Visualizing Single-Participant Data

Table of contents

Introduction

Welcome to the tutorial on "Visualizing Single-Participant Research Data with Line Plots using ggplot2 in R." This tutorial is designed for first-year graduate students who are embarking on their research journey and have little to no experience with data visualization in R.

Single-participant research studies play a vital role in various fields, including psychology, education, and healthcare. They involve the systematic observation of individual participants over time, allowing researchers to gain valuable insights into behavioral patterns, interventions, and treatment effects. One of the most powerful tools at your disposal for exploring and communicating the findings from such studies is data visualization.

We will use the popular ggplot2 package in R, a versatile and powerful tool for creating a wide range of data visualizations.

Throughout this tutorial, you will learn how to: - Set up your R environment and load the necessary packages.

- Import and examine single-participant research data. - Create clear and informative line plots with time on the x-axis and a measure on the y-axis. - Enhance your plots with labels, colors, and error bars. - Customize and format your visualizations to meet publication standards. - Apply your newfound skills to a real-world case study.

By the end of this tutorial, you will have the knowledge and skills to effectively visualize single-participant research data, enabling you to communicate your findings with clarity and precision. So, let's get started with the fundamentals of ggplot2 and begin your journey into the world of data visualization for single-participant research!

Section 1: Getting Started with ggplot2

Installing and Loading ggplot2

Before we dive into creating line plots for single-participant research data, we need to make sure that we have the necessary tools installed and loaded. We'll start by installing and loading the 'ggplot2' package, which is the cornerstone of our data visualization efforts.

Installation

To begin creating data visualizations with ggplot2, we'll first need to install and load the 'tidyverse' package, which includes ggplot2 and other helpful packages for data manipulation and visualization.

```
install.packages("tidyverse")
```

Then, if you want to use the full functionality of the package, you will need to load the tidyverse at the beginning of your code:

```
library(tidyverse)
```

```
-- Attaching packages --------------------------------------- tidyverse 1.3.1 --

v ggplot2 3.3.5     v purrr   1.0.1
v tibble  3.2.0     v dplyr   1.1.0
v tidyr   1.3.0     v stringr 1.5.0
v readr   2.1.1     v forcats 0.5.1

-- Conflicts ------------------------------------------ tidyverse_conflicts() --
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
```

If you ever find yourself *just* wanting `ggplot2` (i.e., and not all of the tidyverse packages), you can always load that individually:

```
library(ggplot2)
```

Let's also install the `scales` package, which we will use later in the tutorial.

```
install.packages("scales")
```

Understanding the Philosophy of ggplot2

Grammar of Graphics

`ggplot2` is built on the **Grammar of Graphics** concept, which breaks down a complex plot into a set of modular and composable components. Think of it as a language for describing how to create visualizations. The Grammar of Graphics consists of the following key components:

- **Data**: The dataset you want to visualize.
- **Aesthetic Mapping**: Mapping variables in your data to visual properties like color, size, or position.
- **Geometric Objects (Geoms)**: The geometric shapes used to represent data points (e.g., points, lines, bars).
- **Faceting**: Dividing your data into subsets and creating separate plots for each subset.
- **Statistics**: Summarizing or transforming your data before plotting (e.g., calculating means or smoothing lines).
- **Coordinate System**: Defining the scales and coordinate system for your plot.

By combining and customizing these components, you have full control over the appearance and structure of your visualizations.

Layers Approach

`ggplot2` uses a **layered approach** to create plots. You build a plot by adding layers, with each layer representing a different aspect of your data visualization. This approach encourages a modular and flexible workflow, allowing you to add or modify layers as needed.

Declarative Syntax

`ggplot2` uses a declarative syntax, meaning you describe what you want your plot to look like rather than specifying how to create it step by step. This results in concise and expressive code, making it easier to convey your visualization intentions.

In the following sections, we'll explore how to apply these principles when creating line plots for single-participant research data using ggplot2. Let's put the Grammar of Graphics into practice!

Section 2: The Anatomy of a ggplot

Let's make our first ggplot! But first, let's simulate some data.

```
# Simulate single-participant data
weeks <- 1:10
scores <- rnorm(10, mean = 75, sd = 5)

# Create a tibble
```

```r
student_data <- tibble(week = weeks, score = scores)

# Display the first few rows of the data
student_data
```

```
# A tibble: 10 x 2
    week score
   <int> <dbl>
 1     1  83.1
 2     2  72.4
 3     3  77.8
 4     4  74.6
 5     5  72.7
 6     6  84.3
 7     7  82.3
 8     8  78.7
 9     9  76.8
10    10  66.4
```

The main function for ggplots, is well, the `ggplot()` function! If you just run this function, you get a blank graph with nothing at all.

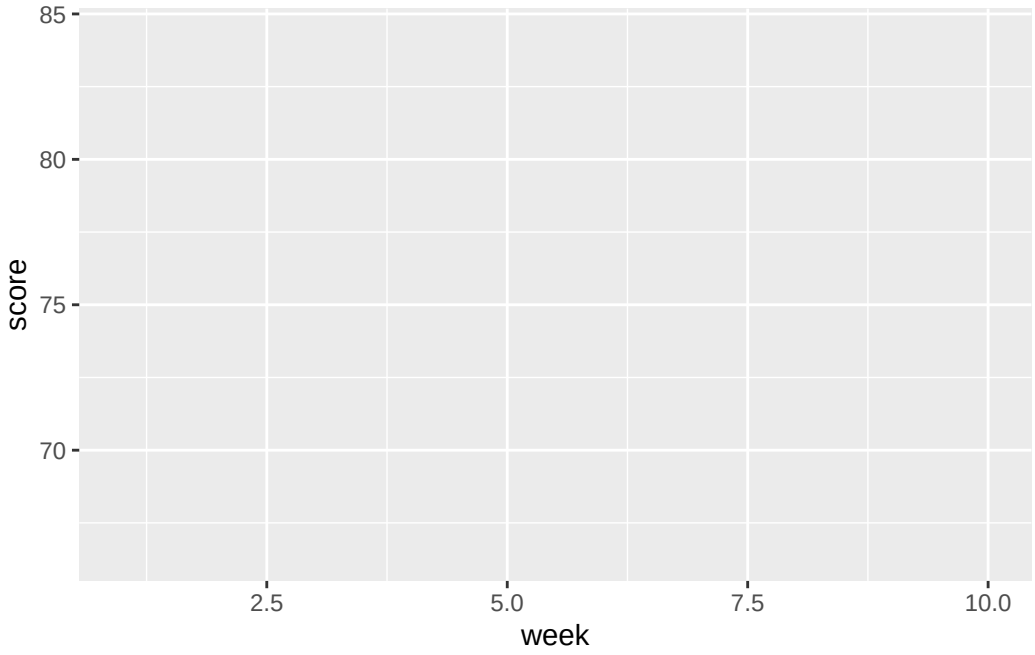
```r
ggplot()
```

But that's no fun! Let's add some data to graph. We can do that using the `data` argument in the `ggplot` function.

```
ggplot(data = student_data)
```

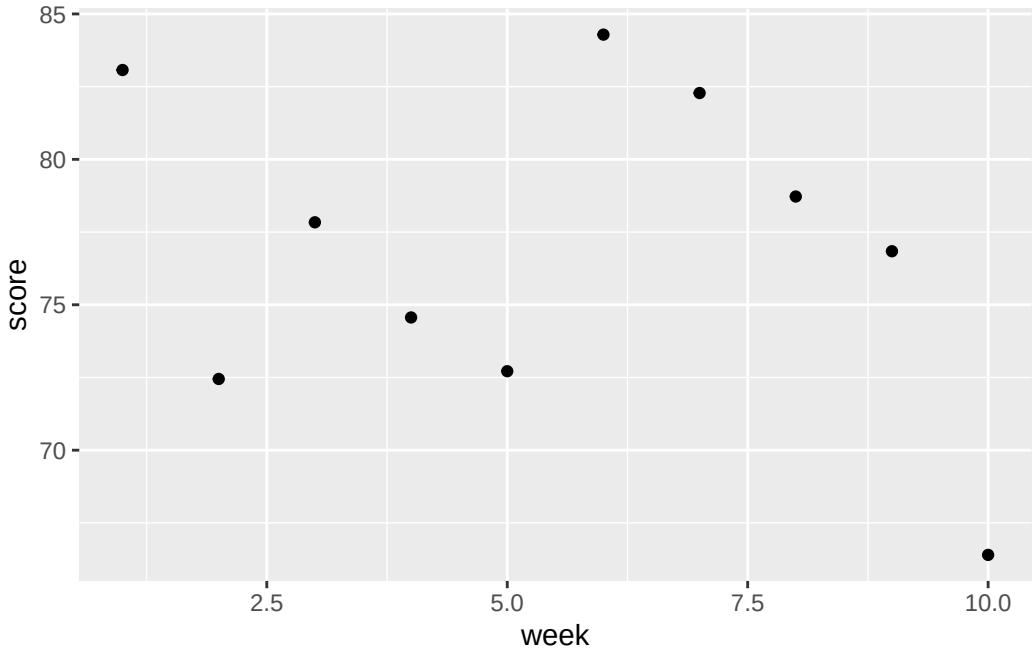
But wait, there isn't anything showing! We have to tell `ggplot` how to turn our data into graphical components, for example, what goes on the x- and y-axes. We do this process using the aesthetics function `aes()`. Here, we will tell it that the `week` variable should go on the x-axis and the `score` data is what we want on the y-axis.

```
ggplot(data = student_data, aes(x = week, y = score))
```



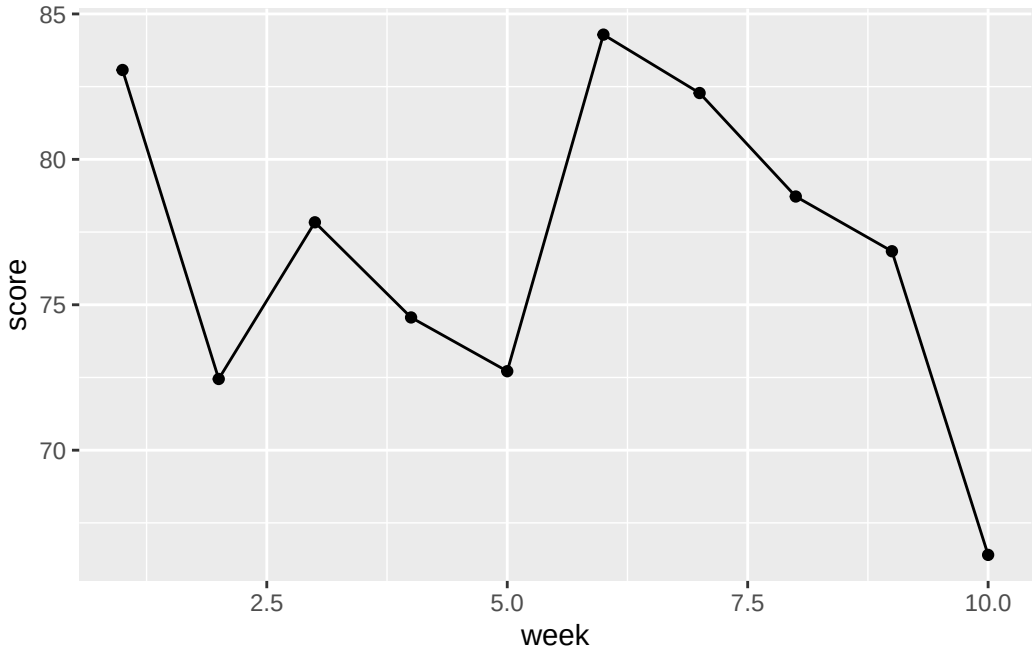
Now, you see that we have x- and y-axes. Next, we will tell `ggplot` how we want to map the data. We add a layer of visualization onto our ggplot using a `geom`. There are many types of geoms, including bar charts, box-plots, and almost any other types of chart. For now, we'll just focus on the ones that are most useful for visualizing single-participant data. Let's start by adding some points that represent our data. We add layers onto our graph by adding a `+`, followed by additional functions.

```
ggplot(data = student_data, aes(x = week, y = score)) +
  geom_point()
```



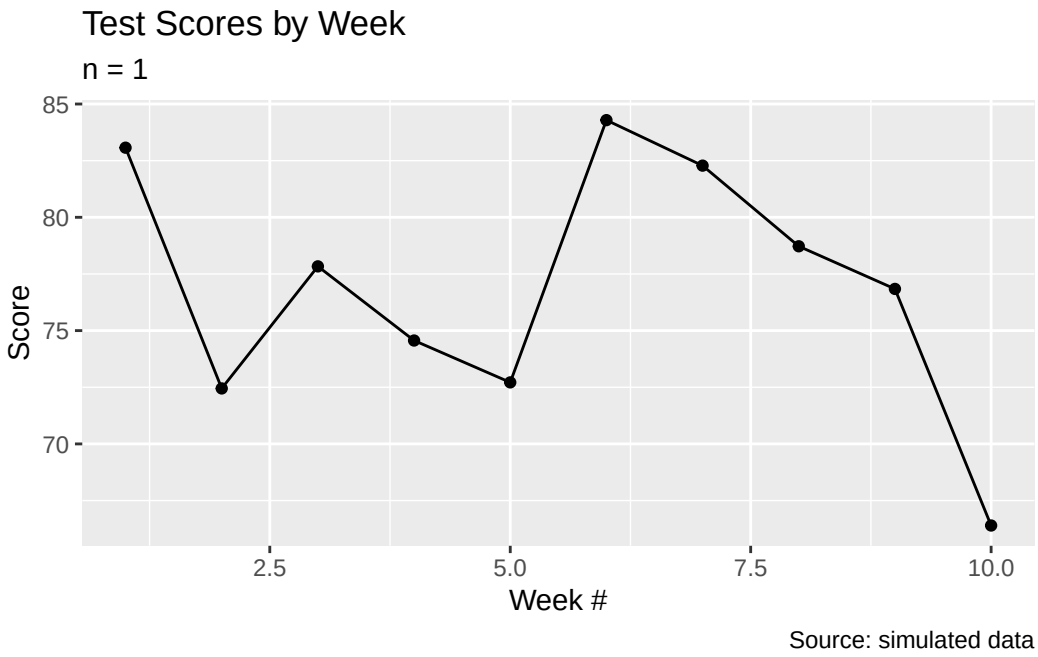
Now, we can see our single participant's scores across all 10 weeks. But, it's a little hard to make out what's going on. Let's add lines connecting all of the points.

```
ggplot(data = student_data, aes(x = week, y = score)) +
  geom_point() +
  geom_line()
```



That's looking pretty good! Let's add some labels to the graph using the `labs` function.

```
ggplot(data = student_data, aes(x = week, y = score)) +
  geom_point() +
  geom_line() +
  labs(title = "Test Scores by Week",
       subtitle = "n = 1",
       x = "Week #",
       y = "Score",
       caption = "Source: simulated data")
```



Congrats, you just made a good-looking ggplot!

Section 3: The Aesthetics Function

In the previous section, we used the aesthetics function, `aes` , to map our data to the x- and y-axes. But it can map to other components of our graph. Let's simulate some data again with additional variables that we may want to incorporate visually into our ggplot.

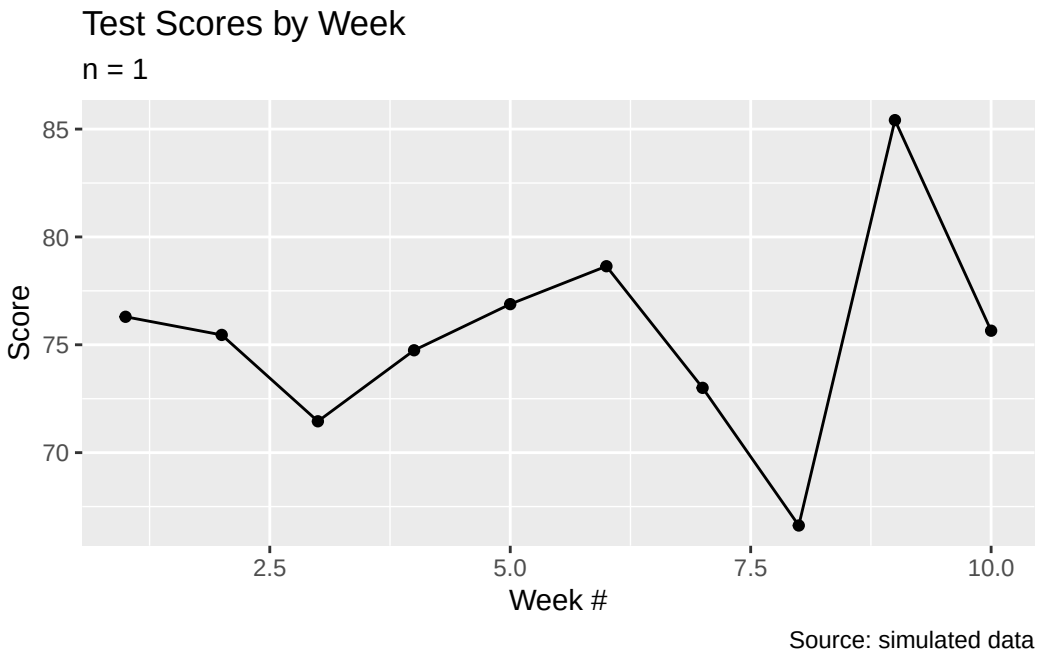
```
# Simulate single-participant data
weeks <- 1:10
scores <- rnorm(10, mean = 75, sd = 5)
scores_math <- rnorm(10, mean = 75, sd = 10)
scores_reading <- rnorm(10, mean = 65, sd = 3)

teacher <- factor(sample(c("Annie", "Brandon"), size = 10, replace = TRUE))
class_size <- sample(5:20, size = 10, replace = TRUE)
teacher_evaluation <- ordered(sample(1:5, size = 10, replace = TRUE),
                        levels = 1:5,
                        labels = c("Very Dissatisfied", "Dissatisfied", "Neutral", "Satisf

# Create a tibble
```

```r
student_data <- tibble(week = weeks, score = scores, teacher = teacher, class_size = class
  mutate(scores_math = if_else(week %in% c(1, 4, 5, 8, 9), scores_math, NA),
         scores_reading = if_else(week %in% c(2, 3, 6, 7, 10), scores_reading, NA))

# Display the first few rows of the data
student_data
```

```
# A tibble: 10 x 7
    week score teacher class_size teacher_eval      scores_math scores_reading
   <int> <dbl> <fct>        <int> <ord>                   <dbl>          <dbl>
 1     1  76.3 Brandon         16 Very Satisfied           75.7           NA
 2     2  75.5 Annie           14 Very Satisfied           NA             62.5
 3     3  71.5 Annie           12 Satisfied                NA             60.5
 4     4  74.7 Brandon          8 Dissatisfied             57.9           NA
 5     5  76.9 Annie           12 Satisfied                65.9           NA
 6     6  78.6 Annie           14 Very Satisfied           NA             65.1
 7     7  73.0 Annie           20 Very Dissatisfied        NA             67.8
 8     8  66.6 Annie           13 Satisfied                68.0           NA
 9     9  85.4 Annie           11 Satisfied                72.1           NA
10    10  75.7 Brandon          5 Very Satisfied           NA             66.3
```

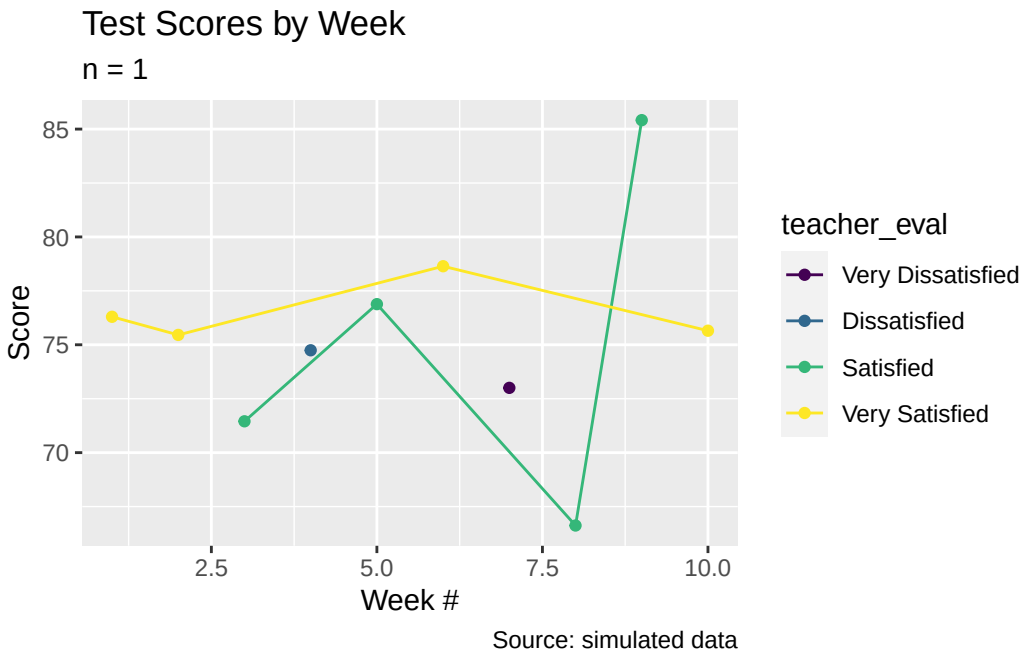
Let's start from our graph from before.

```r
ggplot(data = student_data, aes(x = week, y = score)) +
  geom_point() +
  geom_line() +
  labs(title = "Test Scores by Week",
       subtitle = "n = 1",
       x = "Week #",
       y = "Score",
       caption = "Source: simulated data")
```



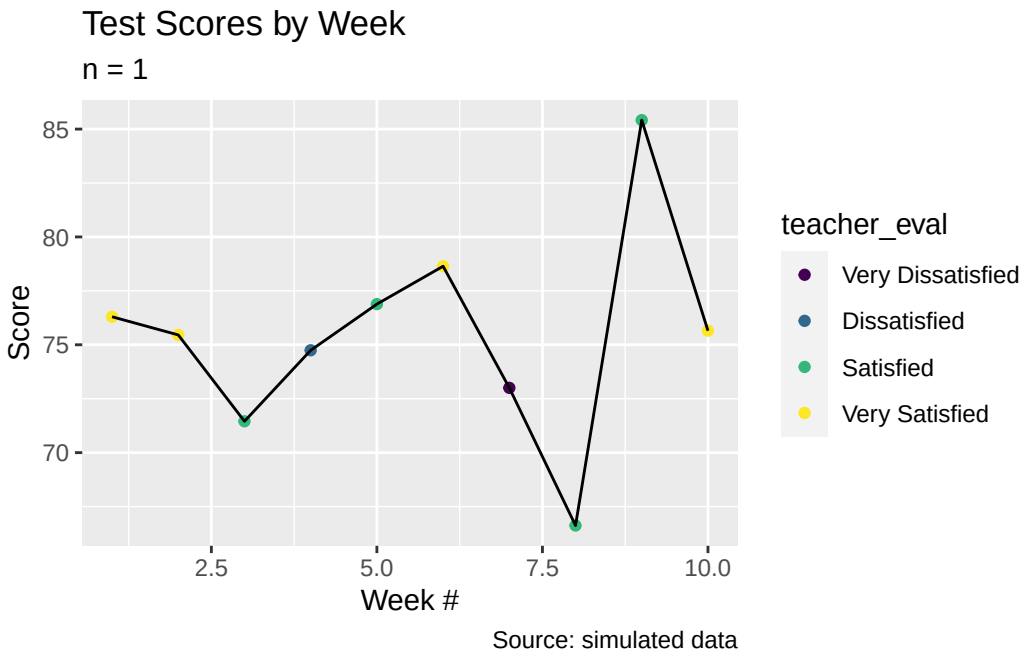
How may we want to incorporate these additional variables? Perhaps, we want to visualize the evaluations given by the teachers for our student. This visualization could tell us if the score and teacher evaluation are related. Let's do that by changing the color of the points based on the evaluations. We do that by using additional arguments in the `aes` function, in this case, the `color` argument.

```
ggplot(data = student_data, aes(x = week, y = score, color = teacher_eval)) +
  geom_point() +
  geom_line() +
  labs(title = "Test Scores by Week",
       subtitle = "n = 1",
       x = "Week #",
       y = "Score",
       caption = "Source: simulated data")
```



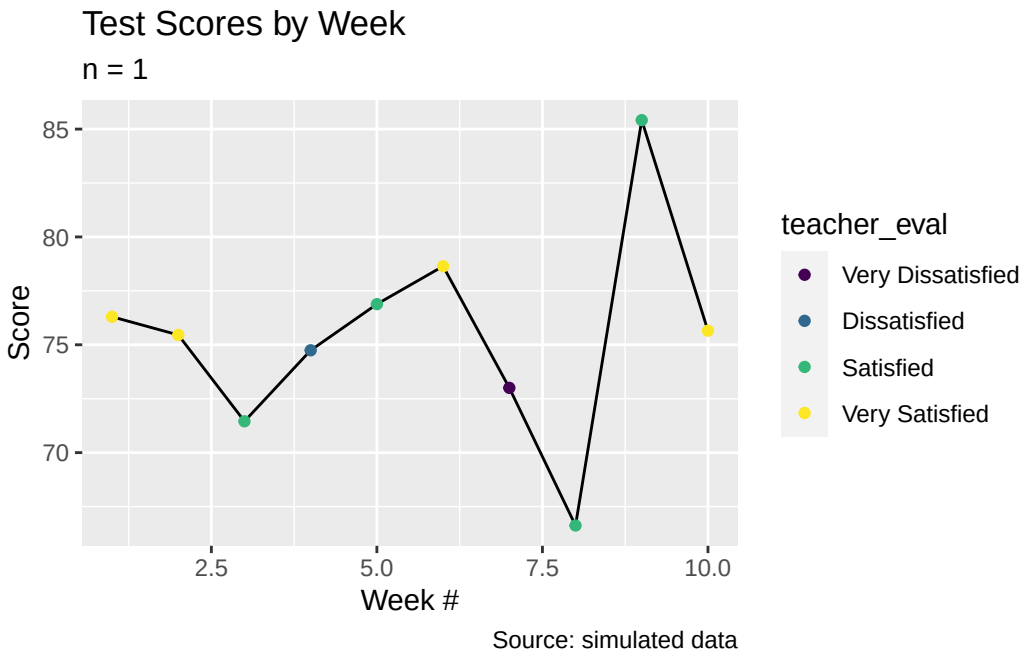
That doesn't look right! Well, when you use the `aes` function in the initial `ggplot` call, it applies the aesthetics to *all* geoms in your graph. In this case, it thinks we want to just connect lines that are the same color. In these cases, we may need to use an `aes` function in an individual geom, like this:

```
ggplot(data = student_data, aes(x = week, y = score)) +
  geom_point(aes(color = teacher_eval)) +
  geom_line() +
  labs(title = "Test Scores by Week",
       subtitle = "n = 1",
       x = "Week #",
       y = "Score",
       caption = "Source: simulated data")
```



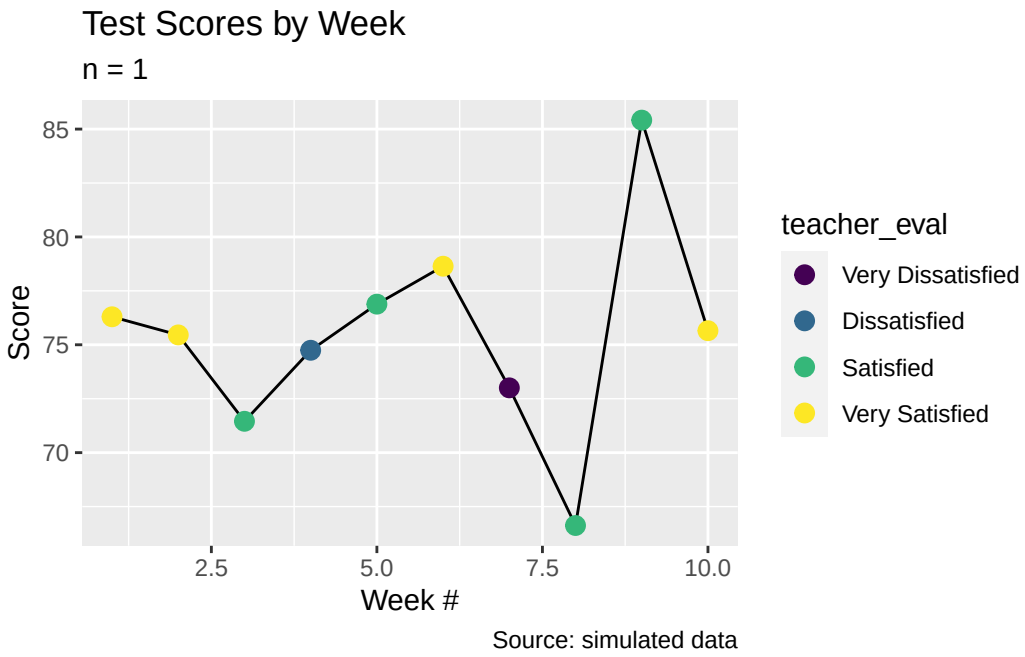
That's better, but it still doesn't look very good. Why not? Well, the black lines are covering up our colored points. This problem occurs because ggplots are sensitive to the order in which you add components. In this case, it adds the `geom_point` layer first, then the `geom_line` layer on top. So, let's switch the ordering of the two layers.

```
ggplot(data = student_data, aes(x = week, y = score)) +
  geom_line() +
  geom_point(aes(color = teacher_eval)) +
  labs(title = "Test Scores by Week",
       subtitle = "n = 1",
       x = "Week #",
       y = "Score",
       caption = "Source: simulated data")
```



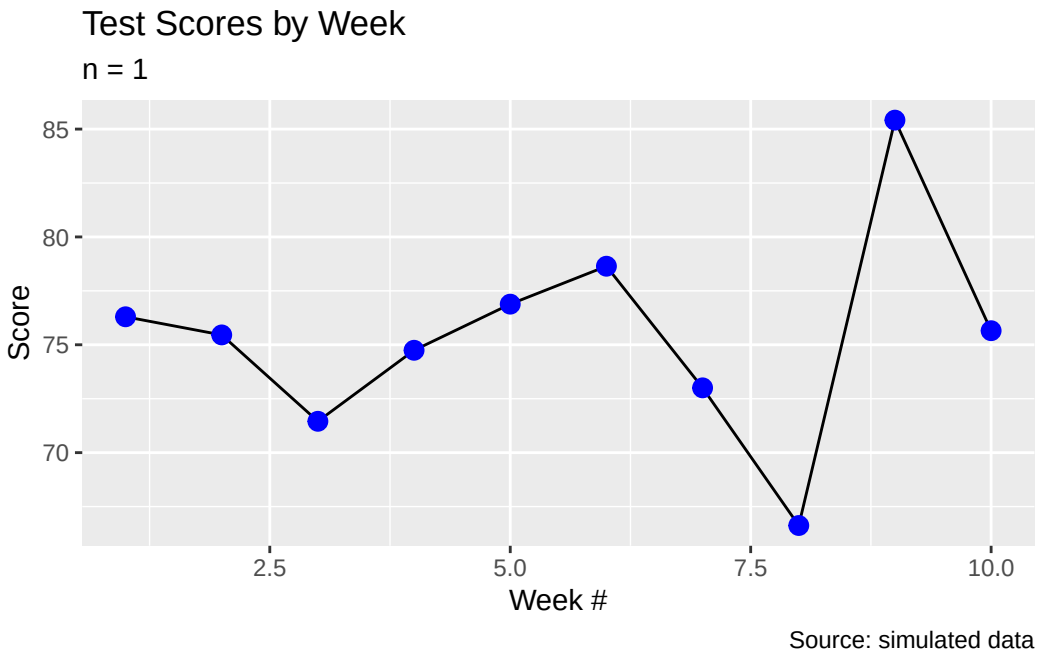
Nice! Let's say that we wanted to make the points a little bigger so that we can see them better. We can do that using one of the arguments that can be found in the `geom_point` function, `size`.

```
ggplot(data = student_data, aes(x = week, y = score)) +
  geom_line() +
  geom_point(aes(color = teacher_eval), size = 3) +
  labs(title = "Test Scores by Week",
       subtitle = "n = 1",
       x = "Week #",
       y = "Score",
       caption = "Source: simulated data")
```



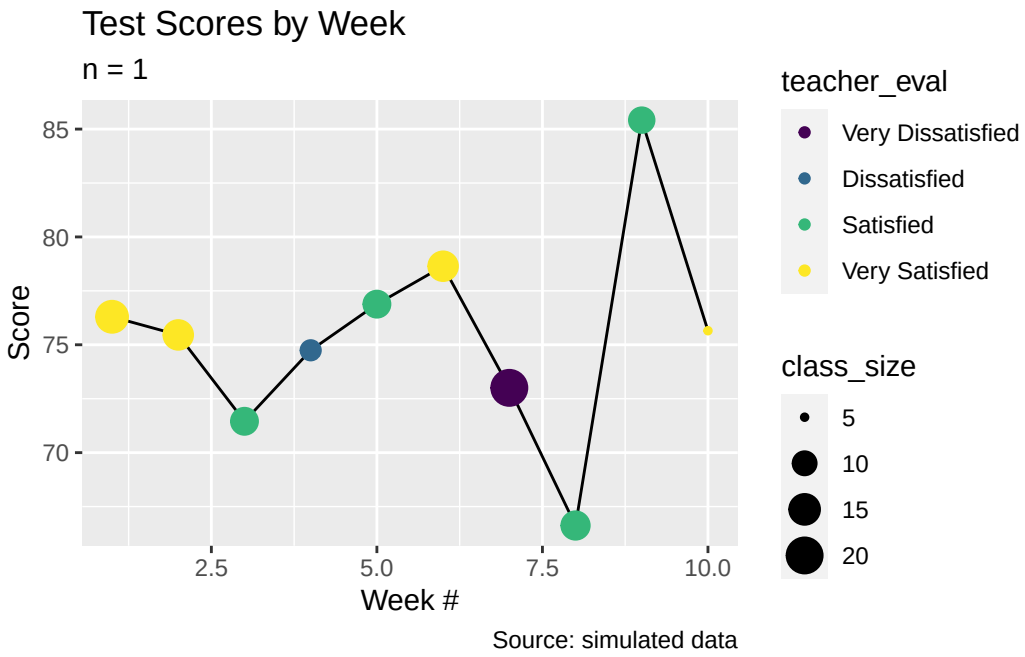
Note that arguments *outside* of the `aes` function are static–they are not tied to a variable. For example, we could turn all of the points blue by using the `color` argument *outside* of the `aes` function.

```
ggplot(data = student_data, aes(x = week, y = score)) +
  geom_line() +
  geom_point(color = "blue", size = 3) +
  labs(title = "Test Scores by Week",
       subtitle = "n = 1",
       x = "Week #",
       y = "Score",
       caption = "Source: simulated data")
```



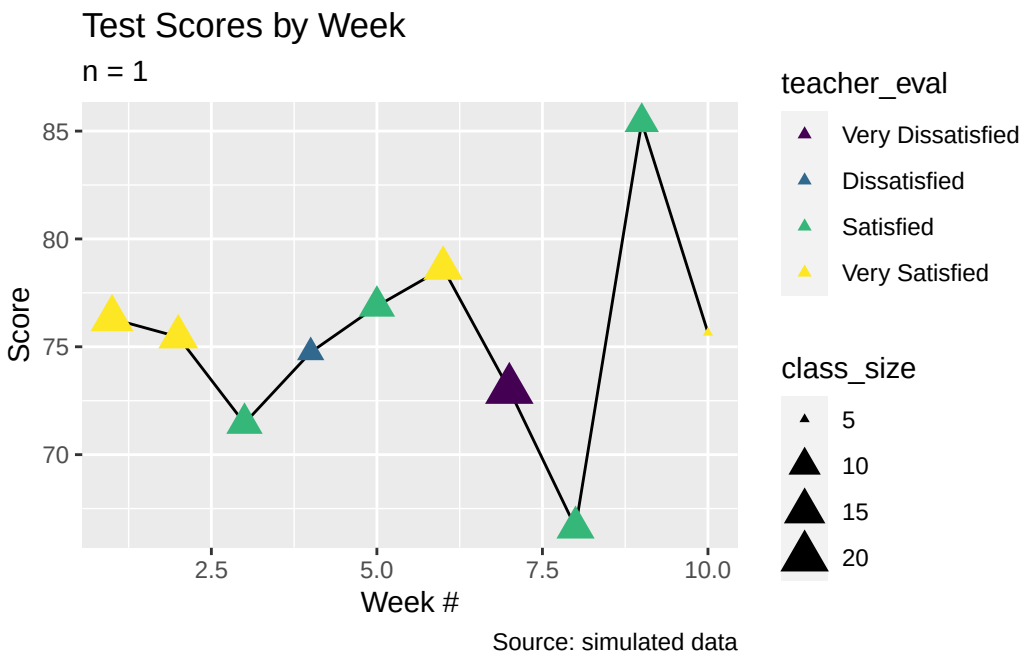
Getting arguments *inside* versus *outside* the `aes` function is very common. So, just remember, arguments inside the `aes` function are variable and are tied to your data. Arguments outside are static. For example, the size of the points doesn't need to be static either, it can be tied to our data, for example, on class size.

```
ggplot(data = student_data, aes(x = week, y = score)) +
  geom_line() +
  geom_point(aes(color = teacher_eval, size = class_size)) +
  labs(title = "Test Scores by Week",
       subtitle = "n = 1",
       x = "Week #",
       y = "Score",
       caption = "Source: simulated data")
```



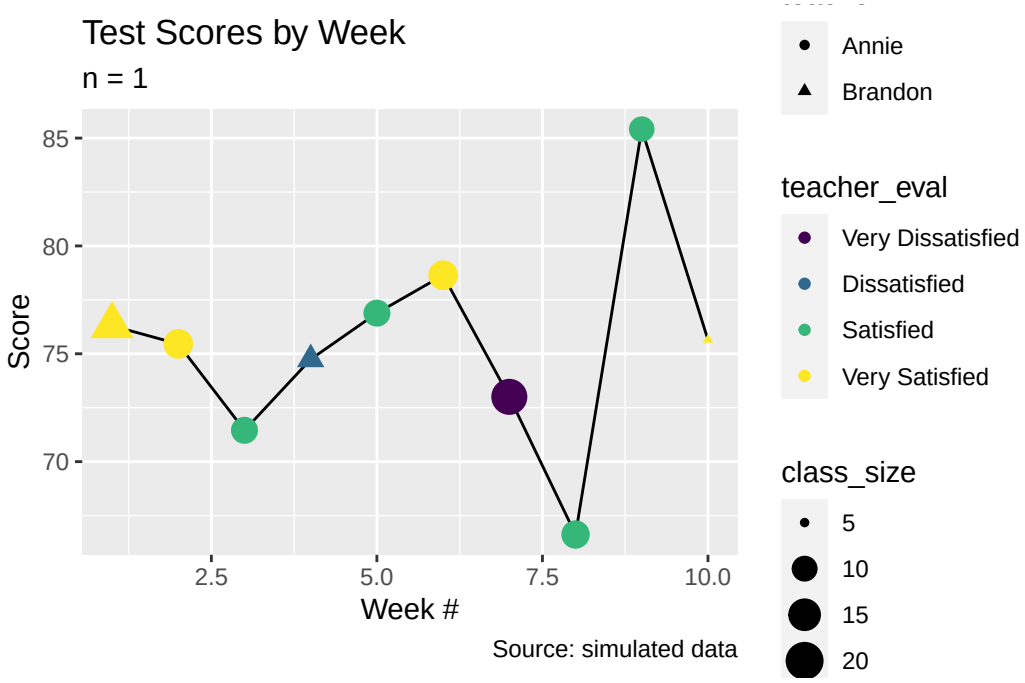
Another commonly-used argument is `shape`. Again, this can be used statically outside of the `aes` function.

```
ggplot(data = student_data, aes(x = week, y = score)) +
  geom_line() +
  geom_point(aes(color = teacher_eval, size = class_size), shape = "triangle") +
  labs(title = "Test Scores by Week",
       subtitle = "n = 1",
       x = "Week #",
       y = "Score",
       caption = "Source: simulated data")
```



Or dynamically inside of the `aes`, for example, to have each teacher be a different shape.

```
ggplot(data = student_data, aes(x = week, y = score)) +
  geom_line() +
  geom_point(aes(color = teacher_eval, size = class_size, shape = teacher)) +
  labs(title = "Test Scores by Week",
       subtitle = "n = 1",
       x = "Week #",
       y = "Score",
       caption = "Source: simulated data")
```



One other important `aes` argument is `group`. This argument is used when you have different groups of variables that you want visualized together. For example, let's say we want to visual the math and reading scores in our data separately. As a reminder, where is what the data look like, with math and reading scores in two different columns.

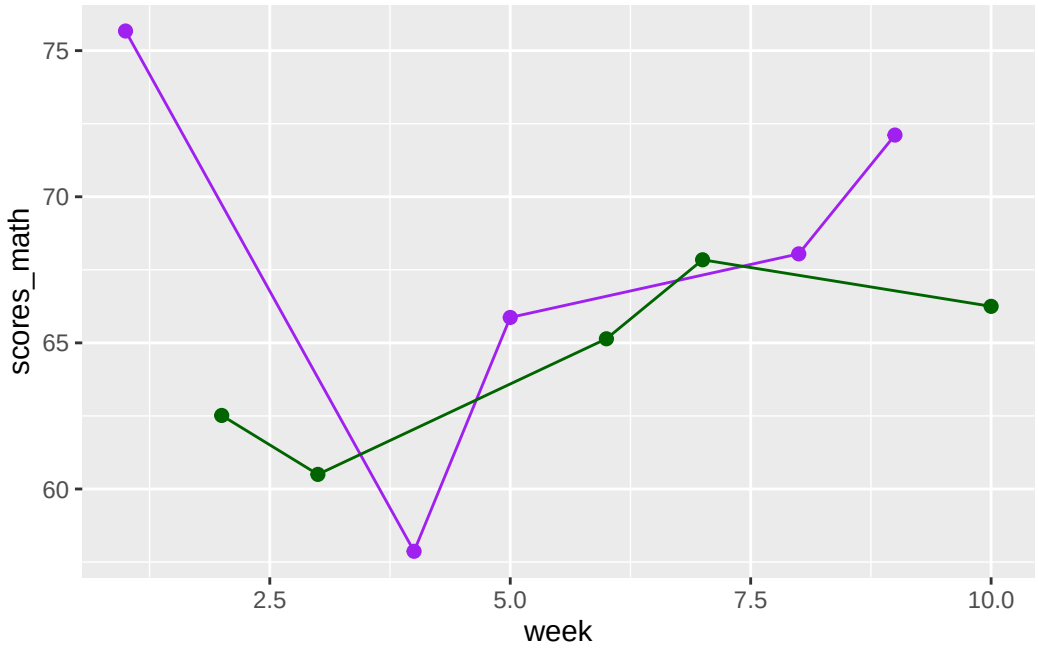
```
student_data
```

```
# A tibble: 10 x 7
    week score teacher class_size teacher_eval      scores_math scores_reading
   <int> <dbl> <fct>        <int> <ord>                   <dbl>          <dbl>
 1     1  76.3 Brandon         16 Very Satisfied           75.7           NA
 2     2  75.5 Annie           14 Very Satisfied           NA             62.5
 3     3  71.5 Annie           12 Satisfied                NA             60.5
 4     4  74.7 Brandon          8 Dissatisfied             57.9           NA
 5     5  76.9 Annie           12 Satisfied                65.9           NA
 6     6  78.6 Annie           14 Very Satisfied           NA             65.1
 7     7  73.0 Annie           20 Very Dissatisfied        NA             67.8
 8     8  66.6 Annie           13 Satisfied                68.0           NA
 9     9  85.4 Annie           11 Satisfied                72.1           NA
10    10  75.7 Brandon          5 Very Satisfied           NA             66.3
```

One inefficient way to visualize these two different groups of scores would be to have to have

separate `geoms` for each column. For example:

```
ggplot() +
  geom_point(data = student_data %>% filter(!is.na(scores_math)), aes(x = week, y = scores
  geom_point(data = student_data %>% filter(!is.na(scores_reading)), aes(x = week, y = sco
  geom_line(data = student_data %>% filter(!is.na(scores_math)), aes(x = week, y = scores_
  geom_line(data = student_data %>% filter(!is.na(scores_reading)), aes(x = week, y = scor
```



But sometimes your data in a tidy/ long format, where each score is a separate row/ observation and there is an additional variable/ column indicating what type of score it is. We will transform our data manually to be in that format.

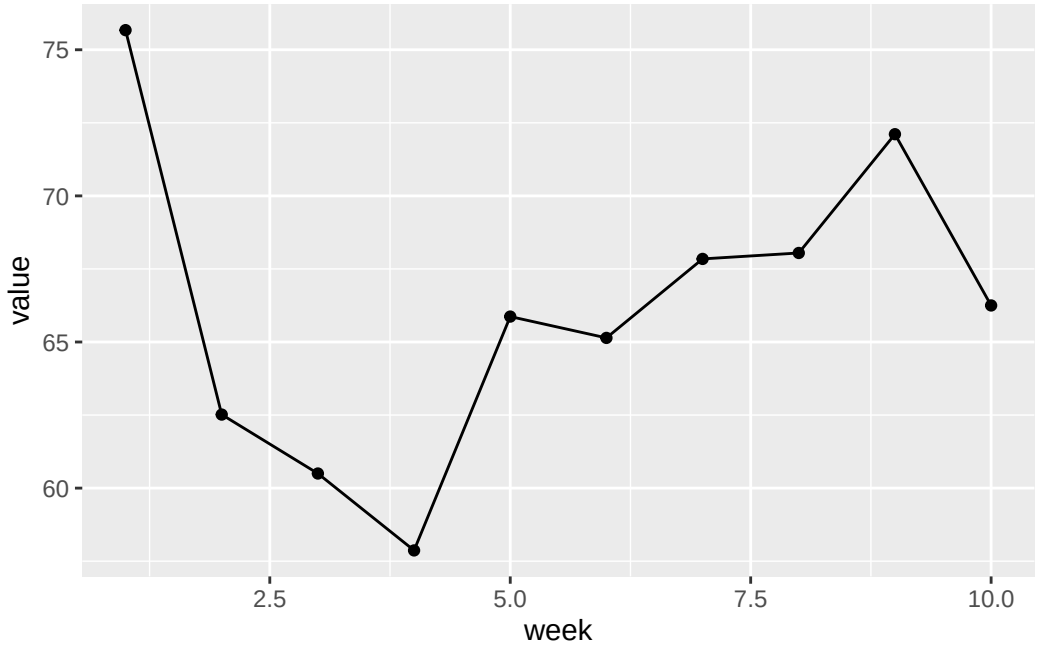
```
student_data_long <- student_data %>%
  pivot_longer(cols = c("scores_reading", "scores_math")) %>%
  filter(!is.na(value))

student_data_long
```

```
# A tibble: 10 x 7
    week score teacher class_size teacher_eval      name           value
   <int> <dbl> <fct>        <int> <ord>             <chr>          <dbl>
```

```
 1     1  76.3 Brandon         16 Very Satisfied    scores_math     75.7
 2     2  75.5 Annie           14 Very Satisfied    scores_reading  62.5
 3     3  71.5 Annie           12 Satisfied         scores_reading  60.5
 4     4  74.7 Brandon          8 Dissatisfied      scores_math     57.9
 5     5  76.9 Annie           12 Satisfied         scores_math     65.9
 6     6  78.6 Annie           14 Very Satisfied    scores_reading  65.1
 7     7  73.0 Annie           20 Very Dissatisfied scores_reading  67.8
 8     8  66.6 Annie           13 Satisfied         scores_math     68.0
 9     9  85.4 Annie           11 Satisfied         scores_math     72.1
10    10  75.7 Brandon          5 Very Satisfied    scores_reading  66.3
```

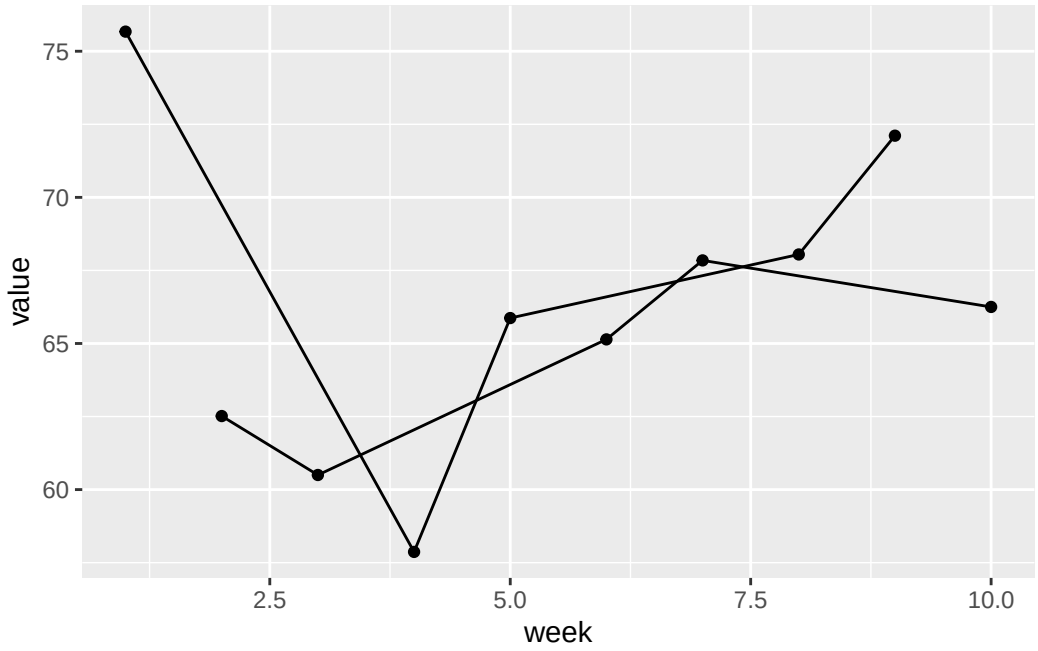
We can no plot this data much more efficiently. But first let's see what happens if we don't use the `group` argument.

```
ggplot(data = student_data_long, aes(x = week, y = value)) +
  geom_line() +
  geom_point()
```



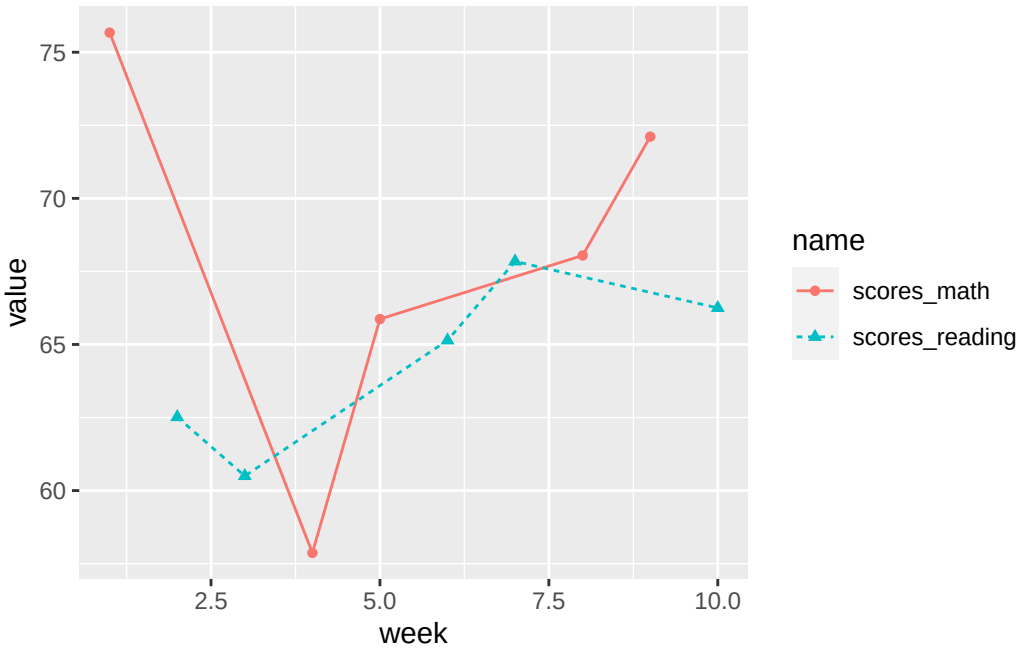
The plot looks like this because it doesn't know which points belong in which group, so it's trying to connect them all sequentially. You can use the `group` argument to specify which points belong to which groups.

```
ggplot(data = student_data_long, aes(x = week, y = value, group = name)) +
  geom_line() +
  geom_point()
```



Of course, in this case, you'd probably want to change the shape or color to distinguish the two groups.

```
ggplot(data = student_data_long, aes(x = week, y = value, group = name, shape = name, colo
  geom_line() +
  geom_point()
```

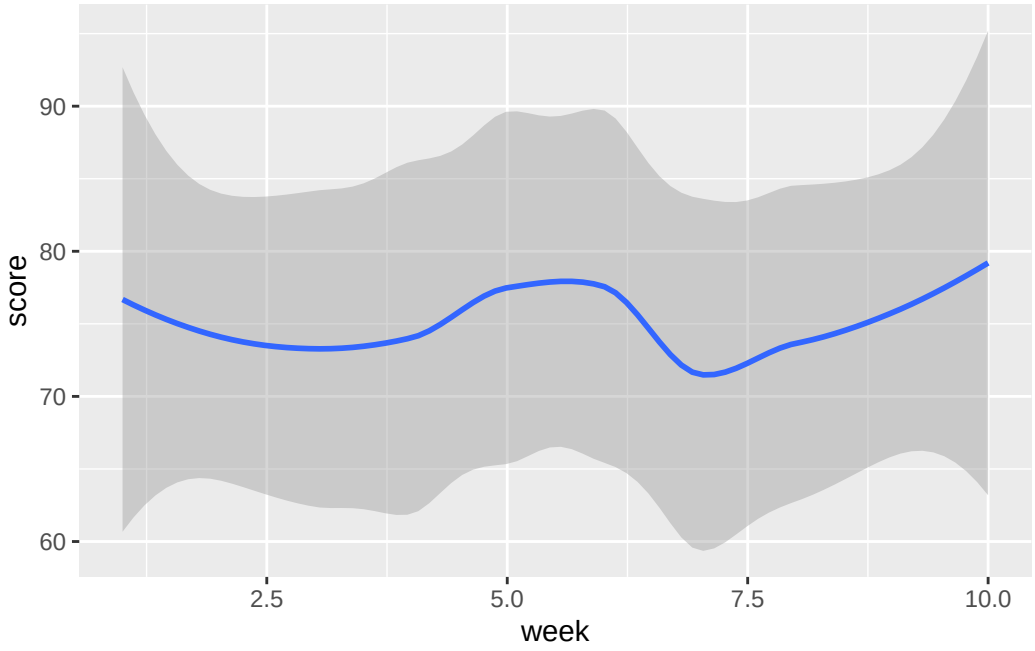


Section 4: Visualizing Statistical Inferences & Summarizations

Ggplot can also do statistical inferences and summaries for you. One of the most common examples of this functionality is visualizing trend lines (with error bars), which primarily can be used via the `geom_smooth` function

```
ggplot(data = student_data, aes(x = week, y = score)) +
  geom_smooth()
```

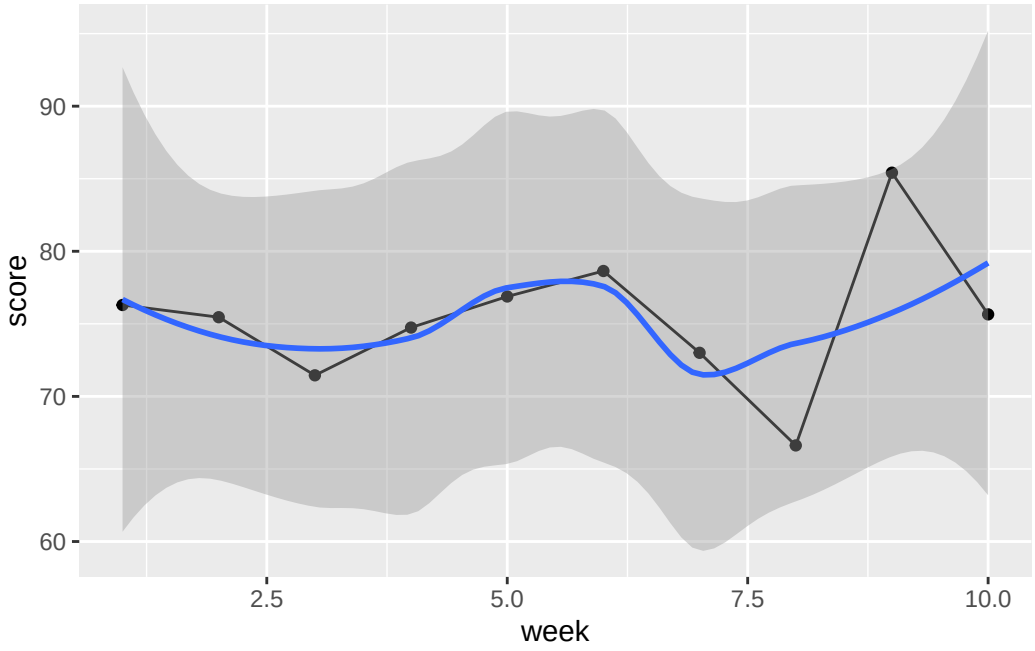
```
`geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



And remember, multiple **geoms** can be used together.

```
ggplot(data = student_data, aes(x = week, y = score)) +
  geom_point() +
  geom_line() +
  geom_smooth()
```

```
`geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

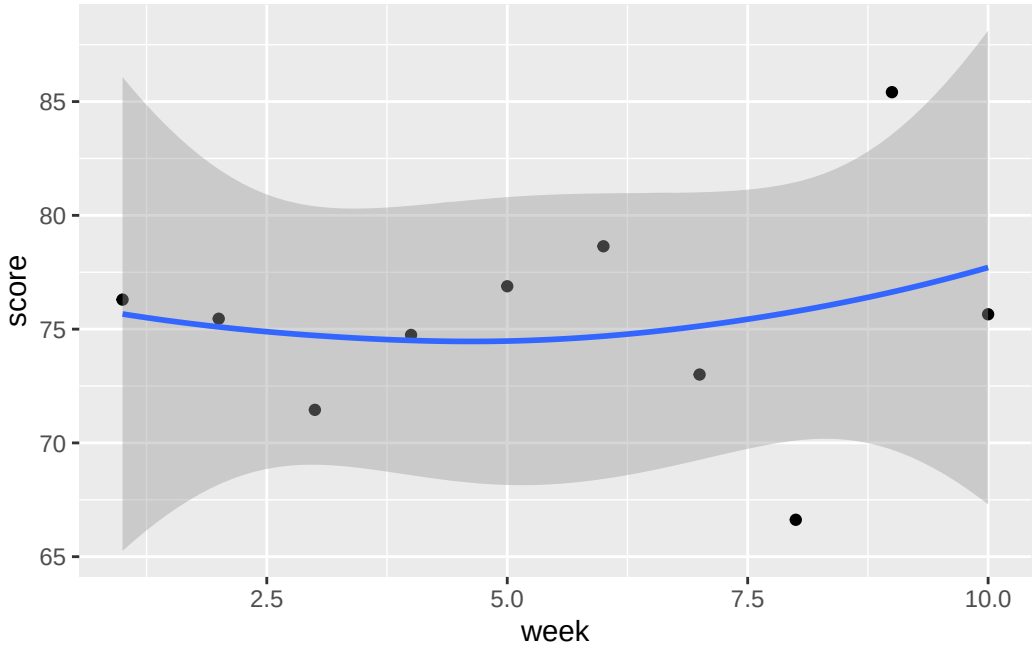


By default, the `geom_smooth` function estimates a loess regression on your data. You can adjust some of the parameters of this function, for example, the "smoothness" of the line using the `span` argument.

For example, to have it be smoother with a larger `span` (>1).

```
ggplot(data = student_data, aes(x = week, y = score)) +
  geom_point() +
  geom_smooth(span = 2)
```

```
`geom_smooth()` using method = 'loess' and formula 'y ~ x'
```



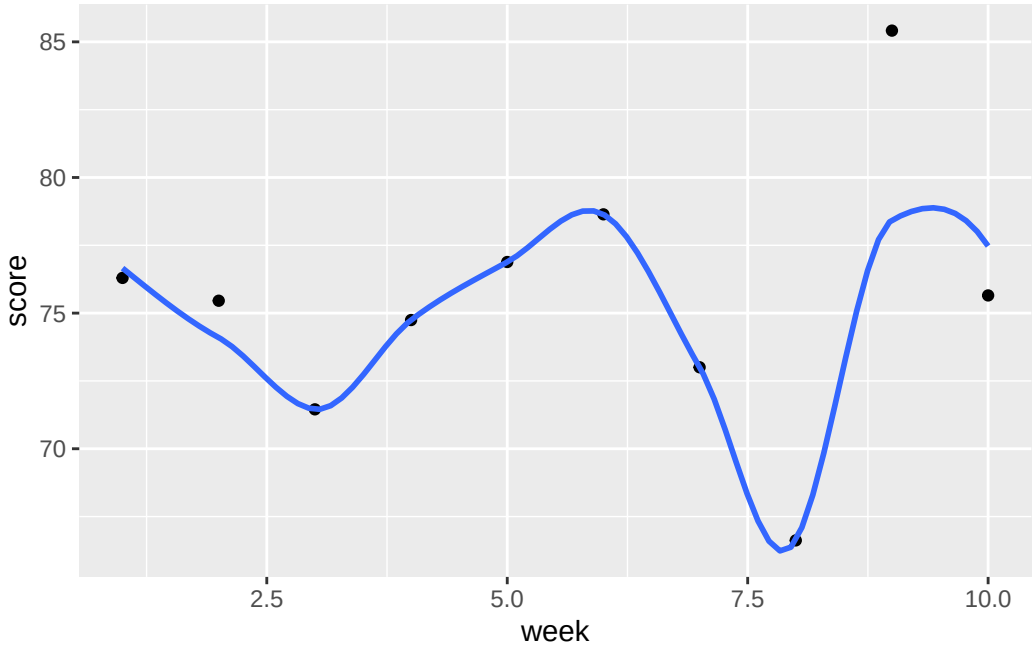
Or to be more "spiky" with a `span` < 1.

```
ggplot(data = student_data, aes(x = week, y = score)) +
  geom_point() +
  geom_smooth(span = .5)
```

```
`geom_smooth()` using method = 'loess' and formula 'y ~ x'
```

```
Warning in stats::qt(level/2 + 0.5, pred$df): NaNs produced
```

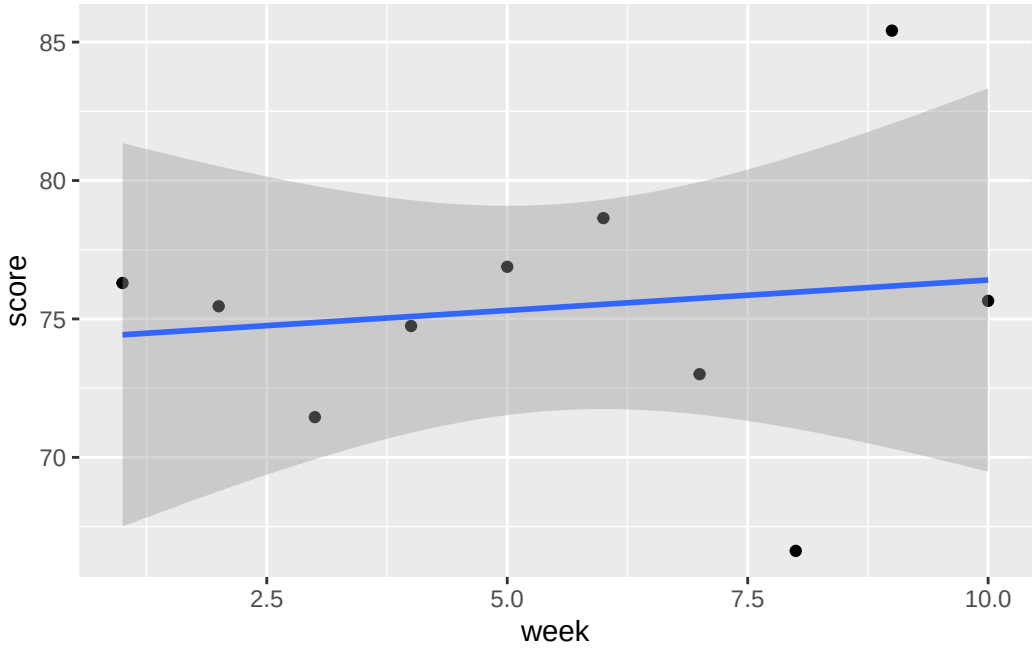
```
Warning in max(ids, na.rm = TRUE): no non-missing arguments to max; returning
-Inf
```



In addition to a loess regression, there are other methods that can be used and accessed via the `method` argument. For example, you can fit a linear model.

```
ggplot(data = student_data, aes(x = week, y = score)) +
  geom_point() +
  geom_smooth(method = "lm")
```

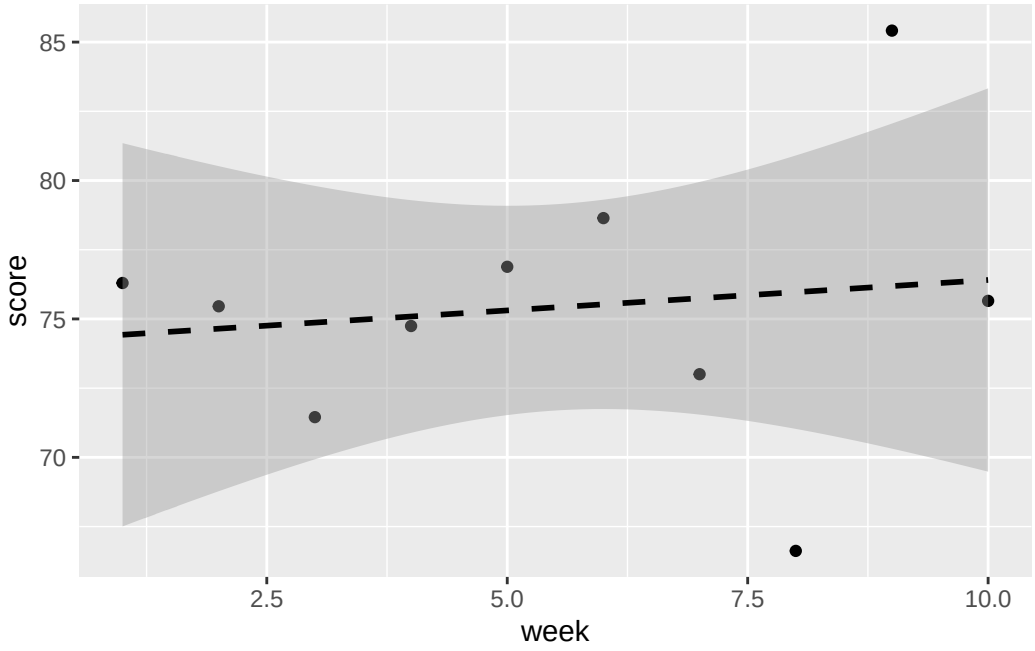
```
`geom_smooth()` using formula 'y ~ x'
```



Just like the other geoms, they can be customized in a variety of ways. For example, we can change the color of the line.

```
ggplot(data = student_data, aes(x = week, y = score)) +
  geom_point() +
  geom_smooth(method = "lm", color = "black", linetype = "dashed")
```

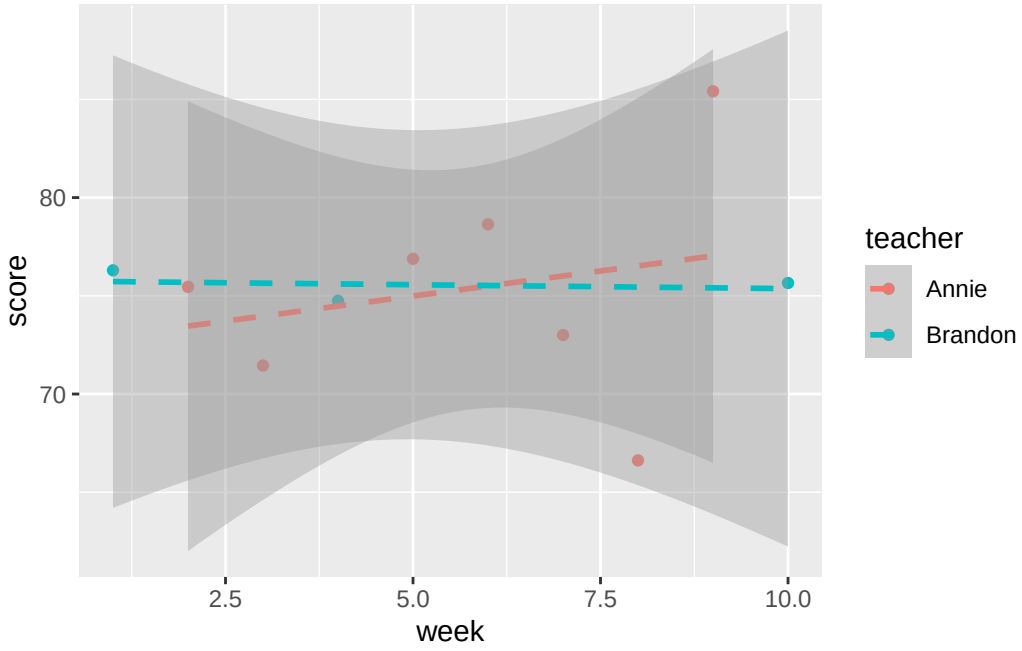
```
`geom_smooth()` using formula 'y ~ x'
```



Or, we can fit these regression on different groups.

```
ggplot(data = student_data, aes(x = week, y = score, color = teacher)) +
  geom_point() +
  geom_smooth(method = "lm", linetype = "dashed")
```

```
`geom_smooth()` using formula 'y ~ x'
```



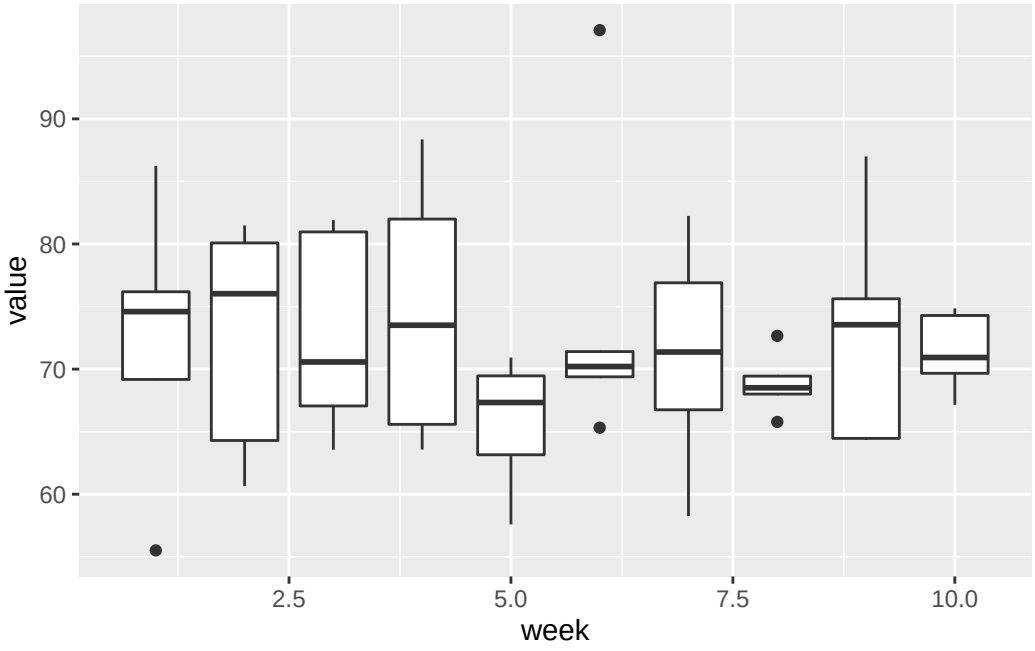
Other `geoms` can be used to visualize statistical outputs, such as uncertainty. There are a variety of functions to visualize uncertainty. Let's simulate some more data to add more subjects besides math and reading.

```
# Simulate single-participant data
weeks <- 1:10
scores <- rnorm(10, mean = 75, sd = 5)
scores_math <- rnorm(10, mean = 75, sd = 15)
scores_reading <- rnorm(10, mean = 65, sd = 5)
scores_science <- rnorm(10, mean = 70, sd = 5)
scores_art <- rnorm(10, mean = 75, sd = 5)
scores_music <- rnorm(10, mean = 75, sd = 10)

# Create a tibble
student_data <- tibble(week = weeks, scores_math = scores_math, scores_reading = scores_re

# Display the first few rows of the data
student_data_long <- student_data %>%
  pivot_longer(cols = c(scores_math, scores_reading, scores_science, scores_art, scores_mu

student_data_long
```

```
# A tibble: 50 x 4
    week teacher name           value
   <int> <fct>   <chr>          <dbl>
 1     1 Brandon scores_math     86.2
 2     1 Brandon scores_reading  55.5
 3     1 Brandon scores_science  69.2
 4     1 Brandon scores_art      74.6
 5     1 Brandon scores_music    76.2
 6     2 Annie   scores_math     81.5
 7     2 Annie   scores_reading  64.3
 8     2 Annie   scores_science  60.7
 9     2 Annie   scores_art      76.0
10     2 Annie   scores_music    80.1
# ... with 40 more rows
```

For example, on the long data, we can make box plots (make sure to use the `group` argument!).

```
ggplot(data = student_data_long, aes(x = week, y = value, group = week)) +
  geom_boxplot()
```



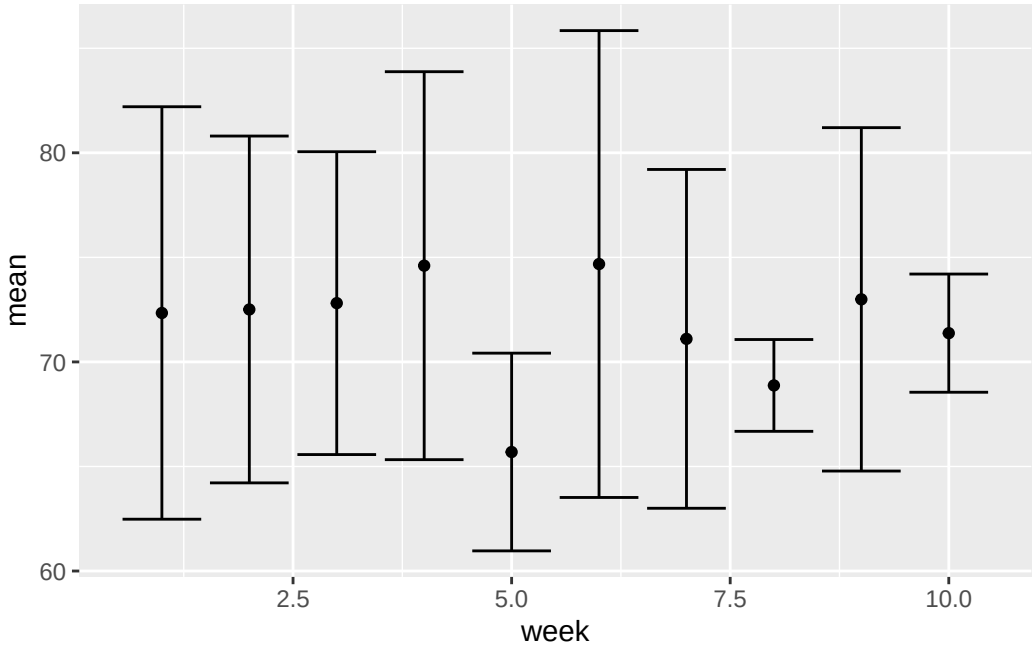
Let's continue and average our data down at the weekly level and estiamte uncertainty.

```r
student_data_summarized <- student_data_long %>%
  group_by(week) %>%
  summarize(mean = mean(value),
            se = sd(value) / sqrt(n())) %>%
  mutate(ci_low = mean - (se * 1.96),
         ci_high = mean + (se * 1.96))

student_data_summarized
```

```
# A tibble: 10 x 5
    week  mean    se ci_low ci_high
   <int> <dbl> <dbl>  <dbl>   <dbl>
 1     1  72.3  5.03   62.5    82.2
 2     2  72.5  4.23   64.2    80.8
 3     3  72.8  3.69   65.6    80.1
 4     4  74.6  4.73   65.3    83.9
 5     5  65.7  2.41   61.0    70.4
 6     6  74.7  5.70   63.5    85.8
 7     7  71.1  4.13   63.0    79.2
 8     8  68.9  1.12   66.7    71.1
 9     9  73.0  4.19   64.8    81.2
10    10  71.4  1.44   68.5    74.2
```

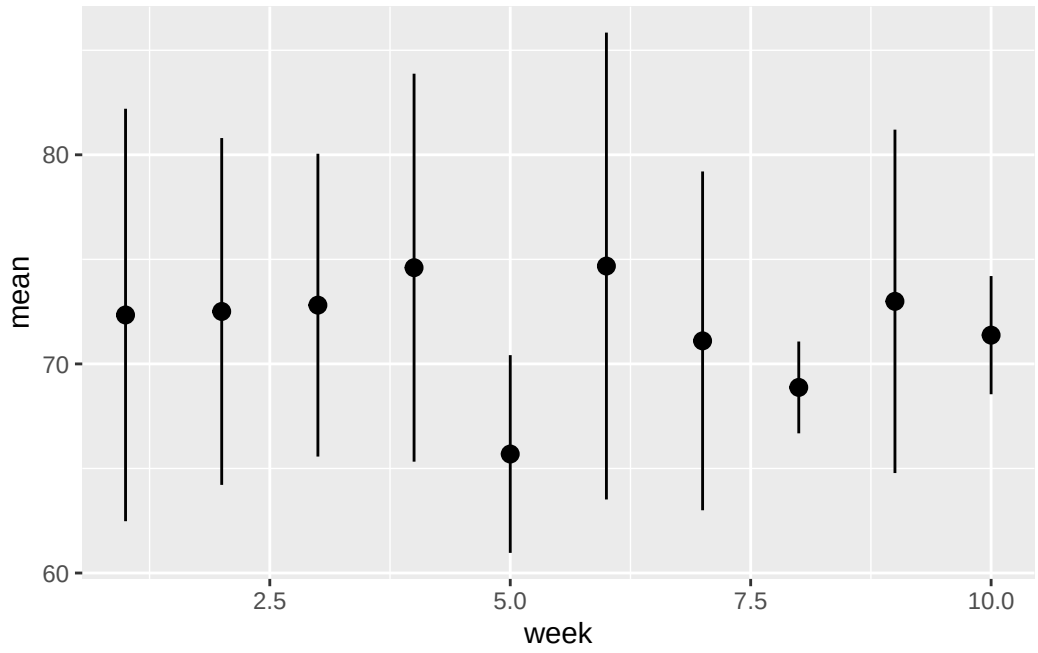
Note that some `geoms` require additional `aes` arguments. For example, here is how you would add error bars using our data.

```r
ggplot(data = student_data_summarized, aes(x = week, ymin = ci_low, y = mean, ymax = ci_hi
  geom_errorbar() +
  geom_point()
```

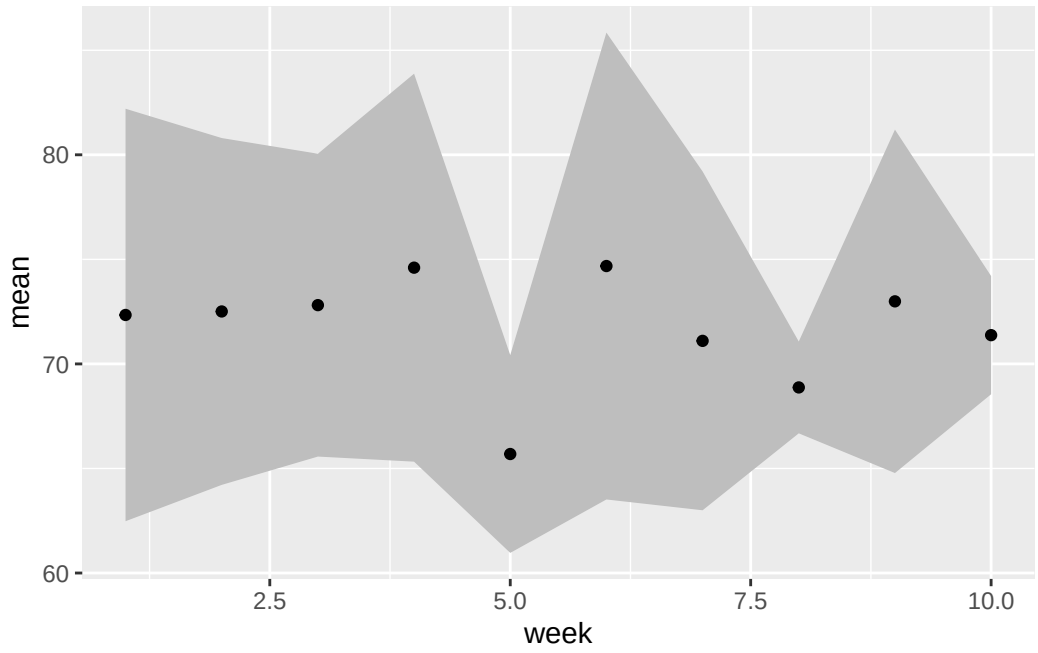


Other options include lineranges (pointranges) and ribbons. Note that the `aes` function is staying the same in all of these examples, we're just changing the `geoms` because the fundamental mapping of our data is remaining the same.

```
ggplot(data = student_data_summarized, aes(x = week, ymin = ci_low, y = mean, ymax = ci_hi
  geom_pointrange()
```




```
ggplot(data = student_data_summarized, aes(x = week, ymin = ci_low, y = mean, ymax = ci_hi
  geom_ribbon(fill = "grey") +
  geom_point()
```



Section 5: Upping Your `ggplot` Game

Annotations

Annotations are a way to add additional information to your graphs. Here, we will go over some of the most useful annotations.

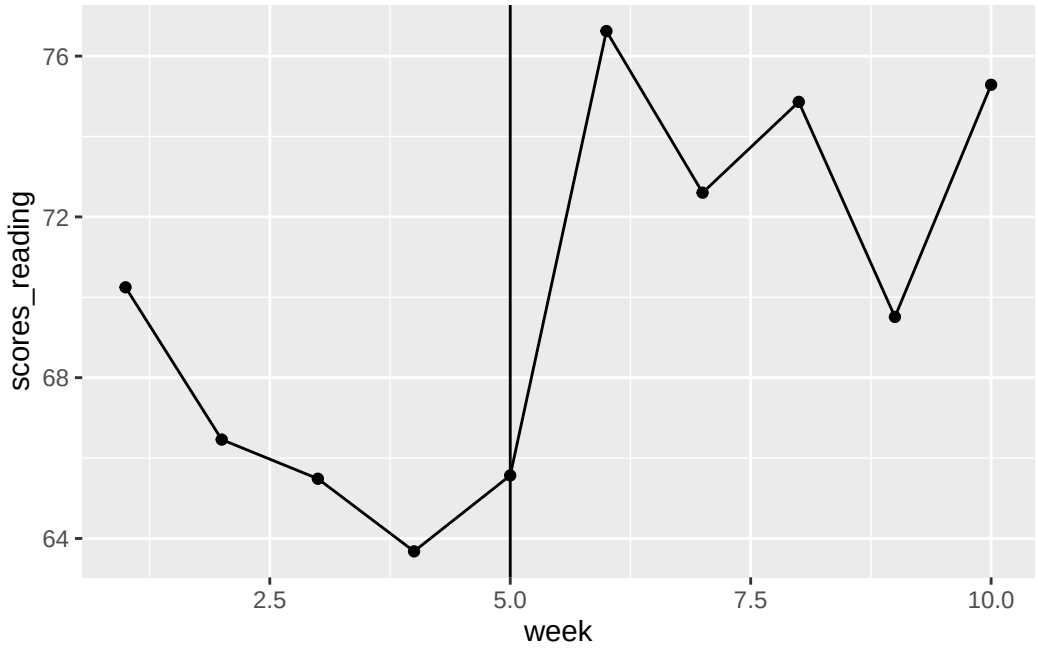
```
# Simulate single-participant data
weeks <- 1:10
scores <- rnorm(10, mean = 75, sd = 5)
scores_math <- rnorm(10, mean = 75, sd = 15)
scores_reading <- rnorm(10, mean = 65, sd = 5)
scores_science <- rnorm(10, mean = 70, sd = 5)
scores_art <- rnorm(10, mean = 75, sd = 5)
scores_music <- rnorm(10, mean = 75, sd = 10)

# Create a tibble
student_data <- tibble(week = weeks, scores_math = scores_math, scores_reading = scores_re

student_data <- student_data %>%
  mutate(period = if_else(week <= 5, "pre-treatment", "post-treatment"),
```

```
        scores_reading = if_else(period == "post-treatment", scores_reading + 10, scores_
        intervention = if_else(week == 5, "Intervention", NA))
```

Vertical Lines

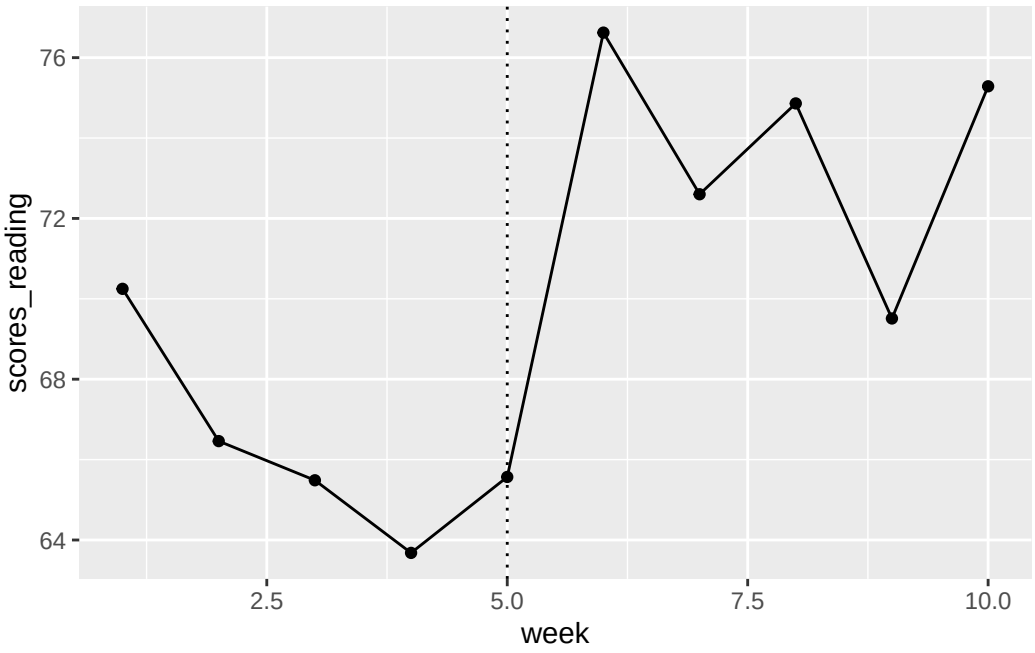
Let's imagine that we implemented an intervention on week 5 and we want to visualize this intervention using a vertical line. We can use the `geom_vline` function. You can set the x-intercept of the vertical line using the `xintercept` argument in `aes`.

```
ggplot(data = student_data, aes(x = week, y = scores_reading)) +
  geom_line() +
  geom_point() +
  geom_vline(aes(xintercept = 5))
```



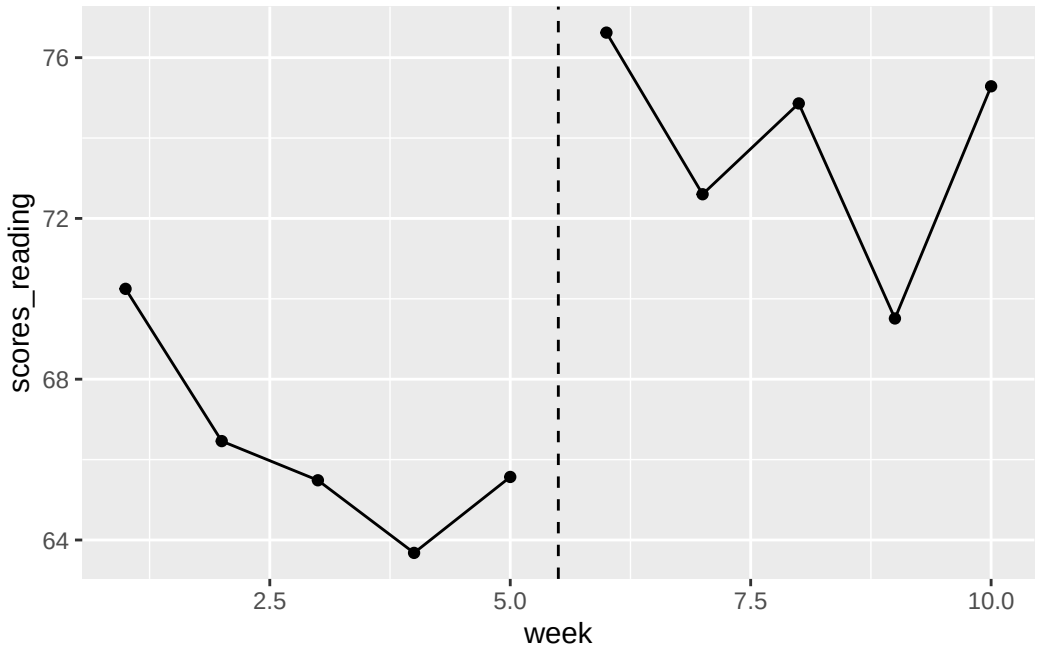
These lines can be customized just like other graphical components.

```
ggplot(data = student_data, aes(x = week, y = scores_reading)) +
  geom_line() +
  geom_point() +
  geom_vline(aes(xintercept = 5), linetype = "dotted")
```



Remember, we can always use the `group` argument in the `aes` function to separate groups.

```
ggplot(data = student_data, aes(x = week, y = scores_reading, group = period)) +
  geom_line() +
  geom_point() +
  geom_vline(aes(xintercept = 5.5), linetype = "dashed")
```



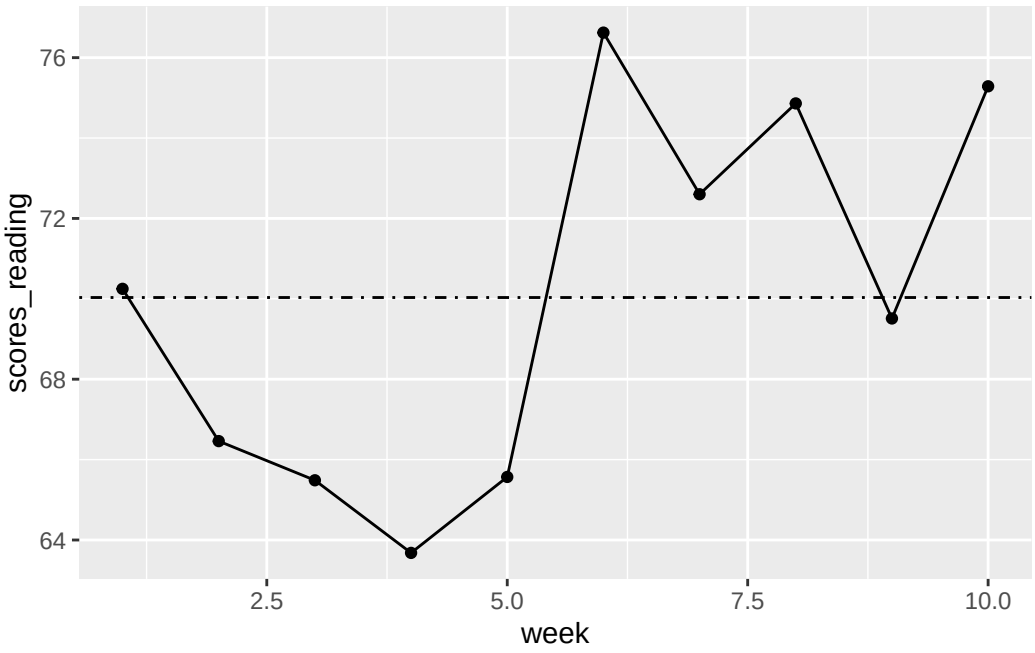
Horizontal Lines

You can add horizontal lines via the `geom_hline` function. For example here, we will calculate the mean of the scores and plot the mean as a horizontal line.

```
scores_reading_mean <- student_data %>%
  summarize(mean = mean(scores_reading)) %>%
  pull(mean)

scores_reading_mean
```

```
[1] 70.03247
```

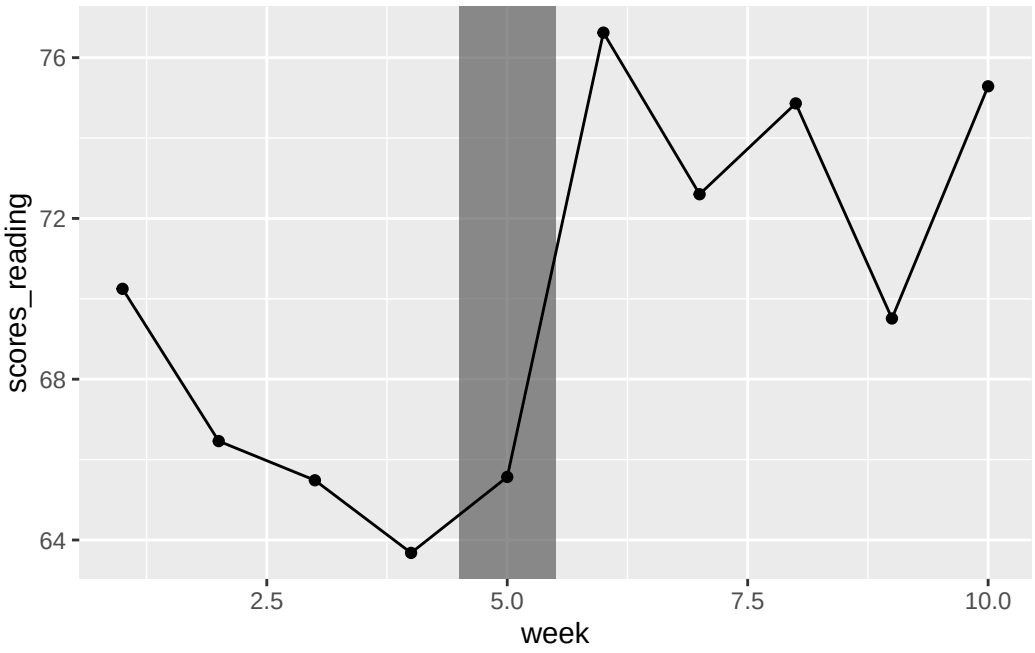
```
ggplot(data = student_data, aes(x = week, y = scores_reading)) +
  geom_line() +
  geom_point() +
  geom_hline(aes(yintercept = scores_reading_mean), linetype = "dotdash")
```



Rectangles

You can also draw shapes, for example, rectangles. Here, we draw a rectangle and shade it to represent the period in which the intervention was delivered.

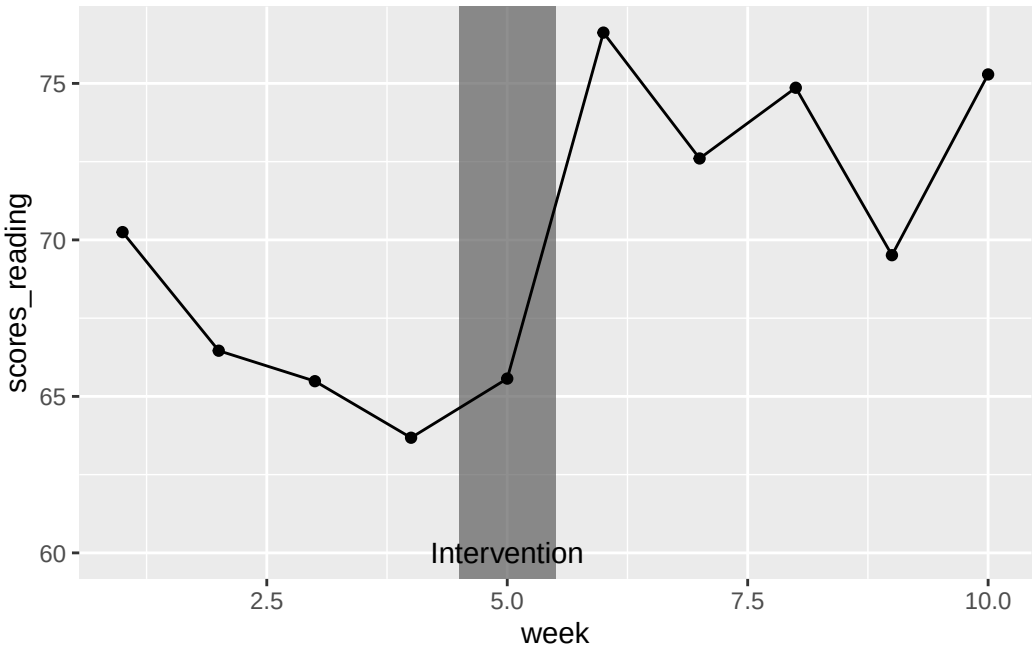
```
ggplot(data = student_data, aes(x = week, y = scores_reading)) +
  geom_rect(aes(xmin = 4.5, xmax = 5.5, ymin = -Inf, ymax = Inf), alpha = .1) +
  geom_line() +
  geom_point()
```



Text

We can also add text.

```
ggplot(data = student_data, aes(x = week, y = scores_reading)) +
  geom_rect(aes(xmin = 4.5, xmax = 5.5, ymin = -Inf, ymax = Inf), alpha = .1) +
  geom_line() +
  geom_point() +
  annotate("text", x = 5, y = 60, label = "Intervention")
```

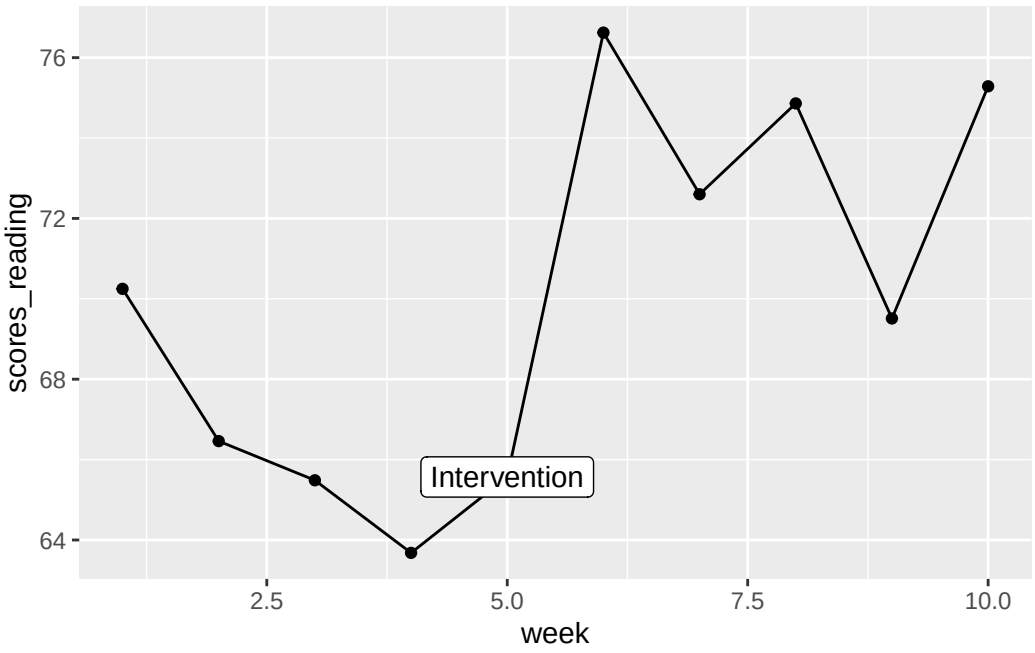


Point Labels

We can also add labels. Make sure you use the `labels` argument in `aes`.

```
ggplot(data = student_data, aes(x = week, y = scores_reading, label = intervention)) +
  geom_line() +
  geom_point() +
  geom_label()
```

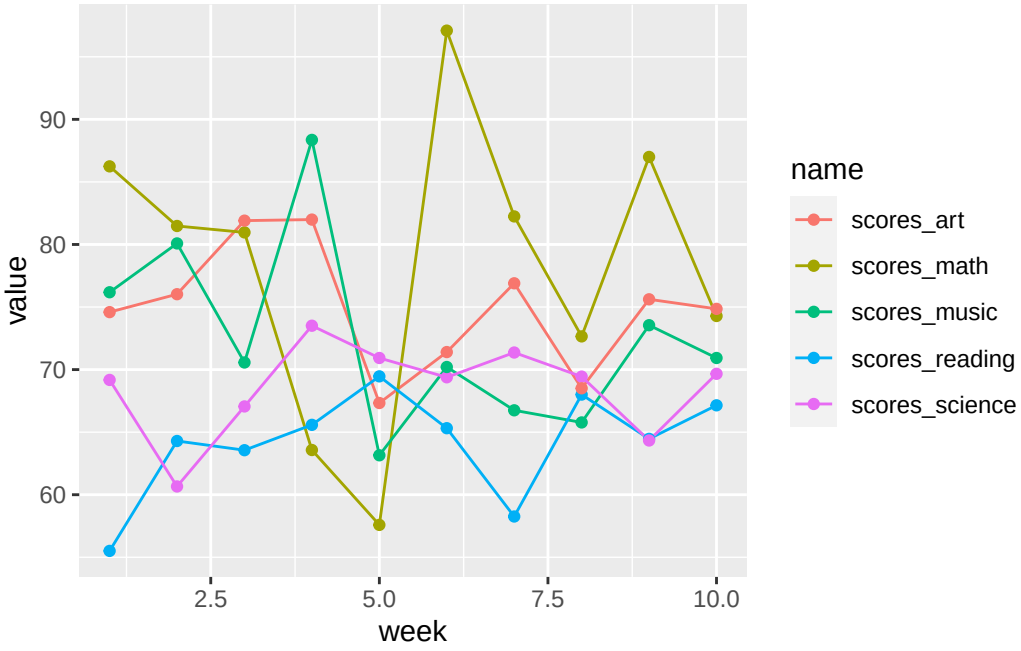
```
Warning: Removed 9 rows containing missing values (geom_label).
```



Faceting

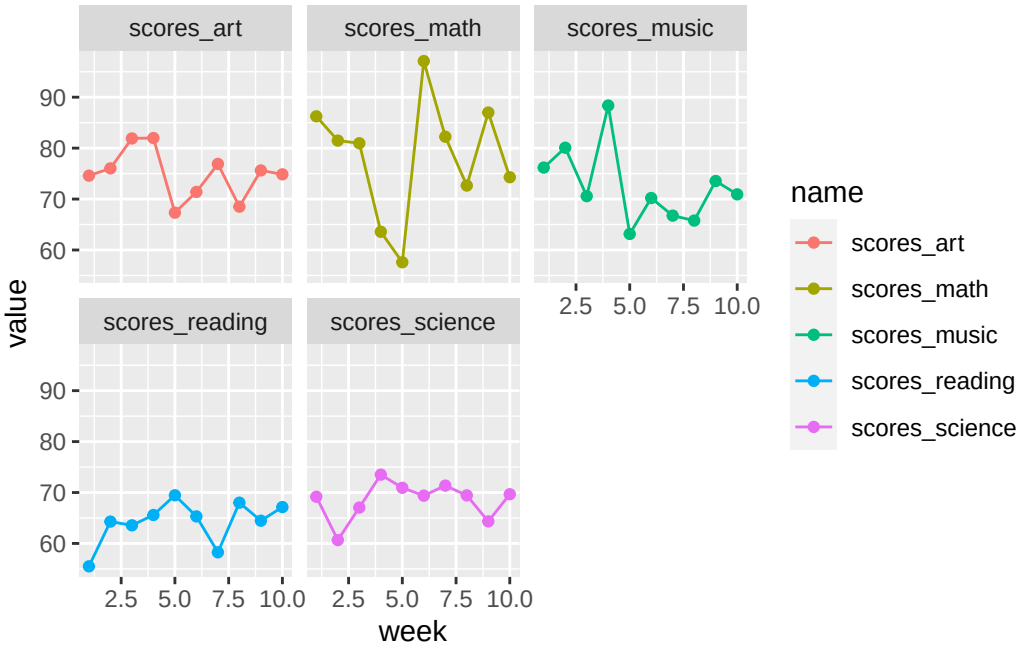
Faceting allows you to break your data up into different plots on a grid. For example, if we wanted to visualize all the different scores, we would have a messy graph that you can't make much out of.

```
ggplot(data = student_data_long, aes(x = week, y = value, color = name)) +
  geom_line() +
  geom_point()
```



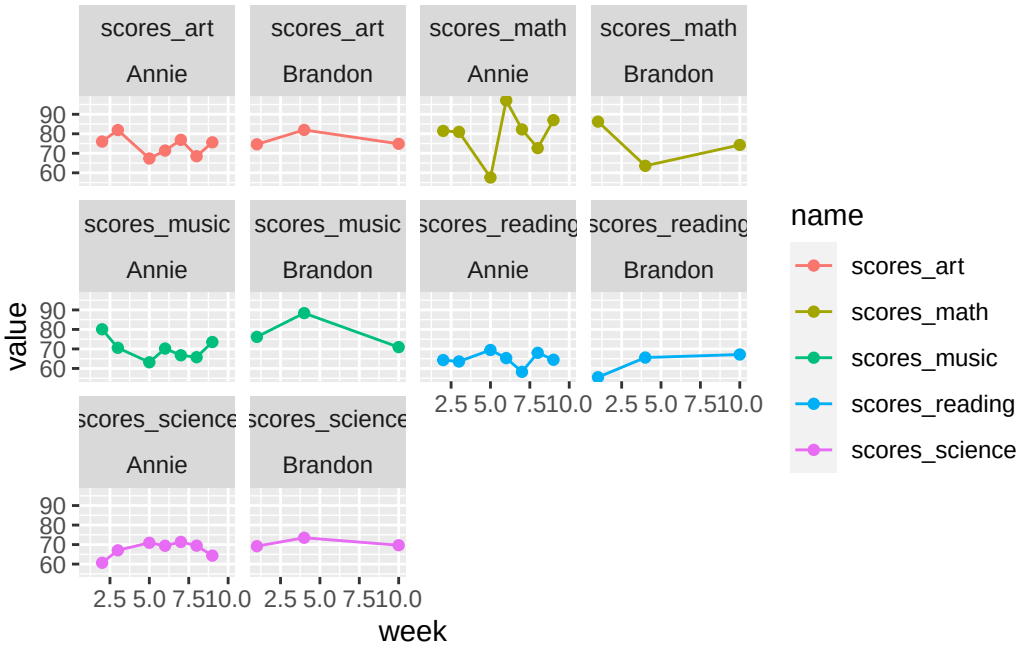
So we can "facet" the data into smaller individual plots. You can facet by a single variable.

```
ggplot(data = student_data_long, aes(x = week, y = value, color = name)) +
  geom_line() +
  geom_point() +
  facet_wrap(name~.)
```



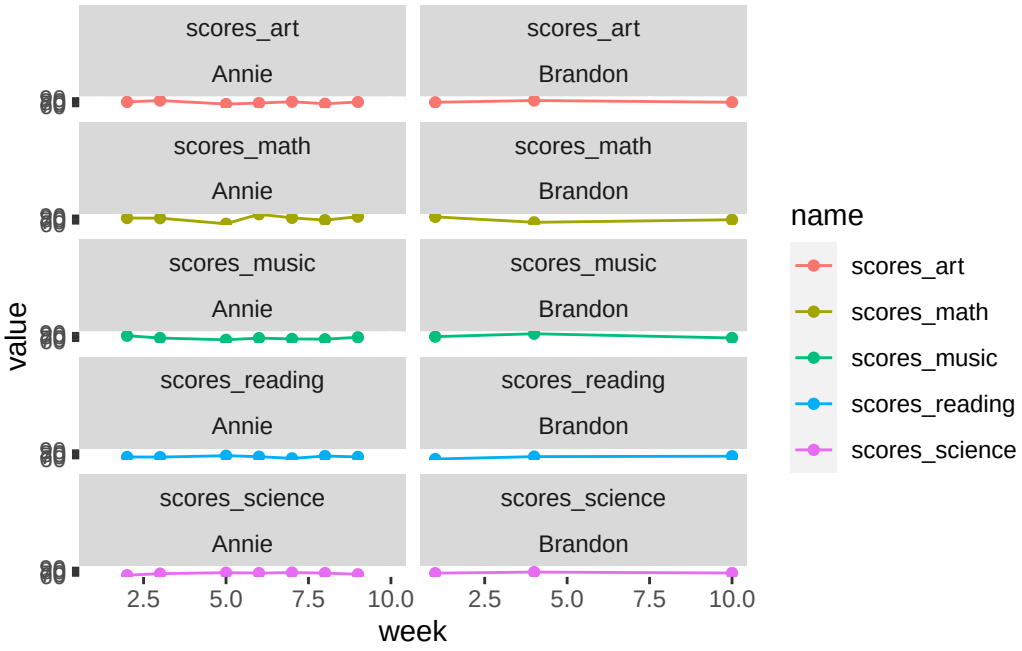
Or by multiple variables.

```
ggplot(data = student_data_long, aes(x = week, y = value, color = name)) +
  geom_line() +
  geom_point() +
  facet_wrap(name~teacher)
```



You can easily adjust the number of columns to fit your analysis needs. For example, in this graph, you can more easily compare scores across teachers for the same subject.

```
ggplot(data = student_data_long, aes(x = week, y = value, color = name)) +
  geom_line() +
  geom_point() +
  facet_wrap(name~teacher, ncol = 2)
```



Multiple-Baseline Plots

One common type of data visualization in applied behavior analysis is multiple-baseline plots. In these plots, we want to visualize how multiple participants respond to baselines at different time periods. Here is how you can create such a plot.

First, let's simulate some data.

```
# Define the number of weeks and participants
num_weeks <- 20
num_participants <- 3

# Create a data frame to store the simulated data
ab_data <- tibble(
  participant = rep(1:num_participants, each = num_weeks),
  week = rep(1:num_weeks, times = num_participants),
  score = rep(NA, num_participants * num_weeks),  # Initialize Score column
  intervention = rep(NA_character_, num_participants * num_weeks)  # Initialize Interventi
)

# Simulate the data
for (participant in 1:num_participants) {
```

```
  # Determine the intervention week for each participant
  intervention_week <- 5 * participant

  # Generate pre-intervention scores
  pre_intervention_scores <- rnorm(intervention_week, mean = 60, sd = 3)

  # Generate post-intervention scores with an increase
  post_intervention_scores <- rnorm(num_weeks - intervention_week, mean = 80, sd = 3)

  # Fill in the Score column for this participant
  ab_data$score[ab_data$participant == participant] <- c(pre_intervention_scores, post_int

  # Fill in the Intervention column for this participant
  ab_data$intervention[ab_data$participant == participant] <- ifelse(
    ab_data$week <= intervention_week,
    "Pre-Intervention",
    "Post-Intervention"
  )
}
```

```
Warning in ab_data$intervention[ab_data$participant == participant] <-
ifelse(ab_data$week <= : number of items to replace is not a multiple of
replacement length

Warning in ab_data$intervention[ab_data$participant == participant] <-
ifelse(ab_data$week <= : number of items to replace is not a multiple of
replacement length

Warning in ab_data$intervention[ab_data$participant == participant] <-
ifelse(ab_data$week <= : number of items to replace is not a multiple of
replacement length
```

```
# Display the first few rows of the simulated data
ab_data
```

```
# A tibble: 60 x 4
   participant  week score intervention
         <int> <int> <dbl> <chr>
 1           1     1  61.1 Pre-Intervention
 2           1     2  55.5 Pre-Intervention
```

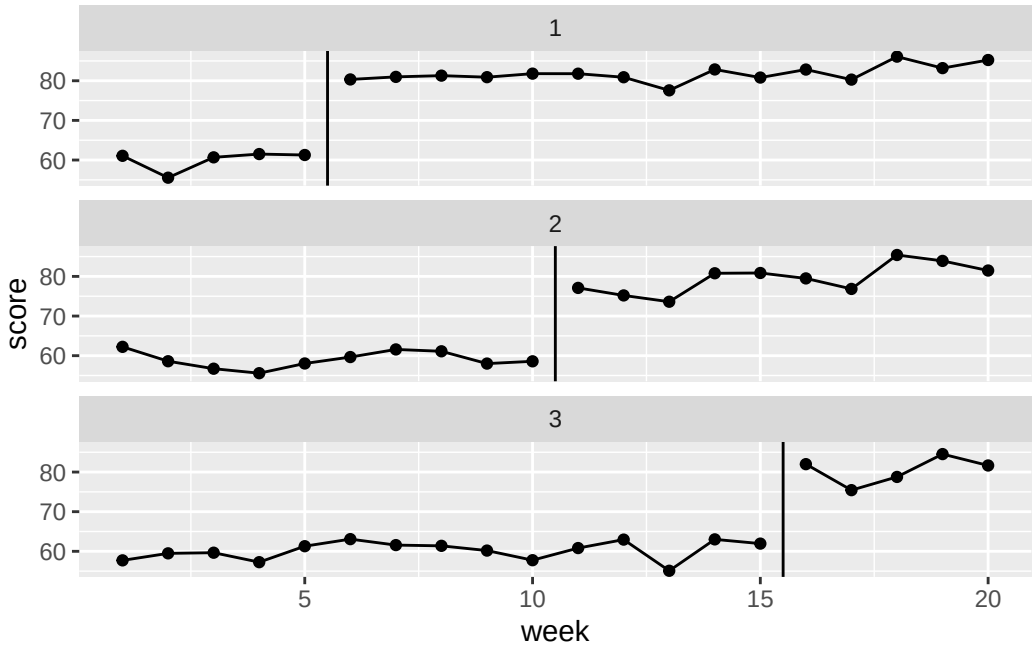
```
 3           1     3  60.7 Pre-Intervention
 4           1     4  61.5 Pre-Intervention
 5           1     5  61.3 Pre-Intervention
 6           1     6  80.3 Post-Intervention
 7           1     7  81.0 Post-Intervention
 8           1     8  81.3 Post-Intervention
 9           1     9  80.9 Post-Intervention
10           1    10  81.8 Post-Intervention
# ... with 50 more rows
```

Here we will dynamically calculate the intervention week for each of the participants. Note that you can create this tibble manually.

```
intervention_dates <- ab_data %>%
  group_by(participant) %>%
  filter(intervention == "Post-Intervention") %>%
  filter(week == min(week)) %>%
  mutate(week = week - 0.5)

intervention_dates
```

```
# A tibble: 3 x 4
# Groups:   participant [3]
  participant  week score intervention
        <int> <dbl> <dbl> <chr>
1           1   5.5  80.3 Post-Intervention
2           2  10.5  77.1 Post-Intervention
3           3  15.5  82.0 Post-Intervention
```

```
ggplot(data = ab_data, aes(x = week, y = score, group = intervention)) +
  geom_line() +
  geom_point() +
  facet_wrap(.~participant, ncol = 1) +
  geom_vline(data = intervention_dates, aes(xintercept = week))
```



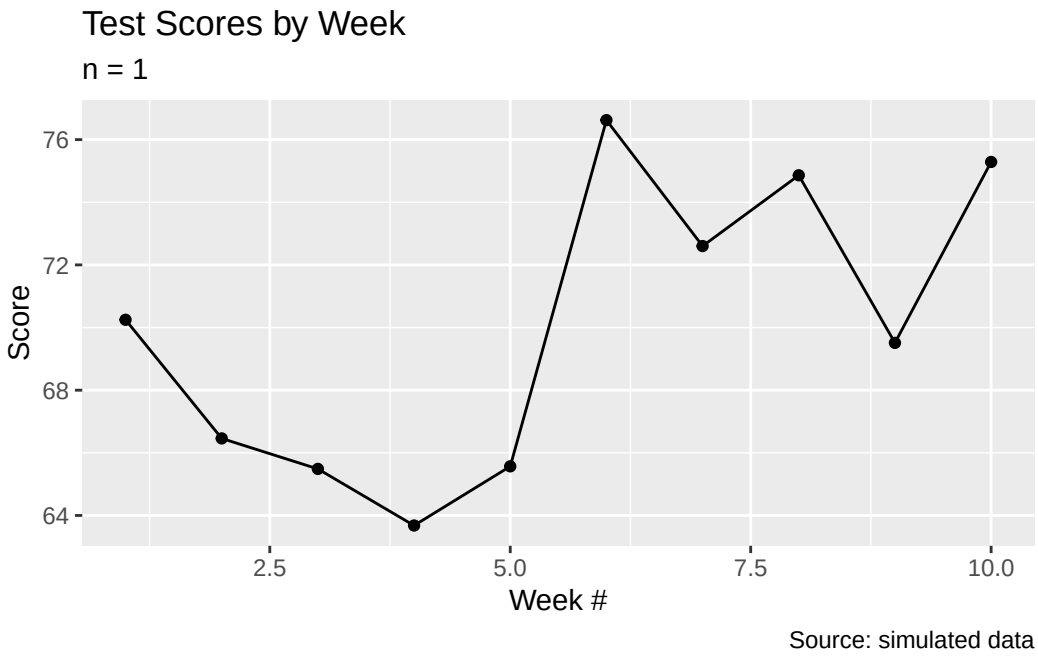
Section 6: Making your `ggplots` look nice

We've done a lot of work to make some visualizations for single-participant studies. However, the graphs can definitely use some tweaking. Here, I'll show you some of the more common ones.

Plot Labels

We added labels to our graph earlier, but here they are again. The easiest way to add labels is to use the `labs` function. You don't have to use all of the options within the `labs` function as there are many to choose from.

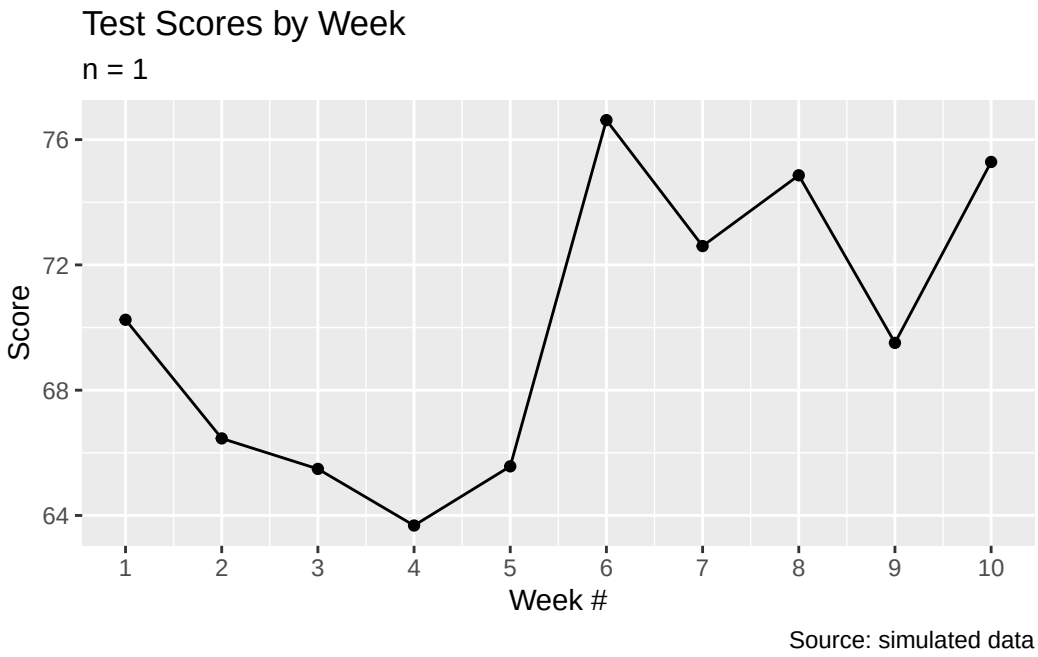
```
ggplot(data = student_data, aes(x = week, y = scores_reading)) +
  geom_point() +
  geom_line() +
  labs(title = "Test Scores by Week",
       subtitle = "n = 1",
       x = "Week #",
       y = "Score",
       caption = "Source: simulated data")
```



Working with Scales

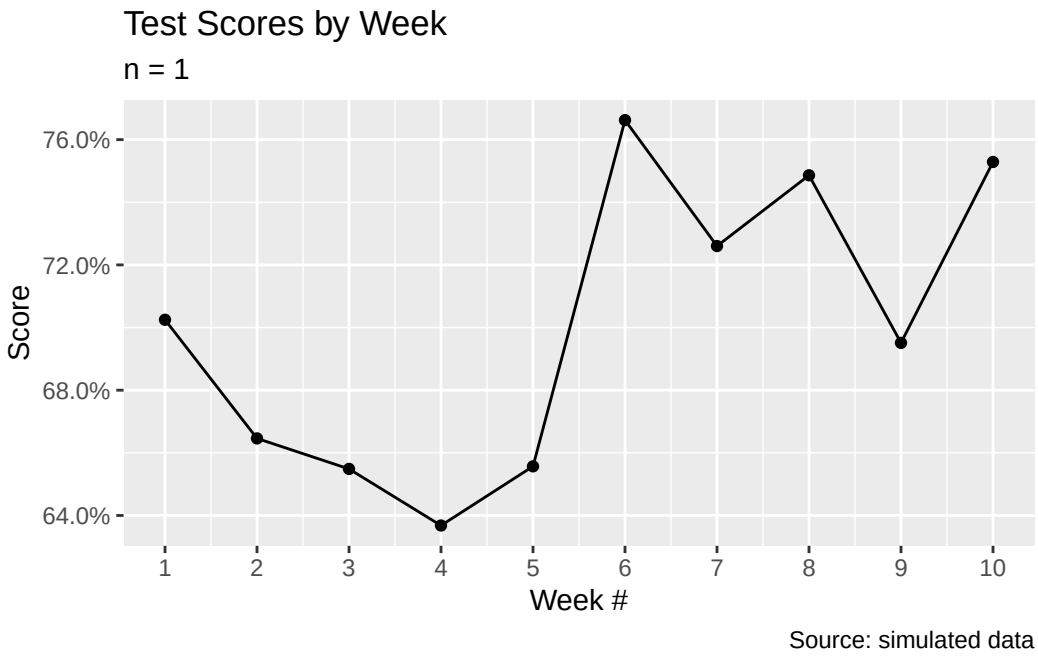
Another common thing you may want to adjust in your graph are the scales. For example, the x-axis doesn't look very good with the half numbers given that we are only working with full weeks. We will use the `scale_x_continuous` function to adjust the x-axis.

```
ggplot(data = student_data, aes(x = week, y = scores_reading)) +
  geom_point() +
  geom_line() +
  labs(title = "Test Scores by Week",
       subtitle = "n = 1",
       x = "Week #",
       y = "Score",
       caption = "Source: simulated data") +
  scale_x_continuous(breaks = seq(1, 10, 1))
```



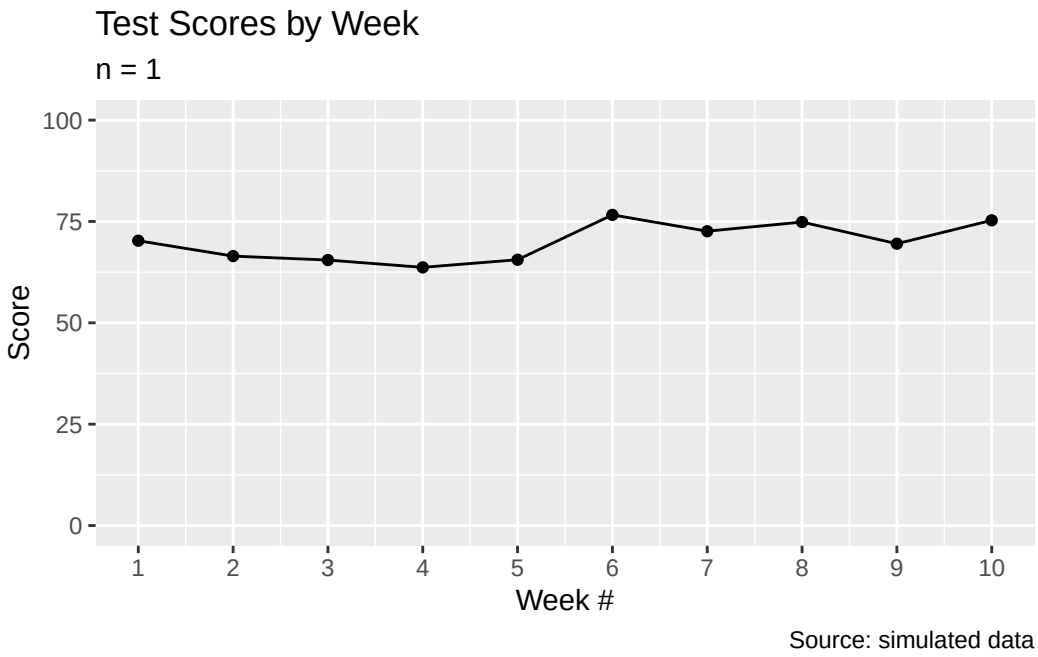
You can also adjust the label format. For example, we can make the y-axis into percentages using the `scale_y_continuous` function.

```
ggplot(data = student_data, aes(x = week, y = scores_reading*.01)) +
  geom_point() +
  geom_line() +
  labs(title = "Test Scores by Week",
       subtitle = "n = 1",
       x = "Week #",
       y = "Score",
       caption = "Source: simulated data") +
  scale_x_continuous(breaks = seq(1, 10, 1)) +
  scale_y_continuous(labels = scales::percent_format())
```



You can also adjust the "window" of the graph via its limits. For example, let's say we wanted the y-axis to go from 0-100.

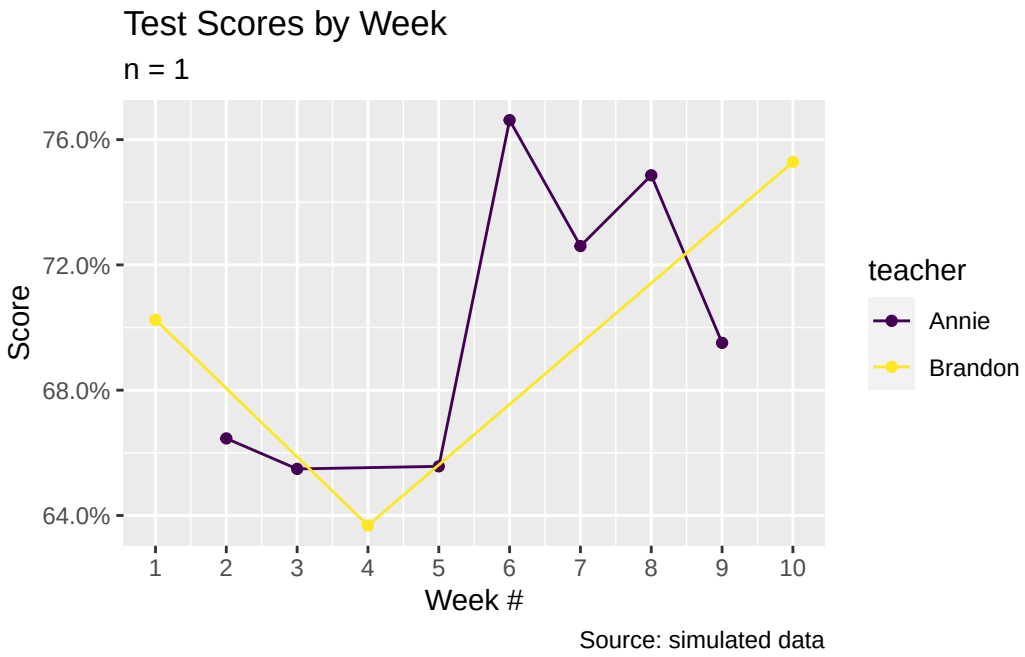
```
ggplot(data = student_data, aes(x = week, y = scores_reading)) +
  geom_point() +
  geom_line() +
  labs(title = "Test Scores by Week",
       subtitle = "n = 1",
       x = "Week #",
       y = "Score",
       caption = "Source: simulated data") +
  scale_x_continuous(breaks = seq(1, 10, 1)) +
  scale_y_continuous(limits = c(0, 100))
```



Working with Colors

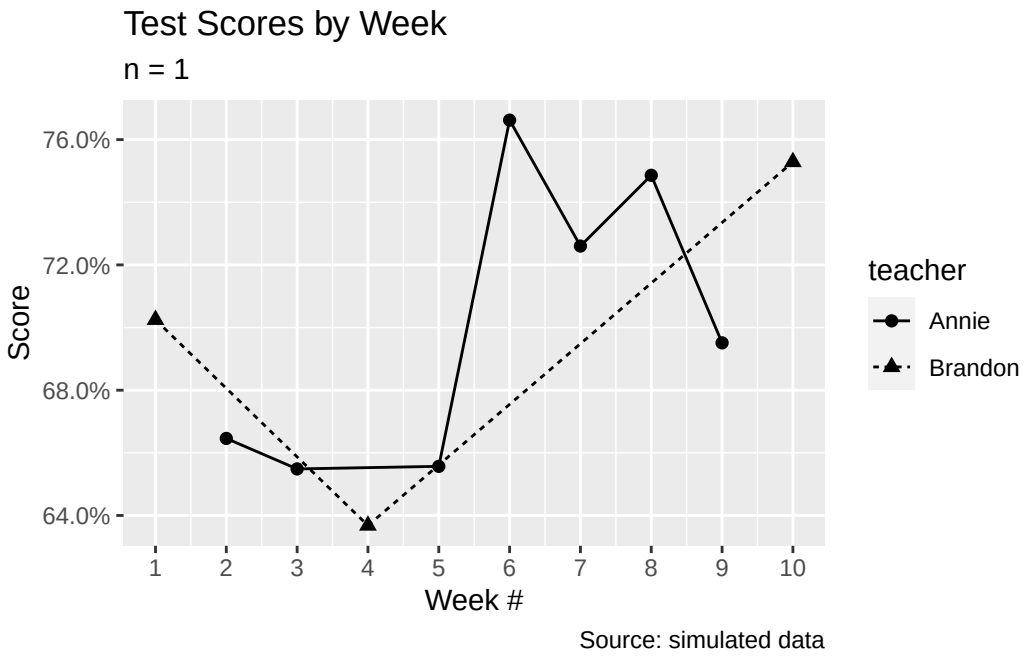
One of our goals in making beautiful ggplots should be to make our graphs as accessible as possible. One important way to make our graphs more accessible is by using color-blind-friendly colors. The default colors in R are not great, so we can use the `viridis` colors that are designed to be color-blind friendly.

```
ggplot(data = student_data, aes(x = week, y = scores_reading*.01, color = teacher)) +
  geom_point() +
  geom_line() +
  labs(title = "Test Scores by Week",
       subtitle = "n = 1",
       x = "Week #",
       y = "Score",
       caption = "Source: simulated data") +
  scale_x_continuous(breaks = seq(1, 10, 1)) +
  scale_y_continuous(labels = scales::percent_format()) +
  scale_color_viridis_d()
```



Of course, another option is to use shapes instead of colors when possible.

```
ggplot(data = student_data, aes(x = week, y = scores_reading*.01, shape = teacher, linetyp
  geom_point(size = 2) +
  geom_line() +
  labs(title = "Test Scores by Week",
       subtitle = "n = 1",
       x = "Week #",
       y = "Score",
       caption = "Source: simulated data") +
  scale_x_continuous(breaks = seq(1, 10, 1)) +
  scale_y_continuous(labels = scales::percent_format())
```



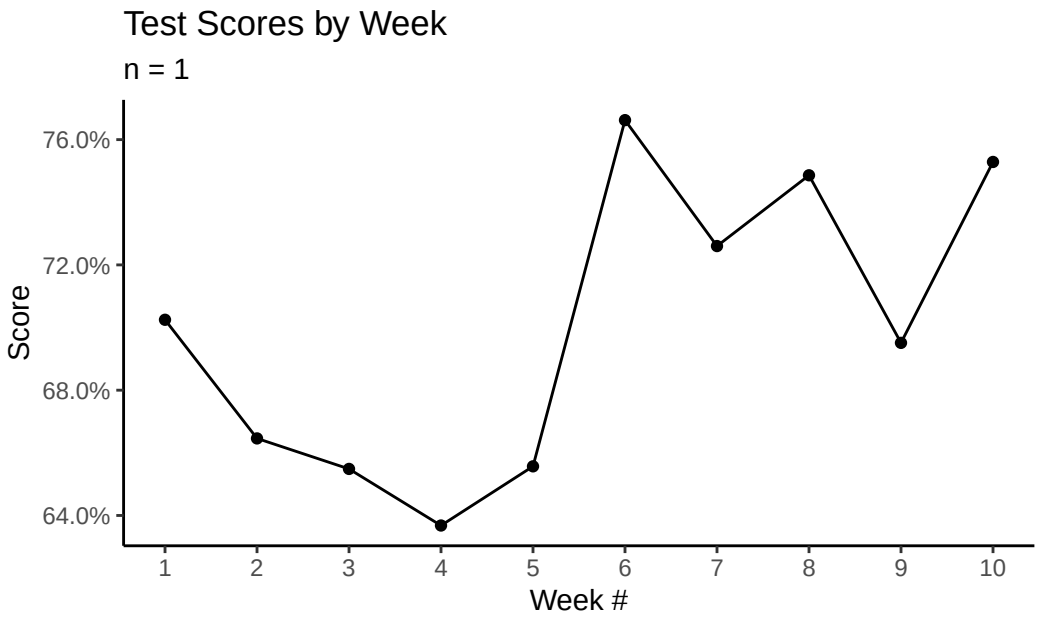
Themes

Finally, we are going to talk about themes. Themes allow you to change the background visuals of your graph. For example, by default, you get a grey background with white grid lines—something I personally really dislike.

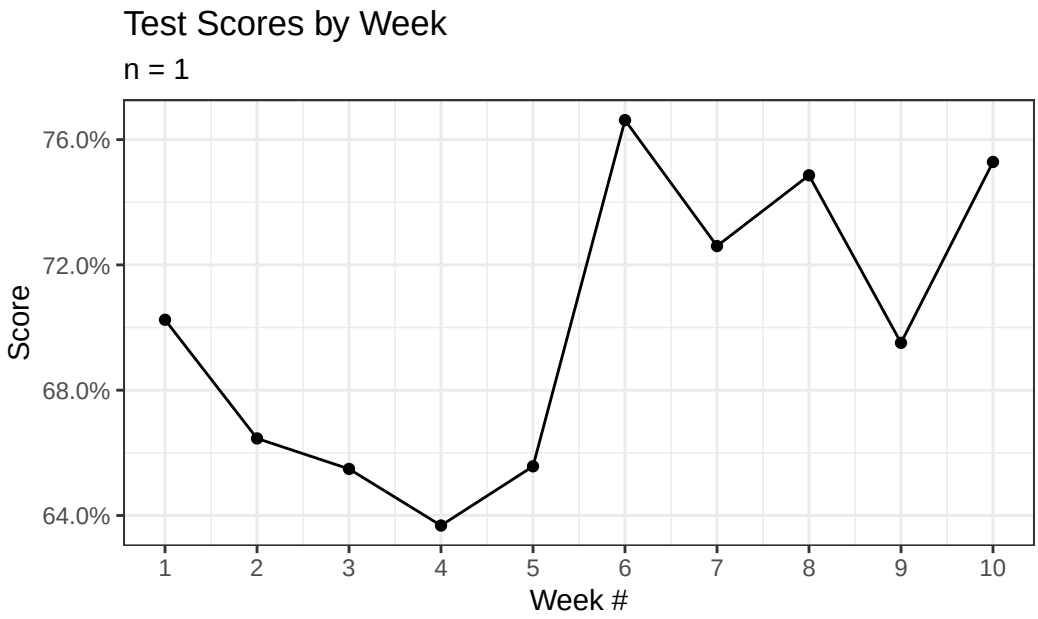
Luckily for us, `ggplot` comes with some pre-made themes we can use.

My personal favorite for academic graphs is `theme_classic` but other popular ones include `theme_bw` and `theme_minimal`

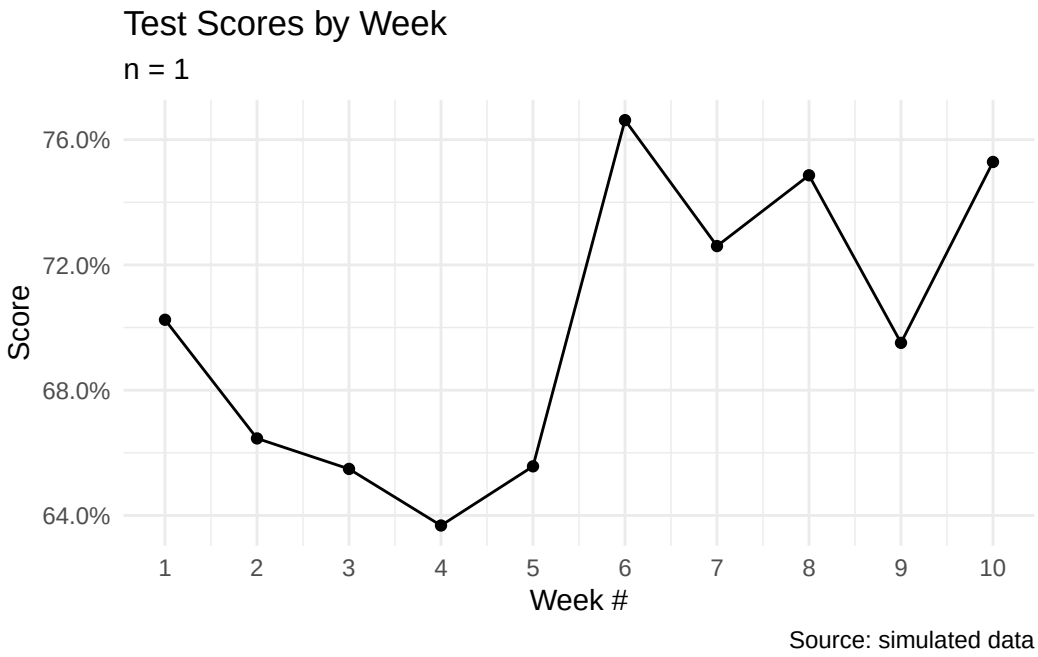
```
ggplot(data = student_data, aes(x = week, y = scores_reading*.01)) +
  geom_point() +
  geom_line() +
  labs(title = "Test Scores by Week",
       subtitle = "n = 1",
       x = "Week #",
       y = "Score",
       caption = "Source: simulated data") +
  scale_x_continuous(breaks = seq(1, 10, 1)) +
  scale_y_continuous(labels = scales::percent_format()) +
  theme_classic()
```




```
ggplot(data = student_data, aes(x = week, y = scores_reading*.01)) +
  geom_point() +
  geom_line() +
  labs(title = "Test Scores by Week",
       subtitle = "n = 1",
       x = "Week #",
       y = "Score",
       caption = "Source: simulated data") +
  scale_x_continuous(breaks = seq(1, 10, 1)) +
  scale_y_continuous(labels = scales::percent_format()) +
  theme_bw()
```



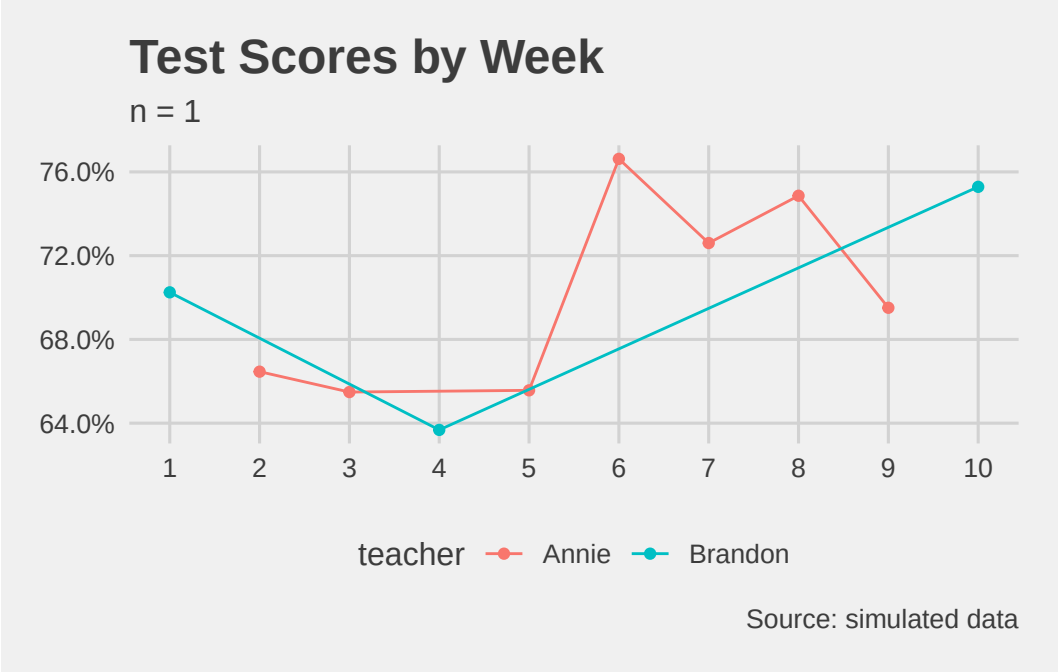

```
ggplot(data = student_data, aes(x = week, y = scores_reading*.01)) +
  geom_point() +
  geom_line() +
  labs(title = "Test Scores by Week",
       subtitle = "n = 1",
       x = "Week #",
       y = "Score",
       caption = "Source: simulated data") +
  scale_x_continuous(breaks = seq(1, 10, 1)) +
  scale_y_continuous(labels = scales::percent_format()) +
  theme_minimal()
```



There are many other themes that can be found in the `ggthemes` package. For example, you can make your plots look like they were made by FiveThirtyEight.

```
install.packages("ggthemes")
```

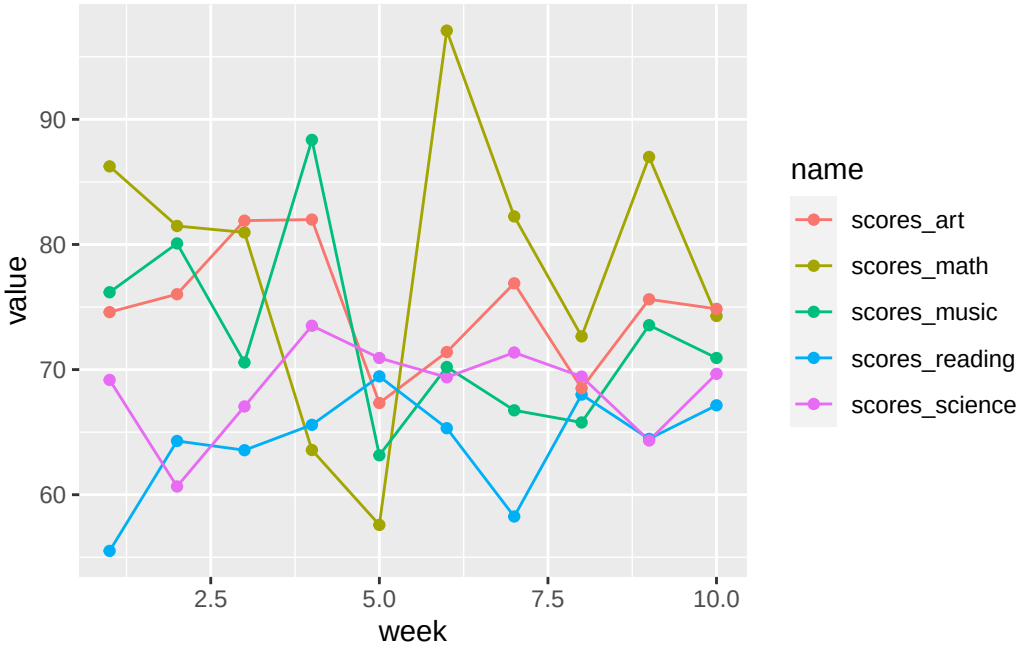
```
ggplot(data = student_data, aes(x = week, y = scores_reading*.01, color = teacher)) +
  geom_point() +
  geom_line() +
  labs(title = "Test Scores by Week",
       subtitle = "n = 1",
       x = "Week #",
       y = "Score",
       caption = "Source: simulated data") +
  scale_x_continuous(breaks = seq(1, 10, 1)) +
  scale_y_continuous(labels = scales::percent_format()) +
  ggthemes::theme_fivethirtyeight()
```



You can also customize individual components of the theme using the `theme` function. There are many, many options. So, I encourage you to explore those possibilities on their. However, one that I use often is adjusting the legend position.

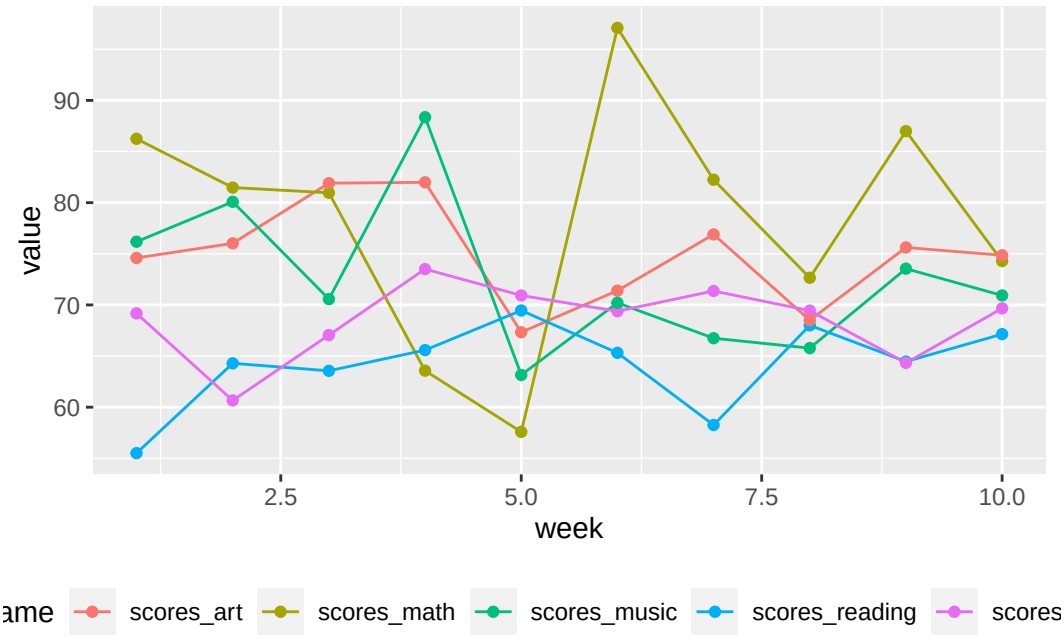
By default, the legend is put to the right of the graph.

```
ggplot(data = student_data_long, aes(x = week, y = value, color = name)) +
  geom_line() +
  geom_point()
```



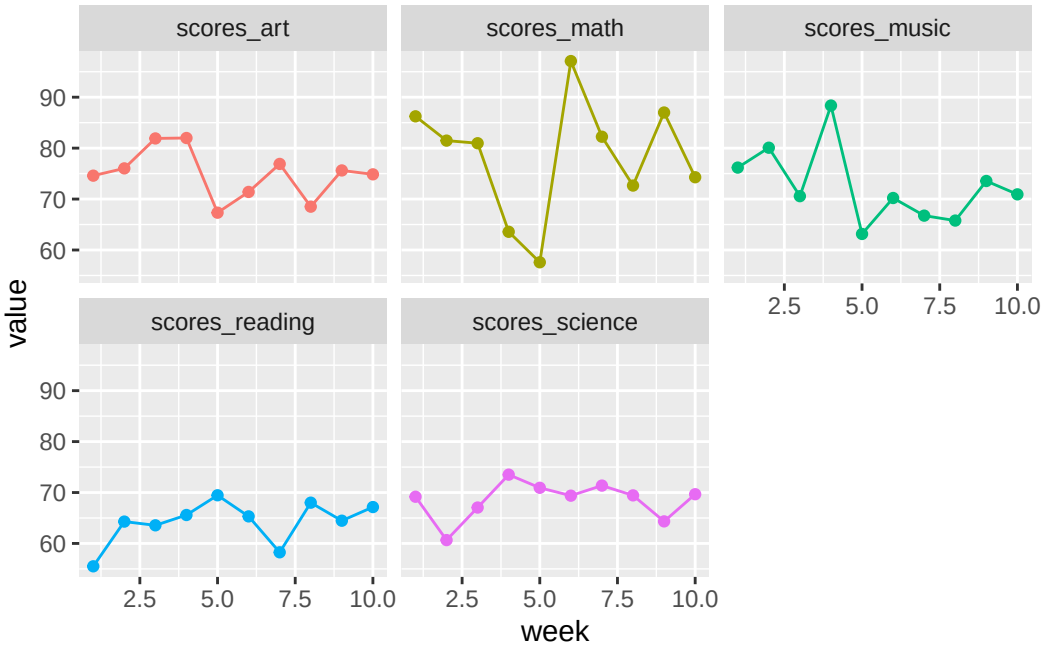
But you can change its location.

```
ggplot(data = student_data_long, aes(x = week, y = value, color = name)) +
  geom_line() +
  geom_point() +
  theme(legend.position = "bottom")
```



Or you can get rid of it all together.

```
ggplot(data = student_data_long, aes(x = week, y = value, color = name)) +
  geom_line() +
  geom_point() +
  facet_wrap(name~.) +
  theme(legend.position = "none")
```



Section 7: Hands-On Example

Here's a hands-on example that incorporates the entire process of reading in one's data, doing some checks for face validity, and visualizing it.

```
case_1 <- read_csv("data/sample1.csv")
```

```
Rows: 15 Columns: 4
-- Column specification --------------------------------------------------------
Delimiter: ","
chr (2): session_type, session_name
dbl (2): session_number, data

i Use `spec()` to retrieve the full column specification for this data.
i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
case_1 %>%
  group_by(session_name) %>%
  summarize(mean = mean(data))
```

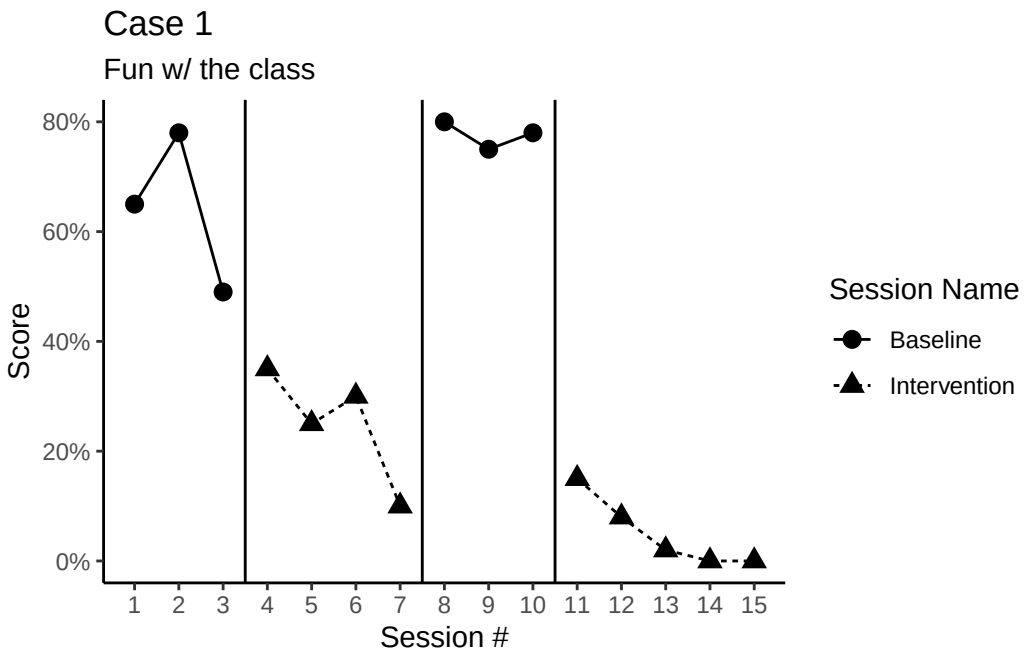
```
# A tibble: 2 x 2
  session_name  mean
  <chr>        <dbl>
1 Baseline     0.708
2 Intervention 0.139
```

```
case_1 %>%
  group_by(session_name, session_type) %>%
  summarize(mean = mean(data))
```

```
`summarise()` has grouped output by 'session_name'. You can override using the
`.groups` argument.
```

```
# A tibble: 4 x 3
# Groups:   session_name [2]
  session_name session_type  mean
  <chr>        <chr>        <dbl>
1 Baseline     A1           0.64
2 Baseline     A2           0.777
3 Intervention B1           0.25
4 Intervention B2           0.05
```

```
period_breakers <- tibble(break_point = c(3.5, 7.5, 10.5))

ggplot(data = case_1, aes(x = session_number, y = data, group = session_type, shape = sess
  geom_line() +
  geom_point(size = 3) +
  geom_vline(data = period_breakers, aes(xintercept = break_point)) +
  scale_x_continuous(breaks = seq(1, 15, 1)) +
  scale_y_continuous(labels = scales::percent_format()) +
  theme_classic() +
  labs(title = "Case 1",
       subtitle = "Fun w/ the class",
       x = "Session #",
       y = "Score",
       shape = "Session Name",
       linetype = "Session Name")
```


Case 1
Fun w/ the class
80%
60%
40%
20%
0%
Score
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Session #
Session Name
Baseline
Intervention