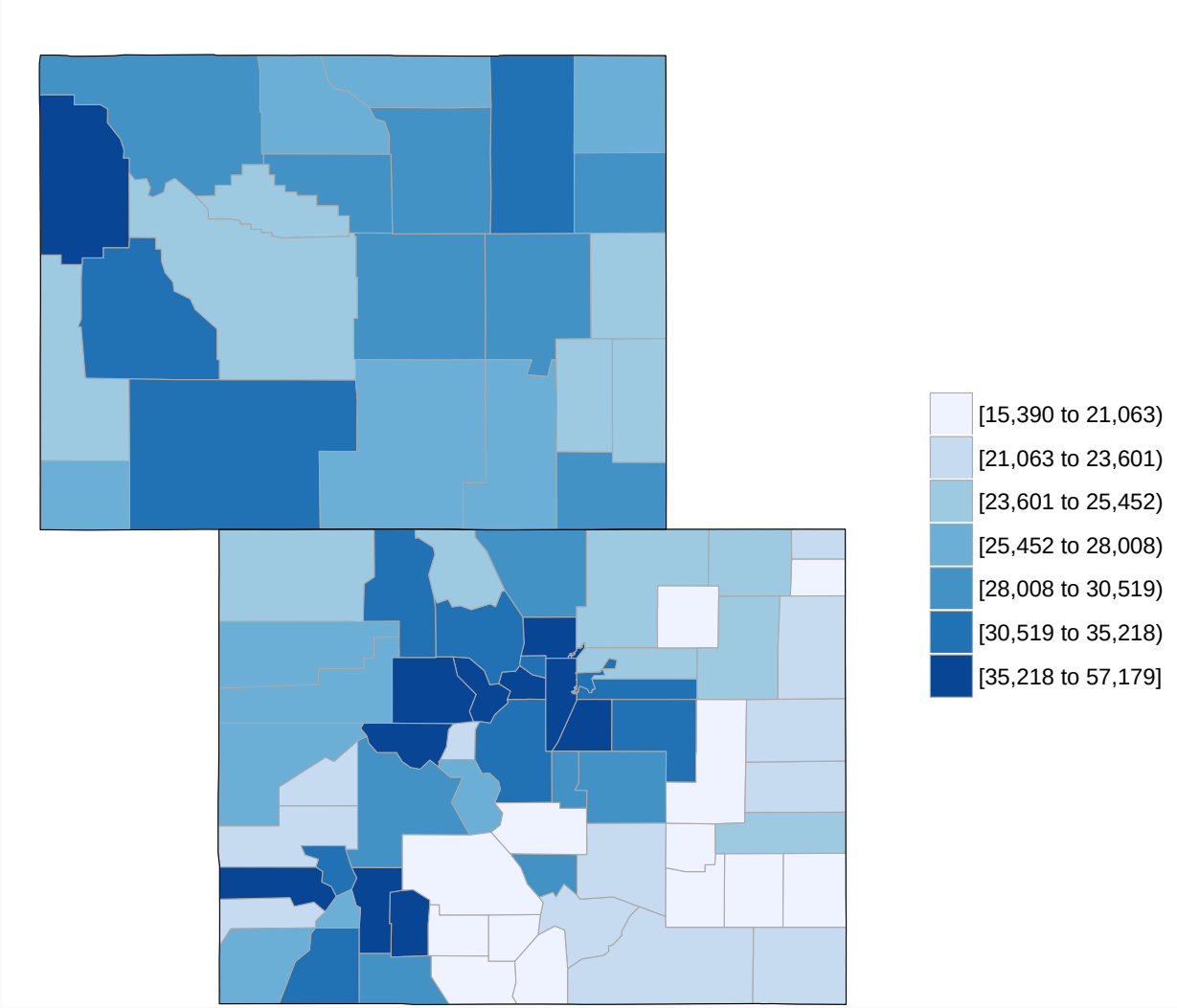
Choropleths in R

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

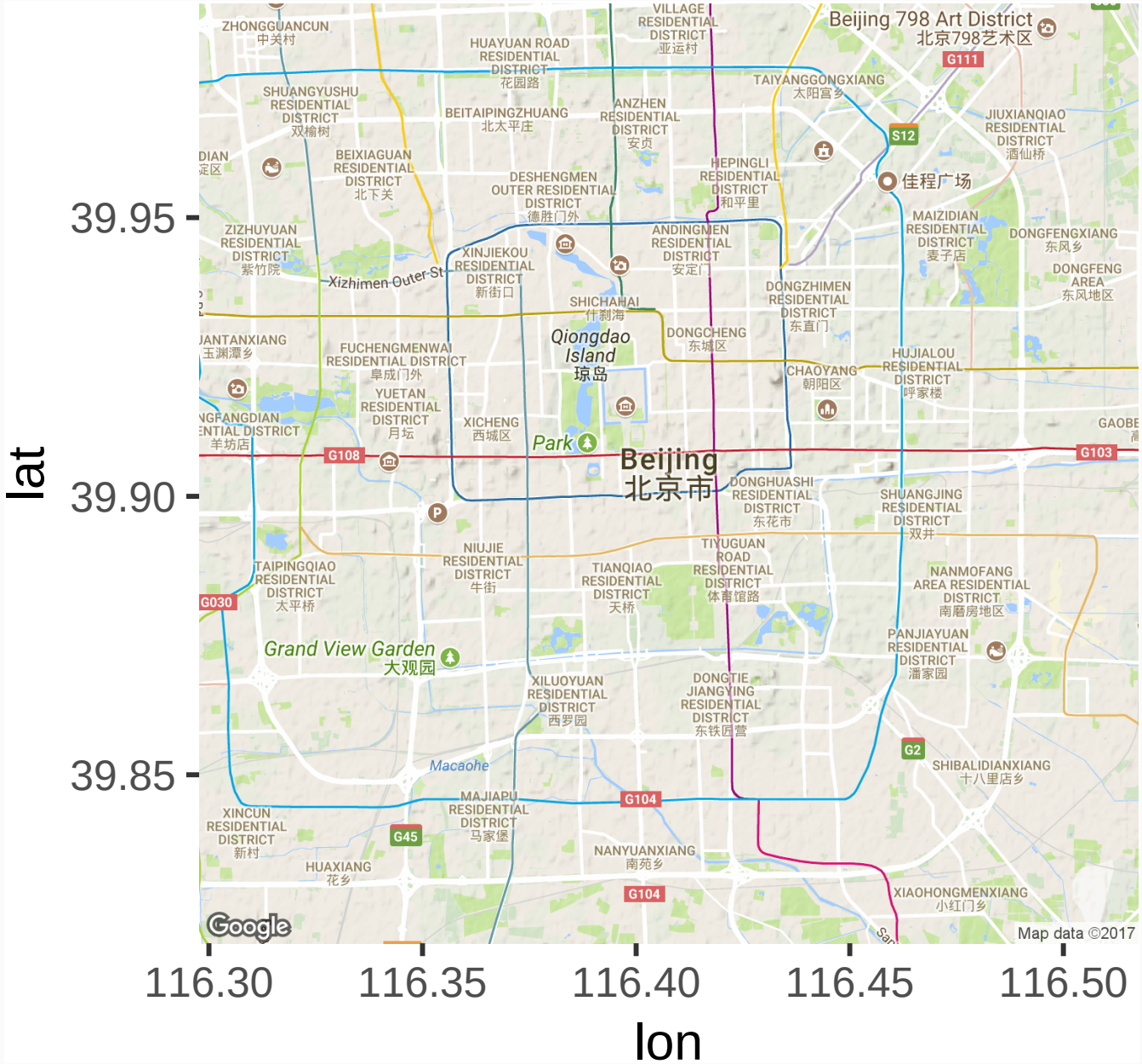
You can use the `get_map` function to get maps for different locations.

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.)

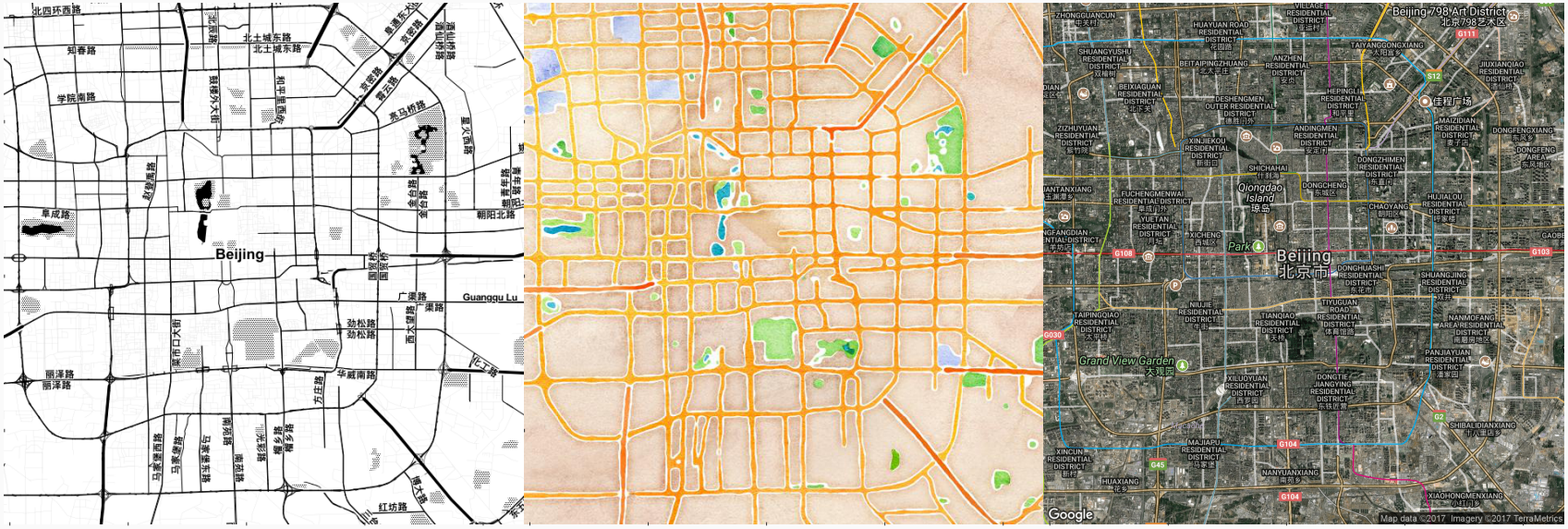
```
beijing_a <- get_map("Beijing", zoom = 12,
                     source = "stamen", maptype = "toner")
a <- ggmap(beijing_a, extent = "device")

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 -105.0865 40.57344
```

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 1670.784 24.68139
```

String operations

String operations

For these examples, we'll use some data on passengers of the Titanic. You can load this data using:

```
# install.packages("titanic")
library(titanic)
data("titanic_train")
```

We will be using the `stringr` package:

```
library(stringr)
```

String operations

This data includes a column called "Name" with passenger names. This column is somewhat messy and includes several elements that we might want to separate (last name, first name, title). Here are the first few values of "Name":

```
titanic_train %>% select(Name) %>% slice(1:3)
```

```
## # A tibble: 3 x 1
##                                                  Name
##                                                 <chr>
## 1                              Braund, Mr. Owen Harris
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer)
## 3                               Heikkinen, Miss. Laina
```

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"
```

Regular expressions

Regular expressions

We've already done some things to manipulate strings. For example, if we wanted to separate "Name" into last name and first name (including title), we could actually do that with the `separate` function:

```
titanic_train %>%
  select(Name) %>%
  slice(1:3) %>%
  separate(Name, c("last_name", "first_name"), sep = ", ")
```

```
## # A tibble: 3 x 2
##   last_name                                first_name
## *     <chr>                                    <chr>
## 1    Braund                              Mr. Owen Harris
## 2   Cumings Mrs. John Bradley (Florence Briggs Thayer)
## 3 Heikkinen                                 Miss. Laina
```

Regular expressions

Notice that `separate` is looking for a regular pattern (“,”) and then doing something based on the location of that pattern in each string (splitting the string).

There are a variety of functions in R that can perform manipulations based on finding regular patterns in character strings.

Regular expressions

The `str_detect` function will look through each element of a character vector for a designated pattern. If the pattern is there, it will return `TRUE`, and otherwise `FALSE`. The convention is:

```
## Generic code
str_detect(string = [vector you want to check],
           pattern = [pattern you want to check for])
```

For example, to create a logical vector specifying which of the Titanic passenger names include "Mrs.", you can call:

```
mrs <- str_detect(titanic_train$Name, "Mrs\\.")
head(mrs)
```

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

Regular expressions

The result is a logical vector, so `str_detect` can be used in `filter` to subset data to only rows where the passenger's name includes "Mrs.":

```
titanic_train %>%
  filter(str_detect(Name, "Mrs\\.")) %>%
  select(Name) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 1
##                                                    Name
##                                                   <chr>
## 1 Cumings, Mrs. John Bradley (Florence Briggs Thayer)
## 2        Futrelle, Mrs. Jacques Heath (Lily May Peel)
## 3   Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)
```

Regular expressions

As a note, in regular expressions, all of the following characters are special characters that need to be escaped with backslashes if you want to use them literally:

```
. * + ^ ? $ \ | ( ) [ ] { }
```

Regular expressions

There is an older, base R function called `grepl` that does something very similar (although note that the order of the arguments is reversed).

```
titanic_train %>%
  filter(grepl("Mrs\\.", Name)) %>%
  select(Name) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 1
##                                                  Name
##                                                 <chr>
## 1 Cumings, Mrs. John Bradley (Florence Briggs Thayer)
## 2        Futrelle, Mrs. Jacques Heath (Lily May Peel)
## 3   Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)
```

Regular expressions

The `str_extract` function can be used to extract a string (if it exists) from each value in a character vector. It follows similar conventions to `str_detect`:

```
## Generic code
str_extract(string = [vector you want to check],
            pattern = [pattern you want to check for])
```

Regular expressions

For example, you might want to extract "Mrs." if it exists in a passenger's name:

```
titanic_train %>%
  mutate(mrs = str_extract(Name, "Mrs\\.")) %>%
  select(Name, mrs) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 2
##                                                  Name   mrs
##                                                 <chr> <chr>
## 1                             Braund, Mr. Owen Harris  <NA>
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer)  Mrs.
## 3                              Heikkinen, Miss. Laina  <NA>
```

Notice that now we're creating a new column (`mrs`) that either has "Mrs." (if there's a match) or is missing (`NA`) if there's not a match.

Regular expressions

For this first example, we were looking for an exact string ("Mrs"). However, you can use patterns that match a particular pattern, but not an exact string. For example, we could expand the regular expression to find "Mr." or "Mrs.":

```
titanic_train %>%
  mutate(title = str_extract(Name, "Mr\\.|Mrs\\.")) %>%
  select(Name, title) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 2
##                                                  Name title
##                                                 <chr> <chr>
## 1                             Braund, Mr. Owen Harris   Mr.
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer)  Mrs.
## 3                              Heikkinen, Miss. Laina  <NA>
```

Note that this pattern uses a special operator (|) to find one pattern **or** another. Double backslashs (\\) **escape** the special character ".".

Regular expressions

Notice that "Mr." and "Mrs." both start with "Mr", end with ".", and may or may not have an "s" in between.

```
titanic_train %>%
  mutate(title = str_extract(Name, "Mr(s)*\\.")) %>%
  select(Name, title) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 2
##                                                 Name title
##                                                <chr> <chr>
## 1                             Braund, Mr. Owen Harris   Mr.
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer)  Mrs.
## 3                              Heikkinen, Miss. Laina  <NA>
```

This pattern uses `(s)*` to match zero or more "s"s at this spot in the pattern.

Regular expressions

In the previous code, we found "Mr." and "Mrs.", but missed "Miss.". We could tweak the pattern again to try to capture that, as well. For all three, we have the pattern that it starts with "M", has some lowercase letters, and then ends with ".".

```
titanic_train %>%
  mutate(title = str_extract(Name, "M[a-z]+\\.")) %>%
  select(Name, title) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 2
##                                                  Name title
##                                                 <chr> <chr>
## 1                             Braund, Mr. Owen Harris   Mr.
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer)  Mrs.
## 3                              Heikkinen, Miss. Laina Miss.
```

Regular expressions

The last pattern used `[a-z]+` to match one or more lowercase letters. The `[a-z]`is a **character class**.

You can also match digits (`[0-9]`), uppercase letters (`[A-Z]`), just some letters (`[aeiou]`), etc.

You can negate a character class by starting it with `^`. For example, `[^0-9]` will match anything that **isn't** a digit.

Regular expressions

Sometimes, you want to match a pattern, but then only subset a part of it. For example, each passenger seems to have a title ("Mr.", "Mrs.", etc.) that comes after "," and before ".". We can use this pattern to find the title, but then we get some extra stuff with the match:

```
titanic_train %>%
  mutate(title = str_extract(Name, ",\\s[A-Za-z]*\\.\\s")) %>%
  select(title) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 1
##      title
##      <chr>
## 1   , Mr.
## 2  , Mrs.
## 3 , Miss.
```

As a note, in this pattern, \\s is used to match a space.

Regular expressions

We are getting things like ", Mr.", when we really want "Mr". We can use the `str_match` function to do this. We group what we want to extract from the pattern in parentheses, and then the function returns a matrix. The first column is the full pattern match, and each following column gives just what matches within the groups.

```
head(str_match(titanic_train$Name,
               pattern = ",\\s([A-Za-z]*)\\.\\s"))
```

```
##      [,1]       [,2]
## [1,] ", Mr. "   "Mr"
## [2,] ", Mrs. "  "Mrs"
## [3,] ", Miss. " "Miss"
## [4,] ", Mrs. "  "Mrs"
## [5,] ", Mr. "   "Mr"
## [6,] ", Mr. "   "Mr"
```

Regular expressions

To get just the title, then, we can run:

```
titanic_train %>%
  mutate(title =
           str_match(Name, ",\\s([A-Za-z]*)\\.\\s")[ , 2]) %>%
  select(Name, title) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 2
##                                                  Name title
##                                                 <chr> <chr>
## 1                              Braund, Mr. Owen Harris    Mr
## 2 Cumings, Mrs. John Bradley (Florence Briggs Thayer)   Mrs
## 3                               Heikkinen, Miss. Laina  Miss
```

The `[ , 2]` pulls out just the second column from the matrix returned by `str_match`.

Regular expressions

Here are some of the most common titles:

```
titanic_train %>%
  mutate(title =
           str_match(Name, ",\\s([A-Za-z]*)\\.\\s")[ , 2]) %>%
  group_by(title) %>% summarize(n = n()) %>%
  arrange(desc(n)) %>% slice(1:5)
```

```
## # A tibble: 5 x 2
##    title     n
##    <chr> <int>
## 1     Mr   517
## 2   Miss   182
## 3    Mrs   125
## 4 Master    40
## 5     Dr     7
```

Regular expressions

The following slides have a few other examples of regular expressions in action with this dataset.

Get just names that start with ("^") the letter "A":

```
titanic_train %>%
  filter(str_detect(Name, "^A")) %>%
  select(Name) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 1
##                                                  Name
##                                                 <chr>
## 1                                Allen, Mr. William Henry
## 2                               Andersson, Mr. Anders Johan
## 3 Asplund, Mrs. Carl Oscar (Selma Augusta Emilia Johansson)
```

Regular expressions

Get names with "II" or "III" ({2,} says to match at least two times):

```
titanic_train %>%
  filter(str_detect(Name, "I{2,}")) %>%
  select(Name) %>%
  slice(1:3)
```

```
## # A tibble: 2 x 1
##                                      Name
##                                     <chr>
## 1  Carter, Master. William Thornton II
## 2 Roebling, Mr. Washington Augustus II
```

Regular expressions

Get names with "Andersen" or "Anderson" (alternatives in square brackets):

```
titanic_train %>%
  filter(str_detect(Name, "Anders[eo]n")) %>%
  select(Name)
```

```
##                                                  Name
## 1 Andersen-Jensen, Miss. Carla Christine Nielsine
## 2                            Anderson, Mr. Harry
## 3                      Walker, Mr. William Anderson
## 4                       Olsvigen, Mr. Thor Anderson
## 5       Soholt, Mr. Peter Andreas Lauritz Andersen
```

Regular expressions

Get names that start with ("^" outside of brackets) the letters "A" and "B":

```
titanic_train %>%
  filter(str_detect(Name, "^[AB]")) %>%
  select(Name) %>%
  slice(1:3)
```

```
## # A tibble: 3 x 1
##                         Name
##                        <chr>
## 1  Braund, Mr. Owen Harris
## 2 Allen, Mr. William Henry
## 3 Bonnell, Miss. Elizabeth
```

Regular expressions

Get names that end with ("$") the letter "b" (either lowercase or uppercase):

```
titanic_train %>%
  filter(str_detect(Name, "[bB]$")) %>%
  select(Name)
```

```
##                          Name
## 1   Emir, Mr. Farred Chehab
## 2 Goldschmidt, Mr. George B
## 3           Cook, Mr. Jacob
## 4          Pasic, Mr. Jakob
```

Regular expression

Some useful regular expression operators include:

Operator	Meaning
.	Any character
	Match 0 or more times (greedy)
?	Match 0 or more times (non-greedy)
+	Match 1 or more times (greedy)
+?	Match 1 or more times (non-greedy)
^	Starts with (in brackets, negates)
$	Ends with
[...]	Character classes

Regular expressions

For more on these patterns, see:

- Help file for the `stringi-search-regex` function in the `stringi` package (which should install when you install `stringr`)
- Chapter 14 of R For Data Science
- `http://gskinner.com/RegExr`: Interactive tool for helping you build regular expression pattern strings