Introduction to Data Science
Exploratory Data Analysis

So far in this course, we've used visualization and data manipulation (via the `ggplot` and `dplyr` packages in the Tidyverse) to explore datasets and begin learning about them. Today, we'll continue that process through exploratory data analysis. Basically, exploratory data analysis is one of the first steps to take whenever you get a dataset: what's going on with my variables? What are the relationships between them? Before we're ready to do more complex statistical analyses of our data—the kinds of analyses we'll discuss in future classes in this sequence—we need to begin with some exploratory analyses.

We've already seen how some basic exploratory analyses can be quite helpful. Think back to our lecture and handout on data visualization with `ggplot`, where we analyzed the relationship between engine size and fuel efficiency. There, we found that bigger engines use more fuel, which isn't terribly surprising. But what was more surprising (at least to me!) was that this relationship varied a great deal by the type of car: the relationship for sub-compact cars looked very different than the relationship for SUVs, for example. This is what we mean by exploratory analysis: digging into our data to find out what patterns are there, so that we can make smart decisions about how to model our data.

Today, we'll start by using another dataset built into R: a dataset on diamonds. This dataset contains information on more than 50,000 diamonds: their price, the "4Cs" (carat, cut, color, and clarity) that determine price, and their length (x), width (y), and depth (z). We'll use these different attributes to try and understand what best predicts a diamond's price.

We'll use this dataset to explore what factors best seem to predict a diamond's price. Let's begin by looking at the distribution of the price of each diamond.

As always, we'll begin by loading the library which contains the data:

```
library(tidyverse)
```

By now, you should know what to do if you see the error that there is no library named tidyverse (make sure you're installed the package on your machine using the `install.packages()` command).

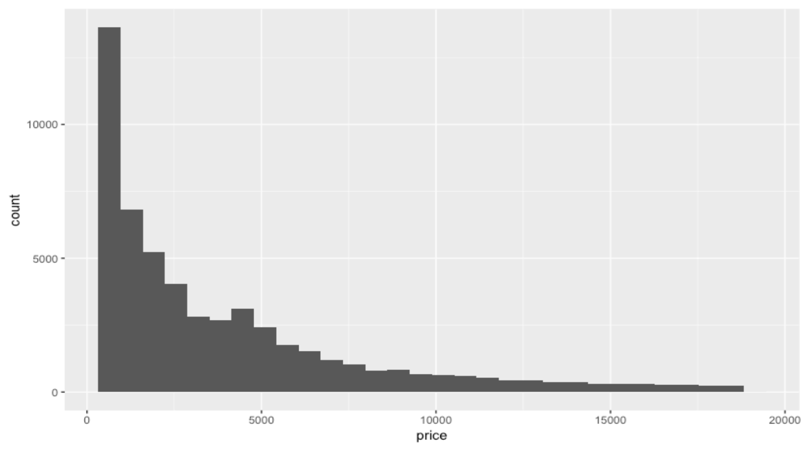
One of the first things I do when I get a new dataset is to use the summary command to just let me see some descriptive statistics on the data:

```
> summary(diamonds)
##      carat               cut        color        clarity
##  Min.   :0.2000   Fair     : 1610   D: 6775   SI1    :13065
##  1st Qu.:0.4000   Good     : 4906   E: 9797   VS2    :12258
##  Median :0.7000   Very Good:12082   F: 9542   SI2    : 9194
##  Mean   :0.7979   Premium  :13791   G:11292   VS1    : 8171
##  3rd Qu.:1.0400   Ideal    :21551   H: 8304   VVS2   : 5066
##  Max.   :5.0100                     I: 5422   VVS1   : 3655
##                                     J: 2808   (Other): 2531
##      depth           table           price             x
##  Min.   :43.00   Min.   :43.00   Min.   :  326   Min.   : 0.000
##  1st Qu.:61.00   1st Qu.:56.00   1st Qu.:  950   1st Qu.: 4.710
##  Median :61.80   Median :57.00   Median : 2401   Median : 5.700
##  Mean   :61.75   Mean   :57.46   Mean   : 3933   Mean   : 5.731
##  3rd Qu.:62.50   3rd Qu.:59.00   3rd Qu.: 5324   3rd Qu.: 6.540
##  Max.   :79.00   Max.   :95.00   Max.   :18823   Max.   :10.740
##
##       y                z
##  Min.   : 0.000   Min.   : 0.000
##  1st Qu.: 4.720   1st Qu.: 2.910
##  Median : 5.710   Median : 3.530
##  Mean   : 5.735   Mean   : 3.539
##  3rd Qu.: 6.540   3rd Qu.: 4.040
##  Max.   :58.900   Max.   :31.800
##
```

This is actually quite a lot of information. For my continuous variables (those that take many different values, like price or x/y/z, which record the dimensions of the diamond), I see the mean, median, inter-quartile range (1st/3rd quartiles), and the range (minimum/maximum). For discrete variables (those that take on a limited set of values), I get a table of values. Note that price is highly skewed, ranging from $326 to $18,823!

Let's see the distribution of price visually by creating a histogram:

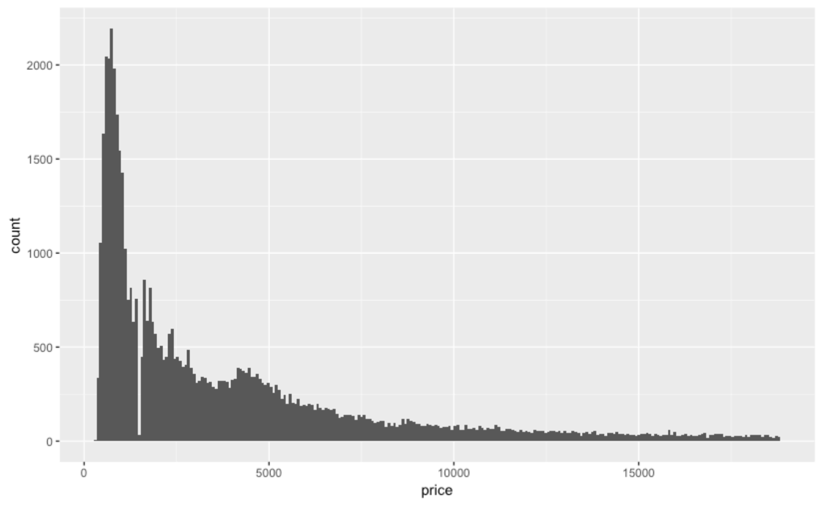
```
ggplot(data=diamonds)+
  geom_histogram(aes(x=price))
```



You should be familiar with this syntax and output from our discussion of visualization. If not, please refer back to that lecture.

Note that this produces an warning message in R: `` `stat_bin()` using `bins = 30`. Pick better value with `binwidth`. `` R is telling us that we need to think more carefully about the number of "bins" the histogram is creating. When R creates a histogram, it does so by "binning" the x variable (here, price). By default, R uses 30 bins: so it divides price into 30 equally-sized bins, and then the graph tells us how many observations are in that bin: that's what the histogram displays. Here, since we have a lot of data—more than 50,000 observations—we want more bins. How many? There's no magic answer, it takes a bit of trial-and-error to see what looks best. Let's try 250 bins: that is, let's re-run the histogram with 250 bins, instead of 30, so we'll see much more fine-grained resolution in the data. There's nothing magic about this figure, I just played around with it until I found one that works.

```
ggplot(data=diamonds)+
  geom_histogram(aes(x=price), bins=250)
```

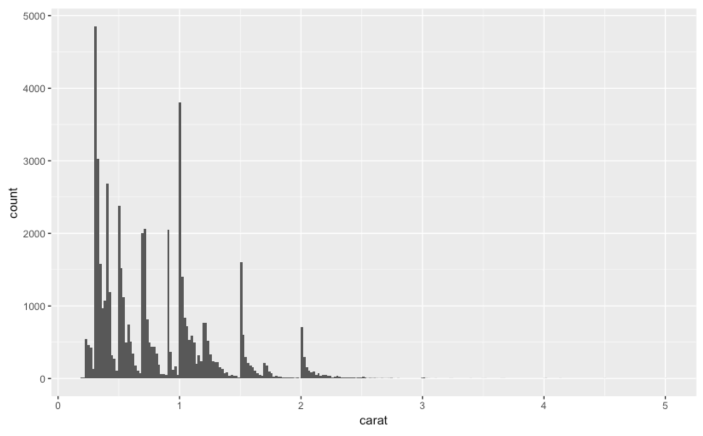


So notice that here, we get quite a bit more resolution, and can see things more clearly (you can play around with larger numbers of bins and see if you think that improves the graph on your own). We can notice a few things right away. First, there's a peak: the modal diamond sells for around $600 (so pretty reasonable as diamonds go). Remember that the mode is just the most common value, so this is just telling us that the most common price of a diamond in our dataset is $600.

But there's a long right tail of very expensive diamonds, which indicates quite a lot of skew. We also see by looking at our summary table from earlier. There, we see a mean (average) value of $3932, but a median value of $2401. Remember that the mean is the average of all values, and the median is the value at the 50th percentile: if you lined up all of the observations from biggest to smallest, the median would be the one right in the middle. When we see a much larger mean than our median value (as we do here), that's a sign that we have a small number of very large values which are inflating our mean. Third, notice that there's a gap in the distribution of price! No diamonds sell for $1500! I don't know why this is the case, but it could be interesting to explore further. Note that this wasn't apparent from the original histogram, so it pays to do some digging into your data.

Now let's start to explore how the various factors of a diamond predict its price. Let's start with the most obvious indicator: the carat (size) of the diamond. A carat by the way is a unit of weight for gemstones & pearls, and is equal to 200 milligrams. Let's begin by plotting the distribution of carat, just as we did for price:

```
ggplot(data=diamonds)+
  geom_histogram(aes(x=carat), bins=250)
```



So here, we see that there is (once again) a very long tail. If you zoom into the graph (click on the button labeled "Zoom" in the plots window in RStudio), you can see that there are a tiny number of diamonds over 3 carats, which is pretty enormous. For example, according to *US Weekly* (a trusted source!), Carrie Underwood and Jennifer Garner got 4-5 carat diamond engagement rings, so those are more celebrity-sized and outside the range of most ordinary people.

But perhaps the more interesting pattern is the spiking that's going on in the graph. We see a huge spike at around 0.3 carats, which probably represent the relatively inexpensive diamonds we saw in our earlier price graph. But note the large spikes at 0.75, 1, 1.5, and 2 carats. It seems that people really like to buy in those nice round numbers. Let's look a bit more carefully at the data surrounding the 1-carat spike:

```
> diamonds %>%
+   filter(carat >= 0.89, carat <= 1.1) %>%
+   count(carat) %>%
+   print(n = 22)
# A tibble: 22 x 2
   carat     n
   <dbl> <int>
 1  0.89    21
 2  0.9   1485
 3  0.91   570
 4  0.92   226
 5  0.93   142
 6  0.94    59
 7  0.95    65
 8  0.96   103
 9  0.97    59
10  0.98    31
11  0.99    23
12  1     1558
13  1.01  2242
14  1.02   883
15  1.03   523
16  1.04   475
17  1.05   361
18  1.06   373
19  1.07   342
20  1.08   246
21  1.09   287
22  1.1    278
```

Note that the `print()` is ensuring that we have all 22 entries printed (rather than just the first 10, which is the default) . So note that there are almost no diamonds at 0.89 or 0.99 carats, but there are a ton of diamonds at 0.9 carats and 1 or 1.01 carats. People seem to like to buy a nice round number of carats (probably because it sounds better to say you bought a 1-carat diamond that a 0.99-carat diamond).

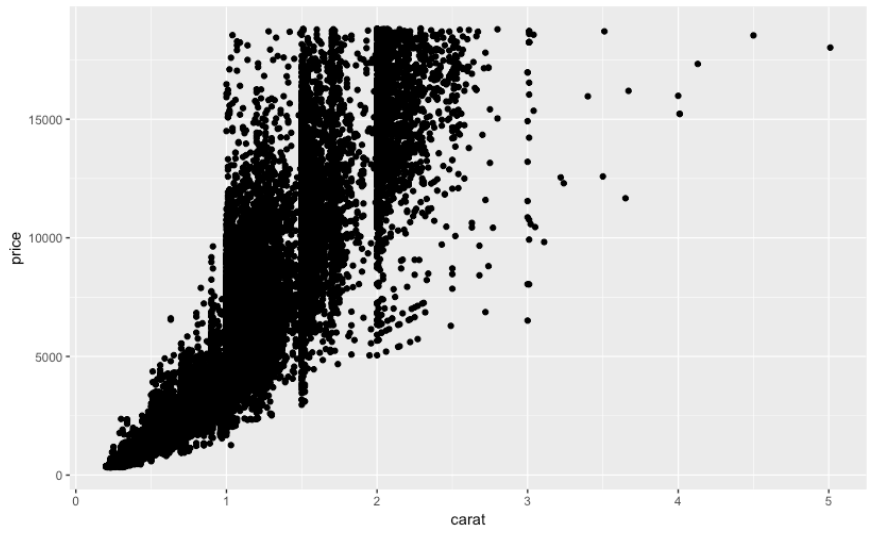
TEST YOURSELF: How would you construct this table around the spike at 1.5 carats?

ANSWER:
```
diamonds %>%
  filter(carat >= 1.44, carat <= 1.55) %>%
  count(carat) %>%
  print(n = 30)
```

I chose 1.44 as an arbitrary cut-off, you just want to show that there aren't many diamonds slightly below 1.5 carats, with a big spike at that value.

So far, we've explored one variable at a time. Let's now start to unpack the relationship between price and carat. Statisticians and data scientists talk about this as covariance or covariation: how do the two variables change together? This is what really makes statistics and data science valuable, when you can explore the relationships between variables. We can do that with a scatterplot, which we've previously used to explore the relationship between engine size and fuel efficiency in our earlier lecture on data visualization. Here, we would tell R to produce this graph with the following command:

```
ggplot(diamonds) +
  geom_point(mapping=aes(x=carat,y=price))
```



So we see that there's definitely a strong relationship there: as the diamond gets larger (the number of carats increases), the price goes up. But it's also worth noting that the relationship isn't linear, as the price increases quite rapidly once the diamond is larger than about 1 carat. Indeed, note that while the largest diamonds are among the most expensive diamonds, there are a rash of diamonds over 1-1.5 carats that are very expensive. So this suggests that we need to consider some of the other variables as well, as not all 1-carat diamonds are created equally!

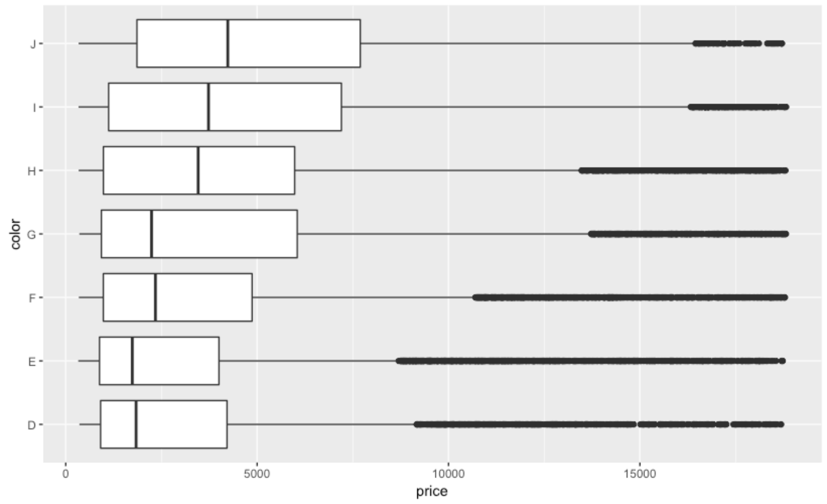
Let's begin by exploring the relationship between color and price. Diamonds are rated on a color scale from colorless (grade D) to yellow (grade Z). For more on this scale, and for examples, see: https://www.lumeradiamonds.com/diamond-education/diamond-color. In general, people typically prefer more colorless diamonds, and so we'd expect diamonds with less color (which look more "pure") to sell for more money.

In our dataset, we have diamonds from grade D (colorless) to grade J (near colorless). Interestingly, these variations are typically not discernable to the naked eye, but may well matter in the price.

TEST YOURSELF: We want to plot the relationship between a discrete variable (color) and a continuous variable (price). How would we examine this relationship?

Answer: We can use a boxplot to examine this relationship:

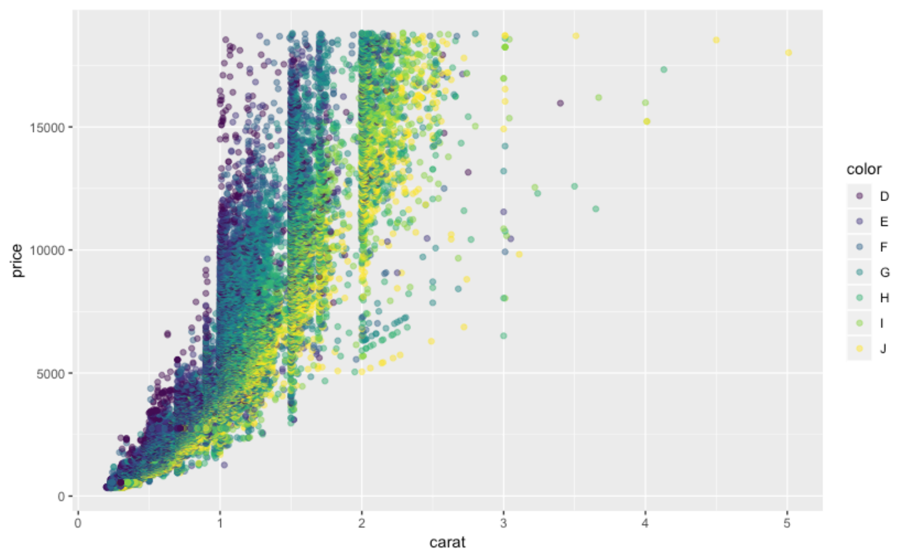
```
ggplot(diamonds) +
  geom_boxplot(mapping=aes(x=color,y=price)) +
  coord_flip()
```



Remember from our visualization lecture how we read a boxplot: the line inside the box is the median, the ends of the box are the inter-quartile range, and the individual points are outliers. But this seems to show us a really odd pattern: the median sale price for a colored diamonds (grade J) is higher than for a colorless diamonds (grade D). That's puzzling!

Let's use one of the tricks we saw in ggplot in our earlier lecture: we'll use a color overlay to add another variable to our scatterplot. That is, we'll re-do our price/carat scatterplot, but we'll show the different grades of diamonds in different colors, setting the color as one of our aesthetics:

```
ggplot(data = diamonds) +
  geom_point(aes(x = carat, y = price, color=color, alpha = 0.5))
```



So note that I've told `ggplot` to make each color of diamond a different color on our graph by setting the "color" aesthetic. This syntax may seem a little confusing since we have `color = color`, but to clarify, the first color is the aesthetic that controls the color of the graph. The second color is the variable name color, so the color on the graph is corresponding with the color of the diamonds. If you wanted to color the graph based off the cut of the diamond, the syntax would be `color = cut.` I've also set the alpha option to increase the transparency of the points: with all of the clustering in the graph, it gets hard to see with more opacity in the points.

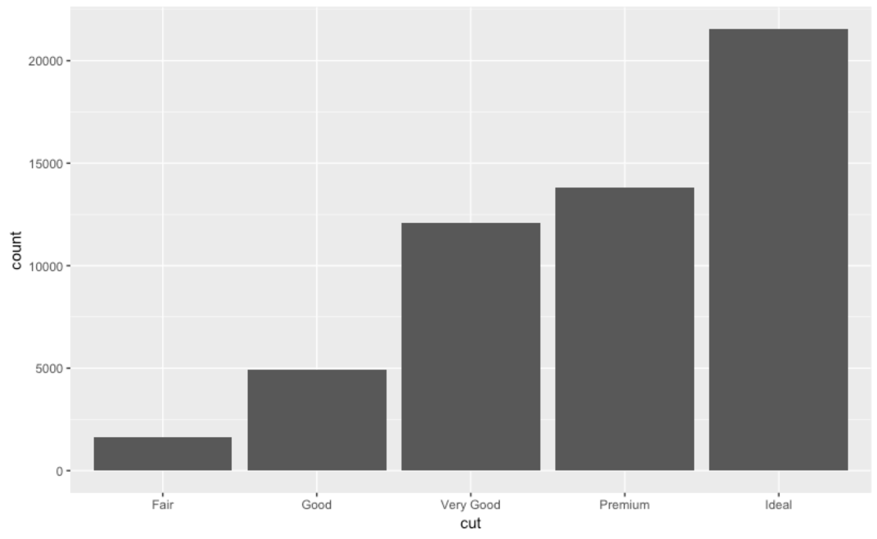
So now we see why our boxplot looks the way it does! Look at the yellow points corresponding to the "J" colored diamonds (those with the most color, which should reduce their price). At a given carat value, the less color in the diamond, the more it sells for: notice that, for a given carat value, the D diamonds have the highest price, and (generally speaking), the I/J diamonds sell for less. But also note that there are many more smaller D diamonds, and larger J diamonds. Indeed, you can imagine that a colorless large diamond might sell for more, but there just aren't many in our dataset.

Let's do a similar analysis with cut, which is the shape of the diamond. The cut determines how reflective the diamond is, so that it does a better job of reflecting the light that hits it. Cut is graded on a scale from Fair to Ideal (though online I see different gradings being used, but that doesn't matter).

We can begin by looking back at the tabulated data for cut, seen above in our data summary. Notice that there are many, many more "Ideal" cut diamonds than fair diamonds. Indeed, contra to what I'd expect ex ante, as the cut quality increases, so does the number of diamonds in the dataset.

If you don't like a table, remember that you can always look at a bar chart if you find that easier:

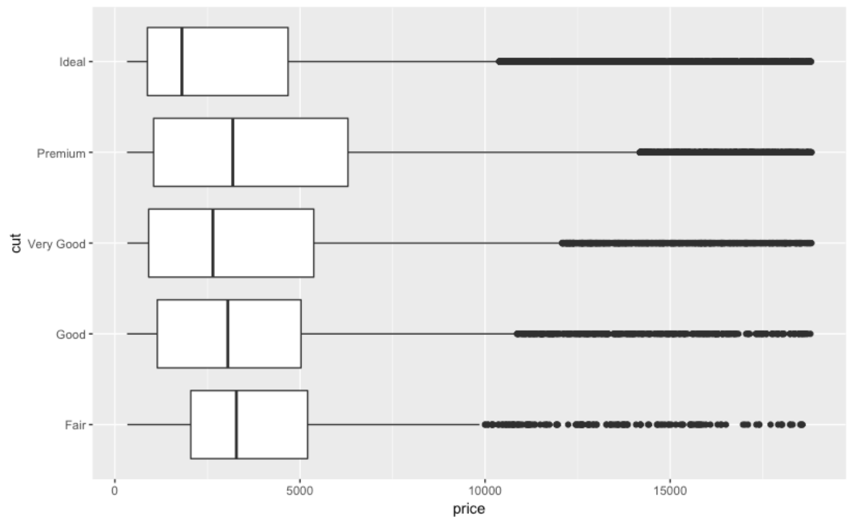
```
ggplot(data=diamonds) +
  geom_bar(aes(x=cut))
```



And note that it gives us the same information. If I want to know the exact numbers, I look at a table. If I just want to get a sense of the scale, I often use a bar chart. But again, this is a personal preference, and you should use the one that seems most natural to you.

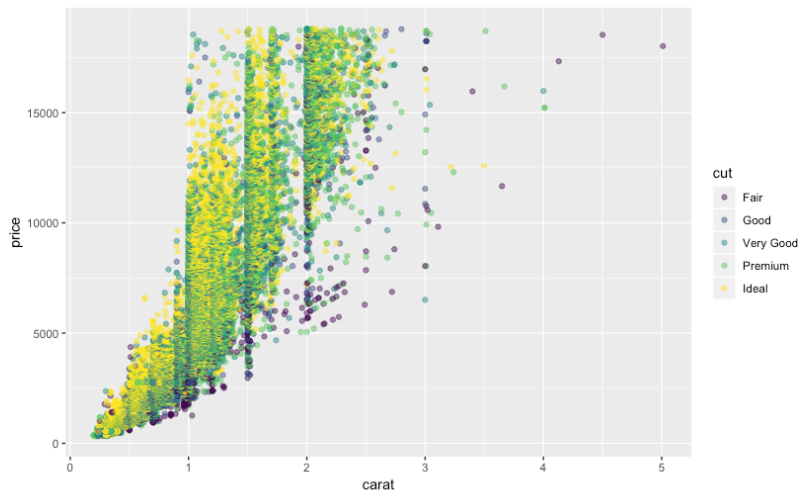
Again, let's start with a boxplot of price, grouped by the different cuts:

```
ggplot(diamonds) +
  geom_boxplot(mapping=aes(x=cut,y=price)) +
  coord_flip()
```



Once again, we see very little variation in price based on cut, which is odd. Let's try the same trick as above: let's look at the price/carat scatterplot, but separating out the different cuts with different colors on the graph:

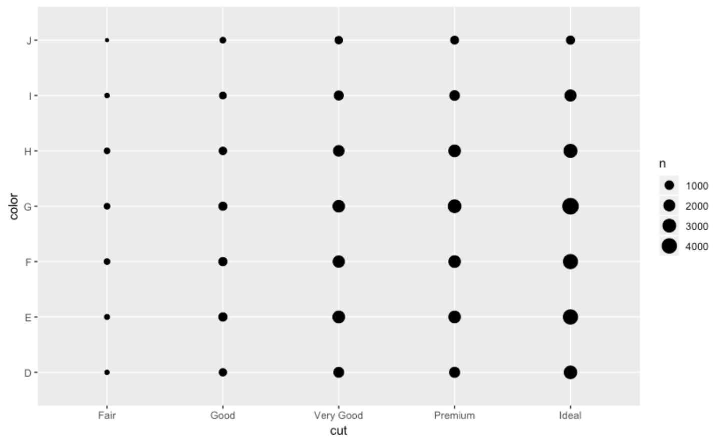
```
ggplot(data = diamonds) +
  geom_point(aes(x = carat, y = price, color=cut), alpha=0.5)
```



Here, we see a relationship, though frankly, it's a bit weaker than our relationship above for the diamond's color. While it's true that our ideal diamonds tend to sell for more, you should also note that premium and very good diamonds are all over the graph. So here, I'd say that cut has a relationship to price (even accounting for carat), but it's weaker than the relationship with color.

One additional thing we can do is look at the relationship between cut and color, to see if there's a relationship there. When we have 2 categorical variables like this, we can display their relationship using the `geom_count()` function built into ggplot:

```
ggplot(data = diamonds) +
  geom_count(mapping=aes(x=cut,y=color))
```



Here, the size of the circle gives the relative fraction of the cases in each combination. So here, we see that the largest number of diamonds in our data are ideal cut diamonds with coloration "G" (right in the middle of the color range).

When I look at this, I don't see much of a pattern. The only thing I notice is that the points get larger as the cut improves, but that's just a function of their being more premium/ideal cut diamonds in our dataset. There doesn't seem to be any obvious relationship between these two variables.

If you want to see a slightly different way of examining this relationship, you can look at the textbook, section 7.5, which covers using the `geom_tile()` function.

In the homework below, you'll have a chance to examine the effect of clarity (the last of the "4 Cs") and see how it is related to price. You might wonder if we could combine price, carat, cut, and color (and clarity for that matter) into one exploratory analysis. That's much harder to do, and takes more sophisticated methods that are beyond the scope of this class.

But note here that we've learned quite a bit about our data! It looks like carat is the most important predictor of a diamond's price, but that color, and to a lesser extent, cut, also matter (and again, you'll explore clarity below in the homework). So even before running any models, we've learned quite a bit about our data just using our simple exploratory tools.

Let's now turn to our friendly nycflights13 data to show a few more interesting types of data exploration that we can do. One interesting set of questions is when we see delays. We can think about this several different ways: what times of the year are most likely to see delays? What destinations are most likely to be delayed? What originating airport is most likely to see delays (remember we have flights from Newark, La Guardia, and JFK in this dataset)?

Let's start by thinking about the seasonality trend. To do this, we can calculate the average delay by day. We know how to attack this problem in several steps, based on what we've learned over the past few lectures: We'll take the data and group it by month & day, and then calculate the average delay by day. Let's also see the 10 days with the average worst delay:

```
> flights %>%
+   group_by(month,day) %>%
+   summarise(avg_delay = mean(arr_delay, na.rm=T)) %>%
+   ungroup() %>%
+   arrange(-avg_delay) %>%
+ head(10)
# A tibble: 10 x 3
   month   day avg_delay
   <int> <int>     <dbl>
 1     3     8      85.9
 2     6    13      63.8
 3     7    22      62.8
 4     5    23      62.0
 5     7    10      59.6
 6     9    12      58.9
 7     7     1      58.3
 8    12    17      55.9
 9     8     8      55.5
10    12     5      51.7
```

Note a few things about this. We've see all of this type of code before, except for `head(x)`, which just gives me the first x rows of the result. I calculated the average arrival delay, since that's typically the most consequential sort of delay (recall from our previous exercise that we saw that planes can and do make up for departure delays in the air, especially on longer flights). But recall that delays also include negative delays, when a flight arrives early. It's worth exploring what happens conditional on not arriving early (i.e., an arrival delay of 0 or more minutes). To do that, we'd use the filter command to select non-negative arrival delays:

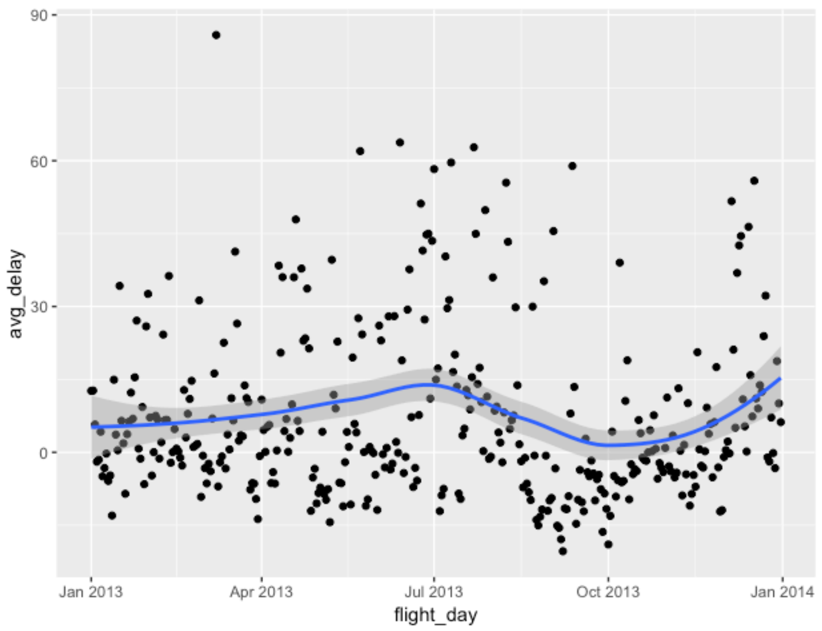
```
> flights %>%
+   filter(arr_delay >= 0) %>%
+   group_by(month,day) %>%
+   summarise(avg_delay = mean(arr_delay, na.rm=T)) %>%
+   ungroup() %>%
+   arrange(-avg_delay) %>%
+   head(10)
# A tibble: 10 x 3
   month   day avg_delay
   <int> <int>     <dbl>
 1     7    10     108.
 2     9     2     100.
 3     7    22     100.
 4     9    12      98.4
 5     3     8      96.9
 6     4    10      91.9
 7     6    24      90.1
 8     5    23      87.6
 9     6    27      86.3
10     7    28      85.4
```

So that changes the average delay quite a bit! Now we have 3 days where the average arrival delay is 100 minutes or more. Which one is correct? There's no clear answer, it depends on the question you want to ask. I tend to think early arrivals matter, so we'll not condition in the analysis below, but it's always good to explore these sorts of effects.

So now that we know how to calculate the average delay, let's plot it over the course of the year to look for patterns. To do that, I need to create a variable that records the day+month as one variable; to do that I need to load the `lubridate()` library, which we'll cover in an upcoming lecture (and if you've never used it, you'll need to install it first with the install.packages() command):

```
library(lubridate)
flights %>%
  mutate(flight_day = make_datetime(year,month,day)) %>%
  group_by(flight_day) %>%
  summarise(avg_delay = mean(arr_delay, na.rm=T)) %>%
  ggplot(., mapping=aes(x=flight_day, y=avg_delay)) +
  geom_point() +
  geom_smooth()
```

You should be familiar with all of this code and what's happening, except for the make_datetime() function. Again, we'll cover this in a later lecture, but basically it creates a variable that R recognizes as a date variable, which is useful for making over-time plots. This code chunk produces the following output:

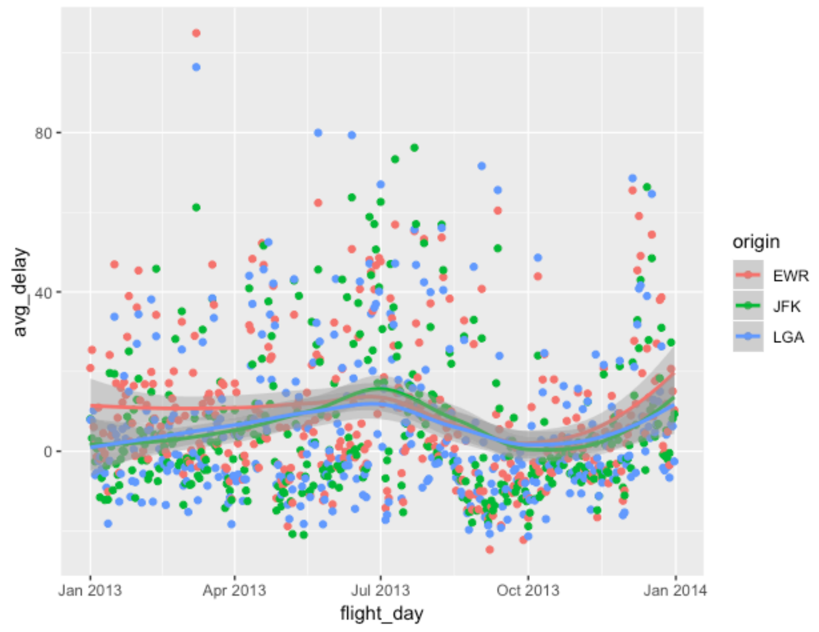


So we can see there are some outlier days, but on average, delays are higher in the summer months (and are lower around October).

TEST YOURSELF: How would you look to examine whether the patterns we found here are different by originating airport?

ANSWER: You'd include originating airport in your call to `group_by()`:

```
flights %>%
  mutate(flight_day = make_datetime(year,month,day)) %>%
  group_by(flight_day,origin) %>%
  summarise(avg_delay = mean(arr_delay, na.rm=T)) %>%
  ggplot(., mapping=aes(x=flight_day, y=avg_delay, color=origin)) +
  geom_point() +
  geom_smooth()
```

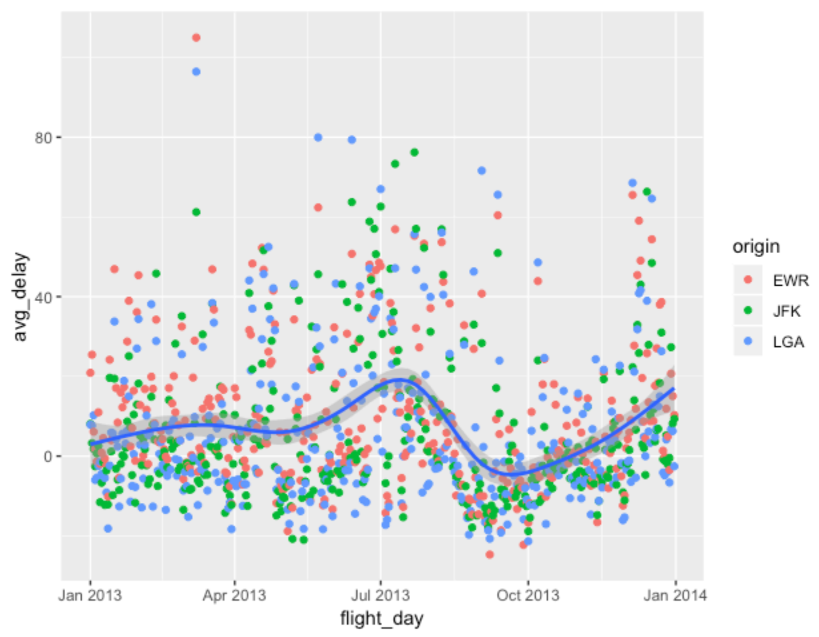


So fascinatingly, the pattern is more or less the same across the year from all three airports. While it seems like La Guardia definitely has some of the worst days for delays (see the points in blue near the top of the plot), on average, it's no worse than the other two. Interestingly, delays are slightly worse at Newark early and late in the year, but worst at JFK in the summer.

TEST YOURSELF: In the example above, we had three different lines from geom_smooth(), one for each airport. How would I edit my R code so that we had one trend line for all 3 airports but still showed the points separated by airport?

ANSWER: You'd put the color option inside `geom_point()` instead of in `ggplot()`. Remember that any aesthetics set in the call to `ggplot()` then determine the aesthetics for all geoms. If I want to just set something for a particular geom, I need to edit that one. Here that looks like this:

```
flights %>%
  mutate(flight_day = make_datetime(year,month,day)) %>%
  group_by(flight_day,origin) %>%
  summarise(avg_delay = mean(arr_delay, na.rm=T)) %>%
  ggplot(., mapping=aes(x=flight_day, y=avg_delay)) +
  geom_point(mapping=aes(color=origin)) +
  geom_smooth()
```



TEST YOURSELF: In the previous two examples, why are the trend lines different?

ANSWER: Now we have a different set of points! We have delays by day & airport, rather than just by day. With a different set of data, you get a slightly different trend line.

Before we leave this section, we can work a bit to improve this presentation on our graph. The information is good, but the labeling/formatting is a bit sloppy in at least 5 ways:

1. "flight_day" and "avg_delay" are good variable names, but not good axis labels
2. The axis tick marks and their labels are odd: in a graph of departures in 2013 I don't want a tick mark labeled "January 2014". Likewise, it would be more natural to have tick marks every 30 minutes on the y-axis, not every 40 minutes.
3. There's no title for the plot, so someone who didn't know this data wouldn't know what this is showing
4. I don't love the gray background and would prefer it to be white
5. The legend is labeled "origin" instead of something more informative, and it uses the airport code rather than its name

These aren't a big deal if I'm just making the plot for myself, but if I'm sharing this with others—say in a publication or a presentation (or even a homework!)—I want a more professional-looking plot. Undoubtedly, you could come up with other ways to improve the plot, but this gives us a good sense of how to change some of the most common features of a plot.

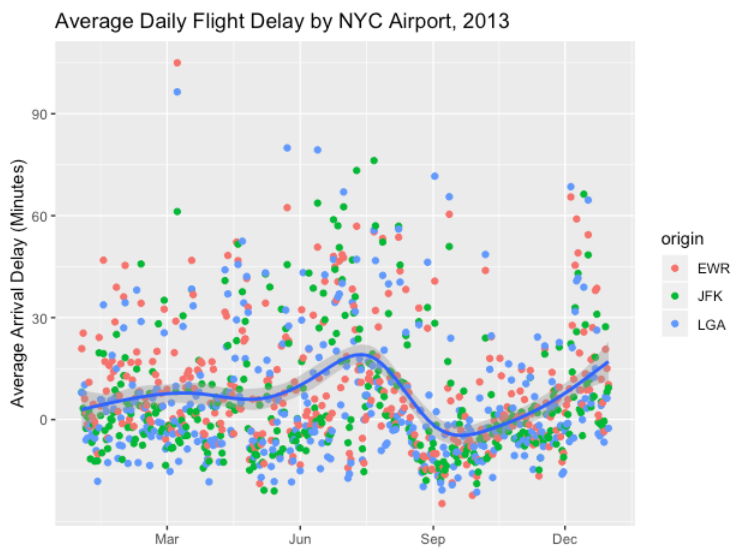
Let's walk through adjusting each of these issues, but know that ggplot() graphics are *extremely* customizable. You can adjust virtually any aspect of the graph, and this is just the tip of the proverbial iceberg. For each of the commands below, you should read the help files to see what is possible.

To begin, so that I don't have to keep repeating the first bit of the code that does the analysis, I'm going to save that to a new object called flight_graph (just to avoid having to repeat that code):

```
flight_graph <- flights %>%
                mutate(flight_day = make_datetime(year,month,day)) %>%
                group_by(flight_day,origin) %>%
                summarise(avg_delay = mean(arr_delay, na.rm=T))
```

Let's begin by fixing the axis labels and tick marks, and let's add the title. I've made that color **bold** in the example below:

```
## step 1: fix the axes & title
library(scales)
ggplot(flight_graph, mapping=aes(x=flight_day, y=avg_delay)) +
  geom_point(mapping=aes(color=origin)) +
  geom_smooth() +
  ylab("Average Arrival Delay (Minutes)") +
  xlab("") +
  scale_x_datetime(date_breaks="3 months",
                   labels=date_format("%b")) +
  scale_y_continuous(breaks=c(0,30,60,90,120)) +
  ggtitle("Average Daily Flight Delay by NYC Airport, 2013")
```



So the `ylab()` and `xlab()` set the labels for the y- and x-axes, respectively, and `ggtitle()` generates the plot title. Notice that given the way my date variable displayed, I didn't really need a label for the x-axis, so I just used " " (an empty space) to overwrite that label.

To change the way the tick marks and tick mark labels work for the x- and y-axes, I need to use the `scales` library, and then I need to tell ggplot what type of variable I think I'm plotting. The general syntax is `scale_variable_type()`, where "scale" tells R you're adjusting a scale, "variable" is either x or y (as appropriate), and "type" is the type of variable: discrete, continuous, or datetime. What's the difference? Discrete variables are variables that can only take on a limited set of values. For example, airport codes are discrete: there are lot of them, but there's a finite number. Continuous variables are things that can take on an infinite number of values. For example, the number of minutes delayed is continuous: I could be 30.24598864 minutes delayed (at least in theory, if I had an accurate enough time measurement). Date-time variables are exactly what they sound like: variables that record a date and time (like the departure date in our example here).

So here, to change the tick marks on the y-axis, I followed the syntax format: it's `scale_y_continuous` because it's a continuous variable on the y-axis, and I wanted the tick mark labels every 30 minutes (notice the label for 120 minutes gets cut off, since the longest average delay is 105 minutes (March 5th, from Newark Airport). Note that I used our c() syntax to tell R that I wanted tick-marks on the y-axis at multiple points (every 30 minutes)

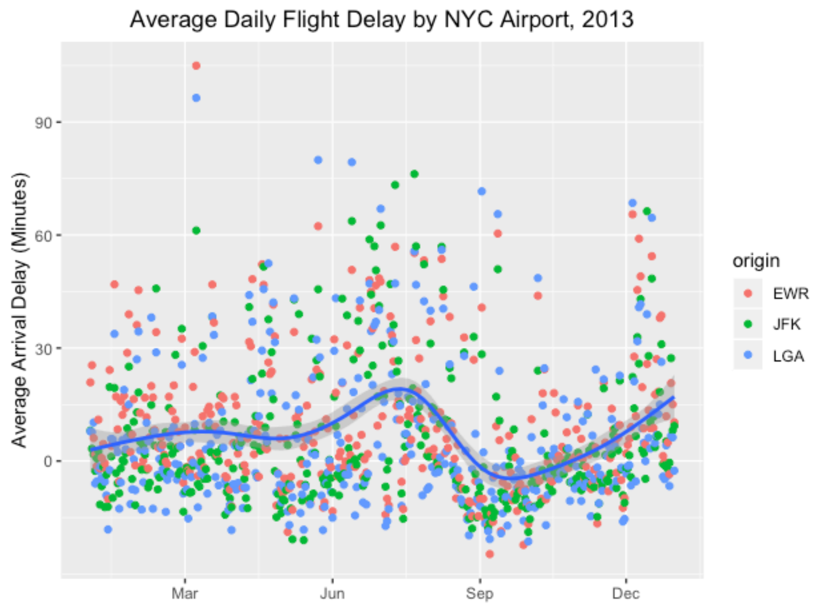
For the x-axis, I did the same pattern: it's a date-time variable on the x-axis, so it's `scale_x_datetime`. Here, I have to tell it how often I want the labels with the option `date_breaks`, and then I have to tell it how to format the dates with the `date_format` syntax. %b tells R that I want it to print the abbreviated month name. Some common date-time options are:

%A: Weekday names (Sunday)
%a: Abbreviated weekday names (Sun)
%e: Day of the week (1-31)
%b: Abbreviated month name (Jan)
%B: Month name (January)
%y: Year without century (18)
%Y: Year with century (2018)

The full list can be found at: https://www.foragoodstrftime.com and click on the reference tab.

Notice that by default, ggplot left-justifies titles, because that works better for multi-line titles (https://github.com/tidyverse/ggplot2/releases/tag/v2.2.0). Personally, I prefer centered titles. If you want your title to be centered, you need to add one more line of code, which is the last line of code here (which is bolded):

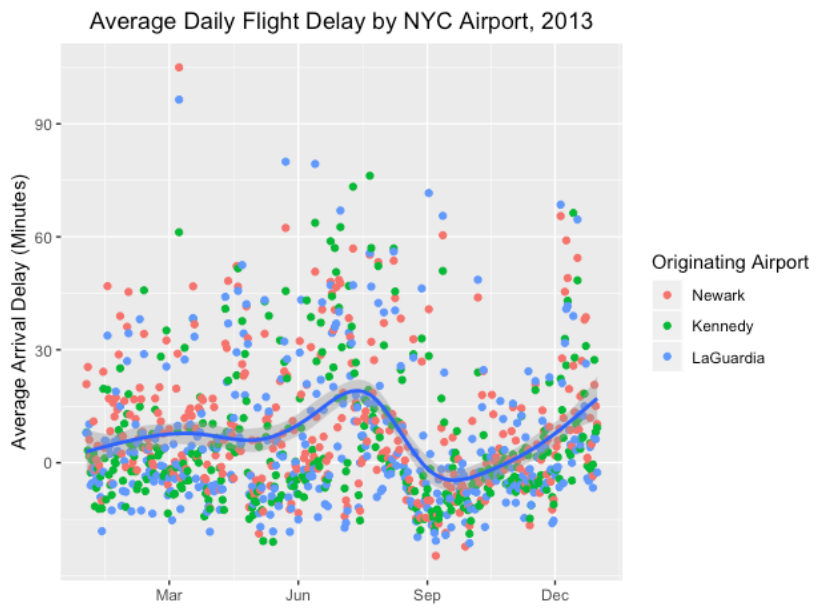
```
library(scales)
ggplot(flight_graph, mapping=aes(x=flight_day, y=avg_delay)) +
  geom_point(mapping=aes(color=origin)) +
  geom_smooth() +
  ylab("Average Arrival Delay (Minutes)") +
  xlab("") +
  scale_x_datetime(date_breaks="3 months",
                   labels=date_format("%b")) +
  scale_y_continuous(breaks=c(0,30,60,90,120)) +
  ggtitle("Average Daily Flight Delay by NYC Airport, 2013")+
  theme(plot.title = element_text(hjust = 0.5))
```



The `theme()` command is extremely powerful, and can control many other aspects of a plot. Here, I've used it to change the horizontal justification of the title to 0.5, or centered, but there are several dozen aspects of the plot that you can set. To see all of your options, look up the help file.

That gives us some decent axis labels and a title, but it doesn't solve the issue with the legend. To do that, let's add some more code (again, the new code is bolded):

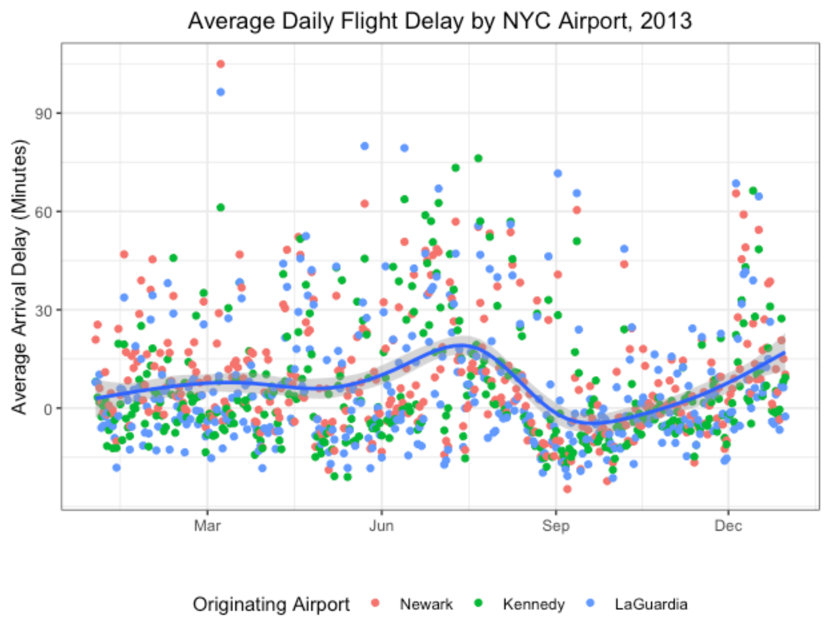
```
ggplot(flight_graph, mapping=aes(x=flight_day, y=avg_delay)) +
  geom_point(mapping=aes(color=origin)) +
  geom_smooth() +
  ylab("Average Arrival Delay (Minutes)") +
  xlab("") +
  scale_x_datetime(date_breaks="3 months",
                   labels=date_format("%b")) +
  scale_y_continuous(breaks=c(0,30,60,90,120)) +
  ggtitle("Average Daily Flight Delay by NYC Airport, 2013")+
  theme(plot.title = element_text(hjust = 0.5)) +
  scale_color_discrete(name="Originating Airport",
                       labels=c("Newark","Kennedy","LaGuardia"))
```



Note that the syntax for changing the legend is exactly the same as for changing the axis tick mark labels. The work here is being done by the `scale_color_discrete()` command. Here, it's color because I've used colors to denote the different airports. The general form is `scale_aesthetic_vartype()`, where scale is a codeword to ggplot that you're changing some aspect of the legend content. Here, I'm changing the color aesthetic, and I have 3 values (Newark, JFK, LaGuardia), so it's a discrete variable, hence the syntax above. But if I had used different shapes, rather than different colors, to denote the different airports, the command would be `scale_shape_discrete()`.

But I don't love that the plot is squashed by that large legend, so let's move the legend so that it's at the bottom of the plot, rather than on the right. Also, I don't love gray background, so let's remove that as well. I can do both of these with just a bit more code (again, bolded):

```
## step 2: fix the background & legend placement
ggplot(flight_graph, mapping=aes(x=flight_day, y=avg_delay)) +
  geom_point(mapping=aes(color=origin)) +
  geom_smooth() +
  ylab("Average Arrival Delay (Minutes)") +
  xlab("") +
  scale_x_datetime(date_breaks="3 months",
                   labels=date_format("%b")) +
  scale_y_continuous(breaks=c(0,30,60,90,120)) +
  ggtitle("Average Daily Flight Delay by NYC Airport, 2013") +
  theme_bw() +
  theme(plot.title = element_text(hjust = 0.5),
        legend.position = "bottom") +
  scale_color_discrete(name="Originating Airport",
                       labels=c("Newark","Kennedy","LaGuardia"))
```



So here, the `theme_bw()` command removes the gray background, and my edited call to `theme()` adjusts the position of the legend.

This is really just a small taste of how much you can customize a graph. You could also change the colors used by ggplot to for the points: instead of orange, green, and blue, you could pick other colors if you prefer them. You can easily adjust almost any aspect of the plot, and the world is your oyster. Chapter 28 of the textbook covers many more ways you can change a plot. The key is to think about how to get the plot to most effectively convey your information.

I wouldn't worry too much about learning all of these options. Rather, when I have a question, or want to change something, I usually consult the ggplot help (online at: https://ggplot2.tidyverse.org/index.html) or search StackOverflow. Someone else has undoubtedly wanted to change the same thing as you!

So now we've seen some analyses one could do with this data, and gotten some good ideas for future analyses. For example, we could now look for weather data (from another source) and use that to analyze whether weather patterns help predict this pattern of delays. We could also look at different times of the day to see if there are more/less delays at certain times of the day (like we did above for flight cancellations). We could also get data on days of the week, and see if delays are more or less likely on particular days of the week. Really, the sky is the limit here (no airline pun intended!).