Data Visualization with ggplot2 (Part 3)

Plots for Specific Data Types (Chapter 3)

Seun Odeyemi

2019-04-19

Contents

Load Libraries

```
library(readr)
library(dplyr)
library(ggplot2)
# library(ggplot2movies)
library(tidyr)
library(skimr)
library(knitr)
library(kableExtra)
library(RColorBrewer)
library(grid)
library(ggthemes)
library(forcats)
library(GGally)
library(here)
library(hexbin)
```

Choropleths

In this chapter we'll wrap up our discussion of specialty plots by considering **maps** and **animations** plus we'll see some concepts from the previous chapter coming to play. Let's begin with maps. Many people who work with maps are turning toward R as a Geographic Information System (GIS). This is because of its capabilities for spatial statistics and mapping are steadily improving. Using R as a full-flegded GIS is a course onto itself. Here my goal is to introduce you to two commonly used map types, which can both be easily produced in `ggplot2`: Choropleths and Carthographic Maps.

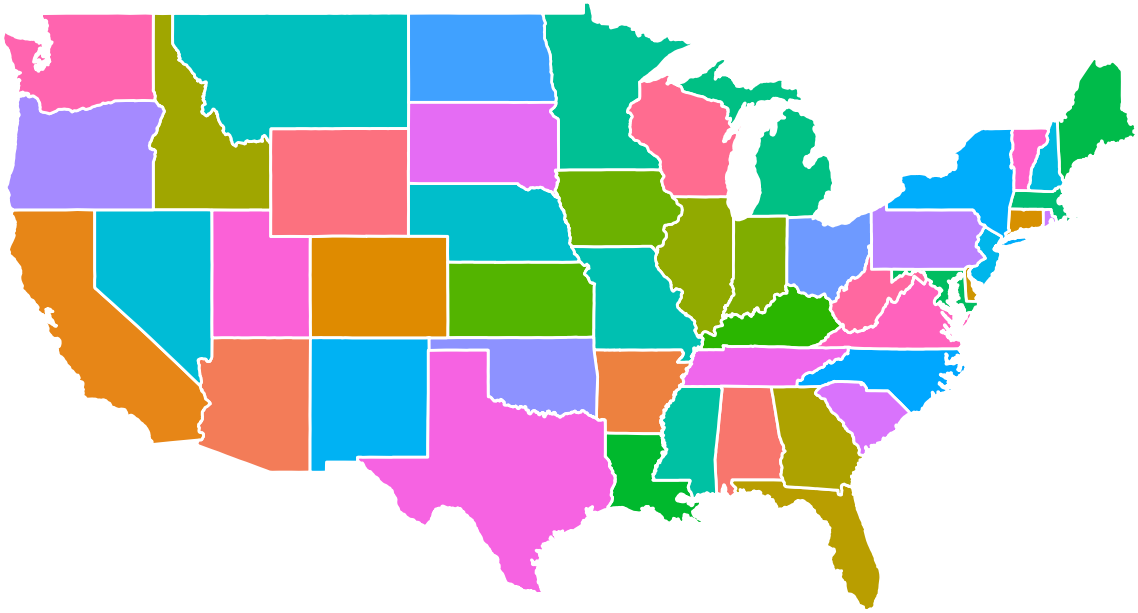
Let's start with Choropleths. You have likely encountered this type of map in popular media, in particular whenever elections are held. If you've completed the Kaggle challenge course, you would have also seen an

example of the Choropleths R package. The thing to remember about Choropleths is that basically we are just drawing a bunch of polygons (ok we could also drwa points and lines, but we'll just stick with polygons for the moment). You can imagine that when we will draw a map, like the outline of the United States shown below, it's basically a large polygon with many sides. All we need is a file which tells us the latitude or longitude of the point of the polygon. In some cases, you can find this included in a R package, like the example below. But, in the exercises we will explore how to use special shape files which can contain information about geograhical and political boundaries. Notice that, this is just a basic `ggplot2` plot. We can event adjust the coordinate system to a different projection (as shown below).

```
usa <- map_data("usa")

ggplot(usa, aes(long, lat, group = group)) +
  geom_polygon() +
  coord_map() +
  theme_nothing()
#> Error in theme_nothing(): could not find function "theme_nothing"

library(ggalt)
ggplot(usa, aes(long, lat, group = group)) +
  geom_polygon() +
  coord_proj("+proj=wintri") +
  theme_nothing()
#> Error in theme_nothing(): could not find function "theme_nothing"
```

Additionally, we may have information on many polygons such as individual states. This will allow us to visualize each individual polygon. Here, we've done that by mapping `region` onto the `fill` aesthetic.

```
states <- map_data("state")

usa <- map_data("usa")

library(ggmap)
ggplot(states,
       aes(long, lat,
           fill = region, group = group)) +
  geom_polygon(color = "white") +
  coord_map() +
  #theme(legend.position = "none")
  theme_nothing()
```



Now, if our polygons had names such as the states' names, then we could map that value associated with a polygon. For example, a dataset which you may have seen elsewhere on datacamp is the average price of weed in each US state. You can probably imagine that the only difference between this and previous plots was merging the weed prices data set with the polygon data set – matching up the state names and then mapping the price onto the fill aesthetic. Again, we can take advantage of all the tools that are provided by `ggplot2` so we may decide to use a more appropriate pallete and we make the darker colors the higher values. **I want to stress at this point that just because you have geographic data and you can make a choropleth does not mean you have to. It is not the only or a necessary visualization.** Many times people with geographic data default to maps, but recall that a continuous color scale is not actually an efficient encoder of continuous data. Choropleths excel when we have a few number of polygons or, in the case when we have many, a clear trend emerges.

An alternative is the classic **Cleveland Dot Plot**. The advantage here is that we can plot additional variables as we saw in previous lessons. We can also order the states alphabetically or, more revealingly, in order of a continuous variable. Now, we notice trends like most trips are between $ $275 $ to $350. North Dakota is the highest price and by relatively wide margin. It is not that this type of plot is better it just allows us to answer a different questions than a choropleth. It is also very unsexy, which is why you are likely to see choropleth instead of a dot plot in the popular press. Another advantage here is that we can facet the data grouping states in line with US Census defined regions. Here, we now see that the lowest prices are predominantly in the West. We can go further defining regions and divisions, but does not necessarily tell us anything new.

Working with Maps from the Maps Package: Adding Points

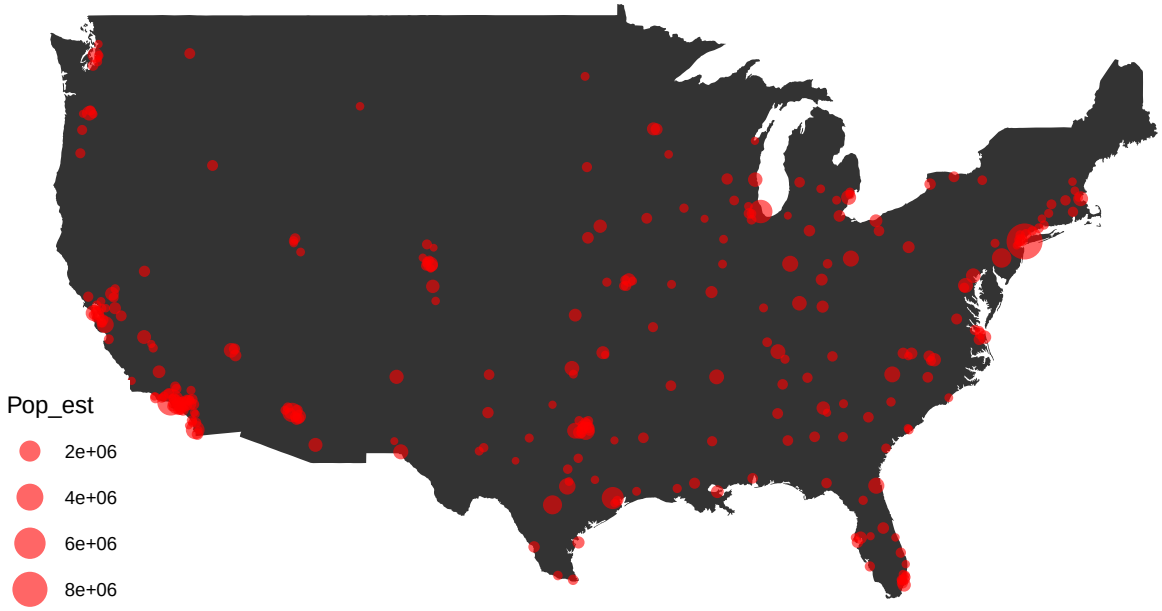
Now that you have some polygons, there are a number of things you can do. Here you'll add some data points, namely the location of US cities with a population over 100,000 (population estimation as of 2015). Since you're only looking at the continental US, Honolulu, Hawaii and Anchorage, Alaska are not included.

The data is stored in the `cities` data frame. You'll begin by drawing points of varying sizes, relative to the estimated population. An alternative is to use color instead of size, and in this case a nice trick is to order the data frame, so that the largest cities are drawn on *top* of the smaller cities. This is so that they will stand out against the background, which is particularly effective when using the viridis color palette.

```
# Finish plot 1

cities <- read_delim("datasets/US_Cities.txt", delim = "\t")

cities %>%
  head() %>%
  kable() %>%
  kable_styling(bootstrap_options = c("striped", "hover", "condensed"))
```

City	State	Pop_est	lat	long
Eugene	Oregon	163460	44.0567	-123.1162
Salem	Oregon	164549	44.9237	-123.0231
Hillsboro	Oregon	102347	45.5167	-122.9833
Santa Rosa	California	174972	38.4468	-122.7061
Portland	Oregon	632309	45.5370	-122.6500
Vancouver	Washington	172860	45.6372	-122.5965

```
ggplot(usa, aes(x = long, y = lat, group = group)) +
  geom_polygon() +
  geom_point(data = cities, aes(group = State, size = Pop_est),
             col = "red", shape = 16, alpha = 0.6) +
  coord_map() +
  theme_map()
```

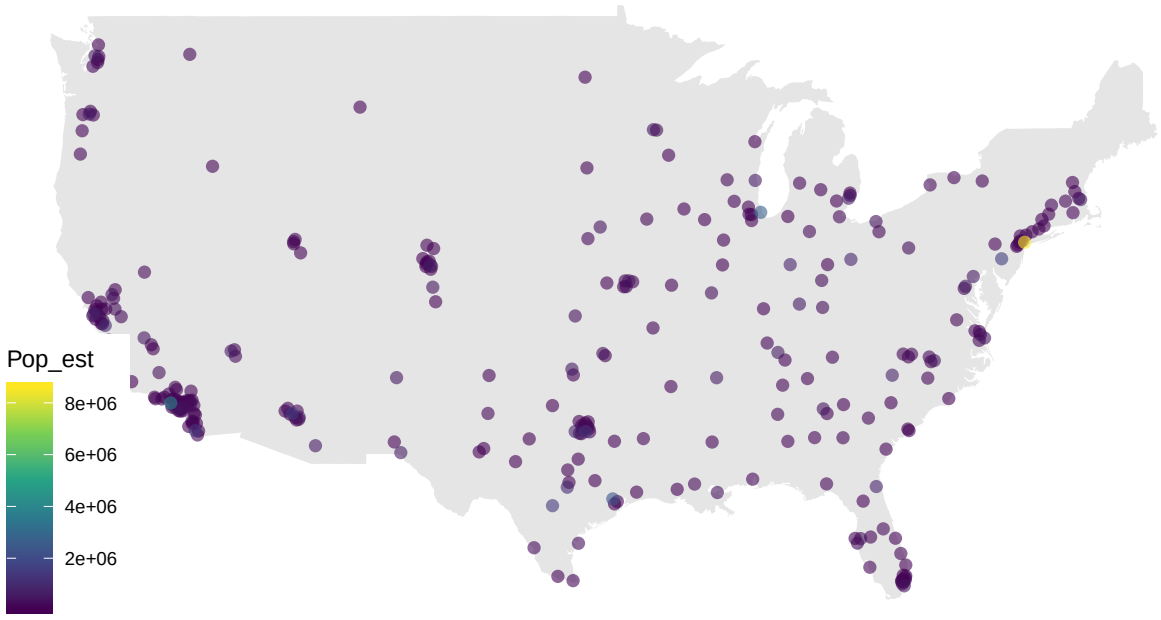



```
# Arrange cities
library(dplyr)
cities_arr <- arrange(cities, Pop_est)

cities_arr %>%
  head() %>%
  kable() %>%
  kable_styling(bootstrap_options = c("striped", "hover", "condensed"))
```

City	State	Pop_est	lat	long
Renton	Washington	100242	47.4867	-122.1953
Jurupa Valley	California	100314	33.0011	-117.4706
San Angelo	Texas	100450	31.4500	-100.4500
Davie	Florida	100882	26.0814	-80.2803
Greeley	Colorado	100883	40.4167	-104.7167
Vista	California	100890	33.1936	-117.2411

```
# Copy-paste plot 1 and adapt
ggplot(usa, aes(x = long, y = lat, group = group)) +
  geom_polygon(fill = "grey90") +
  geom_point(data = cities_arr, aes(group = State, col = Pop_est),
             shape = 16, alpha = 0.6, size = 2) +
  coord_map() +
  theme_map() +
```

```
  scale_color_viridis_c()
```



Great! New York appears as a bright yellow anomly in a sea of darker points. If you didn't set the order of the data, this point would have been obscured. You can also see LA, Chicago and Houston as lighter blue points.

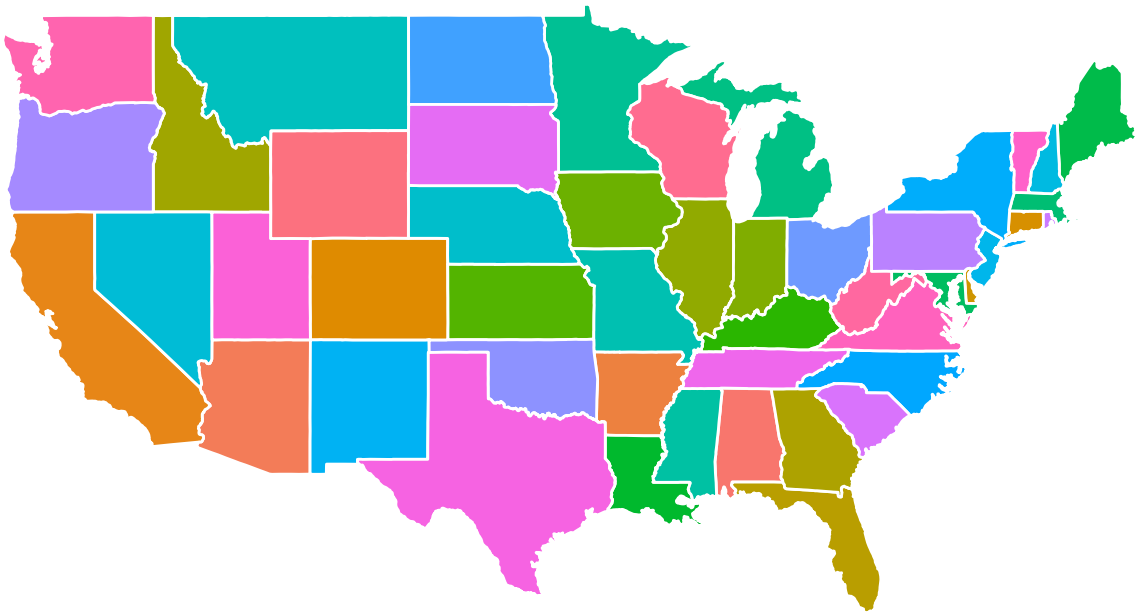
State Choropleth

To make a choropleth (a map in which areas are shaded according to some measure) you'll need information on state boundaries, which you can find in the `maps` package. Once the map information is converted into a data frame, you can merge this with another data frame containing some quantitative information, like the estimated population, and use that variable in our aesthetic mappings.

```
# Use map_data() to create state
state <- map_data("state") %>% as_tibble()

head(state)
```

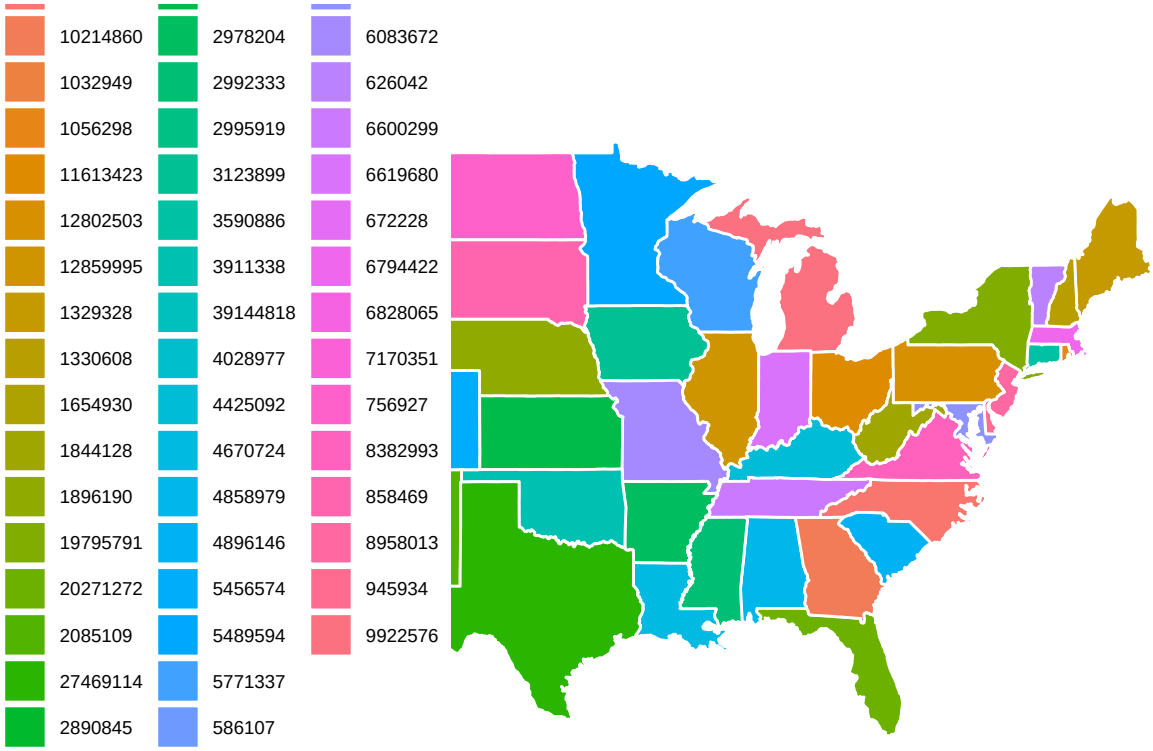
long	lat	group	order	region	subregion
-87.46201	30.38968	1	1	alabama	NA
-87.48493	30.37249	1	2	alabama	NA
-87.52503	30.37249	1	3	alabama	NA
-87.53076	30.33239	1	4	alabama	NA
-87.57087	30.32665	1	5	alabama	NA
-87.58806	30.32665	1	6	alabama	NA

```
state <- state %>% dplyr::rename ("state" = "region")

# Map of states
ggplot(state, aes(x = long, y = lat, fill = state, group = group)) +
  geom_polygon(col = "white") +
  coord_map() +
  theme_nothing()
```



```
# Merge state and pop: state2
library(janitor)

pop <- read_delim("datasets/US_States.txt", delim = "\t") %>% mutate_all(funs(tolower))
#> Warning: funs() is soft deprecated as of dplyr 0.8.0
#> please use list() instead
#>
#> # Before:
#> funs(name = f(.)
#>
#> # After:
#> list(name = ~f(.))
#> This warning is displayed once per session.

pop <- pop %>% clean_names(case = "snake")

state2 <- merge(state, pop)
```

```
# state %>% inner_join(pop, by = c("state" = "State"))

# state2 <- dplyr::select(state2, everything(), -state, -State)

# Map of states with populations
ggplot(state2, aes(x = long, y = lat, fill = pop_est, group = group)) +
  geom_polygon(col = "white") +
  coord_map() +
  theme_map()
```



Maps from Shapefiles

Although the built-in maps from the `maps` package are very convenient, using shapefiles is a more flexible way of accessing geographic and political boundaries.

Shapefiles can be used to describe points, polylines or polygons - here you'll focus on polygons for drawing maps.

A single shapefile actually consists of several files, each describing a specific aspect of the overall geometry. The three core file types are:

- .shp: the shape, the feature geometry itself.
- .shx: the shape index, a positional index.
- .dbf: the attribute, attributes for each shape arranged in columns.

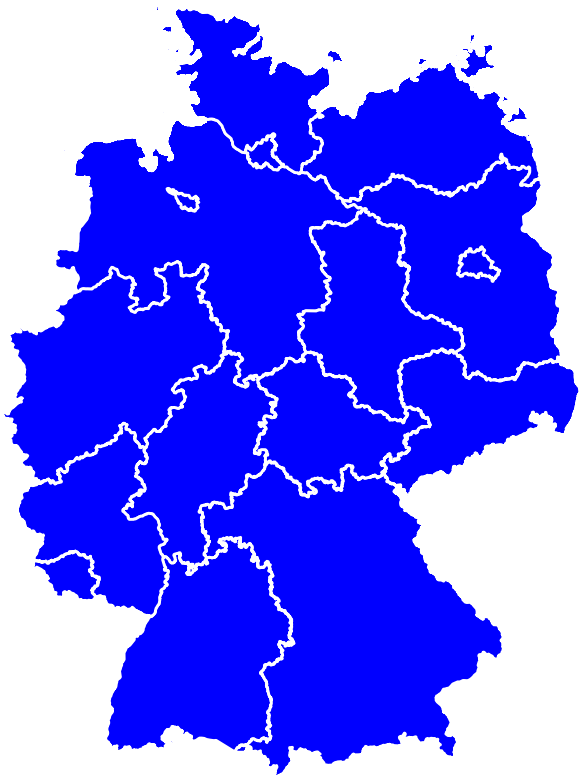
The prefix name of these files must be consistent and they must be kept in the same directory. The files

you'll use are for Germany, and all begin with DEU. The suffix specifies the level of organization:

DEU_adm0 is the administrative (political) boundaries of the entire country (like the usa object before) DEU_adm1 is the administrative boundaries for each of the 16 states (like the state object before) Let's start by importing the shapefile and creating a map of Germany. All shapefiles you need are in the shapes folder of your working directory; you can check it out with dir(). In the next exercise, you'll take things one step further.

NOTE: Building these maps is computationally heavy, so it can take some time before you see your results.

```
library(rgdal)
germany <- readOGR(dsn = "shapes", layer = "DEU_adm1")
#> OGR data source with driver: ESRI Shapefile
#> Source: "C:\Users\USER\repos\datacamp_courses\shapes", layer: "DEU_adm1"
#> with 16 features
#> It has 16 fields

bundes <- fortify(germany)

ggplot(data = bundes, aes(x = long, y = lat, group = group)) +
  geom_polygon(fill = "blue", col = "white") +
  coord_map() +
  theme_nothing()
```



Great! You can also imagine that it would be possible to draw single layers, like only the shape for Berlin.

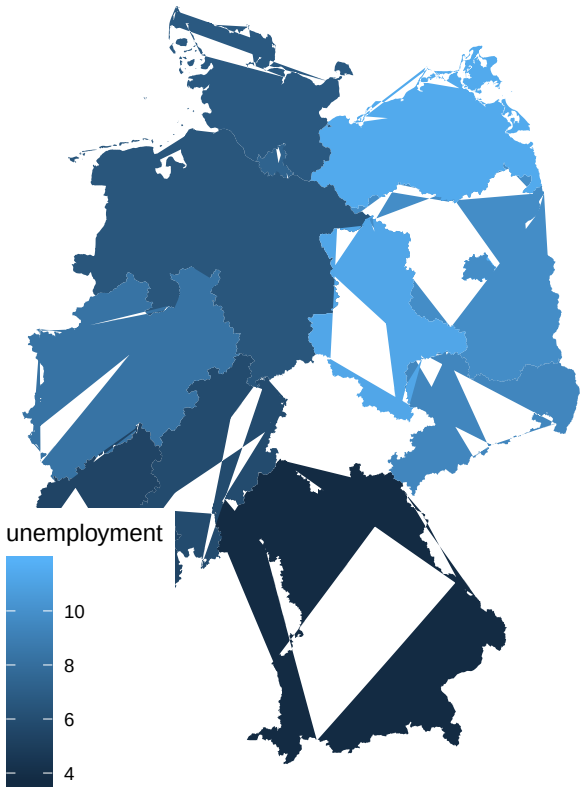
Choropleth from Shapefiles

Now that you have the shape data a neatly formatted data frame called `bundes`, you can easily merge it with state-level data. `germany`, `bundes` and 'unemp" (a data frame on unemployment in Germany) is available in your workspace. Can you do some data munging and then make a fancy choropleth?

If you check out `bundes`, you'll see that you've lost the state names, so merging isn't going to work out of box. That's why we've added two lines of code in the editor, to re-add state names.

NOTE: Building these choropleths is computationally heavy, so it can take some time before you see your results

```
unemp <- read_delim("datasets/germany_unemployment.txt", delim = "\t")

# re-add state names to bundes
bundes$state <- factor(as.numeric(bundes$id))
levels(bundes$state) <- germany$NAME_1

# Merge bundes and unemp: bundes_unemp
bundes_unemp <- merge(bundes, unemp)

# Update the ggplot call
p <- ggplot(bundes_unemp, aes(x = long, y = lat, group = group, fill = unemployment)) +
  geom_polygon() +
  coord_map() +
  theme_map()

p + geom_polygon(data = subset(bundes_unemp, id == "Berlin"))
```



At the outset this looks fine, but if you count the states, you'll see you only have 14! The reason is that Berlin and Bremen are city-states located within the shape definitions of other states. You can solve this by reording the data (like you did before), or by adding two specific layers, for example, `geom_polygon(data = subset(bundes_unemp, id == "Berlin"))`.

Cartograhic Maps

As we saw in the exercise, Choropleths are useful if you have the right data type including the shapefile and associated variables. The other type of maps which you'll commonly see is a Carthographic Map. By this, I mean some kind of a image such as **drawn** (e.g. topographical maps) depicting distinct features of a landscape like altitude and infrastructure or **photographic** such as satelite images or **hybrid** of the two. To obtain this images we'll use the `ggmap` package, which allows us to choose a source and a type of map. For now, we'll stick to the canonical formats although there are various options to customize your map features. For example, we can obtain a google style map with a fairly broad zoom level of 3

This map is centered on Linkoping, Sweden, which is a bit difficult to see since it is so far zoomed out. Increasing the zoom allows us to get closer and closer still. There are many preset map types available such as this `watercolor` style in the `stamen` package (see `?get_stamenmap`) or this black and white style called `toner` style. The `hybrid` style map gives some indication of the road network, but, of course, we can just ask for the `satellite` images in isolation.

```
# watercolor style
wc_13 <- get_map(location = "Berlin, Germany", zoom = 13,
source = "stamen", maptype = "watercolor")
ggmap(wc_13, extent = "device")
```

```
# toner style
```

```
ton_13 <- get_map(location = "Berlin, Germany", zoom = 13,
source = "stamen", maptype = "toner")
ggmap(ton_13, extent = "device")
```


```
# hybrid style
hyb_13 <- get_map(location = "Berlin, Germany", zoom = 13,
source = "stamen", maptype = "toner-hybrid")
ggmap(hyb_13, extent = "device")
```

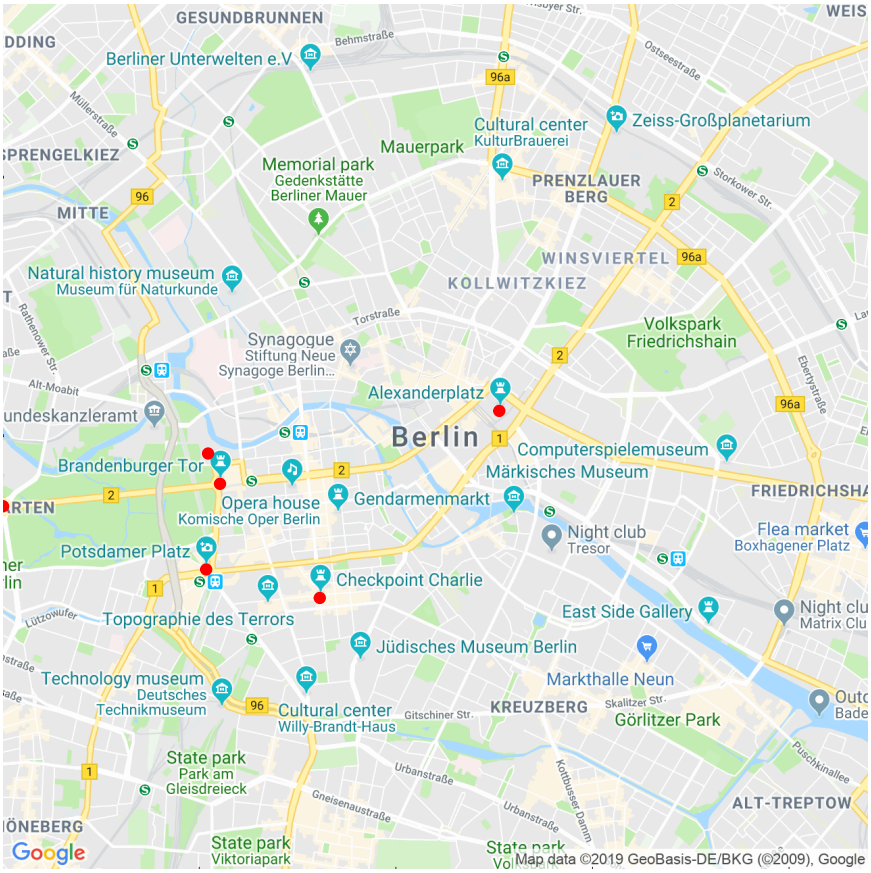

```
# satellite style
sat_13 <- get_map(location = "Berlin, Germany", zoom = 13,
source = "google", maptype = "satellite")
ggmap(sat_13, extent = "device")
```

Now, this is all fine and good, but what we really want is to plot geographic data onto this image using it as a base. For example, if we had a list of sites with their respective latitudes and longitudes. This can be obtained in a number of ways. We may have actual gps data in a field collection; perhaps we have addresses from a database; or, like in the case below, names of landmarks in Berlin. In this case we can use the `geocode` function to query `google` for the lats and longs of these places (as shown below). Since, this is a `ggplot2` object, we can add points using the `geom_point()` layer. The problem here is that the map region is applied before we added this layer. It is not redrawn as we'll typically expect in `ggplot2` because of the way it was obtained.

```
berlin_sites <- c("Brandenburger Tor", "Potsdamer Platz",
"Victory Column Berlin", "Checkpoint Charlie",
"Reichstag Berlin", "Alexander Platz")

places <- geocode(berlin_sites)

# Add column with cleaned up names
places$location <- sub(" Berlin", "", berlin_sites)

glimpse(places)
#> Observations: 6
#> Variables: 3
#> $ lon      <dbl> 13.37770, 13.37594, 13.35012, 13.39039, 13.37619, 13....
#> $ lat      <dbl> 52.51627, 52.50965, 52.51454, 52.50744, 52.51862, 52....
#> $ location <chr> "Brandenburger Tor", "Potsdamer Platz", "Victory Colu...
```

```
# google/roadmap - zoom = 13
road_13 <- get_map(location = "Berlin, Germany", zoom = 13,
```

```
source = "google", maptype = "roadmap")
ggmap(road_13, extent = "device") +
  geom_point(data = places, col = "red")
```



A workaround is to not obtain a general map of our target, but to get a map defined by our points i.e. to define a bounding box (`bbox`) around the points with some buffer room. This is depicted below, but as you can see, it doesn't work perfectly. (Actually, we've lost two points on our map.). We'll solve this problem by changing the buffering around the points, but before we do that, I want to remind you that just all `ggplot2` objects we can modify this by setting mappings.

```
# Workaround 1: Using Boundary Box
bbox <- make_bbox(lon = places$lon, lat = places$lat, f = .1)
boxed_14 <- get_map(location = bbox, zoom = 14,
                    source = "google", maptype = "roadmap")
ggmap(boxed_14, extent = "device") +
  geom_point(data = places, col = "red")
```


```
# Workaround 2: Using Boundary Box and ggplot2 mappings
bbox <- make_bbox(lon = places$lon, lat = places$lat, f = .1)
boxed_14 <- get_map(location = bbox, zoom = 14,
                    source = "google", maptype = "roadmap")
ggmap(boxed_14, extent = "device") +
  geom_point(data = places, aes(col = location), size = 3) +
  scale_colour_brewer(palette = "Set1")
```

Above, we've mapped location onto color. This all comes together in our final plot where we use a `toner` map type and instead of points with a color legend we use `geom_label` to add boxes with text labelling the location of each site.

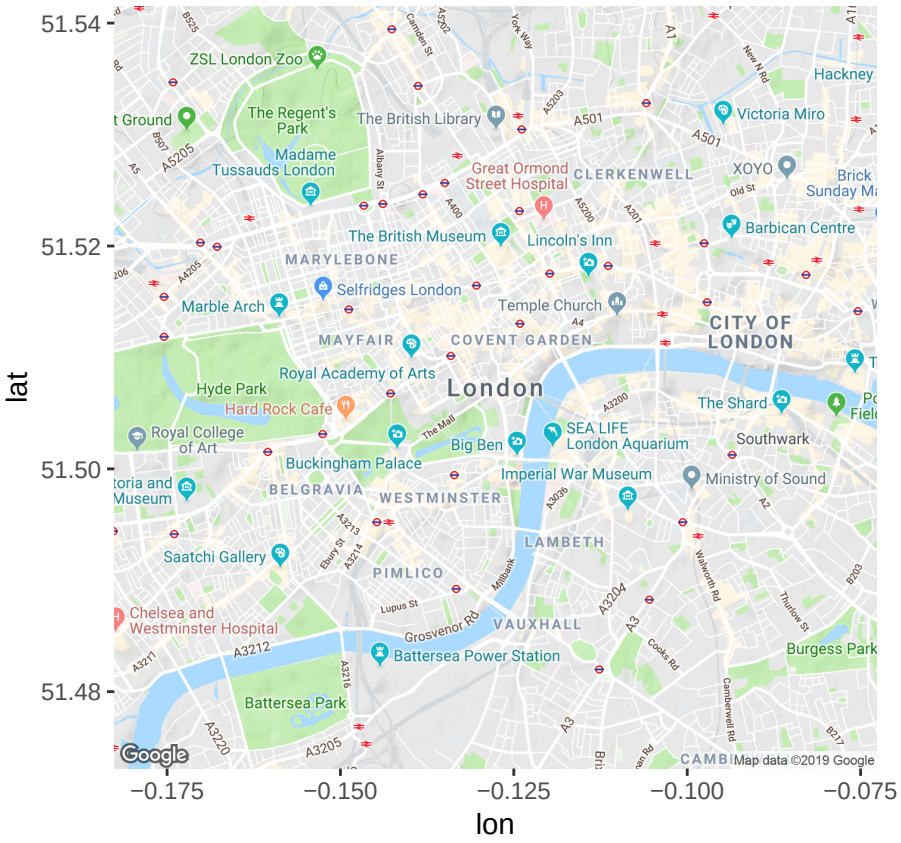
Different templates

`ggmap` provides the `get_map()` function to access Google Maps. In this exercise you'll work with some preloaded `ggmap` objects. The code that created the two maps you'll plot is

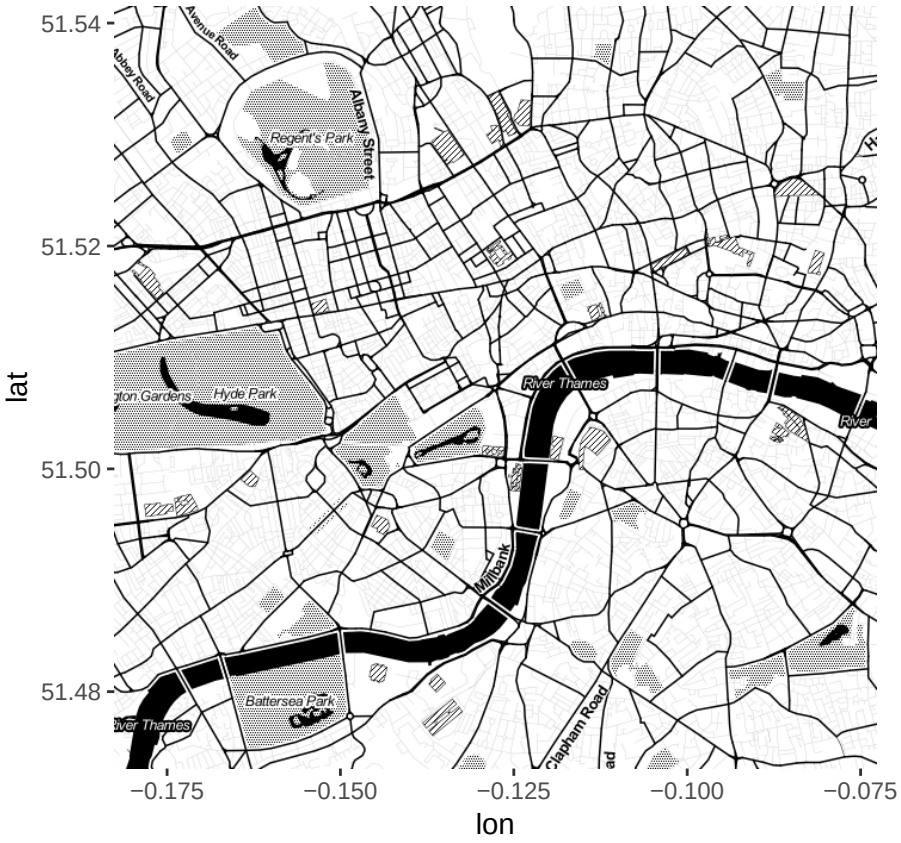
```
london_map_13 <- get_map(location = "London, England", zoom = 13)
london_ton_13 <- get_map(location = "London, England", zoom = 13, source = "stamen", maptype = "toner")
```

These two objects are already loaded in your environment. Feel free to play around with `get_map()` in the console, but be warned, sometimes you might not always get what you expect!

```
london_map_13 <- get_map(location = "London, England", zoom = 13)
london_ton_13 <- get_map(location = "London, England", zoom = 13, source = "stamen", maptype = "toner")

# Create the map of London
ggmap(london_map_13)
```




```
# Create the second map of London
ggmap(london_ton_13)
```



Great! All from inside R, pretty amazing!

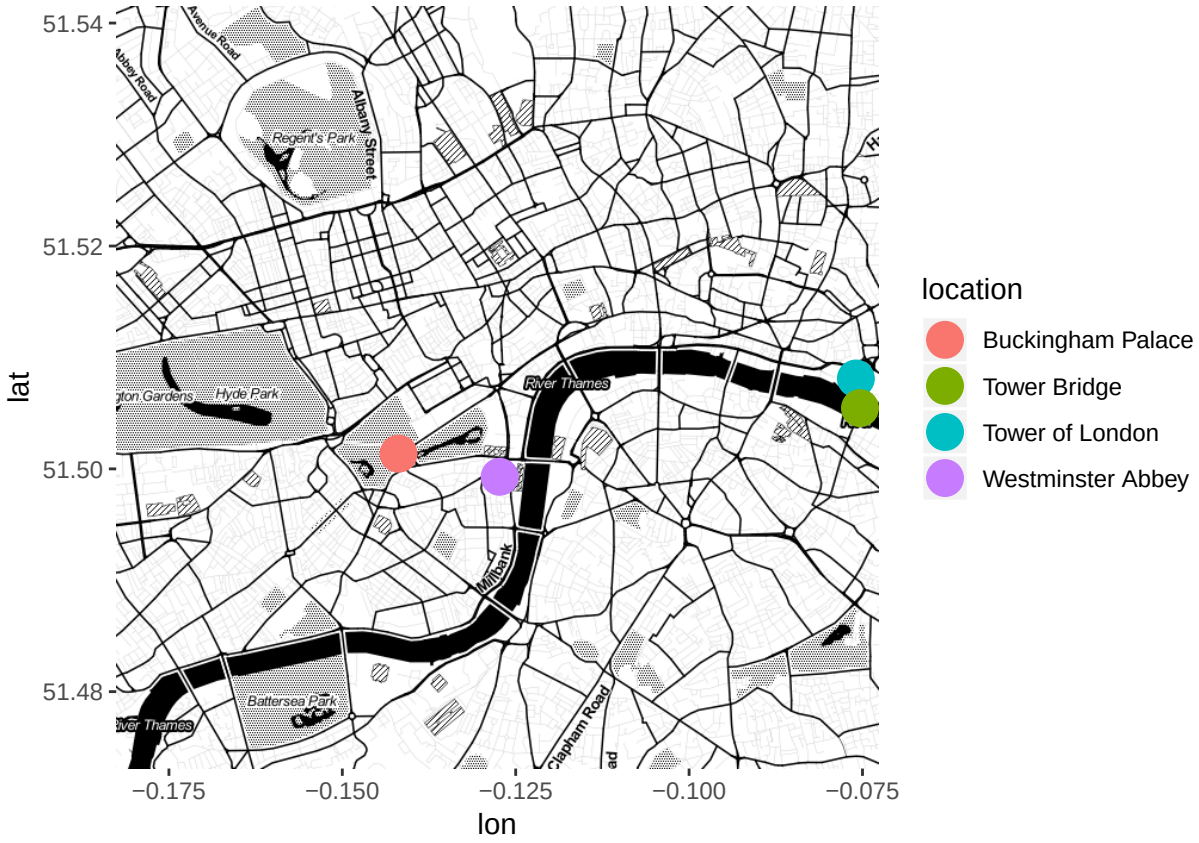
Mapping Points onto a Carthographic Map

Now that you know how to get a cartographic map, you'll use `ggplot2` layers to add points. The object `london_sites`, that's available in your workspace, is a character vector of some locations you'd like to plot. You'll add points to the cartographic map you derived in the previous exercise. We've already geocoded the locations in london_sites for you using the code

xx <- geocode(london_sites)

As you can see, the coordinates are available in a dataframe called `xx`. Additionally, the ggmap object `london_ton_13` from the previous exercise is still available.

```
london_sites <- c("Tower of London, London",   "Buckingham Palace, London",
                  "Tower Bridge, London", "Westminster Abbey, London")

xx <- geocode(london_sites)

# Add a location column to xx
xx$location <- sub(", London", "", london_sites)

# Add a geom_points layer
ggmap(london_ton_13) +
  geom_point(data = xx, aes(col = location), size = 6)
```



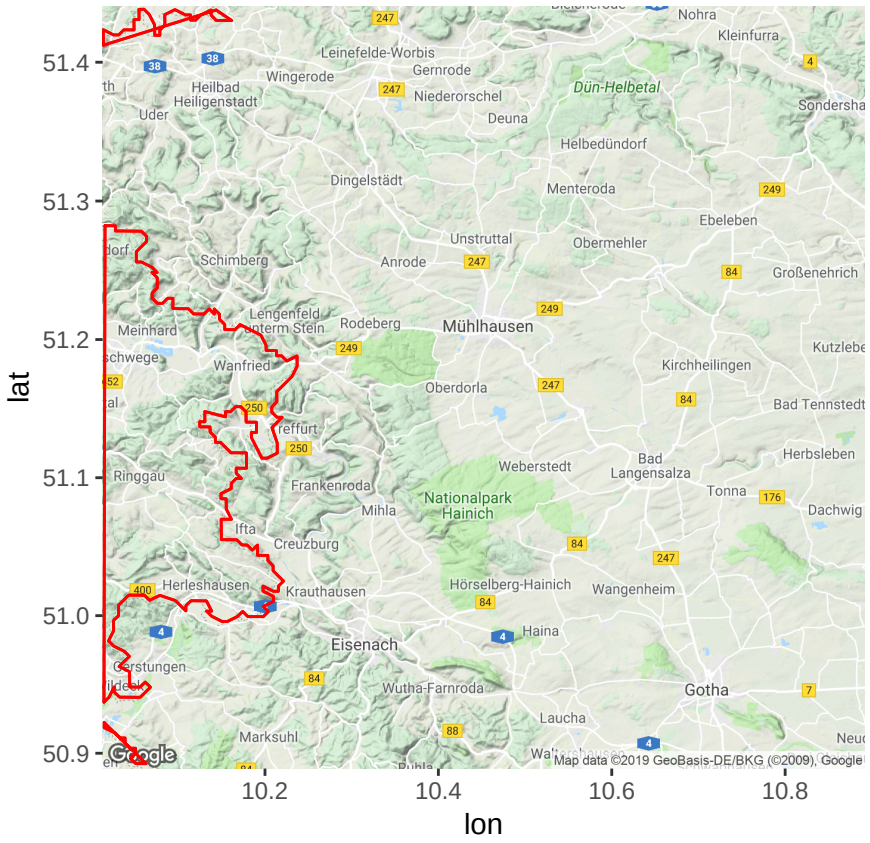
Nice! How do you like your map? It looks so cool!.

Combine Carthographic and Choropleths Maps

Instead of plotting points on a cartographic map, you can also draw polygons, similar to what you saw earlier in this chapter.

The polygons come from the shapefile for the German states that you used previously. You'll use the data frame version, `bundes`, because it's more flexible. The data frame is already available in your workspace along with a `ggmap` of Germany, so you can get started straight away!

```
# get_map of Germany
germany_06 <- get_map (location = "Germany")

# Plot map and polygon on top:
ggmap(germany_06) +
  geom_polygon(data = bundes,
               aes(x = long, y = lat, group = group),
               fill = NA, col = "red") +
  coord_map()
```



Great! As usual, the possibilities with ggplot2-based plots are practically endless!

Animations

Animations are a great tool for dealing with dense temporal data, but actually they can be useful as a great exploratory tool for any integer or categorical variable. There are several ways to produce animations in R.

- The most cumbersome is to produce individual output files in a `for loop` and then using an external program to compile that into an animated gif or movie.
- The next best thing is to use the `animation` package, which saves you from having to externalize the procedure. It can all be done within R.
- Here we'll look at `gganimate` which goes one step further and acts as a wrapper for the `animation` package when using `ggplot2` objects. This allows for easy animations when applicable.

A very effective example of advanced animations is the motion chart developed by Hans Rosling currently professor of international health at the Karolinska Institute in Stockholm and founder of `gapminder`. The `gapminder` project contains UN data on `life expectancy` and `GDP` among many other variables for many countries. In this example we'll look at those two variables plus population and continent as depicted in the dataframe below. We'll try to see the trend over time. As this point, we'll probably make a plot which looks something like the plot below. The aesthetic mapping we use should be clear: population is mapped to size, continent to color, life expectancy is on the y-axis, and gdp per cap is on the x-axis. The problem here is that there is too much information. We may imagine that we could use facets to produce individual plots for each country, but there are 142 countries. We could do that for continents since there are only 5, but that will still not solve the problem of how to represent time for each country.

Han's solution was to produce a motion chart. A full motion chart has controls for adjusting the time, switching the aesthetic mappings and geoms. We'll take a look of how to make a simple motion chart in the exercises. Here, we'll make a basic form of a motion chart, which will just be a basic animation that

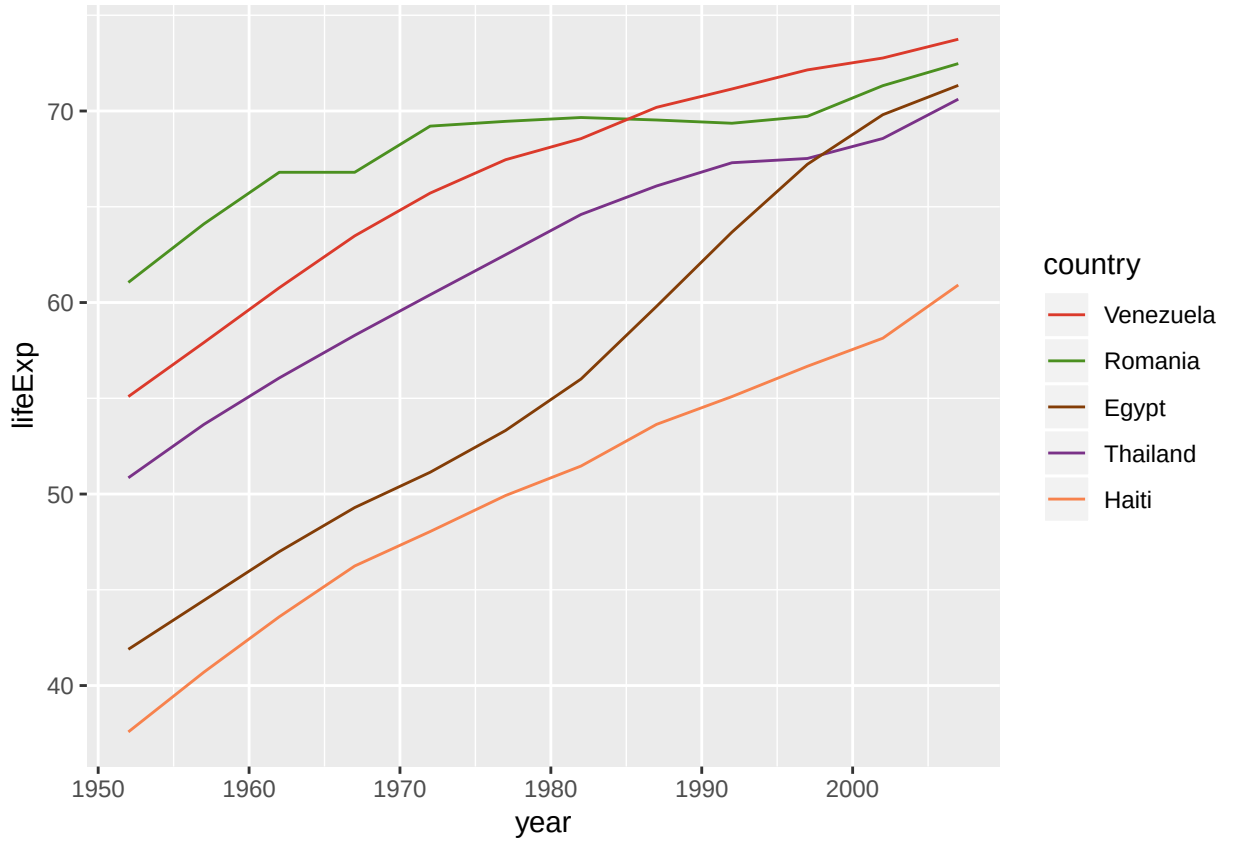
cycles through each year. The resulting plot allows you to see how both gdp and life expectancy have both increased around the world. To do this, we'll use a new aesthetic, `frame`, which will be used as a time point in our animations. We use the frame aesthetic like we use other aesthetics we've seen previously such as x, y, size, color, fill, etc. As I mentioned we are not limited to using the year. For example, we could have switched our aesthetic mappings and cycle through each country. We can also adjust the speed at which each image is cycled allowing more time to look at each plot or speeding up the process to give the impression of continuous fluid change. Since, we can save our animations as standalone images, they can be embedded in presentations or websites without any additional software requirement. Or, we can play them in the viewer quickly moving between plots using the forward and reverse controls.

Let's take a look at another example where we can get a sense of continuous change. Here we consider population size predictions. This animation which is a minimum working example of a publication quality version recently published by Kyle Walker depicts the population prediction for Japan for the coming decades. We have 4 variables:

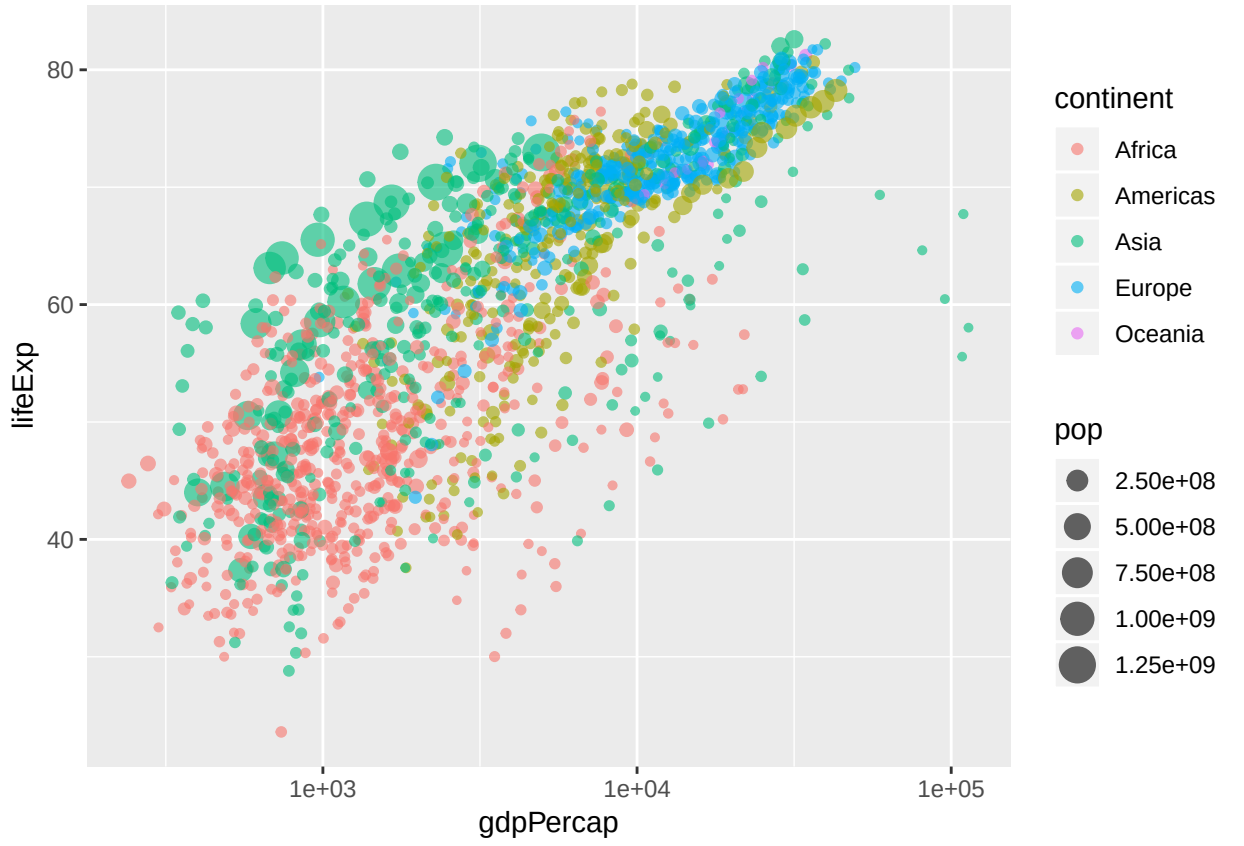
- The population prediction will be mapped onto the x-axis,
- Age onto the y-axis.
- Sex onto color, and
- The animation results from mapping year onto the frame aesthetic.

The nature of this data gives an impression of fluidity as we watch how the population is expected to change in the coming years in Japan.

```
library(gapminder)

glimpse(gapminder)
#> Observations: 1,704
#> Variables: 6
#> $ country   <fct> Afghanistan, Afghanistan, Afghanistan, Afghanistan, ...
#> $ continent <fct> Asia, Asia, Asia, Asia, Asia, Asia, Asia, Asia, Asia...
#> $ year      <int> 1952, 1957, 1962, 1967, 1972, 1977, 1982, 1987, 1992...
#> $ lifeExp   <dbl> 28.801, 30.332, 31.997, 34.020, 36.088, 38.438, 39.8...
#> $ pop       <int> 8425333, 9240934, 10267083, 11537966, 13079460, 1488...
#> $ gdpPercap <dbl> 779.4453, 820.8530, 853.1007, 836.1971, 739.9811, 78...

# gapminder %>% ggplot( aes(year, pop)) +
#   geom_point() +
#   facet_grid(continent ~ .) +
#   scale_y_log10()

h_countries <- c("Egypt", "Haiti", "Romania", "Thailand", "Venezuela")
h_dat <- droplevels(subset(gapminder, country %in% h_countries))
h_dat$country <- with(h_dat, reorder(country, lifeExp, max))
ggplot(h_dat, aes(x = year, y = lifeExp)) +
  geom_line(aes(color = country)) +
  scale_colour_manual(values = country_colors) +
  guides(color = guide_legend(reverse = TRUE))
```




```
ggplot(gapminder, aes(x = gdpPercap, y = lifeExp,
                      colour = continent,
                      size = pop)) +
  geom_point(alpha = 0.6) + # details omitted
  scale_x_log10()
```



The Population Pyramid

Animations are particularly useful for temporal or geospatial data, and they are surprisingly easy to make! Here, you simply loop over the time variable in your dataset, composing a new plot for each subset in the data. These individual images are then cataloged in an animated GIF file.

To show this you'll use a great animated population pyramid that was presented on the Revolutions blog. There are many more adjustments you could have made to the plot, but we'll just make a barebones version here.

```
# Load Japan data

japan <- read_delim("datasets/japanPOP.txt", delim = "\t")

# Inspect structure of japan
str(japan)
#> Classes 'spec_tbl_df', 'tbl_df', 'tbl' and 'data.frame': 8282 obs. of  4 variables:
#>  $ AGE : num  0 1 2 3 4 5 6 7 8 9 ...
#>  $ POP : num  -572954 -581748 -585239 -582223 -568788 ...
#>  $ time: num  2010 2010 2010 2010 2010 2010 2010 2010 2010 2010 ...
#>  $ SEX : chr  "Male" "Male" "Male" "Male" ...
#>  - attr(*, "spec")=
#>   .. cols(
#>   ..   AGE = col_double(),
#>   ..   POP = col_double(),
#>   ..   time = col_double(),
#>   ..   SEX = col_character()
```

```
#>   .. )

# Finish the code inside saveGIF

saveGIF <- function(x, y){

  # Loop through all time points
  # Create a series of plots using a for loop.
  # Use i to iterate over unique(japan$time).
  # Inside for loop, each time create a subset of japan for which japan$time == i.
  # Call this subset data.
  for (i in unique(japan$time)) {

    # Subset japan: data
    data <- subset(japan, time == i)

    # Finish the ggplot command
    p <- ggplot(data, aes(x = AGE, y = POP, fill = SEX, width = 1)) +
      coord_flip() +
      geom_bar(data = data[data$SEX == "Female",], stat = "identity") +
      geom_bar(data = data[data$SEX == "Male",], stat = "identity") +
      ggtitle(i)

    print(p)

  }
  # movie.name = "pyramid.gif", interval = 0.1
}
# saveGIF(x = "pyramid.gif", y = 0.1)
```

Animations with gganimate

Another method of making animations is to use the `gganimate` package. This provides the `gg_animate()` function, which is basically a convenient wrapper for the functions in the `animation` package, so it's pretty useful for straight-forward plots. `gg_animate()` provides a new aesthetic argument, `frame`, that allows you to side-step having to establish a for loop for each time point, like you did in the previous example.

Here you'll see another example of making an animation, to cycle through years and look at changing linear models. The dataset is the `Vocab` data frame from the `car` package which you encountered in the previous course. It's already available in your workspace.

```
library(car)
library(gganimate)

# # Update the static plot
# p <- ggplot(Vocab, aes(x = education, y = vocabulary,
#                        color = year, group = year,
#                        frame = year, cumulative = TRUE)) +
#   stat_smooth(method = "lm", se = FALSE, size = 3)
#
# # Call gg_animate on p
# x <- gg_animate(p, filename = "vocab.gif", interval = 1.0)
#
# x
```