A Guide to Pipes and ggplot

Jennifer Lin

2021-04-14

Goals of this Workshop

There are learning goals for this workshop

1. Increase coding proficiency with `tidyverse` packages, specifically `dplyr` and `ggplot`.
2. Develop skills necessary to communicate results clearly with data visualization tools
3. Apply your knowledge to answer your own research questions.

Data Overview

To complement this workshop, I prepared a cleaned version of the 2018 Cooperative Elections Studies (formerly the Cooperative Congressional Elections Studies) dataset. The data can be accessed from my website or by cloning the workshop GitHub repository. We will largely focus on select variables in each of the sections below but here is a guide to all the variables that I included in the file along with a brief description.

Variable	Description
`starttime`	Start Time
`endtime`	End Time
`caseid`	Respondent ID
`region`	Region in country
`county`	County Name
`countyfips`	County FIPS Code
`state`	State Abbreviations
`inputstate`	State FIPS Code
`gender`	Respondent Gender
`birthyr`	Year of Birth
`votereg`	Registered to vote?
`pid3`	Party ID 3 categories
`amtDonate`	Amount donated to politics
`appTrump`	Approval of Trump
`appCongress`	Approval of Congress
`appSupremeCourt`	Approval of Supreme Court
`vote16`	Vote Choice in 2016
`ideo3`	Ideology 3 category
`Age`	Respondent Age

Now that you know what is in the data, let's load in the data file using the `readr` package's command `read_csv()`

```
library(readr)
CCES_2018 <- read_csv("CCES18_subset.csv")
```

Pipes and dplyr

The `tidyverse` package, especially `dplyr`, provides great tools to help us wrangle our data. The `dplyr` package contains many useful functions mainly for data cleaning and manipulation. Each of these functions, which I will go into detail below, can be connected with pipes (`%>%`) \marginnote{You can get %>% using COMMAND+Shift+M on a Mac}.

Pipes are useful to connect functions. It is akin to a "NEXT" in a chain of `dplyr` functions where R will run the first function, followed by the next after the pipe, and so on.

```
library(dplyr)

data %>%
  step 1 %>%
  step 2 %>%
  step 3 %>%
  steo 4
```

`dplyr` functions

As for all the functions discussed in these notes, see the function documentation for more details. These are presented in order to give you a brief overview but not to go into super specific details on all the possibilities that are available in these functions and packages.

Function	Description
`filter()`	Sift observations down to the ones you want
`select()`	Pick variables that you want
`mutate()`	Create variables out of preexisting ones
`rename()`	Change the name of a variable
`group_by()`	Group analyses by a factor variable
`summarise()`	Create summary statistics for a variable via a desired function. If using `group_by()`, keep groups with `.groups = 'keep'`

For each of these commands (and there are more that `dplyr` offers), you can use them so that `data %>% command(...)`.

To provide you a bit more detail on the `filter()` command, this command allows you to sift the data using conditions on one or more variables. This takes the form `variable + operation + condition`). A list of the possible operations include:

Operator	Description
==	exactly equal to
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
`%in%`	is in this list
!=	not equal to
`is.na()`	NA values

For the `select()` function, you can pick a list of variables or a collection of variables that fit a certain condition. To pick a variable that fit a selection of conditions, you can use one of the following commands in the `select()` function

Operator	Description
`everything()`	everything in data
`last_col()`	last column
`starts_with()`	start with a string
`ends_with()`	ends with a string
`contains()`	contains a string

Here is an example of the functions in `dplyr` applied to our data to understand institutional approval across states and parties:

```
state_party_Congress <- CCES_2018 %>%
  select(state, pid3, region, appCongress, appTrump) %>%
  filter(!is.na(state)) %>%
  filter(!is.na(pid3)) %>%
  mutate(
    region = case_when(
      region == 1 ~ "Northeast",
      region == 2 ~ "Midwest",
      region == 3 ~ "South",
      region == 4 ~ "West"
    )
  ) %>%
  group_by(state, region, pid3) %>%
  summarise(
    mean        = mean(appCongress, na.rm = TRUE),
    pctCongAppv = sum(appCongress == 1, na.rm = TRUE)/length(appCongress),
    pctCongDApv = sum(appCongress == 2, na.rm = TRUE)/length(appCongress),
    pctPresAppv = sum(appTrump == 1, na.rm = TRUE)/length(appTrump),
    pctPresDApv = sum(appTrump == 2, na.rm = TRUE)/length(appTrump),
    .groups     = 'keep'
  )
```

Graphics with ggplot

To help us present our results, the `ggplot2` package in the `tidyverse` series provides useful tools to streamline the process. The way `ggplot` graphs are constructed is with layers, one on top of another, telling R how to enhance your graph next. In thinking about how to best organize these various useful functions, I conceptualize a `ggplot` graph using the following schematic:

```
library(ggplot2)
```

```
ggplot(data layer)+
  graph layer +
  label layer +
  scale loayer +
  theme layer +
  others
```

Here, I use the word "layer" to refer to a group of commands that `ggplot` has to do certain things under each layer. For example, there are multiple possibilities that can go under the "graphs layer" depending on what graph you desire to make. The session will go into each of these layers in more detail and you can use the headers in this document to sort out my discussions on each of these layers.

For `ggplot` there is no hard and fast order that these functions must appear. However, no matter the order, notice the `+` that adds the layers together. This is akin to the `%>%` described earlier, which connects one function to the next.

The Data Layer

To begin building a plot using `ggplot`, you need to establish the data frame in which your data will originate. Plots with `ggplot` often start with the function `ggplot()` which allows us to establish the data and the x and y components. The arguments are as follows

```
ggplot(
  data = dataframe,
  aes(
    x = x_var,
    y = y_var,
    group = grouping_var,
    color = color OR variable to color,
    fill  = color OR variable to fill
  )
)
```

The schematic here illustrates the positions of some of the often used arguments in this layer. The data and the x and y arguments are often necessary for most graphs.

There is an exception to this rule, however, and it occurs when you have multiple data frames that all need to coalesce onto different layers of the same graph. In that case, you do not specify the data frame in the `ggplot()` function. Rather, you will wait for the graphs layer and specify a `data = dataframe` as an argument there.

Beyond this, the grouping argument presents the opportunity to group by a factor variable if you desire. This is often useful if you are not plotting from a summary table and need to do a `group_by()` equivalent in your graphs. The fill and color arguments allow you to add some color to your graphs. Before setting these arguments, think about the graph that you are building. You will use the `fill` setting when there is a space for you to fill. These include the space inside boxplots, bar graphs or histograms. If you are drawing a line graph or scatterplot, you will use `color`

Think: color the lines, fill the space.

The Graph Layer

The next layer is the graph layer, which tells R what graph you are trying to create. The table contains some commonly used graph functions. Each of these contain a `data` and `aes()` option, similar to the data layer, which allows you to specify a data set and options in case these differ from above. These options can be especially useful if you want to put different data on the same graph.

Operator	Description
`geom_line()`	line graph
`geom_bar()`	bar plot
`geom_histogram()`	histogram
`geom_boxplot()`	boxplot
`geom_violin()`	violin plot
`geom_sf()`	maps with shapefiles

The Label Layer

Now that you have the data and the graph specified, the next steps are the adjust it and change the way it looks. These do not have to be completed in a specific order (or at all, but I would recommend it nonetheless).

One of the things that you might want to do is to change the labels. The quickest way to change the x and y axes are using `xlab()` and `ylab()` with the labels that you want inside the parentheses and in quotes. To change the title and the subtitle, use `labs()` such that you include what you want in quotes and using the function's argument specifications:

```
labs(
  title    = "",
  subtitle = "",
  caption  = ""
)
```

The Scale Layer

An additional thing that you might want or need to do is adjust the scales. By default, `ggplot` adjusts the scales based on the best fit of the graph from the data presented. However, sometimes, you want to override these settings.

To start, if you want to adjust the limits on continues x or y axes, you can use `xlim()` or `ylim()` with your desired start and end encased in the parentheses.

However, if you want to adjust the scales and legends for continuous and categorical variables, `ggplot`also contains a variety of `scales` functions to help you do just that. These come in the format of `scale_[SOMETHING]_[SOMEHOW]()` and their arguments vary depending on the thing you are scaling and how you are doing it.

Most scales take the following arguments

- `name` = Name the thing you are scaling
- `breaks` = Locate where you want to break it
- `labels` = Assign each break point a name
- `limits` = Set upper and lower bounds (if applicable)

Here are some examples:

- `scale_x_continuous()`
  - `[SOMETHING]` = x-axis
  - `[SOMEHOW]` = continuously
- `scale_fill_manual()`

  - `[SOMETHING]` = shape fill
  - `[SOMEHOW]` = manually

- `scale_colour_brewer(palette = "[PALETTE NAME]")`
  - `[SOMETHING]` = color
  - `[SOMEHOW]` = Using the R Color Brewer palette

The Theme Layer

As you will notice, the default background for `ggplot` is the gray background with text that is rather small. This might not be the most ideal presentation on a paper, much less a conference presentation. So, you will likely want to change the theme.

In `ggplot`, there are 2 different types of themes. One sets the overall style and another sets fonts and sizes.

THE OVERALL STYLES which I term as "global themes" can be set with `theme_[STYLE]()` from `ggplot` or `ggtheme`, which is an additional package that you will need to install in order to access them. Some great themes, in my opinion are:

Classic Options

- `theme_bw()`
- `theme_classic()`
- `theme_light()`
- `theme_linedraw()`

from `ggthemes`

- `theme_tufte()`
- `theme_gdocs()`
- `theme_calc()`

ON TOP OF THESE STYLES, we can customize fonts and sizes using options in `themes()`. Some of the arguments are available below:

```
theme(
  plot.title      = element_text(hjust  = 0.5),
  axis.title      = element_text(colour ="black"),
  axis.title.y    = element_text(size   = 12),
  legend.position = 'none'
)
```

The arguments are structured rather intuitively such that the arguments are structured as `[Thing you want to customize].[Specific aspect to customize].[other applicable specifications]`

For example, if you want to customize the plot title, you use `plot.title`. If you want to customize the styles at the axes in general, use `axis.title`. Specifically, if you want this style just to apply to the y-axis, use `axis.title.y`. If you want to customize the legend position, use `legend.position`. The list goes on and can be seen in the documentation for `themes()`[1].

[1] `https://ggplot2.tidyverse.org/reference/theme.html`

To customize the text, use `element_text()`, which carries several options like the common ones including `hjust` (horizontal adjustment), color and size. The `hjust` option allows you to left align (the default), center align (`hjust = 0.5`) and right align (`hjust = 1`). Color sets the color of the text to be the color that you specify. Size sets the font size. Keep in mind the paper size that you want to export and the display so that your font size is readable even when you need to shrink the graph on a paper.

The "Others" Layer

The "Others" Layer is the place where I am putting things that do not neatly fit into any of the above categories and are those things that most graphs often do without. This include facets and coordinates.

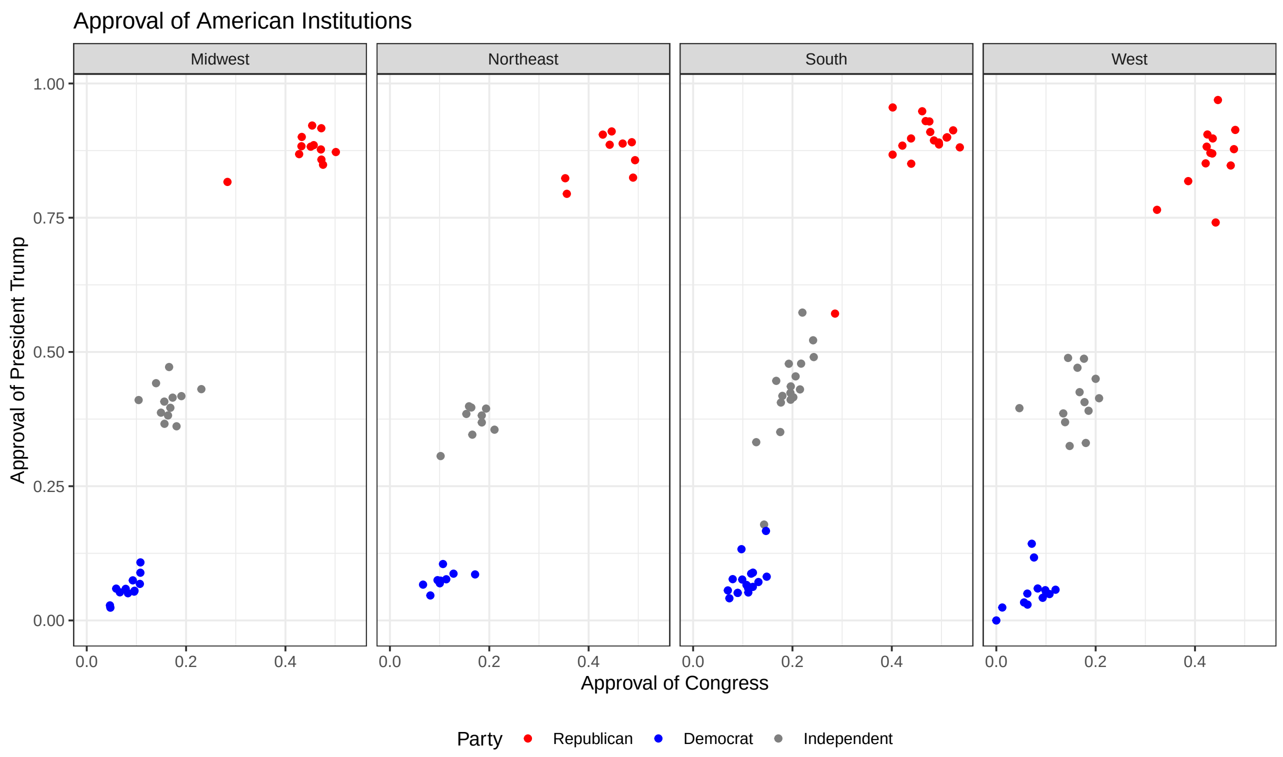
Facets allow us to split the graph pane into several sections based on a grouping variable. To do this, you can add `facet_wrap()` or `faced_grid()` with a grouping variable such that `facet_wrap(~group_var)`.

For coordinates, the most common is `coord_flip()` which is useful if you find that the graph can look better if the x became y and vice versa.

Graph Demo

To tie everything together, here is a graph that includes each of these layers. See if you can identify the various layers in use.

```
ggplot(state_party_Congress, aes(x = pctCongAppv, y = pctPresAppv, group = pid3))+
  geom_point(aes(color = pid3))+
  xlab("Approval of Congress")+
  ylab("Approval of President Trump")+
  ggtitle("Approval of American Institutions")+
  scale_color_manual(
    name = "Party",
    breaks = c("Republican", "Democrat", "Independent"),
    values = c("Republican" = "red", "Democrat" = "blue", "Independent" = "grey50")
  )+
  theme_bw()+
  theme(
    legend.position = 'bottom'
  )+
  facet_grid(~region)
```



Extras

Below, I have some extra `ggplot` tips for working with special kinds of data, including time series and spatial data. I call these "special" because sometimes they require additional packages and a bit extra work on the wrangling and visualization side to produce the best graphics. These are very basic displays of these data with an eye towards highlighting some of the neat features included in the `ggplot` package which complement standard packages used for these data and can enhance the visualization.

Dates

Disclaimer: This section on dates is a quick introduction to what R can do with dates. I might recommend a time series course to learn more tools on dealing with time data.

THE LUBRIDATE PACAKGE is a helpful package in the `tidyverse` series that is excellent for working with dates. Since this workshop is not a time series course, I will briefly introduce the notion of this package but none of the exercises require you to learn how to turn things into dates.

Below is a collection of the many possible ways that dates can be presented in uncleaned data. Notice that there are different ways that R handles the information, and none of the ways are the ones that are ideal for dates analysis.

```
date_1 <- 20210304
class(date_1)
```

```
## [1] "numeric"
```

```
date_2 <- "04mar2021"
class(date_2)
```

```
## [1] "character"
```

```
date_3 <- "March 4, 2021"
class(date_3)
```

```
## [1] "character"
```

In data, we cannot maintain the data in this format for several reasons:

- Opens the opportunity to mistreatment of variables – what might be meant as a date is instead a number
- Lack of standardization in date formats – imagine having `mar042021` and `March 04 2021` and `03 04 2021` and `03-04-2021` all in the same variable.

So `lubridate` can help us fix this problem.

```
library(lubridate)
```

In the most simplest case, we can think about the general pattern of the dates and use the appropriate `lubridate` commands to adjust them into the date format. Below, my examples deliberately bold the first letter in each date component because it coincides with the way the functions are organized and is a good way to remember them.

- `20210304` – **y**ear **m**onth **d**ay

```
ymd("20210304")
```

```
## [1] "2021-03-04"
```

- `04mar2021` – **d**ay **m**onth **y**ear

```
dmy("04mar2021")
```

```
## [1] "2021-03-04"
```

- `March 4, 2021` – **m**onth **d**ay **y**ear

```
mdy("March 4, 2021")
```

```
## [1] "2021-03-04"
```

- `20210403` – **y**ear **d**ay **m**onth

```
ydm("20210403")
```

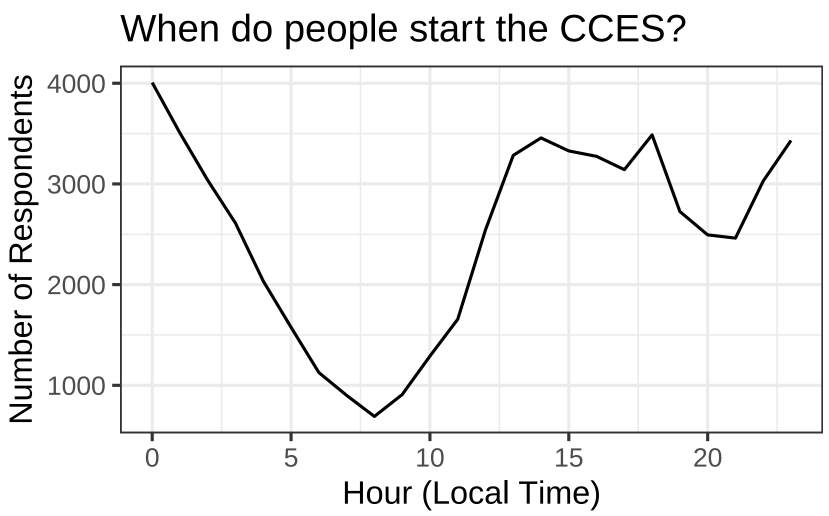
```
## [1] "2021-03-04"
```

For the start and end time variables in the CCES data, I already preprocessed the data so that it fits the date format. In this example, I am interested in the number of respondents who start the survey at each hour during the day (local time). To do this, `lubridate` allows us to extract the hour information using `hour()`. We can do this in the `mutate()` function that we learned before and create a summary table as follows:

```
Hour <- CCES_2018 %>%
  mutate(
    hour = hour(starttime)
  ) %>%
  group_by(hour) %>%
  summarise(
    n = n(),
    .groups = 'keep'
  )
```

Now, we can plot this using the `geom_line()` command

```
ggplot(Hour, aes(x = hour, y = n))+
  geom_line()+
  xlab("Hour (Local Time)")+
  ylab("Number of Respondents")+
  labs(
    title = "When do people start the CCES?"
  )+
  theme_bw()
```



Spatial Data

On Election night, we often see these infamous maps showing who wins each state or each race by shading the state or district red or blue. Outlets like the New York Times often like to go the extra mile, making these maps interactive and easy for users to use (See their 2020 Presidential Elections Map as an example).

You can create a static map pretty easily using the tools outlined in this handout and a few extra steps.

Getting the Outline

It is hard to draw lines by hand that will get the outline for each of the Florida Keys or the Great Lakes in the Midwest. Fortunately, we do not need to create the map shapes by hand. Most of the commonly used geographic boundaries that we may ever need are supplied by the US Census in the Tiger/LINE Shapefiles library. This library contains information for states, counties, census geographic boundaries, congressional districts, state legislative districts, school districts and more. The information is stored in a **shapefile**, or a dataset that contains geographic data commonly used in Geographic Information Systems (GIS) software. You can access the shapefiles in a variety of ways, including the web interface or the `tigris` R package.

On your own, if you desire to explore the point and click user interface to get shape files from the Census Bureau, your files will be downloaded as a `.zip` file with contents that might seem foreign. Most of the files here are for GIS software but the ones that you would want to pay attention to are the `.shp` and the `.prj` file. The `.shp` file contains information on the shape of the state or other geographical boundary of interest and the `.prj` file supplies information on the projection used to draw this map. For the most part, the standard projections of North American Datum 1983 (NAD83) would suffice and that is how most US government shapefiles are made.

Using the `sf` package, you can read in the `.shp` files into R as you would any other data frame using `sf::read_sf()` and R will extract all the information from the auxiliary files that it needs. For example:

This example is from the shape file folder downloaded of US states.

```
library(sf)
states <- read_sf("tl_2020_us_state.shp")
```

Don't believe that R will pull in the information it needs? Run `st_crs(states)` to get the details in the `.prj` file. You should see NAD83 on the second line after "Coordinate Reference System:".

FORTUNATELY FOR US, there is an R package that will expedite this process. Using the `tigris` R package, we can get this loaded into the R environment without worrying about file paths or what file does what.

The `tigris` package is an R package that allows us to connect with the Census API where we can access the shapefiles directly from R. The functions to get the shapefiles are intuitive so think about the geographic land area you are interested in. Some common shapes are:

Function	Description
`states()`	US States
`counties()`	US Counties
`congressional_districts()`	US Congressional Districts
`zctas()`	ZIP Codes
`tracts()`	Census Tracts
`blocks()`	Census Blocks

For the exercises in this section, run the following code, which will give us the shapefile for counties in Illinois. Be sure to install the package using `install.packages("tigris")`.

```
library(tigris)
IL_county <- counties("IL", cb = TRUE)
```

Notice that the function takes an abbreviation for states, but it can also take the Federal Information Processing Standards (FIPS) code, which, for states, is a two digit number that identifies the state. This is in the `inputstate` variable in the CCES dataset and it is what we will use to proceeded.

Mapping with R

To create a dataset that we can use for mapping, I am going to plot the number of respondents from each state in Illinois. To do this, I filter the data to include IL respondents and group by the country FIPS code variable to calculate the number of respondents using `summarise()`.

```
IL <- CCES_2018 %>% filter(state == "IL") %>%
  group_by(countyfips) %>%
  summarise(
    n = n(),
    .groups = 'keep'
  )
```

To put everyone in the same dataset, I use `left_join()` which merges the shapefile and the data together. Since I did not go over this function here, run this code but look to the documentation if you are curious.

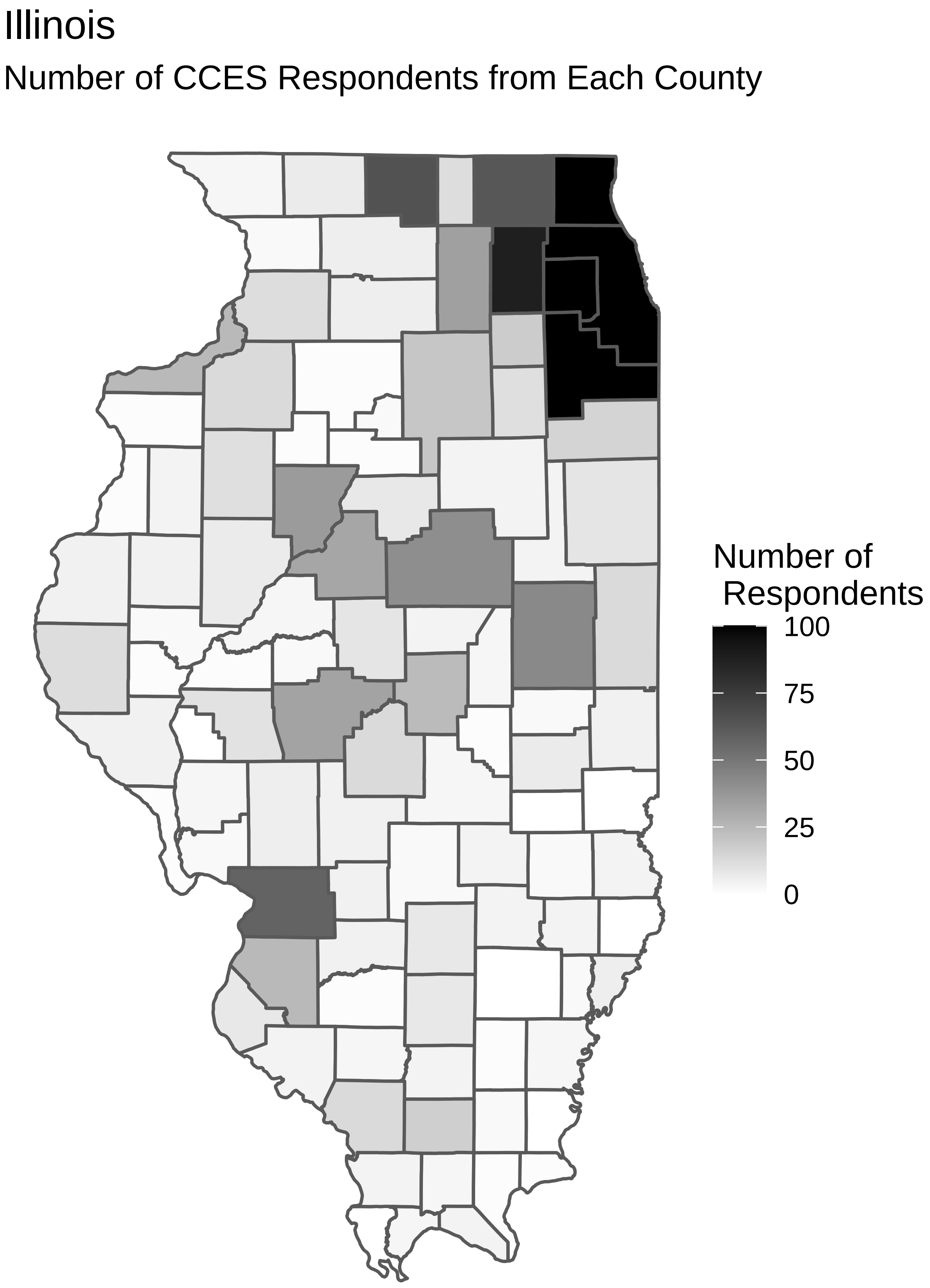
```
IL_respondents <- left_join(
  IL_county,
  IL,
  by = c("GEOID" = "countyfips")
) %>%
  mutate(
    respondents = ifelse(is.na(n), 0, n)
  )
```

To draw the map, I need to load the `sf` package, which contains helpful tools to help us navigate the `geom_sf()` function in `ggplot`.

```
library(sf)
```

Now, we can draw the map. Notice that in this example, I am setting the data layer to be empty and instead adding this information in the graphs layer. Here is another option to navigate `ggplot` especially if you might be working with multiple kinds of graphs or multiple datasets in the same plot.

```
ggplot()+
  geom_sf(data = IL_respondents, aes(fill = respondents))+
  scale_fill_gradient(
    low = "gray100",
    high = "gray0",
    na.value = "black",
    limits   = c(0, 100)
  )+
  labs(
    title    = "Illinois",
    subtitle = "Number of CCES Respondents from Each County",
    fill     = "Number of \n Respondents"
  )+
  theme_void()
```


Illinois
Number of CCES Respondents from Each County
Number of Respondents
100
75
50
25
0