Book of Workshops

April 13, 2025

Table of contents

Preface

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

```
1 + 1
```

```
[1] 2
```

Part I

Getting Started with Git and GitHub

Introduction

03/27/2025

Introduction

In this workshop we delve deeper into the domain specific language of statistical graphics that underpins the `tidyverse` `ggplot2` package syntax: the "Grammar of Graphics". We will explore each discrete grammar layer using laboratory-confirmed RSV hospitalizations data collected by the CDC's Respiratory Virus Hospitalization Surveillance Network (RESP-NET) surveillance program.

With a better understanding of the syntax fundamentals, we will then get introduced to some advanced uses of `ggplot2` that are commonly used in public health:

- Making plots interactive with `plotly`
- Projecting data to a map

We will close the workshop by asking Yale's Clarity Platform to reproduce our code from the plot image alone to exhibit how AI can be used to support data visualization work. Clarity is an AI chatbot that offers similar functions to OpenAI's ChatGPT and Microsoft Copilot with additional data protection. Find out more about Clarity's security guidelines on "AI at Yale".

The cleaned and harmonized version of the RSV-NET dataset was compiled as part of the YSPH's very own PopHIVE project. Special thanks to Professor Daniel Weinberger for allow us to adopt his plot code in this workshop.

Accessing the Materials

Slides, Handouts, and Other Materials

Download the complete slide deck with annotations and the in-person workshop handout. Comments were saved in the bottom left of each slide, and references for this webpage are in its **Appendix**.

Codespaces

In this workshop you will need to access the R code we have prepared for the worked through example and challenge questions. If you have not already, you will need to download R to your local device, and we suggest using the integrated development environment (IDE) software RStudio. Accessing the code for this workshop requires that you have git installed on your local device, a GitHub account, and you have configured the two. If you have not done this, go through Accounts and Configurations first.

> **!** Important
>
> This workshop was generated using R (v 4.4.3) in the RStudio IDE (v 2024.12.1+563). `renv()` is included to reproduce the same coding enviroment, storing all the relevant packages and package versions needed in the code. If you experience trouble running the scripts, you might want to check that the environment was initialized and that you are using the same version of R and RStudio.

Two GitHub repositories have been created to practice using git and GitHub:

- Solo projects: ysph-dsde/JHU-CRC-Vaccinations
- Group projects: ysph-dsde/JHU-CRC-Cases-and-Deaths

In order to practice your skills with git and GitHub using our codespaces, you will need to create a "clean-break" copy of both repositories. This will fully decouple the codespace connections from the ysph-dsde GitHub accout, and allow you full access to its contents. There are two methods to do this: by the "GitHub Importer" tool or your command-line application (i.e. Terminal for Macs and Windows Terminal for windows).

Below we have detailed how you can create a "clean-break" by both methods. We suggest you attempt the "GitHub Importer" tool option first, and if that fails to follow the command-line steps. Please note that the importer tool will sometimes take a few minutes to fully transfer over the files.

Making a Clean-Break Copy

METHOD 1: Copying Using GitHub Importer

> **i** Note
>
> This method is not a Fork. You can learn more about GitHub Importer here.

1. Log in to your personal GitHub account.

2. In the top-right of the page navigation bar, select the dropdown menu and click Import repository.
3. Fill out the following sections:
    a. **Your source repository details:** Paste the https url of the repositories listed above. No credentials are required for this action.
    b. **Your new repository details:** Adjust the GitHub account owner as needed and create the name for the new repository. It is good practice to initially set the repository to "Private".
4. Click the Begin import button to copy the codespace.
5. After a few minutes, the newly created GitHub repository webpage will open up.

If this method is successful, then proceed to the Cloning the Copied Repository section. If this is not successful, you can try using the command-line application menthod detailed in Method 2.

METHOD 2: Copying Using The Command-Line Application

These directions follow GitHub's duplicating a repository page.

1. Log in to your personal GitHub account.
2. Navigate to the ysph-dsde GitHub repository you want to copy by either searching for it by name or opening the url provided above.
3. Near the right side of the page there will be a Begin import button to click. In its drop down menu under the "Local" tab you will see options to copy the SSH key or HTTPS url to the repository.

    For example, if the repository name is "ORIGINAL-REPOSITORY" they will look like:

    ```
    # SSH
    git@github.com:ysph-dsde/ORIGINAL-REPOSITORY.git
    ```

    ```
    # HTTPS
    https://github.com/ysph-dsde/ORIGINAL-REPOSITORY.git
    ```

    Depending on your Git/GitHub configurations, you will copy one of these for the remainder of the steps.

> **!** Important
>
> SSH keys or HTTPS urls are file transfer protocols that are used to pass information between your local git configured directory to the remote GitHub repository. Only one protocol can be set up for one Git/GitHub connection.

4. Open the command-line application (i.e. Terminal for Macs and Windows Terminal for windows) and navigate to the file location you want to temporarily store the repository copy.

```
cd "/file_location/"
```

5. Clone a bare copy of the original repository using its SSH key or HTTPS url:

```
 # SSH
 git clone --bare git@github.com:ysph-dsde/ORIGINAL-REPOSITORY.git
```

```
# HTTPS
git clone --bare https://github.com/ysph-dsde/ORIGINAL-REPOSITORY.git
```

6. Open the project file.

```
cd "ORIGINAL-REPOSITORY.git"
```

7. Back in GitHub, in the top-right of the page navigation bar select the dropdown menu and click New repository.

8. Fill out the following sections:

   a. Adjust the GitHub account owner as needed and create the name for the new repository.

   b. It is good practice to initially set the repository to "Private".

   c. Do **NOT** use a template or include a description, `README.md`, `.gitignore`, or license.

9. In the newly created GitHub repository under "Quick setup" you will find the repository's SSH key or HTTPS url. Copy this.

10. Back in the command-line application, push a mirror of the cloned git file to your newly created GitHub repository using its SSH key or HTPPS url:

```
# SSH
git push --mirror git@github.com:EXAMPLE-USER/NEW-REPOSITORY.git
```

```
# HTTPS
git push --mirror https://github.com/EXAMPLE-USER/NEW-REPOSITORY.git
```

Refresh the new GitHub reposiotry webpage to confirm the push was successful.

11. Delete the bare cloned file used to create a new remote repository.

```
cd ..                                   # Go back one file location
rm -rf ORIGINAL-REPOSITORY.git          # Delete the bare clone
```

12. This completes creating a clean-break copy of the ysph-dsde repository codespace. Proceed with cloning the newly made repository to your local device in the following section.

Cloning the Copied Repository

Now that you have copied this repository into your own GitHub, you are ready to proceed with a standard clone to your local device.

1. Copy the SSH key or HTTPS url to the newly created repository in your GitHub account by finding the codes under the Begin import button.

```
# SSH
git@github.com:ysph-dsde/NEW-REPOSITORY.git
```

```
# HTTPS
https://github.com/ysph-dsde/NEW-REPOSITORY.git
```

2. In the command-line application (i.e. Terminal for Macs and Windows Terminal for windows) navigate to the file location you want to store the repository.

```
cd "/file_location/"
```

3. Clone the the repository.

```
# using SSH
git clone git@github.com:EXAMPLE-USER/NEW-REPOSITORY.git

# or using HTTPS
git clone https://github.com/EXAMPLE-USER/NEW-REPOSITORY.git
```

4. **OPTIONAL:** You can reset the repository history, which will clear the previous commits, by running the following block of code (Source: StackExchange by Zeelot).

```
git checkout --orphan tempBranch          # Create a temporary branch
git add -A                                # Add all files and commit them
git commit -m "Reset the repo"
git branch -D main                        # Deletes the main branch
git branch -m main                        # Rename the current branch to main
git push -f origin main                   # Force push main branch to GitHub
git gc --aggressive --prune=all           # Remove the old files
```

Initializing the Environment

After cloning the codespace to your local device, you will need to initialize the environment using `renv()`. This will install all packages and versions used in the workshop, thus creating a reproducible coding environemnt.

1. In the command-line application (i.e. Terminal for Macs and Windows Terminal for windows) navigate to the file location you want to store the repository.

   ```
   cd "/file_location/"
   ```

2. Launch the project by opening the `*.Rproj` in RStudio.

3. In the R console, activate the enviroment by runing the following lines of code:

   ```
   renv::init()           # initialize the project
   renv::restore()        # download packages and their version saved in the lockfile.
   ```

> **Note**
>
> If you are asked to update packages, say no. The `renv()` is intended to recreate the same environment under which the project was created, making it reproducible. You are ready to proceed when running `renv::restore()` gives the output:
>
> ```
> - The library is already synchronized with the lockfile.
> ```
>
> If you experience any trouble with this step, you might want to confirm that you are using R (v 4.4.3) in the RStudio IDE (v 2024.12.1+563). You can also read more about `renv()` in their vignette.

1 Accounts and Configurations

2 Worked Through Example

Solo Projects and Group Collaborations

This is a book created from markdown and executable code.

See @knuth84 for additional discussion of literate programming.

```
1 + 1
```

```
[1] 2
```

```
# NOTE: renv initializing might need to be run twice after the repo is
#       first copied.
#renv::init()
renv::restore()

suppressPackageStartupMessages({
  library("dplyr")         # For data manipulation
  library("ggplot2")       # For creating static visualizations
  library("plotly")        # For interactive plots
  library("RColorBrewer")  # Load Color Brewer color palettes
  library("viridis")       # Load the Viridis color pallet
})

# Function to select "Not In"
'%!in%' <- function(x,y)!('%in%'(x,y))
```

3 References

Part II

Data Visualization with ggplot2

Introduction

03/27/2025

Introduction

In this workshop we delve deeper into the domain specific language of statistical graphics that underpins the `tidyverse` `ggplot2` package syntax: the "Grammar of Graphics". We will explore each discrete grammar layer using laboratory-confirmed RSV hospitalizations data collected by the CDC's Respiratory Virus Hospitalization Surveillance Network (RESP-NET) surveillance program.

With a better understanding of the syntax fundamentals, we will then get introduced to some advanced uses of `ggplot2` that are commonly used in public health:

- Making plots interactive with `plotly`
- Projecting data to a map

We will close the workshop by asking Yale's Clarity Platform to reproduce our code from the plot image alone to exhibit how AI can be used to support data visualization work. Clarity is an AI chatbot that offers similar functions to OpenAI's ChatGPT and Microsoft Copilot with additional data protection. Find out more about Clarity's security guidelines on "AI at Yale".

The cleaned and harmonized version of the RSV-NET dataset was compiled as part of the YSPH's very own PopHIVE project. Special thanks to Professor Daniel Weinberger for allow us to adopt his plot code in this workshop.

Accessing the Materials

Slides, Handouts, and Other Materials

Download the complete slide deck with annotations and the in-person workshop handout. Comments were saved in the bottom left of each slide, and references for this webpage are in its **Appendix**.

Codespaces

In this workshop you will need to access the R code we have prepared for the in-workshop discussion and after-workshop challenge questions. This assumes that you have downloaded R and RStudio to your local device. After you download the code, you will need to:

1. Move the unzipped directory to the file location you wish to house the project.
2. Open `Data-Visualization-with-ggplot2.Rproj`.
3. Open `Discussion and Challenge Questions.R`.
4. Initialize the environment in the R console by running:

```
renv::init()              # initialize the project
renv::restore()           # download packages and their version saved in the lockfile.
```

NOTE: If you are asked to update packages, say no. The `renv()` is intended to recreate the same environment under which the project was created, making it reproducible. You are ready to proceed when running `renv::restore()` gives the output:

```
- The library is already synchronized with the lockfile.
```

You can read more about `renv()` in their vignette. In this workshop you will need to access the R code we have prepared for the worked through example and challenge questions. If you have not already, you will need to download R to your local device, and we suggest using the integrated development environment (IDE) software RStudio. Accessing the code for this workshop requires that you have git installed on your local device, a GitHub account, and you have configured the two. If you have not done this, go through Accounts and Configurations first.

> **!** Important
>
> This workshop was generated using R (v 4.4.3) in the RStudio IDE (v 2024.12.1+563). `renv()` is included to reproduce the same coding enviroment, storing all the relevant packages and package versions needed in the code. If you experience trouble running the scripts, you might want to check that the environment was initialized and that you are using the same version of R and RStudio.

Initializing the Environment

After cloning the codespace to your local device, you will need to initialize the environment using `renv()`. This will install all packages and versions used in the workshop, thus creating a reproducible coding environemnt.

1. In the command-line application (i.e. Terminal for Macs and Windows Terminal for windows) navigate to the file location you want to store the repository.

   ```
   cd "/file_location/"
   ```

2. Launch the project by opening the `*.Rproj` in RStudio.

3. In the R console, activate the enviroment by runing the following lines of code:

   ```
   renv::init()           # initialize the project
   renv::restore()        # download packages and their version saved in the lockfile.
   ```

> **Note**
>
> If you are asked to update packages, say no. The `renv()` is intended to recreate the same environment under which the project was created, making it reproducible. You are ready to proceed when running `renv::restore()` gives the output:
>
> ```
> - The library is already synchronized with the lockfile.
> ```
>
> If you experience any trouble with this step, you might want to confirm that you are using R (v 4.4.3) in the RStudio IDE (v 2024.12.1+563). You can also read more about `renv()` in their vignette.

4 Worked Through Example

4.1 Environment Set-Up and Data Description

First, we will load the necessary libraries and any special functions used in the script.

```
# NOTE: renv initializing might need to be run twice after the repo is
#       first copied.
#renv::init()
renv::restore()

suppressPackageStartupMessages({
  library("arrow")          # For reading in the data
  library("dplyr")          # For data manipulation
  library("ggplot2")        # For creating static visualizations
  library("plotly")         # For interactive plots
  library("cowplot")        # ggplot add on for composing figures
  library("tigris")         # Imports TIGER/Line shapefiles from the Census Bureau
  library("sf")             # Handles "Special Features": spatial vector data
  library("RColorBrewer")   # Load Color Brewer color palettes
  library("viridis")        # Load the Viridis color pallet
})

# Function to select "Not In"
'%!in%' <- function(x,y)!('%in%'(x,y))
```

Now we will import our cleaned and tidy data, which is ready for plotting. Students who would like to find out more about how to get their data into the plottable, tabular form you will see here can explore our A Journey into the World of tidyverse workshop.

> **Code Like a Pro**
>
> Our data was stored as a parquet file. Para-what? Parquet is a column-oriented data file that allows for efficient data storage and lightweight information retrieval. It is best suited for large data sets that cannot be easily handled "in-memory". Using the `arrow` package, we can read and manipulate files in this form.

> Those interested to learn more about how they can use parquet to efficiently process large datasets are encouraged to review the workshops hosted by one of our guest speakers, Professor Thomas Lumley: Thomas Lumley Workshops.

```
df <- read_parquet(file.path(getwd(), "RSV-NET Infections.gz.parquet"))

# glimpse() allows us to see the dimensions of the dataset, column names,
# the first few entries, and the vector class in one view.
df |> glimpse()
```

```
Rows: 92,519
Columns: 14
$ Region                  <chr> "California", "California", "California", "Cal~
$ Season                  <chr> "2018-19", "2018-19", "2018-19", "2018-19", "2~
$ `Week Observed`         <date> 2018-10-06, 2018-10-13, 2018-10-20, 2018-10-2~
$ MMWRyear                <dbl> 2018, 2018, 2018, 2018, 2018, 2018, 2018, 2018~
$ MMWRweek                <dbl> 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25~
$ MMWRday                 <dbl> 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7~
$ Characteristic          <chr> "Age", "Age", "Age", "Age", "Age", "Age", "Age~
$ Level                   <chr> "1-4 Years", "1-4 Years", "1-4 Years", "1-4 Ye~
$ `Positives Detected`    <dbl> 0, 0, 3, 10, 8, 8, 3, 6, 8, 15, 15, 25, 23, 22~
$ `Scaled Positives`      <dbl> 0.000000, 0.000000, 3.896104, 12.987013, 10.38~
$ Spline                  <dbl> 0.000000, 2.184079, 5.738394, 8.770741, 9.5923~
$ Kernel                  <dbl> 0.000000, 2.897880, 6.721988, 9.740228, 10.945~
$ `Crude Rate`            <dbl> 0.0, 0.0, 0.6, 2.5, 1.9, 1.9, 0.6, 1.3, 1.9, 3~
$ `Cumulative Crude Rate` <dbl> 0.0, 0.0, 0.6, 3.1, 5.0, 6.9, 7.6, 8.8, 10.7, ~
```

This workshop uses the Center for Disease Control's (CDC) Respiratory Syncytial Virus Hospitalization Surveillance Network (RSV-NET) surveillance data. It is one of the CDC's Respiratory Virus Hospitalization Surveillance Network (RESP-NET) Emerging Infections Programs (EIP). This version was downloaded from data.gov in January of 2025.

RSV-NET conducts active, population-based surveillance for laboratory-confirmed RSV-associated hospitalizations. It contains stratification for geolocation, race/ethnicity, age, and sex. The cleaned and harmonized version of the RSV-NET dataset was compiled as part of the YSPH's very own PopHIVE project. You will see that its contents differ slightly from what you would see on the data.gov website.

Description of the variables:

- **Region** - geolocation at the state and 10 Health and Human Services Regions (HHS) level.

- **Season** - the infection season of the observation, defined to span from July to June of the following year.
- **Week Observed** - the week when the record was added, in `Date` format.
- **MMWRyear**, **MMWRweek**, **MMWRday** - the year, week, and day of the entry in Morbidity and Mortality Weekly Report (MMWR) format. MMWR is the standard epidemiological week assigned by National Notifiable Diseases Surveillance System (NNDSS) in their disease reports (MMWR Definition).
- **Characteristic** and **Level** - the type of additional stratification and the group represented, respectively. i.e. "Characteristic = Sex; Level = Female".
- **Positives Detected** - the number of lab-confirmed RSV hospitalizations detected.
- **Scaled Detected** - the relative scale of RSV infections detected for inter-season comparison, controlling for each stratification. Outcomes were grouped by Region and Level then scaled using Positives Detected/max(Positives Detected) $*$ 100.
- **Spline** - `smooth.spline(MMWRweek, Scaled Positives)` from the `stats` package. Parameters were automatically optimized by the algorithm as this is only for smoothing the trendline.
- **Kernel** - `locfit(Scaled Positives ~ lp(MMWRweek, nn = 0.3, h = 0.05, deg = 2))` from the `locfit` package. No rigorous method was applied to optimize parameters as this is only for smoothing the trendline.
- **Crude Rate** - the number of residents in a surveillance area who are hospitalized with laboratory-confirmed RSV infections divided by the total population estimated for that area per 100,000 persons.
- **Cumulative Crude Rate** - the crude rate added cumulatively over the course of one infection season.

4.2 Basic Uses of `ggplot2()`

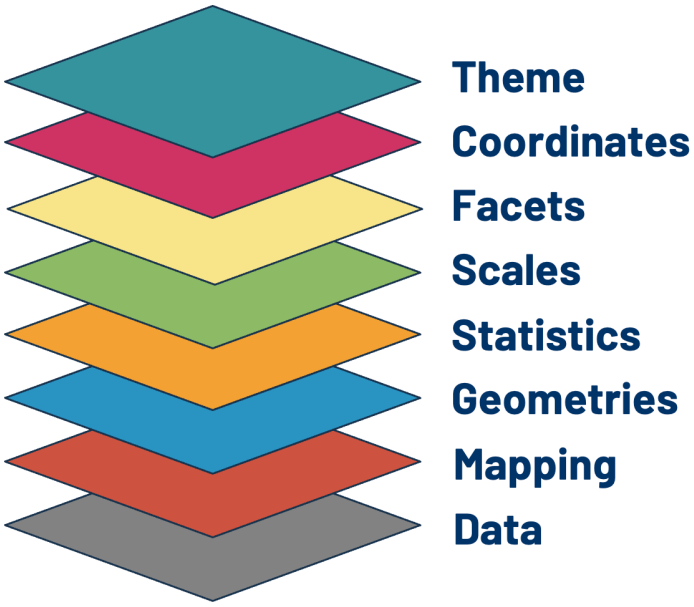


Figure 4.1: Figure from "ggplot2 workshop part 1" by Thomas LinPedersen. Accessed from YouTube March 15th, 2025.

4.2.1 The Data and Mapping Layers

The first (and arguably most crucial!) layer is **Data**, but simply adding it to `ggplot()` will only generate a plot object with nothing else. In the second layer, **Mapping**, we tell the function which variables get used for which aesthetic feature displayable on the plot object.

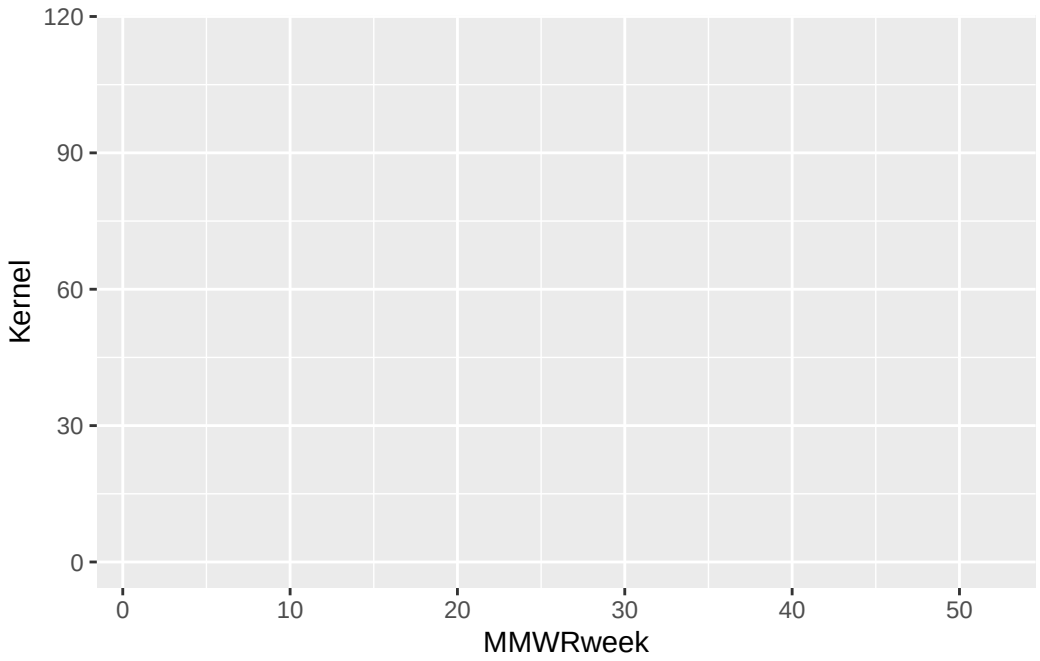
In this layer, we define the position, color, size, or shape that our values take. Every type of data representation requires an aesthetic statement to point variables to relevant aesthetic features. This can help make a plot visually appealing, but an astute user of `ggplot()` will leverage them to highlight underlying patterns in the data as well.

Here we use the mapping function, `aes()`, to point the week in the epidemiological year (MMWR) variable to the x-axis and RSV positive tests (scaled and Gaussian kernel smoothed) to the y-axis.

> **!** Geometry and Statistic Layers Dictate the Aesthetics Mapping
>
> As we will see in the next two layers, the kinds of features that are required to generate the most fundamental visualization of our data will depend on the type of geometry or

statistical transformation we are plotting. **Data** and **Mapping** must always be defined by the user. Some `geom_*()` or `stat_*()` functions also require the plot-specific statistic or position to be provided by the user, but sometimes these elements are already assigned by the default settings.
You can find more about the required and ancillary aesthetic features used by different **Geometry** or **Statistics** layers in the `ggplot2` package references documentation and cheatsheet. Different specifications available for aesthetic features can also be found in one of their vignettes: aesthetic specifications.

```
# The data and aesthetics mapping layers combined.
ggplot(data = df, aes(x = MMWRweek, y = Kernel))
```



! Important

We can use the base R pipe symbol `|>` to pass objects into subsequent functions that build on the previous result. This has the added benefit of organizing operations into digestible pieces, as opposed to gnarly nested functions. That would not be so knarly, man!
Going forward we will no longer explicitly dictate that `data = df`, as this is already implied by the pipe operation. Just keep in mind that the start of a graphical expression begins with a `ggplot() +` statement, and everything preceding this is not part of the graphical layers.

4.2.2 The Geometry Layer

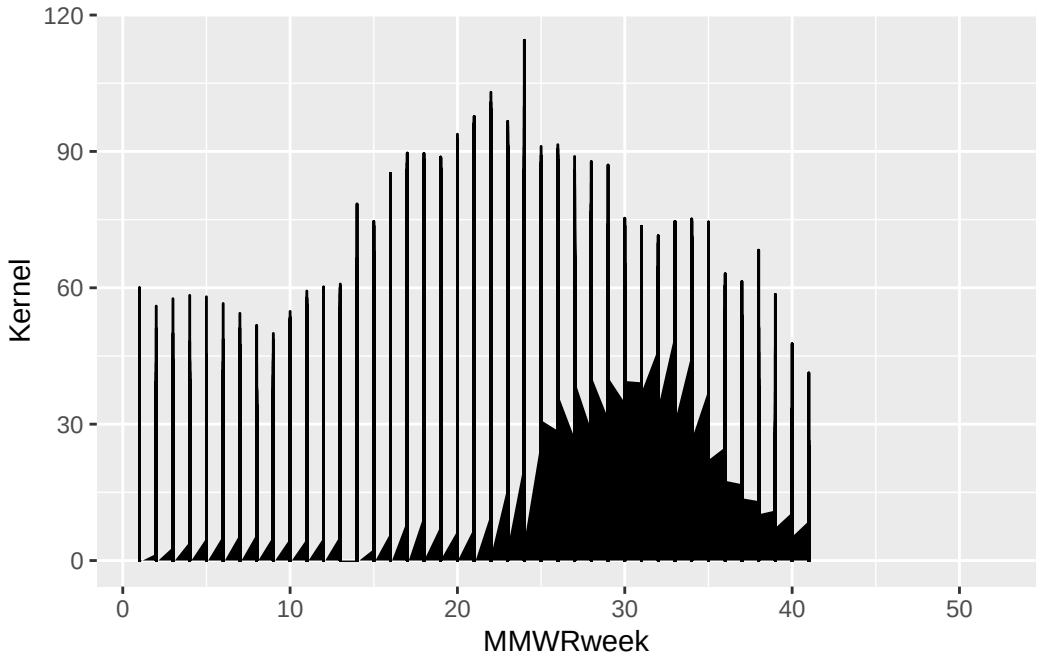
We will now tell the function how we want the data to be displayed in the plot object. We do this by adding the **Geometry** layer. Together with **Data** and **Mapping**, these three layers form the core pieces required to generate any basic visualization with `ggplot2`.

The **Geometry** layer does the computational legwork that translate a data frame with mappings into a discernible plot. Behind the scenes, it is computing how points get spaced and oriented to create the specific geometry defined, whether that be for a histogram, scatter plot, box plot, and so forth.

Each geometry layer can take a specific data and aesthetics mapping, overwriting anything defined in the `ggplot()` plot object. It also interprets a statistical transformation and position adjustment to modulate the graph. Here, we will plot a trend line showing changes in RSV positives detected over the course of an infection season by using `geom_line()`.

> **Note**
>
> In some `ggplot2` references statistics and position are both defined as separate layers in the hierarchy. This workshop separates out statistics, but does not separate out position its own layer.
> Position sets the rules around how objects get placed relative to one another; a setting necessary within some `geom_*` functions. Think of plotting a histogram scaled with an aesthetic color mapping. Do the groupings plot on top of each other, side-by-side, or get scaled to fill [0, 1] on the y-axis? This is what the position setting will determine.

```
# All three required layers combined: data, aesthetics mapping, and geometry.
df |>
  ggplot() +
    geom_line(aes(x = MMWRweek, y = Kernel))
```



Look at this plot, I mean oy vey! If we think back to the glimpse of our dataset we might remember that there are additional columns of information that we can use to further distinguish one infections trend.

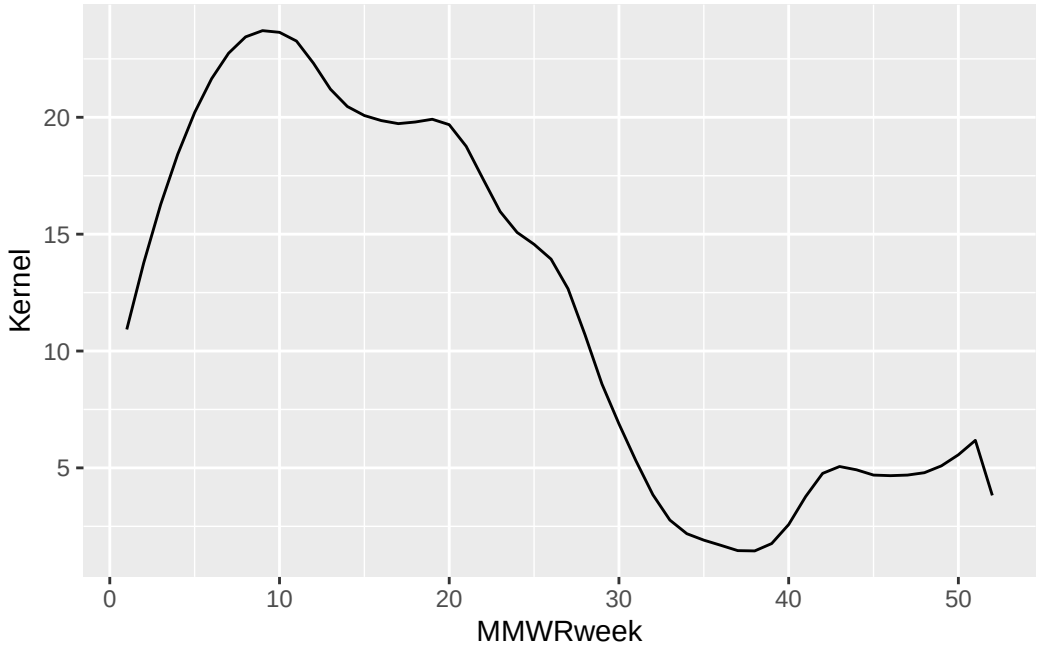
```
$Region
 [1] "California"     "Colorado"       "Connecticut"    "Georgia"
 [5] "Maryland"       "Michigan"       "Minnesota"      "New Mexico"
 [9] "New York"       "North Carolina" "Oregon"         "Region 1"
[13] "Region 10"      "Region 2"       "Region 3"       "Region 4"
[17] "Region 5"       "Region 6"       "Region 8"       "Region 9"
[21] "Tennessee"      "Utah"

$Season
[1] "2016-17" "2017-18" "2018-19" "2019-20" "2020-21" "2021-22" "2022-23"
[8] "2023-24" "2024-25"

$`Characteristic Level`
 [1] "1-4 Years"                      "18-49 Years"
 [3] "5-17 Years"                     "50-64 Years"
 [5] "65-74 Years"                    "75+ Years"
 [7] "<1 Years"                       "American Indian or Alaska Native"
 [9] "Asian or Pacific Islander"      "Black or African American"
[11] "Female"                         "Hispanic or Latino"
[13] "Male"                           "N/A"
```

```
[15] "White"
```

```
df |>
  # Subset the data to one infection trend outcome by selecting one possible
  # instantiation from the array of distinguishing variables.
  filter(Region == "Connecticut", Season == "2021-22", Level == "N/A") |>
  ggplot() +
    geom_line(aes(x = MMWRweek, y = Kernel))
```



This looks much nicer, but it doesn't tell us much about our data. Notice, however, that the same plot can be generated by adding the aesthetics mapping to the plot object (`ggplot()`) or the `geom_*()` layer.

```
# A viable alternative representation of the above code:
df |>
  filter(Region == "Connecticut", Season == "2021-22", Level == "N/A") |>
  ggplot(aes(x = MMWRweek, y = Kernel)) +
    geom_line()
```

Discussion: Can you think of reasons why you would choose one over the other?

You might be asking yourself why I went through the pains of showing the thought process behind tuning the plot so that it shows one distinguishable trend line. Am I not simply

covering the grammar of whatever to plot a simple trend line while you eat a delicious cookie and compartmentalize your upcoming exams?

As I hinted at earlier, the advanced user of `ggplot2` is going to leverage these layers for effective visualization of underlying patterns of the data itself. By controlling for variables that add noise or redundancies to a plot, we are opening the door for inspiring communicating insights through visualization.

Recall:

> "... graphics are instruments of reasoning about quantitative information" – Yale Professor Edward Tufte from his book *The Visual Display of Quantitative Information*

More on this in a moment. But while I have your attention, those cookies are delicious; this is objectively and observably true.

4.2.3 The Statistics Layer

Following the **Geometries** layer in our hierarchy is **Statistics**. In this context, statistics refers to the transformations applied to data primarily for the purposes of generating plottable values. For example, calculating box plot quartiles or bar plot counts.

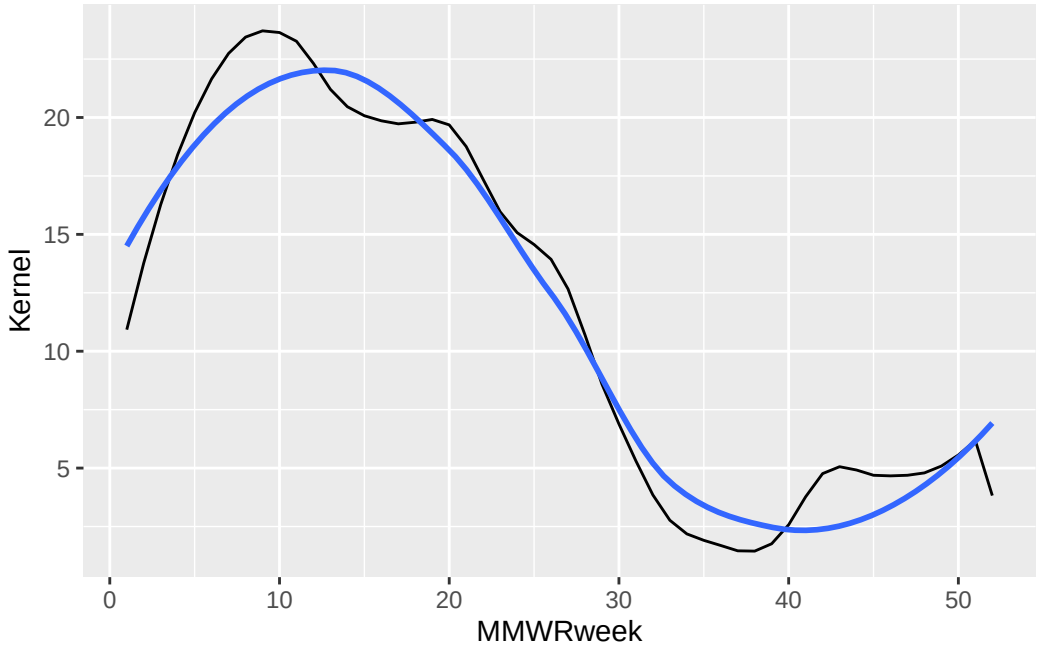
These same transformations are being used under the hood of the `geome_*()` functions. As you might expect, many statistics functions are interchangeable with geometric functions. For example, one can use `geom_bar(stat = "count")` or `stat_count(geom = "bar")` to produce the same bar plot (`ggplot2` Layer statistical transformations - Paired geoms and stats).

Keep in mind, however, that this is not always true, and sometimes statistical functions are sub-components of composite geometric engines. One example being the `stat_ydensity()` function, which generates the smoothed lines along the y-axis of a violin plot.

> **!** Important
>
> If you want to use a statistically transformed variable for an aesthetic like fill or color, you need to use `after_stat()` around the variable name to scale the mapped aesthetic properly. An example provided in the `ggplot2` cheatsheet is: `ggplot(df) + stat_density_2d(aes(fill = after_stat(level)), geom = "polygon")`.

In our example I had separately smoothed the "Scaled Positives" variable using a spline or Gaussian kernel. If I plot a `ggplot2` statistic layer that smooths the same variable using the locally estimated scatter plot smoothing (LOESS) method we see it aligns closely with the Gaussian kernel result.

```r
df |>
  filter(Region == "Connecticut", Season == "2021-22", Level == "N/A") |>
  ggplot() +
    # Values smoothed before plotting using Gaussian kernel.
    # Plotted as the black line.
    geom_line(aes(x = MMWRweek, y = Kernel)) +
    # Values smoothed by the stat_smooth() ggplot function using LOESS.
    # Plotted as the blue line.
    stat_smooth(aes(x = MMWRweek, y = `Scaled Positives`),
                geom = "smooth", method = "loess",
                se = FALSE)
```



Code Like a Pro

When a `stat_*()` is applied to a data frame (either inside a **Geometric** or **Statistic** layer), a modified data frame is generated as the output containing new, transformed variables. It is possible to call these "generated variables" within the plot object and use them for graph tuning.
One example provided in *ggplot2: Elegant Graphics for Data Analysis*, shows how to call the `stat_density()` "generated variable", density, created when the data frame is passed through `geom_histogram()`: Chapter 13 Generated variables.

- `ggplot(df, aes(x)) + geom_histogram(binwidth = 500)` plots a histogram

with counts as the y-axis.
- `ggplot(df, aes(x)) + geom_histogram(aes(y = after_stat(density)), binwidth = 500)` plots that same histogram with the calculated distribution density as the y-axis.

4.2.4 The Scales Layer

In the same way the **Statistics** layer defines the statistical transformations happening under the hood of the geometric functions, **Scales** is what interprets an aesthetic **Mapping** into plottable values. Once the aesthetics are defined by the user, the `ggplot2` algorithm automatically applies the necessary scales. Unless we want to customize the scaling, the user does not need to explicitly write out the default scales needed to generate a plot.

For example, `scale_*_continuous()` gets automatically applied to our plot because of the x and y aesthetics mappings we've defined. The code snippet below explicitly writes out these scaling functions that get used, but because we are not modifying their defaults it is not necessary to include them in our own code.

```
# The automatically applied scale_*() functions to the x and y mapping.
df |>
  filter(Region == "Connecticut", Season == "2021-22", Level == "N/A") |>
  ggplot(aes(x = MMWRweek, y = Kernel)) +
    geom_line() +
    # The following two lines are not required for us to include since
    # they are implied by the type of x and y vectors we assigned.
    scale_x_continuous() +
    scale_y_continuous()
```

One of the greatest advantages to `ggplot2` is it offers users a great deal of flexibility. This concept is especially true for the **Scales** layer. While the preceding layers offer some level of customization, the **Scales** layer increases the available options by adding on top of previous settings. This cross-layer communication makes for a seemingly endless array of options and can make it one of the hardest layers to master.

Most customization options fall into two buckets:

- Position scales and axes (Chapter 10 of *ggplot2: Elegant Graphics for Data Analysis*)
- Color scales and legends (Chapter 11 of the same book)

The different applications of **Scales** covered in these chapters range from canonical axes scaling using `scale_*_log10()`, `scale_*_sqrt()`, and `scale_*_reverse()` to tuning axis breaks and labeling or from scaling color pallets to cleaning the legend. The two chapters I've cited here

from *ggplot2: Elegant Graphics for Data Analysis* by Hadley Wickham, Danielle Navarro and Thomas Lin Pedersen are invaluable introductions to different applications of **Scales**.

In the **Scales** layer of our example, we will use `labs()` to customize the plot labels and set the y-axis limits to span the entire range of our variable, $[0, 100]$. It is also possible to assign customized color pallets to grouped outcomes. Say that for our example we want to compare RSV infection trends between seasons going back three years to 2022, including the current infections season. How do we modify our running plot example to do this?

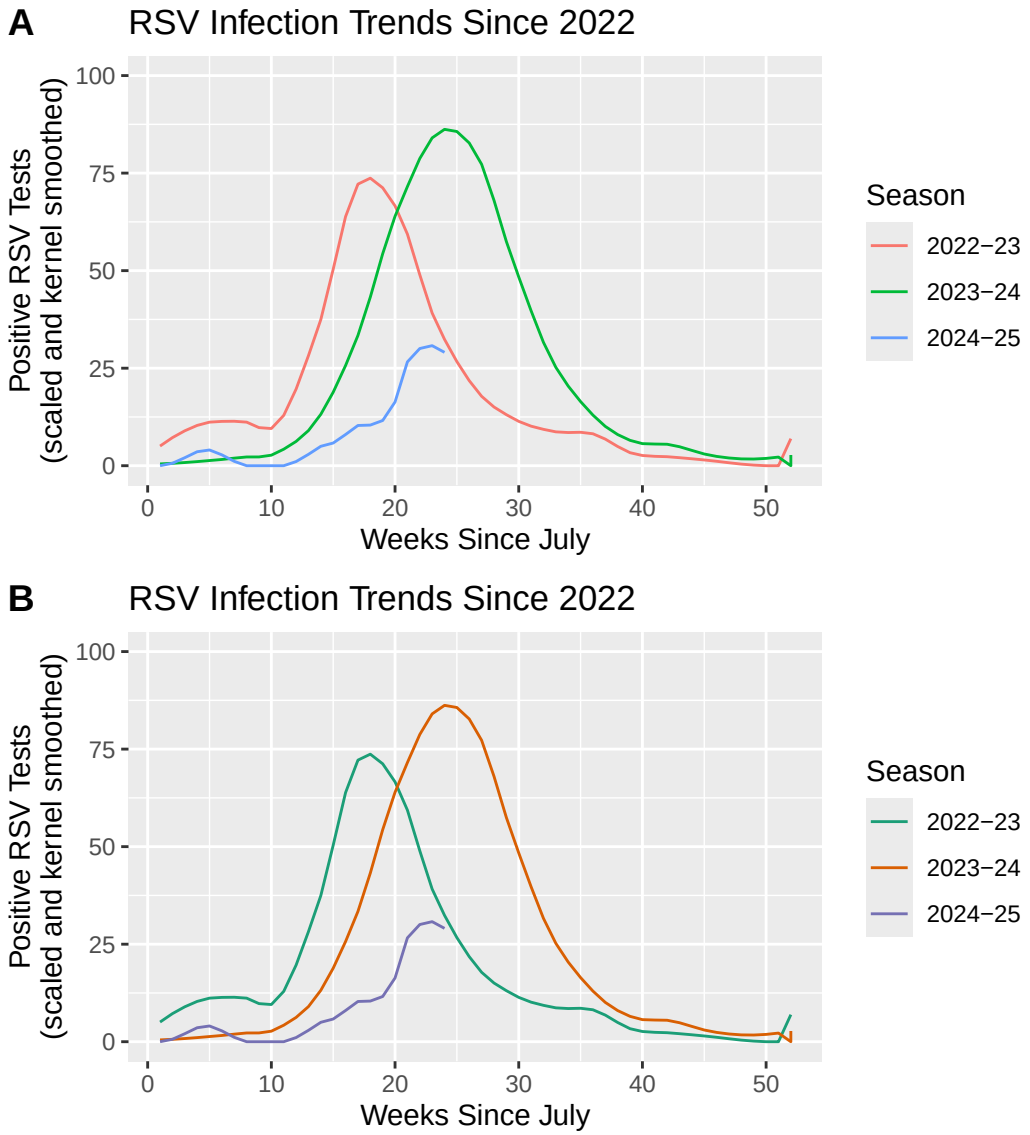
Steps:

1. Add an aesthetic **Mapping** that prompts the plot object to group the unique seasons represented and associate them with a color.
2. Add a `scale_color_*` function that will apply a customized color to the **Scales** layer instead of the default palette.

```
# Specify the infection seasons we'd like to see out of the range of options.
include_seasons <- c("2022-23", "2023-24", "2024-25")

# Using Scales to highlight insights to infection trends across seasons.
plot_A <- df |>
  # Since we are grouping outcomes by Season, we only need to subset the
  # dataset to the three seasons we wish to plot.
  filter(Region == "Connecticut", Season %in% include_seasons, Level == "N/A") |>
  # Add the aesthetic mapping that will color (and group) outcomes based
  # on the unique elements in the Season variable. Also set the y-axis range
  # and customize plot labels.
  ggplot(aes(x = MMWRweek, y = Kernel, color = Season) ) +
    geom_line() +
    labs(title = "RSV Infection Trends Since 2022",
         x = "Weeks Since July",
         y = "Positive RSV Tests\n(scaled and kernel smoothed)") +
    ylim(0, 100)
    # Notice that we do not need to specify a scale_color_*() because
    # we want this plot to use the default color palette.


# Customizing the color applied to that grouping by building on the first plot.
plot_B <- plot_A +
    # "Type" denotes which set of color palette's to choose from. In
    # this case we are choosing from the "Type = Qualitative", from
    # which we can choose the "Palette = Dark2".
    scale_color_brewer(type = "qual", palette = "Dark2")
```



Discussion: We see that associating a variable with an `aes(color)` will also group by that same variable.

- What would happen if we only group the outcomes and exclude the color statement: i.e. `aes(group = Season)`?
- Does adding a `scale_color_*()` statement change the result?
- Why do you see the result you get?

> **Important**
>
> **Scales** allows you to modify the limits of axes (i.e. with `lims()` or `*lim()`) giving the illusion that you can "zoom" into the plot. This is not a good use of the **Scales** layer, as it will sometimes have the unintended effect of distorting your result. Instead use the **Coordinates** layer to zoom into a select portion of your plot.

> **Code Like a Pro**
>
> `ggplot2` does not give users a way to combine multiple plots together into one composite figure that is publication-ready. Fortunately, the community developed `ggplot2` extension, `cowplot`, was created specifically to support generating publication-quality figures. This is the package extension I used to generate plots **A** and **B** in the recent example! `cowplot` pkgdown page developed by Professor Claus Wilke at the University of Texas at Austin.

4.2.5 The Facets Layer

In the recent section, we showed how we can use **Scales** to highlight differences in infection trends between seasons. Doing so will plot the results on the same object, but what if we want to separate group comparisons over multiple plots?

Technically, we could code a new plot for each discrete subgrouping of our data that we want to spread out. If each plot is comprised of the same elements and aesthetic settings, however, there is an easier way to achieve the same result. The **Facets** layer allows us to generate a matrix grid showing the exact same plot for each subgroup available in a discrete variable.

Similar to the **Scales** layer, each plot object we compose implicitly assigns the faceting to `NULL` by default. The user only needs to include a **Facets** layer expression if they wish to change the default settings.

```
# The default faceting is automatically applied.
df |>
  filter(Region == "Connecticut", Season %in% include_seasons, Level == "N/A") |>
  ggplot(aes(x = MMWRweek, y = Kernel, color = Season) ) +
    geom_line() +
    labs(title = "RSV Infection Trends Since 2022",
         x = "Weeks Since July",
         y = "Positive RSV Tests\n(scaled and kernel smoothed)") +
    ylim(0, 100) +
    scale_color_brewer(type = "qual", palette = "Dark2") +
```

```
    # The following line not required for us to include.
    facet_null()
```

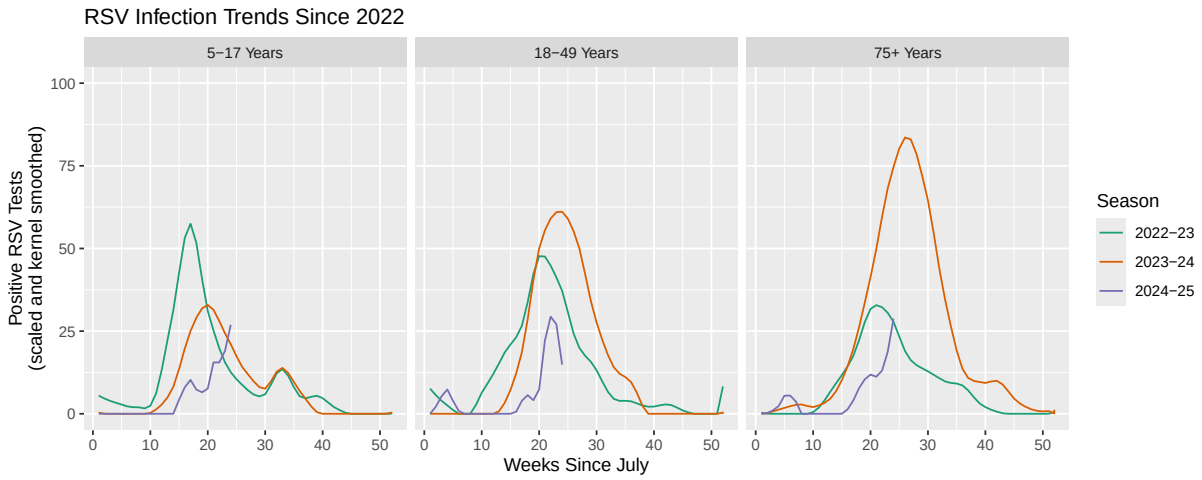
There are two functions for faceting: `facet_grid()` and `facet_wrap()`. Both functions receive similar arguments expressed slightly differently, and thus ultimately do the same thing. The biggest difference between them is `facet_grid()` will divide categories into new columns *or* rows, while `facet_wrap()` will optimize the matrix output into a roughly rectangular view.

It is best practice to choose the right faceting function based on how long the array of discrete outcomes you will be showing is. In our case, we are going to further resolve comparisons of RSV infections by seasons to include an additional comparison between a selection of age groups. The dataset includes seven age groups, but we are primarily interested in comparing infection trends in children, adults, and the elderly. We will therefore, only be plotting three different groups, making `facet_grid()` and acceptable function to use.

`facet_grid()` allows us to specify how the subplots are organized: by new columns *or* new rows. It is possible to compare as many as two variables by separating one over new columns and another over new rows: `facet_grid(rows ~ columns)`. We will separate out the age groups over new columns by dictating `facet_grid(~Level)`. Notice that we want to specify the order we see these panels, which we can do by wrapping our variable name with `factor(Level, levels = ordered_vector)`.

```
# Create our ordered vector that will be used in factor().
ages_ordered <- c("5-17 Years", "18-49 Years", "75+ Years")

# Add column-wise faceting.
df |>
  # We only want to maintain three of the seven age groups recorded. We can
  # use the same variable we generate above to select these.
  filter(Region == "Connecticut", Season %in% include_seasons, Level %in% ages_ordered) |>
  ggplot(aes(x = MMWRweek, y = Kernel, color = Season) ) +
    geom_line() +
    labs(title = "RSV Infection Trends Since 2022",
         x = "Weeks Since July",
         y = "Positive RSV Tests\n(scaled and kernel smoothed)") +
    ylim(0, 100) +
    scale_color_brewer(type = "qual", palette = "Dark2") +
    # Use "~Level" instead of "Level~" to create new plots as columns.
    facet_grid(~factor(Level, levels = ages_ordered))
```



4.2.6 The Coordinates Layer

Up until this point, we have covered layers that attribute variables to geometric-specific aesthetics in **Mappings** and transformations that interpret those assignments into plottable values in **Statistics** and **Scales**. The **Geometry** layer does a combination of statistical transformations and positioning, preparing the attributed variables into values that will take the chart shape we are looking to create.

It is the **Coordinates** layer that then plots these values on a graph. It's important to appreciate this subtlety because some scaled or statistically transformed values will look entirely different if the wrong coordinate system is applied. Such as, if we attempted to plot an angle on a Cartesian plane in Euclidean space instead of in polar coordinates.

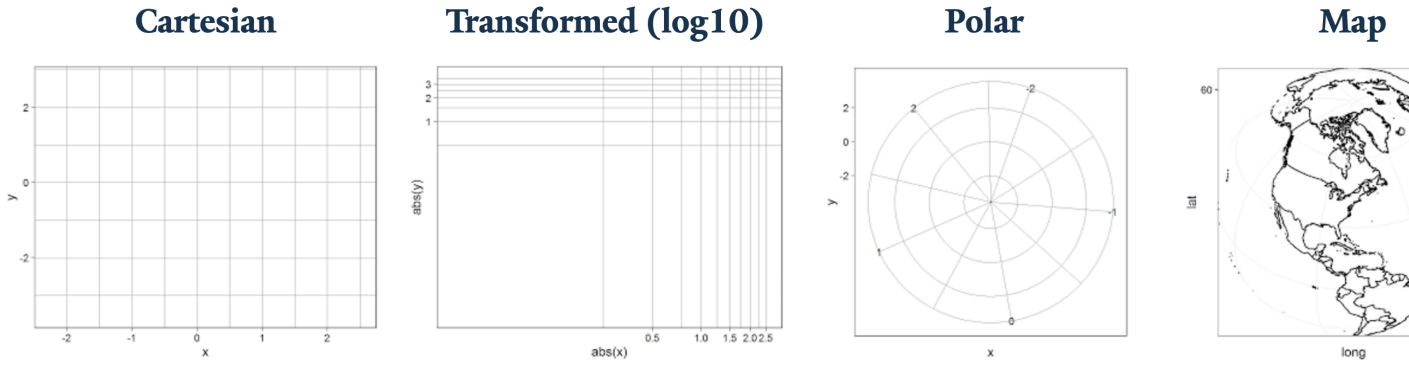


Figure 4.2: Figure 5.5 from "Data Visualization: From Theory to Practice" by James Baglin. Accessed March 22nd, 2025.

Coordinate layer functions fall into two buckets:

- Linear: `coord_cartesian()`, `coord_flip()`, and `coord_fixed()`
- Non-linear: `coord_map()`/`coord_quickmap()`/`coord_sf()`, `coord_polar()`, and `coord_trans()`

Applying the Cartesian, polar, and transformed coordinates is reasonably straightforward, and we will introduce the topic of map projections today. Keep in mind that the **Coordinate** layer plays an abstract role in the plot composite making for some key application notes that the user will need to be aware of. More on this subject can be explored in Chapter 15 Coordinate systems of *ggplot2: Elegant Graphics for Data Analysis*.

Previously, we mentioned that axis limits can be applied in **Scales**, but by doing so we unintentionally skew the results. Instead, it is better to "zoom" into a plot using the **Coordinates** layer. In a similar vein, if we want to swap the axes our x and y variables get plotted to, we want to use `coord_flip()` instead of changing `aes(x = x_var, y = y_var)` to `aes(x = y_var, y = x_var)`.

```
# By default, plots are generated on non-transformed Cartesian coordinates.
df |>
  filter(Region == "Connecticut", Season %in% include_seasons, Level %in% ages_ordered) |>
  ggplot(aes(x = MMWRweek, y = Kernel, color = Season) ) +
    geom_line() +
    labs(title = "RSV Infection Trends Since 2022",
         x = "Weeks Since July",
         y = "Positive RSV Tests\n(scaled and kernel smoothed)") +
    ylim(0, 100) +
    scale_color_brewer(type = "qual", palette = "Dark2") +
    facet_grid(~factor(Level, levels = ages_ordered)) +
    # The following line not required for us to include.
    coord_cartesian()
```

Code Like a Pro

As with any grammar, there are different ways we can make a statement. Say we have a variable that plots with a curve, and you know that you can linearize it using a base 10 log. We now have three approaches to graphically representing its linearized transformation. This example is from the `coord_trans()` documentation, Transformed Cartesian coordinate system.

- Transform before plotting or in `aes()` with `log10()`. **NOTE:** only do simple transformations in `aes()`, like a $\log_{10}$ transformation, and leave more complicated operations to more robust methods.
- **Scales**: `scale_*_log10()` or `scale_*_continuous(trans = "log10")`
- **Coordinates**: `coord_trans(* = "log10")`

Discussion: If we apply each option individually, do they all plot the same way? If not, what do you think is happening and how would you fix the code?

4.2.7 The Theme Layer

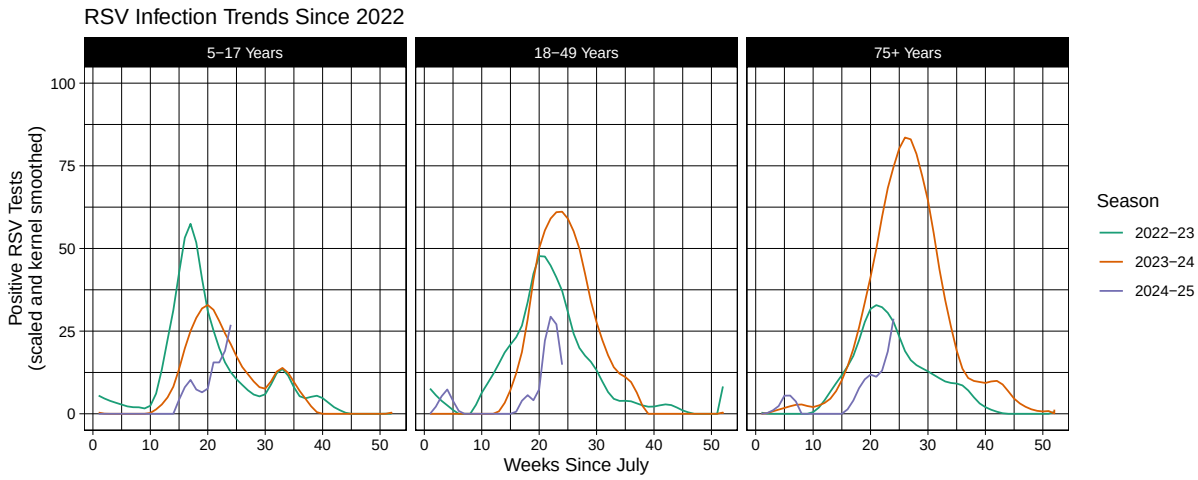
The **Theme** layer gives the users control over non-data aspects of the plot: i.e. styling the fonts, axes ticks, panel strips, backgrounds, location (or the exclusion) of the legend, etc. It is used to format the plot, making it visually appealing or match intended design schemes. Unlike every other layer described, **Theme** will not change how values are transformed, their perceptual properties, or how geometries get rendered.

How the user intends to use **Themes** will depend on where in the data processing cycle they are performing visualization, and how they want their polished plots to look. It is therefore not likely to be helpful if I get granular about the different function arguments in **Themes**. Instead, I encourage students to explore the layer reference page, where they can peruse details for all the possible function arguments at their leisure: Modify components of a theme.

`ggplot2` comes with numerous theme that get installed with the package: Complete themes. There are also numerous community developed themes that you can load, some of which have been curated by R Charts: Themes in ggplot2.

Each time you make a plot, the default is `theme_gray()`. As before, you do not need to explicitly write this out if you do not intend to change the default settings. In our example, we want use another, more visually appealing theme that better displays our results. We will use another `ggplot` provided theme: `theme_linedraw()`.

```
# Using one of ggplot2's provided theme presets.
df |>
  filter(Region == "Connecticut", Season %in% include_seasons, Level %in% ages_ordered) |>
  ggplot(aes(x = MMWRweek, y = Kernel, color = Season) ) +
    geom_line() +
    labs(title = "RSV Infection Trends Since 2022",
         x = "Weeks Since July",
         y = "Positive RSV Tests\n(scaled and kernel smoothed)") +
    ylim(0, 100) +
    scale_color_brewer(type = "qual", palette = "Dark2") +
    facet_grid(~factor(Level, levels = ages_ordered)) +
    # Add the Themes layer at the end of our plot object expression.
    theme_linedraw()
```



Code Like a Pro

Say you regularly generate plots that you want to apply the same custom **Theme** settings to, such as producing visualizations for a group that has specific branding or plots for a cohesive publication. There are two possible solutions you can take that will obviate the need to rewrite the same theme settings in each plot object.

- `theme_set()` allows you to set a new theme globally, overriding the default theme settings for the environment: Get, set, and modify the active theme.
- Creating your own custom theme function by modify an existing `theme_*()` or writing one from scratch: Learning to create custom themes in ggplot by Maddie Pickens and Chapter 20.1 New Themes of *ggplot2: Elegant Graphics for Data Analysis*.

4.2.8 Overlaying Layers

Now that we have become acclimated to each layer that constitutes one cohesive graph, we are ready to discuss how to effectively build more complex graphs by overlaying additional layers. Earlier I hinted that the expressions for each layer stands alone as its own object in the programming environment. When a graph is rendered by `ggplot2` its going to compress these layers one on top of another sequentially.

This is all well and good if the plots we are generating contain the same **Data** and **Mapping** settings, but what if they do not? What if we want to plot a new **Geometry** or **Statistic** layer that doesn't use the same settings used by other layers? Every time you add a new layer, you have the opportunity to override the inherited settings that were given in the leading plot object, `ggplot()`.

For example, consider the following code. What happens if we change aspects of the **Data** and **Mapping** used by `geom_line()`?

```
ggplot(df, aes(x = MMWRweek, y = Kernel)) +
  geom_line()
```

Table 4.1: Overriding `ggplot()`table from 13.4 Aesthetic mappings in *ggplot2: Elegant Graphics for Data Analysis*.

Operation	Layer aesthetics	Result
Add	`aes(colour = Season)`	`aes(x = MMWRweek, y = Kernel, colour = Season)`
Override	`aes(y = Level)`	`aes(x = MMWRweek, y = Level)`
Remove	`aes(y = NULL)`	`aes(x = MMWRweek)`

It is therefore best practice to assign the **Data** or **Mapping** in the leading `ggplot()` plot object when you want those settings carried forward to all subsequent layers. When you need to specify new settings for select **Geometric** or **Statistic** layers, then you can specify those mappings in that layer, overriding any previous designation. You can find more information about plot object layering in `ggplot` in Chapter 13 Build a plot layer by layer in *ggplot2: Elegant Graphics for Data Analysis*.

> Code Like a Pro
>
> The ability to treat each layer as its own object in the coding environment allows for flexible and adaptive programming. For example, you can predefined the settings to generating a smoothed line that you apply to different `ggplot()` objects, leaning on inherited **Data** and/or aesthetics **Mappings** given in each.
> You can find more information about plot object layering in `ggplot` in Chapter 18 Programming with ggplot2 in *ggplot2: Elegant Graphics for Data Analysis*.

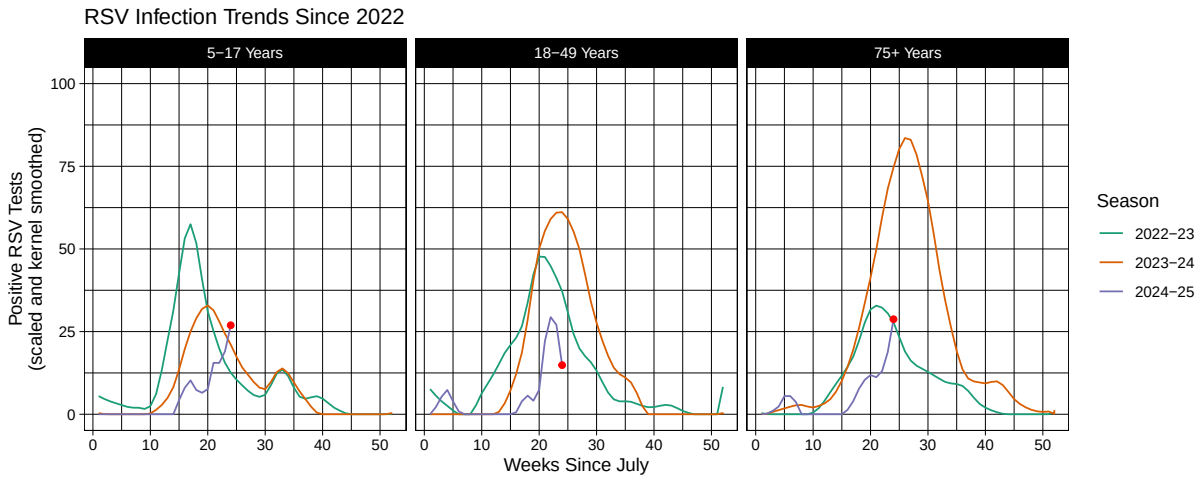
To exhibit this in our running example, let's add our pièce de résistance, the cherry on top, a leading point highlighting the most recently recorded surveillance week. In order to isolate this exact point, we need to prepare a different subset from the same dataset. We do this by filtering for the same distinguishing variables (Region, Season, and Characteristic Level), followed by filtering to the current infections season, and then finally the most recent week recorded.

This point is added to our graph as a fresh layer that is printed on the plot object generated up to the `geom_line()` expression. In the function settings, we override the inherited **Data**

object and leave the **Mappings** blank, allowing the aesthetics settings to be inherited from the `ggplot()` object.

```
# Subset the dataset to isolate the leading data point we wish to plot.
leading_point = df |>
  filter(Region == "Connecticut", Season %in% include_seasons, Level %in% ages_ordered) |>
  filter(MMWRyear == max(MMWRyear)) |>
  filter(MMWRweek == max(MMWRweek))

# Add a new Geometry layer, geom_point(), using a new dataset.
df |>
  filter(Region == "Connecticut", Season %in% include_seasons, Level %in% ages_ordered) |>
  ggplot(aes(x = MMWRweek, y = Kernel, color = Season) ) +
    geom_line() +
    # Show the leading point as a red dot using the leading_point subset.
    geom_point(data = leading_point, color = "red") +
    labs(title = "RSV Infection Trends Since 2022",
         x = "Weeks Since July",
         y = "Positive RSV Tests\n(scaled and kernel smoothed)") +
    ylim(0, 100) +
    scale_color_brewer(type = "qual", palette = "Dark2") +
    facet_grid(~factor(Level, levels = ages_ordered)) +
    theme_linedraw()
```



Look at us, we are such `ggplot` experts now! High five! Oh wait a second, for some of you this is asynchronous material… give yourself a high five for me :+1:

4.3 Advanced Uses of `ggplot2()`

In this section we will cover two advanced uses of `ggplot2` commonly used in public health: projecting data into a map and making the values interactive. These are topics well worth their own workshop, and I will not have time to do them justice here.

The goal of the next two sections will be to introduce you to these rich topics. At the end of this section, you will hopefully find these topics more approachable and are better equip to use them in your own work.

4.3.1 Introduction to Map Projections with `ggoplot`

Canonically, map projections are interpreting the curved surface of the earth into a flat plane. For our purposes, we need to render the polygons that represent this projection and associate them with metadata that will visualization our data. `ggplot2` comes with two different methods to do both these steps: `geom_polygon()` or `geom_sf()`.

The first is not as robust as the second and, in fact, the second method is recommended by the `ggplot2` developers over `geom_polygon()` for this reason. We are therefore only going to cover `geom_sf()`.

Before we apply this function, let's think first about what we are doing. Previously I mentioned that map projections are interpreting a curved shape into a flat plane. This projection will give us boundary points that are plotable on a linear Cartesian plane, called polygons. "Simple features", from which we get "sf", are standard vector data produced by the Open Geospatial Consortium (OGC) that interpret this action into plotable polygons.

Unfortunately, getting a file that contains the region boundaries we are looking for is a rather, shall we say, winding task. This problem gets compounded if we need our resource to contain granular information about the region, such as rivers, lakes, and topology, or even man-made infrastructure, such as roads, cities, or census boundaries.

We're not going to bother with this problem for now. Our dataset contains information about select contiguous U.S. states, and batches those trends into the HHS regions. Even though we do not need to be immediately concerned with how to plot Alaska and Hawaii into the same figure, we will include that as well.

Here, we will reference the Topologically Integrated Geographic Encoding and Referencing, or TIGER/Line Shapefiles, provided by the U.S. Census Bureau. The R package `tigris` allows us to pull these files from the U.S. Census Bureau website and process them for our application.

```r
# Call the TIGER/Line Shapefiles from the Census Bureau.
us_geo <- tigris::states(class = "sf", cb = TRUE) |>
  # Translate the geometric locations for Alaska and Hawaii to plot underneath
  # the contiguous states.
  shift_geometry()

head(us_geo)
```

```
Simple feature collection with 6 features and 9 fields
Geometry type: MULTIPOLYGON
Dimension:     XY
Bounding box:  xmin: -2356114 ymin: -782290.7 xmax: 1419556 ymax: 970427.9
Projected CRS: USA_Contiguous_Albers_Equal_Area_Conic
  STATEFP  STATENS     GEOIDFQ GEOID STUSPS         NAME LSAD        ALAND
1      35 00897535 0400000US35    35     NM   New Mexico   00 314198519809
2      46 01785534 0400000US46    46     SD South Dakota   00 196341670967
3      06 01779778 0400000US06    06     CA   California   00 403673433805
4      21 01779786 0400000US21    21     KY     Kentucky   00 102266755818
5      01 01779775 0400000US01    01     AL      Alabama   00 131185561946
6      13 01705317 0400000US13    13     GA      Georgia   00 149485762701
       AWATER                       geometry
1   726531289 MULTIPOLYGON (((-1231344 -5...
2  3387563375 MULTIPOLYGON (((-633765.6 8...
3 20291632828 MULTIPOLYGON (((-2066923 -2...
4  2384136185 MULTIPOLYGON (((584560 -886...
5  4581813708 MULTIPOLYGON (((760323.7 -7...
6  4419221858 MULTIPOLYGON (((1390722 -58...
```

In this dataset, we see that there are various state identifiers

- **STATEFP** - Federal Information Processing System (FIPS) ID
- **STATENS** - United States Geological Survey (USGS) ID
- **AFFGEOID** - American Fact Finder Geographic Identifiers
- **GEOID** - the standard geographic identifiers

and land type identifiers

- **LSAD** - Legal/Statistical Area Description (LSAD)
- **ALAND** - land size in square miles
- **AWATER** - water size in square miles
- **Geometry** - the longitude and latitude for the polygon vertices.

Together, these are used by `geom_sf()` to create map projections out of our spacial data. And believe it or not, that was the hardest part about map projections. Once we have these polygons, we are well on our way to visualizing our data on a map.

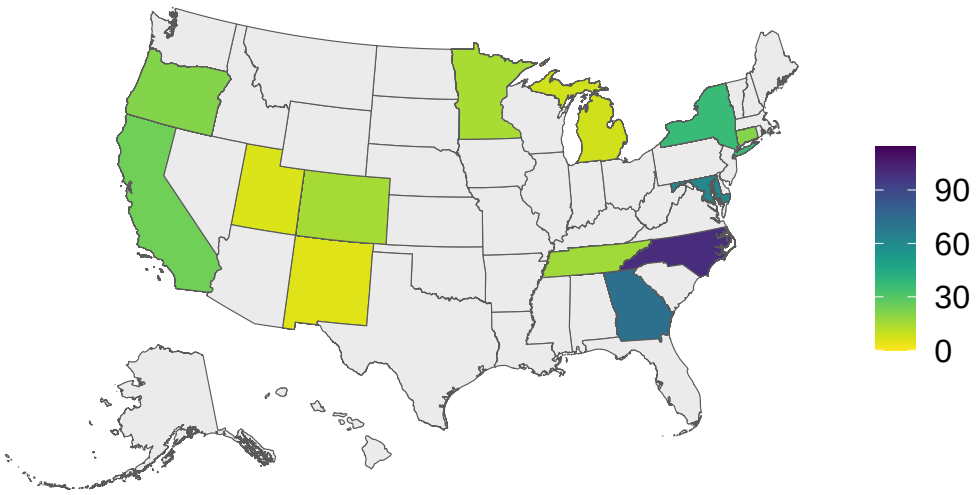
Going back to our running example, let's show the distribution of infection trends near the peak of the 2024-25 season, around weeks 24 to 26.

```
subset <- df |>
  # Keep only the states and the timeframe we want to plot.
  filter(Region %in% c(datasets::state.name, "District of Columbia"),
         Season %in% "2024-25", MMWRweek %in% c(24:26))

# Combine the two datasets by Region in df and NAME in us_geo.
geo_df <- us_geo |>
  left_join(subset, c("NAME" = "Region")) |>
  # Using GEOID, we will only keep the U.S. states.
  filter(GEOID < 60)

map_plot <- geo_df |>
  # Color the polygon with the number of RSV positives tested.
  ggplot(aes(fill = Kernel), color = "black") +
    # Apply the SF compatible Geometry.
    geom_sf() +
    # Scale the aesthetics to apply the viridis colors and do not
    # add colors for the NA's
    scale_fill_viridis(direction = -1, na.value = "grey92") +
    labs(fill = NULL, title = "Peak 2024-25 Season RSV Infection Trends\nResults Scaled and C
    # Apply the SF compatible Coordinates.
    coord_sf() +
    theme_map()

map_plot
```



Voilà!

> **!** Geometry and Statistic Layers Dictate the Aesthetics Mapping
>
> `ggplot` is a convenient way to render a plot, but it is not the most efficient package option out there! If you are looking for something that is faster and more flexible, you might want to check out packages like `leaflet` (Cran R documentation).

4.3.2 Introduction to Interactive Plotting with `plotly`

A static graphic can communicate a lot about your data, but with html widgets, like `plotly`, you can make the plot user interactive. These widgets allow actions like zooming, hovering, and selecting variables in the legend to be displayed. There are different widgets to choose from, and usually you will pick one that is best suited to make the graphic you intend. You can find some inspiration from the R Graph Gallery.

Today, I will show you how to apply one of the most commonly used interactive packages, `plotly`. This package is cross platform compatible and can be used in Python, Julia, and MATLAB. It has its own standalone function for plot generation that come with more options, but if you are looking to quickly make a `ggplot()` interactive all you need to do is wrap it with `ggplotly()`.

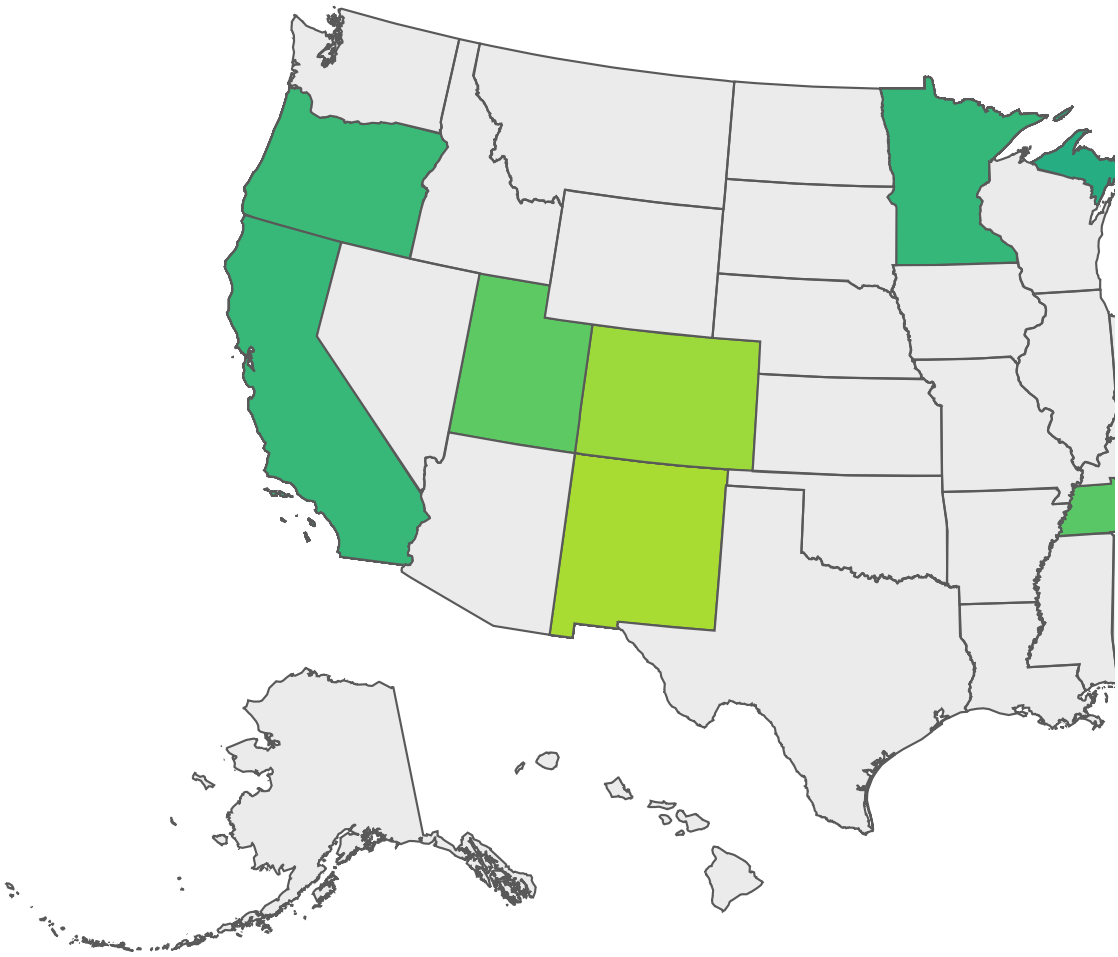
You can find more about `plotly` and the kinds of interactive plots it can generate with `ggplot()` objections in their documentation: Plotly R Open Source Graphing Library and Plotly ggplot2 Open Source Graphing Library.

```
ggplotly(map_plot)
```

```
PhantomJS not found. You can install it with webshot::install_phantomjs(). If it is installe

file:////private/var/folders/9f/rwy2b8vj3m90x_s1fvx553cn5v3lr9/T/RtmpnHF8Kf/fileb4782894b213/
```

Peak 2024-25 Season RSV Infection Trends
Results Scaled and Gaussian Kernel Smoothed



4.4 Yale's Clarity AI

We will now ask Yale's Clarity Platform to reproduce the code from one of our plots using the image alone.

1. Take a screenshot of one of the plots from the workshop or one of your own.
2. Navigate to ai-chat.yale.edu/signin-oidc.
3. Upload the screenshot into the chat and ask " Suggest R code that could make this plot in ggplot2."

Discussion: How did the AI chatbot do?

4.5 Challenge Questions

You can find the code for these questions and their suggested solutions in `Discussion and Challenge Questions.R`.

1. In the worked through example we leverage two of the three distinguishing variables: Season and Characteristic Level. In this questions, we want to focus on the third one, Region. Using the final line-graph code from the worked through example, add a row-wise facet to compare two states or HHS regions.
2. In the polished version of the line plot we layered an additional point on the graph that highlighted the most currently represented date in the active infection season. To get this data, we externally filtered down the dataset to give that point, repeating some of the filtering we are already doing before the plot gets generated.
    a. Can you modify this approach so that we no longer require the external dataset, thus simplifying our code?
    b. Now try using `stat_summary(filtered_data, geom = "point", fun = "max", color = "red")` instead of `geom_point()`. If you only filter to MMWRyear, do you get the same result, and if not why?
3. In our line graph we faceted by age groups. But we see that if we apply a simple **Facet** layer to our map plot the NA values get plotted in their own panel.
    a) Why is this happening?
    b) Fix this code to prevent this panel from being generated.

4.6 Appendix

This workshop was generated using R (v 4.4.4) in the RStudio IDE (v 2024.12.1+563). `renv` was used to store all the relevant packages and package versions with the workshop codespace. All of the workshop materials can be accessed on the ysph-dsde GitHub: Data Visualization with ggplo2. All relevant citations for this document can be found in the **Appendix** of the slides.

This site was built on a ysph-dsde.github.io page. It was rendered using Quarto (v 1.6.3) and theme sandstone.